**Politechnika Krakowska  
im. Tadeusza Kościuszki**

Wydział Fizyki, Matematyki i Informatyki

**Kraków 2017**

Praca wykonana pod kierunkiem:  
**dr inż. Stanisława Plichta**

**Uzgodniona ocena:**......................................

...........................................................................podpisy promotora i recenzenta





**Presentation of selected design patterns on the example of a Web application**

**Praca inżynierska   
 na kierunku Informatyka**

**Prezentacja wybranych wzorców projektowych i architektonicznych na przykładzie aplikacji webowej**

**Bartłomiej Zięba**

108742

**Spis treści**

1. **Wstęp i zakres pracy1**
2. **Opis użytych technologii4**
3. **Ogólny opis architektury oraz podejścia DDD4**
4. **Autoryzacja z wykorzystaniem Oauth2.04**
5. **Opis oraz implementacja CQRS4**
6. **Opis oraz implementacja Unit of Work4**
7. **Opis oraz implementacja Strategii4**
8. **Opis oraz implementacja Gateway4**
9. **Wzorce Front - end4**
10. **Wstęp**
11. **Cel i zakres pracy**

Opis celu i zakresu pracy….

1. **Opis użytych technologii**
   1. **Technologie Back-End**

Do implementacji serwerowej części aplikacji została wykorzystana technologia oparta na języku programowania C# oraz bibliotekach zawartych w .Net Framework, którego twórcą jest Microsoft. Za pomocą tej technologii zostały napisane wszystkie moduły składające się na aplikację serwerową. Za serwis REST odpowiadają biblioteki WebApi2, dostęp oraz obsługa zrealizowane są przy pomocy ORMa Entity Framework. Wysyłanie maili, komunikacja z Docusign, tworzenie logów oraz modele domeny również powstały w oparciu o język C# oraz technologię .Net.

Język C# jest to obiektowy język programowania, stworzony przez pracowników Microsoft. Do jego głównych cech należą obiektowość, automatyczne zarządzanie pamięcią, deletegaty oraz zdarzenia, system refleksji i atrybuty klas. W kolejnych wydaniach tego języka, C# doczekał się szeregu rozszerzeń, takich jak typy generyczne, zarządzanie wielowątkowością za pomocą „Tasków”, ułatwienie programowania asynchronicznego przy pomocy słów kluczowych async i await. Ponadto język ten posiada wsparcie w postaci technologii LINQ. LINQ umożliwia programiście tworzenie zapytań przypominających w składni SQL bezpośrednio na obiektach oraz ich kolekcjach. Posiada on szereg własnych zapytań umożliwiających w szybki i wygodny sposób zarządzać kolekcjami obiektów. W połączeniu z Entity Framework daje on potężne narzędzie, za pomocą którego obsługa bazy różnorakich źródeł danych jest niezwykle prosta i intuicyjna.

Entity Framework jest to biblioteka ORM (Object Relational Mapping) umożliwiająca odwzorowanie relacyjnej bazy danych przy pomocy modelu obiektowego. Jedną z głównych zalet tej biblioteki jest możliwość rozdzielenia warstw logiki biznesowej oraz warstwy dostępu do danych. Do operowanie na danych w Entity Framework wykorzystywana jest składnie LINQ z pomocą specjalnych bibliotek LINQ to Entities oraz LINQ to SQL. Wyróżniamy trzy różne podejścia do tworzenia bazy danych przy pomocy Entity Framework. Pierwszym z nich jest „Database First”. Najczęściej korzystamy z niego, kiedy baza, z której chcemy skorzystać już istnieje. Przy pomocy odpowiedniego narzędzia zawartego w Visual Studio jesteśmy odwzorować w obiektowym kodzie strukturę tabel, widoków oraz procedur stworzonych w podłączonej bazie danych. Kolejnym podejściem jest „Model First”. Jego główną zaletą jest uniknięcie przez programistę konieczności pisania jakiegokolwiek kodu SQL. Za pomocą narzędzia ADO.NET Entity Data Model Designer definiujemy odpowiedni model danych, który będzie posiadał zarówno odwzorowanie obiektowe w naszej aplikacji, jak i utworzy odpowiednią strukturę tabel oraz relacji w bazie danych. Ostatnim podejściem jest tak zwane podejście „Code First”. Programista najpierw tworzy klasy oraz powiązania między nimi. Następnie przy pomocy odpowiedniego silnika mapującego Entity Framework tworzy odpowiednie tabele oraz klucze główne i prywatne. Podejście to jest uważane za najlepsze, wpływa na to szereg czynników. Twórca takiego kodu modyfikując strukturę klas, ma praktycznie bezpośredni wpływ na strukturę bazy danych, z której korzysta. Ponadto zwolniony jest z pisania kodu SQL, który by odzwierciedlał zmiany jakie zaszły w czasie modyfikacji modelu domenowego w takiej aplikacji. Aby utrzymać synchronizację i jednoznaczność modelu danych, programista zmuszony jest tworzyć tak zwane „migracje” po każdej zmianie, jaka zaszła w modelu obiektowym. Migracja taka, generowana automatycznie, aktualizuje strukturę tabel. Ponadto programista ma możliwość dodawania własnej logiki, która wygeneruje dodatkowy kod SQL mający wpływ na bazę danych.

WebApi jest to framework stworzony na potrzeby platformy .Net umożliwiający sprawne i wydajne tworzenia serwisów REST. Zanim nastąpi prezentacja możliwości samego WebApi, przybliżona zostanie sama idea serwisów REST. REST, czyli z angielskiego Representational State Transfer jest zarówno pewnym wzorcem dobrych praktyk w tworzeniu architektury aplikacji rozproszonych, jak implementacją usług sieciowych, bazującą na protokole http. Oczywiście taka usługa musi spełniać założenia, jakie REST sam w sobie niesie. Główną ideą serwisów takiego typu, jest podzielenie komunikacji sieciowej, na cztery główne metody pełniące różne funkcje. I tak do pobierania danych z serwera korzysta się z metody GET, dodawanie danych następuje przy pomocy metody POST, za edycję danych odpowiada metoda PUT, a usuwaniem zajmuje się metoda DELETE. Taka klasyfikacja metod wiąże niepodzielnie metody http z typem operacji, jaką chcemy przeprowadzić na naszych danych po stronie serwera. Podejście takie posiada taką zaletę, że programista wywołujący którąś z wybranych metod http, może bezproblemowo określić jaka operacja zostanie wykonana.

**2.2 Technologie Fron-End**

## **3. Ogólny opis architektury oraz podejścia DDD**

**3.1. Domain Driven Design oraz Onion Architecture**

Domain Driven Design jest pewnym szczególnym podejściem do projektowania i wytwarzania oprogramowania. Przykłada ono szczególną uwagę do definiowania i modelowania obiektów i komponentów tak, aby te maksymalnie odzwierciedlały rzeczywistość, z którą powiązany jest projekt. Dzięki takiemu założeniu, programiści i architekci oprogramowania w znacznym stopniu mogą być odciążeni, od modelowania logiki biznesowej w danej aplikacji. Praca ta może być w znacznym stopniu wykonana przez ekspertów specjalizujących się w danym zagadnieniu. Struktury, które opisują tą rzeczywistość nazywamy domeną. Model takiej domeny jest jądrem całej aplikacji, wokół której powstają kolejne komponenty i interfejsy.

Jednym z częstych aspektów idących w parze z podejściem DDD jest pewien specyficzny podział aplikacji na osobne warstwy, które zawierają w sumie moduły o zbliżonych właściwościach. Taki podział często wiąże się z określeniem „architektura cebulowa (onion architecture)”. Każda z warstw pełni inną rolę w architekturze całej aplikacji. Najbardziej tradycyjnym podziałem jest występowanie warstwy domeny, aplikacji, infrastruktury oraz interfejsu użytkownika. Czasami wyróżniamy również warstwę integracji.

Najważniejszym elementem takiej architektury jest warstwa domeny. Komponenty które się nią składają, specyfikują struktury danych, które odzwierciedlają modelowaną rzeczywistość. Ponadto, warstwa ta może zawierać logikę biznesową, jaka jest związana z domeną. Preferowane jest jedna specyfikowanie logiki biznesowej w warstwie aplikacji. Domena powinna być lekka oraz posiadać minimalną zależność od bibliotek zewnętrznych. Jest ona jakby sercem całej architektury, wokół którego buduje się kolejne moduły. Bardzo istotne jest, żeby w początkowej fazie projektowania aplikacji jak najdokładniej przeanalizować i zamodelować strukturę domeny. Praktycznie każda późniejsza zmiana, jaka zajdzie w tej strukturze, może być poważną przyczyną rozległych zmian w innych komponentach.

Następnym elementem „onion architecture” jest tak zwana warstwa aplikacji. W aplikacjach webowych służy ona generalnie do dwóch celów. Pierwszym jej zadaniem jest implementacja logiki biznesowej, jaką niesie za sobą domena. To tutaj programiści definiują główne procesy operujące na strukturach zawartych w najbardziej wewnętrznej warstwie. Drugim, nie mniej istotnym, jest bycie spoiwem pomiędzy poszczególnymi modułami warstwy infrastruktury. Dążymy do tego, aby każdy z poszczególnych elementów infrastruktury był maksymalnie od siebie niezależny, więc to tutaj powinna następować komunikacja między nimi. To również w tym miejscu następuje wymiana informacji między logiką, która była zdefiniowana w domenie, a konkretnymi modułami funkcjonalnymi z warstwy infrastruktury.

Warstwa infrastruktury służy nam do przygotowania bibliotek, które następnie będą wykorzystywane przez warstwę aplikacji, aby zrealizować logikę biznesową danego systemu. Moduły tworzące taką infrastrukturę mogą pełnić wielorakie działania, w zależności od funkcjonalności jakie musi zapewnić dana aplikacja. Jako przykłady mogą, służyć elementy aplikacji zajmujące się takimi funkcjonalnościami jak komunikacja ze środowiskiem zewnętrznym z wykorzystaniem REST, wysyłanie maili, dostęp do bazy danych, zapewnienie autoryzacji i autentykacji. Każdy z tych elementów służy do zapewnienia pewnej funkcjonalności, która potem zostanie wykorzystana na rzecz logiki biznesowej. Istotne jest, aby minimalizować zależności pomiędzy konkretnymi komponentami w tej warstwie. W idealnej sytuacji, żaden komponent z warstwy infrastruktury nie posiada bezpośredniej zależności z innym. Dzięki temu, w późniejszym czasie programista może prościej zmienić sposób, w jaki jest wykonywana dana funkcjonalność, bez znacznej ingerencji w pozostałe elementy aplikacji. Zmiana taka zachodzi wtedy tylko na warstwie aplikacji, która spina ze sobą całą infrastrukturę.

Jeżeli w danej aplikacji integrujemy się z wieloma systemami zewnętrznymi, nad którymi nie mamy bezpośredniej kontroli, dobrym pomysłem jest wydzielenie z infrastruktury osobnego bytu – tak zwanej warstwy integracji. Moduły takiej warwy służą nam do definiowania i enkapsulacji sposobów komunikacji z odrębnymi systemami. W warstwie infrastrukturalnej zostają tylko te elementy, nad którymi mamy pełną władzę, jeśli chodzi o sposób ich działania. Głównym zadaniem tej warstwy, jest maksymalne uniezależnienie naszego produktu od zmian, jakie zachodzą w systemach, z którymi się integrujemy. Przykładem takiego uniezależnienia może być opakowanie metod wystawionych przez zewnętrznych klientów, w nasze własne metody, które zostaną potem wykorzystane na niższych warstwach. Dzięki takiemu podejściu, potencjalna zmiana w sposobie działania biblioteki klienckiej, będzie miała swoje następstwo tylko w module, który taką bibliotekę obsługuje. Cała reszta aplikacji nie powinna odczuć jakichkolwiek zmian w sposobie działania klienta. Idealnym przykładem takiego podejścia są wzorce projektowe Gateway lub Fasada, które w bardzo dobry sposób zapewniają nam jakże ważną niezależność. Zostaną one omówione w dalszej części pracy.

Do prezentacji danych dla użytkowników, służy warstwa UI (User Interfaces). W niej definiujemy sposób prezentacji danych oraz interakcji z użytkownikiem. Ponadto określamy tam również technologie, jakie zostaną do tego wykorzystane. W przypadku aplikacji webowych najczęściej są to wariacje HTML5 oraz różnych frameworków javascriptowych. Nic nie stoi jednak na przeszkodzie, żeby stworzyć klienta mobilnego lub desktopowego, który będzie komunikował się z serwerem, na którym stoi nasza aplikacja. Do komunikacji elementów warstwy UI z resztą aplikacji, służą odpowiednie komponenty zdefiniowane w warstwie infrastrukturalnej lub rzadziej – integracyjnej. W przypadku aplikacji webowej, zazwyczaj wykorzystuje się protokół http.

Miejsce na graf.

## **3.2. Command Query Responsibility Segregation**

CQRS jest to rozwinięcie idei stworzonej w latach osiemdziesiątych przez Bertranda Meyera o nazwie CQS (Command Query Separation). Idea ta głosiła, że zapytanie o pewne dane, nie powinno zmieniać odpowiedzi jaką otrzymamy. Sama logika rozdzielenia zapytań o dane do ich alteracji i zmian miała być zrealizowana poprzez osobne metody w danych klasach bądź komponentach. CQRS idzie o krok dalej. Po pierwsze rozdzielenie tych dwóch kluczowych funkcjonalności zachodzi już na poziomie nie metod, a klas. Po drugie, w przypadku CQRS często zachodzi dublowanie domeny – tworzymy przykładowo osobne modele do modyfikacji danych oraz do ich pobierania. Ponadto w ramach jednego typu operacji również może istnieć taki podział. Weźmy na przykład jakże często stosowany interfejs master-detail. Z poziomu użytkownika widzimy listę wielu rekordów (master) zawierającą podstawowe informacje o danej strukturze. Jednak dany wiersz listy można przekierować do widoku detail, gdzie dostajemy znacznie więcej szczegółów. W tradycyjnym podejściu do tworzenia domeny, posiadalibyśmy jeden model obrazujący informacje wyświetlane na obu widokach. Przy takim podejściu programista musiałby szukać kompromisu pomiędzy optymalizacją zapytania o dane w przypadku widoku master, a szczegółowością w przypadku widoku detail. Prowadziłoby to do psucia modelu – z jednej strony mógłby być zbyt mało szczegółowy, z drugiej zapytanie do bazy danych mogłoby być nadal zbyt czasochłonne. W podejściu CQRS tworzymy dwa osobne modele obrazujące ten sam byt, lecz wykorzystywane w różnych sytuacjach. Dla widoku master przygotujemy lekki model zawierający minimum informacji. Jeśli chcemy poznać pełną ilość danych wykorzystujemy zapytanie z bardziej szczegółową strukturą.

W bardziej szczegółowym przypadku traktujemy CQRS jak wzorzec architektoniczny, który składa się z kilku istotnych elementów. Większość jego logiki znajduje się na warstwie integracji, jednak ma on również wpływ na działanie pozostałych komponentów. W skład implementacji tego wzorca wchodzą takie elementy jak Command, CommandHandler, CommandHandlerInvoker oraz QueryHandler. Implementacja tych modułów jest ściśle związana z technologią, w jakiej została wykonana aplikacja. Zostanie to szczerzej omówione w opisie komponentów implementacji danego wzorca.

**3.3. Dependency Injection oraz Inversion of Controll**

Z pojęciem IoC (Inversion of Controll) nierozerwalnie wiąże się wzorzec architektoniczny i projektowy DI (Dependency Incjection), więc to jemu przyjrzymy się najpierw.

Wstrzykiwanie zależności ma na celu usuwanie bądź minimalizowanie bezpośrednich zależności pomiędzy komponentami na rzecz asocjacji w ramach klas. Klasa taka sama nigdy nie tworzy obiektów, które będzie wykorzystywała w czasie swojego działania. Obiekty te wstrzyknięte zostają z zewnątrz, przez co programista zyskuje kontrolę nad działaniem takiej klasy. Wyróżniamy kilka różnych sposobów na wstrzykiwanie zależności.

Pierwszy z nich to wstrzykiwanie przez konstruktor. W konstruktorze danej klasy definiujemy jakiego typu obiektów będziemy potrzebować do jej działania. Następnie w momencie tworzenia jej instancji, dostarczamy obiekty o zdefiniowanych przez nas właściwościach. Daje nam to wpływ na sposób, w jaki dana klasa będzie spełniać swoje działanie. Nietrudno sobie wyobrazić sytuację, w której wstrzyknięcie obiektów o różnych parametrach, zapewni nam różnorakie działanie takiej klasy.

Drugą metodą stosowania dependency injection, jest wykorzystanie tak zwanej metody ustawiającej bądź mutującej. Zasada działania jest podobna jak w przypadku konstruktora, w parametrach metody podajemy parametry, które ustawią w wybrany przez nas sposób naszą klasę. Zaletą tego sposobu jest fakt, że może być on zastosowany wielokrotnie dla jednej instancji danej klasy w czasie działania programu. Powoduje to, że jeden obiekt w czasie swojego życia, może wykonać swoje operacje na różne sposoby. Wadą takiego rozwiązania jest fakt, że pomiędzy utworzeniem instancji danej klasy a uruchomieniem metody mutującej, część parametrów definiujących jej zachowania jest w stanie nieokreślonym. Może to powodować problem z działaniem takiego obiektu. Aby zapobiec takiemu przypadkowi istnieje możliwość ustawienia stanu początkowego obiektu poprzez konstruktor.

Trzecią metodą na wstrzykiwania zależności jest wykorzystanie interfejsów. Metodę tę można rozumieć dwojako. Po pierwsze sama implementacja danego interfejsu wymaga na autorze danej klasy, aby zrealizował pewną funkcjonalność. Po drugie wykorzystując kontenery DI (lub szerzej IoC) jesteśmy w stanie spiąć ze sobą dany interfejs z jego wybraną implementacją. Nic nie stoi na przeszkodzie, aby w czasie działania aplikacji podmieniać daną implementację na inną. Dzięki temu z zewnątrz sprawujemy kontrolę nad danym zachowaniem. Ponadto wstrzykiwanie przez interfejs bardzo dobrze współpracuje z konstruktorami. Te zamiast przyjmować jako parametr typy danych obiektów, powinny wymagać tylko interfejsów, jakie te typy implementują.

W powyższych akapitach została poruszona kwestia sterowania z wykorzystaniem DI. Właśnie sterowanie i zależności są istotą IoC. Celem programisty jest uniezależnienie modułów z warstw wyższych, od działania komponentów z warstw niższych. Dzięki temu, twórca systemu ma większą władzę nad sposobem w jaki pracują dane komponenty oraz może w odpowiedni sposób je konfigurować. Ponadto dzięki luźnym wiązaniom, które zapewnia nam Dependency Injection, w stosunkowo bezbolesny sposób jesteśmy w stanie podmieniać dane klasy, a nawet całe komponenty na nowe, bez wchodzenia w implementację modułów, które z nich korzystają.

W przypadku stosowania praktycznie każdego z nowoczesnych frameworków służących do łatwiejszej implementacji IoC, spotkamy się z modułami służącymi do konfiguracji zależności, fachowo zwanymi kontenerami IoC bądź kontenerami DI. W takich kontenerach rejestrujemy w jaki sposób mają przebiegać powiązania między konkretnymi klasami. Ponadto taki kontener zajmuje się dostarczaniem odpowiedniej implementacji interfejsów do wykorzystujących je komponentów.

## **4. Autoryzacja z wykorzystaniem Oauth2.0**

W rozdziale tym zostanie poruszona kwestia autoryzacji oraz autentykacji ze szczególnym naciskiem na omówienie standardu Oauth 2.0 oraz jego implementacji w technologii .Net. Pokazane zostanie również, jak taka implementacja współgra z pozostałymi modułami oraz całą architekturą aplikacji.

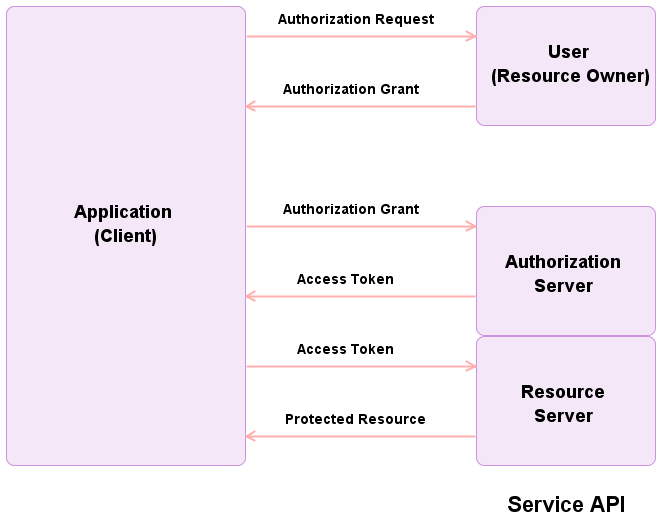
Oauth jest to standard autoryzacji, który pozwala uzyskać ograniczony dostęp zasobów użytkownika, poprzez delegowania operacji autoryzacji do trzeciego komponentu, przechowującego informacje o koncie użytkownika. Komponent ten po otrzymaniu odpowiednich informacji w czasie logowania, mogą to być przykładowo login i hasło, zwraca zaszyfrowany token. Następnie token ten, służy jako przepustka do wybranych przez nas zasobów. Token ma wcześniej zdefiniowany czas życia, po którym wygasa i nie da się już za jego pomocą zautoryzować. Dzięki standardowi oauth można wykorzystać konta takich serwisów jak Facebook, Google czy GitHub do autentykacji w danym systemie oraz uzyskać wybrane dane, które zostały udostępnione przez te serwisy.

Standard ten posiada kilka ról, które odpowiadają za konkretny fragment działania całego systemu autentykacji i autoryzacji. Wyróżniamy takie pojęcia jak Resuource Owner, Client, Resource Server oraz Authorization Server. W zależności od implementacji, dany fragment aplikacji może pełnić odrębną rolę bądź kilka ról na raz. Oba podejścia są zgodną z samą ideą standardu Oauth. Poniżej zostanie przybliżone ogólne znaczenie każdej z ról.

Resource Owner, czyli właściciel zasobu, jest to użytkownik, który wysyła żądanie o dostęp do własnego konta poprzez aplikację. Zakres dostępu do zasobów danego konta jest ograniczony do zezwoleń, na jakie zdecydował się użytkownik.

Resource Server zawiera zasoby, z których dany użytkownik chce skorzystać. Mogą to być zarówno dane jak i różnorakie funkcjonalności świadczone przez aplikację. Authorization Server weryfikuje tożsamość użytkownika, a następnie przydziela tokeny dostępu do zasobów. W przypadku omawianej aplikacji obie role, zostały zaimplementowane jako jedno api, na jednym serwerze, posiadające odpowiednie metody wystawiające zarówno zasoby jak i pozwalające na autentykacje.

Client jest aplikacją, która chce uzyskać dostęp do konta użytkownika oraz jego zasobów. Zanim będzie mogła to uczynić, musi to zostać potwierdzone przez użytkownika. To potwierdzenie musi zostać zweryfikowane poprzez api służące to autentykacji, które następnie wystawi przepustkę w postaci tokenu. Token ten pozwoli aplikacji na dostęp do danego zasobu posiadanego przez użytkownika.



Rysunek 1. Diagram przedstawiający przypływ danych pomiędzy rolami w OAuth

Powyższy diagram obrazuje przepływ kolejnych wiadomości w czasie pracy z Oauth2.0. Oczywiście wiele zależy od konkretnej implementacji, jednak większość z nich trzyma się ram, nakreślonych przez ten schemat. Aplikacja początkowo prosi użytkownika o dostęp do jego zasobów. Jeżeli użytkownik zezwolił na taki dostęp, w odpowiedzi odsyła Authorization Grant. Następnie aplikacja żąda wystawienia tokenu, przez serwer autoryzacyjny. W żądaniu tym znajduje się Authorazition Grant, wystawiany przez użytkownika. Serwer autoryzacji weryfikuje tożsamość użytkownika na podstawie przesłanego upoważnienia a następnie odsyła token, zezwalający na dostęp od zasobów. Od tego momentu, przy każdym żądaniu zasobów przez aplikację kliencką przesyłany jest token. Na jego podstawie Resource Server weryfikuje, czy dany użytkownik może mieć do nich dostęp.

Implementacja Oauth2.0 w omawianej aplikacji, składa się z kilku elementów. Każdy z nich pełni osobnę rolę w autentykacji oraz przyznawaniu dostępów do zasobów. Klasa ApplicationDbContext służy do przygotowania dostępu do bazy danych. Jako parametr generyczny przyjmuje ona klasę ApplicationUser, w której zdefiniowano informacje o użytkowniku.

Kolejnym elementem implementacji, jest ApplicationUserManger, który rozszerza klasę, udostępnioną przez jedną z bibliotek .Net.Identity. Zajmuje się ona zarządzaniem użytkownikami oraz ich właściwościami. To za jej pomocą, utworzymy nowego użytkownika, nadamy mu odpowiednią rolę systemową bądź uprawnienia. To również ona będzie weryfikowała, czy podany login i hasło są poprawne oraz czy przy ich pomocy, można wystawić token do autentykacji. Klasa ta nie powinna korzystać bezpośrednio z dostępu do bazy danych. Jej działanie powinno skupić się na innym poziomie abstrakcji, w oderwaniu od samego źródła danych. Operacje na ApplicationDbContext powinny być zaimplementowane w ApplicationUserStore, z których następnie będzie korzystał omawiany menadżer. Dzięki takiemu podejściu, w bardzo łatwy sposób można zmienić źródła danych o użytkownikach oraz sposób ich przetwarzania. Zmiana taka jednak nie będzie zmuszać programistów do modyfikacji logiki stojącej za autoryzacją oraz autentykacją. Jak widać cały czas wspierany jest wzorzec projektowy Dependency Injection (Wstrzykiwanie Zależności).



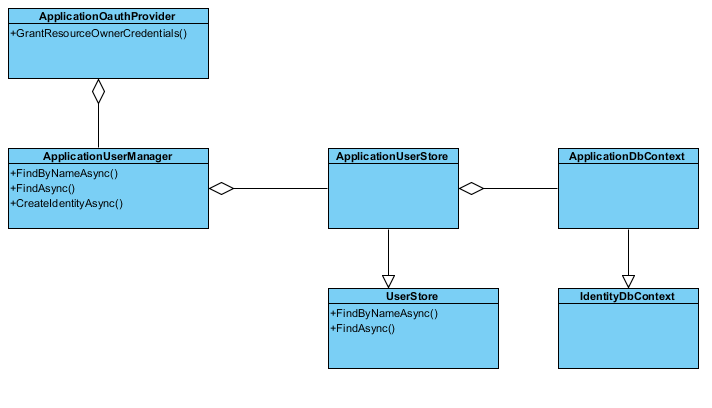
Kolejnym elementem tej układanki jest ApplicationOauthProvier. To on definiuje sposób, w jaki dany użytkownik ma potwierdzić swoją tożsamość. To tutaj przykładowo odbierane są login i hasło po wysłaniu Athorization Request przez aplikację kliencką. Następnie wykorzystywany jest ApplicationUserManager w celu potwierdzenia, czy dane przesłane do autoryzacji są poprawne. Jeżeli weryfikacja ta przejdzie pomyślnie, to właśnie ten komponent zwraca żądany token. Następnie w module konfiguracyjnym API, ustawiamy adres pod jakim ma być wystawiony endpoint, który będzie odbierał dane do weryfikowania. Specyfikujemy również rodzaj Proividera, który taką weryfikacje ma przeprowadzić.



Rysunek 2. ApplicationOathProvider



Rysunek 3. Funkcja konfiguracyjna endpoint wykorzystywany do uzyskania tokenu



Rysunek 2. Diagram klas obazujący fragment implementacji OAuth2.0 w .Net Web.Api

## **5. Opis i implementacja CQRS**

W tym rozdziale zostanie omówiona wdrożenie i implementacja wzorca architektonicznego CQRS (Command Query Responsibility Segragation). Skupimy się tu bardziej na części praktycznej. Opis ten będzie dość ściśle powiązany z wybraną technologią, jednak budowa modułów jest na tyle reużywalna, że bez większego problemu można go zastosować do stworzenia implementacji na bazie innych języków i platform.

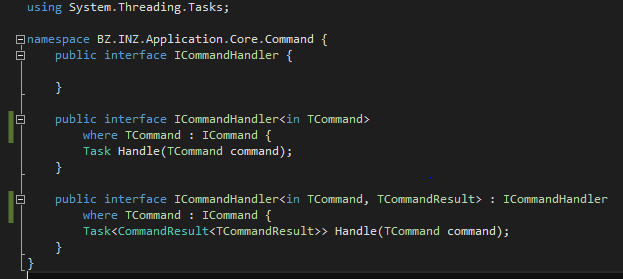
W ramach krótkiego przypomnienia, CQRS w swoim najbardziej podstawowym zadaniu służy do separowania operacji pobierania danych, od operacji ich zmiany bądź dodawania. Aby mieć łatwiejszą kontrolę nad tą separacją, zdefiniowaliśmy sobie odpowiednie handlery, w których zamknięta została logika danej operacji. Wyróżniamy dwa podstawowe typy handlerów, których implementacja jest ściśle powiązana z jego rolą w systemie. Pierwszym typem jest QueryHandler. Ten typ handlera służy do pobierania danych z wybranego źródła bądź źródeł. Drugim typem jest CommandHandler. Jego istotą jest obsługą poleceń, tak zwanych Command, których wynikiem powinna być jakaś zmiana w systemie lub strukturze danych bądź wykonanie jakiejś akcji. Z obsługom CommandHandlerów wiąże się również pojęcie HandlerInvokera. Jego znaczenie będzie przybliżone w dalszej części rozdziału.

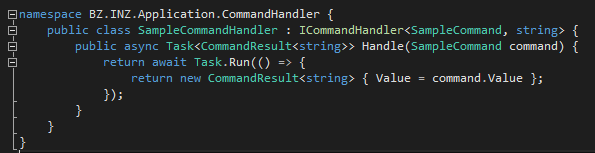
Praktycznie cała implementacja struktury i logiki stojącej za działaniem CQRS znajduje się na warstwie aplikacji.

**5.1. Implementacja CommandHandler’a**

Jak już wcześniej zostało napisane, CommandHandler służy do wykonania jakiejś akcji w systemie. Według niektórych publikacji nie powinien on zwracać żadnych danych, jednak w moim mniemaniu jest to zbyt duża restrykcja. Istnieje spora grupa operacji, których wykonanie powinno zwrócić pewien rezultat do wywołującego. Wykorzystywanie do tego osobnych zapytań przy pomocy QueryHandlerów niepotrzebnie komplikowałoby przepływ danych. Pierwszym z brzegu przykładem jest chociażby wstawienie nowego rekordu do relacyjnej bazy danych, w wyniku tej operacji użytkownik powinien otrzymać id, które zostanie wygenerowane w czasie tworzenia wpisu.

Każdy CommandHandler powiązany jest z odpowiadającą mu komendą, która dostarcza mu odpowiednich danych, niezbędnych do wykonania dedykowanego mu zadania. Wszystkie Commands implementują ten sam interfejs – ICommand. Natomiast każdy CommandHandler dziedziczy po generycznym interfejsie, którego parametrami są właśnie typy implementujące ICommand. W ten sposób uzyskaliśmy połączenie między poleceniem wykonania danej czynności, oraz komponentem, który jest w stanie to polecenie wykonać. Zostanie to zaprezentowane na poniższym listingu:



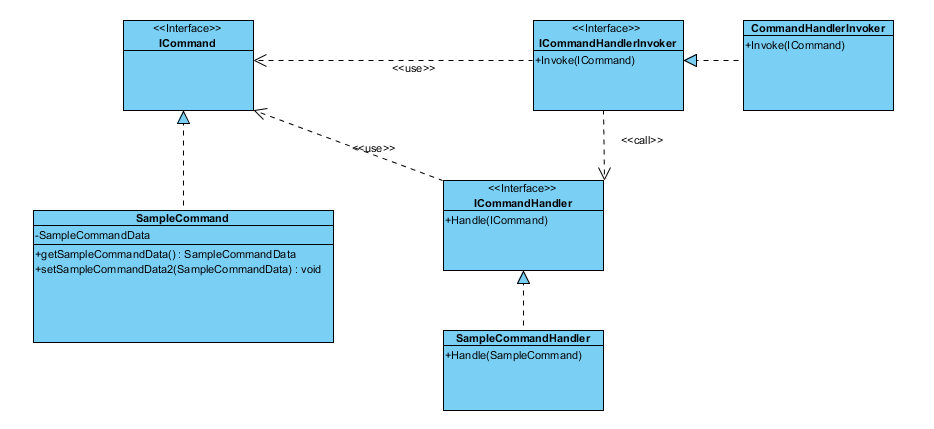


Powyższy listing uchwycił istotę połączenia pomiędzy komendą oraz handlerem, który jest do niej przypisany. Istnieją co prawda założenia, w których jeden handler może obsługiwać kilka komend, jednak na potrzeby tej pracy, przyjęto założenie, że zdecydowanie bardziej czytelnym podejściem jest relacja jeden do jeden.

Poruszając temat komend i handlerów nie można zapomnieć o jeszcze jednym, bardzo istotnym elemencie – HandlerInvokerze. HandlerInvoker jest klasa, która zajmuje się przechwytywaniem i rozdysponowaniem komand pomiędzy dedykowane im handlery. To właśnie ten element implementacji tego wzorca stanowi o jego sile i prostocie w dalszej z nim pracy. Dzięki niemu, głównym zadaniem programistów pracujących przy warstwie aplikacji jest tworzenie kolejnych par: komenda – handler. Jeszcze prościej prezentuje się sposób obsługi API, wspomaganego przez invoker. Z perspektywy programistów zajmujących się front-endem takiej aplikacji, wywołanie jakiegoś zdarzenia po stronie serwera, sprowadza się do przygotowania odpowiedniej struktury danych i wysłanie jej na odpowiedni interfejs. Dzięki wcześniej przygotowanemu invokerowi, jesteśmy w stanie zebrać wszystkiego przychodzące struktury do jednego interfejsu. Następnie struktury te są parsowane na obiekty w odpowiedni dla danego języka sposób, obiekty te reprezentują już nasze docelowe komendy. Kolejnym etapem jest przesłanie takiego obiektu do invokera, który zajmie się wywołaniem odpowiedniego CommandHandlera. Dobrą praktyką jest, aby wydzielić odpowiedni moduł na warstwie aplikacji, który zajmie się walidacją przychodzących poleceń. Moduł ten może sprawdzać zarówno poprawność parsowania jak i struktury. Może również przechowywać dodatkowe reguły walidacyjne, istotne z poziomu logiki biznesowej, dedykowane dla konkretnych typów poleceń. Taką walidację nazywamy fachowo „Deep validation” lub „Custom validation”.

Sprzężenie ze sobą interfejsu wystawianego na zewnątrz oraz HandlerInvokera prowadzi do pewnej ciekawej i istotnej zależności. Mianowicie warstwa komunikacji z GUI zostaje bardzo uproszczona. Praktycznie cała logika biznesowa została wyciągnięta na zewnątrz i zamknięta pod postacią komend i handlerów. Moduł ten zajmuje się wtedy tylko tym, do czego natywnie jest dedykowany, zapewnia komunikację między środowiskiem zewnętrznym a resztą aplikacji. Ponadto, może zajmować się autoryzacją i autentykacją, oraz walidacją dostępów do danych zasobów. W bardziej rozbudowanych aplikacjach wskazane, jest jednak wydzielenie dedykowanego komponentu, który zajmowałby się takimi zadaniami. Komponent ten wystawia odpowiednie metody, w zależności od potrzeb. Wszystkie te zabiegi powodują, że taki spłaszczony moduł, jest bardzo prosty do zastąpienia. Jeżeli z jakiś powodów postanowimy zmienić typ komunikacji na inny, przykładowo zastąpić serwis REST usługą SOAP, to taka zmiana nie naruszy istniejącej logiki biznesowej. Cała praca developerów nadal skupi się tylko na implementacji nowego sposobu komunikacji.

Poniżej zostanie zaprezentowany diagram klas, który przedstawia główną infrastrukturę stojąca za działaniem i obsługom komand.



**5.2 Implementacja QueryHandler’a**

Ostatnim z głównych elementów składających się na implementacje CQRS są QueryHandlery. W przeciwieństwie do elementów opisanych poprzednio, ich celem nie jest zmiana, lecz pobieranie danych. Każdy QueryHandler jako parametr może przyjąć wcześniej przygotowane klucze, które będą służyć do odpowiedniego filtrowania wyników. Kolejnym ich zastosowaniem jest przygotowanie zapytań, które będą wykorzystywane przez warstwy integracji bądź infrastruktury do pobierania danych z systemów zewnętrznych. Na każdy handler może przypaść kilka źródeł danych. Dane mogę być pobierane synchronicznie bądź asynchronicznie. Po zebraniu wszystkich informacji, odpowiednie modele danych wypełniane są informacji. Często zdarza się, że przed samym wysłaniem następuje jeszcze odpowiednie przygotowanie struktur, chociażby przez wykonanie odpowiednich obliczeń bądź formatowanie. Istotne jest tutaj, żeby rozróżnić takie przygotowanie danych od ich zmiany. QueryHandler nie może posiadać żadnej logiki, która mogłaby w trwały sposób oddziaływać na pobierane dane. Wszystkie działania sprowadzamy do trybu „tylko do odczytu”.

Biorąc pod uwagę bardziej techniczną stronę implementacji, tworzymy generyczny interfejs, który jako parametry będzie przyjmował typ zwracanych danych oraz typ klucza, po którym będziemy te dane filtrować. Dzięki takiej parze parametrów, jesteśmy w stanie stworzyć wiele różnych implementacji tego interfejsu, zwracających dane tego samego typu, lecz znaczenie różniących się swoim źródłem bądź sposobem filtrowania. Przykładowo, użyjemy innego klucza, aby pobierać dane z własnej bazy danych oraz innego do pobierania takich danych z zewnętrznego systemu. Ponadto, w takim kluczu, możemy zawrzeć informację na temat danych uwierzytelniających, do systemów zewnętrznych, jeżeli z takowych korzystamy. Innym rozwiązaniem, jest pobieranie takich danych z odpowiedniego komponentu, już w samej implementacji danego QueryHandlera. Zazwyczaj jednak, przynajmniej część z nich przesyłana jest wraz z kluczem.

Jako klucz, możemy użyć zarówno dowolnego typu prostego, jak i bardziej skomplikowanej struktury, opakowanej w klasę. Dane zwracane przez dany QueryHandler, opakowane zazwyczaj są w interfejs generyczny IQueryable<T>, gdzie T jest to typ zwracanych informacji. Takie rozwiązanie daje nam co najmniej dwie korzyści, pierwszą z nich jest identyczny format przesyłanych danych, przez każdą z jego implementacji. Drugą, co najmniej równie ważną, jest wykorzystanie specyficznego działania interfejsu IQueryable w kontakcie z bazą danych. Jeżeli pobieramy dane do zwyczajnej kolekcji, implementującej interfejs IEnumerable, na przykład Listy, to zawsze pobieramy wszystkie rekordy z danej tabeli do pamięci RAM. Następnie już w tej pamięci następują operacje sortowania i filtrowania danych, które następnie zostaną przekazane przez QueryHandler do miejsca jego wywołania. W przypadku, gdy tabela zawiera wiele milionów rekordów, a my chcemy pobrać przykładowo tylko pięć pierwszych, jest to skrajnie nieoptymalne rozwiązanie. Inaczej sprawa prezentuje się z interfejsem IQueryable. Każde operacja wywołana na IQueryable powoduje, że przygotowane jest odpowiednie zapytanie w języku SQL, które zostaje wywołane bezpośrednio na bazie danych. Powoduje to, że do pamięci zostaną pobrane tylko te dane, które faktycznie chcemy pobrać. Na poniższych listingach zaprezentowany został interfejs IQueryHandler oraz jego przykładowe implementacje.







Jak widać, sam interfejs służący do pobierania danych, występuje w dwóch postaciach. W pierwszym wypadku zwracamy bezpośrednio kolekcję z danymi. W drugim wypadku wykorzystujemy ponadto słowo kluczowe Task. Daje nam to możliwość asynchronicznego pobierania danych, w tym samym czasie - możemy wywołać wiele IQueryHandlerów jednocześnie.

Pierwszą z przedstawionych implementacji jest SampleQueryHandler. Jako klucz przyjmuje on zwykły string, który następnie zwraca jako wynik działania handlera. Implementacja ta ma na celu jedynie przetestowania działania przygotowanej infrastruktury.

Znacznie ciekawszym przykładem jest JobOfferQueryHandler. Po pierwsze zarówno jako klucz jak i wynik swojego działania, używa wcześniej przygotowanej struktury danych. Ponadto sam handler w czasie swojego działania, wykorzystuje obiekt, który pochodzi z warstwy infrastruktury. W tym wypadku widzimy użycie interfejsu reprezentującego Unit of Work. Unit of Work jest to wzorzec projektowy, służący do usystematyzowania dostępu do bazy danych oraz zapewniający metody do pracy nad danymi, które w tej bazie się znajdują. Jego bardziej szczegółowy opis znajduje się w dalszej części pracy. Pokazane tu zostało, w jaki sposób wykorzystany jest prosty klucz do filtrowania pobieranych danych. Widzimy, że przesłane będą tylko te dane, które mieszczą się w danym zakresie czasowym. Nic nie stoi na przeszkodzie, aby rozszerzać klucze o nowe parametry, które będą następnie używane przy pozyskiwaniu danych.

**6. Opis i implementacji Unit of Work**

W tym rozdziale zostanie omówiona kwestia dostępu do danych przetrzymywanych bezpośrednio w naszej bazie danych. Zostanie poruszona kwestia wzorców projektowych Unit of Work i Repozytorium oraz porównanie ich zastosowania.

Zanim przejdziemy do szczegółów implementacji samego wzorca, przyjrzyjmy się technologiom pozwalającym nam na odczyt i zapis danych do bazy. W omawianej aplikacji zrezygnowano z ręcznego tworzenia zapytań SQL na rzecz technologii ORM. ORM, czyli Object Relational Mapping to sposób odwzorowania relacyjnej bazy danych lub innej relacyjnej struktury na obiektową architekturę. Jako silnik ORM został użyty Entity Framework. Bliżej zarówno o Entity Framework jak i samej idei ORM można przeczytać w rozdziałach poświęconych wykorzystanym technologiom. Tutaj skupimy się na praktycznym wykorzystaniu powyższych technologii przy implementacji omawianych wzorców.

Repozytorium jest wzorcem, który jest pomostem pomiędzy warstwą logiki biznesowej, a modułami zajmującymi się dostępem do danych zapisanych w bazie danych. Wprowadza ona dodatkową warstwę abstrakcji, dzięki której w łatwiejszy sposób możemy implementować operacje oparte o zapytania kierowane do bazy. Dzięki niemu jesteśmy w stanie odprawować zapytania SQL bądź technologię ORM od naszego warstw wyższych, zasilanych danymi pobranymi z bazy danych. Przyjmowane jest założenie, że każda implementacja repozytorium dedykowana jest konkretnej relacji: tabela – obiektowy model danych. Wszystkie repozytoria implementują wspólny interfejs IRepository, w którym zadeklarowane są podstawowe metody do obsługi danych – dodawanie, usuwanie, edycja oraz pobieranie. Nic nie stoi na przeszkodzie, aby każda konkretna implementacja repozytorium była rozszerzona o dodatkowe metody, specyficzne dla wybranego typu danych.

Stosowanie powyższego wzorca niesie za sobą jednak pewne konsekwencje. W sytuacji, gdy stosujemy technologię ORM, część jej właściwości jest zdublowana przez repozytorium. Ponadto niektóre z metod i właściwości zapewnianych przez silnik ORM jest niedostępna, bez ich zaimplementowania w repozytorium. Może być to sytuacja pożądana, jeżeli chcemy, aby pozostałe warstwy miały dostęp tylko do wybranych przez nas funkcji. Czasami prowadzi to jednak do dużej ilości nadmiarowego kodu. Kolejnym problem w pracy z repozytoriami w klasycznym tego słowa znaczeniu jest tworzenie transakcji. Spowodowane jest to tym, że każda tabela posiada własną implementację repozytorium, a co za tym idzie, każde repozytorium posiada osobną metodę SaveChanges. Jeżeli dana operacja przeprowadzana jest tylko na jednej tabeli, to żaden problem nie występuje. Po prawnym wprowadzeniu zmian, możemy je bezproblemowo zapisać. Jednak zdecydowana większość operacji przeprowadzana jest równocześnie na większej ilości tabel. Powoduje to, że to programiście spoczywa uruchomienie funkcji SaveChanges w odpowiednim momencie. Brak zapisu zmian którymś z używanych repozytorium może powodować niespójność danych, nad którymi pracujemy.

Jak widać wzorzec Repozytorium ma swoje wady i zalety. Jego główne atuty widzimy wtedy, jeżeli używamy bezpośredniego dostępu do bazy danych z użyciem kwerend SQL. Następuje wtedy enkapsulacja kodu SQL pod postacią implementacji konkretnych metod Repozytorium. Postać tych zapytań różni się, w zależności od tabeli, która jest źródłem danych dla danej implementacji. Posiadamy wtedy zbiór klas implementujących metody zawarte w danym interfejsie, jednak robiących to na różne sposoby. Nieco inaczej sytuacja wygląda jednak w przypadku użycia Frameworków ORM. Jak już wiemy, w naszej aplikacji został zastosowany Entity Framework. Przy jego pomocy uniknęliśmy konieczności pisania kwerend. Cała obsługa bazy danych tworzona jest w C# oraz bazuje na modelu obiektowym. Wszystkie operacje przeprowadzone na dowolnej tabeli, wykonywane według tego samego schematu, różnią się tylko modelem na którym pracujemy. W zaistniałej sytuacji tworzenie dodatkowych repozytoriów, enkapsulujących operacje na danym modelu danych (danej tabeli) może być tworzeniem nadmiarowej abstrakcji.

Alternatywą lub rozszerzeniem dla wzorca Repozytorium, jest wzorzec projektowy Unit Of Work, a w szczególności jego generyczna implementacja. Unit Of Work w założeniu daje dostęp operacji na dowolnej tabeli, jaka zostanie pod niego podpięta. Zasilany może być zarówno przez repozytoria, które dostarczają mu danych do pracy na konkretnych tabelach, jak i przez silnik ORM.

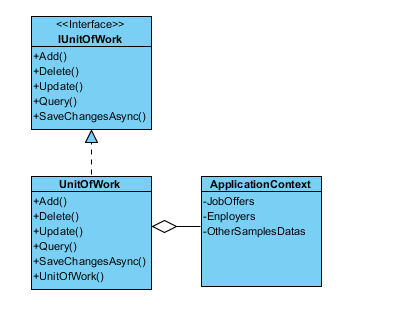
Zanim przejdziemy do omówienie implementacji samego Unit of Work, przyjrzyjmy się klasie ApplicationContext.



Jak widać, klasa ta dziedziczy po klasie DbContext, która wchodzi w skład biblioteki EntityFramework. Jednym ze skutków takiego dziedziczenia jest możliwość tworzenia generycznych DbSetów. Każdy DbSet przyjmuje jako generyczny parametr strukturę danych. Na podstawie tych struktur oraz relacji między nimi, może zostać wygenerowana odpowiadająca im baza danych. Z drugiej strony istnieje możliwość wygenerowania odpowiedniej implementacji DbConext na podstawie istniejącej bazy danych. Z perspektywy wzorca UoW istotną właściwością DbSetów, są metody, które umożliwiają dodawanie, usuwanie, edycje oraz pobieranie danych. Implementują więc wszystkie podstawowe funkcje repozytoriów. Spójrzmy teraz na interfejs IUnitOfWork oraz implementującą go klasę.

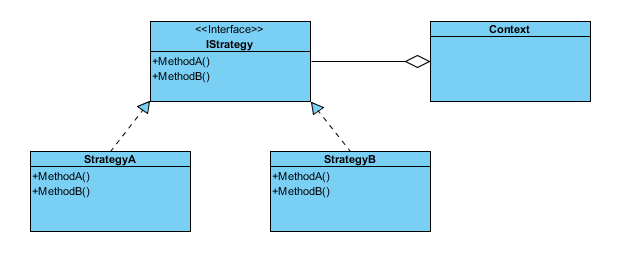


 Jak widać na powyższych listingach w interfejsie IUnitOfWork zadeklarowane są metody CRUD (Create, Read, Update, Delete). Metody te są generyczne, dzięki klauzuli where ich parametrem może być dowolna klasa. W klasie implementującej ten interfejs możemy się przyglądnąć ciałom tych metod. Każda z nich wykorzystuje na swój sposób omówioną wcześniej klasę ApplicationContext. Operacje dodawania, usuwania i pobierania, korzystają z metody Set, która dzięki generycznemu parametrowi odwołuje się do wcześniej zdefiniowanego DbSetu. Dzięki temu, mogą one przyjąć dowolny typ danych, jaki został wcześniej zarejestrowany w ApplicationContext. Na podobnej zasadzie działa operacja Update, która wywołuje generyczną metodę Entry. Istotny jest tutaj fakt, że parametrem, z którym ta funkcja zostanie wywołana, jest wcześniej pobrany i przemapowany do obiektu, rekord z bazy danych. Ostatnią funkcją zawartą w UoW jest SaveChangesAsync. Służy ona do faktycznego zapisania zmian w bazie danych. To jej wywołanie zapisuje zmiany wprowadzone przez pozostałe funkcje. Warto zwrócić uwagę, że każda z metod zwraca typ Task bądź Task<T>. Dzięki czemu mogą być uruchomiane równolegle na osobnych wątkach, co znacznie zwiększa wydajność.



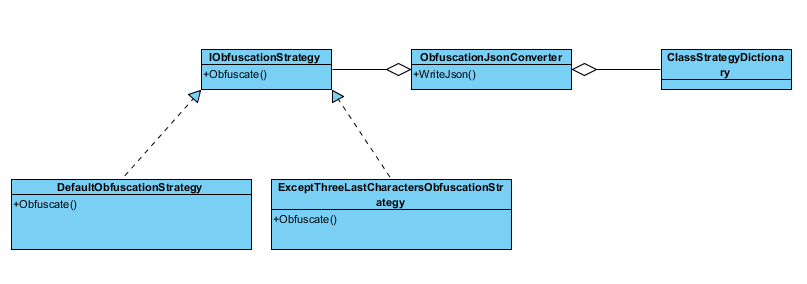
**7. Omówienie i implementacja Strategii**

Strategia należy do grona wzorców czynnościowych. Opiera się ona na definiowaniu grupy wymiennych rozwiązań pewnego problemu oraz opakowaniu ich w klasy. Dzięki czemu można je stosować zamiennie w czasie działania aplikacji.

W ogólności składa się ona z IStrategy, interfejsu, który deklaruje operacje służącą do rozwiązania problemu, przy pomocy wzorca. Konkretnej strategii, jest to implementacja wybranego algorytmu służącego do rozwiązania problemu. Klasa, w której zawarty jest taki algorytm musi implementować interfejs IStrategy. Kontekst to klasa, która odbiera zlecenie wykonania pewnego zadania, a następnie dokonuje wyboru, przy pomocy jakiego algorytmu dane zadania ma zostać wykonane. To ona decyduje, kiedy i która z implementacji strategii ma zostać zastosowana.

Wzorzec ten został zastosowany, w realizacji zaciemniania logów. W aplikacjach webowych ważnym elementem jest logowanie informacji, jakie przepływają między kolejnymi komponentami. Dzięki takiemu zbiorowi zapisanych danych programiści są w stanie wykryć błędy oraz znaleźć ich przyczynę. Często jednak zdarza się, że część zapisanych danych to dane wrażliwe, które nie powinny być w pełni dostępne dla osób nieuprawnionych. Tutaj przychodzi na pomoc moduł, który zajmuje się zaciemnianiem takich informacji. Przy jego implementacji został wykorzystany wyżej poruszony wzorzec Strategia. Dzięki niemu można zdefiniować różne sposoby na zaciemnienie konkretnych informacji, a następnie przypisać je do konkretnych struktur danych, które następnie będą podlegać logowaniu. Następnie klasa zajmująca się formatowaniem i transformacją obiektów na struktury płaskie wykorzystuje słownik, gdzie kluczem jest pole, które ma zostać zaciemnione, a wartością dana strategia.

Dzięki takiemu podejściu, bardzo łatwo można zmieniać sposób zapiania danej informacji. Ponadto nic nie stoi na przeszkodzie, by przygotować kilka słowników, które będą wykorzystywane na różnych poziomach logowania, w zależności od konfiguracji. Przykładowo, można przygotować inne strategie logowania i zaciemnia informacji, ze względu na użytkowników, dla których dane logi będą prezentowane. Inne informacje będą niezbędne programistom, rozwiązującym błędy w działaniu oprogramowania. A inne analitykom, próbującym doszukać się problemów w projekcie logiki biznesowej danej aplikacji.





Rysunek 5. Poszczególnie implementacje strategii

****

Rysunek 6. Użycie wybranej strategii przy zaciemnianiu logów.

****