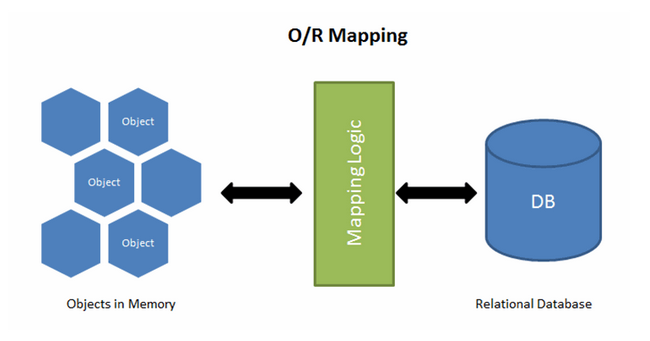
**ORM – Object Relational Mapping**Narzędzie mapowania obiektowo relacyjnego



**Entity Framework**

Generuje modele zgodne z tabelami baz danych  
Możemy mapować obiekt np. Student na odpowiadającą mu tabelę w bazie danych.  
Nazwy mogą być takie same, ale mogą być również różne

Możemy pisać logikę w C#, która będzie pod spodem zamieniana na kwerendy SQL.  
**Inne frameworki** ADO .NET, NHibernate, dopper

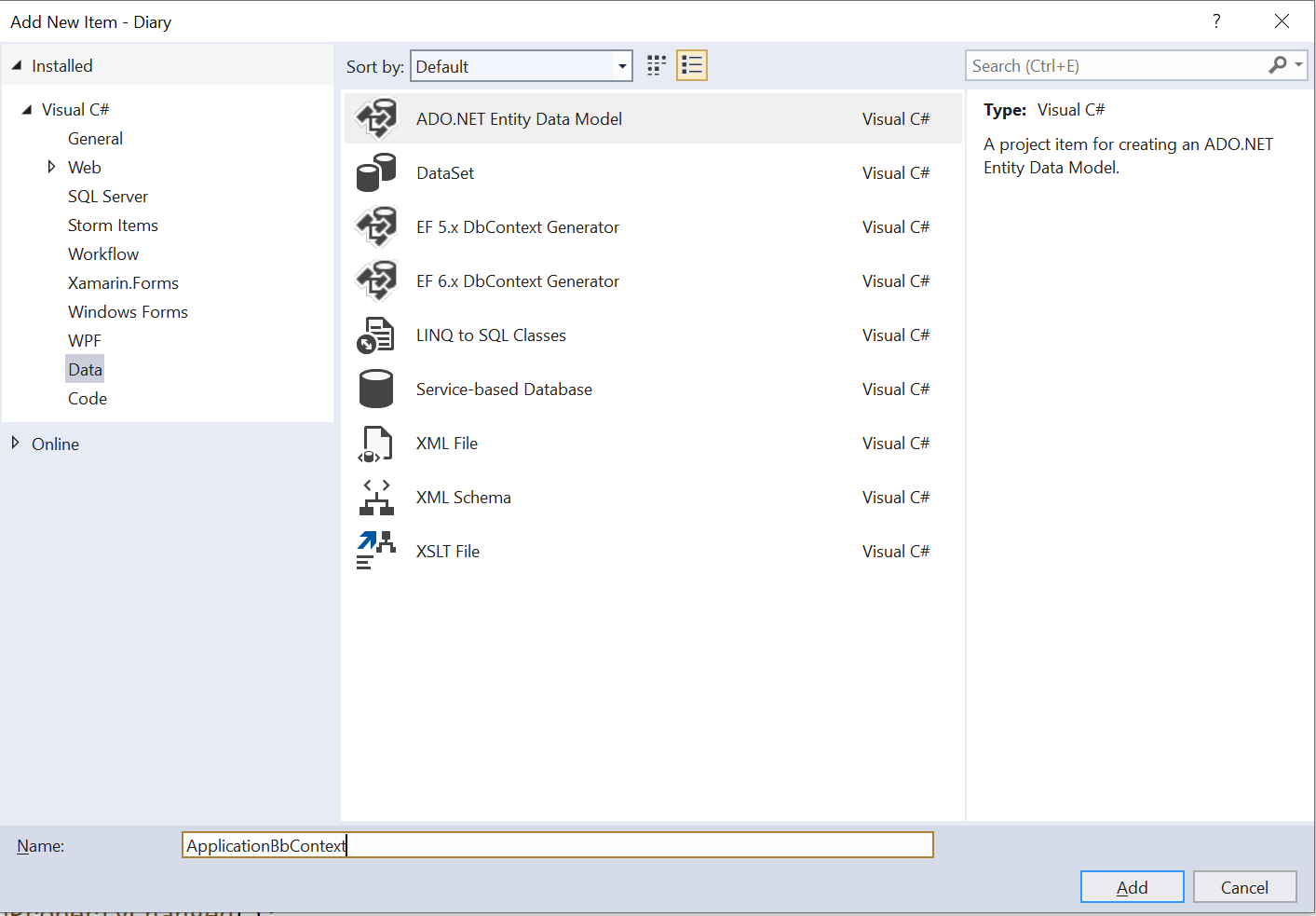
 **NHibernate** jest rozwiązaniem ORM dla platformy .NET. Dostarcza środowisko do mapowania obiektowo-relacyjnego dla tradycyjnego, relacyjnego modelu bazy danych. Jego podstawową funkcją jest mapowanie z klas platformy .NET do tabeli baz danych oraz od typów danych CLR do typów danych SQL.

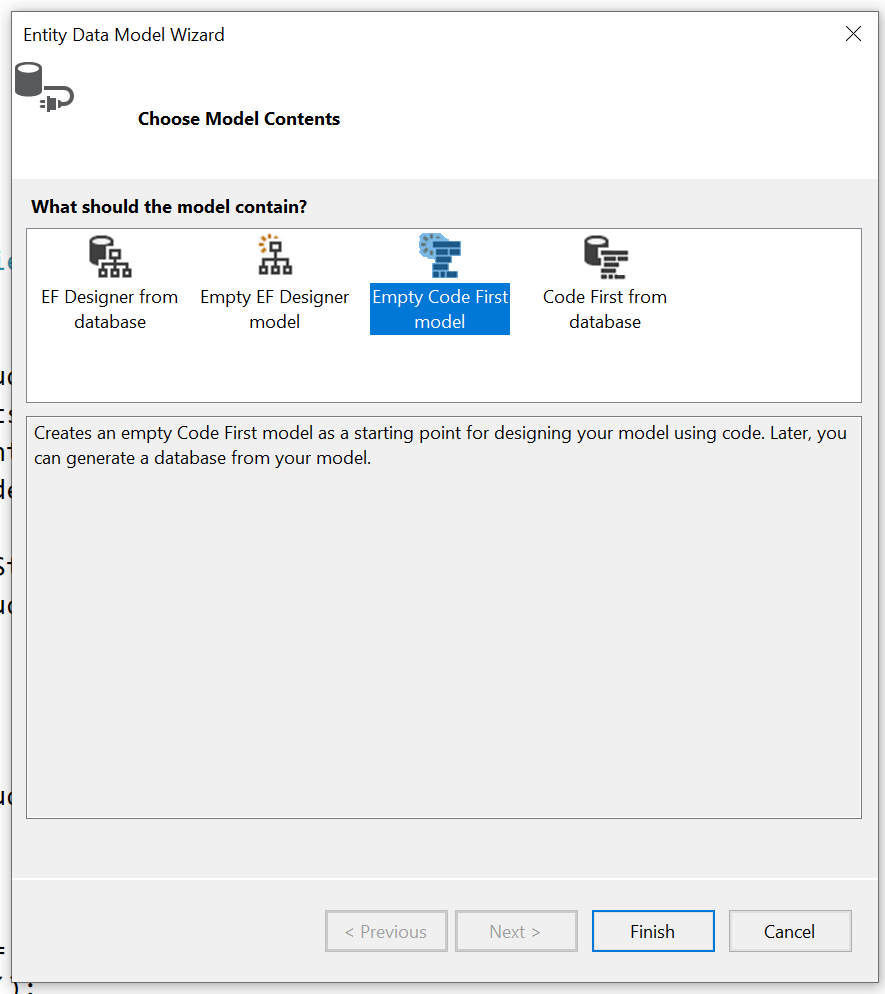


Dapper jest przykładem ORMa, którego funkcjonalność jest ograniczona do minimum. Robi on tylko i aż mapowanie pomiędzy zapytaniami i obiektami w kodzie. Jednak o to jak wyglądać będzie zapytanie musimy zatroszczyć się sami. Dapper wykonuje jedynie kod SQL, który sami napiszemy. Jego odpowiedzialnością jest jedynie zamiana zwróconych wartości na obiekty w kodzie C# albo wstawienie wartości z kodu C# do zapytania SQL. Dzięki temu mamy pełną kontrolę nad tym jak wyglądać będzie zapytanie. Efektem minimalnej funkcjonalności Dappera jest jego wydajność. Dapper do czasu odświeżenia Entity Frameworka wraz z nadejściem wersji Core był nie do doścignięcia przez standardowe ORMy jeżeli chodzi o czas wykonania zapytań. Nadal jest w czołówce.

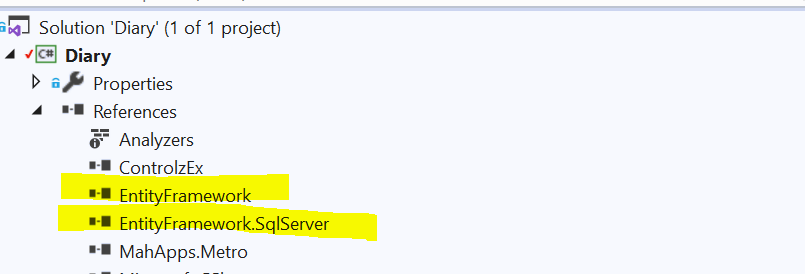
Code First – gdy nie mamy jeszcze bazy  
Code first from database – gdy mamy już bazę danych  
Database First – gdy mamy już bazę  
Model First – najgorsze podejście

**Instalacja**





Referencje powstałe po instalacji



Główna klasa

|  |
| --- |
| namespace Diary  {  public class ApplicationBbContext : DbContext  {  public ApplicationBbContext()  : base("name=ApplicationBbContext")  {  }  }  } |

W App.config wpisujemy connectionString

|  |
| --- |
| <connectionStrings>  <add name="ApplicationBbContext" connectionString="data source=(LocalDb)\MSSQLLocalDB;initial catalog=Diary.ApplicationBbContext;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework" providerName="System.Data.SqlClient" />  </connectionStrings> |

|  |
| --- |
| <connectionStrings>  <add  name="ApplicationBbContext"  connectionString="Server=127.0.0.1;Database=Diary;Uid=user1;Pwd=alamakota;"  providerName="System.Data.SqlClient" />  </connectionStrings> |

|  |  |
| --- | --- |
| W katalogu Models | tworzymy podkatalogi: |
| **Domains** | wszystkie obiekty domenowe na podstawie których będziemy tworzyć tabele w bazie danych |
| **Wrappers** | modele, które będziemy używać w aplikacji , właściwości i metody |
| **Configurations** | konfiguracje obiektów domenowych |
| **Converters** | konwertery obiekty domenowe na wrappery i odwrortnie |

|  |
| --- |
| namespace Diary.Models.Domains  {  public class Student  {  public Student()  {  Ratings = new Collection<Rating>();  }  public int Id { get; set; }  public string FirstName { get; set; }  public string LastName { get; set; }  public string Comments { get; set; }  public string Activities { get; set; }  public int GroupId { get; set; }  public Group Group { get; set; }  public ICollection<Rating> Ratings { get; set; }  }  } |

W obiekcie student mamy Id Grupy, ale dodatkowo mamy też obiekt Grupa aby łatwiej zadawać zapytania, mamy też kolekcję ocen

|  |
| --- |
| namespace Diary.Models.Domains  {  public class Rating  {  public int Id { get; set; }  public int Rate { get; set; }  public int StudentId { get; set; }  public int SubjectId { get; set; }  public Student Student { get; set; }  }  } |

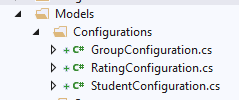
Podobnie w obiekcje z oceną mamy również oprócz Id Studenta obiekt Student

W klasie ApplicationDbContext

Wskażemy klasy domenowe

|  |
| --- |
| namespace Diary  {  public class ApplicationBbContext : DbContext  {  public ApplicationBbContext()  : base("name=ApplicationBbContext")  {  }  // wskazujemy klasy domenowe  // Entity Framework wie, że ma stworzyć 3 tabele jak poniżej  public DbSet<Student> Students { get; set; }  public DbSet<Group> Groups { get; set; }  public DbSet<Rating> Ratings { get; set; }  }  } |

W katalogu **Configurations** tworzymy dla każdej klasy domenowej osobny plik konfiguracyjny



|  |
| --- |
| namespace Diary.Models.Configurations  {  public class StudentConfiguration : EntityTypeConfiguration<Student>  {  public StudentConfiguration()  {  ToTable("dbo.Students");  HasKey(x => x.Id);  }  }  } |

|  |
| --- |
| namespace Diary.Models.Configurations  {  public class GroupConfiguration : EntityTypeConfiguration<Group>  {  public GroupConfiguration()  {  ToTable("dbo.Groups");  Property(x => x.Id)  .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);    Property(x => x.Name)  .HasMaxLength(20)  .IsRequired();  }  }  } |

W klasie ApplicationDbContext musimy dać znać aplikacji, że istnieje taki plik konfiguracyjny.  
Musimy w tym celu przysłonić metodę **OnModelCreating**

|  |
| --- |
| // wskazujemy klasy domenowe  // Entity Framework wie, że ma stworzyć 3 tabele jak poniżej  public DbSet<Student> Students { get; set; }  public DbSet<Group> Groups { get; set; }  public DbSet<Rating> Ratings { get; set; }  protected override void OnModelCreating(DbModelBuilder modelBuilder)  {  base.OnModelCreating(modelBuilder);  modelBuilder.Configurations.Add(new StudentConfiguration());  modelBuilder.Configurations.Add(new GroupConfiguration());  modelBuilder.Configurations.Add(new RatingConfiguration());  } |

W konstruktorze

|  |
| --- |
| public MainWindowViewModel()  {  // zostanie utworzone pierwsze zapytanie i utworzone bazy  using ( var context = new ApplicationBbContext())  {  var students = context.Students.ToList();  } |

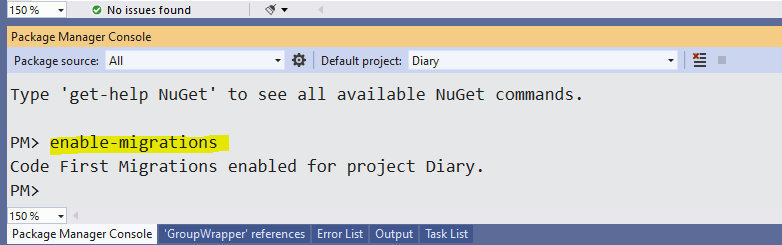
Migracje, aktualizowanie bazy danych  
Podczas rozwoju aplikacji często istnieje konieczność aktualizacji struktur bazy danych

W EntityFramework są różne strategie

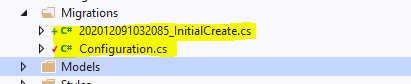
CreateDatabaseIfNotExists  
DropCreatedatabaseIfModelChanges  
DropCreateDataBaseIfAlways

Migracje – zmiany w strukturze bez utraty danych  
dwa rodzaje – manualne i automatyczne

Odblokowujemy okno: Package Manage Console w View -> Other Windows



Gdy coś nie wyjdzie : add-migration init



|  |
| --- |
| internal sealed class Configuration : DbMigrationsConfiguration<Diary.ApplicationBbContext>  {  public Configuration()  {  AutomaticMigrationsEnabled = false;  ContextKey = "Diary.ApplicationBbContext";  }  protected override void Seed(Diary.ApplicationBbContext context)  {  // This method will be called after migrating to the latest version.  // You can use the DbSet<T>.AddOrUpdate() helper extension method  // to avoid creating duplicate seed data.  }  } |

W drugim pliku informacje na temat pierwszej migracji, która tworzy tabela na podstawie  
obecnego kontekstu

Up() - wywołana przy aktualizacji  
Down() – gdy chcemy przywrócić do poprzedniej wersji, wycofuje to co zrobiło Up()

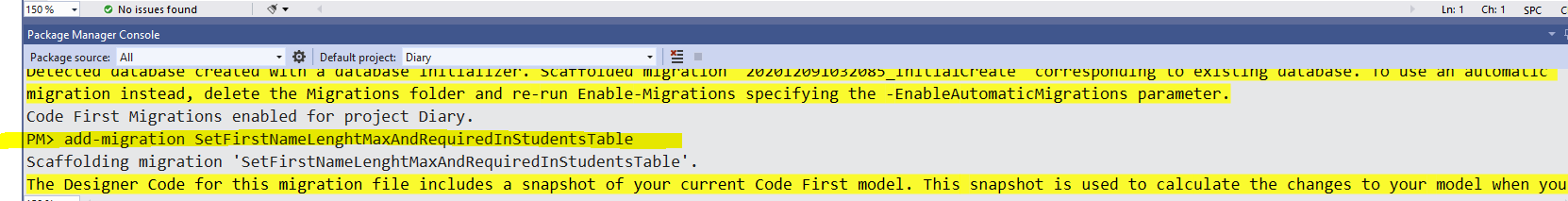
Chcemy zmienić np. FirstName w klasie student ustawić na max100 znaków.  
W tym celu przeprowadzamy modyfikacje modelu domenowego – w pliku StudentConfigurations

|  |
| --- |
| public class StudentConfiguration : EntityTypeConfiguration<Student>  {  public StudentConfiguration()  {  ToTable("dbo.Students");  HasKey(x => x.Id);  Property(x => x.FirstName)  .HasMaxLength(100)  .IsRequired();  }  } |

Gdybyśmy teraz wywołali aplikację, jeżeli w tym momencie uruchomimy aplikację to na pierwszym odwołaniu do bazy powstał by wyjątek, dlatego uruchamiamy migrację jak poniżej.

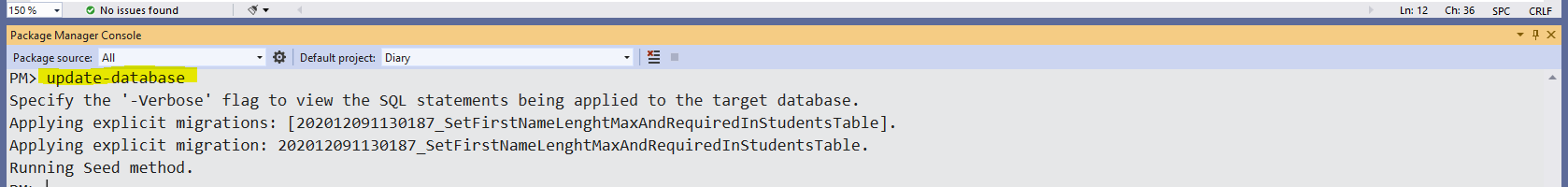
PM> add-migration SetFirstNameLenghtMaxAndRequiredInStudentsTable

Dodajemy unikalną nazwę, najlepiej oddającą co dana migracja robi



W konsoli spisujemy

PM> update-database



Aby wycofać migrację:

PM> update-database – targetmigration: nazwa\_migracji