POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Wojciech Filipowicz
Nr albumu s24026

Analiza systemów opartych o oprogramowanie wielodostępne

Praca magisterska

Dr inż. Mariusz Trzaska

Warszawa, Styczeń, 2023

Streszczenie

Praca magisterska dotyczy analizy systemów opartych o oprogramowanie wielodostępne (ang. *Multitenancy*). Autor pracy bada różne sposoby przechowywania oraz izolacji danych. Skupia się również na architekturze, warstwie danych oraz infrastrukturze systemów. Wskazuje ich wady oraz zalety. Dostarcza również sugestie w jakich przypadkach dane rozwiązanie powinno zostać zaimplementowane. Eksperymenty przeprowadzane są na przykładowym projekcie systemu sprzedażowego dla wielu firm. System korzysta z generycznej biblioteki Multitenancy stworzonej na potrzeby pracy magisterskiej. Praca zawiera również rozdziały dokładnie opisujące wspomnianą bibliotekę oraz jej implementacje w analizowanym projekcie. Autor opisuje interfejsy, klasy oraz serwisy niezbędne do poprawnego działania programu. Ukazuje on również jak w kliku krokach dodać bibliotekę do własnego systemu. Rozdział opisujący ten aspekt zawiera również propozycje architektury z użyciem wielu przydatnych wzorców projektowych. Implementowane systemy zgodne są z zasadami czystego kodu (ang. *Clean code*), czystej architektury (ang. *Clean architecture*) oraz są projektowane w oparciu o domenę (ang. *Domain-driven design*). Autor porusza również bardzo ważne kwestie migracji oraz analizy danych. Jeden z rozdziałów w pełni poświęcony jest budowie hurtowni jako jednego z rozwiązań problemu integracji danych. Przedstawione są w nim również praktyczne przykłady użycia.

Słowa kluczowe:

Oprogramowanie wielodostępne, system, architektura, multitenancy, tenant

Spis treści

1. WSTĘP 5

1.1. Motywacja 5
1.2. Kontekst pracy 5
1.3. Cel pracy 5
1.4. Zakres pracy 6
1.5. Technologie 6
1.6. Rezultaty pracy 7
1.7. Organizacja pracy 7

2. OPIS PROBLEMU 8

2.1. Architektura wielodostępna 9
2.2. Dane dzierżawców 10

3. SPOSOBY REALIZACJI 11

3.1. Klasyczne podejście – jedna baza 11
3.2. Osobna baza dla każdej firmy 13
3.3. Osobny schemat bazy dla każdej firmy 17

4. BAZA DANYCH NA SERWERZE FIRMY 19

4.1. Korzyści rozwiązania 19
4.2. Minusy rozwiązania 19
4.3. Integracja danych – Hurtownia danych 20
4.3.1 Diagram hurtowni danych 20
4.3.2 Procesy ETL 21
4.3.3 Kostka OLAP 25
4.3.4 Raporty 26
4.3.5 Podsumowanie 29

5. OPIS AUTORSKIEJ BIBLIOTEKI MULTITENANCY 30

5.1. Opis 30
5.2. Architektura 31
5.2.1 ITenant 32
5.2.2 MultitenancyType 33
5.2.3 MultitenancyContextBase 33
5.2.4 Tenant Service 35
5.2.5 ITenant Storage 36
5.2.6 TenantMiddleware 36
5.2.7 TenantFilter 37
5.2.8 ITenantResolutionStartegy (Identyfikacja dzierżawcy) 38
5.3. Autoryzacja 40
5.4. Implementacja w projekcie 41
5.4.1 Architektura 41
5.4.2 Wzorce projektowe 42
5.4.3 Tryb Multitenancy 45
5.4.4 Użycie w projekcie 45
5.4.5 Migracje 46

6. ARCHITEKTURA WIELODOSTĘPNA W SYSTEMIE ROZPROSZONYM 48

6.1. Wielu dzierżawców 49

7. PODSUMOWANIE 51

PRACE CYTOWANE ORAZ ŹRÓDŁA WIEDZY 54

DODATKI **55**
Spis rysunków 55
Spis listingów 56

1. Wstęp

Oprogramowanie wielodostępne (ang. *Multitenancy*) to coraz bardziej popularny temat w świecie IT. Termin ten oznacza system obsługujący wielu dzierżawców (ang. *Tenants*) przy pomocy pojedynczej instancji serwera. Niestety nie ma zbyt wielu artykułów, książek oraz innej literatury przedstawiającej ten temat wraz z opisem poszczególnych rozwiązań, wynikających z nich korzyści oraz problemów z jakimi programista musi się zmierzyć. Autor skupia się więc na tym temacie w pracy magisterskiej.

1.1. Motywacja

Motywacją do podjęcia tego tematu jest chęć rozwoju w kierunku projektowania architektury systemów informatycznych oraz obowiązujących obecnie trendów. System jest reprezentowany przez backend stworzony w technologii C# .NET - jest to monolit z możliwym użyciem dodatkowych serwisów, służących do skalowania najbardziej obciążonych elementów systemu. Temat pracy wynika z doświadczeń zawodowych oraz zainteresowań. Przy wyborze tematu autor kierował się również dalszymi perspektywami zawodowymi. Projektował on wiele prostych systemów z architekturą monolit. Ostatnio mierzył się z architekturą wielodostępną. Okazało się, że istnieje znaczna ilość sposobów, w jaki można podejść do tego tematu. W pracy magisterskiej przeanalizowane zostały poszczególne rozwiązania. Podjęty został również temat skalowalności takiego systemu.

1.2. Kontekst pracy

Praca skupia się na analizie rożnych realizacji systemu opartego o oprogramowanie wielodostępne. Projekt przedstawiony jest na przykładzie systemu sprzedażowego, gdzie wiele firm może posiadać swoje sklepy oraz sprzedawać swoje produkty. W projekcie przedstawione jest również wykorzystanie autorskiej biblioteki Multitenancy. Dzięki niej programista jest w stanie zaimplementować oprogramowanie wielodostępne do swojego projektu przy pomocy kilku linijek kodu. Biblioteka jest bardzo generyczna i łatwo przenaszalna, dzięki czemu sprawdzi się również w systemach rozproszonych.

1.3. Cel pracy

Celem pracy jest analiza systemu opartego o oprogramowanie wielodostępne badając przy tym korzyści oraz wady poszczególnych sposobów realizacji. Na potrzeby badań powstał również projekt techniczny. Praca może pomóc każdemu, kto będzie podejmował się tworzenia podobnego systemu w przyszłości. Obecnie ciężko znaleźć dokładne porównania poszczególnych rozwiązań. Realizacja tego tematu zdecydowanie ułatwi wybór podejścia do projektowania systemu.

Z wielu względów, jak np. zmniejszenie kosztów, programiści decydują się na stworzenie jednego systemu obsługującego wiele firm. Tworzenie osobnych instancji serwera dla każdego dzierżawcy systemy (ang. *Tenant*) byłoby bardzo uciążliwe oraz kosztowne. Zdecydowanie praca znajdzie zastosowanie w wielu miejscach, gdyż może być wykorzystana praktycznie w każdej firmie zajmującej się projektowaniem systemów informatycznych. Przynosi ona wiele korzyści wynikających z braku konieczności analizy poszczególnych rozwiązań oraz potencjalnego przepisywania projektu w przypadku napotkania nieprzewidzianych konsekwencji w zaawansowanej

fazie rozwoju projektu. Wiąże się to z dużą oszczędnością pieniędzy oraz czasu. Praca ułatwi tworzenie i projektowanie systemów opartych o oprogramowanie wielodostępne. W dodatku dokument nie wymaga znajomości konkretnej technologii i jest uniwersalnym źródłem wiedzy. Dzięki temu programiści niezależnie od używanych narzędzi mogą czerpać z niego korzyści.

1.4. Zakres pracy

W ramach pracy zostały przeprowadzone badania oraz analiza poszczególnych rozwiązań problemu architektury wielodostępnej. W celu ułatwienia analizy została stworzona również autorska biblioteka Multitenancy. Pozwala ona na dodanie wielodostępności do projektu w kilku prostych krokach. Biblioteka jest generyczna oraz łatwo przenaszalna. Powstał również projekt, który przedstawia przykładowe użycie biblioteki. Analiza poszczególnych rozwiązań dokonana jest na systemie sprzedażowym dla sieci sklepów.

Uwzględniana jest ogólna architektura pozwalająca na rozpoznawanie danej firmy na podstawie zapytania do serwera oraz trzy sposoby przechowywania danych poszczególnych firm:

- Przechowywanie danych różnych firm w jednej bazie danych, w jednym schemacie bazy danych. Dane poszczególnych firm są identyfikowane poprzez dodatkową kolumnę w każdej tabeli zawierającą Id firmy, do której należy dany rekord w tabeli.
- Przechowywanie danych różnych firm w jednej bazie danych, lecz dane każdej z firm będą znajdować się w innym schemacie bazy danych.
- Przechowywanie danych firm w różnych bazach danych. Każda firma ma osobną bazę danych.

1.5. Technologie

Do implementacji systemu oraz biblioteki Multitenancy użyta została platforma .NET oraz język C# w wersji 5.0 wraz z Entity Framework. Baza danych wykorzystana w systemie to Microsoft SQL. Do stworzenia diagramów zostało użyte darmowe narzędzie draw.io. Na potrzeby testów oraz integracji danych używana jest również platforma chmurowa Microsoft Azure. Wykorzystywane są takie zasoby jak:

- Bazy danych (MSSql)
- Azure App Services
- Wirtualne Maszyny
- DataFactory

Zastosowane technologie są nowoczesne, wieloplatformowe oraz ciągle utrzymywane i rozwijane przez ich producentów. Do pełnego zrozumienia pracy nie jest wymagana wiedza z zakresu wykorzystanych w projekcie technologii. Wszystkie rozwiązania przedstawione są w taki sposób, aby były zrozumiałe dla każdego. Celem tego podejścia jest powszechny dostęp do wiedzy zawartej w pracy oraz czerpanie korzyści niezależnie od wykorzystywanych w projektach technologii.

1.6. Rezultaty pracy

W rezultacie powstał dokument szeroko opisujący architekturę wielodostępną, problemy z nią związane oraz proponowane rozwiązania. W celu lepszego zobrazowania oraz potwierdzenia wyników analiz powstała również generyczna oraz przenaszalna biblioteka Multitenancy dla projektów .NET napisana w języku C#. Znajdzie ona zastosowanie w wielu projektach informatycznych. W znacznym stopniu ułatwi zrozumienie oraz implementacji wielodostępności.

Obszerny opis zagadnienia wielodostępności oraz opis poszczególnych podejść i rozwiązań problemów zdecydowanie pomoże w zagłębieniu wiedzy z dziedziny architektury systemów informatycznych. Z pewnością okaże się przydatny dla przedsiębiorstw zajmujących się wytwarzaniem oprogramowania.

1.7. Organizacja pracy

Praca przedstawia opis problemu w celu dokładnego zrozumienia wyzwania przed jakim stają programiści. Pokazuje trudności, które można napotkać oraz rozwiązania potencjalnych problemów. Opisy oraz diagramy pomagają zrozumieć architekturę oraz sposób przechowywania i integracji danych w tego typu systemach. Następnie praca skupia się na porównaniu poszczególnych sposobów realizacji rozwiązania problemu wielodostępności. Wskazuje ich wady oraz zalety. Zawiera również krótkie podpowiedzi, kiedy dany sposób powinien zostać wdrożony. Kolejny rozdział pokazuje ogromne możliwości rozwiązania – możliwość przetrzymywania bazy na serwerze klienta. Następnie praca skupia się na bardzo poważnym oraz ciekawym problemie integracji danych. W dalszej części dokładnie przedstawiona zostaje biblioteka Multitenancy oraz sposoby jej implementacji dla różnych rozwiązań. Autor przedstawia również strukturę kodu, architekturę biblioteki oraz problemy i ich rozwiązania podczas implementacji. W końcowym rozdziale pracy, tuż przed podsumowaniem autor opisuje możliwości jakie daje opisywana architektura w systemach rozproszonych.

2. Opis problemu

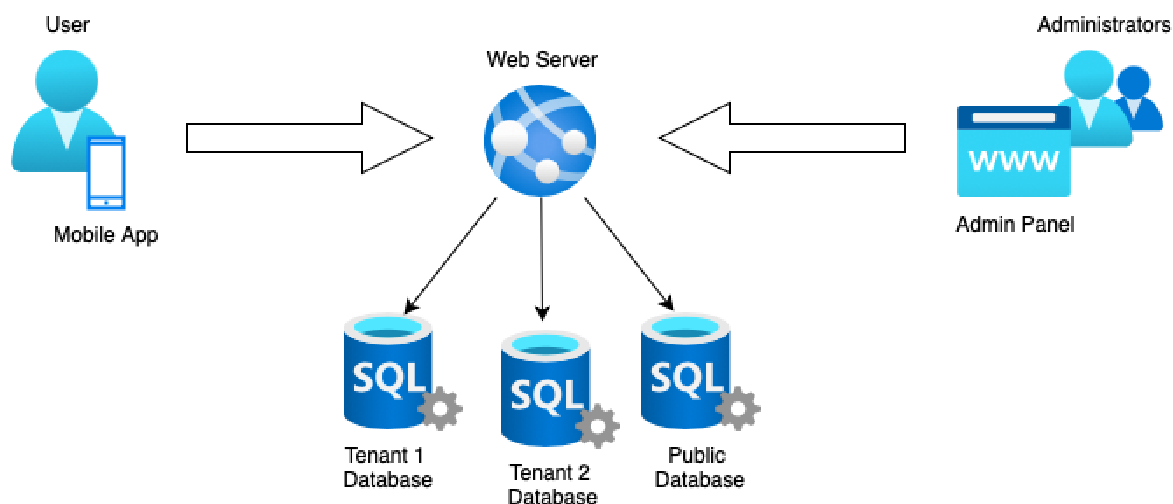
Projektowanie systemu wielodostępnego bez odpowiedniej wiedzy może okazać się bardzo zdradliwe. Z pozoru prosty system, wraz z czasem oraz implementacją kolejnych założeń biznesowych przerodzić się może w praktycznie niemożliwy do utrzymania projekt. Dobrze jest więc wiedzieć z czym ma się do czynienia zaczynając pracę nad tego typu programem.

Jako przykład praca przedstawia system sprzedażowy obsługujący sieci sklepów. Pierwszymi założeniami biznesowymi będą więc możliwości dodawania nowej sieci (firmy), jej sklepów, produktów oraz użytkowników (administratorów). Tworzony jest jeden serwer, który będzie obsługiwał zarówno administratorów sklepów jak i zwykłych użytkowników, którzy dokonują zakupów poprzez aplikację mobilną. Zakłada się, że istnieje jedna aplikacja mobilna, w której użytkownik może wybrać sklep, w którym chce zrobić zakupy a następnie dokonać transakcji online. Rozważana jest sytuacja, gdzie aplikacja jest agregatorem sklepów, a nie dedykowaną aplikacją danego sklepu. System składa się z kilku komponentów:

- Serwer
- Aplikacja mobilna
- Panel administracyjny

Na potrzeby pracy magisterskiej rozważane są rozwiązania tylko w części serwerowej, jako iż aplikacja mobilna oraz panel administracyjny mogą zostać stworzone w klasyczny, prosty sposób.

Poglądowy diagram działania systemu przedstawiony jest na Rysunku 1.

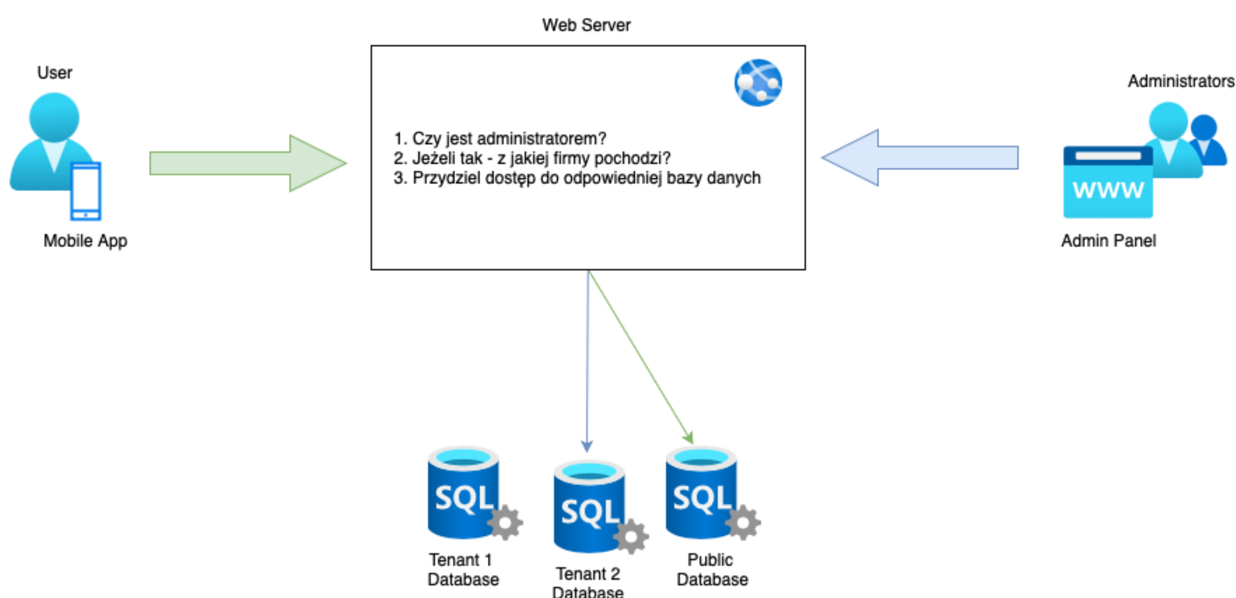


Rysunek 1. Poglądowy diagram działania systemu

W tej sytuacji należy przeanalizować kilka kwestii:

- Przechowywanie danych różnych, niezależnych od siebie firm. Jako że, gromadzone są dane personalne pracowników sieci, należy zachować szczególną ostrożność, aby były one odpowiednio od siebie odizolowane. Niektóre sieci sklepów mogą wymagać, aby baza danych przetrzymywana była na ich serwerze.
- Identyfikacja użytkownika (czy jest administratorem oraz z jakiej firmy pochodzi) na podstawie zapytania do serwera. Administrator, niezależnie od tego z jakiej firmy pochodzi powinien mieć dostęp wyłącznie do zasobów swojej firmy.
- Integracja danych z różnych firm. Użytkownik, który chce zrobić zakupy w aplikacji powinien mieć dostęp do publicznych danych wszystkich firm.

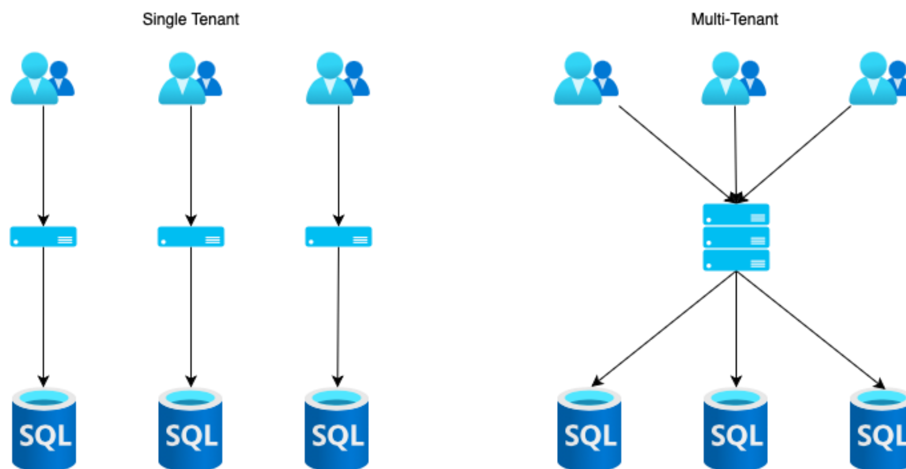
Powyższe scenariusze zostały przedstawione na Rysunku 2 dla lepszego zobrazowania problemu.



Rysunek 2. Scenariusze zapytań do serwera

2.1. Architektura wielodostępna

Architektura wielodostępna (ang. *Multitenancy*) pozwala na użycie pojedynczej instancji serwera do obsługi wielu dzierżawców (ang. *Tenants*). W pracy magisterskiej jest ona zaprezentowana na przykładzie systemu sprzedażowego obsługującego wiele sieci sklepów. W przedstawionym systemie dzierżawcą jest firma/sieć sklepów. Rysunek 3 przedstawia różnice pomiędzy klasycznym podejściem „Single-Tenant” oraz oprogramowaniem wielodostępnym „Multi-Tenant”.



Rysunek 3. Różnice pomiędzy „Single-Tenant”, a „Multi-Tenant”

2.2. Dane dzierżawców

Ważnym tematem, na którym skupia się praca jest kwestia przechowywania danych należących do dzierżawców. Czy przetrzymywanie niezależnych danych różnych firm w jednej bazie danych bądź w jednej tabeli bazy danych jest bezpieczne? Jak zmieniają się koszty oraz wydajność systemu w zależności od umiejscowienia danych? Z każdym kolejnym pytaniem nasuwa się coraz więcej wątpliwości dotyczących przechowywania danych.

Przykładowy projekt pokazuje różne realizacje systemu. Począwszy od najprostszego rozwiązania, gdzie dane przechowujemy w jednej bazie, aż do rozwiniętego systemu, gdzie każda z organizacji/firm może przechowywać dane na własnym serwerze. Porównanie schematów danych zostało pokazane na Rysunku 4. Część oznaczona symbolem „A” przedstawia tabele z bazy, która przechowuje dane wszystkich dzierżawców. Natomiast fragment oznaczony jako „B” pokazuje przykładową strukturę tabel w osobnych bazach danych.

A

Database

Id	Name	CompanyId
1	John	Tenant1
2	Adam	Tenant1
3	Tom	Tenant2

B

Tenant 1 Database

Id	Name
1	John
2	Adam

Tenant 2 Database

Id	Name
3	Tom

Rysunek 4. Porównanie schematów tabel w bazach danych

Korzyści oraz problemy wynikające z różnych rozwiązań tego problemu szczegółowo opisuje punkt 3 - Sposoby realizacji.

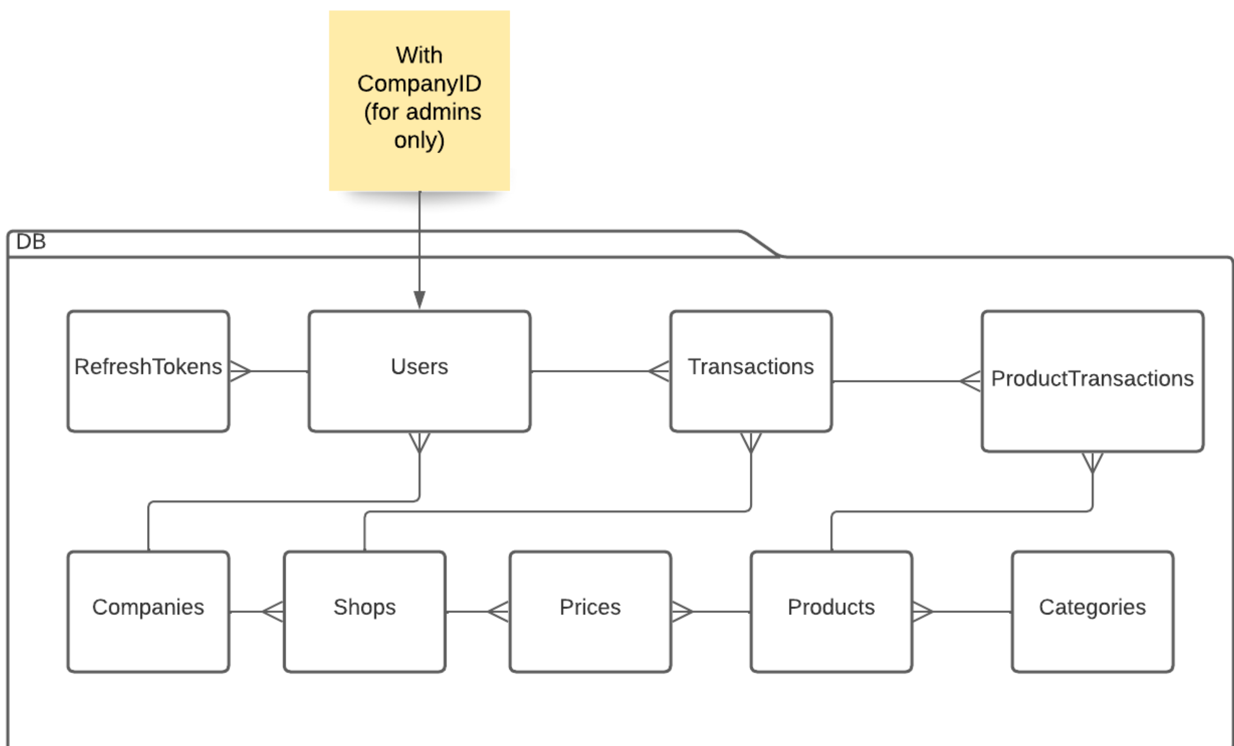
3. Sposoby realizacji

Istnieje kilka sposobów realizacji systemu opartego o oprogramowanie wielodostępne. Każde z tych rozwiązań ma swoje wady oraz zalety. W zależności od potrzeb system powinien opierać się na jednym z przedstawionych podejść. Zdecydowanie inne wymagania będą obowiązywać w mniejszym projekcie, gdzie budżet oraz zasoby ludzkie do utrzymania systemu są ograniczone. W przypadku gdy ilość organizacji korzystających z serwisu nie jest wystarczająco duża, a dane przechowywane w bazach nie są wrażliwe, lepiej sprawdzi się prostsze oraz tańsze rozwiązanie. Z drugiej strony, gdy mamy do czynienia z większym systemem przechowującym dane osobowe, z pewnością bezpieczniejszą opcją okaże się izolacja danych. Nie ma jednego, perfekcyjnego rozwiązania. Każde z nich powinno być wybierane w zależności od potrzeb, budżetu oraz wymagań biznesowych.

Na implementacje przedstawionych rozwiązań 3.1, 3.2 oraz 3.3 pozwala autorska biblioteka Multitenancy, która opisana została w punkcie 5.

3.1. Klasyczne podejście – jedna baza

Rozwiązanie z jedną bazą danych to klasyczne podejście, które jest najprostsze i najbardziej powszechne. Przypomina standardową architekturę. Diagram związków encji (ERD) bazy danych przedstawiony jest na Rysunku 5.



Rysunek 5. Diagram związków encji – jedna baza

W tym przypadku dane wszystkich sieci sklepów przechowujemy w tych samych tabelach. Są identyfikowane za pomocą klucza obcego CompanyId. Jest to dość wygodne rozwiązanie jednak mały błąd podczas tworzenia zapytania do bazy np. brak odpowiedniego filtra CompanyId = X może mieć poważne konsekwencje w postaci wycieku danych wszystkich firm. Jest to zdecydowany minus tego rozwiązania. Ponadto jest ono bardzo podatne na błędy programisty. Przykład obrazujący błąd programisty przedstawiony został na listingu 2. Listing 1 to poprawnie napisana funkcja, która pobiera z bazy danych ceny produktów z konkretnej firmy dla danego produktu. W projekcie zostały użyte takie wzorce projektowe jak m.in. wzorzec jednostki pracy (ang. *UnitOfWork*) [9] oraz specyfikacja (ang. *Specification*) [9]. Znacznie ułatwia to czytanie poniższego kodu.

Listing 1. Poprawny kod pobierający ceny produktów.

```
Var prices = UnitOfWork.Prices

.Include(p=>p.Product).ThenInclude(p=>p.Category)
.Include(p=>p.Shop).ThenInclude(s=>s.Company).AsNoTracking()
.DefaultFilter().FromCompany(GetCompanyId())
.WithProductName(product).DefaultOrder();
```

Listing 2. Błędny kod pobierający ceny produktów.

```
Var prices = UnitOfWork.Prices

.Include(p=>p.Product).ThenInclude(p=>p.Category)
.Include(p=>p.Shop).ThenInclude(s=>s.Company).AsNoTracking()
.DefaultFilter().WithProductName(product).DefaultOrder();
```

Jak zostało przedstawione wystarczy tylko pominąć jeden filtr FromCompany, aby doprowadzić do poważnego błędu niosącego za sobą spore konsekwencje.

Z drugiej strony to rozwiązanie jest najprostsze w implementacji i nie wymaga większej wiedzy ani umiejętności programistów. Nie jest jednak aż tak proste jak może się wydawać na pierwszy rzut oka. Wraz z rozrastaniem i ewolucją systemu natrafić można na kolejne problematyczne kwestie:

- Filtrowanie i pobieranie danych jest bardzo podatne na błąd programisty. Konsekwencje małego niedopatrzenia mogą być ogromne. Może dojść do ujawnienia danych osobowych administratorów innej firmie.
- Odróżnienie użytkownika od administratora firmy jest możliwe poprzez dodanie kolumny CompanyId do tabeli „Users”. Wciąż jednak przetrzymujemy administratorów wszystkich firm w jednej tabeli. Ponadto w tym szczególnym przypadku w tej samej tabeli trzymamy również użytkowników systemu – klientów sklepów.
- W przypadku otrzymania dostępu do bazy przez niepowołaną do tego osobę np. hakera (ang. *Hacker*), ma on dostęp do wszystkich danych wszystkich firm.
- Rozrost danych poszczególnych firm może doprowadzić do znacznego zwiększenia rozmiaru bazy danych. Może mieć to wpływ na działanie systemu nawet dla firm, które nie przechowują dużej ilości danych w bazie. Utrudni to również skalowanie systemu.

To rozwiązanie niesie ze sobą również szereg korzyści takich jak:

- Relatywnie prosty system z prostą architekturą. Jest on również dużo łatwiejszy w utrzymaniu.
- Dużo niższe koszty utrzymania. Wiąże się to również z brakiem konieczności angażowania do projektu bardziej doświadczonej kadry ekspertów, co przekłada się na mniejszy koszt projektu.

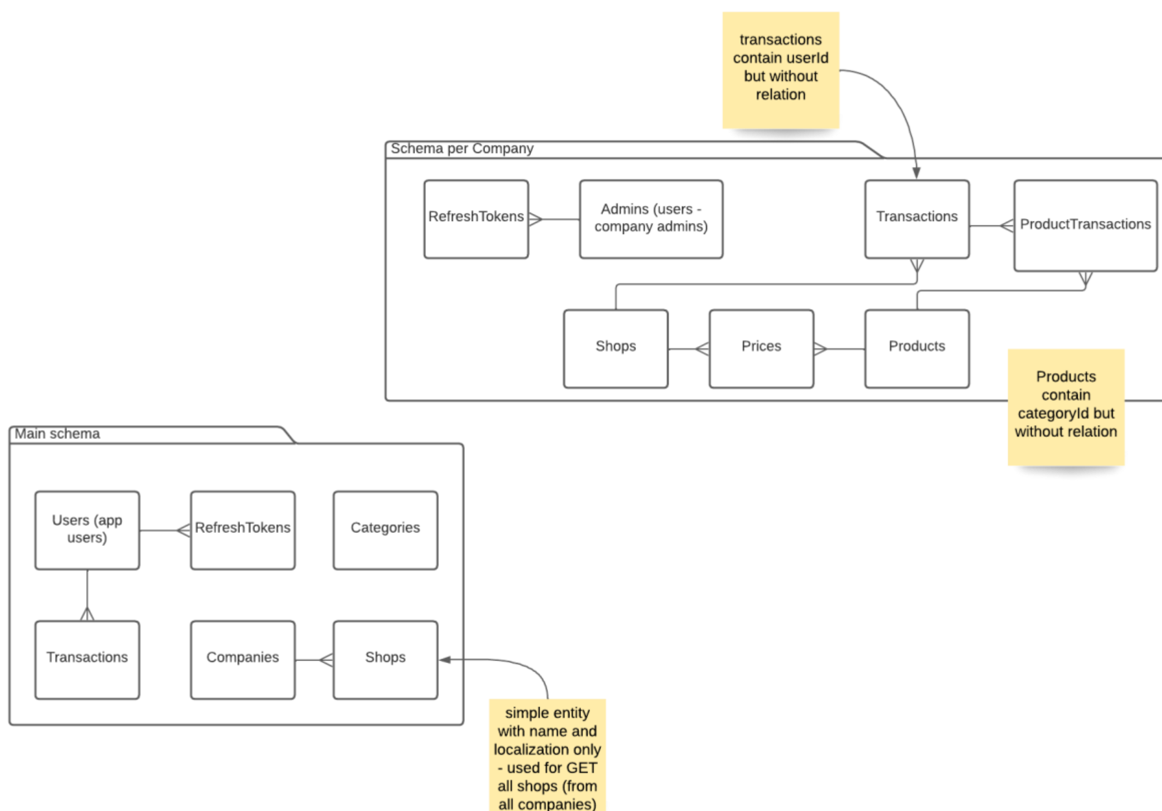
To rozwiązanie dobrze sprawdzi się w małych projektach oraz jako prototyp, bądź też MVP (ang. *Minimum Viable Product*) w przypadku, gdy czas na stworzenie systemu jest kluczowy.

3.2. Osobna baza dla każdej firmy

Ciekawym rozwiązaniem tego problemu może być stworzenie osobnej bazy danych dla każdej firmy. Daje to bardzo duże możliwości rozszerzania oraz modyfikacji warstwy danych. Dane są całkowicie odizolowane, znajdują się na innych serwerach, do których dostęp mają tyko uprawnione do tego osoby. Minimalizuje to podatność na wyciek danych oraz niebezpieczeństwo wynikające z błędu programisty. Wiąże się to oczywiście z większym kosztem utrzymania oraz zdecydowanie bardziej skomplikowaną architekturą systemu.

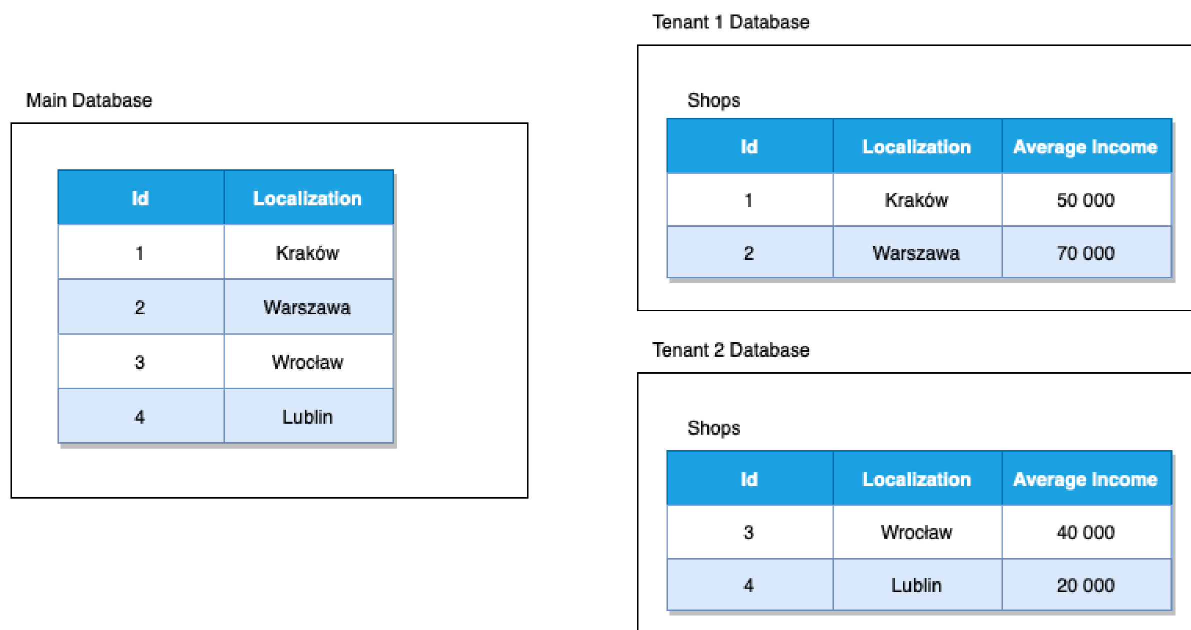
Takie podejście do problemu pozwala na bezpieczniejsze przechowywanie danych. Wymaga jednak większego nakładu pracy na proces implementacji systemu. Co ważne, diagram bazy danych staje się bardziej skomplikowany oraz pojawiają się nowe trudności w postaci integracji danych.

Aby dopasować diagram do tego podejścia, należało go nieco zmienić. Diagram przedstawiony jest na Rysunku 6.



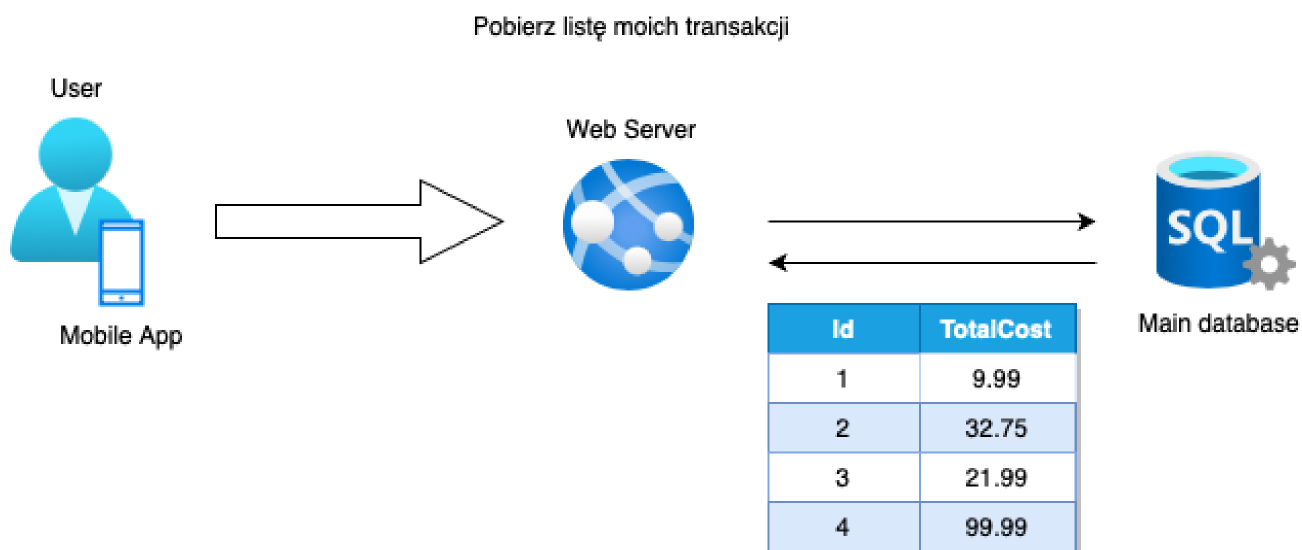
Rysunek 6. Diagram związków encji – osobna baza dla każdej firmy

Jak można zauważyć, nie wystarczy w tym przypadku stworzyć osobnej bazy dla każdej z firm, ale również wymagana jest jedna główna baza danych, która będzie przechowywać dane dostępne dla wszystkich użytkowników oraz wszystkich firm. W tym przypadku w głównej bazie przechowujemy podstawowe dane użytkowników, korzystających z aplikacji – nie możemy ich przypisać do konkretnej sieci sklepów. Znajdują się tu również dane dotyczące firm oraz podstawowe dane ich sklepów po to, żeby użytkownicy mogli wybrać odpowiedni sklep, w którym chcą zrobić zakupy bez konieczności integracji danych ze wszystkich schematów bazy. W głównej bazie pojawiają się również transakcje użytkowników. Są to uproszczone rekordy transakcji dokonanych przez użytkowników. Jest to duplikat danych z baz danych poszczególnych sklepów. Duplikaty tworzymy po to, aby uprościć proces integracji danych ze wszystkich baz. Załóżmy przykładowy scenariusz, w którym użytkownik chce pobrać listę sklepów w okolicy, czy też listę dokonanych transakcji. Serwer jest w stanie zwrócić odpowiednie dane w błyskawiczny sposób, gdyż są one przechowywane na serwerze głównym. Z tą różnicą, że zwrócona lista sklepów lub transakcji będzie zawierała jedynie bardzo podstawowe informacje jak np. nazwa sklepu oraz adres lub data transakcji oraz kwota całkowita. Przedstawia to Rysunek 7.

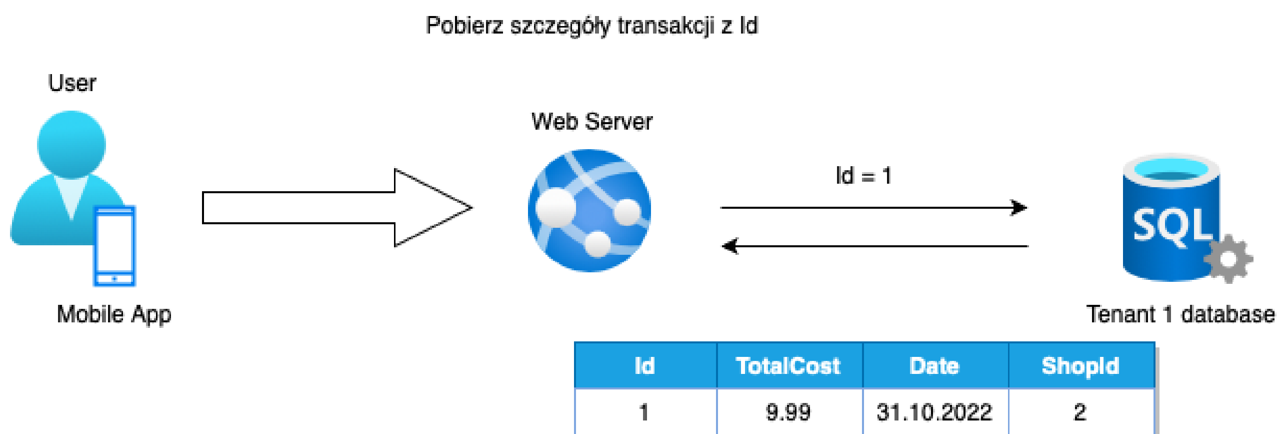


Rysunek 7. Duplikaty danych w głównej bazie (osobna baza dla każdej firmy)

W przypadku, gdy użytkownik będzie chciał pobrać szczegóły danego sklepu lub transakcji, wtedy aplikacja wykona zapytanie do konkretnego serwera danej firmy, z której pochodzi dany zasób. Tabele te są czymś w rodzaju interfejsu. Wystawiają one publicznie dane pochodzące z konkretnych firm, ale w bardzo okrojonej wersji, niezawierającej wrażliwych danych. Pobranie transakcji z głównej tabeli ilustruje diagram przedstawiony na Rysunku 8. Rysunek 9 natomiast przedstawia pobieranie konkretnej transakcji po Id z bazy danych dzierżawcy.

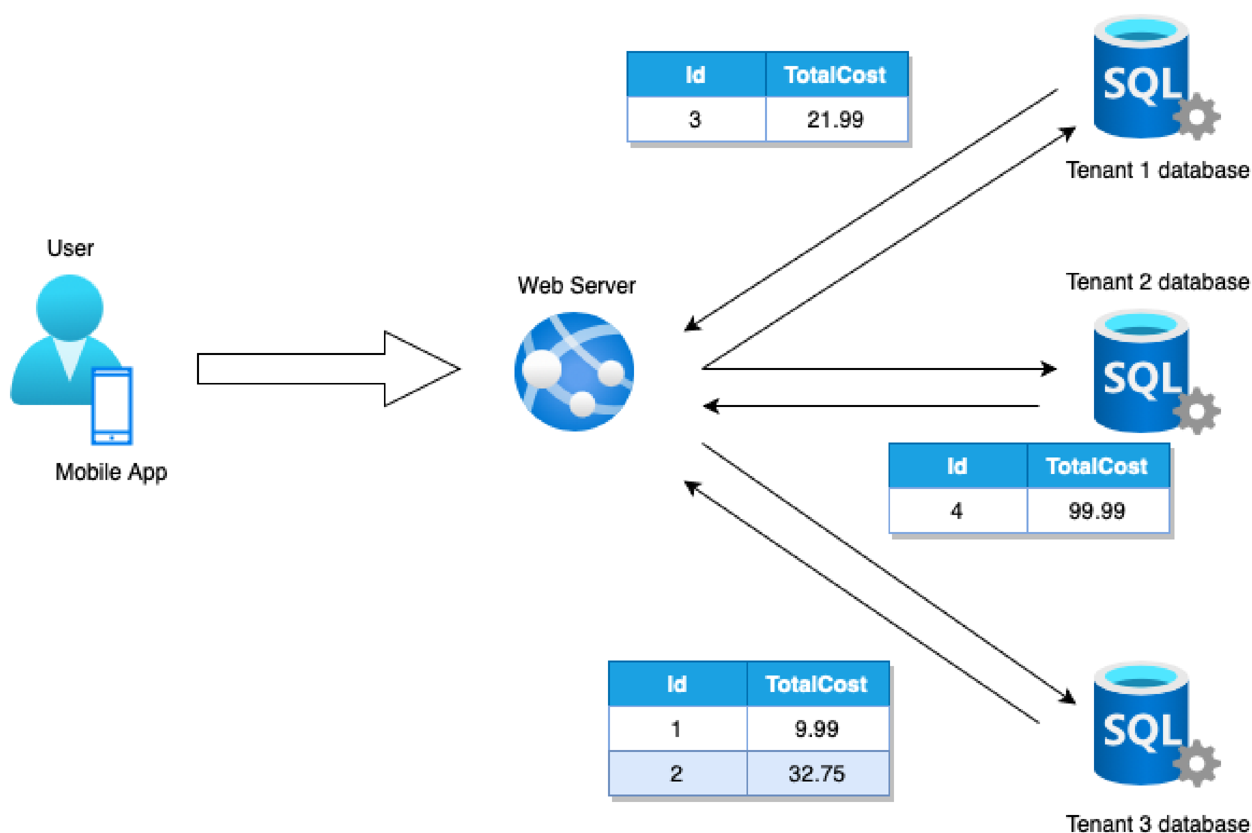


Rysunek 8. Pobranie listy transakcji z bazy głównej



Rysunek 9. Pobieranie transakcji z Id z bazy dzierżawcy

Gdybyśmy nie zdecydowali się na takie rozwiązanie, pobranie listy transakcji wiązałoby się z wykonaniem zapytania do serwera w celu uzyskania informacji o wszystkich sieciach sklepów a następnie wykonaniem serii zapytań do serwerów wszystkich sieci sklepów w celu pobrania listy transakcji. Przedstawia to Rysunek 10.



Rysunek 10. Pobranie listy transakcji z baz dzierżawców.

Innym rozwiązaniem dla tego problemu mogą być hurtownie danych. Integracja danych w ten sposób w przykładowym systemie sprzedażowym została opisana w punkcie 4.3.

Takie podejście do rozwiązania problemu daje nam wiele korzyści takich jak:

- Dobra izolacja danych,
- Możliwość wycieku danych spowodowana błędem programisty ograniczona do minimum,
- Znacznie większa elastyczność systemu pozwalająca na dostosowanie rozmieszczenia danych w celu maksymalizacji wydajności systemu. Różne bazy systemu mogą być umieszczone na różnych serwerach np. bliżej danego klienta – baza dla firmy z USA znajdować się może na serwerach w Ameryce, natomiast baza firmy z Polski – w Europie.
- Możliwość dostosowania baz w systemie rozproszonym. Różne serwisy mogą korzystać z innych baz w celu zwiększenia wydajności. Rozszerzony opis rozwiązania wielodostępności w architekturze mikroserwisowej znajduje się w punkcie 7 pracy magisterskiej.
- Dzięki temu rozwiązaniu możemy również pozwolić klientowi (firmie) na przechowywanie danych na własnym serwerze. To rozwiązanie zostało zaimplementowane w ramach pracy magisterskiej oraz opisane w punkcie 4.
- W przypadku ataku hakerskiego na daną bazę lub serwer narażone będą dane tylko jednej z firm, a nie całego systemu.

To rozwiązanie ma również kilka poważnych minusów:

- Warstwa danych staje się dość skomplikowana logicznie,
- Utrzymanie systemu wymaga doświadczonej kadry,
- Eksperci pracujący nad systemem oraz skomplikowana architektura chmurowa doprowadzają oczywiście do znacznego zwiększenia kosztów systemu,
- Pojawia się również problem integracji danych. Rozwiązaniem są tutaj duplikaty encji/tabel opisane w punkcie 3.2, bądź też hurtownie danych opisane w punkcie 4.3.

To podejście świetnie sprawdzi się w bardzo dużych systemach biznesowych, gdzie wydajność oraz elastyczność ma kluczowe znaczenie, natomiast koszt projektu odgrywa drugorzędną role.

3.3. Osobny schemat bazy dla każdej firmy

Innym rozwiązaniem, które nieco lepiej izoluje dane użytkowników niż pojedyncza baza opisana w punkcie 3.1 i nie jest tak podatne na błędy programisty to stworzenie osobnego schematu bazy danych, dla każdej firmy.

Schemat w bazie danych to niezależna warstwa bazy. Istnieje możliwość nadania odpowiednich uprawnień tylko do konkretnego schematu bazy. Domyślnie (w przypadku MS SQL) wszystkie dane zapisywane są do schematu „dbo" – tak dzieje się w przypadku opisanym w punkcie 3.1.

W tym przypadku diagram bazy będzie identyczny jak w punkcie 3.2, przedstawiony na Rys. 6. System będzie posiadał jeden schemat główny oraz osobne schematy stworzone dla każdej firmy. Dzięki temu dane będą lepiej izolowane, będzie trudniej o błąd programisty, a integracja danych nie będzie stanowiła aż tak dużego wyzwania. W obrębie jednej bazy zdecydowanie prościej integrować dane do wyświetlenia np. poprzez widoki. Widok (perspektywa) w bazie danych pozwala na tworzenie czegoś w rodzaju tymczasowej tabeli, w której znajdują się tylko potrzebne dane pochodzące z jednej lub z wielu tabel. Ponadto istnieje możliwość ograniczenia dostępu do danego widoku, co w tym przypadku byłoby bardzo pomocne. W dodatku integracja danych w obrębie jednej bazy danych nie wymagałaby nawiązywania połączeń sieciowych z innymi bazami/serwerami, co również znacznie uproszcza sytuację.

Jak już zostało wspomniane, takie podejście niesie ze sobą wiele korzyści:

- Izolacja danych na poziomie jednej bazy danych,
- Zminimalizowana szansa na błąd programisty prowadzący do wycieku danych,
- Łatwiejsza integracja danych niż dla podejścia opisanego w 3.2,
- Niższe koszty utrzymania systemu ze względu na prostą infrastrukturę chmurową

Niestety pojawiają się również minusy tego rozwiązania takie jak:

- Warstwa danych staje się dość skomplikowana logicznie. Pomimo iż wykorzystywana jest jedna bazy danych to projekt staje się bardzo podobny do tego z podejścia 3.2,
- Utrzymanie systemu jest trudniejsze niż w przypadku jednej bazy z jednym schematem,
- Przetrzymywane dane znajdują się wciąż na jednym serwerze, na jednej bazie danych. Sprawia to, że rozwiązanie jest mniej elastyczne. Nie ma możliwości przenoszenia poszczególnych danych na inne serwery bądź też dostosowywania danych do funkcjonalności serwisu w systemie rozproszonym,
- Ponadto rozwiązanie jest zdecydowanie bardziej podatne na atak hakera. W przypadku dostania się do bazy, istnieje większe ryzyko pozyskania danych wszystkich firm korzystających z systemu.

Autor twierdzi, iż to podejście dobrze sprawdzi się jako przystanek pomiędzy projektem z jedną bazą, a projektem z osobną bazą dla każdej z firmy. Jest ono pośrednim rozwiązaniem, dla projektów które w przyszłości planują tworzyć osobne bazy dla każdej z firm, lecz w obecnej fazie projektu nie jest to konieczne oraz budżet jest zbyt niski. Jako iż implementacja projektu jest bardzo zbliżona do podejścia opisanego w punkcie 3.2 bardzo łatwo przejść do tego rozwiązania w przyszłości. Z uwagi na to, że infrastruktura chmurowa jest zdecydowanie prostsza, gdyż nadal przetrzymujemy dane w jednej bazie – koszty są znacznie niższe. Podejście to jest dobrym rozwiązaniem dla przedsiębiorców, którzy w momencie startu projektu wiedzą, że chcą dążyć do osobnych baz klientów, ale w początkowej fazie projektu nie jest to potrzebne.

4. Baza danych na serwerze firmy

Rozwiązanie z osobną bazą dla każdej z firm daje programistom ogromne możliwości dostosowywania systemu pod klienta. Istnieje możliwość pozwolenia klientowi na przechowywanie danych na własnym serwerze z własną bazą danych o określonej strukturze.

Na implementację tego rozwiązania pozwala autorska biblioteka Multitenancy, która opisana została w punkcie 5.

4.1. Korzyści rozwiązania

Rozwiązanie z pewnością ma dla niektórych klientów ogromne znaczenie. W niektórych krajach nie jest dozwolone wprowadzanie oprogramowania, które wysyła dane poza terytorium kraju, bez zgody władz. Przykładem takiego kraju są Chiny [1]. Przechowywanie więc danych na serwerach klienta mogłoby przyczynić się do ułatwienia procesu wdrożenia systemu.

Ponadto trzymając się przykładu systemu sprzedażowego dla wielu sieci sklepów można zauważyć kolejne korzyści płynące z tego podejścia. Szczegółowe dane produktów, cen, użytkowników oraz dokonywanych przez nich transakcji są poufne. Duże sieci sklepów nie chcą dzielić się z konkurencją jakie produkty najlepiej sprzedają się w danej lokalizacji. Nie chcą udostępniać informacji o tym w jakim wieku użytkownicy najczęściej kupują dane produkty. Takich przykładów można znaleźć znacznie więcej. Są to kluczowe dane, które bardzo często decydują o tym, która z firm ma przewagę na rynku. Takie przedsiębiorstwa zdecydowanie nie będą chciały przechowywać tych informacji we wspólnej bazie danych.

Z tym rozwiązaniem wiążą się również korzyści wynikające z przechowywania danych w osobnych bazach dla każdej z firm:

- Dobra izolacja danych,
- Możliwość wycieku danych spowodowana błędem programisty ograniczona do minimum,
- Znacznie większa elastyczność systemu pozwalająca na dostosowanie rozmieszczenia danych w celu maksymalizacji wydajności systemu. Różne bazy systemu mogą być umieszczone na różnych serwerach,
- Możliwość dostosowania baz w systemie rozproszonym. Różne serwisy mogą korzystać z innych baz w celu zwiększenia wydajności. Rozszerzony opis rozwiązania wielodostępności w architekturze mikroserwisowej znajduje się w punkcie 7 pracy magisterskiej.
- W przypadku ataku hakerskiego na daną bazę lub serwer narażone będą dane tylko jednej z firm, a nie całego systemu.

4.2. Minusy rozwiązania

Głównym minusem takiego rozwiązania jest integracja danych. Przykładowo w przypadku, gdy dany klient udzieli zgody na wykorzystywanie danych z własnego serwera do raportów pozwalających na analizę preferencji zakupowych danej grupy społecznej w danym regionie pojawia się problem zebrania danych od klientów. W przypadku jednej bazy problem praktycznie nie istnieje, natomiast dla wielu baz w dodatku na serwerach klientów problem staje się dość poważny. Jest to główny problem, nad którym autor skupia się w pracy oraz prezentuje jego potencjalne rozwiązanie.

W przykładowym systemie sprzedażowym dla wielu firm została dodana hurtownia danych. Opis jej implementacji oraz działania znajduje się w podpunkcie 4.3.

To rozwiązanie ma również kilka poważnych minusów wynikających z przechowywania danych w osobnych bazach dla każdej z firm:

- Warstwa danych staje się dość skomplikowana logicznie,
- Utrzymanie systemu wymaga doświadczonej kadry,
- Eksperci pracujący nad systemem oraz skomplikowana architektura chmurowa doprowadzają oczywiście do znacznego zwiększenia kosztów systemu,

4.3. Integracja danych – Hurtownia danych

Jak zostało opisane w punkcie 4.2 pojawia się tutaj problem integracji danych. Rozwiązanie tego problemu zostanie przedstawione na przykładowym projekcie omawianym już we wcześniejszych rozdziałach pracy magisterskiej.

Jako iż każda z firm może posiadać własną bazę danych nie jest prostym zadaniem pobranie danych z wielu sieci sklepów jednocześnie. Co w przypadku, gdy właściciele aplikacji będą chcieli sprawdzić, z którymi sklepami praca jest najbardziej owocna. Które sieci sklepów oraz które konkretnie sklepy przynoszą największe zyski oraz przeprowadzane jest w nich najwięcej transakcji. Co w przypadku próby uzyskania informacji o kategoriach produktów najlepiej sprzedających się w danych miastach/sieciach sklepów (aby na przykład wdrożyć plan promocji na dane artykuły).

Kolejną kwestią są statystyki poszczególnych użytkowników. Jak użytkownik sprawdzi na jakie kategorie produktów wydał najwięcej pieniędzy, jeżeli jego transakcje będą przetrzymywane w różnych bazach danych (inna dla każdej sieci sklepów).

Rozwiązaniem jest tutaj hurtownia danych.

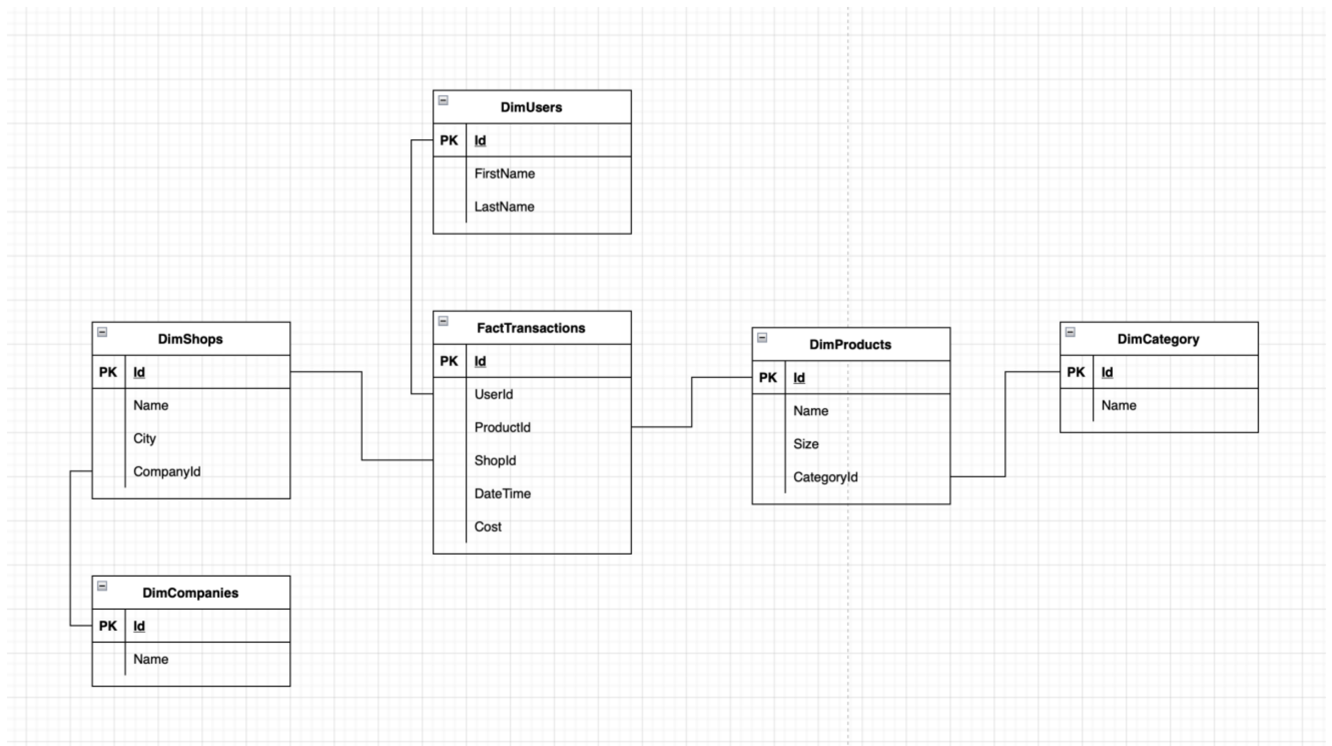
Hurtownia danych to rodzaj bazy danych, który trwale przechowuje zintegrowane dane, zazwyczaj zoptymalizowana pod konkretne cele. Hurtownie są wykorzystywane do integracji danych ze wszystkich źródeł danych w systemie. [2]

Wyodrębnionych zostało kilka pytań, na które zostały udzielone odpowiedzi, dzięki stworzonej hurtowni danych oraz wygenerowanym raportom:

- Ile transakcji użytkownik X wykonał w sklepach danej firmy?
- Ile pieniędzy użytkownik X wydal na produkty danej kategorii?
- Ile transakcji użytkownik X wykonał w danym mieście?
- Jaki jest najczęściej kupowany rozmiar produktu?
- Która firma (*tenan*t) sprzedała najwięcej produktów danej kategorii?
- Który sklep sprzedał najwięcej produktów danej kategorii?
- Produkty jakich kategorii najlepiej sprzedają się w danych miastach?

4.3.1 Diagram hurtowni danych

Diagram hurtowni danych został przedstawiony na Rysunku 11.



Rysunek 11. Schemat płatka śniegu - hurtownia danych

Fakt to pojedyncze zdarzenie będące podstawą analiz [3].

W przedstawionym przykładzie tabelą faktów jest „Fact transaction". Jedna transakcja reprezentuje zakup pojedynczego produktu, w przypadku transakcji zawierającej kilka produktów, rozbijana jest ona na pojedyncze produkty.

Wymiar to cecha opisująca dany fakt [4].

Przykładowa hurtownia danych zawiera następujące wymiary:

- Dim Users
- Dim Shops -> Dim Companies
- Dim Products -> Dim Categories

4.3.2 Procesy ETL

Na potrzeby projektu zostały stworzone 3 bazy sieci sklepów (*tenant*) oraz baza główna „Main". Wszystkie źródła danych to bazy MSSQL postawione w chmurze Azure. Do tabeli „Companies" zostały dodane następujące firmy:

- tenant0,
- tenant1,
- tenant2.

Tabela „Categories” zawiera 10 kategorii. Pozostałe tabele zostały załadowane również sztucznie generowanymi danymi (od 500 do 10000 wierszy). Została również utworzona hurtownia danych.

Rysunek 12 przedstawia stworzone zasoby na platformie Azure.

Name ↑↓	Type ↑↓	Location ↑↓
sales-sysetem-server	SQL server	Norway East
SalesSystemDataWarehouse (sales-sysetem-server/SalesSystemDataWareho...	SQL database	Norway East
SalesSystemDF	Data factory (V2)	West Europe
SalesSystemTenant1 (sales-sysetem-server/SalesSystemTenant1)	SQL database	Norway East
SalesSystemTenant3 (sales-sysetem-server/SalesSystemTenant3)	SQL database	Norway East
SystemSalesMain (sales-sysetem-server/SystemSalesMain)	SQL database	Norway East

Rysunek 12. Zasoby na platformie Azure

Do importu danych zostało użyte Azure Data Factory.

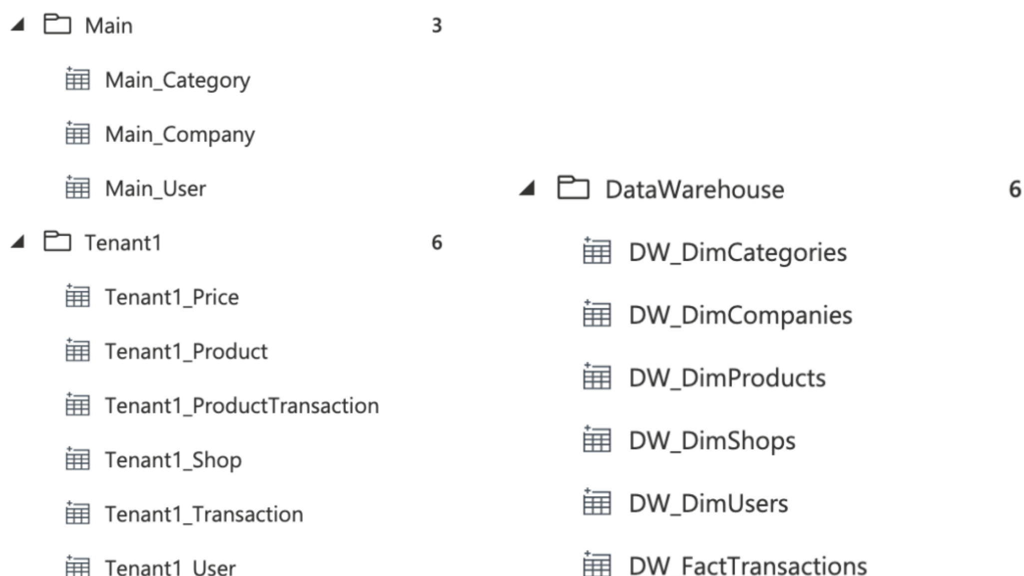
Azure Data Factory to usługa ETL w chmurze platformy Azure do skalowanej w poziomie i bezserwerowej integracji i transformacji danych [5].

Następnie został stworzone przepływ danych (ang. *Data flow*) dla głównej bazy „Main” oraz dla każdej z firm, w celu transferu danych do hurtowni danych. Stworzone zasoby przedstawia Rysunek 13.

Pipelines 1
SalesSystemPipeline
Datasets 21
DataWarehouse 6
Main 3
Tenant1 6
Tenant3 6
Data flows 5
MainDataFlow
Tenant1DataFlow
Tenant1TransactionsDataFlow
Tenant3DataFlow
Tenant3TransactionsDataFlow

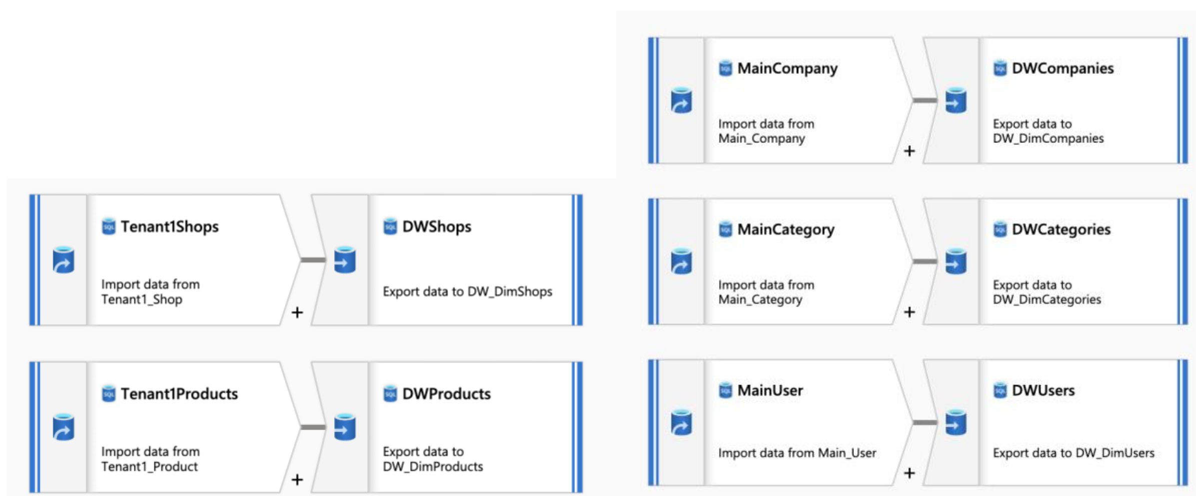
Rysunek 13. Stworzone zasoby hurtowni danych

Następnie konieczne było stworzenie źródeł oraz ujść danych. Źródła i ujście danych (hurtownia danych) przedstawione są na Rysunku 14.



Rysunek 14. Źródła oraz ujścia danych hurtowni.

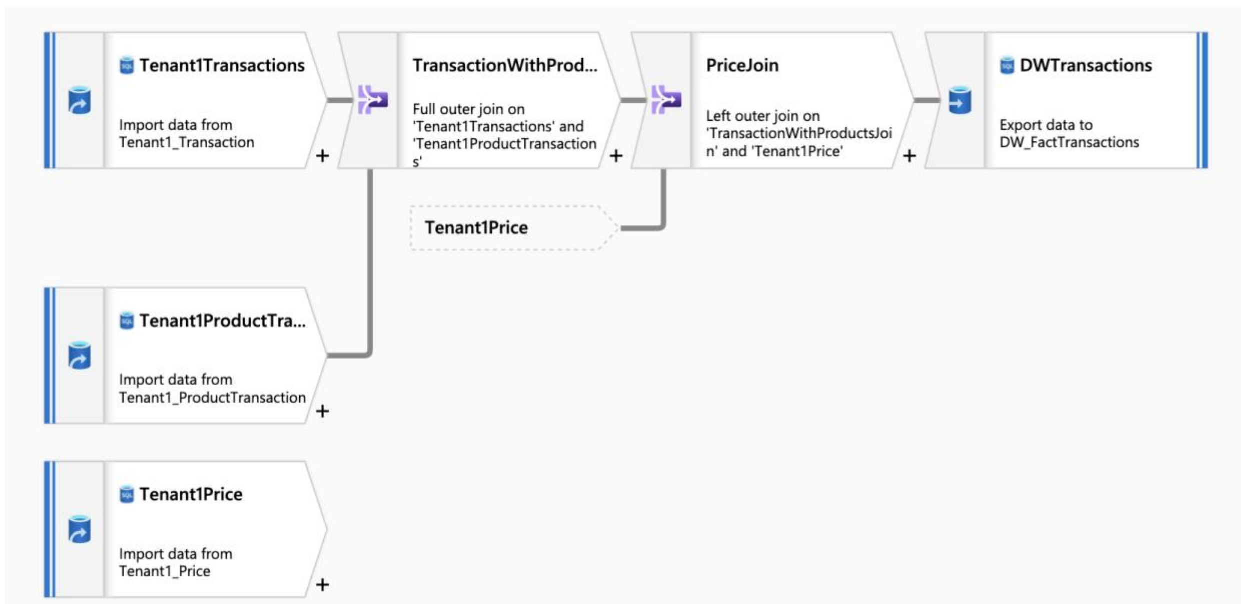
Stworzone przepływy danych dla tabeli „Main" oraz „Tenant1" przedstawione zostały na Rysunku 15.



Rysunek 15. Przepływy danych dla tabeli „Main" oraz „Tenant1"

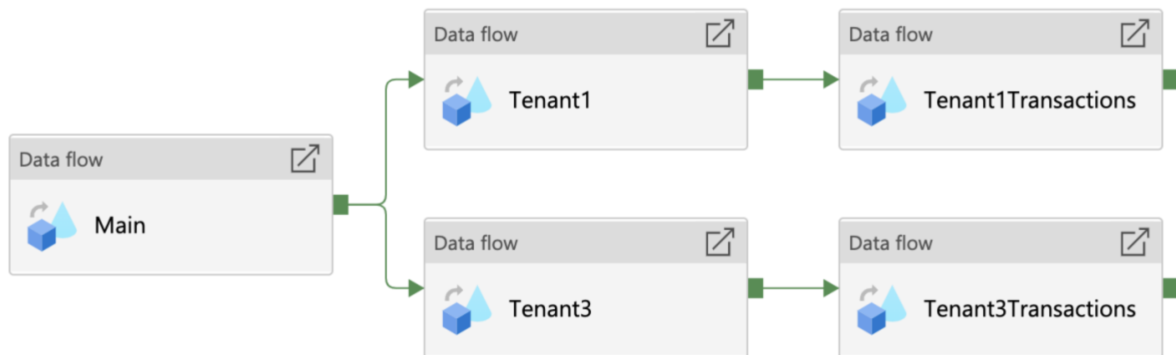
Przepływ danych dla transakcji był nieco bardziej skomplikowany. Musiała zostać połączona tabela transakcji z tabelą „ProductTransaction" oraz „Price" w taki sposób, aby ostatecznie transakcja została rozdzielona na pojedyncze elementy oraz zapisana z odpowiednią ceną. Połączenie przedstawione jest na Rysunku 16.

Analogicznie zostały stworzone przepływy danych dla pozostałych dzierżawców. Procesy ETL były o tyle uproszczone, że każde ze źródeł danych miało taką samą strukturę oraz przetrzymywało dane w tym samym formacie (ponieważ dane do każdej z baz dodawał jeden główny serwer systemu).



Rysunek 16. Złączenia danych dla przepływu danych transakcji

Rysunek 17 przedstawia potok (ang. *Pipeline*) zawierający przepływy danych dzierżawców do hurtowni danych. Warto tutaj również zaznaczyć, że kolejność DataFlow nie jest bez znaczenia.

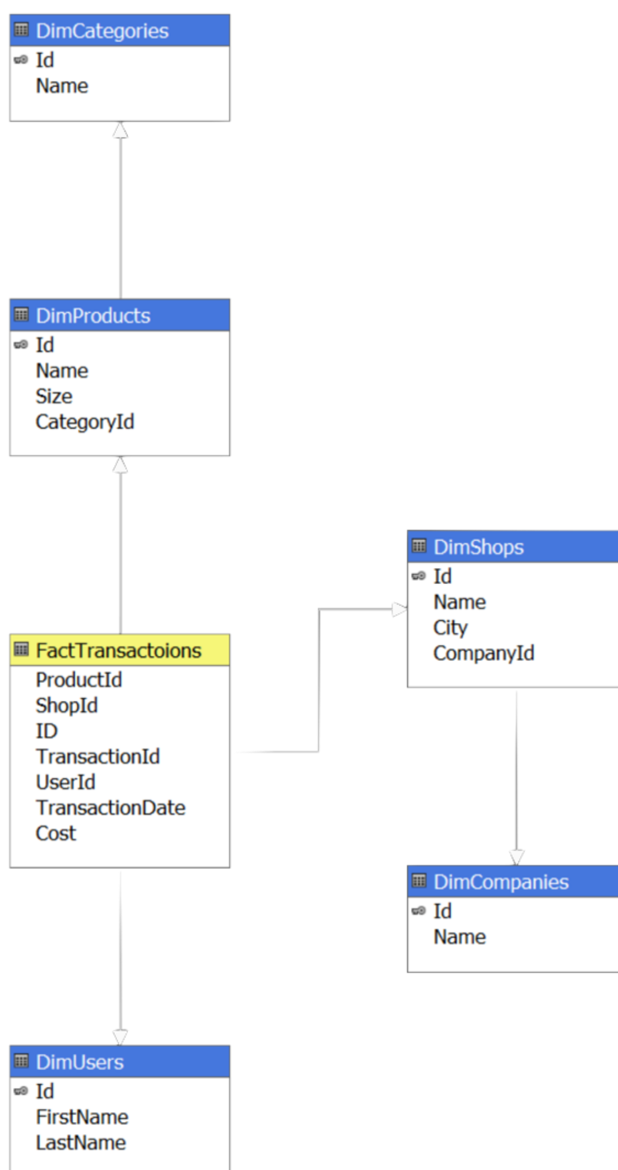


Rysunek 17. Potok przepływu danych dzierżawców do hurtowni danych

4.3.3 Kostka OLAP

Kostka OLAP to struktura danych, która pozwala na szybką analizę danych. Jest to wielowymiarowa baza danych.

Do stworzenia kostki OLAP użyto Visual Studio z rozszerzeniem Analysis Services [6]. Wymagało to dodania źródła danych (hurtowania danych z Azure), stworzenia Widoku dla hurtowni oraz utworzenia kostki wraz z wymiarami. Struktura kostki OLAP przedstawiona została na Rysunku 18.



Rysunek 18. Struktura kostki OLAP

4.3.4 Raporty

Do tworzenia raportów wykorzystany został program MS Excel. Dane zostały przedstawione w taki sposób, aby odpowiadały na wcześniej postawione pytania.

Do celów testowych, użytkownik X to użytkownik z Id: ”001F457B-A687-4E15-9187-167A80BB0DC3”.

Przykładowe raporty przedstawiające odpowiedzi na wcześniej postawione pytania:

- Ile transakcji użytkownik X wykonał w sklepach danej firmy? (Rys. 19)

	A	B	C
1	**Row Labels**	**Fact Transactoins Count**	**Cost**
2	**001F457B-A687-4E15-9187-167A80BB0DC3**		
3	**tenant0**		
4	Shop240	7	20
5	**tenant1**		
6	Shop480	6	28
7	Tenant1-Shop213	14	64
8	**tenant2**		
9	Shop352	3	21
10	**Grand Total**	**30**	**133**

Rysunek 19. Transakcje użytkownika w sklepach danej firmy

- Ile pieniędzy użytkownik X wydal na produkty danej kategorii? (Rys. 20)

	A	B	C
1	**Row Labels**	**Fact Transactoins Count**	**Cost**
2	**001F457B-A687-4E15-9187-167A80BB0DC3**		
3	**Category0**		
4	Product257	1	6
5	Product477	1	7
6	Product797	1	3
7	Product841	1	7
8	**Category1**		
9	Product304	1	5
10	Product473	1	5
11	Product652	1	2
12	Tenant1-Product497	2	18
13	**Category2**		
14	Product245	1	2
15	Product398	1	9

Rysunek 20. Wydane pieniądze użytkownika na produkty danej kategorii

- Ile transakcji użytkownik X wykonał w danym mieście? (Rys. 21)

	A	B	C
1	Row Labels	Fact Transactoions Count	Cost
2	⊟001F457B-A687-4E15-9187-167A80BB0DC3		
3	⊟tenant0		
4	City0	7	20
5	⊟tenant1		
6	City18	14	64
7	City5	6	28
8	⊟tenant2		
9	City14	3	21
10	Grand Total	30	133

Rysunek 21. Transakcje użytkownika w danym mieście

- Jaki jest najczęściej kupowany rozmiar produktu? (Rys. 22)

	A	B	C
1	Row Labels	Fact Transactoions Count	Cost
2	⊟Product0		
3	3	20	109
4	⊟Product1		
5	4	30	153
6	⊟Product10		
7	6	34	153
8	⊟Product100		
9	6	22	102
10	⊟Product101		
11	5	23	95

Rysunek 22. Najczęściej kupowany rozmiar produktu

- Która firma (*tenant*) sprzedała najwięcej produktów danej kategorii? (Rys. 23)

	A	B	C
1	Row Labels	Fact Transactoions Count	Cost
2	⊟Category0		
3	⊞tenant0	2808	14251
4	⊞tenant1	2846	14190
5	⊞tenant2	2781	13963
6	⊟Category1		
7	⊞tenant0	2690	13489
8	⊞tenant1	2791	13900
9	⊞tenant2	2686	13442
10	⊟Category2		
11	⊞tenant0	2463	12404
12	⊞tenant1	2516	12674
13	⊞tenant2	2572	13034

Rysunek 23. Najwięcej sprzedanych produktów danej kategorii (Firma)

- Który sklep sprzedał najwięcej produktów danej kategorii? (Rys. 24)

Row Labels	Fact Transactoions Count	Cost
⊟Category0		
⊞tenant0	2808	14251
⊟tenant1		
Shop1	7	36
Shop100	6	25
Shop102	5	27
Shop104	5	32
Shop109	4	15
Shop110	7	26
Shop115	4	29
Shop116	7	34
Shop118	7	34
Shop12	8	46
Shop123	3	16

Rysunek 24. Najwięcej sprzedanych produktów danej kategorii (Sklep)

- Produkty jakich kategorii najlepiej sprzedają się w danych miastach? (Rys. 25)

Row Labels	Fact Transactoions Count	Cost
⊟City0		
Category0	353	1851
Category1	364	1887
Category2	337	1654
Category3	273	1256
Category4	319	1552
Category5	254	1255
Category6	290	1569
Category7	292	1436
Category8	335	1648
Category9	344	1815
⊟City1		
Category0	469	2343
Category1	427	2107
Category2	410	2148
Category3	442	2198
Category4	419	2060
Category5	337	1740
Category6	375	1918
Category7	391	1951
Category8	462	2347
Category9	441	2260

Rysunek 25. Sprzedaż produktów danej kategorii w miastach

4.3.5 Podsumowanie

Hurtownie danych to potężne narzędzie, które znajduje zastosowanie w wielu praktycznych projektach. Z pewnością wdrożenie tak potężnego zasobu jakim jest hurtownia danych musi być poparte celami biznesowymi.

Hurtownia może być wykorzystywana nie tylko do analizy sieci sklepów oraz dostosowywania ich w jak największym stopniu do klienta, ale również do prowadzenia dokładnych statystyk danych użytkowników.

Jak widać integracja danych w takim systemie nie jest czymś banalnie prostym. Wymaga dodatkowego nakładu pracy oraz pieniędzy.

5. Opis autorskiej biblioteki Multitenancy

W ramach pracy magisterskiej powstała generyczna biblioteka Multitenancy pozwalająca na implementacje rozwiązania w dowolnym systemie informatycznym stworzonym w technologii .NET z użyciem języka C#. Pozwala ona na identyfikację dzierżawcy na podstawie zapytania do serwera oraz na implementację warstwy danych zgodną z architekturą wielodostępną. Biblioteka została stworzona zgodnie z wytycznymi czystego kodu (ang. *Clean Code*) [7].

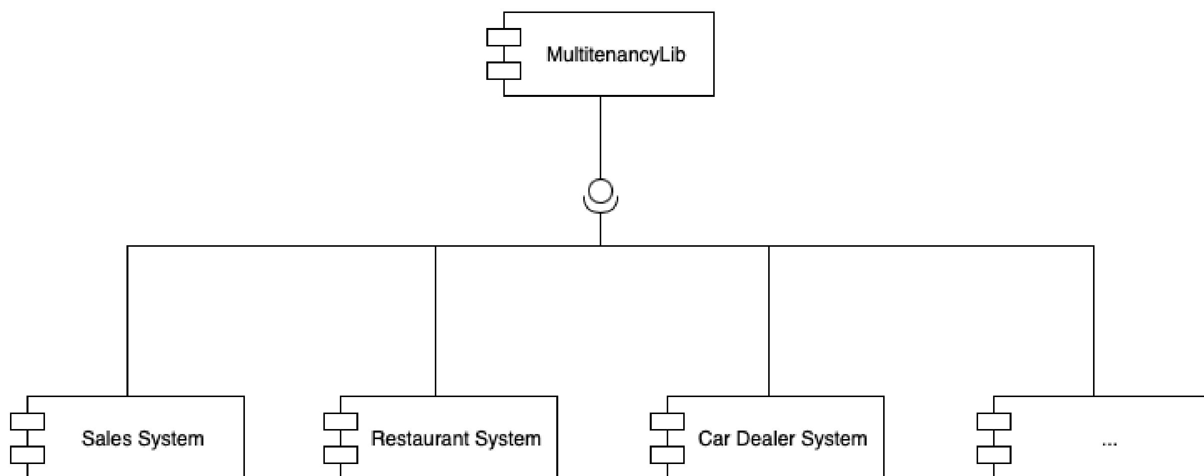
5.1. Opis

Projekt informatyczny powstał w celu weryfikacji analiz przedstawionych w pracy magisterskiej. Dzięki niemu autor pracy sprawdził, jak wygląda implementacja architektury w praktyce co dało mu lepszy pogląd na temat systemów opartych o oprogramowanie wielodostępne.

Pomysł na bibliotekę powstał podczas tworzenia przykładowego systemu sprzedażowego dla wielu sklepów. Zostały stworzone generyczne interfejsy oraz klasy bazowe dające możliwość łatwego ponownego użycia biblioteki w projekcie. Ogólny zamysł przedstawia Rysunek 26.

Celem biblioteki było:

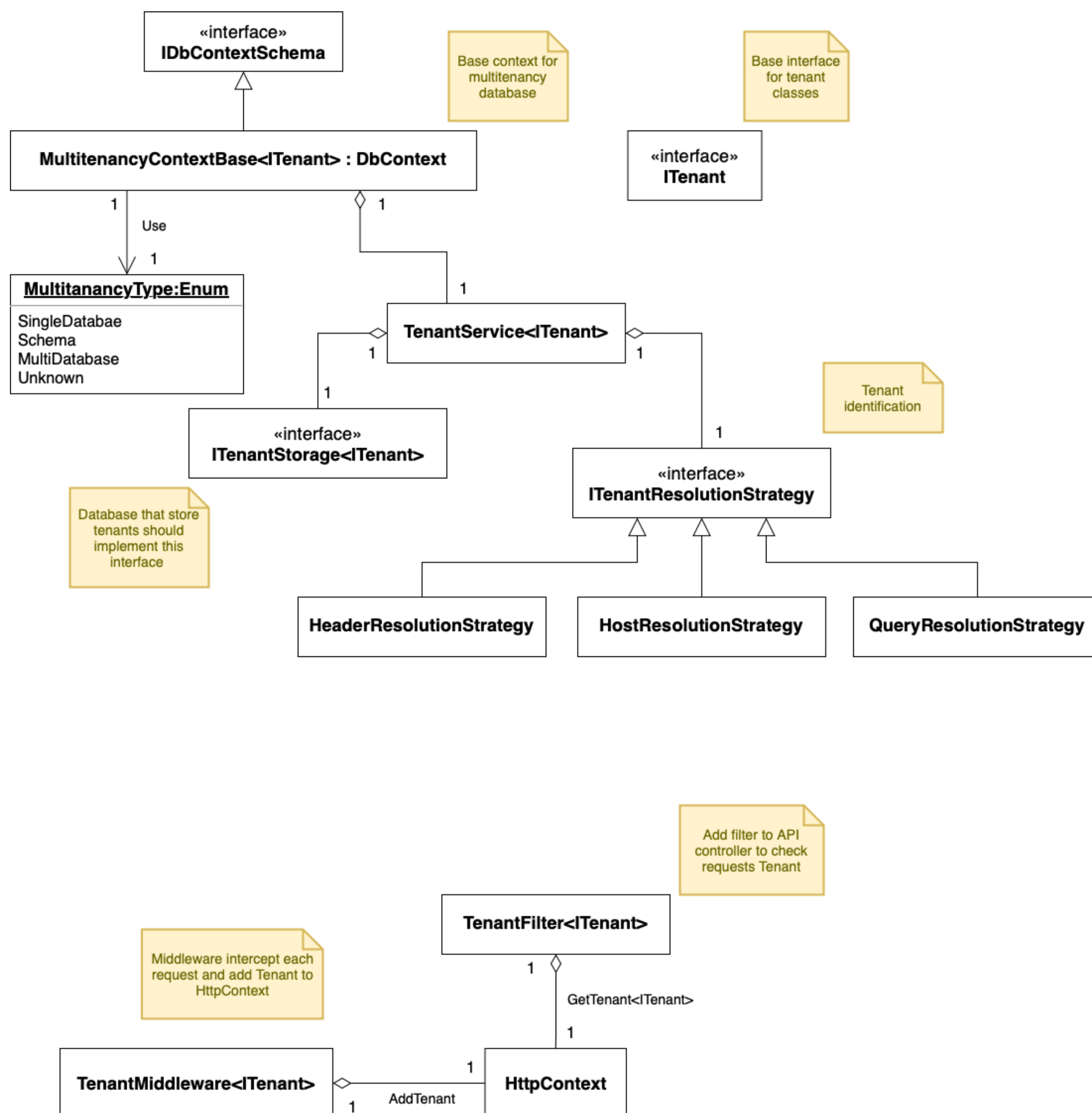
- Zestaw klas oraz interfejsów pozwalający na implementację architektury wielodostępnej w systemie informatycznym,
- Generyczność pozwalająca na dostosowanie biblioteki do potrzeb systemu,
- Przenaszalność dająca możliwość użycia zestawu klas oraz interfejsów w każdym projekcie,
- Prosta implementacja możliwa dzięki zestawowi rozszerzeń umożliwiającym dodanie biblioteki przy pomocy kilku linijek kodu.



Rysunek 26. Diagram komponentów przedstawiający wykorzystanie biblioteki Multitenancy w wielu systemach

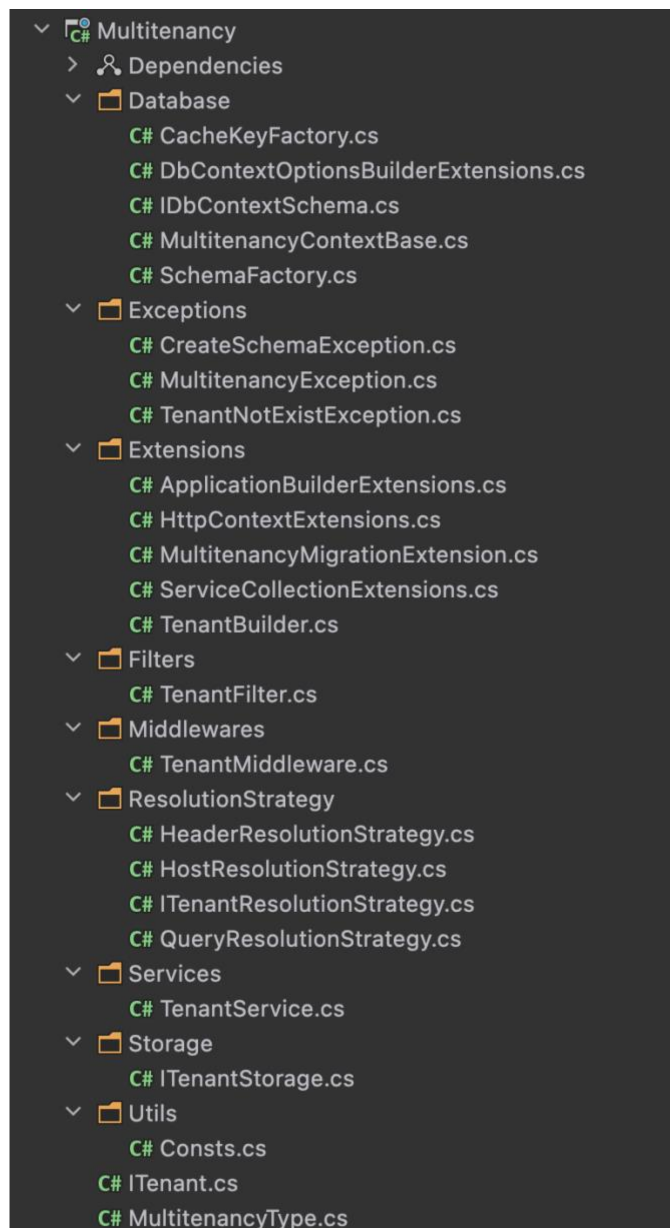
5.2. Architektura

Biblioteka składa się z wielu klas, interfejsów oraz serwisów pozwalających na implementację architektury wielodostępnej. Zawiera ona bazowe klasy oraz interfejsy dla typowych elementów systemu. Rysunek 27 przedstawia diagram biblioteki.



Rysunek 27. Diagram klas przedstawiający architekturę biblioteki Multitenancy

Ze względu na czytelność diagramu (Rys. 27) nie zawiera on pomocniczych klas, ułatwiających użycie rozwiązania w projekcie. Są one natomiast opisane w punkcie 5.4 – Implementacja w projekcie. Struktura kodu oraz podział klas i katalogów w projekcie przedstawiony został na rysunku 28. Dokładny opis poszczególnych elementów znajduje się w kolejnych podpunktach punktu 5.2.



Rysunek 28. Struktura kodu biblioteki Multitenancy

5.2.1 ITenant

`ITenant` to bazowy interfejs, który musi być zaimplementowany przez klasę opisującą dzierżawcę i jest używany w wielu serwisach, klasach oraz interfejsach do jego identyfikacji w systemie. Istnieje możliwość stworzenia więcej niż jednej encji implementującej `ITenant`. Wymaga to jednak stworzenia osobnych instancji odpowiednich serwisów. Interfejs przedstawiony jako listing 3. składa się z właściwości takich jak:

- `Id` – identyfikuje dzierżawcę wewnątrz system, unikalna wartość,
- `Identifier` – identyfikuje dzierżawcę publicznie,
- `ConnectionString` – umożliwia połączenie z bazą danych dzierżawcy.

Listing 3. Interfejs ITenant

```
public interface ITenant
{
    public Guid Id { get; set; }
    public string Identifier { get; set; }
    public string ConnectionString { get; set; }
}
```

5.2.2 MultitenancyType

MultitenancyType to typ wyliczeniowy (Enum) ułatwiający identyfikację wybranego rozwiązania. Biblioteka umożliwia wybranie dwóch możliwości:

- Schema – jedna baza danych z osobnym schematem dla każdego z dzierżawców,
- MultiDatabase – osobna baza danych dla każdeg dzierżawcy.

Opcja SingleDatabase oznacza, że klasy biblioteki Multitenancy nie są używane. Enum zawiera również opcję Unknown. W przypadku braku ustalenia konkretnego rozwiązania wybierane jest rozwiązanie domyślne. Domyślne rozwiązanie może być ustawione dowolnie przez programistę. Listing 4 przedstawia typ wyliczeniowy MutitenancyType.

Listing 4. Enum MultitenancyType

```
public enum MultitenancyType
{
    SingleDatabase,
    Schema,
    MultiDatabase,
    Unknown
}
```

5.2.3 MultitenancyContextBase

MultitenancyContextBase to bazowa klasa dla wielodostępności. Każda klasa w projekcie, w której będzie używane wiele schematów lub baz danych musi dziedziczyć po tej klasie. Nie musi natomiast dziedziczyć po DbContext ponieważ robi to już MultitenancyContextBase. Implementuje ona również interfejs IDbContextSchema. Jest on używany w przypadku jednej bazy danych z osobnymi schematami dzierżawców. Interfejs jest bardzo prosty i zawiera jedno pole Schema. Przedstawiony został jako listing 5.

Listing 5. Interfejs IDbContextSchema

```
public interface IDbContextSchema
{
    string Schema { get; set; }
}
```

Klasa MultitenancyContextBase przyjmuje w konstruktorze niezbędne do poprawnego działania argumenty. Przedstawia to listing 6.

Listing 6. Konstruktor klasy MultitenancyContextBase

```
public MultitenancyContextBase(
    MultitenancyType multitenancyType, string connectionString,
    TenantService<T> tenantService, string schema = null)
{
    _multitenancyType = multitenancyType;
    _connectionString = TenantConnectionString = connectionString;
    TenantService = tenantService;
    Schema = schema;
}
```

Nadpisana metoda dziedziczona klasy bazowej `DbContext` o nazwie `OnConfiguring` wykonuje się podczas każdego zapytania do bazy danych. Definiuje to dodanie serwisu poprzez użycie `AddDbContext` [8]. W tym miejscu wykorzystywany jest `TenantService` (opisany w punkcie 5.2.4) do pobrania obiektu odpowiedniego dzierżawcy. Następnie ustawiany jest schemat oraz parametry połącznia bazy. W zależności od wybranego `MultitenancyType` ustawiany jest odpowiedni schemat lub baza, na której będzie wykonywane zapytanie. Kod metody został przedstawiony jako listing 7.

Listing 7. Metoda OnConfiguring klasy MultitenancyContextBase

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
{
    var tenant = TenantService.GetTenantAsync().Result;
    if (tenant != null)
    {
        Schema = tenant.Id.ToString();
        TenantConnectionString =!
string.IsNullOrWhiteSpace(tenant.ConnectionString)
            ? tenant.ConnectionString
            : _connectionString.Replace(Consts.TemplateDatabaseName,
tenant.Id.ToString());
        }

        switch (_multitenancyType)
        {
            case MultitenancyType.Schema:
                options.UseMySchema();
                options.UseSqlServer(_connectionString, x =>
x.MigrationsHistoryTable(HistoryRepository.DefaultTableName, Schema));
                break;
            case MultitenancyType.MultiDatabase:
                options.UseSqlServer(TenantConnectionString);
                break;
            case MultitenancyType.SingleDatabase:
            case MultitenancyType.Unknown:
            default:
                throw new Exception("Invalid type of context.");
    }
}
```

Można zauważyć użycie niestandardowej metody `UseMySchema`, której domyślnie nie ma w klasie `DbContextOptionsBuilder`. W celu użycia konkretnego schematu bazy danych w trakcie działania programu (ang. *Runtime*) zostało użyte tutaj `UseSqlServer` z odpowiednimi parametrami oraz rozszerzenie klasy `DbContextOptionsBuilderExtensions`. Kod rozszerzenia (ang. *Extension*) został przedstawiony jako listing 8.

Listing 8. Metoda UseMySchema

```
public static DbContextOptionsBuilder UseMySchema (this
DbContextOptionsBuilder optionsBuilder)
{
   optionsBuilder.ReplaceService<IModelCacheKeyFactory, CacheKeyFactory>();
   return optionsBuilder;
}
```

Użyta została tutaj własna klasa `CacheKeyFactory` w celu zdefiniowania klucza modelu [9]. Kod klasy przedstawia listing 9.

Listing 9. Klasa CacheKeyFactory

```
public class CacheKeyFactory : IModelCacheKeyFactory
{
    public object Create(DbContext context)
    {
        return (context.GetType(),
                context is IDbContextSchema schema ? schema.Schema : null);
    }
}
```

Podsumowując, `MultitenancyContextBase` to kluczowy element systemu. Jest on niezbędny do działania z wieloma schematami oraz wieloma bazami danych dzierżawców. W czasie rzeczywistym dostosowuje schemat lub bazę danych do odpowiedniego dzierżawcy. Dzięki temu po jego zidentyfikowaniu (opisanym w punkcie 5.2.8) baza danych ustawiana jest na odpowiednie tabele. Wyklucza to całkowicie możliwość pomyłki lub udzielenia dostępu nieodpowiedniemu użytkownikowi do danych.

5.2.4 Tenant Service

`TenantService` to fasada [10]. Serwis ten umożliwia dostęp do pozostałych elementów systemu takich jak `TenantStorage` (opisany w 5.2.5) oraz `TenantResolutionStrategy` (opisany w 5.2.8). `TenantService` wykorzystywany jest we wcześniej opisanym `MultitenancyContextBase` oraz w `TenantMiddleware` (opisany w 5.2.6). W obu przypadkach serwis służy do pobrania obiektu reprezentującego dzierżawce. W pierwszej kolejności dzierżawca identyfikowany jest dzięki użyciu odpowiedniej implementacji `ITenantResolutionStrategy`, a następnie jest on pobierany z obecnie używanej implementacji `ITenantStorage`.

Kod serwisu jest stosunkowo bardzo prosty. Przyjmuje on w konstruktorze instancje `ITenantResolutionStrategy` oraz `ITenantStorage`. W metodzie `GetTenantAsync` wykorzystuje powyższe elementy i asynchronicznie [11] zwraca obiekt odpowiedniego dzierżawcy. Kod klasy przedstawiony został jako listing 10.

Listing 10. TenantService

```
public class TenantService<T> where T : ITenant
{
    public TenantService(ITenantResolutionStrategy
tenantResolutionStrategy, ITenantStorage<T> tenantStore)
    {
        _tenantResolutionStrategy = tenantResolutionStrategy;
        _tenantStorage = tenantStore;
    }

    public async Task<T> GetTenantAsync()
    {
        var tenantIdentifier = await
_tenantResolutionStrategy.GetTenantIdentifierAsync();
        return await _tenantStorage.GetTenantAsync(tenantIdentifier);
    }

    private readonly ITenantResolutionStrategy _tenantResolutionStrategy;
    private readonly ITenantStorage<T> _tenantStorage;
}
```

5.2.5 ITenant Storage

`ITenantStorage` to interfejs, który powinien zostać zaimplementowany przez encję przechowującą dane dzierżawców. Zwykle będzie to `DbContext`. Nie ma jednak takiego ograniczenia. Równie dobrze może to być zwykła klasa, która przechowuje informacje o dzierżawcach.

Interfejs jest bardzo prosty. Przedstawia go listing 11. Zawiera dwie definicje metod:

- `GetTenants` – pobranie wszystkich dzierżawców,
- `GetTenantAsync` – asynchroniczne pobranie konkretnego obiektu dzierżawcy z id.

Listing 11. ITenantStorage

```
public interface ITenantStorage<T> where T : ITenant
{
    List<T> GetTenants();
    Task<T> GetTenantAsync(string identifier);
}
```

5.2.6 TenantMiddleware

`TenantMiddleware` to pośrednia warstwa oprogramowania (ang. *Middleware*), która przechwytuje każde zapytanie do serwera. Analizuje ona zawartość danych w obiekcie `HttpContext` [12]. Pobiera dane dzierżawcy za pomocą `TenantService`, a następnie dodaje je do obiektu `HttpContext`. Ma to na celu jego identyfikacje wewnątrz jednego zapytania http. Wykorzystywane jest to np. w `TenantFilter` opisanym w punkcie 5.2.7. Kod `TenantMiddleware` został przedstawiony jako listing 12.

Listing 12. TenantMiddleware

```
public class TenantMiddleware<T> where T : ITenant
{
    private readonly RequestDelegate _next;

    public TenantMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        if (!context.Items.ContainsKey(Consts.HttpContextTenantKey))
        {
            if
(context.RequestServices.GetService(typeof(TenantService<T>)) is
TenantService<T> tenantService)
            {
                var tenant = await tenantService.GetTenantAsync();
                context.Items.Add(Consts.HttpContextTenantKey, tenant);
            }
        }

        if (_next != null)
            await _next(context);
    }
}
```

5.2.7 TenantFilter

`TenantFilter` to filtr [13] mający na celu sprawdzenie czy `httpContext` zawiera dane dzierżawcy dodane przez `TenantMiddleware`. Brak danych oznacza brak dzierżawcy w systemie. Klasa informuje wtedy użytkownika rzucając wyjątek. Kod klasy filtra przedstawiony został jako listing 13.

Listing 13. TenantFilter

```
public class TenantFilter<T> : IAsyncActionFilter where T : ITenant
{
    public async Task OnActionExecutionAsync(ActionExecutingContext
context, ActionExecutionDelegate next)
    {
        var httpContext = context.HttpContext;
        var tenant = await Task.FromResult(httpContext.GetTenant<T>());

        if (tenant == null)
        {
            throw new TenantNotExistException();
        }

        await next();
    }
}
```

5.2.8 ITenantResolutionStartegy (Identyfikacja dzierżawcy)

Dzierżawca może być identyfikowany na różne sposoby. Istnieje możliwość dodania własnej implementacji dostosowanej do potrzeb systemu. Wystarczy stworzyć nową klasę i zaimplementować interfejs `ITenantResolutionStartegy`. Sam interfejs jest bardzo prosty. Zawiera tylko jedną metodę `GetTenantIdentifierAsync`. Został przedstawiony jako listing 14.

Listing 14. ITenantResolutionStrategy

```
public interface ITenantResolutionStrategy
{
    Task<string> GetTenantIdentifierAsync();
}
```

Obecnie w systemie istnieją trzy implementacje tego interfejsu:

- `HeaderResolutionStrategy`
- `HostResolutionStrategy`
- `QueryResolutionStrategy`

HeaderResolutionStrategy

Identyfikacja użytkownika na podstawie nagłówka (ang. *Header*) http. Zapytanie musi mieć dodany nagłówek z kluczem: „x-tenant”. Wartością nagłówka powinien być identyfikator dzierżawcy. Implementacja metody dla strategii „Header” została przedstawiona jako listing 15.

Listing 15. HeaderResolutionStrategy

```
public async Task<string> GetTenantIdentifierAsync()
{
    if (_httpContextAccessor?.HttpContext?.Request == null)
        return null;

    var tenantIdentifier = string.Empty;

    if (await Task.FromResult
(_httpContextAccessor.HttpContext.Request.Headers.TryGetValue(HeaderKey,
out var headerValues)))
    {
        tenantIdentifier = headerValues.FirstOrDefault();
    }

    return tenantIdentifier;
}
```

HostResolutionStrategy

Identyfikacja dzierżawcy na podstawie domeny. Korzystając z tej metody należy również w odpowiedni sposób dostosować frontend aplikacji. Użytkownik wysyłający żądanie (ang. *Request*) do serwera powinien wysyłać go z domeny poprzedzonej subdomeną zawierającą `IDENTIFIER` dzierżawcy np. **tenant1**.salessystem.com. Implementacja metody dla strategii „Host" została przedstawiona jako listing 16.

Listing 16. HostResolutionStrategy

```
public async Task<string> GetTenantIdentifierAsync()
{
    if (_httpContextAccessor?.HttpContext?.Request == null)
        return null;

    return await Task.FromResult (_httpContextAccessor
.HttpContext.Request.Host.Host.Split('.').FirstOrDefault());
}
```

QueryResolutionStrategy

Identyfikacja dzierżawcy na podstawie parametrów zapytania (ang. *Query*). Użytkownik wysyłając zapytanie do serwera zobowiązany jest do dodania parametru zapytania o nazwie „tenant" oraz podania identyfikatora dzierżawcy jako wartości np. salessystem.com/shops?tenant=**tenant1**. Implementacja metody dla strategii „Query" przedstawiona jest jako listing 17.

Listing 17. QueryResolutionStrategy

```
public async Task<string> GetTenantIdentifierAsync()
{
    if (_httpContextAccessor?.HttpContext?.Request == null)
        return null;

    var tenant = await Task.FromResult
(_httpContextAccessor.HttpContext.Request.Query[ParamName]);
    return tenant;
}
```

5.3. Autoryzacja

Strategia identyfikacji użytkowników powinna być dobrana do potrzeb projektu. Należy jednak pamiętać, że w przypadku danych wrażliwych musi również zostać dodana autoryzacja użytkownika [14] względem danych dzierżawcy. W tym celu można dodać `AuthorizationRequirement` oraz `AuthorizationRequirementHandler` [15]. Przykładowy kod `TenantRequirement` został przedstawiony jako listing 18.

Listing 18. TenantRequirement

```
public class TenantRequirement : IAuthorizationRequirement
{
    public bool IsTenantValid(string tenantFromToken, string
resolvedTenant)
    {
        if (string.IsNullOrEmpty(tenantFromToken)
           || string.IsNullOrEmpty(resolvedTenant))
              return false;

        return tenantFromToken == resolvedTenant;
    }
}
```

Część kodu odpowiadająca za obsługę (ang. *Handler*) `TenantRequirement` przedstawiono jako listing 19.

Listing 19. Obsługa TenantRequirement - autoryzacja

```
protected override async Task
HandleRequirementAsync(AuthorizationHandlerContext context,
TenantRequirement requirement)
{
    var httpContext = _contextAccessor.HttpContext;
    var tenant = await Task.FromResult(httpContext.GetTenant<Tenant>());
    var tenantFromToken = context.User.Claims.FirstOrDefault(c => c.Type ==
"tenantId")?.Value;
    var tenantFromHostname = tenant?.Id.ToString();

    if (!requirement.IsTenantValid(tenantFromToken, tenantFromHostname))
    {
        context.Fail();
    }

    context.Succeed(requirement);
}
```

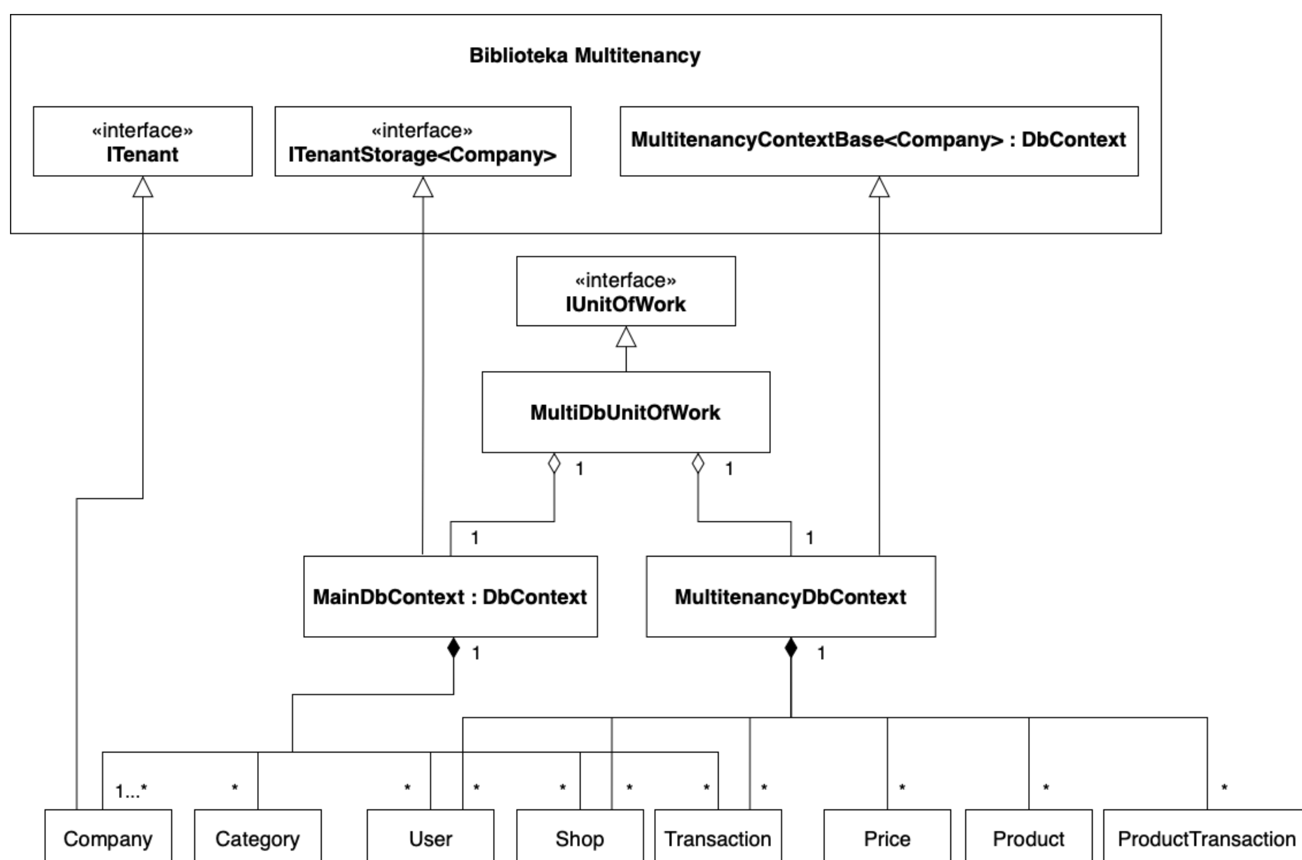
W tym przypadku należy pamiętać o dodaniu prawa (ang. *Claim*) o nazwie „tenantId' zawierającego wartość identyfikatora dzierżawcy podczas tworzenia tokenu uwierzytelniającego dla użytkownika.

5.4. Implementacja w projekcie

Biblioteka Multitenancy była tworzona w taki sposób, aby była łatwo przenaszalna, uniwersalna oraz bardzo prosta do zaimplementowania w docelowym systemie. Przykładowa implementacja biblioteki została przedstawiona na podstawie wcześniej już analizowanego systemu do sprzedaży dla wielu firm.

5.4.1 Architektura

Istnieje wiele sposobów implementacji interfejsów oraz klas bazowych z biblioteki Multienancy. Przykładowa propozycja architektury znajduje się na Rysunku 29. Jest to warstwa danych projektu, który został stworzony zgodnie z zasadami projektowania opartego na domenie (ang. *Domain-Driven Design*) [16].



Rysunek 29. Diagram klas przedstawiający architekturę projektu implementującego bibliotekę Multitenancy

W projekcie użyte zostały dwie klasy bazy danych:

- `MainDbContext` – zawiera encje znajdujące się w głównej bazie danych, opisanej w punkcie 3.2 oraz przedstawionej na Rysunku 6.
- `MultitenancyDbContext` – zawiera encje, które znajdują się w osobnych bazach (lub schematach) dzierżawców.

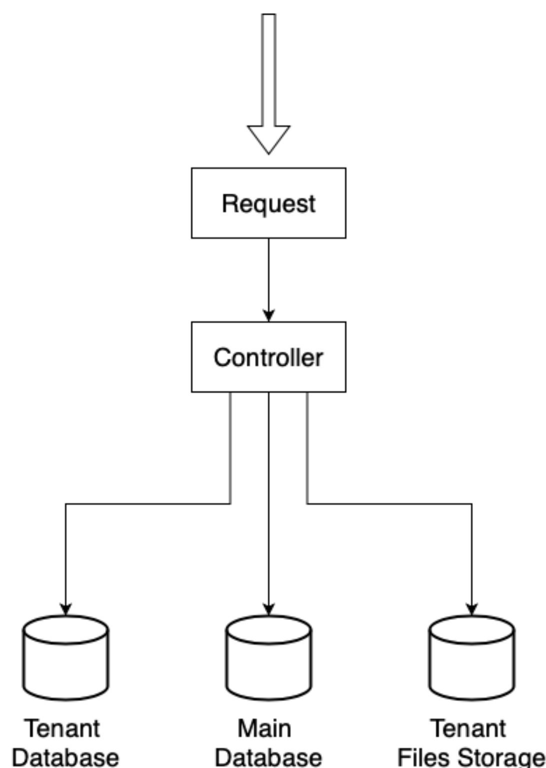
Interfejs `ITenant` implementowany jest tutaj przez klasę `Company`, ponieważ w tym przykładzie to firma jest dzierżawcą. Dlatego też `DbContext`, który zawiera w sobie `Companies` implementuje interfejs `ITenantStorage<Company>`.

5.4.2 Wzorce projektowe

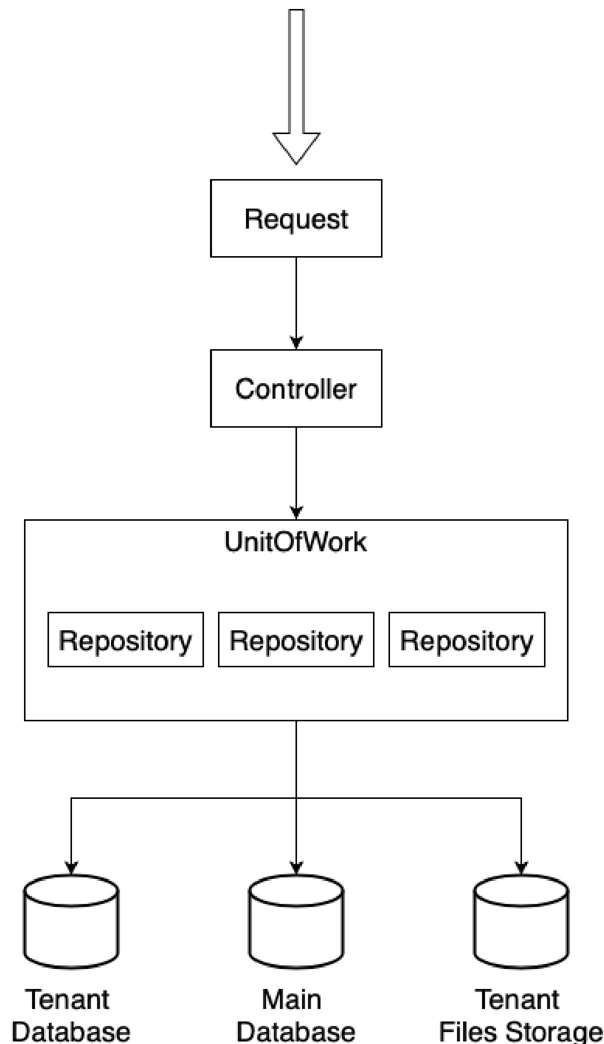
Dla ułatwienia pracy nad projektem użyty został wzorzec projektowy **Jednostki pracy (ang. *UnitOfWork*)** [10]. Jest to wzorzec, który pomaga w pracy z wieloma repozytoriami oraz zbiera operacje z kilku repozytoriów i wysyła je w formie jednej transakcji do bazy danych.

Repozytorium (ang. *Repository*) [16] to wzorzec, który dostarcza interfejs do komunikacji ze źródłem danych. W projekcie nie zostały stworzone osobne interfejsy, gdyż na potrzeby pracy wystarczyło użycie klasy DbSet [17].

Wzorce jednostki pracy oraz repozytorium bardzo dobrze ze sobą współgrają. Są one przeznaczone do tworzenia warstwy abstrakcji między warstwą danych i warstwą logiki biznesowej. Użycie tych wzorców izoluje aplikację od zmian w bazie danych. Ponadto takie rozwiązanie pozwala na jednoczesny zapis wszystkich zmian również w przypadku wielu ujść danych. Nie ma konieczności wykonywania zapisów do każdej z baz z poziomu kontrolera. Unikamy sytuacji przedstawionej na Rysunku 30. Poprawne działanie wzorca jednostki pracy wraz z wzorcem repozytorium przedstawia Rysunek 31.



Rysunek 30. Zapis danych do różnych ujść danych bez użycia wzorców projektowych



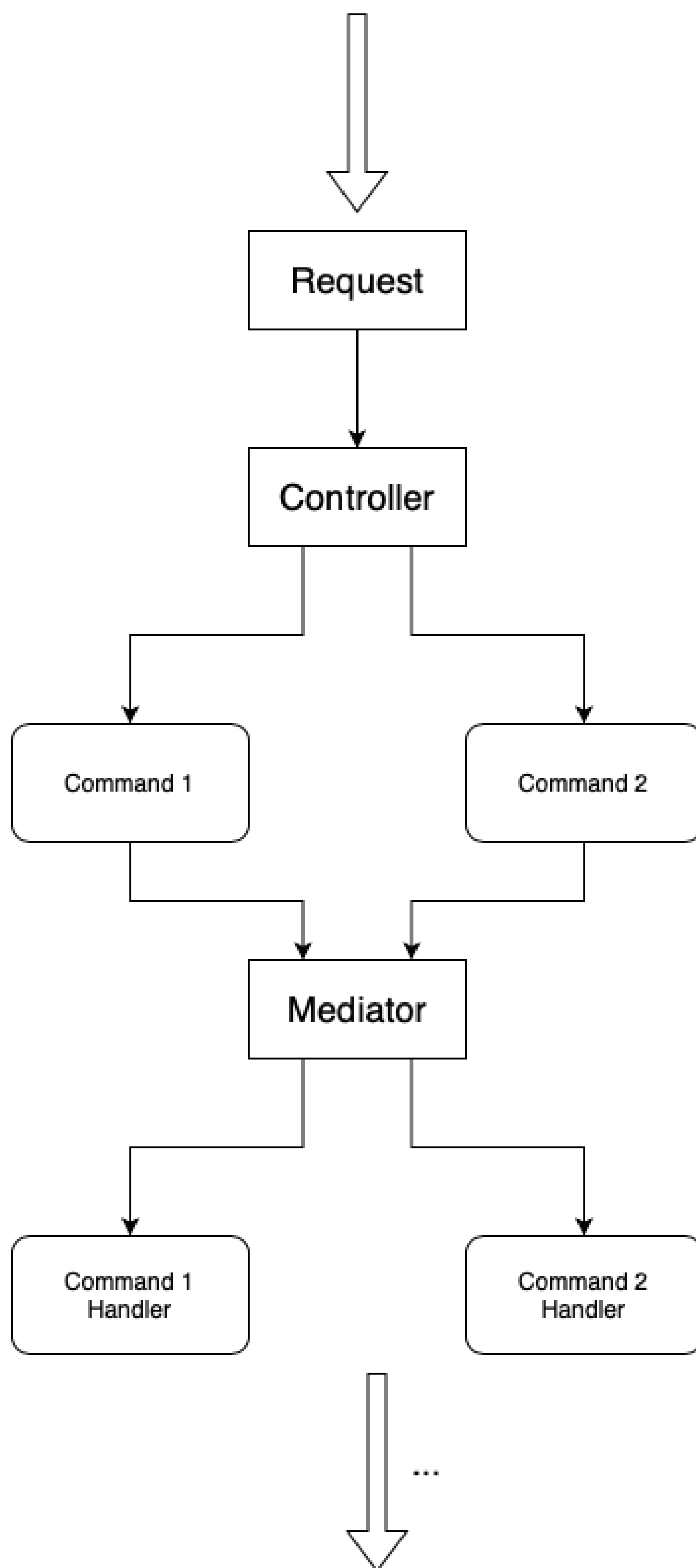
Rysunek 31. Zapis danych do różnych ujść danych z użyciem wzorców projektowych

Mediator [18] jest mechanizmem ułatwiającym komunikację z elementami podsystemu. Wykorzystuje do tego jednolity interfejs. W projekcie użyty został do komunikacji warstwy aplikacji z warstwą danych. Zapytanie przychodzące do serwera jest analizowane przez kontroler, a następnie tworzony jest rozkaz (ang. *Command*). Mediator zajmuje się dostarczeniem rozkazu do części kodu odpowiadającej za obsługę żądania (ang. *Handler*). Wybiera przy tym odpowiednią implementacje danego interfejsu. Rysunek 32 ilustruje działanie tego wzorca projektowego.

Przykładowo serwer dostaje żądanie dodania nowego sklepu do danej sieci. Mediator wybiera odpowiednią implementację interfejsu na podstawie wybranego trybu *multitenancy*:

- Jedna baza danych,
- Jedna baza danych z różnymi schematami dla każdego dzierżawcy,
- Wiele baz danych.

Żądanie trafia następnie do odpowiedniego *handler'a*, który przetwarza je w odpowiedni dla wybranego trybu sposób.



Rysunek 32. Wzorzec Mediator

5.4.3 Tryb Multitenancy

Powyższa architektura przedstawiona w punkcie 5.4.1 świetnie sprawdzi się zarówno w przypadku użycia osobnego schematu jak i osobnej bazy dla każdego dzierżawcy. Klasy biblioteki są przygotowane w taki sposób, aby działały poprawnie w obu przypadkach. Mechanizm ten jest dokładniej opisany w punkcie 5.2.3.

Trzeba jednak pamiętać o tym, że w przypadku użycia osobnej bazy dla każdej z firm należy podać parametry połączenia (ang. *Connection string*) podczas tworzenia obiektu reprezentującego dzierżawcę – takie pole zawiera interfejs `ITenant` przedstawiony w punkcie 5.2.1.

Istnieje również możliwość udostępnienia publicznie dostępu danemu dzierżawcy do właściwości `ConnectionString`. Daje to ogromne możliwości, ponieważ wtedy użytkownik może podać parametry połączenia do bazy danych znajdującej się na jego własnym serwerze. Jeżeli baza danych będzie miała odpowiedni format to klient będzie mógł bez żadnego problemu przetrzymywać swoje dane na własnym serwerze.

5.4.4 Użycie w projekcie

Aby móc używać architektury wielodostępnej w projekcie należy dodać serwisy, klasy oraz interfejsy za pomocą odpowiednich rozszerzeń w domyślnie istniejącej metodzie `ConfigureServices` w klasie `Startup`. Należy użyć kodu przedstawionego jako listing 20.

Listing 20. Dodanie biblioteki Multitenancy do projektu – ConfigureServices

```
services.AddDbContext<MultiDbContext>();
services.AddMultiTenancy<Company>()
    .WithStore<MultiDbContext>()
    .WithResolutionStrategy<HeaderResolutionStrategy>();
services.AddDbContext<MultiDbMultitenancyContext>();
services.AddScoped<IUnitOfWork, MultiDbUnitOfWork>();
```

Jak widać użyte zostały tutaj klasy przedstawione na diagramie na Rysunku 27. Zastosowana została również metoda znajdująca się w bibliotece Multitenancy – `AddMultiTenancy`. Pozwala ona na zdefiniowanie implementacji `ITenant`, `ITenantStorage` oraz `ITenantResolutionStrategy`.

Należy również dodać warstwę pośrednią `TenantMiddleware` opisaną w punkcie 5.2.6. Aby to zrobić należy dodać do domyślnie istniejącej metody `Configure` w klasie `Startup` kod przedstawiony jako listing 21.

Listing 21. Użycie warstwy pośredniej TenantMiddleware

```
app.UseMultiTenancy<Company>();
```

Warto również użyć filtra `TenantFilter` opisanego w punkcie 5.2.7. Aby to zrobić należy dodać do kontrolera API filtr. Przykładowy kontroler z dodanym filtrem przedstawia listing 22.

Listing 22. Użycie TenantFilter

```
[ApiController]
[Route("api/users")]
[ServiceFilter(typeof(TenantFilter<Tenant>))]
public class UserController : BaseController
{
 …
}
```

W tych kilku prostych krokach można dodać tak znaczącą bibliotekę, która może mieć ogromny wpływ na architekturę tworzonego system.

5.4.5 Migracje

Migracje z użyciem Entity Framework umożliwiają aktualizowanie schematu bazy danych w celu utrzymania jej synchronizacji z modelem danych przy jednoczesnym zachowaniu istniejących danych w bazie [19]. O ile w przypadku jednej bazy z jednym schematem nie jest to zazwyczaj problematyczna kwestia to w sytuacji, gdy ma się do czynienia z wieloma schematami nie jest to takie trywialne. Do tego celu zostały stworzone:

- Klasa `MigrationMultitenancyContext` dziedzicząca po klasie `MultitenancyDbContext`. Służy ona do przeprowadzania migracji. Używa określonej wartości jako zmiennej nazwy schematu w skrypcie migracji. Podczas tworzenia migracji wymagane jest podanie odpowiedniej implementacji `DbContext` – należy podać klasę stworzoną do migracji. Kod klasy przedstawiony jest jako listing 23.
- Z nowo stworzonej migracji generowany jest skrypt SQL, który następnie jest użyty do tworzenia nowych schematów oraz migracji danych w poszczególnych schematach dzierżawców.
- W bibliotece Multitenancy istnieje rozszerzenie `MigrateMultitenancyDatabase` (listing 24 przedstawia kod metody migrującej), które używa wygenerowanych skryptów SQL i aplikuje je na bazie. Funkcja podmienia stałą zmienną zawierającą nazwę schematu na `Id` istniejących obiektów opisujących dzierżawcę. Do tego celu używa klasy `SchemaFactory` przedstawionej jako listing 25.

Listing 23. Klasa MigrationMultitenancyContext

```
public class MigrationMultitenancyContext : MultiDbMultitenancyContext
{
    public MigrationMultitenancyContext(IConfiguration configuration,
TenantService<Company> tenantService)
        : base(configuration, null)
    {
        Schema = Consts.TemplateSchemaName;
    }
}
```

Listing 24. Metoda migrująca MigrateMultitenancyDatabase

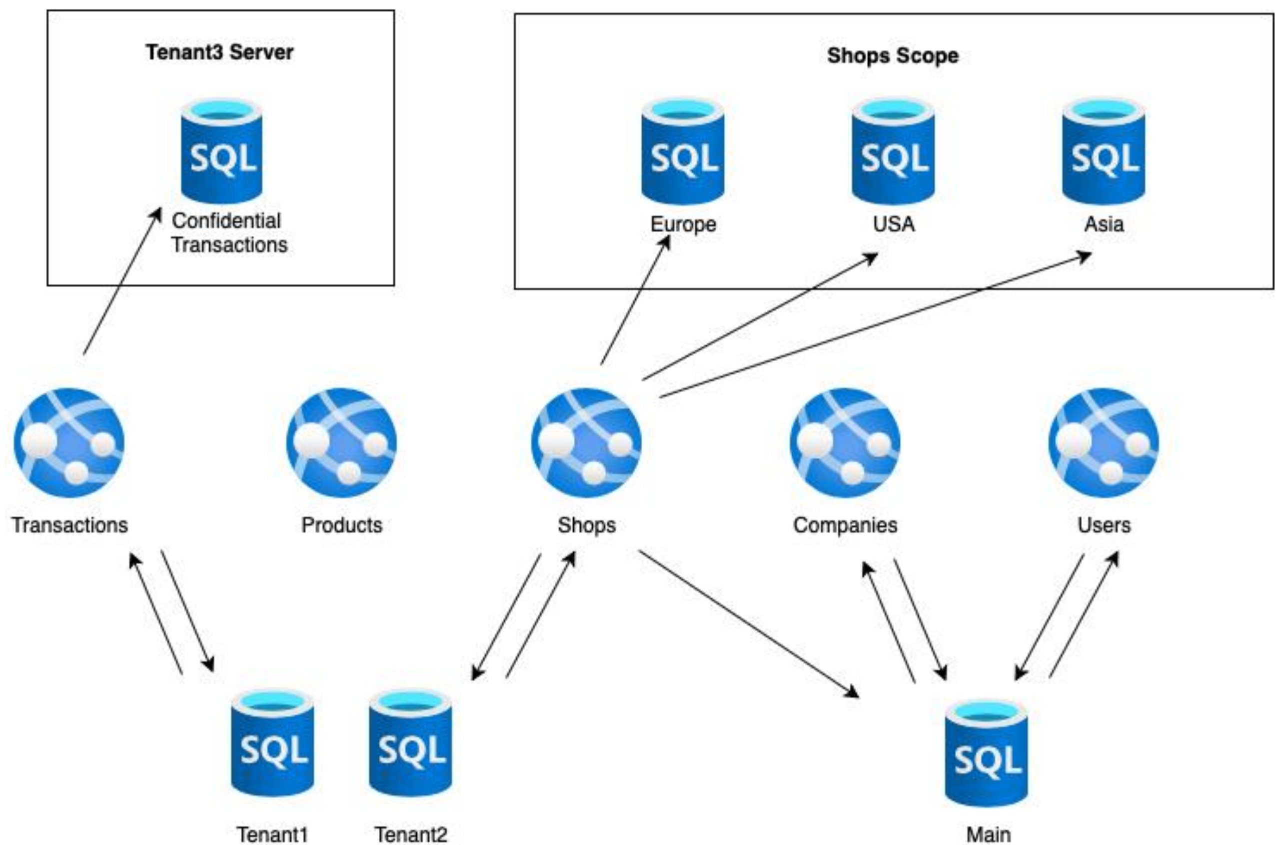
```
public void MigrateMultitenancyDatabase(IApplicationBuilder
applicationBuilder, string sqlPath)
{
    using var serviceScope =
applicationBuilder.ApplicationServices.GetService<IServiceScopeFactory>()?.
CreateScope();

    if (serviceScope == null)
    {
        return;
    }

    var multitenancyContext =
serviceScope.ServiceProvider.GetRequiredService<V>();
    var tenantStorage =
serviceScope.ServiceProvider.GetRequiredService<U>();

    if (multitenancyContext == null || tenantStorage == null)
    {
        return;
    }

    var tenants = tenantStorage.GetTenants();
    tenants.ForEach(tenant =>
SchemaFactory.CreateSchema(multitenancyContext, tenant.Id.ToString(),
sqlPath));
}
```

Listing 25. SchemaFactory

```
public static class SchemaFactory
{
    public static void CreateSchema(DbContext dbContext,
                                    string schemaName, string sqlFilePath)
    {
        var sql = File.ReadAllText(sqlFilePath).Replace("GO", "");
        var finalSql = sql.Replace(Consts.TemplateSchemaName, schemaName);
        dbContext.Database.ExecuteSqlRaw(finalSql);
    }
}
```

6. Architektura wielodostępna w systemie rozproszonym

Rozwiązanie świetnie wpisuje się w coraz bardziej popularne systemy rozproszone działające w oparciu o wiele mikroserwisów. Daje możliwość dowolności zapisu danych nie tylko do osobnych baz organizacji, ale nawet do osobnej bazy dla każdego z serwisów. Wiąże się to oczywiście ze zwiększonym poziomem trudności utrzymania systemu oraz rosnącymi kosztami. Jednak, w niektórych przypadkach takie rozwiązanie może okazać się niezbędne. Rysunek 33 przedstawia przykładowe zastosowania architektury wielodostępnej w systemie rozproszonym.

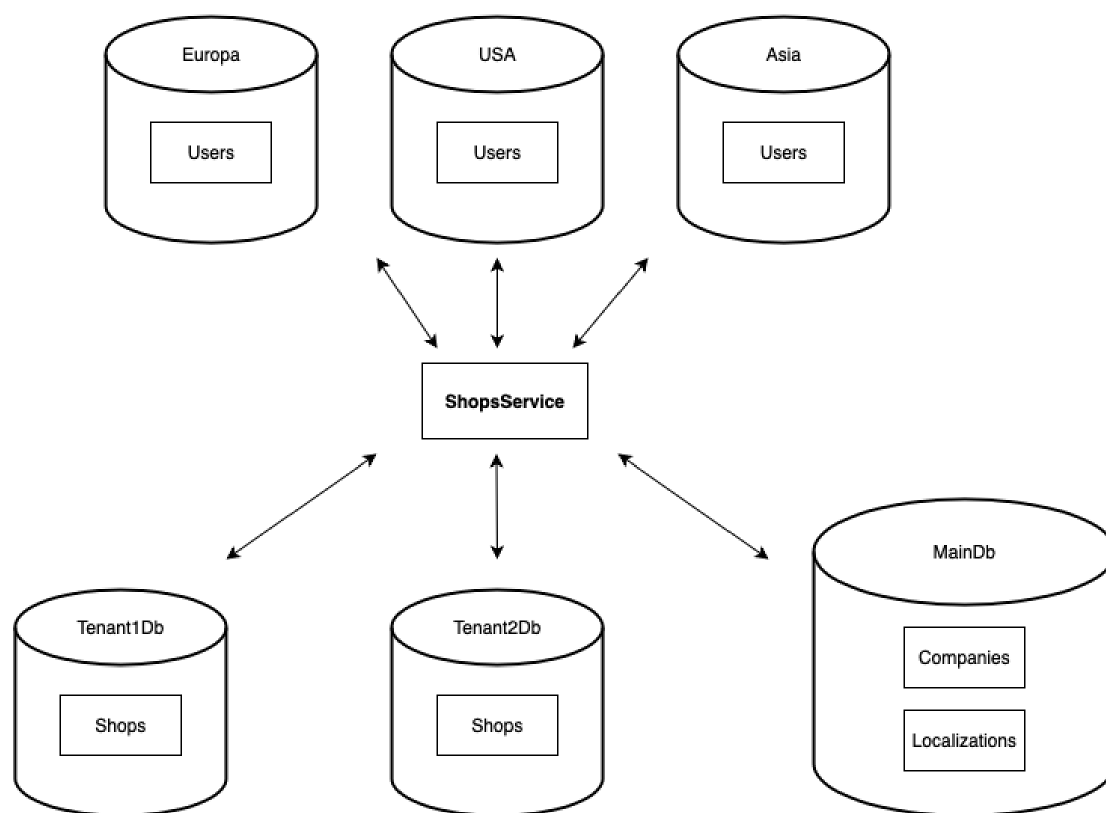


Rysunek 33. Przykładowe zastosowania architektury wielodostępnej w systemie rozproszonym

W przykładzie (Rys. 33) przedstawione zostało kilka serwisów. Jak widać jest wiele zastosowań architektury wielodostępnej. Przykładowo serwis „Transactions" korzysta z baz przeznaczonych dla „Tenant1" i „Tenant2" oraz z bazy z tajnymi transakcjami znajdującej się na serwerze „Tenant3". W serwisie „Shops" natomiast system został wykorzystany w inny sposób. Dzierżawcą jest tutaj nie tylko firma, ale również lokalizacja. Założono, że *tenant* – lokalizacja, potrzebny nam jest tylko w przypadku sklepów. Dzięki zastosowaniu architektury rozproszonej jest to możliwe.

6.1. Wielu dzierżawców

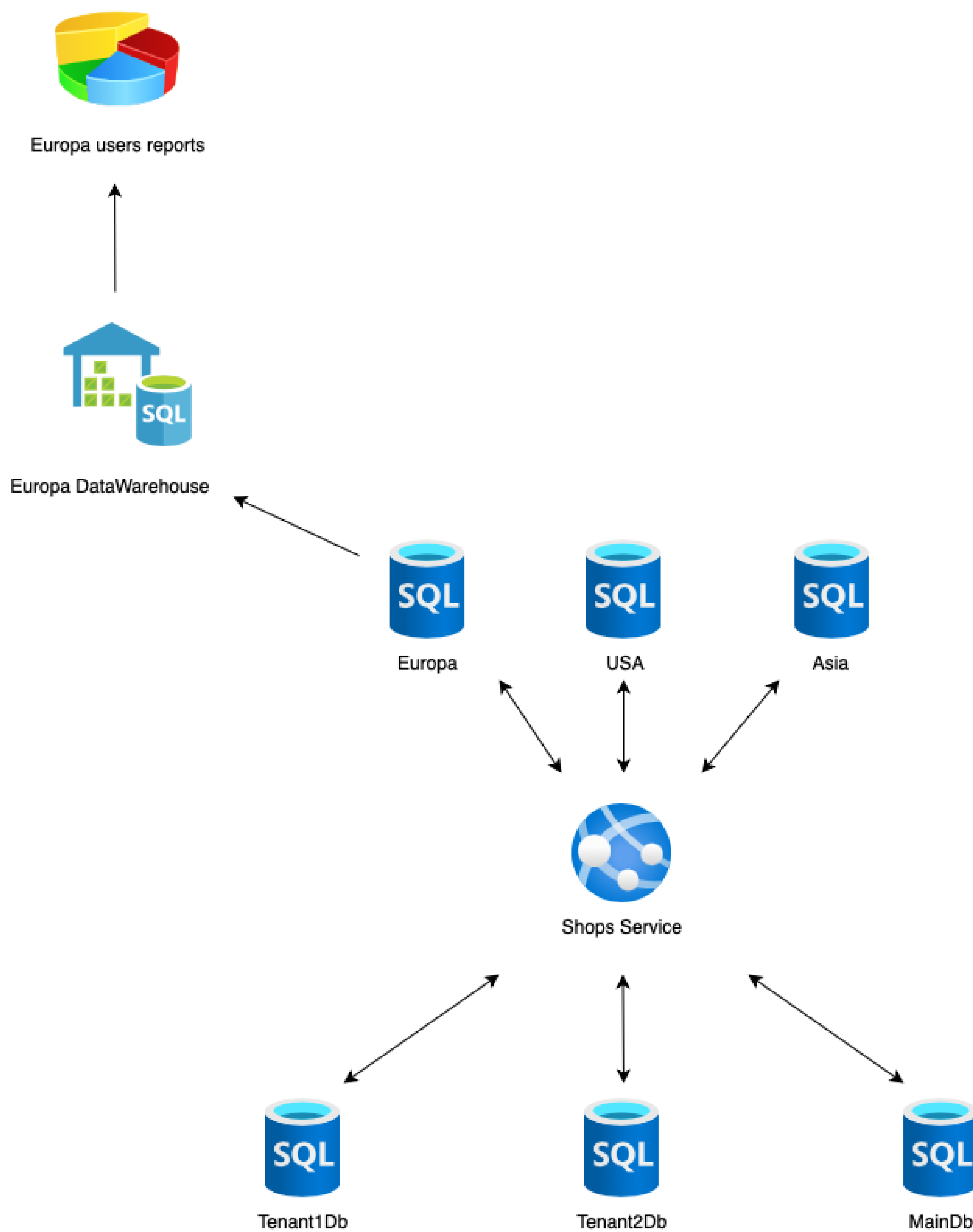
Jak zostało przedstawione na Rysunku 33 istnieje możliwość dodania więcej niż jednego dzierżawcy do poszczególnego serwisu. Daje to nowe możliwości. Można w ten sposób kategoryzować poszczególne encje. Na przykładzie serwisu „Shops” widać, że *tenant'em* może być tam zarówno „Company” jak i lokalizacja - „Localization”. Można zatem stworzyć różne lokalizacje takie jak np. Europa, USA i Azja. Dzięki temu będzie możliwość np. podzielić użytkowników aplikacji na odpowiednie regiony. Użytkownicy z Europy będą w bazie danych Europa itd. Przykładowe rozmieszczenie danych zostało przedstawione na Rysunku 34.



Rysunek 34. Poglądowy rysunek przedstawiający przykładowe rozmieszczenie danych w serwisie „Shops”

W dodatku serwery poszczególnych lokalizacji powinny znajdować się w odpowiednich miejscach np. USA w Stanach Zjednoczonych itd. Przyspieszy to działanie i pozytywnie wpłynie na jakość połączenia.

Taki przykład może mieć zastosowanie w przypadku, gdy będzie konieczność analizy użytkowników aplikacji, korzystających ze sklepów, w różnych regionach świata. Należy wziąć pod uwagę sytuację, w której, europejski oddział chce uzyskać informacje wyłącznie o klientach z Europy oraz analizować ich zwyczaje zakupowe. Do badań nie są potrzebni użytkownicy z innych kontynentów, gdyż mogą oni zaburzyć wyniki, które zostaną wykorzystane do stworzenia lepszych modeli marketingowych w Europie. Taka sytuacja została przedstawiona na Rysunku 35.



Rysunek 35. Poglądowy rysunek przedstawiający pomysł na wykorzystanie lokalizacji jako *tenant*

Inne serwisy mogą zachowywać się całkowicie inaczej, gdyż *tenant* może być izolowany względem serwisu. Przedstawiony przykład jest jedynie poglądowym pomysłem na wykorzystanie możliwości jakie daje ta architektura.

7. Podsumowanie

Praca magisterska analizuje architekturę systemu opartego o oprogramowanie wielodostępne. Wykorzystuje do tego przykładowy projekt systemu sprzedażowego dla wielu firm stworzony na potrzeby oceny poszczególnych rozwiązań. Praca skupia się na problemie architektury oraz sposobie przechowywania danych poszczególnych dzierżawców korzystających z systemu. Autor pracy wskazuje na kluczowe zalety oraz wady konkretnych podejść do problemu. Wyróżnia on trzy sposoby realizacji:

- Jedna baza danych,
- Osobna baza danych dla każdej z firm,
- Jedna baza danych z osobnymi schematami bazy dla każdej z firm.

Każde z powyższych rozwiązań zostało przeanalizowane pod kątem elastyczności, dostępności, użyteczności oraz skalowalności. Uwzględniony został również nakład pracy, bezpieczeństwo oraz trudność utrzymania projektu opartego o konkretną architekturę.

Jedna baza danych

Zalety:

- Prosty system,
- Łatwy w utrzymaniu,
- Niskie koszty wytworzenia oraz utrzymania.

Wady:

- Kod podatny na błąd programisty,
- Duże ryzyko ujawnienia danych poszczególnych dzierżawców,
- Brak spójności niektórych danych w tabelach,
- Dostęp do danych wszystkich firm w przypadku ataku hakerskiego,
- Ilość danych w bazie oraz jej wydajność zależna od innych firm.

Wniosek:

Rozwiązanie dla mniejszych projektów jako prototyp lub MVP.

Osobna baza danych dla każdej z firm

Zalety:

- Dobra izolacja danych,
- Lepsza wydajność poszczególnych elementów systemu,
- Małe ryzyko ujawnienia danych,
- Duża elastyczność systemu,
- Możliwość przechowywania danych na serwerze klienta,
- Bezpieczeństwo danych.

Wady:

- Skomplikowany projekt,
- Kosztowne oraz trudne utrzymanie systemu,
- Integracja danych.

Wniosek:

Rozwiązanie dla dużych systemów biznesowych z uzasadnieniem użycia skomplikowanych struktur oraz infrastruktury chmurowej.

Jedna baza danych z osobnymi schematami bazy dla każdej z firm

Zalety:

- Izolacja danych na poziomie jednej bazy danych,
- Zminimalizowana szansa na wyciek danych spowodowany błędem programisty,
- Niższe koszty wytworzenia oraz utrzymania systemu.

Wady:

- Dość skomplikowana warstwa danych pomimo użycia jednej bazy danych,
- Migracja danych,
- Integracja danych,
- Dostęp do danych wszystkich firm w przypadku ataku hakerskiego,
- Ilość danych w bazie oraz jej wydajność zależna od innych firm.

Wniosek:

Rozwiązanie dla średniej wielkości projektów, przygotowanie projektu do dalszej rozbudowy.

Szeroko opisany został również temat integracji danych. Okazał się on sporym problemem podczas tworzenia projektu opartego o bibliotekę Multitenancy w szczególności dla przypadku osobnych baz danych dla dzierżawców z uwzględnieniem utrzymywania baz na serwerach firm. Ze względu na poziom komplikacji tego zagadnienia autor postanowił dokładniej zbadać ten temat oraz przedstawić każdy krok analizy danych w takim systemie. Zastosowana do tego celu została hurtownia danych. Cały proces został wnikliwie opisany. Wygenerowane zostały następujące raporty:

- Ilość transakcji wykonanych przez użytkownika w sklepach danej firmy,
- Ilość pieniędzy wydanych przez użytkownika na produkty danej kategorii,
- Ilość transakcji wykonanych przez użytkownika w danym mieście,
- Najczęściej kupowany rozmiar produktu,
- Najwięcej sprzedanych produktów danej kategorii (Firma),
- Najwięcej sprzedanych produktów danej kategorii (Sklep),
- Sprzedaż produktów danej kategorii w miastach.

Raporty potwierdzają możliwość oraz zasadność integracji danych w systemach opartych o oprogramowanie wielodostępne.

W pracy przedstawiono również autorską uniwersalną bibliotekę Multitenancy dla projektów .NET napisaną w języku C#. Została ona stworzona, aby lepiej zwizualizować i potwierdzić wyniki badań. Biblioteka pozwala na implementację architektury wielodostępnej w dowolnym projekcie w języku C# w kilku prostych krokach, które zostały jasno zaprezentowane. Autor pracy przedstawił najważniejsze zagadnienia związane z biblioteką takie jak:

- Opis,
- Architektura,
- Opis poszczególnych klas, interfejsów oraz funkcji,
- Autoryzacja.

Został również przedstawiony proces implementacji biblioteki w projekcie. Tutaj również temat został rozszerzony o dodatkowe elementy:

- Propozycja architektury projektu,
- Opis rozszerzeń biblioteki pozwalających na dodanie jej za pomocą kilku linijek kodu,
- Tryby *multitenancy*,
- Migracje danych dla bazy danych korzystającej z wielu schematów.

Powyższe zagadnienia zostały dokładnie opisane. Załączone również zostały fragmenty kodu pokazujące najważniejsze elementy systemu.

Poruszony został również temat architektury wielodostępnej w systemach rozproszonych. Pokazane zostały różnie niestandardowe możliwości użycia oprogramowania wielodostępnego:

- *Tenant* nie musi być firmą, można wybrać dowolną encję,
- Jeden serwis/projekt może posiadać wiele implementacji *tenant'a*,
- Każdy z serwisów może mieć innego dzierżawce,
- Serwisy mogą również współdzielić dzierżawców i ich zasoby.

Rozdział dotyczący architektury serwisów w systemach rozproszonych pokazuje jak wiele zastosowań i możliwości daje użycie architektury wielodostępnej.

Ostatecznie powstał dokument, który szczegółowo opisuje architekturę wielodostępną, jej wyzwania oraz proponowane rozwiązania. Praca jest obszernym źródłem wiedzy na temat systemów opartych o oprogramowanie wielodostępne i zdecydowanie znajdzie zastosowanie w wielu systemach informatycznych.

Prace cytowane oraz źródła wiedzy

[1]. https://www.msadvisory.com/china-data-privacy-laws/ (Dostęp: 14.11.2022)

[2]. Vincent Rainardi . Building a Data Warehouse: With Examples in SQL Server (Expert's Voice) 1st ed. Edition; ISBN: 1590599314 (2008)

[3]. https://edu.pjwstk.edu.pl/wyklady/hur/scb/wyklad3/w3.htm (Dostęp: 14.11.2022)

[4]. https://edu.pjwstk.edu.pl/wyklady/hur/scb/wyklad3/w3.htm (Dostęp: 14.11.2022)

[5]. https://learn.microsoft.com/pl-pl/azure/data-factory/ (Dostęp: 14.11.2022)

[6]. https://learn.microsoft.com/en-us/analysis-services/tools-and-applications-used-in-analysis-services?view=asallproducts-allversions (Dostęp 05.12.2022)

[7]. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Financial Times Prentice Hall; Edycja 1; ISBN: 9780132350884 (2008)

[8]. https://learn.microsoft.com/pl-pl/ef/core/dbcontext-configuration/ (Dostęp 11.12.2022)

[9]. https://learn.microsoft.com/en-us/ef/core/modeling/dynamic-model (Dostęp 11.12.2022)

[10]. Robert C. Martin. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Financial Times Prentice Hall; Edycja 1; ISBN: 0134494164 (2017)

[11]. Jon Skeet. C# in Depth: Fourth Edition. Manning Publications; Edycja 4th ed.; ISBN: 9781617294532 (2019)

[12]. https://learn.microsoft.com/pl-pl/dotnet/api/system.web.httpcontext (11.12.2022)

[13]. https://learn.microsoft.com/pl-pl/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncactionfilter (Dostęp 11.12.2022)

[14]. Michał Bentkowski. Bezpieczeństwo aplikacji webowych. Securitum; ISBN: 9788395485305 (2019)

[15]. https://learn.microsoft.com/en-us/aspnet/core/security/authorization/policies (Dostęp 11.12.2022)

[16]. Eric Evans. Domain-Driven Design. Zapanuj nad złożonym systemem informatycznym. Helion; ISBN: 9788328305250 (2015)

[17]. https://learn.microsoft.com/pl-pl/dotnet/api/system.data.entity.dbset-1 (12.12.2022)

[18]. Dmitri Nesteruk. Design Patterns in .NET: Reusable Approaches in C# and F# for Object-Oriented Software Design; ISBN: 9781484243664 (2019)

[19]. https://learn.microsoft.com/pl-pl/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli (Dostęp 12.12.2022)

Dodatki

Spis rysunków

Rysunek 1. Poglądowy diagram działania systemu 8
Rysunek 2. Scenariusze zapytań do serwera 9
Rysunek 3. Różnice pomiędzy „Single-Tenant”, a „Multi-Tenant” 10
Rysunek 4. Porównanie schematów tabel w bazach danych 10
Rysunek 5. Diagram związków encji – jedna baza 11
Rysunek 6. Diagram związków encji – osobna baza dla każdej firmy 13
Rysunek 7. Duplikaty danych w głównej bazie (osobna baza dla każdej firmy).................... 14
Rysunek 8. Pobranie listy transakcji z bazy głównej 15
Rysunek 9. Pobieranie transakcji z Id z bazy dzierżawcy 15
Rysunek 10. Pobranie listy transakcji z baz dzierżawców. 16
Rysunek 11. Schemat płatka śniegu - hurtownia danych 21
Rysunek 12. Zasoby na platformie Azure 22
Rysunek 13. Stworzone zasoby hurtowni danych 22
Rysunek 14. Źródła oraz ujścia danych hurtowni. 23
Rysunek 15. Przepływy danych dla tabeli „Main” oraz „Tenant1” 23
Rysunek 16. Złączenia danych dla przepływu danych transakcji 24
Rysunek 17. Potok przepływu danych dzierżawców do hurtowni danych 24
Rysunek 18. Struktura kostki OLAP 25
Rysunek 19. Transakcje użytkownika w sklepach danej firmy 26
Rysunek 20. Wydane pieniądze użytkownika na produkty danej kategorii 26
Rysunek 21. Transakcje użytkownika w danym mieście 27
Rysunek 22. Najczęściej kupowany rozmiar produktu 27
Rysunek 23. Najwięcej sprzedanych produktów danej kategorii (Firma) 27
Rysunek 24. Najwięcej sprzedanych produktów danej kategorii (Sklep) 28
Rysunek 25. Sprzedaż produktów danej kategorii w miastach 28
Rysunek 26. Diagram komponentów przedstawiający wykorzystanie biblioteki Multitenancy w wielu systemach 30
Rysunek 27. Diagram klas przedstawiający architekturę biblioteki Multitenancy 31
Rysunek 28. Struktura kodu biblioteki Multitenancy 32
Rysunek 29. Diagram klas przedstawiający architekturę projektu implementującego bibliotekę Multitenancy 41
Rysunek 30. Zapis danych do różnych ujść danych bez użycia wzorców projektowych 42
Rysunek 31. Zapis danych do różnych ujść danych z użyciem wzorców projektowych 43
Rysunek 32. Wzorzec Mediator 44
Rysunek 33. Przykładowe zastosowania architektury wielodostępnej w systemie rozproszonym 48
Rysunek 34. Poglądowy rysunek przedstawiający przykładowe rozmieszczenie danych w serwisie „Shops” 49
Rysunek 35. Poglądowy rysunek przedstawiający pomysł na wykorzystanie lokalizacji jako *tenant* 50

Spis listingów

Listing 1. Poprawny kod pobierający ceny produktów. 12
Listing 2. Błędny kod pobierający ceny produktów. 12
Listing 3. Interfejs ITenant 33
Listing 4. Enum MultitenancyType 33
Listing 5. Interfejs IDbContextSchema 33
Listing 6. Konstruktor klasy MultitenancyContextBase 34
Listing 7. Metoda OnConfiguring klasy MultitenancyContextBase 34
Listing 8. Metoda UseMySchema 35
Listing 9. Klasa CacheKeyFactory 35
Listing 10. TenantService 36
Listing 11. ITenantStorage 36
Listing 12. TenantMiddleware 37
Listing 13. TenantFilter 37
Listing 14. ITenantResolutionStrategy 38
Listing 15. HeaderResolutionStrategy 38
Listing 16. HostResolutionStrategy 39
Listing 17. QueryResolutionStrategy 39
Listing 18. TenantRequirement 40
Listing 19. Obsługa TenantRequirement - autoryzacja 40
Listing 20. Dodanie biblioteki Multitenancy do projektu – ConfigureServices 45
Listing 21. Użycie warstwy pośredniej TenantMiddleware 45
Listing 22. Użycie TenantFilter 46
Listing 23. Klasa MigrationMultitenancyContext 46
Listing 24. Metoda migrująca MigrateMultitenancyDatabase 47
Listing 25. SchemaFactory 47