## Implementacja CQRS

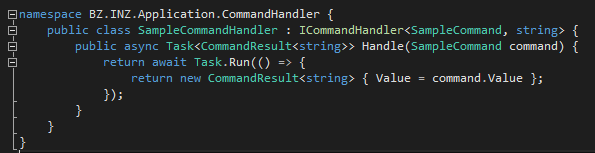
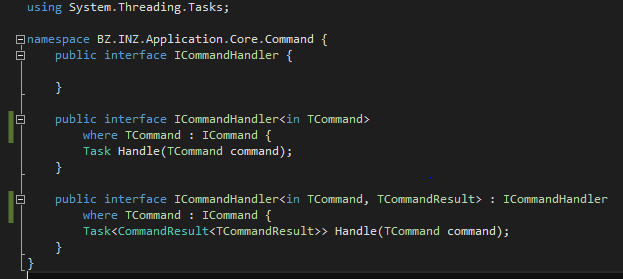
W tym rozdziale zostanie omówiona wdrożenie i implementacja wzorca architektonicznego CQRS (Command Query Responsibility Segragation). Skupimy się tu bardziej na części praktycznej. Zagadnienia bardziej teoretyczne zostały poruszone w rozdziałach dotyczących ogólnemu opisowi zastosowanej architektury. 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 implantacja 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.

Jak już wcześniej zostało napisane, CommandHandler służy nam w 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 wstanie to polecenie wykonać. dZostanie to zaprezentowane na poniższym listingu:



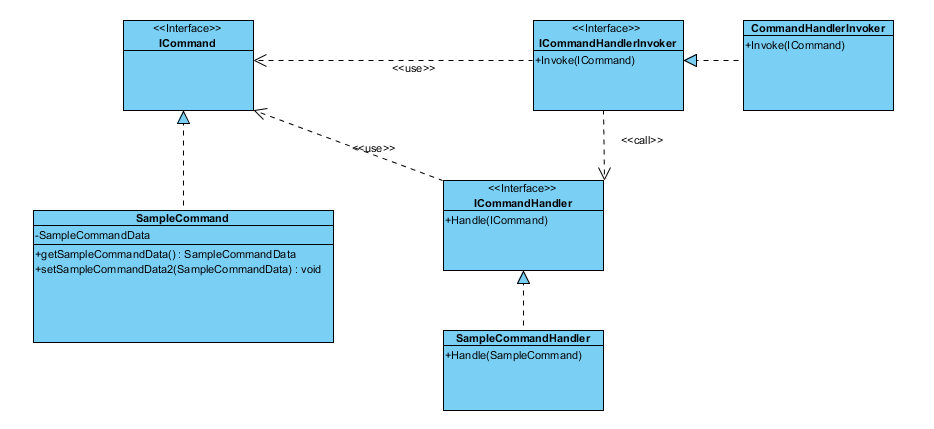
Powyższy listing uchwycił istotę połączenia pomiędzy komendą oraz handlerem który jest do niej przypisany. Istnieją co prawda założenia, w których jeden handler może obsługiwać kilka komend, jednak na potrzeby tej pracy, przyjęto założenie, że zdecydowanie bardziej czytelnym podejściem jest relacja jeden 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 jakieś zdarzenia po stronie serwera, sprowadza się do przygotowania odpowiedniej struktury danych i wysłanie jej na odpowiedni interfejs. Dzięki wcześniej przygotowanemu invokerowi, jesteśmy wstanie 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 została 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 wykorzystywane 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 implementacją nowego sposobu komunikacji.

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



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 każdych 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 może w trwały sposób oddziaływać na pobierane dane. Wszystkie działania sprowadzamy do trybu „tylko do oczytu”.

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 wstanie 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 uwspólnianie formatu 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 przygotowany 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.