## Implementacja CQRS

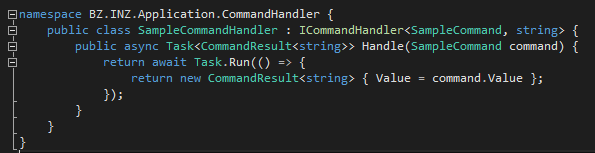
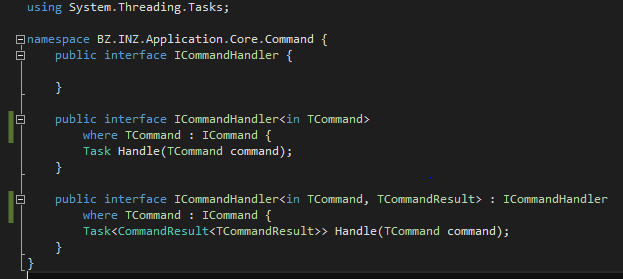
W tym rozdziale zostanie omówiona wdrożenie i implementacja wzorca architektonicznego CQRS (Command Query Responsibility Segragation). Skupimy się tu bardziej na części praktycznej. Zagadnienia bardziej teoretyczne zostały poruszone w rozdziałach dotyczących ogólnemu opisowi zastosowanej architektury.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

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. Sprawdzanie takie 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.

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 dekorowanie danych od ich zmiany. QueryHandler nie może posiadać żadnej logiki, która może w trwały sposób wpłynąć na pobierane dane, bądź zmusić inny komponent do podobnego czynu. Wszystkie działania sprowadzamy do trybu READ.

Biorąc pod uwagę bardziej techniczną stronę implementacji,