POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania
Inżynieria Oprogramowania i Baz Danych

Hubert Chylik
Nr albumu 15737

Dekompozycja złożonego systemu informatycznego na mikrousługi na przykładzie systemu klasy CRM

Praca magisterska napisana
pod kierunkiem:

dr inż. Mariusza Trzaski

Warszawa, czerwiec 2019

Streszczenie

Praca jest opisem procesu dekompozycji monolitycznego, złożonego systemu klasy CRM na mikrousługi. Zestawia różnice pomiędzy architekturą monolityczną oraz architekturą mikrousług w kontekście rozwiązań korporacyjnych. Zawiera opis implementacji prototypów zgodnych z obiema architekturami, które realizują typowe funkcjonalności biznesowe zapewnianie klientom przez większość dużych organizacji.

Prototyp został wykonany głównie w języku Java (framework Spring) oraz Javascript, przy wsparciu relacyjnych i nierelacyjnych silników bazodanowych oraz innych technologii okalających. Jest niejako narzędziem do potwierdzenia / zaprzeczenia założeń przyjętych na początku pracy.

Na jego podstawie zostały wysnute wnioski zawarte w podsumowaniu.

Podziękowania

Pracę pragnę zadedykować moim wspaniałym rodzicom Agnieszce i Piotrowi Chylik, którzy stale mnie mobilizowali i wspierali przez okres trwania studiów na Polsko-Japońskiej Akademii Technik Komputerowych.

Za wyrozumiałość, zaangażowanie oraz pomoc przy realizacji tej pracy pragnę złożyć serdecznie podziękowania mojemu promotorowi dr Mariuszowi Trzasce.

Spis treści

1. WSTĘP **5**
1.1. Cel pracy 5
1.2. Rozwiązania przyjęte w pracy 5
1.3. Rezultaty pracy 6
1.4. Organizacja pracy 6
2. SYSTEMY KLASY CRM **7**
3. ANALIZA ISTNIEJĄCYCH ARCHITEKTUR **11**
3.1. Architektura monolityczna 11
3.2. Architektura mikrousług 13
3.3. Porównanie 15
4. PROPOZYCJA ROZWIĄZANIA **16**
4.1. Wymagania funkcjonalne 17
4.2. Wymagania jakościowe 17
5. WYKORZYSTANE NARZĘDZIA I TECHNOLOGIE **18**
5.1. IntelliJ Idea 18
5.2. Javascript 19
5.3. NodeJS 19
5.4. Java 20
5.4.1 Spring Framework *20*
5.4.2 JSP (JavaServer Pages) *20*
5.4.3 Swagger (Swagger UI) *21*
5.5. HTML 22
5.6. CSS 22
5.7. Bootstrap 22
5.8. MVC 23
5.9. REST 24
5.10. HTTP 25
5.11. MongoDB 25
5.12. H2 Database 26
5.13. JMS (ActiveMQ) 26
5.14. Nginx 26
5.15. JMeter 27
6. PROTOTYP **28**
6.1. Serwer front-end 28
6.2. Rozwiązanie monolityczne 34
6.2.1 Warstwa trwałości *36*
6.2.2 Warstwa wizualna *37*
6.3. Rozwiązanie oparte o mikrousługi 41
6.3.1 Warstwa trwałości *42*
6.3.2 Notification – dodatkowy komponent w ramach mikrousługi w innej technologii *44*
6.3.3 Warstwa wizualna *45*
6.3.4 Równoważenie obciążenia *46*
7. MOŻLIWOŚCI ROZWOJU **48**
8. PODSUMOWANIE **49**
PRACE CYTOWANE **52**

1. Wstęp

Od lat, zarówno złożone systemy korporacyjne klasy CRM jak i mniejsze aplikacje są budowane z wykorzystaniem tradycyjnego podejścia architektonicznego, czyli monolitu.

Podejście to dominowało od lat 90 do czasów współczesnych, czyli do momentu, w którym ta architektura przestała być wystarczająca. W dobie rozwoju Internetu i technologii konieczne jest zapewnienia wysokiej dostępności systemu, na potrzeby setek tysięcy jednoczesnych żądań. Użytkownik końcowy oraz biznes zakładają, że wszystkie czynności możliwe do wykonania w aplikacji zostaną zrealizowane natychmiast i bez opóźnień, a system będzie cały czas działał stabilnie. Dodatkowo, architektura systemu musi być na tyle elastyczna, aby dało się szybko wdrażać kolejne funkcjonalności biznesowe bez wpływu na istniejące.

Architektura mikrousług ma być rozwiązaniem problemów monolitu i receptą na budowę wysokodostępnego rozwiązania.

1.1. Cel pracy

Praca ma na celu ocenę słuszności dekompozycji monolitycznych systemów klasy CRM na mikrousługi.

Celami pobocznymi pracy są:

- Wytworzenie prototypu aplikacji typu CRM w architekturze monolitu;
- Wykonanie prototypu zdekomponowanej aplikacji w architekturze mikrousług;
- Porównanie architektury monolitycznej do architektury mikrousług, zbadanie ich wad, zalet oraz ocenę obu podejść w kontekście wydajności, skalowalności oraz kosztu wdrożeń;
- Przeprowadzenie testów wydajnościowych;
- Przedstawienie możliwości rozwoju.

1.2. Rozwiązania przyjęte w pracy

Implementacje prototypów zostały wykonane z użyciem języka Java oraz wiodącego frameworku Spring. W celu zapewnienia integracji została użyta architektura REST oraz system kolejek ActiveMQ. Dodatkowo, jeden z modułów został wykonany w technologii Javascript, a warstwa prezentacji została zrealizowana przy użyciu JSP oraz Bootstrap. Za warstwę trwałości odpowiadają silniki baz danych: H2 oraz MongoDB.

1.3. Rezultaty pracy

Rezultatem pracy są:

- Opis procesu dekompozycji systemu i zbudowane na jego podstawie prototypy;
- Wynik porównania obu architektur oraz zgodnych z nimi implementacji;
- Ocena obu podejść oraz ocena słuszności migracji istniejących systemów monolitycznych do architektury mikrousług.

1.4. Organizacja pracy

W pierwszych trzech rozdziałach przedstawiono opis systemów klasy CRM oraz porównanie obecnie najpopularniejszego stylu architektonicznego budowanych aplikacji – monolitu z nowym podejściem – opartym o mikrousługi.

Kolejne rozdziały zawierają opis propozycji prototypów oraz przedstawienie użytych w nich technologii oraz szczegółów implementacyjnych.

Ostatni rozdział przedstawia podsumowanie, wnioski i ocenę obu architektur oraz słuszności procesu.

2. Systemy klasy CRM

Koncepcja CRM (Customer-relationship management) to idea, w której główny nacisk skierowany jest na zarządzanie i utrzymanie relacji z aktualnym, bądź potencjalnym klientem [1].

System klasy CRM to system, który wspomaga te procesy. W praktyce, jego rolą jest automatyzacja pracy wykonywanej w działach:

- Marketingu;
- Sprzedaży;
- Obsługi klienta;
- Zarządu.

Z tego podziału, wynikają konkretne zadania, które systemy tego typu realizują takie jak:

- Automatyzacja marketingu – tworzenie kampanii reklamowych, automatyczne wysyłanie wiadomości reklamowych;
- Automatyzacja sprzedaży – zarządzanie danymi sprzedażowymi klientów, analiza preferencji i potrzeb klienta na podstawie jego zamówień i przeglądanych produktów;
- Automatyzacja obsługi klienta – automatyzacja wszelakich form komunikacji na poziomie klient – usługodawca w celu jak najszybszej obsługi zgłoszenia.

W dobie rozwoju Internetu, każda z firm oferująca jakiekolwiek usługi (niekoniecznie związane bezpośrednio z technologią) wyróżnia potrzebę przyśpieszenia procesu kontaktu z klientem, jak i procesów funkcjonujących w przedsiębiorstwie.

Przykładowe sektory odnotowujące potrzebę wdrożenia systemu CRM to:

- Branża telekomunikacyjna (portal do zarządzania usługami klienta);
- Branża e-commerce (zakupy, reklamacje zamówień);
- Branża finansowa (samodzielne składanie wniosków o kredyt);
- Wszystkie branże związane z opłatami subskrypcyjnymi (płatności rachunków przez Internet).

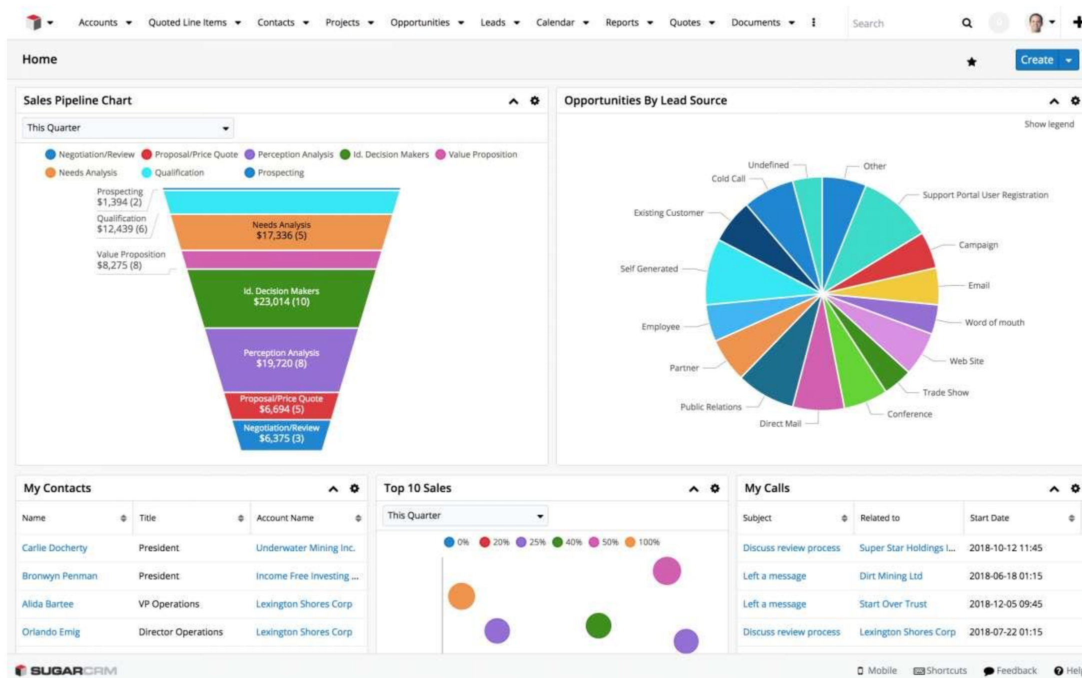
Nie sposób wymienić wszystkich obszarów potrzebujących rozwiązań tego typu. Faktem jest, iż systemy te muszą działać sprawnie, stabilnie i szybko w dobie rosnących oczekiwań klienta.

Wiele firm decyduje się na implementacje autorskich systemów CRM. W takim przypadku, są ono projektowane i wdrażane specjalnie na potrzeby pojedynczego klienta. System posiada wtedy funkcjonalności „szyte na miarę”, związane ze konkretnym obszarem biznesowym i w pełni zgodne z upodobaniami i życzeniami klienta.

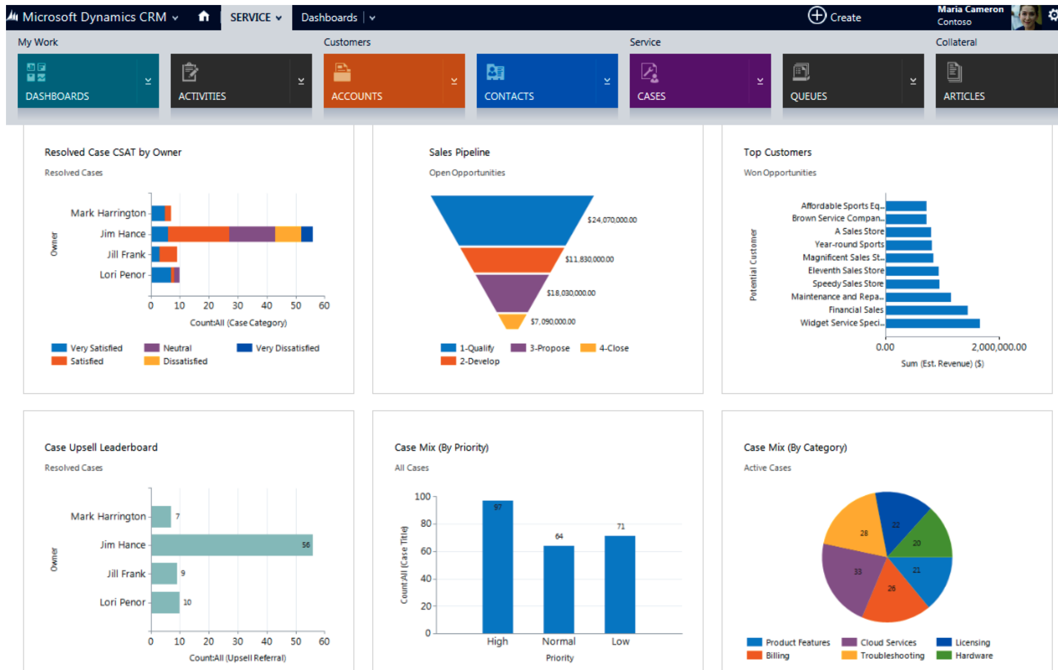
Istnieje jednak możliwość wdrożenia gotowego systemu tego typu, a następnie skonfigurowania go na potrzeby firmy. Przykładem takich rozwiązań są:

- SugarCRM (przedstawiony na Rysunku 1) [2] – oprogramowanie realizujące typowe funkcjonalności CRM. Występuje w dwóch wersjach: darmowej (instalowanej na własnym serwerze aplikacyjnym) oraz płatnej (sprzedawanej jako usługa w chmurze). Do końca roku 2018 była dostępna wersja „community" na licencji Open Source;
- Microsoft Dynamics CRM (przedstawiony na Rysunku 2) [3] – oprogramowanie stworzone przez firmę Microsoft. Jest częścią grupy produktów Microsoft Dynamics 365 służących do zarządzania przedsiębiorstwem. Sprawnie integruje się z oprogramowaniem typu ERP. Sprzedawane jest głównie jako usługa w chmurze, oferuje aplikacje klienckie na wszystkie popularne platformy oraz dostęp poprzez przeglądarkę internetową
- Salesforce (przedstawiony na Rysunku 3) [4] – kompleksowa platforma CRM / ERP wspierająca funkcjonowanie przedsiębiorstwa. Zawiera kilkanaście modułów, umożliwia integracje z większością konkurencyjnych rozwiązań. Dostarczane tylko jako usługa w chmurze. Jest to jedno z najpopularniejszych „gotowych" rozwiązań CRM ze względu na swoją elastyczność.

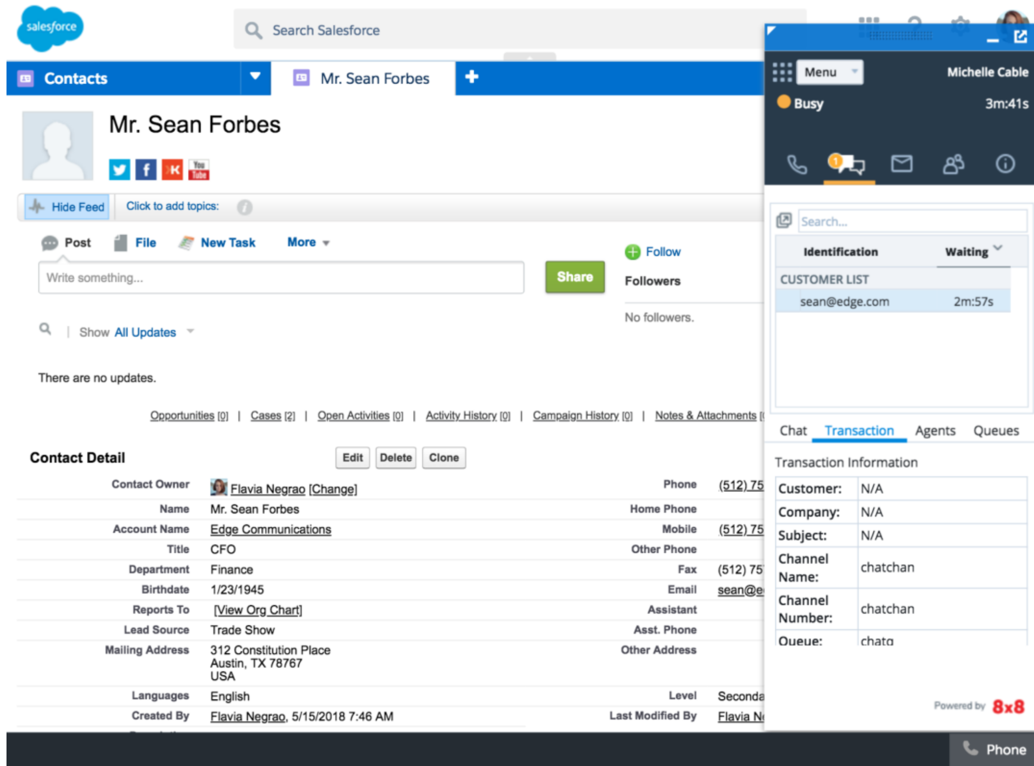
Wszystkie te platformy łączy jeden wspólny mianownik – większość typowych funkcjonalności takich jak zarządzanie relacjami z klientami czy też automatyzacja sprzedaży jest domyślnie zaimplementowana. W praktyce, każde z rozwiązań oferuje ten sam zestaw funkcjonalności, różniący się niewielkimi szczegółami, często związanymi z integracją. Amerykańska firma SelectHub pokusiła się o wydanie artykułu [5] w którym porównuje trzy wymienione w tej pracy platformy. Wynik porównania w kontekście konkretnych funkcjonalności został zawarty w Tabeli 1. Wynika z niego, że pod kątem biznesowym zdecydowanie dominują Mircosoft Dynamics CRM i Salesforce (zaawansowana obsługa wielu z nich).



Rysunek 1. Zrzut ekranu platformy SugarCRM [6].



Rysunek 2. Zrzut ekranu platformy Microsoft Dynamics CRM [7].



Rysunek 3. Zrzut ekranu platformy Salesforce [8].

Nr	Funkcjonalność	Opis	Microsoft Dynamics CRM	Salesforce	SugarCRM
1	Selekcja klientów	Dostarcza narzędzia do tworzenia kampanii marketingowych ukierunkowanych na konkretnych typ klienta	Podstawowa obsługa	**Zaawansowana obsługa**	Podstawowa obsługa
2	Analiza kampanii	Obsługa metryk typu. Zwrot z inwestycji etc.	**Zaawansowana obsługa**	Podstawowa obsługa	Podstawowa obsługa
3	Pozyskiwanie potencjalnych klientów	Dostarcza narzędzia pozwalające na generowanie jak największej liczby potencjalnych klientów	Podstawowa obsługa	Podstawowa obsługa	Podstawowa obsługa
4	Zarządzanie zgłoszeniami	Dostarcza narzędzia do zarządzania zgłoszeniami (funkcjonalności helpdeskowe, namierzanie zamówień, sprzedaż dóbr)	**Zaawansowana obsługa**	Podstawowa obsługa	Podstawowa obsługa
5	Narzędzia email	Dostarcza narzędzia do masowej wysyłki wiadomości email, kategoryzacji i analizy	Podstawowa obsługa	**Zaawansowana obsługa**	Podstawowa obsługa
6	Integracja z mediami społecznościowymi	Dostarcza mechanizmy do integracji z mediami społecznościowymi	**Zaawansowana obsługa**	Podstawowa obsługa	Podstawowa obsługa

Tabela 1. Porównanie oprogramowań CRM: Microsoft Dynamics CRM, Salesforce oraz SugarCRM [5].

W przypadku gdy przedsiębiorstwo potrzebuje wdrożenia specyficznych dla siebie procesów, następuje potrzeba dostosowania gotowego oprogramowania do tych potrzeb. Każde z rozwiązań oferuje możliwości rozszerzenia go poprzez pisanie odpowiednich wtyczek / modułów. Niestety, często wymaga to zatrudnienia wysokiej klasy konsultantów co w większości przypadków jest bardzo kosztowne. Nierzadko, gotowe produkty zawierają funkcjonalności / moduły za które trzeba zapłacić nawet jeśli nie są wykorzystywane.

Trudno jest ocenić, które z podejść jest lepsze: zaprojektowanie własnego systemu CRM czy też użycie gotowego i dostosowanie go. W pierwszym przypadku przedsiębiorstwo samo musi zadbać o implementacje systemu, utrzymanie i infrastrukturę. W drugim; infrastruktura jest dostarczana przez dostawcę narzędzia co znosi odpowiedzialność ze strony firmy, jednak wymaga to uiszczania cyklicznych opłat subskrypcyjnych.

W tej pracy, skupiono się na analizie stylów architektonicznych używanych przy projektowaniu systemów CRM od podstaw. Aktualnie, zdecydowana większość z nich jest zaprojektowana w architekturze monolitycznej, która wydaje się nie zaspokajać powyższych wymagań. Istnieje więc potrzeba wprowadzenia zmian architektonicznych, które będą odpowiedzią na te problemy.

3. Analiza istniejących architektur

W poniższym rozdziale zostały omówione konkurencyjne architektury z uwzględnieniem wszystkich istotnych aspektów projektowania oraz definicje ułatwiające zrozumienie dalszej treści. Dodatkowo, aspekty te zestawiono w ramach tabeli z czynnikami porównawczymi, aby wprost podsumować różnice dzielące oba podejścia .

Definicja 1

Wdrożenie (ang. deployment) - zbiór wszystkich czynności technicznych które sprawiają, że oprogramowanie jest dostępne do użycia.

Definicja 2

Interfejs Programistyczny Aplikacji (ang. Application Programming Interface) [9] - ściśle określony zestaw reguł i ich opisów, w jaki programy komputerowe komunikują się między sobą**.**

Definicja 3

Dług technologiczny – zjawisko, które powoduje wzrost złożoności produktu rosnący z powodu szybkich, nieprzemyślanych wdrożeń w przeciwieństwie do wolniejszych, stabilnych aktualizacji.

Definicja 4

Redundancja danych [10] – nadmiarowość w stosunku do tego, co konieczne lub zwykłe. W szczególności w odniesieniu do relacyjnych baz danych dąży się do sytuacji, gdy każda relacja zawiera unikalne informacje oraz klucze łączące z innymi relacjami. W szczególnych przypadkach, w celu przyśpieszenia obróbki danych, następuje rezygnacja z relacji, jednak może to być źródłem błędów i wewnętrznej niespójności bazy.

3.1. Architektura monolityczna

Monolit jest to wzorzec, który charakteryzuje jedno podstawowe założenie; wszystkie komponenty systemu działają w ramach jednej aplikacji. Są więc od siebie ściśle zależne implementacyjne i traktowane jako jedna całość przez serwer aplikacyjny. W praktyce, wdrożenie systemu tego typu sprowadza się do wyprodukowania złożonej aplikacji (często z jednym schematem bazodanowym) oraz wdrożenia całego systemu i utrzymaniu go na pojedynczej maszynie obliczeniowej.

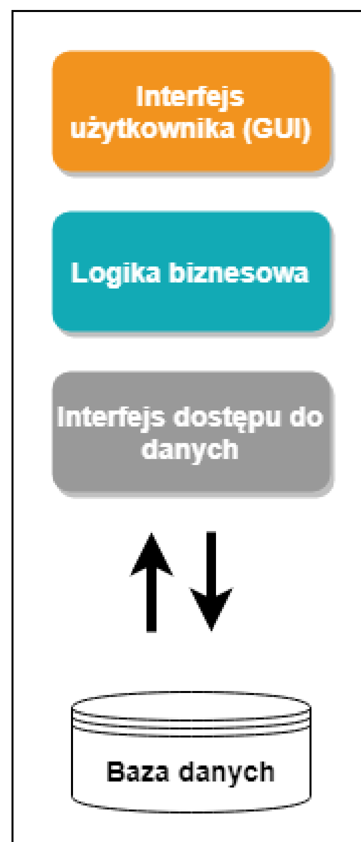
Podejście zostało to przedstawione na Rysunku 4 i ma niewątpliwie swoje zalety:

- Logika biznesowa znajduje się w jednym miejscu. Kod programistyczny opisujący w jaki sposób ma zachowywać się system zapisany jest wewnątrz jednej aplikacji;
- Brak zależności między innymi usługami. System w takim przypadku bardzo rzadko integruje się z innymi (wszystkie komponentu wewnątrz systemu) co jest zaletą, ponieważ brak zależności czyni go bardziej dostępnym;
- Szybka komunikacja między komponentami na poziomie pamięci serwera aplikacyjnego. Integracja zawsze jest kosztowna, ponieważ z reguły odbywa się na poziomie warstwy sieciowej co zawsze kosztuje określoną ilość czasu. W przypadku gdy komponenty działają w ramach jednej aplikacji – komunikacja odbywa się na poziomie pamięci RAM co jest najwydajniejszą z możliwych opcji.

W dobie błyskawicznego rozwoju Internetu, monolit przestaje być wystarczający, aby zaspokoić potrzeby dzisiejszego biznesu oraz prywatnych konsumentów.

Jego słabe strony to:

- Ograniczona możliwość modularyzacji (lub jej brak). W przypadku złożonych systemów, niemożliwe jest proste wydzielenie jego części do osobnego modułu. W praktyce, wymaga to ogromnych nakładów pracy programistyczno – administratorskiej co często jest nieopłacalne;
- Kosztowna skalowalność (konieczność skalowania całego systemu, wszystkich funkcjonalności jednocześnie). W przypadku, gdy pojedyncza funkcjonalność systemu jest najbardziej eksploatowana (np. wysyłka wiadomości) – nie ma prostej możliwości przeskalowania tylko jej - wszystkie komponenty to jeden serwer aplikacyjny;
- Nieelastyczność (wprowadzenie najdrobniejszej zmiany wymaga wdrożenia całego systemu) – jest to konsekwencja braku podziału na moduły;
- Rosnący dług technologiczny będzie dotyczył implementacji wszystkich komponentów systemu. Szczególnie, dotyczy to technologii bazodanowych. Raz wybrany silnik baz danych wykorzystywany przez cały system jest związany z nim do końca życia systemu. W przypadku gdy przestanie być wystarczający, zmiana go będzie dotyczyła wszystkich funkcjonalności co zwiększy koszt wdrożenia.



Rysunek 4. Architektura systemu monolitycznego.

3.2. Architektura mikrousług

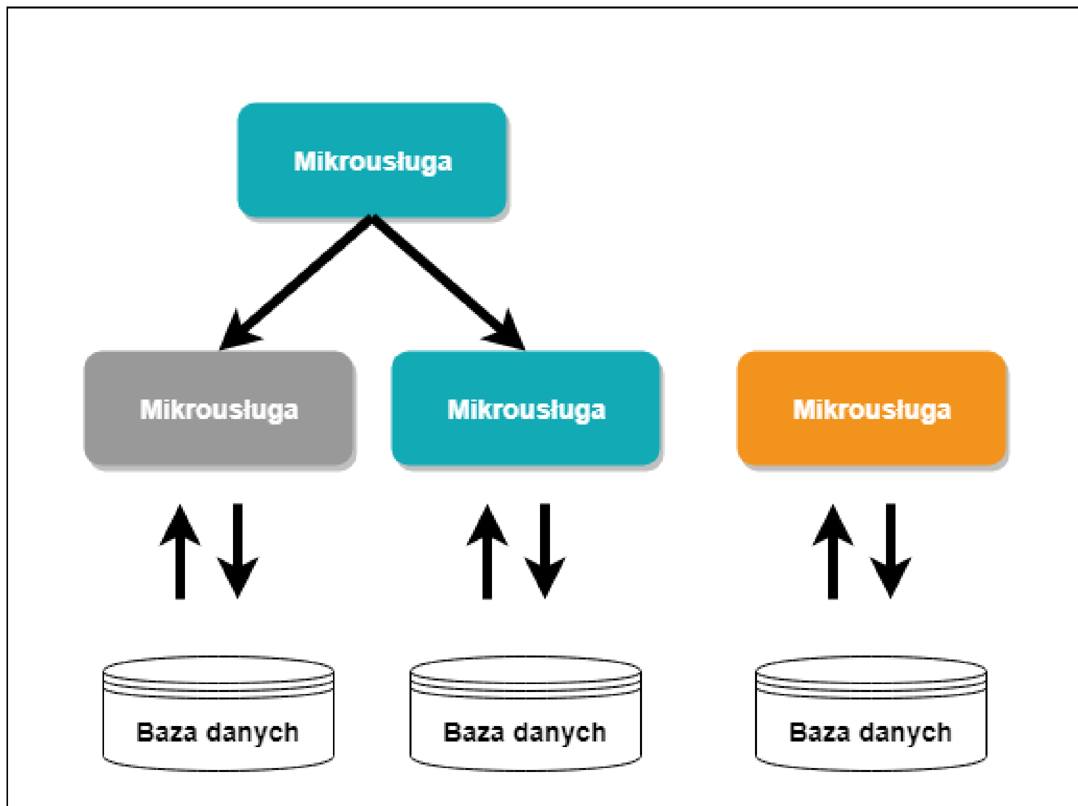
Rozwiązaniem tych problemów ma stać się architektura mikrousług, zobrazowana Rysunkiem 5. Podejście to zakłada, że cały system (traktowany logicznie, jako jeden byt) składa się z wielu rozdzielonych i autonomicznych usług, co pozwala na niezależne ich skalowanie [11]. Dodatkowo, jednym z założeń takiej kompozycji jest komunikacja głownie na poziomie API lub kolejek. Tak więc, jeśli jedynym kontraktem między poszczególnymi usługami jest specyfikacja interfejsu, to każda funkcjonalność aplikacji może zostać wydzielona i napisana w innym języku programowania. Istotne jest jedynie to, aby przesyłane komunikaty były zrozumiałe dla każdego z integratorów. Poniżej wypisano zalety i wady tego rozwiązania w odniesieniu do analogicznych kwestii rozważanych wcześniej dla monolitu.

Zalety podejścia:

- System jest modularny z założenia. Już na początku projektowania wiadome jest o modularności systemu. Warto zauważyć, że jest to bardzo ważnym aspektem podziału na moduły (usługi) jest jego kryterium. Platforma może być dzielna na moduły w zależności od obszarów biznesowych (oddzielne moduły obsługujące klientów polskich / zagranicznych) lub technicznych (oddzielne moduły do wysyłki wiadomości / rejestracji klientów);
- Tania skalowalność (skalowane są konkretne usługi). Jest to rozwiązanie zdecydowanie tańsze niż skalowanie całego systemu chociażby ze względu na koszty zasobów (mniejsza ilość pamięci RAM / CPU do zakupu / wydzierżawienia) ;
- Cykl wydawniczy zależny od konkretnej usług a nie całego systemu. Naturalna konsekwencja modularności – wdrażany jest tylko ten fragment, który ma być zmieniony – reszta funkcjonalności w teorii powinna zostać nienaruszona;
- Potencjalny dług technologiczny będzie dotyczył implementacji pojedynczej usługi. Zakładając, że każdy z modułów ma oddzielny silnik bazodanowy – mniejsze ryzyko awarii systemów poniesiemy robiąc migracje kolejno pojedynczych modułów niz całego systemu za jednym razem)

Wady podejścia:

- Logika biznesowa znajduje się w wielu miejscach. Duża ilość zależności między innymi usługami oznacza, że fragmenty kodu znajdują się w kilku miejscach. Powoduje to rozwarstwienie logiki i brak prostej struktury;
- Stosunkowo wolna komunikacja między usługami na poziomie warstwy sieciowej. Modularność powoduje dużą ilość integracji, która jest kosztowna. Każde połączenie sieciowe to utrata czasu – nawet jeśli usługi znajdują się w tej samej sieci;
- Redundancja danych która jest konsekwencją separacji logiki biznesowej i odpowiedzialności. Podział na moduły powoduje, że prawdopodobnie będzie konieczna duplikacja części kodu źródłowego jak i danych przechowywanych w warstwach trwałości poszczególnych usług (temat omówiony szczegółowo w kolejnych rozdziałach);
- Problematyczna implementacja transakcyjności. Brak możliwości wykorzystania prostych transakcji na poziomie ORM (Hibernate) które mogą działać tylko w ramach pojedynczych modułów.



Rysunek 5. Architektura systemu opartego o mikrousługi.

3.3. Porównanie

W Tabeli 2 przedstawiono porównanie obu podejść w kontekście różnych czynników/obszarów związanych z wytwarzaniem oprogramowania.

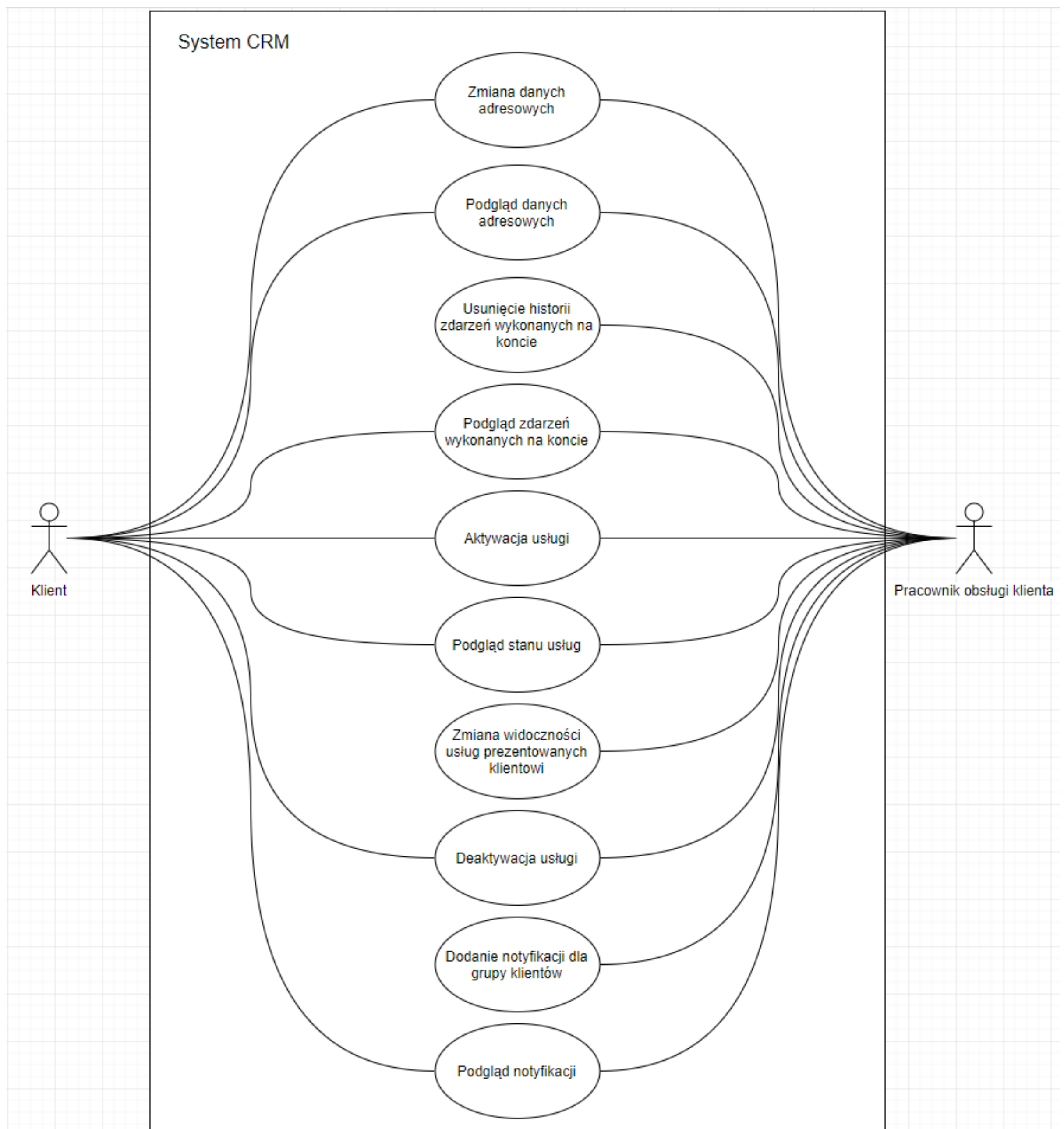
Nr	Czynnik porównania	Architektura monolityczna	Architektura mikrousług
1	Ilość modułów	Jeden	Wiele – ilość zwiększa się w raz rozwojem systemu
2	Skalowalność	Cały system	Każda mikrousługa może być skalowana oddzielnie, w zależności od potrzeb
3	Logika biznesowa	Logika biznesowa zawarta w jednym miejscu	Rozproszona logika biznesowa. Każda mikrousługa odpowiada za inną część systemu
4	Zależność systemu	Brak	System funkcjonalnie jest zależny od wielu mikrousług
5	Komunikacja między komponentami	Szybko – wszystkie komponenty w pamięci serwera	Wolniej – komponenty rozproszone na wiele mikrousług komunikują się na poziomie sieciowym poprzez API
6	Technologia	Jeden zestaw technologii	Każda mikrousługa może być napisana w innej technologii
7	Stabilność	Niedostępność systemu powodują niedostępność wszystkich jego funkcjonalności	Niedostępność pojedynczej mikrousługi powoduje niedostępność tylko części funkcjonalności biznesowych
8	Rozwój	Jakakolwiek zmiana w systemie powoduje konieczność wdrożenia całego systemu	Zmiana w systemie może dotyczyć pojedynczej mikrousługi, nie wpływa na funkcjonowanie reszty systemu
9	Warstwa trwałości	Jedna baza danych	Każda mikrousługa posiada swoją bazę danych. Czasem konieczna. może być redundancja danych
10	Ilość środowisk produkcyjnych / deweloperskich	Jedno per typ (staging, prod, etc.)	Wiele (w praktyce każda mikrousługa to oddzielne środowiska per typ)
11	Koszt utrzymania środowisk	Niższy (mniejsza liczba)	Większy (w praktyce wymagana konteneryzacja)

Tabela 2. Porównanie architektur.

4. Propozycja rozwiązania

W celu weryfikacji czynników porównawczych zawartych w Tabeli 2 zrealizowano prototyp aplikacji, który realizuje typowe funkcjonalności biznesowe zapewnianie klientom przez większość dużych organizacji.

Na Rysunku 6 przedstawiono fragment diagramu przypadków użycia systemu w kontekście interakcji możliwych do dokonania przez klienta oraz pracownika obsługi. Wyróżnione zostały te czynności, które występują w praktycznie wszystkich systemach CRM z obszaru telekomunikacji / bankowości / mediów i są dla nich typowe.



Rysunek 6. Diagram przypadków użycia systemu w kontekście klienta oraz pracownika obsługi klienta.

Wykonany prototyp dotyczy części systemu i nie uwzględnia wszystkich przedstawionych przypadków użycia. Implementuje funkcjonalności możliwe do wykonania tylko przez klienta (wykluczając pracownika obsługi) takie jak:

- Zmiana, podgląd danych adresowych;
- Podgląd zdarzeń wykonanych na koncie;
- Aktywacja / deaktywacja / podgląd stanu usług;
- Podgląd notyfikacji.

Warstwa prezentacji została ograniczona do strony głównej oraz modali kontekstowych. Rozwiązanie jest jednak projektowane generycznie, aby w przyszłości była możliwość jego łatwej rozbudowy. Mechanizm stron oraz komponentów wytworzony w ramach serwera front-end (Rozdział 6.1) pozwala na edycje i dodawanie nowych funkcjonalności w czasie rzeczywistym, bez potrzeby restartu aplikacji.

4.1. Wymagania funkcjonalne

Poniżej przedstawiono listę wymagań funkcjonalnych które określają funkcje możliwe do wykonania w systemie.

1. Użytkownik ma możliwość podglądu swoich aktywnych usług.
2. Użytkownik ma możliwość aktywacji konkretnej usługi.
3. Użytkownik ma możliwość podglądu swoich danych.
4. Użytkownik ma możliwość podglądu swoich adresów.
5. Użytkownik ma możliwość ustawienia konkretnego adresu jako główny.
6. Każda akcja wykonana w systemie (zmiana adresu głównego, aktywacja usługi) powinna być logowania w formie zdarzeń.
7. Wszystkie zdarzenia powinny być wyświetlone użytkownikowi w postaci osi czasu.
8. Użytkownik ma możliwość podglądu komunikatów skierowanych do niego przez administratorów systemu.

4.2. Wymagania jakościowe

Poniżej przedstawiono listę wymagań jakościowych (niefunkcjonalnych), opisujących kryteria działania systemu w kontekście technologii / wydajności / dostępności i innych poza funkcjonalnych cech.

1. Wysoka dostępność systemu – system powinien być przygotowany na duże ilości jednoczesnych żądań.
2. System powinien być zaimplementowany w wiodących, sprawdzonych technologiach w świecie technologii webowych.
3. System powinien być dostępny dla użytkownika z poziomu przeglądarki Internetowej.

5. Wykorzystane narzędzia i technologie

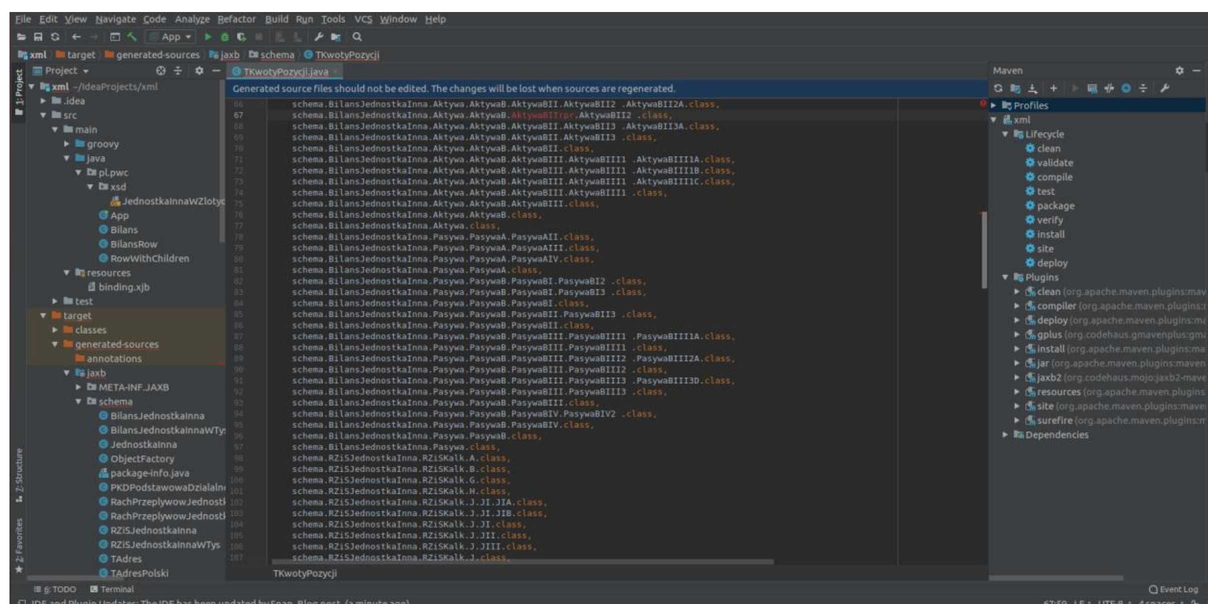
W rozdziale zostały zaprezentowane technologie (oraz powiązane z nimi definicje) i narzędzia wykorzystane do wytworzenia prototypu aplikacji monolitycznej, mikro-usługowej oraz serwera front-end.

Definicja 5

Zintegrowane środowisko programistyczne, IDE (ang. integrated development environment) [12] – program lub zespół programów (środowisko) służących do tworzenia, modyfikowania, testowania i konserwacji oprogramowania.

5.1. IntelliJ Idea

IntelliJ Idea [13] to wiodące środowisko programistyczne do tworzenia aplikacji. Zawiera wsparcie narzędziowe dla większości języków programowania oraz moduły, które przyśpieszają wiele koniecznych do wykonania czynności, jak na przykład instalacja aplikacji na serwerze lub tworzenie dokumentacji na podstawie metod. Występuje w dwóch wersjach: Community oraz Ultimate. Pierwsza - bezpłatna zawiera okrojoną część wszystkich funkcji dostępnych w wersji Ultimate. Samo środowisko jest zamknięte, jednak jest aktywnie rozwijane przez niezależnych deweloperów, którzy implementują nowe funkcjonalności i dostarczają je w postaci wtyczek. Na Rysunku 7 przedstawiono zrzut ekranu okna uruchomionego środowiska.



Rysunek 7. Środowisko programistyczne IntelliJ Idea.

5.2. Javascript

Javascript [14] – język programowania, którego pierwsza wersja powstała w 1995 roku. Obsługiwany przez wszystkie wiodące przeglądarki. Wykorzystuje się go w celu manipulacji stroną Internetową (drzewem DOM) w celu zapewnienia interaktywności. Javascript uruchamia się po stronie klienta (przeglądarki) i pozwala na modyfikacje kodu HTML już po jego załadowaniu, dzięki czemu możliwe jest wykonanie takich czynności jak żądania AJAX czy dynamiczna zmiana treści / animacje.

Główne cechy podstawowej wersji języka to:

- Typowanie dynamiczne – nie ma potrzeby używania typów, kompilator sam je ustala na podstawie wartości zmiennych;
- Brak kompilacji – kod jest interpretowany na żywo;
- Wieloplatformowość – wszystkie przeglądarki (niezależnie od systemu operacyjnego) które obsługują język Javascript są w stanie wykonać ten sam kod i otrzymać deterministyczne wyniki;
- Jednowątkowość – kod wykonuje się tylko na jednym wątku procesora.

W dzisiejszych czasach bardzo rzadko używa się „czystego Javascriptu" – z reguły wykorzystuje się frameworki bazujące na nim takie jak jQuery, czy też bardziej nowoczesne: Angular bądź React.js.

5.3. NodeJS

NodeJS [15] to środowisko uruchomieniowe pozwalające na wykonanie kodu Javascript po stronie serwera, stworzone przez Google w ramach silnika V8. Łączy zalety podstawowej wersji języka z innymi, kompilowalnymi językami. Silnik nie interpretuje kodu w czasie rzeczywistym (tak jak przeglądarka) tylko kompiluje go do kodu maszynowego co pozytywnie wpływa na wydajność. Główną ideą twórców była unifikacja rozwiązań używanych w typowych stronach Internetowych. Javascript, poza frameworkiem do tworzenia warstwy prezentacji może być teraz używany również do tworzenia rozwiązań backendowych.

NodeJS pozwala na łatwe wykorzystanie wbudowanych pakietów oferujących dodatkowe funkcjonalności poprzez menedżer NPM. Pozwala to na ponowne użycie typowych komponentów (jak np. komponent do obsługi protokołu http). W Listingu 1 przedstawiono przykładowy kod źródłowy napisany dla środowiska NodeJS.

Listing 1. Przykładowy kod wykonywany w środowisku JS tworzący prosty serwer HTTP.

```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'ContentType': 'text/plain'});
  res.end('Hello World!');
}).listen(8080);
```

5.4. Java

Java [16] to wysokopoziomowy, obiektowy język programowania stworzony przez firmę Sun Microsystems a następnie przejęty przez Oracle. W przeciwieństwie do Javascriptu, służy do uruchamiania aplikacji po stronie serwera, jest silnie typowany oraz wielowątkowy. Kod napisany w tym języku kompiluje się do kodu bajtowego który następnie uruchamiany jest na JVM (Java Virtual Machine) – wirtualnej maszynie Java.

Obsługuje wszystkie standardowe paradygmaty programowania obiektowego takie jak:

- Dziedziczenie;
- Abstrakcja;
- Hermetyzacja;
- Polimorfizm.

Producenci języka udostępniają dwa główne zestawy pakietów: JRE (Java Runtime Environment) służący do uruchamiania aplikacji oraz JDK (Java Development Kit) służący do programowania aplikacji.

5.4.1 Spring Framework

Spring Framework to framework do tworzenia aplikacji utworzony w ramach alternatywy dla EJB (ang. *Enterprise Java Beans*) [17].

Składa się z kilkudziesięciu modułów zapewniających takie funkcjonalności jak m.in:

- Data – dostęp do baz danych;
- MVC – implementuje wzorzec Model View Controler w ramach aplikacji webowych;
- Core Container – implementuje wzorzec Inversion of Control w ramach wstrzykiwania zależności (Dependency Injection);
- Test – pozwala na łatwe i szybkie pisanie testów integracyjnych;
- AOP – implementuje paradygmaty programowania aspektowego.

5.4.2 JSP (JavaServer Pages)

Technologia umożliwiająca tworzenie stron WWW z wykorzystaniem języka Java wplecionego w kod HTML. Pozwala na dostęp do zmiennych utworzonych w ramach JVM i użycia ich do produkcji widoków.

Strony JSP składają się z następujących elementów:

- Treść statyczna – czysty kod HTML bez modyfikacji;
- Dyrektywy – metadane definiujące stronę;
- Elementy skryptowe;
- Akcje JSP – tagi XML wywołujące metody serwerowe.

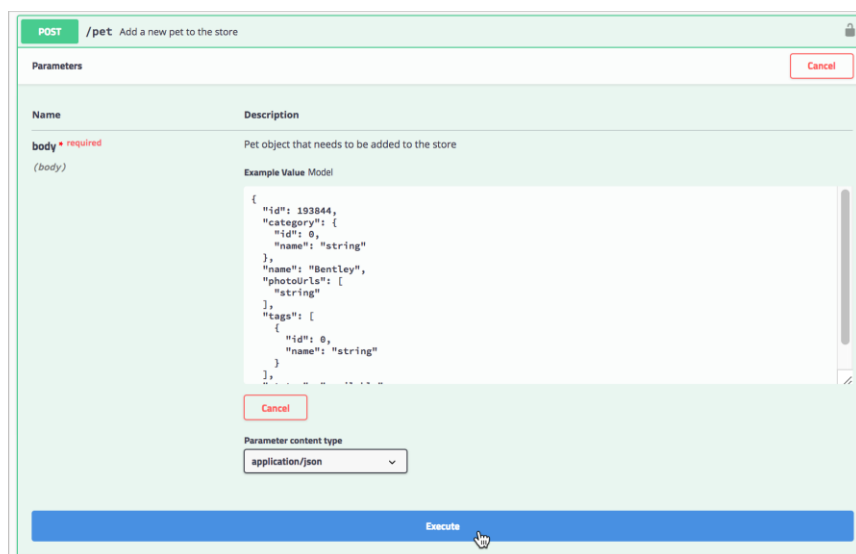
W Listingu 2 przedstawiono przykładowy kod źródłowy napisany w składni JSP.

Listing 2. Przykładowy plik JSP zawierający kod Java wpleciony w kod HTML.

```
<html>
<head><title>First JSP</title></head>
<body>
 <%
   double num = Math.random();
   if (num > 0.95) {
 %>
    <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
 <%
   } else {
 %>
    <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
 <%
   }
 %>
 <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
</body>
</html>
```

5.4.3 Swagger (Swagger UI)

Framework do projektowania i tworzenia dokumentacji dla API w aplikacjach napisanych w języku Java. Na podstawie adnotacji oraz zewnętrznej konfiguracji tworzy automatycznie stronę internetową (Rysunek 8) zawierającą auto-generowany opis metod HTTP oraz panel umożliwiający testowanie bezpośrednio z poziomu przeglądarki.



Rysunek 8. Zrzut ekranu z aplikacji Swagger UI zawierający udokumentowaną metodę HTTP wraz z panelem do testu interfejsu.

5.5. HTML

HTML [18] (*Hypertext markup language*) – hipertekstowy język znaczników definiujący stronę Internetową. HTML pozwala opisać strukturę informacji strony Internetowej. Definiuje takie elementy jak akapity, paragrafy, nagłówki. Posiada znaczniki pozwalające na osadzenie obiektów multimedialnych takich jak filmy lub dźwięk. Interpretowany przez wszystkie przeglądarki Internetowe.

Język HTML składa się z poniższych komponentów:

- Znaczniki i ich atrybuty (np. „img" który pozwala na osadzenie zdjęcia);
- Typy danych (np. „script" który pozwala na osadzenie kodu Javascript);
- Referencje znakowe (np. „amp" pozwalający wyświetlić znak „&");
- Deklaracje typu dokumentu (pozwalające określić użytą wersję HTML np. „<!DOCTYPE html>").

5.6. CSS

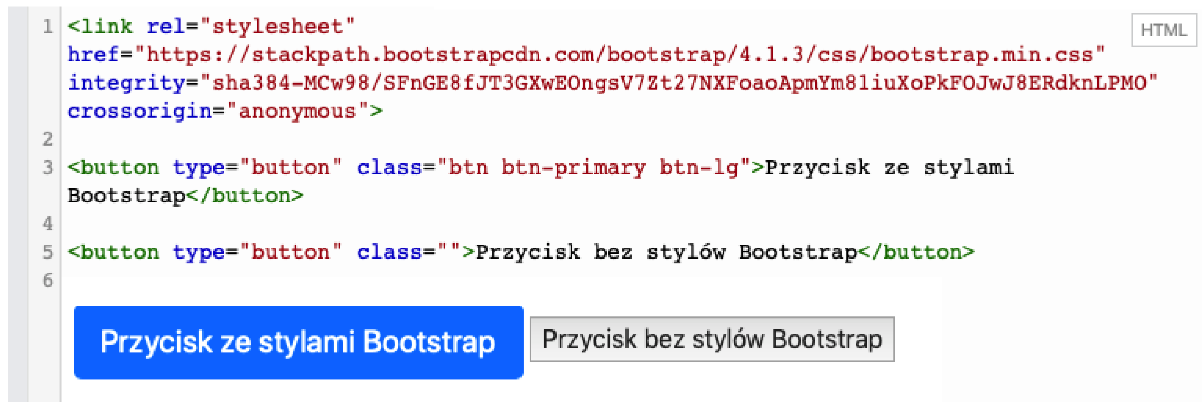
CSS [19] (*Cascading Style Sheets*) – język służący do opisu warstwy prezentacji stron HTML. Może definiować atrybuty takie jak: rodzaj, kolor czcionki, marginesy czy odstępy. Określa zestaw reguł (selektorów) i instrukcji do wykorzystania przez programistę w celu nadania stylu elementom drzewa DOM definiowanym przez HTML. Przykładowym wykorzystaniem może być określenie reguły, która sprawi, że każdy element węzła <h1> oraz jego dzieci zostaną wyświetlone kursywą. Zaletą takiego podejścia jest ponowne wykorzystanie kodu w przeciwieństwie do określania styli dla każdego elementu oddzielnie i kopiowanie go. Dodatkowo, CSS pozwala na przeniesie wszystkich informacji na temat wyglądu dokumentu do oddzielnego pliku, co powoduje uproszczenie samego źródła HTML. Daje to również możliwość łatwej zmiany stylu na inny. W Listingu 3 przedstawiono przykładową regułę CSS.

Listing 3. Przykładowa definicja reguły CSS która w rezultacie sprawi, iż wszystkie elementy <p> zostaną wyświetlone na środku strony Internetowej, kolorem czerwonym.

```
p {
  text-align: center;
  color: red;
}
```

5.7. Bootstrap

Bootstrap [20] to biblioteka CSS zawierająca gotowe do użycia, atrakcyjne wizualnie style dla elementów HTML. Główną ideą jest możliwość szybkiej budowy warstwy prezentacji na zasadzie reużycia definicji gotowych komponentów. Aby dodać bibliotekę do swojego projektu wystarczy zaimportować plik CSS ze strony producenta. Biblioteka została stworzona i udostępniona przez programistów serwisu Twitter, więc większość komponentów zdecydowanie przypomina te z serwisu. Na Rysunku 9 przedstawiono kod HTML z dodaną biblioteką Bootstrap oraz przykładem użycia jego styli do nadania atrakcyjnego wyglądu elementowi <button>.



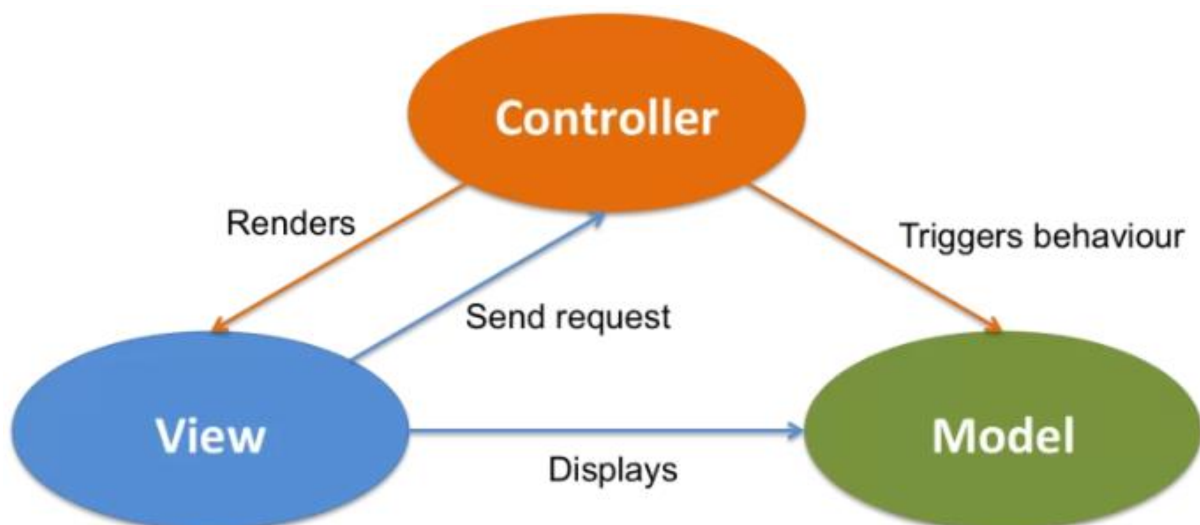
Rysunek 9. Kod HTML z dodaną biblioteką Bootstrap oraz przykładem użycia jego styli do nadania atrakcyjnego wyglądu elementowi <button>.

5.8. MVC

MVC [21] (*Model-View-Controller*) – wzorzec architektoniczny stosowany przy tworzeniu systemów informatycznych posiadających interfejs użytkownika. Wymusza podział aplikacji na trzy niezależne, komunikujące się ze sobą warstwy:

- Model – opis struktury, typów danych używanych do produkcji widoku;
- View (widok) – definicja interfejsu wizualnego, korzysta z modelu;
- Controller (kontroler) – główny punkt wejścia aplikacji. Zawiera logikę aplikacji, przygotowuje dane, aktualizuje model i generuje widok.

Na Rysunku 10 przedstawiono diagram odpowiedzialności i komunikacji między warstwami.



Rysunek 10. Architektura MVC [22].

Schemat działania MVC w przypadku jego implementacji w frameworku Spring wygląda następująco:

1. W momencie, gdy do kontrolera trafi żądanie wygenerowania określonej strony, ten musi w jakiś sposób skonstruować i wypełnić model danymi. Najczęściej dzieje się to poprzez jawne pobranie danych z bazy danych, lub też z zewnętrznych serwisów;
2. Następnie, model jest kojarzony (*bindowany*) z widokiem, którego referencja przekazywana jest do silnika szablonów;
3. Silnik szablonów generuje widok (uruchamia jego wewnętrzną logikę) i zwraca kod HTML, który jest efektem wykonania żądania.

MCV narzuca ściśle określone granice odpowiedzialności, które ma wiele zalet, takich jak:

- Podział na moduły (czytelność kodu);
- Model nie jest ściśle związany z widokiem. Może istnieć wiele widoków które odwołują się do tego samego modelu. Dzięki temu, istnieje możliwość prezentacji tych samych danych w różnych kontekstach (użytkownik zalogowany/niezalogowany);
- Widoki mogą być modyfikowane niezależnie od modelu;

Jego wady to:

- Utrudnione testowanie widoków - widoki często zawierają własną, dodatkową logikę. Fakt, iż model jest również wygenerowany na podstawie określonej logiki sterowanej w kontrolerze czyni testowanie jednostkowe nieefektywnym;
- Złożoność – inicjalnie wdrożony wzorzec w projekcie staje się jego integralną częścią przez cały czas jego życia. Odejście od niego i przystosowanie implementacji do innego jest w praktyce niemożliwe. W przypadku projektów o architekturze monolitycznej jest to dość istotne, ponieważ może dojść do powstania długu technologicznego.

5.9. REST

REST (*Representational state transfer*) – styl architektury oprogramowania definiujący zasady tworzenia API [23]. Ściśle powiązany z metodami HTTP za pomocą których odbywa się dostęp do zasobów.

Określa następujące założenia:

- Bezstanowość – każde żądanie musi zawierać komplet informacji;
- URL jako zasób – żądania muszą operować na konkretnych zasobach a nie metodach biznesowych, np. /customer/1 w przeciwieństwie do /getFirstCustomer;
- Obsługa cache – serwer musi jasno zwracać informacje czy wynik żądania może być cache'owany;
- Samoopisywalność komunikatów – serwer zawsze przesyła informacje jakiego typu zwraca dane (*MIME type*).

5.10.HTTP

HTTP (*Hypertext transfer protocol*) to protokół przesyłania dokumentów hipertekstowych (w praktyce - jakichkolwiek danych) w sieci Internetowej. Zdecydowana większość informacja w sieciach komputerowych jest wymieniania dzięki jego użyciu. Wyróżnia się następujące metody HTTP (w kontekście REST):

- GET – pobranie zasobu wskazanego przez URI;
- HEAD – pobiera nagłówki zasobu (te same, co w przypadku GET);
- PUT – przyjęcie danych w celu aktualizacji całej zawartości encji;
- POST – przyjęcie danych w celu utworzenia nowej encji;
- DELETE – żądanie usunięcia danej encji;
- OPTIONS – informacje o opcjach i wymaganiach serwera;
- TRACE – diagnostyka;
- CONNECT – żądanie przeznaczone dla serwerów pośredniczących pełniących funkcje tunelowania;
- PATCH – przyjęcie danych w celu aktualizacji części zawartości encji.

Metody HTTP są efektywnie wykorzystane przez styl architektury REST (większość z nich) oraz SOAP (metoda POST).

5.11.MongoDB

Nierelacyjny, otwarto dostępny system zarządzania bazą danych typu NoSQL.

Charakteryzuje się dużą wydajnością i skalowalnością. Posiada gotowe rozwiązania umożliwiające replikacje danych, oraz frameworki to strumieniowego przetwarzania. Producent udostępnia również usługę w chmurze, która nie wymaga instalacji silnika na dysku. W odróżnieniu od bazy relacyjnej (gdzie dane są przechowywane w postaci tabel) dane są zawarte w formie dokumentów (BSON - wariacja JSON) i zawierają obiekty typu klucz-wartość. Obsługuje indeksowanie, kursory oraz od pewnego czasu transakcje pomiędzy dokumentami. Przykładowy dokument w bazie danych MongoDB został przedstawiony w Listingu 4.

Listing 4. Przykładowy dokument w bazie danych MongoDB.

```
{
 _id: ObjectId("5099803df3f4948bd2f98391"),
 name: { first: "Alan", last: "Turing" },
 birth: new Date('Jun 23, 1912'),
 death: new Date('Jun 07, 1954'),
 contribs: [ "Turing machine", "Turing test", "Turingery" ],
 views : NumberLong(1250000)
}
```

5.12. H2 Database

Relacyjny, otwarto dostępny system zarządzania bazą danych typu SQL.

Jego podstawową zaletą jest możliwość uruchomienia go w wersji „standalone” lub w ramach aplikacji Java. W takim przypadku serwer bazodanowy startuje w pamięci JVM naszej aplikacji. Nie ma potrzeby dodatkowej konfiguracji, instalacji (jak w przypadku Microsoft SQL Server lub Oracle) lub konfiguracji uprawnień. Jest świetnym rozwiązaniem, dla aplikacji które potrzebują prostego silnika bazodanowego, bez możliwości inwestycji w zewnętrzne utrzymanie serwera.

5.13. JMS (ActiveMQ)

JMS [24] (*Java Message Service*) – standard definiujący zestaw interfejsów i klas służący do asynchronicznego przesyłania komunikatów między komponentami napisanymi w języku Java. Specyfikacja została przedstawiona przez firmę SUN i jest darmowa. Komunikacja w JMS jest oparta na wysyłce wiadomości.

JMS składa się z następujących komponentów:

- JMS Provider – broker (system wymiany wiadomości);
- JMS Clients – producenci i konsumenci wiadomości;
- Messages (wiadomości i zawarte w nich obiekty);
- Obiekty administracyjne (fabryki połączeń etc.).

JMS obsługuje dwa tryby działania:

- Queue (kolejka - wiadomość ma zawsze jednego odbiorcę);
- Topic (kanał subskrypcji – wiadomość może mieć więcej niż jednego odbiorcę).

Jedną z implementacji JMS jest ActiveMQ [25] – otwarto odstępny broker wiadomości udostępniony jako jeden z projektów fundacji Apache.

5.14. Nginx

Serwer WWW oraz serwer proxy zaprojektowany z myślą o wysokiej dostępności i nastawiony na obsługę silnie obciążonych aplikacji.

Cechy serwera:

- Obsługa SSL;
- Wbudowane Reverse proxy;
- Obsługa plików statycznych / indeksów;
- Równoważenie obciążenia (*Load balancing* [26]) ;
- Brak obsługi .htaccess (w przeciwieństwie do konkurencyjnego rozwiązania Apache HTTP Server).

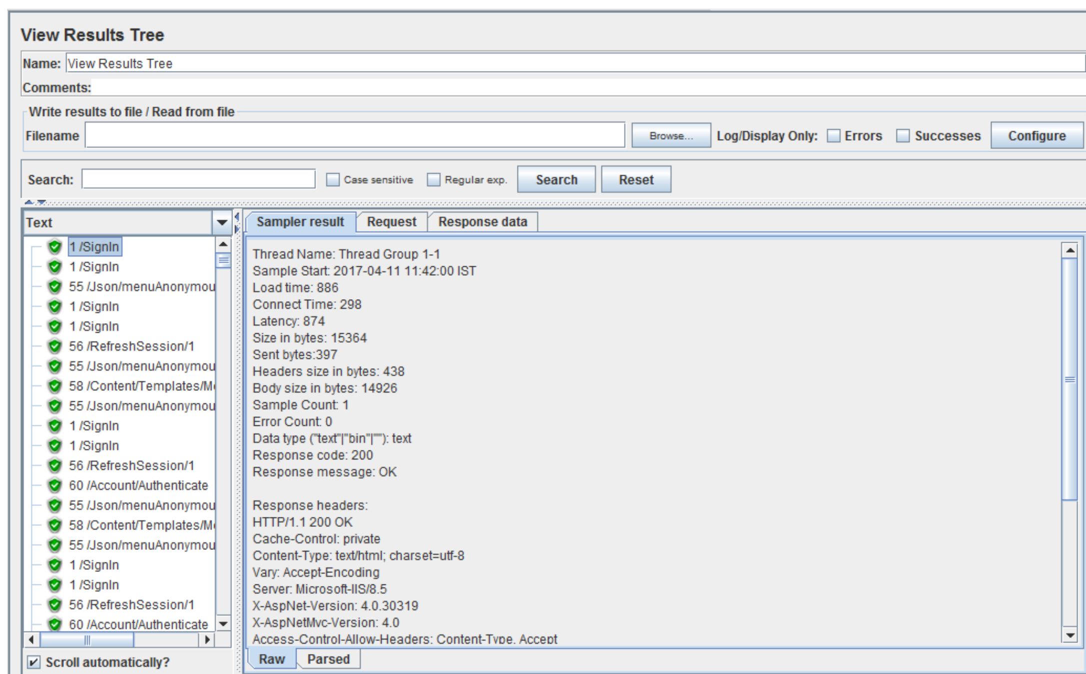
Jedną z zalet, poza podstawową funkcjonalnością jaką jest serwowanie plików HTML jest wbudowane rozwiązanie Równoważenia obciążenia. Pozwala ono na przekierowanie ruchu do różnych instancji serwerów na podstawie algorytmów takich jak:

- *round-robin* – zgodnie z algorytmem szeregowania procesów Round Robin;
- *least-connected* - żądanie zostanie przekierowane do serwera, który ma najmniej aktualnie aktywnych połączeń;
- *ip-hash* – żądanie zostanie przekierowane do serwera, który obsługuje daną pulę adresów na podstawie wyniku wywołania funkcji hashującej na adresie IP klienta.

5.15. JMeter

JMeter – narzędzie napisane w języku Java do przeprowadzania testów aplikacji [27]. Pierwotnie, obsługiwał tylko testy aplikacji WWW oraz serwerów FTP. Aktualnie obsługuje różnorodne scenariusze testowe dla testów funkcjonalnych, bazodanowych etc. Jego główne cechy to:

- Obsługa HTTP, JDBC, LDAP, SOAP a nawet natywnych procesów systemowych;
- Parametryzacja testów;
- Walidacja odpowiedzi żądań;
- Możliwość manipulacji nagłówkami w żądaniach;
- Możliwość automatyzacji i parametryzacji testów;
- Intuicyjny interfejs (przedstawiony na Rysunku 11);
- Prezentacja wyników w postaci wykresów, tabel.



Rysunek 11. Okno programu JMeter przedstawiające rezultaty przeprowadzonych testów.

6. Prototyp

W ramach prototypu został wykonany serwer front-end który serwuje warstwę prezentacji dla obu systemów: monolitycznego oraz opartego o mikrousługi. Jest to jedyny moduł współdzielony w ramach tego rozwiązania.

6.1. Serwer front-end

Serwer front-end został zaprojektowany z wyróżnieniem następujących komponentów:

- Strona (page) – definiuje stronę oraz jej adres pod jakim ma być dostępna;
- Slot – definiuje fragment strony który zostanie wyświetlony za pomocą jednego z serwisów;
- Serwis (service) – definiuje adres serwera zdalnego w raz jego adresem URL który odpowiada za dostarczanie widoków HTML umieszczone w slotach.

Wszystkie te parametry zostały zaimplementowane w ramach modelu konfiguracyjnego i są zawarte w pliku JSON (Listing 5) który jest ładowany podczas startu serwera.

Listing 5. Konfiguracja serwera front-end.

```
{
  "services": {
    "SUBSCRIPTION": "http://localhost:8014/subs/view",
    "SUMMARY": "http://localhost:8013/view",
    "ADDRESS": "http://localhost:8016/view",
    "EVENT": "http://localhost:8015/view",
    "MONOLITH": "http://localhost:8012/view",
  },
"pages":
[
  {
    "id": "monolithHome",
    "name": "Customer Service - monolith",
    "url": "monolith",
    "slots": {
      "slot1": {
        "path": "home",
        "service": "MONOLITH"
      }
    }
  },
  {
    "id": "microservicesHome",
    "name": "Customer Service - microservices",
    "url": "microservices",
    "slots":
      "summary": {
        "path": "home",
        "service": "SUMMARY"
      },
      "address": {
        "path": "home",
        "service": "ADDRESS"
      },
      "event": {
        "path": "home",
        "service": "EVENT"
      },
      "subsctription": {
        "path": "home",
        "service": "SUBSCRIPTION"
      }
    }
  }
]
}
```

W obiekcie **services**, zostało zdefiniowanych pięć usług, które serwują różne funkcjonalności biznesowe:

- **SUBSCRIPTION**
- **SUMMARY**
- **ADDRESS**
- **EVENT**
- **MONOLITH**

Każda usługa musi posiadać unikalny klucz oraz adres, pod którym jest dostępna.

W sekcji obiekcie **pages** zdefiniowano dwie strony:

- **monolithHome** (dostępna pod adresem /monolith);
- **microservicesHome** (dostępna pod adresem /microservices);

Każda ze stron, zawiera swoje zestawy slotów w których zostaną umieszczone komponenty.

Strona **monolithHome** zawiera następujące sloty:

- **slot1** – który zostanie zasilony widokiem z serwisu **MONOLITH** dostępnym pod zasobem „/home".

Strona **microservicesHome** zawiera następujące sloty:

- **summary** – który zostanie zasilony widokiem z serwisu **SUMMARY** dostępnym pod zasobem „/home";
- **address** – który zostanie zasilony widokiem z serwisu **ADDRESS** dostępnym pod zasobem „/home";
- **event** – który zostanie zasilony widokiem z serwisu **EVENT** dostępnym pod zasobem „/home";
- **subscription** – który zostanie zasilony widokiem z serwisu **MONOLITH** dostępnym pod zasobem „/home".

Konfiguracją można zarządzać wykorzystując specjalnie przygotowane API obsługujące dwie metody HTTP:

- **GET** /properties - po wykonaniu żądania zostanie zwrócona aktualnie załadowana konfiguracja.
- **POST** /properties - po wykonaniu żądania z nową wersją konfiguracji, aktualna zostanie nadpisana.

Serwer umożliwia definicje szablonów JSP z możliwością wskazania, w którym miejscu zostaną wyświetlone konkretne sloty. Szablon, musi nazywać się tak samo jak nazwa strony. Aby umieścić dany slot we własnym szablonie, należy wykorzystać tag **slotRenderer**, a w argumencie **name** przekazać nazwę slotu do wyświetlenia. W przypadku, gdy nie znaleziono zdefiniowanego szablonu, zostanie użyta wersja **generic** która wyświetli każdy ze slotów pod sobą w kolejności w jakiej zostały zdefiniowane w konfiguracji.

Przykładowe definicje szablonów zostały przedstawione w Listingach 7 i 8 poniżej.

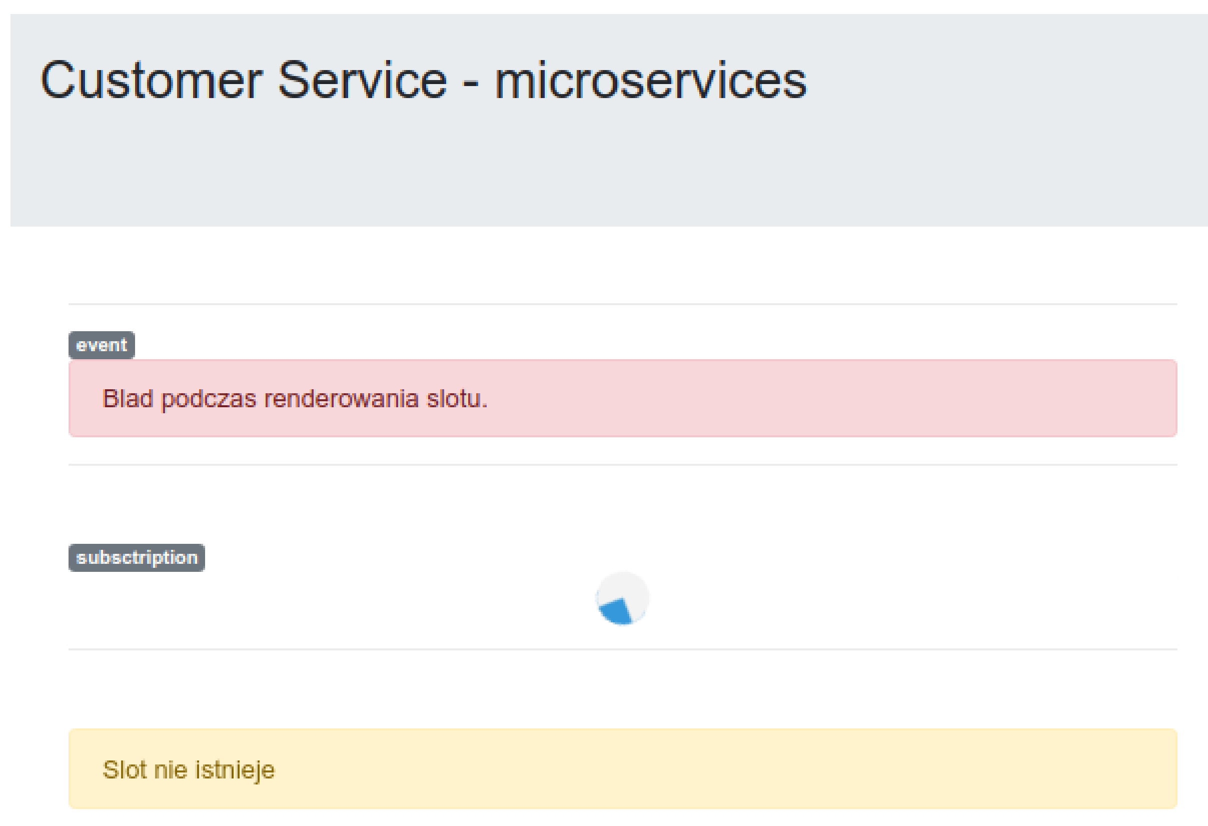
Listing 7. Definicja szablonu monolithHome.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib tagdir="/WEB-INF/tags" prefix="tag" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<tag:header/> <html>
<body>
<div class="jumbotron">
   <h2>${page.name}</h2>
</div>
<div class="container">
   <div class="row">
      <div class="col-md-12">
         <tag:slotRenderer name="slot1" slots="${page.slots}"
queryString="${queryString}"/>
      </div>
   </div>
</div>
</body>
</html> <tag:footer/>
```

Listing 8. Definicja szablonu generic.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib tagdir="/WEB-INF/tags" prefix="tag" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<tag:header/><html>
<body>
<div class="jumbotron">
   <h2>${page.name} – generic template</h2>
</div>
<div class="container">
   <c:forEach var="slot" items="${page.slots}">
      <div class="row">
         <div class="col-md-12">
            <tag:slotRenderer name="${slot.key}" slots="${page.slots}"
queryString="${queryString}"/>
         </div>
      </div>
   </c:forEach>
</div>
</body>
</html><tag:footer/>
```

W momencie, gdy klient (przeglądarka) wywoła żądanie załadowania którejś ze stron, zostanie początkowo zwrócona strona HTML z definicją szablonu. Po stronie przeglądarki, wywoła się kod JavaScript, który rozpocznie asynchroniczne ładowanie slotów. Czynność ta (Rysunek 12), zostanie zasygnalizowana poprzez wyświetlenie loaderów w każdej z sekcji. Każdy slot jest oznaczony nazwą oraz portem zasobu, z którego został załadowany. Są to funkcje deweloperskie, służące do sprawdzenia poprawności generowania widoków oraz działania Równoważenia obciążenia. W celu zapewnienia kontekstowości, każdy z parametrów przekazanych w ramach żądania GET w celu pobrania strony (np. /portal/monolith?**userId**=1&**anyOtherParam**=test) zostanie przekazany do każdego ze slotów.



Rysunek 12. Załadowany szablon z loaderem symbolizującym ładowanie slotu przez przeglądarkę.

W przypadku gdy wystąpią jakiekolwiek problemu z ładowaniem (błędy HTTP klasy 500 i 400), lub slot nie istnieje - serwer pokaże w tym przypadku odpowiednie komunikaty: „Błąd podczas generowania slotu", lub „Slot nie istnieje". Poniższy rysunek (Rysunek 13) przedstawia całkowicie załadowaną stronę, z prawidłowo wczytanym slotem oraz dwoma „problematycznymi".

Customer Service - microservices

event

Blad podczas renderowania slotu.

subsctription 8017

Subskrypcja	Subskrypcja
Roaming UE	Darmowe minuty dla rodziny
Koszt: 10 zl/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca	Koszt: 0 zl/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj	Aktywuj
Nieaktywny	Nieaktywny

Subskrypcja	Subskrypcja
Blokada numerów 0-700	Netflix
Koszt: 0 zl/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca	Koszt: 50 zl/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj	Aktywuj
Nieaktywny	Nieaktywny

Slot nie istnieje

Rysunek 13. Załadowany szablon z prawidłowo załadowanym slotem, nieistniejącym oraz takim, gdzie został zwrócony błąd HTTP 500.

6.2. Rozwiązanie monolityczne

W celu zapewnienia wymagań funkcjonalnych przedstawionych w rozdziale 4.1, zrealizowano prototyp monolitycznej aplikacji webowej. Serwuje on widoki HTML których ładowanie będzie inicjowane przez serwer front-end. W tym przypadku – jest to jeden widok, który zostanie załadowany do pojedynczego slotu.

Aplikacja monolityczna nie zawiera podziału na moduły. Wszystkie funkcjonalności trafiają do jednej paczki .jar, która jest wdrażana na serwerze Tomcat. Diagram przedstawiający architekturę aplikacji został zawarty w Rysunku 14:

Rysunek 14. Architektura aplikacji monolitycznej.

W Listingu 9 przedstawiono kod kontrolera odpowiedzialnego za procesowanie żądania pobrania widoku monolitu. W momencie, gdy żądanie zostanie wysłane, kontroler wyodrębni jego parametry kontekstowe (w tym przypadku **userId**) i użyje ich do pobrania potrzebnych danych (w tym przypadku reprezentacje *Customer*, *TimelineEntries* oraz usług w kontekście użytkownika) za pomocą przygotowanych serwisów. Następnie, do modelu zostaną przekazane te parametry, potrzebne URLe do obsługi akcji modalnych, oraz parametry deweloperskie do celów debuggingu aplikacji (server.port, service). Na końcu, zwracana jest referencja do widoku JSP (slots/home) który wygeneruje odpowiedni HTML zasilony modelem.

Listing 9. Kontroler obsługujący żądanie pobrania widoku dla usługi “monolit”.

```
    @RequestMapping(value = "/view/home", method = {RequestMethod.GET,
RequestMethod.POST})
    public ModelAndView getSummary(ModelAndView modelAndView,
HttpServletRequest httpServletRequest) {
        Map<String, String[]> params = httpServletRequest.getParameterMap();

        String userId = params.get("userId")[0];

        Optional<Customer> customer =
customerRepository.findById(Long.valueOf(userId));
        customer.ifPresent(customer1 -> modelAndView.getModel().put("customer",
customer1));
        customer.ifPresent(customer1 ->
modelAndView.getModel().put("timelineEntries", customer1.getTimelineEntries()));
        customer.ifPresent(customer1 ->
modelAndView.getModel().put("timelineEntriesSize",
customer1.getTimelineEntries().size()));
        customer.ifPresent(customer1 ->
modelAndView.getModel().put("subscriptions",
subscriptionService.getSubscriptions(customer1.getId())));


        String service = httpServletRequest.getHeader("Service");
        modelAndView.getModel().put("ajaxSetMain",
service.substring(0,service.indexOf("?")) + "/setMain");
        modelAndView.getModel().put("ajaxSetActive",
service.substring(0,service.indexOf("?")) + "/setActive");


        modelAndView.getModel().put("params", params);
        modelAndView.setViewName("slots/home");

        modelAndView.addObject("instance", environment.getProperty("server.port"));
        return modelAndView;
    }
```

6.2.1 Warstwa trwałości

W celu zachowania stanu aplikacji, zaprojektowano model bazodanowy używany przez aplikacje w warstwie trwałości – bazie danych H2, która zawiera następujące encje:

1. *Customer* – przechowuje informacje na temat zarejestrowanych klientów systemu:

Nr	Nazwa pola	Typ	Opis
1	*id*	Long	Unikalne id użytkownika
2	*firstName*	String	Imię
3	*lastName*	String	Nazwisko

2. *TimelineEntry* (z relacją do Customer) – przechowuje informacje na temat akcji użytkownika:

Nr	Nazwa pola	Typ	Opis
1	*id*	Long	Unikalne id akcji
2	*title*	String	Tytuł akcji
3	*description*	String	Opis akcji
4	*customer_id*	Long	Id klienta który wykonał akcje

3. *Address* (z relacją do Customer) – przechowuje adresy użytkowników oraz ich typy:

Nr	Nazwa pola	Typ	Opis
1	*id*	Long	Unikalne id adresu
2	*Street*	String	Ulica
3	*postalCode*	String	Kod pocztowy
4	*city*	Long	Miasto
5	*province*	String	Dzielnica
6	*num*	String	Numer domu
7	*main*	Boolean	Flaga mówiąca, czy adres jest adresem głównym dla klienta
8	*customer_id*	Long	Id klienta który wykonał akcje

Funkcjonalnie, wymagania odzwierciadlają pewne obszary biznesowe, które można podzielić na logiczne komponenty:

Nr	Logiczny komponent	Funkcja	Użyte tabele z warstwy trwałości (silnik)
1	Podsumowanie klienta	Wyświetla ogólne dane klienta, jego adres główny oraz ilość wykonanych akcji	*Customer* (H2), *TimelineEntry* (H2), *Address* (H2)
2	Twoje ostatnie akcje	Wyświetla ostatnie akcje klienta na osi czasu	*TimeLineEntry* (H2)
3	Twoje dane adresowe	Wyświetla dane adresowe, pozwala zmienić adres na główny	*Address* (H2)
4	Twoje usługi	Wyświetla aktywne usługi, pozwala aktywować usługę	*Subscription* (JVM)

6.2.2 Warstwa wizualna

Warstwa wizualna komponentów została przedstawiona na Rysunkach 15 - 21.

Podsumowanie klienta:

Podsumowanie klienta

Hubert Chylik

Adres główny:

Władysława Jagiełły 21/51

08-110

Siedlce

Mazowieckie

Ilość akcji:

3

Rysunek 15. Widok komponentu „Podsumowanie Klienta”.

Twoje ostatnie akcje:

Twoje ostatnie akcje

Aktywowano subskrypcje 2019-03-25 14:12:45.035
Darmowe minuty dla rodziny

Zmieniono adres główny 2019-03-25 14:12:32.216
Zmieniono adres główny na Władysława Jagiełły 21/51, 08-110 Siedlce, Mazowieckie

Konto utworzone 2018-11-11 13:23:44.0
Konto utworzone w systemie

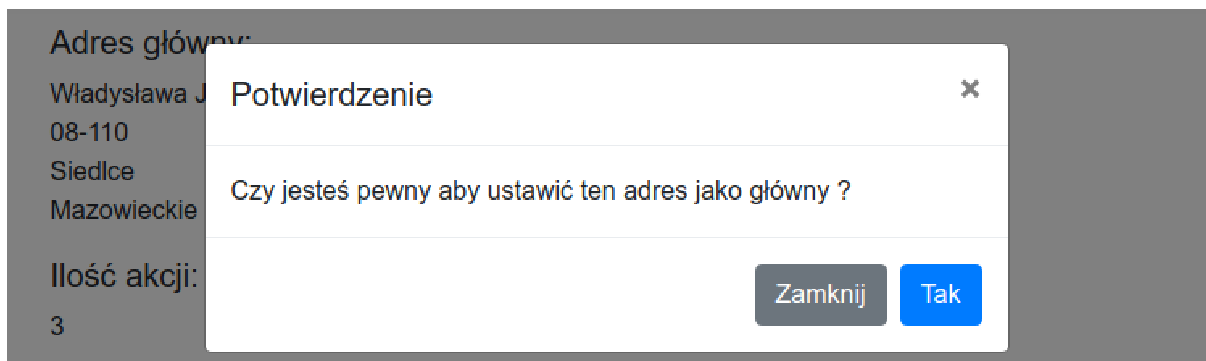
Rysunek 16. Widok komponentu „Twoje ostatnie akcje".

Twoje dane adresowe:

Twoje dane adresowe

#	Ulica	Kod pocztowy	Miasto	Województwo	Główny
0	Górczewska 200c/13	01-460	Warszawa	Mazowieckie	Ustaw jako główny
1	Płocka 30/256	01-460	Warszawa	Mazowieckie	Ustaw jako główny
2	Władysława Jagiełły 21/51	08-110	Siedlce	Mazowieckie	Tak

Rysunek 17. Widok komponentu „Twoje dane adresowe".



Rysunek 18. Widok okna modalnego potwierdzającego zmianę adresu na główny w komponencie „Twoje dane adresowe”

Twoje usługi:

Twoje usługi

Subskrypcja

Roaming UE

Koszt: 10 zl/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca

Aktywuj

Nieaktywny

Subskrypcja

Darmowe minuty dla rodziny

Koszt: 0 zl/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca

Aktywuj

Aktywny od Mon Mar 25 14:12:41 CET 2019

Subskrypcja

Blokada numerów 0-700

Koszt: 0 zl/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca

Aktywuj

Nieaktywny

Subskrypcja

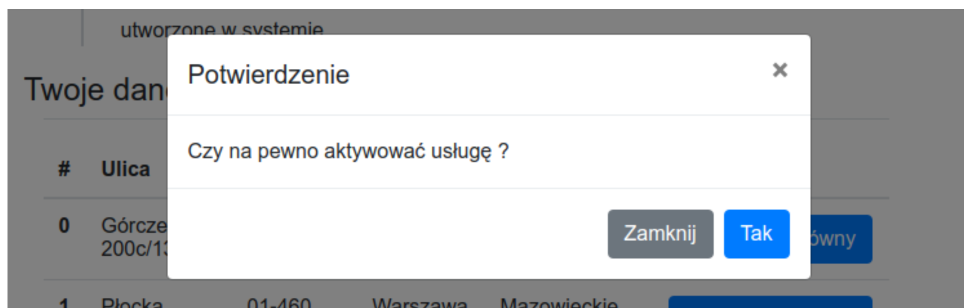
Netflix

Koszt: 50 zl/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca

Aktywuj

Nieaktywny

Rysunek 19. Widok komponentu „Twoje usługi”.



Rysunek 20. Widok okna modalnego potwierdzającego aktywacje usługi w komponencie „Twoje usługi”.

Widok całej strony:

Customer Service - monolith

slot1 8012

Podsumowanie klienta

Hubert Chylik

Adres główny:

Górczewska 200c/13
01-460
Warszawa
Mazowieckie

Ilość akcji:

1

Twoje ostatnie akcje

Konto utworzone
Konto utworzone w systemie 2018-11-11 13:23:44.0

Twoje dane adresowe

#	Ulica	Kod pocztowy	Miasto	Województwo	Główny
0	Płocka 30/256	01-460	Warszawa	Mazowieckie	Ustaw jako główny
1	Górczewska 200c/13	01-460	Warszawa	Mazowieckie	Tak
2	Władysława Jagiełły 21/51	08-110	Siedlce	Mazowieckie	Ustaw jako główny

Twoje usługi

Subskrypcja
Roaming UE
Koszt: 10 zł/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj
Nieaktywny

Subskrypcja
Darmowe minuty dla rodziny
Koszt: 0 zł/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj
Nieaktywny

Subskrypcja
Blokada numerów 0-700
Koszt: 0 zł/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj
Nieaktywny

Subskrypcja
Netflix
Koszt: 50 zł/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj
Nieaktywny

Rysunek 21. Widok podstrony /monolith serwujący wszystkie funkcjonalności biznesowe (komponenty logiczne) w pojedynczym slotcie.

6.3. Rozwiązanie oparte o mikrousługi

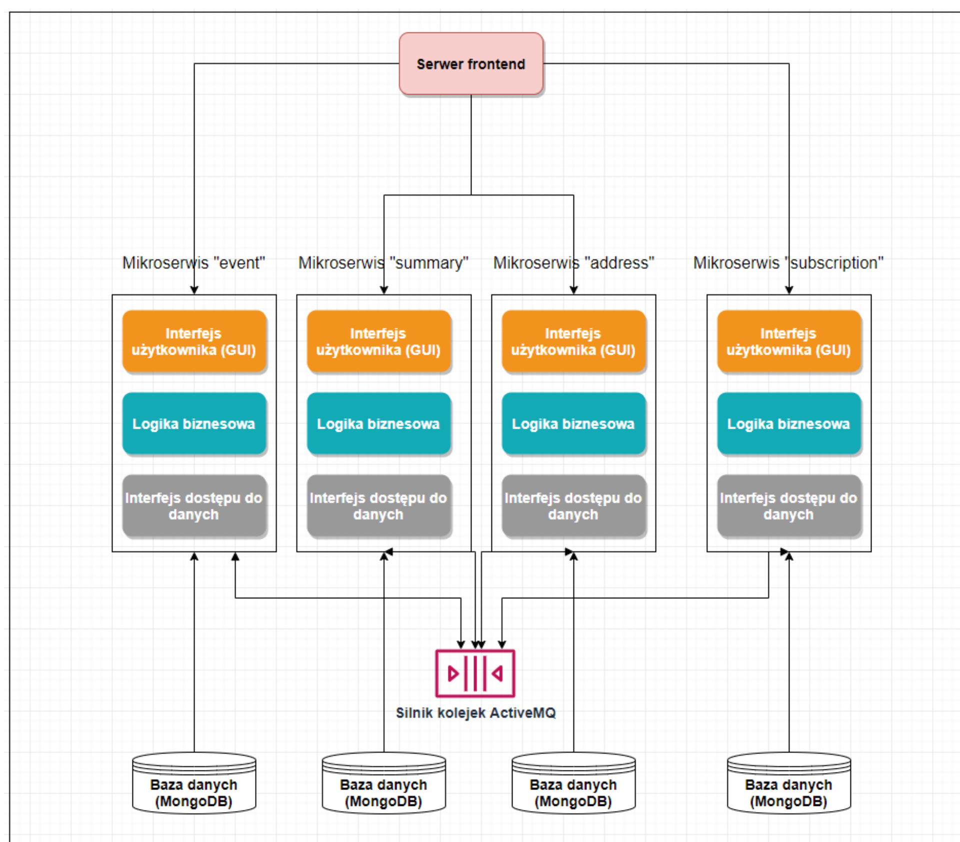
W celu zapewnienia tych samych wymagań funkcjonalnych zrealizowano prototyp aplikacji internetowej w architekturze mikrousług. Każda z mikrousług serwuje widoki HTML których ładowanie będzie inicjowane przez serwer front-end. W tym przypadku – każda mikrousługa to oddzielny slot. Ponadto, zaprojektowano dodatkową mikrousługę (Notification) serwującą informację na temat statusu usług klienta w technologii NodeJS.

Aplikacja w tej architekturze zawiera podział na moduły. Każdy z logicznie wyodrębnionych komponentów ma swoją reprezentacje w postaci mikrousługi. Każda z nich trafia do osobnej paczki .jar, które są wdrażane na serwerze Tomcat.

Wyszczególniono następujące mikrousługi:

- Event („Twoje ostatnie akcje);
- Summary („Podsumowanie klienta);
- Address („Twoje dane adresowe”);
- Subscription („Twoje usługi);
- Notification („Notyfikacje”).

Mikrousługi komunikują się ze sobą za pomocą silnika kolejek ActiveMQ w celu zapewnienia asynchroniczności komunikacji. Diagram przedstawiający architekturę aplikacji został zawarty w Rysunku 22:



Rysunek 22. Ogólna architektura podejścia mikrousługowego.

6.3.1 Warstwa trwałości

W celu zachowania stanu aplikacji, zaprojektowano model bazodanowy używany przez mikrosuługi w warstwie trwałości – bazach danych MongoDB.

Mikrousługa „summary”

1. *Customer* – przechowuje informacje na temat zarejestrowanych klientów systemu:

Nr	Nazwa pola	Typ	Opis
1	*id*	String (ObjectId)	Unikalne id użytkownika w konwencji MongoDB (np. 507f191e810c19729de860ea)
2	*businessId*	Long	Unikalne id użytkownika (liczbowe)
3	*firstName*	String	Imię
4	*lastName*	String	Nazwisko

2. *SummaryReadModel* – przechowuje zbiorcze informacje na temat widoku podsumowania klienta w kontekście danego klienta:

Nr	Nazwa pola	Typ	Opis
1	*id*	Long	Unikalne id akcji
2	*customerId*	String	Id klienta dla którego wygenerowano dokument
3	*numberOfTimelineEntries*	String	Liczba wykonanych akcji przez kleinta
4	*newAddress*	String	Aktualny adres główny klienta

Mikrousługa „address”

1. Address – przechowuje informacje na temat adresów użytkownika:

Nr	Nazwa pola	Typ	Opis
1	*id*	String (ObjectId)	Unikalne id adresu w konwencji MongoDB (np. 507f191e810c19729de860ea)
2	*Street*	String	Ulica
3	*postalCode*	String	Kod pocztowy
4	*city*	Long	Miasto
5	*province*	String	Dzielnica
6	*num*	String	Numer domu
7	*main*	Boolean	Flaga mówiąca, czy adres jest adresem głównym dla klienta
8	*customer_id*	Long	Id klienta który wykonał akcje

Mikrousługa „event”

1. Event – przechowuje informacje na temat akcji wykonanych przez użytkownika:

Nr	Nazwa pola	Typ	Opis
1	*id*	String (ObjectId)	Unikalne id adresu w konwencji MongoDB (np. 507f191e810c19729de860ea)
2	*title*	String	Tytuł akcji
3	*description*	String	Opis akcji
4	*customer_id*	Long	Id klienta który wykonał akcje

Każda z usług posiada oddzielny schemat, odseparowany od pozostałych. Wymagało to zaprojektowania modelu w taki sposób, aby usługi posiadały w swoich schematach komplet danych do wygenerowania potrzebnych informacji. W większości przypadków, schematy relacyjne do nierelacyjnych są odwzorowane 1:1. Wyjątkiem jest usługa „Podsumowanie klienta”, której logika jest zasilana danymi z oddzielnej kolekcji – *SummaryReadModel* posiadającej lokalną kopię danych z innych kolekcji (w przypadku monolitu, analogiczna tabela nie istnieje). Jest to ściśle związane z koncepcją „Read Model”.

Koncepcja „Read Model” [28] to styl projektowania baz danych, którego celem jest minimalizacja ilość relacji i złączeń (ang. *joins*) w modelu bazodanowym, nawet kosztem redundancji danych. W przypadku monolitu, aby wygenerować widok podsumowania klienta należy pobrać dane o kliencie, jego adresach oraz wykonanych akcjach. Wiąże się to w praktyce z trzema żądaniami do bazy danych oraz wykonania złączeń, które są obciążające w kontekście wydajności. Warto wspomnieć, że kosztowne łączenia będą wykonywane przy każdym odświeżeniu strony. Mamy dzięki temu pewność, że dane zawsze są aktualne.

„Read model” dla rozwiązania w kontekście mikrousług, zakłada działanie na kopii danych na potrzeby tylko do odczytu. Zamiast pobierać za każdym razem informacje z odpowiednich encji (w tym przypadku, konieczny byłoby żądanie sieciowe do innych mikrousług), system posiada ich lokalną kopię, w optymalnym dla siebie schemacie (brak złączeń). Aktualizacja tych danych odbywa się asynchronicznie. W przypadku, gdy w jakimkolwiek komponencie zostanie wyzwolona akcja wymagająca aktualizacji read-modelu, nastąpi emisja odpowiedniego zdarzenia na kolejkę ActiveMQ. Mikroserwis odpowiedzialny za daną kolekcję w czasie, gdy będzie miał odpowiednie zasoby zaktualizuje swoją lokalną kopię.

Proces ten, można w prosty sposób opisać przykładem komunikacji między mikrousługami **address** oraz **summary**. W momencie, gdy usługa **summary** zostanie uruchomiona, subskrybuje się na kolejkę **summary**. Za każdym razem, gdy zostanie przesłany do niej zdarzenie typu „**mainAddressChange**” lub „**newEntry**”, zaktualizuje dokument w kolekcji *Summary Read Model* dla odpowiedniego użytkownika. W przypadku, gdy dokument nie istnieje (użytkownik loguje się do systemu pierwszy raz) – zostanie utworzony nowy. Konsekwencją takiej architektury jest brak konieczności wystosowania więcej niż jednego żądania do bazy danych (jedno żądanie do *SummaryReadModel*, zamiast trzy do *Customer*, *Address*, *TimelineEntry*). Istnieje również ryzyko chwilowej niespójności danych. Jeśli podczas ładowania slotów zostanie zmieniony adres główny klienta, możliwe jest, że komponent podsumowania klienta nie zostanie o tym poinformowany w tej konkretnej chwili i dostarczy nieaktualne dane. Są to opóźnienia na poziomie mikrosekund, jednak jak najbardziej możliwe do wystąpienia.

6.3.2 Notification – dodatkowy komponent w ramach mikrousługi w innej technologii

W celu zaprezentowania jednej z zalet podejścia mikrousług została przygotowana dodatkowa usługa **notification** w technologii NodeJS. Warto wspomnieć, iż jest to technologia zupełnie inna niż Java, wymagająca innego serwera aplikacyjnego i innych umiejętności deweloperskich. Dzięki temu podejściu, można bez problem zintegrować taką usługę z istniejącym rozwiązaniem. Nie ma ona żadnego wpływu na resztę aplikacji. Poniżej przedstawiono zrzut ekranu slotu (Rysunek 23) który jest udostępniany przez tę usługę oraz jej kod źródłowy (Listing 10).

Rysunek 23. Widok komponentu „Notyfikacje".

Listing 10. Kod kontrolera odpowiadającego za załadowanie widoku usługi „Notification" napisany w NodeJS.

```
'use strict';

const express = require('express');
const app = express();
const port = 3000;
const querystring = require('querystring');
const request = require('sync-request');
const format = require("string-template");
var bodyParser    =       require("body-parser");

let headers = {
   'Content-Type': 'application/x-www-form-urlencoded',
   'accept': 'application/json'
};

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

app.get('/view/home', (req, res) => {

   let response = `<div class=\"alert alert-success\" role=\"alert\"\>Wszystkie usługi
dla userId = <b>${req.query.userId}</b> działają prawidłowo</div>`;
   res.send(response);
   }
);

app.listen(port, () => console.log(`Service Notification listening on port ${port}!`))
```

6.3.3 Warstwa wizualna

Komponenty logiczne wyglądają analogicznie do rozwiązania monolitycznego, z tą różnicą, że są serwowane z oddzielnych usług (różne instancje serwerowe), więc każdy z nich umieszczony jest w oddzielnym slocie (Rysunek 24).

Customer Service - microservices

status

Wszystkie usługi dla userId = 1 działają prawidłowo

summary 8013

Hubert Chylik

Adres główny:

Płocka 30/256, 01-460 Warszawa, Mazowieckie

Ilość akcji:

2

address 8016

#	Ulica	Kod pocztowy	Miasto	Województwo	Główny
0	Górczewska 200c/13	01-460	Warszawa	Mazowieckie	Ustaw jako główny
1	Władysława Jagiełły 21/51	08-110	Siedlce	Mazowieckie	Ustaw jako główny
2	Płocka 30/256	01-460	Warszawa	Mazowieckie	Tak

event 8015

Konto utworzone 2018-11-11 13:23:44.0
Konto utworzone w systemie

Zmieniono adres główny 2019-04-01 15:57:33.152
Zmieniono adres główny na Płocka 30/256, 01-460 Warszawa, Mazowieckie

subscription 8017

Subskrypcja
Roaming UE
Koszt: 10 zł/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj
Aktywny od Mon Apr 01 15:51:13 CEST 2019

Subskrypcja
Darmowe minuty dla rodziny
Koszt: 0 zł/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj
Nieaktywny

Subskrypcja
Blokada numerów 0-700
Koszt: 0 zł/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj
Nieaktywny

Subskrypcja
Netflix
Koszt: 50 zł/msc. Okres rozliczeniowy rozpoczyna się pierwszego dnia miesiąca
Aktywuj
Aktywny od Mon Apr 01 15:55:50 CEST 2019

Rysunek 24. Widok podstrony /microservices serwujący wszystkie funkcjonalności biznesowe (komponenty logiczne) w oddzielnych slotach.

6.3.4 Równoważenie obciążenia

Aby zaprezentować jedną z zalet podejścia mikrousług, założono, że klienci będą najczęściej korzystać z funkcjonalności włączania/wyłączania usług. Jest to częsta sytuacja, gdy dany fragment systemu jest bardziej eksploatowany niż inne. W związku z tym, usługa, która za to odpowiada została zeskalowana do dwóch instancji serwerowych. Z racji tego, że zrealizowane komponenty są bezstanowe, można to zrobić w prosty sposób korzystając z równoważenia obciążenia (*Load balancing* [26]). W tym przypadku, wykorzystano serwer nginx i algorytm *round-robin* (konfiguracja rozwiązania została zawarta w Listingu 11).

Listing 11. Konfiguracja serwera nginx na potrzeby funkcjonalności równoważenia obciążenia.

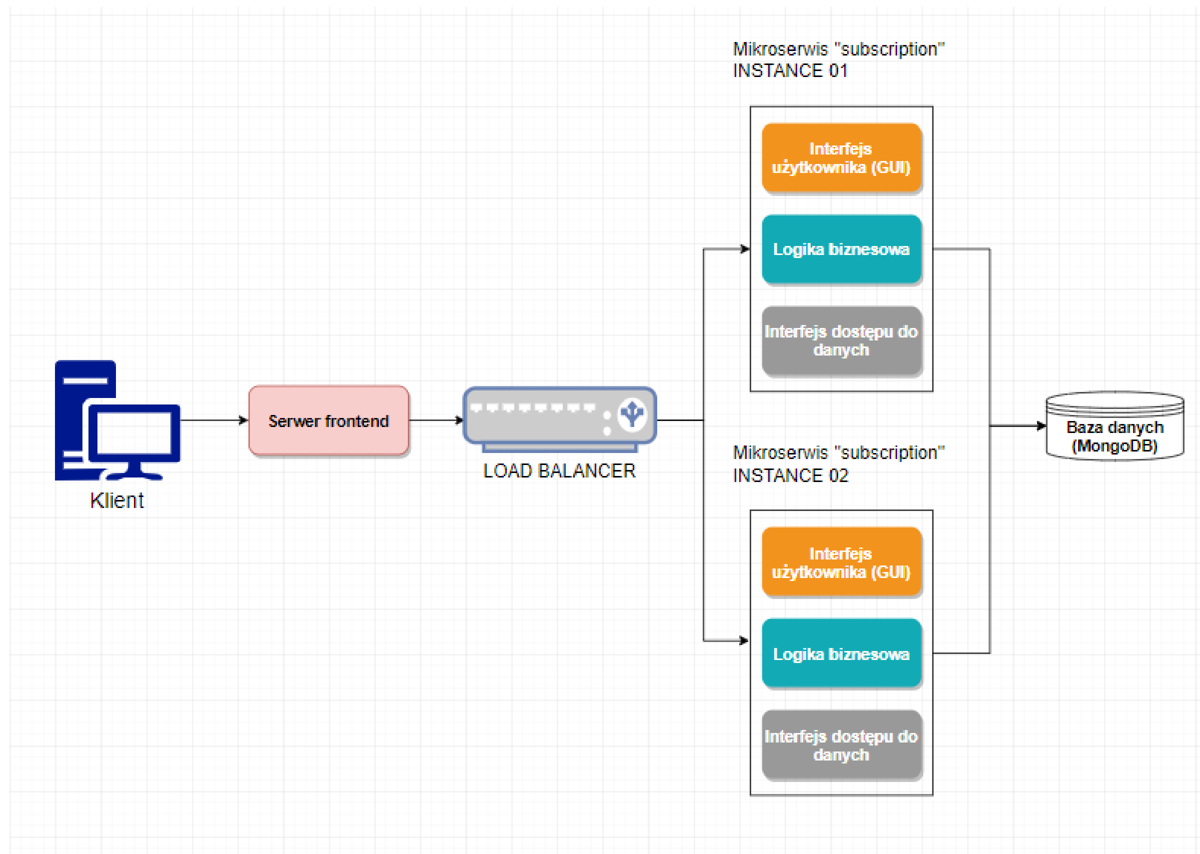
```
include       mime.types;
   default_type  application/octet-stream;

upstream subs {
      server localhost:8017;
      server localhost:8019;
   }

   server {
         access_log /var/log/monolith.access.log;
         error_log /var/log/monolith.error.log;
      listen 8014;
         server_name localhost;

      location /subs {
         proxy_pass http://subs/;
      }
   }
```

Serwer Front-end nie komunikuję się bezpośrednio z mikrousługą, tylko z równoważnikiem obciążenia który decyduje do której instancji przekierować żądanie. W przypadku podejścia round-robin, ruch będzie rozłożony równomiernie. Architekturę tego podejścia przedstawiono na Rysunku 25.



Rysunek 25. Architektura podejścia mikrousług w kontekście równoważenia obciążenia.

7. Możliwości rozwoju

Zrealizowane aplikacje są tylko prototypami. Na poziomie implementacji wykryto następujące, potencjalne możliwości rozwoju takie jak:

- Zrealizowanie warstwy prezentacji w nowszych technologiach (React/Angular zamiast JSP);
- Wdrożenie orkiestracji;
- Rezygnacja z równoważenia obciążenia na rzecz service discovery. Na ten moment mikrousługi nie mają świadomości o istnieniu więcej niż jednej instancja każdej z nich. Świadomość tą ma równoważnik. Rozwiązanie to można zastąpić automatycznym wykrywaniem innych usług poprzez rejestracje w oddzielnym systemie;
- Utrzymywanie konfiguracji aplikacji na oddzielnych serwerach zamiast w plikach konfiguracyjnych. Pozwoli to na współdzielenie jej między usługami.

8. Podsumowanie

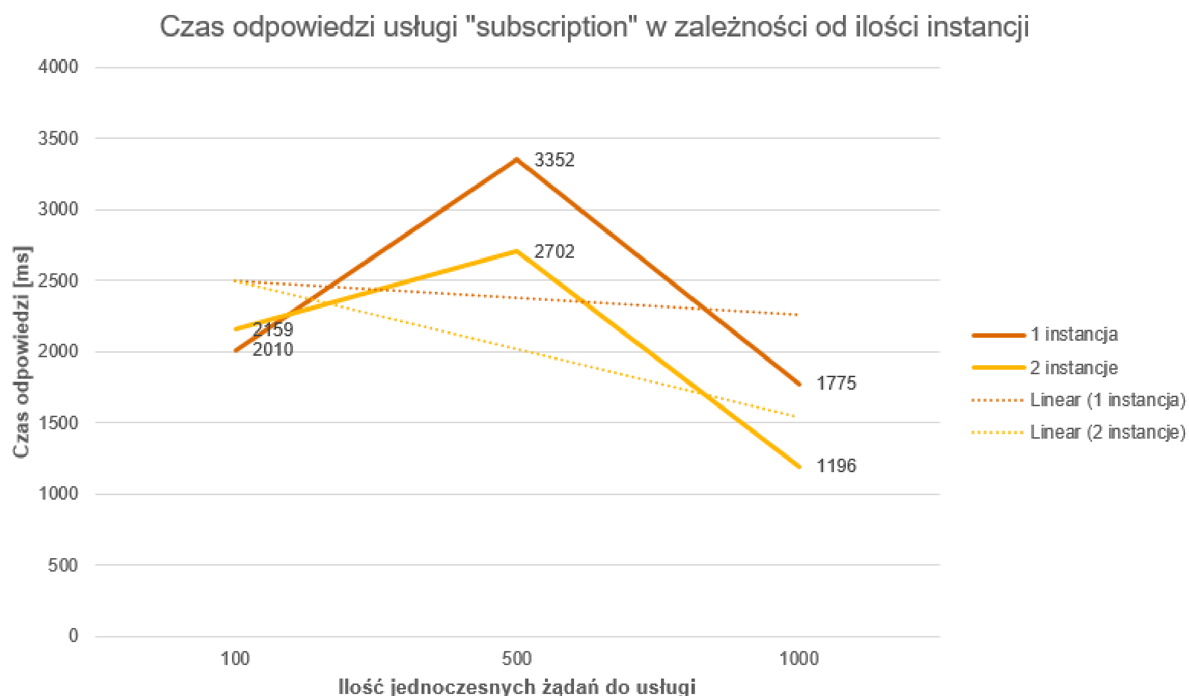
Analizując implementacje zrealizowanych prototypów można stwierdzić, że w obu przypadkach wymagania jakościowe zostały spełnione.

Nr	Wymaganie jakościowe	Monolit	Mikrousługi
1	Wysoka dostępność systemu – system powinien być przygotowany na duże ilości jednoczesnych żądań.	Tak – w przypadku przeskalowania całej instancji.	Tak – możliwość skalowania konkretnych usług w zależności od zapotrzebowania
2	System powinien być zaimplementowany w wiodących, sprawdzonych technologiach w świecie technologii webowych.	Tak	Tak
3	System powinien być dostępny dla użytkownika z poziomu przeglądarki Internetowej.	Tak	Tak

Celem pracy była ocena słuszności dekompozycji monolitycznych systemów klasy CRM na mikrousługi. W tym przypadku może wydawać się, że nie ma sensu podejmować próby transformacji, skoro w obu rozwiązaniach możemy dojść do tych samych rezultatów.

Warto jednak spojrzeć na wymagania jakościowe szerzej, z innej perspektywy. W przypadku monolitu istnieje możliwość zapewnienia wysokiej dostępności systemu, jednak tylko w przypadku skalowania całej aplikacji. Jest to maksymalnie niewydajne oraz wymaga dużych zasobów. Zakładając, że tylko część aplikacji jest poddana dużemu obciążeniu – ciągle jedyną odpowiedzią na ten problem jest uruchomienie kolejnej instancji całości. W konsekwencji, serwer powołuje do życia wszystkie komponenty techniczne – nawet te, które nie będą wykorzystywane. Dodatkowo, jeśli monolit zachowuje stan systemu w pamięci RAM (z reguły tak jest), wymagane jest zapewnienie mechanizmu *sticky-session*. Jeśli użytkownik za pierwszym żądaniem trafił na jedną z instancji monolitycznych (która trzyma stan użytkownika) – każde kolejne żądanie musi trafić do tej konkretnej instancji. Rodzi to wiele komplikacji. W przypadku gdy ta instancja ulegnie awarii – sesja może zostać utracona. Istnieją rozwiązania polegające na replikacje sesji, jednak wymagany jest do tego oddzielna infrastruktura i jest to skomplikowany proces – zdecydowanie nie rekomendowany. Na pewno nie jest to rozwiązanie problemu, tylko proteza. W przypadku podejścia mikrousług, powyższe problemy nie występują. Mikrousługi same z siebie nie trzymają stanu aplikacji (wykorzystywane są kolejki) więc możemy je skalować. Dodatkowo, skalowaniu ulegają tylko te fragmenty systemu które muszą być zeskalowane. Nie ma więc niepotrzebnej redundancji i tak nieużywanych komponentów.

Aby sprawdzić zachowanie fragmentu systemu w warunkach przypominających produkcyjne, przeprowadzono testy wydajnościowe polegające na pomiarze czasu odpowiedzi usługi **subscription** z zależności od ilości instancji oraz ilości jednoczesnych żądań. Symuluje to duże obciążenie konkretnej funkcjonalności w danym momencie czasu. Dwie konfiguracje (pojedyncza instancja oraz dwie instancje) poddano próbie 100/500/1000 żądań jednocześnie. Wyniki w postaci wykresu przedstawiono na Rysunku 26.



Rysunek 26. Wykres przedstawiający czasy odpowiedzi usługi „subscription" w zależności od ilości instancji.

Widać wyraźnie, że w przypadku 100 żądań, nie mamy uzysku z wykorzystania skalowalnych mikrousług. Uzyskujemy lepszy (mniejszy) wyniki dla 1 instancji, ponieważ ilość żądań jest na tyle mała, że pojedyncza usługa jest w stanie sobie z nią poradzić w rozsądnym czasie. Dla 2 instancji występuje opóźnienie na poziomie równoważnika obciążenia, który dodaje dodatkowy narzut ponieważ musi zdecydować do której instancji przekierować żądanie. Sytuacja zmienia się, gdy zwiększymy ilość jednoczesnych żądań do 500, a potem do 1000. Wtedy, pojedyncza usługa wyraźnie nie radzi sobie z takim ruchem. Zastosowanie kilku instancji (dwóch) powoduje zdecydowane przyśpieszenie czasu odpowiedzi na żądania, mimo zastosowania rozłożenia ruchu. Ilość instancji można dalej zwiększać, aby uzyskać jeszcze korzystniejsze wyniki. Na Rysunku 27 przedstawiono rezultat jednego ze scenariuszy testowych testu wykonanego w aplikacji JMeter. Poza średnią, która została użyta w porównaniach program liczy też inne parametry, takie jak: min, max czy ilość odebranych danych.

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename | Browse... | Log/Display Only: Errors Successes | Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
subscription	100	2010	2003	2015	2.66	0.00%	33.1/sec	213.15	5.49	6596.0
TOTAL	100	2010	2003	2015	2.66	0.00%	33.1/sec	213.15	5.49	6596.0

Rysunek 27. Rezultat wykonania jednego ze scenariuszy testowych. W tym przypadku próbki 100 żądań dla pojedynczej instancji usługi.

Uzyskane wyniki podczas testów oraz doświadczenie zebrane w trakcie implementacji i projektowania prototypów nie zostawiają cienia wątpliwości w kontekście porównania tytułowych architektur. Podejście do mikrousług jest zdecydowanie lepszym rozwiązaniem z perspektywy technicznej i utrzymaniowej.

Mikrousługi wyróżniają większą elastycznością, jeśli chodzi o wybór technologii. Projektowane są fragmenty, więc mogą być wdrażane oddzielnie i niezależnie od wybranego języka programowania. Ważne, aby został zdefiniowany jasny interfejs komunikacyjny. W teorii, jest to jedyny kontrakt jaki usługi muszą spełnić, aby mogły ze sobą „rozmawiać”

Podejście mikrousług posiada również wady w zestawieniu z monolitem. Mikrousługi są zdecydowanie droższe w utrzymaniu, szczególnie na początku implementacji i wdrożenia. Zakładając, że każdy fragment jest wydzielony – istnieje potrzeba zarzadzania zaawansowanym cyklem wydawniczym. Monolit jest wdrażany jako całość – mikrousługi oddzielnie. W związku z tym, cały proces wdrożenia, testów oraz utrzymania musi zostać skoordynowany. Wymagany jest też system orkiestracji – platforma, która będzie zarządzała instancjami serwerowymi, uruchamiała proces skalowania w przypadku potrzeby i raportowania awarii.

Może się więc okazać, że wdrożenie mikrousług w początkowym etapie będzie droższe. Bardziej skomplikowana architektura wymaga większej ilości wyspecjalizowanych deweloperów. Podobnie ma się kwestia technologii. To, co wydaje się być zaletą (każda mikrousługa może być napisana w innym języku programowania) z perspektywy kosztów jest wadą. Mogą być potrzebni deweloperzy z wielu obszarów, a nie tylko jednego.

W przypadku wdrażania nowego projektu – system od razu powinien być projektowany i implementowany w tej architekturze. Dużo trudniej jest przepisywać istniejącą architekturę na nowe podejście.

W przypadku transformacji istniejącego systemu – należy podjąć próbę kalkulacji jego obciążenia. Jeśli istnieje potrzeba, aby system był wysokodostępny i odporny na masową ilość żądań – nie ma innej możliwości niż wybór mikrousług i konwersja. Zaprocentuje to w przyszłości, przy wdrażaniu kolejnych klientów i nowych funkcjonalności. Natomiast, jeśli biznes nie widzi perspektywy na rosnące zapotrzebowanie funkcjonalne/jakościowe (proste aplikacje wewnętrzne) – monolit może być wystarczającym rozwiązaniem i zdecydowanie tańszym.

Prace cytowane

[1]. *Czym jest system CRM? – Oprogramowanie CRM dla firm.* https://www.gonetcrm.pl/crm, Dostęp: Marzec 2019

[2]. *SugarCrm, Product.* https://www.sugarcrm.com/product, Dostęp: Kwiecień 2019

[3]. *Microsoft Dynamics CRM.* https://dynamics.microsoft.com/pl-pl/crm/crm-software/, Dostęp: Kwiecień 2019

[4]. *Salesforce.com – products.* https://www.salesforce.com/eu/products/, Dostęp: Kwiecień 2019

[5]. *Salesforce vs Microsoft Dynamics vs SugarCRM: Which CRM is the Winner in 2019?* https://selecthub.com/customer-relationship-management/salesforce-com-vs-microsoft-dynamics-vs-sugarcrm/, Dostęp: Maj 2019

[6]. *SugarCRM Polska – Sugar Sales.* https://sugarcrm.com.pl/sugarcrm/sugar-sales/, Dostęp: Maj 2019

[7]. *Microsoft Dynamics CRM hotfix for Chrome users.* http://dynamicsfeed.com/tips/microsoft-dynamics-crm-hotfix-chrome-users/, Dostęp: Maj 2019

[8]. *8x8 Virtual Contact Center for Salesforce.* https://www.8x8.com/call-center/features/contact-center-salesforce-integration, Dostęp: Maj 2019

[9]. *Interfejs programowania aplikacji.* https://pl.wikipedia.org/wiki/Interfejs_programowania_aplikacji, Dostęp: Kwiecień 2019

[10]. *Redundancja.* https://pl.wikipedia.org/wiki/Redundancja, Dostęp: Kwiecień 2019

[11]. *Mikrousługi vs monolit. Nie zawsze architektura oparta o mikroserwisy to najlepszy pomysł.* https://www.cloudforum.pl/2018/08/10/mikrouslugi-vs-monolit-nie-zawsze-architektura-oparta-o-mikroserwisy-to-najlepszy-pomysl/, Dostęp: Marzec 2019

[12]. *Zintegrowane środowisko programistyczne.* https://pl.wikipedia.org/wiki/Zintegrowane_%C5%9Brodowisko_programistyczne, Dostęp: Kwiecień 2019

[13]. *IntelliJ IDEA.* https://pl.wikipedia.org/wiki/IntelliJ_IDEA, Kwiecień 2019

[14]. *Javascript.* https://developer.mozilla.org/pl/docs/Learn/JavaScript/Pierwsze_kroki/What_is_JavaScript, Dostęp: Kwiecień 2019

[15]. *Dokumentacja NodeJS.* https://nodejs.org/en/docs/, Dostęp: Marzec 2019

[16]. *Oracle Certified Professional Java SE 8 Programmer II Study Guide*. Jeanne Boyarsky, Scott Selikoff. Sybex. 2015

[17]. *Pivotal Certified Professional Spring Developer Exam*. Cosmina Iuliana. Apress. 2016

[18]. *HTML*. https://developer.mozilla.org/pl/docs/Learn/Getting_started_with_the_web/HTML_basics, Dostęp: Kwiecień 2019

[19]. *CSS*. https://www.w3schools.com/css/, Dostęp: Kwiecień 2019

[20]. *Bootstrap*. https://getbootstrap.com/, Dostęp: Kwiecień 2019

[21]. *MVC*. http://vavatech.pl/technologie/architektura/mvc, Dostęp: Kwiecień 2019

[22]. *MVC, Delivery Mechanism and Domain Model*. https://www.javacodegeeks.com/2017/09/mvc-delivery-mechanism-domain-model.html, Dostęp: Maj 2019

[23]. *Wstęp do REST API*. https://devszczepaniak.pl/wstep-do-rest-api/, Dostęp: Marzec 2019

[24]. *JMS*. http://stencel.mimuw.edu.pl/abwi/20020430.b.JMS/, Dostęp: Kwiecień 2019

[25]. *Messaging with JMS*. https://spring.io/guides/gs/messaging-jms/, Dostęp: Marzec 2019

[26]. *Równoważenie obciążenia*. https://pl.wikipedia.org/wiki/R%C3%B3wnowa%C5%BCenie_obci%C4%85%C5%BCenia, Dostęp: Marzec 2019

[27]. *Dokumentacja JMeter*. https://jmeter.apache.org/usermanual/get-started.html, Dostęp: Grudzień 2018

[28]. *Pattern: Command Query Responsibility Segregation (CQRS)*. https://microservices.io/patterns/data/cqrs.html, Dostęp: Kwiecień 2019