Politechnika Śląska

Wydział Automatyki, Elektroniki i Informatyki

Kierunek Informatyka

Mateusz Stanik

###### **Mapowanie obiektowo-relacyjne – badanie wydajności dostępu do bazy przy wykorzystaniu technologii Entity Framework**

##### praca dyplomowa magisterska

Promotor: Aleksandra Werner

Konsultant:

Gliwice, lipiec 2018

Załącznik Nr 2 do Zarz. Nr 97/08/09

**Oświadczenie**

Wyrażam zgodę/nie wyrażam\* zgody na udostępnienie mojej pracy dyplomowej/rozprawy doktorskiej\*

…………….………., dnia …………………………

……………………………………..……………….……

*(podpis)*

……………………………………………………………

*(poświadczenie wiarygodności podpisu przez Dziekanat)*

*\* właściwe podkreślić*

**Oświadczenie promotora**

Oświadczam, że praca „Mapowanie obiektowo-relacyjne – badanie wydajności dostępu do bazy przy wykorzystaniu technologii Entity Framework” spełnia wymagania formalne pracy dyplomowej magisterskiej.

|  |  |
| --- | --- |
| Gliwice, dnia ……………………… | ………………..……………….……  *(podpis)* |

Spis treści

[1. Wstęp 9](#_Toc516145423)

[2. Technologia ORM 11](#_Toc516145424)

[2.1. Technologia Entity Framework 12](#_Toc516145425)

[2.1.1. Możliwości tworzenia bazy 12](#_Toc516145426)

[2.1.2. Klasa kontekstowa Entity Framework 14](#_Toc516145427)

[3. Aplikacja 17](#_Toc516145428)

[3.1. Wymagania funkcjonalne 18](#_Toc516145429)

[3.2. Wymagania niefunkcjonalne 19](#_Toc516145430)

[3.3. Opis bazy danych aplikacji 20](#_Toc516145431)

[3.4. Testy aplikacji 23](#_Toc516145432)

[4. Część badawcza 26](#_Toc516145433)

[4.1. Metodyka badań/Plan eksperymentów/testów 27](#_Toc516145434)

[4.2. Konfiguracja środowiska testowego/Środowisko testowe 29](#_Toc516145435)

[4.3. Zbiory danych 31](#_Toc516145436)

[4.4. Uzyskane rezultaty/Wyniki 33](#_Toc516145437)

[4.5. Wnioski 34](#_Toc516145438)

[5. Podsumowanie 36](#_Toc516145439)

[Bibliografia i](#_Toc516145440)

[Spis skrótów i symboli ii](#_Toc516145441)

[Zawartość dołączonej płyty iii](#_Toc516145442)

[Spis rysunków iv](#_Toc516145443)

[Spis tabel v](#_Toc516145444)

# Wstęp

W dzisiejszych czasach tworzenie relacyjnych baz danych możliwe jest przy wykorzystaniu dwóch kluczowych podejść. Pierwszym z nich jest strukturalny język zapytań SQL, a drugim technologia ORM oparta   
o technikę obiektową. Architekci tworzący systemy informatyczne ciągle stoją przed trudnym wyborem jakim jest zdecydowanie się na jedno   
z wymienionych podejść. Jest to kluczowa decyzja w procesie wytwarzania oprogramowania ze względu na późniejszy proces tworzenia   
i administrowania bazą. Od momentu kiedy pojawiła się technologia ORM programiści zastanawiają się czy na pewno zastosowanie jej w swoim projekcie jest właściwą decyzją. Istnieje przypuszczenie, że owe podejście tworzy chaos w kodzie przez co staje się „*nieczysty*”, a ponadto zapytania wykonywane do bazy danych są przetwarzane kilkukrotnie dłużej od zwykłych zapytań w języku SQL. Niestety nie jest to jasno sprecyzowane czy środowiska oparte o technologię obiektową są wydajniejsze i łatwiejsze w zarządzaniu od tradycyjnego podejścia, przez co staje się to odwieczną bolączka programistów piszących systemy informatyczne. W niniejszej pracy zostaną przeprowadzone badania weryfikujące postawioną hipotezę mówiącą o wyższości zapytań SQL-owych w aspekcie wydajności. Napisane testy będą rozpatrywane w obu podejściach, poczym zostaną opisane wnioski dla otrzymanych rezultatów. Taka ocena będzie mogła pomóc programistom w podjęciu decyzji o wyborze stylu oprogramowania bazy danych. Praca magisterska składa się z trzech zasadniczych części, ich opis zostanie opisany poniżej.

W części pierwszej została omówiona technologia ORM. Opisano tu na jakiej zasadzie działa tworzenie bazy danych opartej o model programowania obiektowego. Zestawiono w tym rozdziale odpowiedniki tworzenie baz danych w zapisie SQL z zapisem ORM, w skład których wchodzą deklaracje tabele, jej atrybutów oraz relacji. W tym miejscu zaprezentowano również przykładowe biblioteki programistyczne umożliwiające zapis relacyjnych baz danych w takim stylu programowania.

Kolejną częścią pracy jest zaprezentowanie obiektu badawczego. Aby możliwe było rozpoczęcie badań wydajnościowych konieczne było utworzenie wcześniej aplikacji na której zostaną przeprowadzane badania. W związku z czym przygotowano testową aplikację internetową mającą na celu wystawianie, a następnie sprzedaż sprzętu żeglarskiego. Dla aplikacji stworzono relacyjną bazę danych na której kolejno zostały przeprowadzane zapytania testujące wydajność bazy opartej o technologię obiektową.

W ostatniej, trzeciej części pracy zostały zaprezentowane przeprowadzone badania. Znajduje się w tym miejscu ich opis oraz wnioski dla każdego z nich. Jak wspomniano na początku rozdziału, badania miały na celu porównanie wydajności zapisanych żądań bazodanowych w języku SQL, a wybranej technologii ORM. Praca zawiera nie tylko porównania dotyczące szybkości zapytań wykonanych w obu typach bazy, ale również łatwości ich zapisania przez developera wytwarzającego kod.

Praca badawcza została zakończona wnioskami wyciągniętymi po przeprowadzeniu badań. W tej części rozwiano wątpliwości dotyczące wydajności podejść, oraz opisano dodatkowe aspekty czyniące daną technologię lepszą od drugiej.

# Technologia ORM

Jednym ze sposobów zarządzania relacyjnymi bazami danych jest język SQL. Dzięki niemu możliwe jest tworzenie tabel, zakładanie dla nich atrybutów, dodawanie między nimi relacji dzięki nadawaniu kluczy głównych i obcych czy też tworzenie widoków umożliwiających dostęp do podzbioru składającego się z kolumn i wierszy tabel. Istnieją dodatkowo dla tego języku rozszerzenia T-SQL (Transact-SQL), oraz PL-SQL (Procedural Language/Structured Query Language). Wykorzystuje się je do tworzenia procedur, funkcji, wyzwalaczy, tworzenia zmiennych w zapytaniach, dodawania pętli czy też instrukcji warunkowych. Wszystkie funkcjonalności opisane dla języka zapytań sprawiają, że wykorzystanie go w aplikacjach daje znaczne pole manewru dla programistów w aspekcie zarządzania oraz manipulacją danych przechowywanych w bazie.

Opisywane dotychczas podejście jest podstawowym sposobem zarządzania bazą danych, który został utworzony w latach siedemdziesiątych przez Amerykańską firmę *IBM*. Jednakże z biegiem czasu została rozpowszechniona technologia obiektowa, którą uznawano i uznaje się za przyszłość programowania. W dzisiejszych czasach dostępnych jest całe mnóstwo narzędzi wykorzystujących obiektowy model programowania, w związku z czym postanowiono również wprowadzić zmiany w relacyjnych bazach danych. Na początku lat dziewięćdziesiątych utworzono technologię mapowania obiektowo relacyjnego ORM  (ang. Object-Relational Mapping), która tak jak język SQL umożliwiała tworzenie relacyjnych baz danych. Niestety, istnieje w tym rodzaju zarządzania danymi szereg problemów wydajnościowych, które między innymi zostaną zweryfikowane dzięki przeprowadzonym badaniom w niniejszej pracy. Kluczową różnicą przy tym podejściu jest zamiana danych w postaci tabelarycznej (relacji w bazie danych) na język obiektowy, albo w drugą stronę. Jest nowoczesnym podejściem dla zagadnień związanych ze współpracą z bazą danych, wykorzystującą programowanie obiektowe. Jak sama nazwa wskazuje – obiektowo relacyjne mapowanie bazy danych opiera się na jakichś obiektach. Ideą tego podejścia jest aby zdefiniować model relacyjnej bazy danych na obiekty obiektowego języka programowania. Na tak zdefiniowanym modelu można wykonywać operacje na bazie danych tak jak na zwykłych obiektach. Aby móc stworzyć w takim podejściu tabele, konieczne jest stworzenie klasy która jest jej odpowiednikiem. Jej ciało jest miejscem na utworzenie zawartości encji, czyli na zdefiniowanie nazw pól wraz z typami danych. Nadawanie kluczy głównych oraz obcych morze odbywać się poprzez adnotację dla danego atrybutu, tak samo jak nadawanie im typów. Relacje w większości bibliotek nadawane są również poprzez adnotację jak i metody wirtualne wskazujące nazwę tabeli docelowej. W takim podejściu jeśli programista chce odwołać się do dowolnego elementu w bazie może utworzyć obiekt bazy z którego może dostać się do żądanego w nim miejsca. Odpowiednikiem takiej operacji w języku SQL byłoby stworzenie podłączenia tabel (przy pomocy *join-ów*).

Powyższy opis technologii ORM mówi o jej ogólnej zasadzie działania, tworzeniu struktury, i zapisie bazy. Dla frameworków dostępnych aktualnie na rynku możliwe są jednak drobne różnice między nimi najczęściej wynikającymi ze składni. Przykładowo można zaprezentować różnice dla najbardziej popularnych dodatków ORM, gdzie według statystyk pakietu NuGet firmy Microsoft na stan 2018 roku **[1]** jest *Entity Framework* oraz *Hibernate*. Pierwszym przykładem pokazującym różnice w obu tych bibliotekach jest zapis kluczy głównych, oraz obcych, jak i również zapis zależności między encjami. W Entity Framework-u odbywa się poprzez metody wirtualne wskazujące tabele docelową, dzięki czemu wiadomo czy relacja jest jeden do jeden lub podając kolekcję tworząc w ten sposób relację jeden do wielu. Natomiast w *Hibernate* używanie relacji odbywa się przy użyciu adnotacji dla atrybutów odpowiednio - *@OneToOne, @OneToMany*. Kolejną różnicą jest generacja identyfikatorów dla rekordów. W technologiach Microsoft-owych, mamy do dyspozycji *Identity Generation* oraz nadawanie kluczy specjalnych *GUID’s.* *Hibernate* pozsiada znacznie większy zakres generatorów w których skład wchodzą – *Identity, Sequence, Trigger-Based, HiLo,* oraz wiele innych.Znaczącą różnicą składniową może by również zapis *getter-ów* i *setter-ów*, ponieważ w technologii *Hibernate* ich zapis wykonywany jest w postaci metody przekazującej daną wartość, a następnie ją zwracając, zaś w *Entity* *Framewrok* zapis *getter-a* i *setter-a* przybiera następującą wartość - *<zakres> <typ> Id { get; set; }.* Jak widać jest to znacznie prostszy zapis czyniąc go wydajniejszym i przejrzystszym dla programisty*.* Poza różnicami merytorycznymi miedzy obydwoma bibliotekami można uznać wyższość jednego nad drugim pod względem dostępnej literatury uwarunkowanej językiem programowania. Entity Framework został stworzony prze firmę Microsoft, z myślą o wykorzystaniu jego dla platformy *.NET* wykorzystującej język obiektowy C#, przez co normalnym staje się, że ilość dostępnej dokumentacji będzie znacznie większa do niej, niż dla języka Java i frameworków wykorzystujących ją. Ostatnią znaczącą różnicą jest to, że nie wszystkie bazy danych są obsługiwane przez obydwie technologie. Na przykład biblioteka firmy Microsoft przeznaczona jest tylko dla basy *SQL Server*, a dostęp do innego rodzaju baz takich jak *MySQL* czy *Oracle* możliwy jest jedyne za pośrednictwem dodatkowej biblioteki. Technologia *Hibernate* nie potrzebuje do połączenia się z jakąkolwiek bazą dodatkowego oprogramowania.

## Technologia Entity Framework

Z pośród wszystkich dostępnych na rynku bibliotek ORM, do celów badawczych zdecydowano wykorzystać Entity Framework. Został on wybrany spośród innych frameworków ze względu na najlepszą dokumentację oraz ze względu na jego niezawodną komunikację z technologiami firmy Microsoft, a w szczególności z bazą *SQLServer,* do której możliwy był dostęp na czas badań*.*

Entity Framework umożliwia tworzenie modelu bazy danych na podstawie zaimportowanego skryptu SQL, jak również daje możliwość generacji jej w drugą stronę, co sprawia że jest bardzo elastyczną biblioteką dającą duże pole manewru dla programisty tworzącego aplikacje jak i migrującego bazę do nowych systemów. W celu utworzenia relacyjnej bazy danych dla aplikacji, możliwe jest oparcie jej o jedno z trzech dostępnych podejść dostarczanych przez omawianą technologię. Poniżej zostaną zaprezentowane każde z nich.

* **Database First**

Jest najbardziej spopularyzowanym podejściem ze względu na jego powszechność i standardowość. W tym modelu użytkownicy wychodzą od utworzenia bazy w typowym stylu, a mianowicie pisząc relacyjną bazę danych w języku SQL, w których skład wchodzą zapytania tj. *DDL* (ang*. Data Definition Language*), odpowiadające za strukturę bazy. Następnie, użytkownik przystępuje do generacji kodu obiektowego. Proces tworzenia bazy od strony developera odbywa się dzięki kreatorowi dostarczonemu przez Visual Studio, w którym użytkownik musi wybrać wcześniej utworzoną bazę w SQL, po czym generator na jej podstawie tworzy obiektowo odwzorowanie bazy wraz z relacjami. Dodatkowo oprócz plików związanych z tabelami, znajdą się tu również pliki zawierający kontekst do tabel dzięki którym developer ma możliwość odwoływania się w kodzie do każdej z nich.

* **Code First**

Polega ono na tworzeniu bazy wychodząc w pierwszej kolejności od obiektowej części. Tabele tworzone są w oddzielnych plikach *.cs*, natomiast relacje są dodawane w dwóch możliwych podejściach. Pierwszy z nich to racji przy wykorzystaniu adnotacji oraz metod abstrakcyjnych określających połączenie z kolejną tabelą. Adnotacje wykorzystuje się w celu wskazania między innymi klucza obcego (*ForeignKey*), czy klucza głównego (*PrimaryKey*). Dodatkowo przy użyciu adnotacji nadaje się właściwości dla pól, tj. liczbę dopuszczalnych znaków, wymagalność, typ. Drugim możliwym podejściem tworzenia relacji jest wykorzystanie tzw. *Fluent Api.* Polega ono na wskazaniu relacji między klasami tabel w oddzielnym pliku. W tym podejściu nie stosuje się adnotacji - oczywiście za wyjątkiem właściwości pól. Po utworzeniu tabel i relacji, developer w celu utworzenia lub zaktualizowania bazy na serwerze musi wykonać migracje w dodatku *Visual Studio* o nazwie *Package Manager Console.* Migracje dodaje się przy pomocy komendy *Add-migration name-„<nazwa migracji>”*, z kolei samą aktualizację wykonuje się podając *Update-database.* Jeżeli potrzebne jest wygenerowanie skryptu z aktualizacji bazy, musimy po komendzie dodać słowo *– verbose.*

* **Model First**

W tym podejściu tworzenia relacyjnej bazy danych, wykorzystujemy specjalny generator dostarczony w Visual Studio. Jest to najprawdopodobniej najłatwiejsza wersja ze wszystkich trzech podejść, ze względu na prostotę generacji tabel i ich relacji. W generatorze użytkownik dodaje nową tabele klikając prawym przyciskiem myszy na diagramie bazy. Następnie we właściwościach wcześniej utworzonej encji dodawane są kolejne pola o sprecyzowanych typach. Po utworzeniu przynajmniej dwóch tabel możliwe jest dodanie relacji miedzy nimi. Wszystkie zmiany wprowadzone w tabelach dotyczące na przykład pól czy relacji, prezentowane są na diagramie bazy danych. W celu zapisania projektu lub jego utworzeniu na serwerze, użytkownik musi wygenerować odpowiedni skrypt SQL. Taka generacja skryptu jest również łatwa do przeprowadzenia ponieważ wykonuje ją generator, po czym taki skrypt jest uruchamiany na serwerze bazodanowym tworząc na nim wcześniej utworzoną bazę.

### Klasa kontekstowa Entity Framework

Dostęp do bazy danych w technologii ORM możliwy jest przy zastosowaniu obiektowego stylu programowania. Aby umożliwić dostęp i korzystanie z bazy danych konieczne jest utworzenie obiektu klasy tzw. *Kontekstowej,* dziedziczącej po klasie *DbContext* dostarczonej przez bibliotekę Entity Framework.

W ciele tej klasy dodaje się jako pola kolekcje tabel utworzonych wcześniej jako model *CodeFirst.* Kolekcja reprezentująca jest typu *DbSet*, a przykładowa składnia wygląda następująco: public DbSet<Advert> adverts { get; set; }. W przypadku jeśli wybrano dodawanie relacji w bazie przy użyciu FluentApi, a nie poprzez adnotacje i metody wirtualne, wtedy do klasy dodajemy metodę - protected override void OnModelCreating(DbModelBuilder modelBuilder), gdzie rozpisywane są konkretne relacje.

Aby utworzony kontekst wiedział z jakiej bazy powinien korzystać, należy odwołać się w konstruktorze do klasy bazowej i podać we właściwości *name* nazwę bazy, która znajduje się w sekcji *ConnectionString* w pliku konfiguracyjnym. Składnia takiego konstruktora wygląda następująco:

*public EFDbContext() : base("name=BoatsAdverts")*, natomiast sekcję konfiguracyjną zapisuje się w taki sposób:

*<connectionStrings>*

*<add name = "UnitOfWork.Properties.Settings.BoatsAdvertsConnectionString" connectionString="Data Source = (LocalDb)\v11.0; AttachDbFilename=&quot; C:\Users\Mateusz\Desktop\App\App\_Data\BoatsAdverts.dbo&quot;; Initial Catalog = BoatsAd; Integrated Security=True " providerName = "System.Data.SqlClient" />*

*</connectionStrings>*

Konstruktor jest również wykorzystywany do ustawiania inicjalizatora. Dzięki niemu wskazuje mu co ma się zadziać po stronie bazy przy każdej kolejnej kompilacji kodu. Entity Framework udostępnia cztery takie inicjalizatory. Pierwszy z nich *–* CreateDatabaseIfNotExists, umożliwia utworzenie nowej bazy danych jeżeli wieszcze taka o konkretnej nazwie nie istniej. Druga – DropCreateDatabaseIfModelChanges, za każdym razem jeżeli model bazy jest zmieniony, EF usuwa całkowicie dane i bazę, a następnie od nowa ją zakłada. Trzeci inicjalizator – DropCreateDatabaseAlways, za każdym razem bez względu na zmianę lub nie modelu, uzuwa baze i tworzy ją na nowo. Ostatni inicjalizator - Custom DB Initializer: daje użytkownikowi całkowitą dowolność, ponieważ możemy w tym przypadku dodać swój własny inicjalizator jeśli żaden z powyższych nie spełnia wymaganych oczekiwań.

W miejscu aplikacji gdzie chcemy odwołać się do danej tabeli musimy wcześniej utworzyć obiekt takiej klasy kontekstowej, a następnie dla niego odwoływać się do poszczególnych kolekcji (tabel), przy użyciu technologii *Linq* będącej składnikiem języka C#. Warto jednak wykorzystać w odwoływaniu się do kontekstu przy pomocy wzorca projektowego *Dependency Injection*, dzięki któremu nie będzie konieczne za każdym razem tworzenia nowego obiektu.

# Środowisko testowe

Na rynku polskim istnieje wiele aukcji internetowych umożliwiających sprzedaż wszelakiego rodzaju przedmiotów używanych jak i nowych począwszy od biżuterii, a skończywszy na samochodach. Największymi na rynku aplikacjami zajmującymi się taką sprzedażą są m.in. *Allegro, Olx,* czy też nowo wchodzący na krajowy rynek *Amazon.* Wymienione aplikacje przeznaczone są dla wszystkich możliwych kategorii, dlatego próżno jest szukać odpowiednich pod kategorii dla pewnego drobiazgu zaliczanego do osprzętu żeglarskiego, w związku z czym użytkownicy nie mogą lub mają problemy z wystawianiem swoich przedmiotów, natomiast jeśli przedmiot zostanie dodany wtedy potencjalny kupiec ma problemy w jego odnalezieniu. Wychodząc naprzeciw zapotrzebowaniu rynku, powstał pomysł wykonania innowacyjnej aplikacji wykorzystywanej do ogłoszeń przedmiotów tylko i wyłącznie związanych z tematyką żeglarską.

Aplikacja ta ma pełnić rolę aukcji internetowej. Jej użytkownicy mają możliwość wystawiania jachtów żaglowych, motorowych, silników zaburtowych oraz stacjonarnych, przyczep podłodziowych i wszelkiego rodzaju osprzętu żeglarskiego. W celu wystawienia takiego ogłoszenia, osoba musi być zalogowanym użytkownikiem serwisu. Po wypełnieniu formularza oraz dodaniu zdjęć przedmiotu, użytkownik wybiera okres ważności ogłoszenia oraz dodatkowe funkcjonalności takie jak wyróżnienie ogłoszenia, które jest dodatkowo płatne. Dzięki możliwość wykorzystywania zewnętrznego system płatności internetowego *PayU* aplikacja może automatycznie przekierować użytkownika do swojego banku gdzie wykona przelew za naliczone opłaty związane z usługami świadczonymi przez system.

## Wymagania funkcjonalne

* Aplikacja umożliwia wystawianie do sprzedaży przedmiotów żeglarskich.
* System powinien posiadać graficzny interfejs użytkownika składający się dwóch widoków: kliencki, oraz administratorski.
* Widok administratorski ma możliwość bezpośredniego dodawania użytkowników indywidualnych jak i firmowych bez weryfikacji mailowej.
* Każdy z użytkowników ma możliwość przeglądania ofert wystawionych na stronie.
* Zarejestrowany użytkownik ma możliwość wystawienia ogłoszenia związanego z przedmiotem lub ogłoszenia związanego z usługą.
* Aplikacja powinna prezentować dane kontaktowe do administracji serwisu w razie ewentualnych niejasności używania systemu.
* Każdy użytkownik ma mieć wgląd do instrukcji dodawania przedmiotu na stronie.
* Logowanie może odbywać się poprzez wcześniej utworzone konto w aplikacji bądź zalogowanie się przy użyciu zewnętrznego konta (np. konto Google).
* Kreator ogłoszenia powinien umożliwiać dodawanie zdjęć wystawianego przedmiotu.
* Użytkownik w ramach jednej rejestracji zgłoszenia może wystawić tylko jeden przedmiot.

## Wymagania niefunkcjonalne

* Użytkownik może dodać ogłoszenie po wcześniejszym zarejestrowaniu, lub zalogowaniu do systemu.
* Rejestracja użytkownika jest możliwa po wprowadzeniu wymagalnych pół i uwierzytelnieniu mailowym.
* Pola wymagalne przy rejestracji to login będący adresem mailowym, oraz hasło powtórzone dwukrotnie zawierające małe i duże litery, znaki oraz cyfry.
* Zalogowany użytkownik ma możliwość dodania przedmiotu dopiero po poprawnym wypełnieniu formularza, którego pola są odpowiednio walidowane.
* Wgląd do produktów serwisu internetowego ma każdy użytkownik niezależnie czy jest on zalogowany, czy też nie.
* Zdjęcia dodawane w kreatorze ogłoszenia, muszą być walidowane pod względem ich ilości, rozmiaru plików oraz rozszerzenia plików. Walidacja musi odbywać się po obu częściach aplikacji tj. klienckiej oraz serwerowej.
* Modele dostarczane do kontrolerów na serwer muszą być odpowiednio walidowane w taki sam sposób ja po stronie klienckiej.

## Opis bazy danych aplikacji

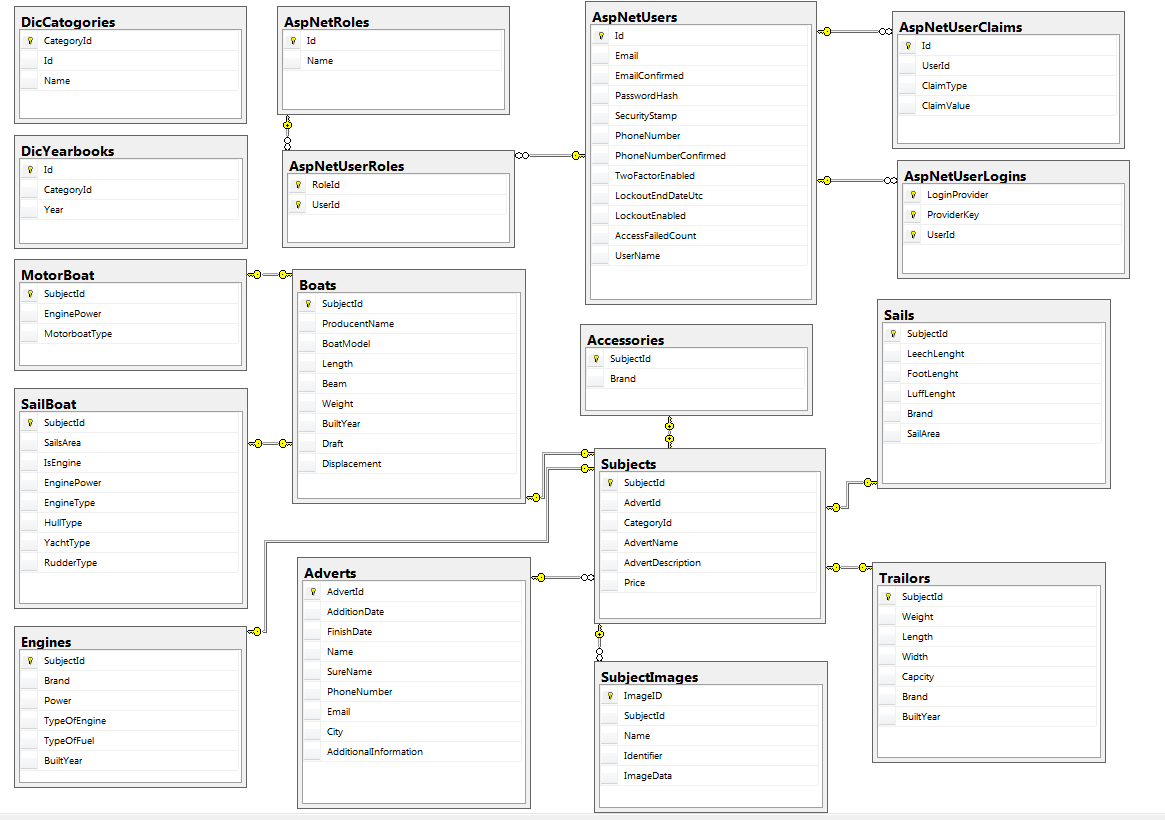
Baza danych na której przeprowadzano badania została zaprojektowana w technologii ORM przy użyciu biblioteki Entity Framework. Przy projektowaniu wykorzystano podejście Code-First, którego działanie dokładniej zostało opisane w rozdziale *2.2.1 Rodzaje podejść tworzenia bazy.*

W skład bazy wchodzą dwie kluczowe podsekcje wykorzystywane w systemie. Pierwsza z nich odpowiada za logowanie, natomiast kolejna jest wykorzystywana do kategoryzowania produktów wraz z ich słownikami. W celu umożliwienia logowania użytkowników do systemu, baza danych zaopatrzona jest w szereg tabel odpowiedzialnych za tą czynność. Zawierają one podstawowe informacje na temat użytkownika, którym może być klient jak i administrator aplikacji. W skład danych takiej osoby wchodzą między innymi dane kontaktowe (mail, numer telefonu), informacje potrzebne do logowania (nazwa użytkownika, hasło), informacje statystyczne dotyczące prób logowania (licznik ilości prób logowania, data wylogowania). Ze względu na to, że w aplikacji konieczne jest rozróżnienie użytkowników w celu poszerzenia dla nich funkcjonalności wprowadzono dodatkowo słownik ról. Na obecny stan aplikacja posiada zaimplementowane dwie role - Administrators, RegisteredUsers, które nadawane są dla następujących przypadków.

Prawa administratorskie nadawane są dla użytkowników zarządzających całym systemem oraz mogącym dodawać i sterować wszystkimi przedmiotami znajdującymi się w systemie. Takie uprawnienia może wprowadzać tylko i wyłącznie programista bądź developer wspierający aplikacje poprzez odpowiedni wpis do bazy danych.

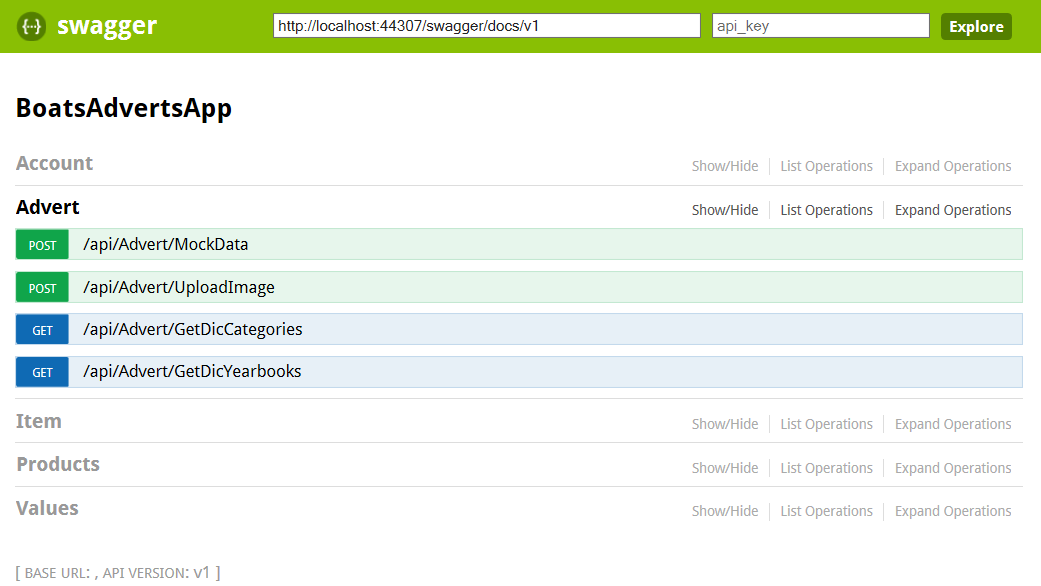
Kolejną rolę – RegisteredUsers, nadaję automatycznie system dla nowo zarejestrowanych użytkowników. Mają oni dostęp do podstawowych danych informacyjnych znajdujących się na stronie, oraz umożliwia się im dodawanie i zażądanie tylko swoimi przedmiotami.

Druga sekcja, która wchodzi w skład relacyjnej bazy danych przeznaczona jest do zarządzania wstawianymi przedmiotami w aplikacji. Główną tabelą opisującą przedmiot jest tabela *Advert* zawierająca podstawowe pola na temat ogłoszenia takie jak data jego dodania, przewidziana data jego zakończenia, imię i nazwisko właściciela, telefon kontaktowy, mail, miasto oraz informacje dodatkowe. Tak stworzone ogłoszenie może posiadać wiele przedmiotów, które opisuje tabela *Subjects*. Pola tu zawarte opisują dodatkowe informacje na temat ogłoszenia oraz przedmiotu tj. nazwa ogłoszenia, opis przedmiotu, cena. Dla przedmiotu użytkownik ma możliwość wstawienia wielu zdjęć prezentujących ofertę. Reprezentuję to tabela *SubjectImages*. Dla nazwy zdjęcia przeznaczone jest pole *Name*, do przechowywania meta danych pliku – *ImageData*, natomiast identyfikator zdjęcia – *Identifier.*



## Testy aplikacji

Zgodnie z dobrą praktyką wytwarzania oprogramowania, stworzona aplikacja została dogłębnie przetestowana na wypadek niechcianej ingerenci do systemu przez niepowołane do tego celu osoby, oraz na wypadek nieprawidłowego zachowania się funkcjonalności. Każda użyta metoda wykorzystywana do działania systemu używa technologii RESTapi w celu jej udostępnienia. Dla testów wykrywających poprawne działanie, czyli sprawdzenie, czy dana metoda zwraca żądany wynik, należy ją każdorazowo uruchomić – takie testy noszą nazwę testami jednostkowymi. Ciężkim podejściem byłoby uruchamianie za każdym razem aplikacji i wyszukiwanie w interfejsie graficznym żądanej funkcjonalności. Aby usprawnić ten proces, wykorzystano zewnętrzną bibliotekę open-source o nazwie *Swagger.* Dzięki niej możliwe jest z jednego poziomu interfejsu graficznego uruchamianie każdej metod z danej klasy zawartej w aplikacji. Po odpowiednim skonfigurowaniu narzędzia, uruchomienie go odbywa się poprzez dodanie na końcu adresu url wpisu */swagger,* gdzie po zatwierdzeniu użytkownik zostaje przekserowany panelu głównego. Na rysunku 1.1 załączono zrzut ekranu opisywanego dodatku. Wylistowane są tu wszystkie klasy używane w api systemu, po najechaniu na jedną z nich rozwija się jej zawartość. W omawianym przykładzie rozwinięto klasę *Advert*, która zawiera w sobie cztery metody – *MockData, UploadImage, GetDicCategories,* oraz *GetDicYearbooksn,* po kliknięciu kolejny raz w jedną z nich, możliwe będzie wprowadzenie parametrów o ile je przyjmuje. Jeśli wszystkie dane zostały wypełnione, możliwe jest już przejście do uruchomienia danego żądania wybierając przycisk *Try it out!.* Jako rezultat wykonanej metody pojawią nam się kolejne cztery sekcje – *Request URL, Response Body, Response Code, Response Header.* Wykorzystanie takiego narzędzie znacznie przyśpiesza prace testera bądź developera chcącego na szybko przetestować daną część systemu. Pomimo aspektów testowych, możliwe jest również użycie Swagger-a jako źródło dokumentacji programu. W jasny i przejrzysty sposób wypisuje specyfikację wewnętrzną aplikacji, zbierając ją w jednym miejscu ze wszystkimi klasami, metodami, oraz parametrami.



Rysunek .1

Nie zabrakło również najważniejszego aspektu każdych testów, a mianowicie testów jednostkowych. Ten rodzaj testów jest przeprowadzany przez testerów bądź deweloperów bezpośrednio na kodzie źródłowym. Ich celem jest zweryfikowanie poprawności działania danego fragmentu kodu oraz jego zabezpieczenie przed niewłaściwym użyciem. Testy powinny zostać pokryte w najlepszym przypadku dla całości kodu, wyjątkiem mogą jedynie być fragmenty napisane przez generatory, czyli m.in. *gettery* oraz *settery*.

Na poczet powyższych testów stworzono osobny projekt *UnitTest* w programie *Visual Studio*. Do ich przeprowadzenia wykorzystano bibliotekę **…**. Jest ona napisana pod technologię .NET, przez co możliwe jest w łatwy i szybki sposób skonfigurować ją pod własne potrzeby. Do przetestowania metod zawartych w aplikacji korzystającej z bazy danych, konieczne jest jej zasymulowanie potocznie nazywane *mokowanie*. Do tego celu użyto kolejnej biblioteki *Moq*, udostępnionej przez pakiet dodatków Visual Studio – Nuget, której zadaniem jest zasymulowanie bazy danych wraz z przykładowymi danymi. Konieczne jest również napisanie niezbędnych metod wykorzystywany w trakcji zapytań w technologii EntityFramework, dla tego utworzono nowy plik z potrzebnymi metodami. Jedyną przeszkodą na jaką można jeszcze natrafić w trakcie pisania testów, jest dostosowanie ich pod wzorzec projektowy *Depency Injection*, który został wykorzystany w programie. Jego zadaniem jest dociągnięcie dodatkowych zależności w momencie tworzenie nowego obiektu klasy, czyli w opisywanym przykładzie będą to dwa parametry – interfejs IMapper i interfejs IEFDbContext umożliwiające odpowiednio mapowanie danych z obiektu do obiektu oraz dostęp do tabel w technologii ORM. Aby możliwe było testowanie kontrolerów korzystających z opisanego wzorca, konieczne jest dodanie do programu testującego odpowiednich parametrów używanych przez taką klasę. W bibliotece **…** obywa się to w taki sposób jak pokazano poniżej.

Po takim skonfigurowaniu środowiska możliwe jest przystąpienie do pisania testów jednostkowych dla kontrolera wykonującego zapytania bazodanowe. Zgodnie z ich ideą, każde z nich zostało przeprowadzone pod kątem weryfikacji poprawności zwracanych typów, wartości, czy też sprawdzenia poprawności zakresu danych. Wszystkie testy w aplikacji zostały wykonane z wynikiem pozytywnym.

# Część badawcza

Praca badawcza ma za zadanie wykazać jak wydajne są środowiska zarządzające bazą danych, oraz sprecyzować dlaczego jedno jest lepsze od drugiego. Jako środowiska badawcza w pracy wykorzystano – SQL Management Studio, używane do zapytań w języku SQL, oraz biblioteka ORM Entity Framework w którym zapytania wykonujemy przy użyciu technologii obiektowej.

Testy wykonywano na Laptopie Lenovo ThinkPad W530, wyposażony w procesor Intel i7 3630QL o taktowaniu 2.40 GHz, oraz pamięć RAM o rozmiarze 16 GB. Jako system operacyjny wykorzystano Windows 7 w wersji 64-bitowej. Warto również zaznaczyć, że stacja badawcza była podczas każdych testów podłączona do zasilenia, w przeciwnym wypadku jeśli komputer działał by tylko na baterii, wyniki mogłyby być przekłamane.

## Metodyka badań/Plan eksperymentów/testów

W celu wykazania, które z dwóch badanych środowisk jest wydajniejsze, należało przeprowadzić szereg zapytań bazodanowych. Zapytania te polegały na przetworzeniu konkretnej ilości rekordów, oraz dostępu do nich na określonym zagłębieniu rekordu w relacyjnej bazie danych. Tak przygotowane zapytania uruchamiano w trzech możliwych podejściach (w SQL management studio, ORM – przy pomocy Entity Framework oraz czystym języku SQL uruchomionym w aplikacji .NET), poczym weryfikowano które z nich jest wydajniejsze.

Wyliczanie czasu potrzebnego na otrzymanie odpowiedzi z serwera w przypadku SQL Management Studio otrzymywano za pomocą wbudowanej funkcjonalności nazywającej się *ClientStatistic.* Jest to narzędzie prezentujące statystyki dotyczące wykonanych każdorazowo zapytań. W skład statystyk wchodzą cztery kluczowe sekcje. Pierwsza z nich nosi nazwę *Query Profile Statistic*, informująca o której godzinie dane zadanie zostało wykonane. Durga – *Query Profile Statistic* prezentuje dane na temat otrzymanych danych (tj. iloć zwróconych rekordów, ilość zmian typu *Insert, Update, Delete,* oraz liczba transakcji). Kolejna sekcja – Network Statistic opisuje zachowania sieci komputerowej w trakcji wykonania zapytania (podaje liczbę dostępów do serwera, ilość bajtów wysłanych od klienta oraz otrzymanych przez serwer, oraz ilość paczek TDS wysłanych do klienta i otrzymanych przez serwer). Ostatnia sekcja – czwarta, podaje statystyki czasowe. Jest ona kluczowa dla przeprowadzanych testów ze względu na jej zawartość, a dokładniej na zwracane pole *TotalExecutionTime*. Dzięki niemu wiadomo ile czasu zajmuje systemowi zwrócenie żądanego wyniku. Na podstawie tej wartości możliwe jest oszacowanie, które zapytanie jest wydajniejsze. Dla wszystkich wymienionych pól istnieje kolumna o nazwie *Trial n* do której wstawiane są odpowiednie wartości kolejnych zapytań. Ponadto dodatkowym atutem omawianego narzędzia jest możliwość prezentacji średniej wartości z otrzymanych wyników.

W przypadku zapytań ORM oraz SQL w aplikacji .NET czas potrzebny na odpowiedź wyliczano przy pomocy języka C# i jego wbudowanych bibliotek. W tym celu, korzystno z klasy wyliczeniowej Stopwatch, dla której nowo utworzony obiekt zbierał dane na temat czasu wykonania danej operacji. Klasa ta pochodzi z biblioteki System.Diagnostics. Aby obiekt mógł rozpocząć wyliczanie czasowe należy wywołać metodę StartNew(). Po takim użyciu można rozpocząć wykonywanie badanych operacji bazodanowych. Następnie po zakończeniu tej sekcji, na wcześniej utworzonym obiekcie klasy Stopwatch należy zakończyć obliczanie poprzez uruchomienie metody Stop(). W celu pobrania czasu potrzebnego na wykonanie powyższej operacji, należy odwołać się do pola obiektu Elapsed.TotalMilliseconds. Przykładowy kod obliczający czas wykonania testowej metody zamieszczono poniżej.

Stopwatch stopWatch = Stopwatch.StartNew();

TestMethod();

stopWatch.Stop();

Technologia ORM wykonująca zapytanie utworzone w języku obiektowym musi przejść przez własny translator. Zadaniem takiego mechanizmu jest, aby zamienić owy język na czysto SQL-owe zapytanie bazodanowe.

Poniżej zaprezentowano składnie wypisującą w konsoli *Debug* treść przetworzonego zapytania w strukturalnym języku zapytań.

this.Database.Log = s => System.Diagnostics.Debug.WriteLine(s);

## Konfiguracja środowiska testowego/Środowisko testowe

W celu wykonania badań, należało wcześniej zaprojektować obiekt na którym zostałyby później przeprowadzone. Obiektem badań tej pracy jest aplikacja internetowa korzystająca z relacyjnej bazy danych. Projekt został w głównie mierze wykonany w oparciu o technologie wytwarzane i wspierane przez firmę Microsoft oraz źródła ogólnie dostępne (ang. open source).

Ze względów na łatwość i szybkość pobierania dodatkowych bibliotek do aplikacji, zdecydowano o wykorzystywaniu   Visual Studio w wersji Enterprise 2015, jako [zintegrowanego środowiska programistyczne](https://pl.wikipedia.org/wiki/Zintegrowane_%C5%9Brodowisko_programistyczne)go. Z powodu, iż badania przeprowadzano na aplikacji internetowej, wykorzystano w tym celu platformę ASP.NET, która umożliwia w Visual Studio tworzenie zaawansowanych serwisów internetowych. Projekty w tej technologii są tworzone w oparciu o obiektowy styl programowania dodatkowo daje możliwość odseparowania części serwerowej od części prezentacyjnej (klienckiej), przez co zwiększona jest przejrzystość kodu, oraz jego wydajność. Ponadto możliwe jest dzięki Visual Studio w krótkim czasie wykonać publikację aktualnej wersji projektu na serwer gdzie znajduję się jego hosting. Jako bazę danych wykorzystano MS SQL Server, z powodu dobrej komunikacji z Visual Studio oraz dobrego wsparcia technicznego ze strony wydawcy. Do zarządzania wszystkimi komponentami dostarczonymi z serwerem wykorzystano zintegrowane środowisko SQL Management Studio. Dzięki niemu możliwe jest podejrzenie tabel, przejrzenie danych oraz generowanie diagramu tabel wraz z relacjami zawartymi miedzy nimi. Baza danych została wygenerowana wraz z relacjami dzięki technologii opartej o Mapowanie obiektowo-relacyjne (ORM). Do jej generacji wykorzystano narzędzie Entity Framework. Technologia ta zostanie dokładniej opisana w jednym z wcześniejszych rozdziałów (2.1 Technologia Entity Framework). Dla oprogramowania części klienckiej użyto Framework javascript – knockout.js wraz z dodatkiem Durandal.js który wykorzystuje wzorzec projektowy MVC, dzięki czemu w łatwy sposób możliwe jest połączenie go z wcześniej opisywaną platformą ASP.NET. Kolejnym argumentem determinującym wybór jest stabilność platformy Duranda.Js oraz frameworka Knockout.js. Dla usprawnienia interakcji użytkownika z aplikacją oraz rozszerzenia funkcjonalności powyższego frameworka wykorzystano bibliotekę jQuery dla języka JavaScript. Dzięki niemu możliwe są wszelkiego rodzaju akcje między innymi na każdym z elementów DOM (tj. Obiektowy model dokumentu, ang. Document Object Model). Komunikacja między częścią kliencką, a serwerową odbywa się dzięki żądaniom AJAX do WebApi będącego częścią platformy ASP.NET. Do samej części prezentacyjnej zastosowano gotową bibliotekę styli CSS – Bootstrap. W delikatnej mierze zostały zmodyfikowane i przystosowane do wymagań wyglądu strony.

Spis technologii wykorzystanych w projekcie oraz w badaniach:

* Visual Studio Professional 2015
* SQL Server wraz z SQL Server Management Studio
* Entity Framework
* Knockout.js
* Durandal
* Jquery
* Ajax
* Bootstrap

## Zbiory danych

Przeprowadzanie badań w niniejszej pracy byłoby niemożliwe bez przykładowych danych testowych. Próbowano znaleźć udostępnione w sieci przykładowe spisy danych możliwe do wykorzystania w pracy, starano się również pozyskać je z firm specjalizujących się w produkcji takich produktów. Niestety poszukiwania nie dały żadnych rezultatów, przez co konieczne było wygenerowanie ich samodzielnie. W zawiązku z czym, napisano specjalną metodę generującą dane, która znajduje się w kontrolerze aplikacji. Proces generacji odbywa się poprzez wpisanie adresu [http://localhost:44307/api/Advert/MockData/{id}](http://localhost:44307/api/Advert/MockData/%7bid%7d) do przeglądarki internetowej, gdzie jako parametr *id* podaje się liczbę rekordów do wstawienia do bazy. Na czas testów, wykonano 10 tyś. żądań dodających rekordy na serwer.

Dane, które są dodawane do bazy przez metodę *MockData*, reprezentują ogłoszenie produktu. W skład takiego ogłoszenia wchodzą z reguły od czterech do pięciu encje, główne z nich to tabela *Adverts, Subjects, SubjectImages, Boats,* oraz dodatkowo jedna z tabel *Sails, SailBoat, MotorBoat, Engines, Trailors,* i *Accessories.* Wykonywanie działań bazodanowych odbywa się tu przy pomocy technologii ORM. Ilość żądań dodających rekordy determinowana jest przez parametr metody, który jest podstawiany do pętli *for* odpowiedzialnej za ową czynność*.* Aby sprawić żeby dane testowe nie były ujednolicone, wykorzystano metodę *Next* z klasy *Random* języka C#, zwracającą losową wartość z danego przedziału. Jeśli oczekiwana jest wartość całkowita, w takim przypadku składnia wyglądała by następująco: new Random().Next(1, 6), natomiast jeśli potrzebna jest wartość znajdująca się w tablicy, na przykład typu *string* wtedy należało by zapisać składnie w taki sposób: Names[new Random().Next(0, 20)], gdzie tablica *Names* jest typu *string* o rozmiarze 20. Jak wspomniano już wcześniej, metoda dodaje do bazy dane związane z ogłoszeniem przedmiotów, aby to było możliwe wykorzystano w tym celu warunek *switch – case* dzięki któremu wcześniej wybrana losowa wartość całkowita z przedziału od 1 do 6 wstawia przedmiot (kolejno silnik, przyczepa, łódź żaglowa, łódź motorowa, żagiel) na serwer. Kod warunku zaprezentowano poniżej.

int rand = new Random().Next(1, 6);

…

switch (rand)

{

case n:

break;

case n+1:

break;

…

}

Do danych testowych można również zaliczyć dodatkowe rekordy utworzone podczas przeprowadzania testów wydajnościowych których celem było sprawdzenie szybkości zapytań dodających dane do bazy. Jednakże na tle wygenerowanych wcześniej 10 tyś. rekordów, są to znacznie mniejsze wartości.

## Uzyskane rezultaty/Wyniki

Proces badawczy miał za zadanie zbadać w jak długim czasie dane żądanie zwróci wynik. Zapytania były odpytywane z poziomu technologii ORM jak i również języka SQL, w obu przypadkach oczekiwany wynik miał być taki sam. Do ich uruchamiania wykorzystano język C# wraz z dodatkowymi bibliotekami umożliwiającymi ich przeprocesowanie (opis wszystkich dodatków został opisany w rozdziale *4.2 Konfiguracja środowiska testowego/Środowisko testowe*). Część przeprowadzanych zapytań była wykonywana na stałych wartościach, czyli na przykład zapytanie dodające wartości do tabeli wstawiały 1000 rekordów na serwer po czym następowało weryfikowanie ich zajętość czasowej z drugim zapytaniem. Kolejno wykonywano żądania ilościowe polegające na wykonaniu tego samego zapytania dla coraz to większej porcji danych, a następnie ich powtórzeniu przez pięć razy. Dzięki takiemu badaniu można zweryfikować jak zachowuje się system przy większym obciążeniu danymi.

W pracy badano również wydajność oraz poprawność zapytań generowanych przez sam silnik technologii Entity Framework. Polegało ono na pobraniu zapytania wygenerowanego przez omawianą bibliotekę, następnie skopiowaniu jej do programu *SQL Management Studio*, gdzie dla odpowiednich wartości uruchomiano je wraz z pobieraniem czasu wykonania. Po zapisaniu otrzymanych rezultatów pisano własnoręcznie odpowiednik wygenerowanego zapytania, poczym powtarzano czynności sprawdzające czasy ich przeprocesowania. Ostatnim krokiem dla tego typu badań, było zweryfikowanie, czy aby na pewno Technologia ORM tworzy wydajne zapytania bazodanowe.

1. Dla każdego z testów wykonano od pięciu do dziesięciu zapytań, poczym wyciągano z nich średnią arytmetyczną. Taki zabieg miał na celu wyeliminowanie różnego rodzaju przekłamań spowodowanych niewłaściwą pracą systemu, zajęciem danego wątku przez inną aplikację, czy też przegrzaniem komputera. Dopiero po takim wyliczeniu możliwe jest porównanie między sobą danych zapytań. W ciągu całego procesu badawczego przeprowadzono piętnaście badań dla każdej z dwóch gałęzi zapytań, ich wyniki zaprezentowano poniżej wraz z opisem uzyskanych wyników. Dodatkowo do pracy został dołączony plik *[1] Testy wydajnościowe.pdf,* zawierający wszystkie badania wraz z wykonanymi pomiarami.

### Zapytanie do pojedynczej tabeli

Zadanie to miało na celu wskazanie, czy przyjęta koncepcja wyliczania potrzebnego czasu na odpowiedź zadziałała prawidłowo. Możliwe było, że czasy mogły być znacznie różne od standardu co wiązałoby się z koniecznością innego podejścia do ustawienia startu, czy też stopu pomiarów. Pierwotnie ustawiono początek pomiarów na samym początku wykonywanego kodu, co niestety było błędnym podejściem. W zapytaniach SQL koniecznej jest za każdym razem podłączenie się do bazy danych, wskazanie parametrów startowych, następnie otwarcia połączenia, wykonanie żądania i na sam koniec zamknięcie połączenia. Natomiast w żądaniach ORM-owych nie ma potrzeby konieczności otwierania/zamykania połączenia z bazą, za tę czynność odpowiedzialny jest jego silnik, więc w tym przypadku piszemy tylko to co nas interesuje z bazy. Tak więc po analizie obydwóch podejść ustawiono czasy pomiarów przed samym zapytaniem oraz na jego zakończeniu w miejscu gdzie otrzymujemy wynik. W celu potwierdzenia poprawności metody wyliczającej zajętość czasową, napisano najprostsze z możliwych zapytań do tabeli zawierającej klauzule *where*. Żądanie miało pobrać z pojedynczej encji *Adverts*, informacje okrojone do warunku, gdzie data dodania rekordu była większa i mniejsza od zadanej wartości oraz pole miasto było równe „Katowice”. Takie zapytanie w zapisie SQL wygląda następująco.

SELECT \* from [BoatsAd].[dbo].Adverts where AdditionDate > '2018-05-29 17:32:00' and AdditionDate < '2018-05-29 17:35:00' and City = 'Katowice'"

Jak widać zapytanie nie należy do skomplikowanych, a zajętość nie jest również spora. Podobnie jest z zapytaniem w Entity Framework-u:

var newQuery = \_db.adverts.Where(x => x.AdditionDate > new DateTime(2018, 5, 29, 17, 32, 00) && x.AdditionDate < new DateTime(2018, 5, 29, 17, 35, 00) && x.City=="Katowice").ToList();

Powyższe zadania zostały uruchomione po dziesięć razy każde, następnie wyliczono z uzyskanych wyników średnią wartość. Po tak prostych poleceniach nie można się spodziewać sporych interwałów czasowych w odpowiedziach z bazy danych. Po uruchomieniu ich nie okazało się inaczej. Dla pierwszego zadania otrzymano odpowiedź już po 20,2 ms, a dla drugiego po 25 ms. Warto również wspomnieć, że dla obu przypadków operowano na takiej samej liczbie rekordów.

W trakcie tego badania tradycyjne zapytanie SQL-owe okazało się wydajniejsze, ORM w tym przypadku był gorszy o 4,8 ms. Nie jest to jednak wyznacznik mówiący o tym, że dany rodzaj zapytań jest lepszy od drugiego. W tym badaniu sprawdzano dane dla jednej wartości rekordów. Dla tego w dalszych rozdziałach będą rozpatrywane badania dla innych wolumenów danych jak i różnych operacji (insert, update).

### Dodanie pojedynczego rekordu

W tym teście badano zachowanie systemu podczas dodawania pojedynczego rekordu do bazy. Encja do której zostały wstawiane dane to tabela *Advert.* Nie wstawiano tu dodatkowych danych do tabel powiązanych z bazową, ze względu na próbę przebadania szybkości insercji pojedynczej krotki, nie mniej jednak w rozdziale 4.4.11 będą przeprowadzane test wstawiania danych do większej ilości encji.

Jako pierwszy przypadek dla tego badania zaprezentowany zostanie sposób podejścia dodawania danych w technologii ORM. Zgodnie z paradygmatem programowania obiektowego w celu dodania danego rekordu do bazy, konieczne jest utworzenie wcześniej żądanego obiektu, a następnie przypisania dla niego wartości. Poniżej zamieszczono kod w języku C# prezentujący utworzenie nowego rekordu potrzebnego do dodania do bazy.

Advert advert = new Advert

AdditionDate = new DateTime(2018, 5, 29, 17, 32, 00),

AdditionalInformation = "BRAK",

City = "Gliwice",

Email = "matstanik@gmail.com",

FinishDate = new DateTime(2018, 5, 29, 17, 35, 00),

Name = "TEST",

PhoneNumber = "485928375",

SureName = "TEST"

};

Jak widać, tworzenie nowej krotki jest bardzo przejrzyste i łatwe w interpretacji. Równie zgrabnie wygląda jej dodanie, oraz zapis na serwer który odbywa się odpowiednio poprzez składnię „var newQuery = \_db.adverts.Add(advert);”, oraz ”\_db.SaveChanges();”. W tym przypadku nadmiernym jest przypisywanie do zmiennej *newQuery* żądania *Add(advert)*, ze względu na brak konieczności korzystania z niej w dalszych testach, jednakże warto pokazać, że istnieje taka możliwość w celu późniejszego odwoływania się do dodanego rekordu. Oczywistą jest również rzeczą, iż przypisywanie do jakiejkolwiek zmiennej zapisu danych *SaveChanges()* nie ma tu racji bytu. Podczas insercji w technologii ORM nie ma potrzeby przypisywania kluczowi głównemu wartości, ponieważ silnik Entity Framework sam to za nas zrobi.

"INSERT INTO [dbo].[Adverts] ([AdditionDate], [FinishDate], [Name], [SureName], [PhoneNumber], [Email], [City], [AdditionalInformation]) VALUES ('2018-05-29 17:32:00', '2018-05-29 17:35:00', 'TEST', 'TEST', '485928375', 'matstanik@gmail.com', 'Gliwice', 'BRAK')";

W języku SQL, w tym jak i w kolejnych przypadkach nie ma żadnych większych zmian, które by odchodziły od standardowego podejścia zapytań. Mamy tu zwykłe przypisanie wartości po przecinku wylistowanym polom tabeli *Adverts*. Aby wyliczanie czasu w trakcie badań było poprawne w tym, jak i we wcześniejszym przykładzie podano takie same wartości dla obu przypadków, w przeciwnym razie mogły by wyjść minimalne anomalia czasowe.

Po dziesięciokrotnym wykonaniu obydwóch zapytań, oraz wyliczeniu średniej uzyskano wyniki, które pokazały przewagę zapytania napisanego w SQL, i tym razem wyniosła ona znacznie więcej niż w poprzednim badaniu. Zajętość pierwszego podejścia wyniosła 32,8 ms, natomiast drugiego 18 ms, co daje 14,8 ms różnicy. Warto również zwrócić tu uwagę na fakt iż obydwie wartości są dużo większe od testu z rozdziału 4.4.1, tak więc wykonanie innej operacji niż żądanie danych z serwera sprawia, że różnica staje się większa. Nie jest to jednakże wyznacznik, aby to potwierdzić konieczne jest przeanalizowanie jeszcze kilku innych testów.

### Wielokrotne dodanie rekordów

Jak już wspomniano w poprzednim rozdziale, błędnym jest na podstawie jednego rodzaju testu określić, czy dana operacja jest wolniejsza w jednym podejściu od drugiego. W związku z czym, w trakcie tego badania, przeprowadzono insercje do bazy tysiąca rekordów, z każdorazowym zapisem. Mierzenie czasu odbywało się tutaj tak jak dotychczas, a mianowicie przed tworzeniem nowego obiektu uruchamiano licznik, i po zakończeniu dodawania ostatniego rekordu i jego zapisie wyłączano go.

W przypadku ORM-a konieczne było wcześniejsze utworzenie obiektu tak samo jak w rozdziale *4.4.2.* Następnie dodawano go do bazy poczym go zapisywano. Wszystkie zmiany mają miejsce w pętli for, która wykonywana była tysiąc racy. Operacje były przeprowadzany tylko na pojedynczym obiekcie, dane po każdej insercji nie zmieniały swojej postaci. Kod zapisujący krotkę do bazy wygląda następująco:

for(int i =0; i<1000; i++){

var newQuery = \_db.adverts.Add(advert);

\_db.SaveChanges();

}

Przy zapisie danych po każdym dodaniu nowego rekordu Entity Framework jest delikatnie odciążany, w przeciwnym razie jeśli chciałoby się zapisać po dodaniu na raz wszystkich rekordów, możliwe było by znaczne przeciążenie generatora SQL.

### Pobranie wszystkich danych z tabeli

### Pobranie danych z dwóch tabel

### …

## Wnioski

|  |
| --- |
|  |

# Podsumowanie

# Bibliografia

1. http://nugetmusthaves.com/Category/ORM
2. [Freeman](http://helion.pl/autorzy/adam-freeman,adamfreeman.htm) A., *ASP.NET MVC 5. Zaawansowane programowanie*, Helion, Gliwice, 2015.
3. Gamma W., Helm R., Johnson R., Vlissides J., *Wzorce Projektowe Elementy oprogramowania obiektowego wielokrotnego użytku,* Helion, Gliwice, 2010.
4. Rockoff L., *Język SQL. Przyjazny podręcznik*, Helion, Gliwice, 2014.
5. Wiegers K., Beatty J., *Specyfikacja oprogramowania. Inżynieria Wymagań*, Helion, Gliwice, 2014.

# Spis skrótów i symboli

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

# Zawartość dołączonej płyty

1. Testy wydajnościowe.pdf

# Spis rysunków

# Spis tabel