Politechnika Śląska

Wydział Automatyki, Elektroniki i Informatyki

Kierunek Informatyka

inż. 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 10](#_Toc517208373)

[2. Technologia ORM 12](#_Toc517208374)

[2.1. Technologia Entity Framework 15](#_Toc517208375)

[2.1.1. Klasa kontekstowa Entity Framework 17](#_Toc517208376)

[3. Środowisko testowe 20](#_Toc517208377)

[3.1. Cel 22](#_Toc517208378)

[3.2. Opis bazy danych aplikacji 25](#_Toc517208379)

[3.3. Testy aplikacji 28](#_Toc517208380)

[4. Część badawcza 31](#_Toc517208381)

[4.1. Metodyka badań/Plan eksperymentów/testów 33](#_Toc517208382)

[4.2. Zbiory danych 36](#_Toc517208383)

[4.3. Uzyskane rezultaty 38](#_Toc517208384)

[Celem tego testu było zweryfikowanie sposobu zachowań dla zapytań pobierających dane z więcej niż jednej tabeli, oraz pobraniu z nich kilku atrybutów. Dodatkowo znajduje się tu również warunek zawężający zakres pobieranych danych o datę dodania i zakończenia wystawianego przedmiotu w systemie, dokładnie w taki sam sposób jak miało to miejsce w pierwszym badaniu o tytule *Zapytanie do pojedynczej tabeli.* Porównując obydwa zapytania można przypuszczać, że są one bardzo podobne jednakże przeprowadzane badania w tym teście sprawdzają zachowanie łączenia tabel przy wykorzystaniu relacji, natomiast w poprzednim zapytaniu odwoływano się tylko do pojedynczej encji. 43](#_Toc517208385)

[4.4. Wnioski 47](#_Toc517208386)

[5. Podsumowanie 48](#_Toc517208387)

[Bibliografia i](#_Toc517208388)

[Spis skrótów i symboli ii](#_Toc517208389)

[Zawartość dołączonej płyty iii](#_Toc517208390)

[Spis rysunków iv](#_Toc517208391)

[Spis tabel v](#_Toc517208392)

# 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 środowiska badawczego. Aby możliwe było rozpoczęcie testów 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>*

W miejscu konstruktora możliwe jest również wybranie odpowiedniego inicjalizatora pochodzącego z klasy I*nitializer*. Jego ustawienie odpowiada za czynności zachodzące przy każdorazowym uruchomieniu aplikacji. 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, usuwa bazę i tworzy ją na nowo. Ostatni inicjalizator - *Custom DB Initializer*: daje użytkownikowi całkowitą dowolność, ponieważ w tym przypadku możliwe jest dodawanie swojego własnego inicjalizator o ile żaden z powyższych nie spełnia wymaganych oczekiwań.

W miejscu aplikacji gdzie chcemy odwołać się do danej tabeli, konieczne jest utworzenie wcześniej obiektu takiej klasy kontekstowej, a następnie dla niego odwoływanie się do poszczególnych kolekcji (tabel), przy użyciu technologii *Linq* będącej składnikiem języka C#. W przypadku odwoływania się do kontekstu bazy danych warto jednak rozważyć dodanie wzorca projektowego *Dependency Injection*, któr został zaczerpnięty z pozycji [3]*.* Dzięki nimu nie ma konieczności przy każdorazowym dostępie do danych tworzenia nowego obiektu reprezentującego bazę danych. Raz przekazany kontekst w konstruktorze danej klasy będzie widoczny dla wszystkich metod, które potrzebują dostępu do danych. Przykładowe przypisanie kontekstu, zaprezentowano na klasie *ProductController* będącej jednym z kontrolerów aplikacji.

public class ProductsController : ApiController

{

IEFDbContext \_db;

public ProductsController(IEFDbContext db)

{

\_db = db;

}

}

# 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, zadecydowano 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 bibliotekę 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.

Podsumowując opis oprogramowania środowiska testowego, poniżej zamieszczono zestawienie technologii wchodzącej w jej skład.

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

Jeżeli chodzi o aspekt techniczny, badania przeprowadzano 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 zasilania.

## Cel

Na polskim rynku 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.
* 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.
* Każdy użytkownik ma wgląd do instrukcji dodawania przedmiotu na stronie.
* Kreator ogłoszenia umożliwia dodawanie zdjęć wystawianego przedmiotu.
* Użytkownik w ramach jednej rejestracji zgłoszenia może wystawić tylko jeden przedmiot.
* Użytkownik może przez aplikacje komunikować się ze sprzedawcą czy też klientem.
* Zarejestrowany użytkownik może przeglądać wszystkie oferty użytkowników lub tylko swoje będące aktualnie wystawione.

Wymagania niefunkcjonalne

* Użytkownik może dodać ogłoszenie po wcześniejszym zarejestrowaniu lub zalogowaniu do systemu.
* Rejestracja użytkownika jest możliwa po wprowadzeniu wymaganych pół i uwierzytelnieniu mailowym.
* Pola wymagane 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 od tego czy jest on zalogowany, czy też nie.
* Zdjęcia dodawane w kreatorze ogłoszenia, muszą być walidowane pod względem ich liczby, 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.
* System posiada graficzny interfejs użytkownika składający się dwóch widoków: kliencki, oraz administratorski.
* Logowanie może odbywać się poprzez wcześniej utworzone konto w aplikacji bądź zalogowanie się przy użyciu zewnętrznego konta (np. konto Google).
* Aplikacja ma prezentować dane kontaktowe do administracji serwisu w razie ewentualnych niejasności używania systemu.

## Opis bazy danych aplikacji

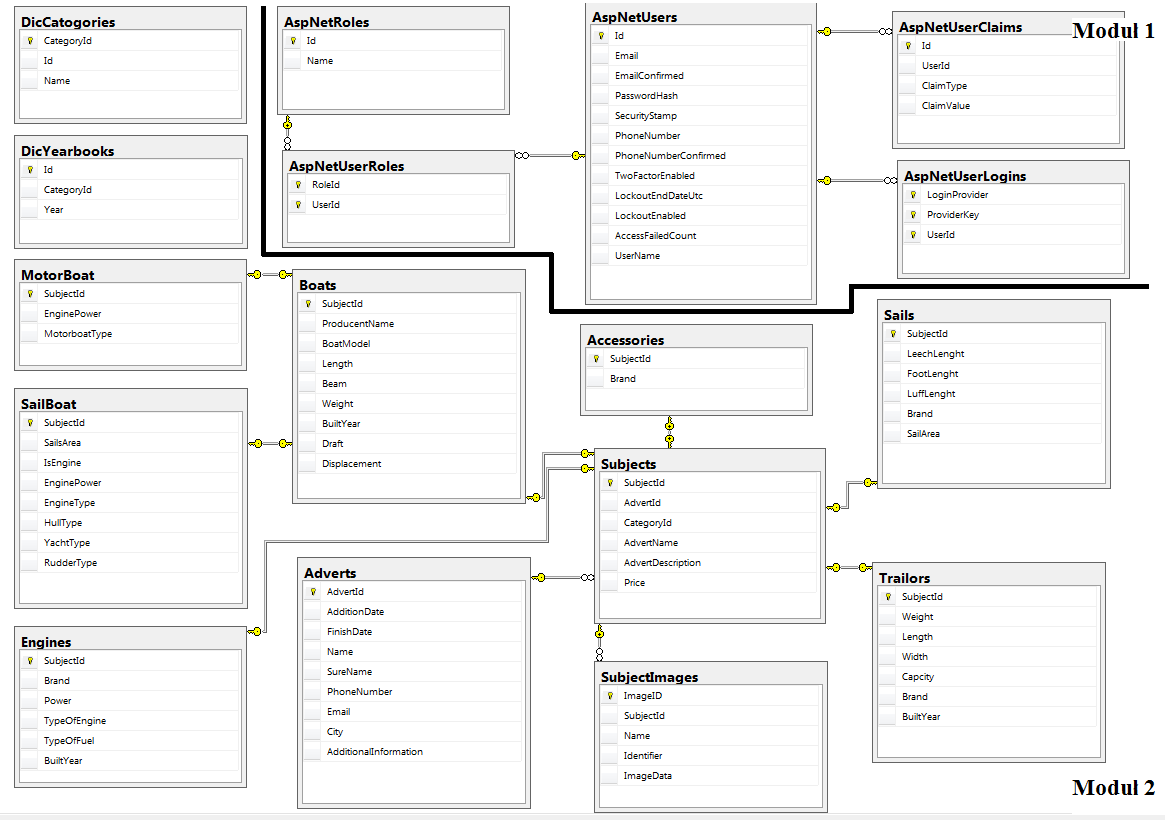
Baza danych na której przeprowadzano badania została zaprojektowana w technologii ORM przy użyciu biblioteki Entity Framework. Przy projektowaniu wykorzystano podejście Code-First, którego działanie dokładniej zostało opisane w rozdziale *2.2.1 Rodzaje podejść tworzenia bazy.*

W skład bazy wchodzą dwa kluczowe moduły 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*, system nadaje automatycznie 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.

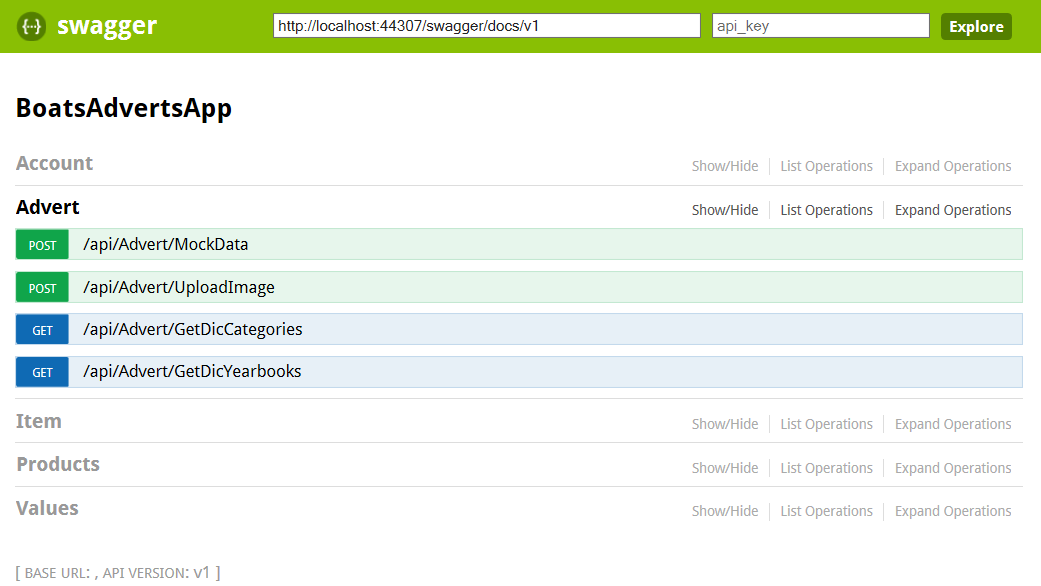
Druga sekcja, która wchodzi w skład relacyjnej bazy danych przeznaczona jest do zarządzania wstawianymi przedmiotami w aplikacji. Główną tabelą opisującą przedmiot jest tabela *Advert* zawierająca podstawowe pola na temat ogłoszenia takie jak data jego dodania, przewidziana data jego zakończenia, imię 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.*



Rysunek

## Testy aplikacji

Zgodnie z dobrą praktyką wytwarzania oprogramowania stworzona aplikacja została dogłębnie przetestowana na wypadek niechcianej ingerencji w systemie przez niepowołane do tego celu osóby 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. Niezwykle żmudnym i czasochłonnym 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 uruchamianie każdej metod z danej klasy zawartej w aplikacji z jednego poziomu interfejsu graficznego. Po odpowiednim skonfigurowaniu narzędzia uruchomienie go odbywa się poprzez dodanie na końcu adresu url aplikacji wpisu */swagger,* gdzie po przejściu na niego użytkownik zostaje przekierowany do panelu głównego aplikacji *swagger*. Na rysunku 1.1 załączono zrzut ekranu opisywanego dodatku. Wylistowane są tu wszystkie klasy używane w api systemu. 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

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 *NSubstitute*. Jest ona napisana z myślą o technologi .NET, przez co można w łatwy i szybki sposób skonfigurować ją pod potrzeby danego użytkownika wytwarzającego oprogramowanie. 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*NSubstitute* obywa się to w taki sposób jak pokazano poniżej.

**TODO**

Dla tak skonfigurowanego 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 sprawdzić jak wydajne są wybrane środowiska zarządzające relacyjną bazą danych oraz sprecyzować w jakich aspektach dane podejście jest lepsze od drugiego. Zapytania były odpytywane z dwóch poziomów dostępowych do bazy - technologii ORM oraz języka SQL. Założeniem było aby w obu przypadkach uzyskać taką samą porcję danych. Do ich uruchamiania wykorzystano język C# wraz z dodatkowymi bibliotekami umożliwiającymi ich przeprocedowanie (opis dodatków został opisany w rozdziale *3. 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 następowała weryfikacja 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 danych.

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 zapisie otrzymanych rezultatów pisano odręcznie odpowiednik wygenerowanego zapytania, a następnie powtarzano czynności sprawdzające czasy ich przeprocedowania. Ostatnim krokiem dla tego typu badań, było zweryfikowanie, czy aby na pewno Technologia ORM tworzy wydajne zapytania bazodanowe.

Dla każdego z testów wykonano od pięciu do dziesięciu zapytań, a następnie obliczano na ich podstawie średnią arytmetyczną, co miało na celu wyeliminowanie różnego rodzaju przekłamań czasowych 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.

## Metodyka badań/Plan eksperymentów/testów

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);

Warto również wspomnieć o właściwym przeanalizowaniu składni oraz zaznajomieniu się z wybraną technologią przez przystąpieniem do badań aby nie popełnić błędów w trakcie ich wykonywania.   
Na przykład może istnieć różnica i dostępie zapytań do bazy danych.   
W zapytaniach *SQL-*owych konieczne 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 konieczności otwierania/zamykania połączenia z bazą, za tę czynność odpowiedzialny jest jego silnik, więc w tym przypadku powinno zostać zawarte tylko to co użytkownika interesuje z bazy. Jak widać   
w opisywanym przypadku istnie możliwość zbyt wczesnego rozpoczęcia pomiarów zapytania, czym mogłoby skutkować przekłamanie wyliczonego czasu potrzebnego na połączenie lub rozłączenie się   
z bazą.

## Zbiory danych

Przeprowadzanie badań w niniejszej pracy byłoby niemożliwe bez przykładowych danych testowych. Aby je pozyskać napisano w tym celu metodę generującą takie dane. Została ona zaimplementowana   
w kontrolerze aplikacji testowej. 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

W celu wykazania, które z dwóch badanych środowisk jest wydajniejsze, należało przeprowadzić szereg zapytań bazodanowych. Zapytania te zostały dogłębniej przeanalizowane w niniejszym rozdziale.

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 rozpoczęcie pomiarów na samym początku wykonywanego kodu, co niestety było błędnym podejściem. Po weryfikacji obydwóch podejść ustawiono czasy pomiarów przed samym zapytaniem oraz na jego zakończeniu   
w miejscu gdzie otrzymywany jest 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 i zajmuje mało linii. 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 dziesięć razy, następnie wyliczono z uzyskanych wyników średnią wartość arytmetyczną. Dla tak prostych poleceń nie można spodziewać się wielkich interwałów czasowych w odpowiedziach z bazy danych. Po ich uruchomieniu 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, niemniej jednak w kolejnych badaniach będą przeprowadzane testy 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 przypadku jak i w kolejnych nie ma żadnych większych zmian, które by odstawały 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 badaniu, 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

Kolejnym testem było zweryfikowanie zajętości czasowej dla pobrania wszystkich dostępnych rekordów dla tabeli *Advert*. Jest to jedno z najłatwiejszych z możliwych zapytań bazodanowy, ze względu na brak warunków dotyczących wartości pól czy też ich zależności do innych tabel. Nie wybierane są tu również pola, tylko pobierane są wszystkie dostępnie w encji. Celem tego zapytania było sprawdzenia   
w jaki sposób zachowa się dana technologia przy pobieraniu jednorazowo dziesięciu tysięcy rekordów. Dla tego testu zostało przeprowadzony dziesięć powtórzeń. Składnia dla tego zapytania   
w języku *SQL* wygląda następująco:

*cmd.CommandText = "Select \* from [dbo].[Adverts]";*

Jak widać zajętość tekstowa dla powyższego przykładu nie jest wielka. Nie inaczej jest również w przypadku zapytania w bibliotece *ORM*.

*var newQuery = \_db.adverts.ToList();*

Pomimo pobierania znacznej ilości dany z bazy, zajętości czasowe po wyliczeniu średnich arytmetycznych nie osiągnęły znacznych wartości. Dla pierwszego zapytania otrzymano wynik po około 34 ms,   
a dla drugiego po 42 ms. Jak widać kolejny raz technologia *ORM* jest mniej wydajna niż zapytanie *SQL*-owe. Możliwe jest, że wbudowany translator Entity Framework tworzy mniej optymalny kod będący trudniejszy w interpretacji przez serwer bazodanowy.

Pobranie danych z dwóch tabel

### Celem tego testu było zweryfikowanie sposobu zachowań dla zapytań pobierających dane z więcej niż jednej tabeli, oraz pobraniu z nich kilku atrybutów. Dodatkowo znajduje się tu również warunek zawężający zakres pobieranych danych o datę dodania i zakończenia wystawianego przedmiotu w systemie, dokładnie w taki sam sposób jak miało to miejsce w pierwszym badaniu o tytule *Zapytanie do pojedynczej tabeli.* Porównując obydwa zapytania można przypuszczać, że są one bardzo podobne jednakże przeprowadzane badania w tym teście sprawdzają zachowanie łączenia tabel przy wykorzystaniu relacji, natomiast w poprzednim zapytaniu odwoływano się tylko do pojedynczej encji.

Pobieranie danych odbywa się w tym przypadku z dwóch encji – *Advert* oraz *Subject,* które połączone są ze sobą kluczem głównym *AdvertId* umożliwiającym pobranie odpowiadającym sobie rekordom   
z obu tabel w zależności od relacji *jeden do wielu* czy *wiele do wielu*. Jest to istotny atrybut ze względu na skonstruowane zapytanie *SQL*-owe zawierające w sobie słowo kluczowe *InnerJoin* dla którego podawany jest atrybut powiązań tabelarycznych. W przypadku zapytania ORM, atrybut *AdvertId* nie jest uwzględniany w celu połączenia ze sobą tabel, ze względu na zaszytą relację odwołującą się do siebie poprzez metody wirtualne. Po skonstruowaniu takiego połączenia w zapytaniu, oraz dodania warunku dotyczącego dat opisujących ważność oferty, wybierane są z niego pola *AdvertDescription, AdvertName, Price.* Ich pobranie w obu technologiach odbywa się dzięki słowu kluczowemu *selekt.* W przypadku biblioteki *ORM,* jeśli by zabrakło takiej selekcji   
w wyniku otrzymano by wszystkie dostępne atrybuty dla wybranych rekordów. Jednakże takie zapytanie znacznie by odbiegało od wersji *SQL*-owej i nie pełniłoby właściwej funkcji badawczej uzyskując dłuższy bądź krótszy czas na odpowiedź z serwera.

Omawiane badanie, podobnie jak dla poprzednie zostało wykonane dziesięciokrotnie. Po obliczeniu czasów, zajętość zapytania *SQL* wyniosła 22,7 ms, natomiast odpowiednik *ORM* – 33.5. Ich składnia została odpowiednio zaprezentowana poniżej.

*cmd.CommandText = "select AdvertName, AdvertDescription, Price from Subjects INNER JOIN Adverts on Adverts.AdvertId = Subjects.AdvertId where (Adverts.AdditionDate > '2018-05-31 12:09:50' and Adverts.AdditionDate < '2018-05-31 12:11:30')";*

*var newQuery = \_db.subjects.Where(x=>x.Advert.AdditionDate > new DateTime(2018, 5, 31, 12, 09, 50) && x.Advert.AdditionDate < new DateTime(2018, 5, 31, 12, 11, 30)).Select(y=>new { y.AdvertDescription, y.AdvertName, y.Price}).ToList();*

Pomimo znacznego podobieństwa tego zapytania z pierwszym testem w tym rozdziale, pojawiły się minimalne różnice w czasach. Dla pierwszego zapytania na poziomie 2,5 ms, a dla drugiego 8,5ms. Wynikać to może tylko i wyłącznie poprzez pobieranie danych z więcej niż jednej tabeli, natomiast na zmianę czasów można wykluczyć różnicę w pobieraniu ilości rekordów ze względu na przeprowadzanie badań na takiej samej próbie danych.

Niemniej jednak, kolejny test pokazuje przewagę standardowego podejścia nad technologią obiektową, w której nie było konieczności bezpośredniego wskazywania relacji z inną tabelą.

Pobieranie danych z dwóch tabel – badanie SSMS (SQL Management Studio)

Jak wspomniano już w rozdziale *4. Część badawcza*, testy weryfikujące wydajność technologii *ORM* oraz zapytań *SQL*, miały zostać przeprowadzone w środowisku programistycznym *Visual Studio*, tak jak dotychczas prezentowano. Dodatkowo miały zostać przeprowadzone badania dla składni *SQL*-owej wygenerowanej przez silnik *ORM* biblioteki *Entity Framework.* W tym badaniu zostanie zaprezentowany pierwszy taki test.

Po przeprowadzeniu testów dla wcześniejszego badania zwracającego dane z dwóch tabel, pobrano wygenerowany kod przez kompilator biblioteki *Entity Framework.* Na jego podstawie wykonano porównania wydajnościowe w programie *SQL Management Studio* tego zapytania   
z wcześniej napisanym odręcznie w języku *SQL.* Niestety otrzymany kod źródłowy w technologii obiektowej okazał się być znacznie bardziej skomplikowany oraz cięższy w interpretacji przez programistę. Dodatkowo zajętość zapytania wygenerowanego wyniosła dziesięć linii, a autorskiego zaledwie trzy linie.

Dla sekcji wydajności czasowej próżno jest również szukać pozytywnych rezultatów. Po przeprowadzeniu dwukrotnych badań   
z dziesięciokrotnymi powtórzeniami dla każdego z zapytań wynika, że podejście odręcznie napisane jest średnio o 2ms. wydajniejsze od wygenerowanego. Na tabeli *Tabela 1* zamieszczono do wglądu czasy dla obydwóch podejść. Jedynym z możliwych czynników wpływających kolejny raz niekorzystnie na złe wyniki czasowe z biblioteki obiektowej jest niepotrzebne dodawanie przez generator przypisu atrybutom nazw reprezentacyjnych odbywających się przy pomocy słowa kluczowego *as*. Dodatkowo niepotrzebnym jest konwersja czasowa daty *dateTime2* na datę reprezentowaną po stronie serwera.

Ze względów optymalizacyjnych całość kodu źródłowego została zamieszczona w załączniku numer *[1] Testy wydajnościow.pdf*.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Powtórzenie | | | | | | | | | |  |
| Typ zapytania | I | II | III | IV | V | VI | VII | VIIII | IX | X | Średnia |
| SQL | 15 | 13 | 17 | 10 | 8 | 11 | 8 | 8 | 9 | 11 | 11 ms |
| 9 | 10 | 10 | 11 | 9 | 9 | 9 | 9 | 7 | 10 | 9,3 ms |
| ORM | 11 | 11 | 15 | 12 | 13 | 13 | 12 | 10 | 15 | 17 | 12,9 ms |
| 14 | 12 | 12 | 12 | 12 | 13 | 12 | 10 | 12 | 11 | 12 ms |

Tabela

Wielokrotna aktualizacja pola

Kolejne badanie jakie zostało przeprowadzone dotyczyło tym razem aktualizacji pola tabeli *Subject*. Dotychczasowe badania sprawdzane były pod kątem pobierania danych oraz ich dodawania. Tym razem tę czynność wykonywano poniekąd za jednym razem. W celu zaktualizowania danego pola w rekordzie konieczne jest wcześniejsze jego wybranie, poczym dla wyszukanych danych zostaje przypisana nowa wartość atrybutu. W tym przypadku jest to wykonywane dla wielu rekordów, więc konstrukcja zapytań dla obu przypadków będzie się w lekkim stopniu różnić. Zapytania *SQL* dają możliwość pozyskania zbioru rekordów spełniających dany warunek – w badanym przypadku jest spełnienie aby moc silnika łodzi była większa od 500. Technologia *ORM* nie dostarcza takiej funkcjonalności przez co konieczne jest wykorzystanie pętli *foreach* dla pobranego zbioru danych, a następnie zapisie danych. Różnice tę można zobaczyć na poniżej zamieszczonym kodzie źródłowym.

*cmd.CommandText = "Update Subjects Set Price = 51234 from Subjects INNER JOIN SailBoat on Subjects.SubjectId = SailBoat.SubjectId where SailBoat.EnginePower > 500";*

*List<Boat>boat=\_db.boats.Where(x=>x.SailBoat.EnginePower>500).ToList();foreach(Boat singleBoat in boat){*

*singleBoat.Price = 10101;*

*}*

*\_db.SaveChanges();*

Wykorzystanie pętli *foreach* w przypadku technologii obiektowej poskutkowało gorszymi wynikami wyliczającymi interwały czasowe po przeprowadzonych badaniach. Dla tego podejścia wyniki można było uzyskać po średnio 27,6 ms. co w rzeczywistości jest niewielką wartością, jednakże w zestawiające go z rezultatem 25 ms. dla zapytania *SQL*w dalszym ciągu jest gorszym podejściem.

Aktualizacja pojedynczego rekordu

Analogicznie dla testu aktualizującego jednocześnie wiele danych,   
w tym przypadku będzie rozpatrywany pojedynczy rekord. Tabela z której będą pobierane dane to encja *Boat.*  Dla tego zapytania zostały sprecyzowane trzy warunki dotyczące spełnienia odpowiedniego ciągu tekstowego pól *Name, Surname* i *Email*. Jeśli chodzi natomiast   
o aktualizacje pola, to również w tym jak i poprzednim przypadku będzie to atrybut *EnginePower.* Składnia również nie różni się w znacznym stopniu od poprzedniego zapytania ze względu na konieczność połączenia się   
w języku *SQL* z dwiema tabelami poprzez *InnerJoin*. Jeśli chodzi   
o technologię obiektową, zapytanie stało się krótsze ze względu na brak użycia pętli przechodzącej po wszystkich znalezionych danych ustępując miejsca pobraniu pojedynczego rekordu tak jak pokazano poniżej.

Boat boat = \_db.boats.Where(x=>x.Advert.Name == "Abby" && x.Advert.SureName == "Halama" && x.Advert.Email == "testUpdate@gmail.com").FirstOrDefault(); boat.SailBoat.EnginePower = 11;

Uzyskany czas odpowiedzi dla tego zapytania zwracającego pojedynczy rekord okazał się znacznie dłuższy od poprzedniego badania gdzie aktualizowanych było ponad tysiąc rekordów. Różnica wyniosła około 16 ms, z poprzednim testem. Na dłuższy czas odpowiedzi może mieć wpływ większa ilość warunków w sekcji *where*. Dla zapytania w podejściu *SQL* interwał po wykonaniu kolejnych żądań nie zmienił się i pozostał na poziomie dwudziestu pięciu milisekund. Jedyny wniosek jaki można wyciągnąć po wykonaniu powyższego badania jest taki, że wzrost ilości warunków ma wpływ na szybkość ich interpretacji po stronie translatora biblioteki *ORM.* Natomiast zapytanie tekstowe *SQL,* nie ma tak dużej zwłoki czasowej ze względu na wykonanie go po stronie serwera bazodanowego operującego bezpośrednio na danych pomijając przy okazji narzędzia konwertujące w odróżnieniu od technologii Entity Framework.

Aktualizacja danych - badanie SSMS

Aby zweryfikować niepoprawne zachowanie zapytania stworzonego   
w technologii ORM, postanowiono zbadać jak zachowuję się wygenerowany skrypt po stronie serwera bazy danych w zestawieniu z ręcznie utworzonym odpowiednikiem.

Po wygenerowaniu skryptu i pobraniu go z konsoli biblioteki *Entity Framework* okazało się, że jest ono dłuższe aż pięć razy od zapytania testowanego poprzednio. W trakcie analizowania kodu, translator do zapytania ponownie zastosował przypisanie typu *as* atrybutom takie same nazwy jakie posiadają pola. Jest to jeden z większych czynników wpływających niekorzystnie na długość tego zapytania. Dodatkowo w tym zapytaniu wykorzystywane są zmienne dla których przypisuje się nowe zapytania. Dopiero po tym wszystkim aktualizowane jest dane pole. Tabela *Tabela 2* prezentuje uzyskane wyniki z wyliczeń dla podejścia *SQL* oraz *ORM*.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Powtórzenie | | | | | | | | | |  |
| Typ zapytania | I | II | III | IV | V | VI | VII | VIIII | IX | X | Średnia |
| SQL | 11 | 10 | 8 | 11 | 8 | 9 | 6 | 9 | 9 | 9 | 9 ms |
| 8 | 12 | 7 | 8 | 9 | 8 | 9 | 8 | 10 | 7 | 8,6 ms |
| ORM | 15 | 14 | 12 | 11 | 13 | 10 | 12 | 13 | 10 | 15 | 12,5 ms |
| 11 | 12 | 17 | 12 | 12 | 13 | 11 | 16 | 12 | 12 | 12,8 ms |

Tabela

Jak widać po tak długim zapytaniu oraz wykorzystaniu zmiennych nie można było spodziewać się pozytywnych wyników dla zapytania pochodzącego z obiektowego podejścia. Różnica na poziomie około trzech milisekund dla pojedynczego rekordu jest stanowczo za duża. Można więc przypuszczać, że dla większego wolumeny danych sięgających tysięcy powtórzeń, interwał może się znacznie zwiększyć.

Dodawanie danych z zapisem końcowym

W tym oraz następnym badaniu zostaną przeprowadzone badania weryfikujące wpływ zapisu do bazy danych pojedynczych rekordów oraz zbiorów danych. Różnica tych zapytań polega na utworzeniu dla pierwszego przypadku zbioru zapytań w języku *SQL* oraz stworzenia listy zawierającej dane tabeli *Boats* dla technologii *ORM*, a następnie zapisie do bazy danych. Dla drugiego przypadku zapis będzie się odbywać przy każdorazowym tworzeniu nowego obiektu tabeli*.* Istnieje przypuszczenie, że przy zapisie większej ilość danych nastąpi zacieśnienie kolejki dodającej rekordy do bazy danych przez co zapytanie może zająć więcej czasu niż pojedyncza insercja. Taki test ma za zadanie potwierdzić lub zaprzeczyć tą tezę oraz pokazać która z technologii lepiej sobie radzi z takim problemem.

W tym badaniu będziemy rozpatrywać podejście w którym zapis będzie wykonywany tylko raz dla zbioru danych. Na czas testów zostało utworzonych 20 rekordów. W podejściu obiektowym te dane są reprezentowane dzięki kolekcji *List*, do której wpisywano kolejne partie danych. Deklaracja tej kolekcji wyglądała w taki sposób *- var boats = new List<Boat> { <obiekty typu Boat> }.* Następnie obiekt boats został dodany do tabeli *boats - \_db.boats.AddRange(boats);,* a na samym końcu wprowadzone zmiany zapisano poleceniem - *\_db.SaveChanges();*. Zapytania w języku *SQL* również zapisywały wiele rekordów przy pojedynczym zapisie, jednakże konstrukcja ta nie wykorzystywała w tym celu pętli bądź tworzenia obiektów. Zapytania w tym języku przyjmują postać tekstową którą wykonuje się bezpośrednio po stronie serwera bazy danych, w związku z czym koniecznym jest wykorzystanie konkatenacji - czyli łączenia ze sobą ciągów tekstowych. W wykorzystywanym języku programowania w trakcie badań odbywa się to przy pomocy operatora *+*. Przykład takiej konkatenacji zaprezentowano poniżej.

cmd.CommandText = "<insert into Adverts…>" + "<…>" + … ;

cmd.CommandType = CommandType.Text;

reader = cmd.ExecuteReader();

Po części deklarującej zapytanie wykonywana jest standardowa procedura wysłania i uruchomienia go po stronie bazy danych.

Porównując obydwa podejścia, widać znaczną łatwość w zapisie nowych rekordów dla języków obiektowych. Jednakże są w tym miejscu tworzone nowe obiekty, które z pewnością zajmują dużą ilość czasu na przetworzenie ich przez kompilator. Przypuszczalnym jest, że zajętość czasowa tego podejścia będzie większa niż dla języka *SQL.* Po uruchomieniu utworzonych zapytań wyniki pokrywają się z wcześniejszymi założeniami. Niestety ich różnice są drastycznie duże. Zajętość czasowa dla zapytania tekstowego wyniosła zaledwie 32ms. Natomiast dla podejścia wykorzystującego *Entity Framework* uzyskano wynik na poziomie 1650 ms. czyniąc go daleko w tyle za tradycyjnym podejściem. Jak widać, po uzyskaniu takich rezultatów warto jest się zastanowić czy dodawanie zbioru rekordów przy jednym podejściu jest dobrym rozwiązaniem.

Dodawanie danych z ciągłym zapisem

Podczas tego testu zostanie zweryfikowane dodawanie rekordów do bazy danych z każdorazowym zapisem. Jest to odwrotna operacja dla wcześniejszego badania wykonującego zapis dla gotowego zbioru danych. Na strukturę zapytania składa się samo zapytanie odpowiednie dla danej technologii oraz pętla *for* powtarzająca się dwadzieścia razy, gdzie pod koniec każdej operacji wykonywany jest zapis do bazy danych. W tym zapytaniu operowano na takim samych obiekcie jak utworzono w poprzednim teście jednakże jest ono w wykorzystywane tylko jeden raz w każdej z iteracji.

Problematyczne w tym badaniu było zarządzanie kluczami głównymi dla tabel *Advert, Subject,* oraz *Boats*. Zapytania dla wersji *SQL*-owej łączyły ze sobą wymienione tabele przy pomocy *InnerJoin*-a, gdzie istotnym była kontrola zgodności odpowiadającym sobie kluczom. Zgodność tą zapewniono dzięki wykorzystaniu dwóm zmiennym całkowitym, które przed rozpoczęciem operacji były inicjowane kolejnymi wartościami identyfikatorów tabel. Dla przykładu w trakcie przeprowadzania tego badania w bazie widniał klucz główny o wartości 14216 dla tabeli *adverts* w związku z czym w aplikacji konieczne było przypisanie wartości 14217. Po każdorazowym dodaniu rekordu do bazy wartości te były zwiększane o jeden. Dla technologii *Entity Framework* taki zabieg nie był konieczny ze względu na jego silnik kontrolujący zależności relacyjne zwiększając wartości kluczy głównych.

Po skonstruowaniu zapytań przystąpiono do wykonania powyższych zapytań. Niestety uzyskane rezultaty zostały powtórzone z poprzednich testów – technologia *ORM* ponownie odstaje od zapytania *SQL*-owego. Wynik wyniósł odpowiednio 1750 ms. oraz 266 ms.. Jak widać podejście z każdorazowym okazała się gorszym w obu przypadkach od wersji z jednokrotnym zapisem dla całej kolekcji. Z różnic wynikających ze składni można wnioskować że na tak długi czas odpowiedzi wpływa tylko pętla for. Niestety jest to czynnik wynikający z konieczności kompilacji kodu którego nie da się obejść w tym przypadku.

Przyrostowe dodawanie rekordów

## 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