Politechnika Wrocławska

Wydział Elektroniki

—————————————————

Kierunek: Informatyka

Specjalność: Inżynieria Internetowa

**Praca Dyplomowa**

**Inżynierska**

Prosty framework przeznaczony do tworzenia aplikacji internetowych w języku C#

Simple web application framework for C# programmers

Autor:

Jan Romaniak

Prowadzący Pracę:

dr inż. Henryk Maciejewski

1. Wstęp
   1. Założenia Projektowe

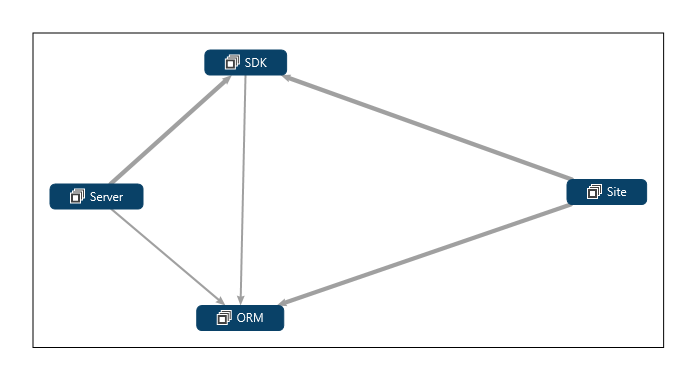
Głównym celem projektu było stworzenie prostego w obsłudze framework'a do tworzenia aplikacji internetowych. Językiem jaki wybrałem był C# pozwala on łatwe tworzenie aplikacji dla systemu Windows. Jego funkcjonalność miała być zbliżona do tej jaką prezentuje framework Django stworzony w języku Python.

1. Architektura Framework'a
   1. Główne moduły aplikacji.

Framework składa się z czterech głównych modułów:

* Serwera aplikacji - Pełnoprawnego serwera http, bądź serwera oferującego interfejs FastCGI
* SDK - Zawiera zestaw niezbędnych funkcji oraz pośredniczy w komunikacji pomiędzy serwerem
* ORM - Moduł obsługujący bazę danych oraz pozwalający na wykorzystanie mapowania obiektowo relacyjnego.
* Aplikacja użytkownika - jedna bądź wiele aplikacji użytkownika
  1. Sposób połączenia pomiędzy modułami

Na podstawia przykładowej aplikacji internetowej, poniższy graf przedstawia zależności, pomiędzy modułami.



Rys. Graf przedstawiający zależności pomiędzy głównymi modułami framework'a

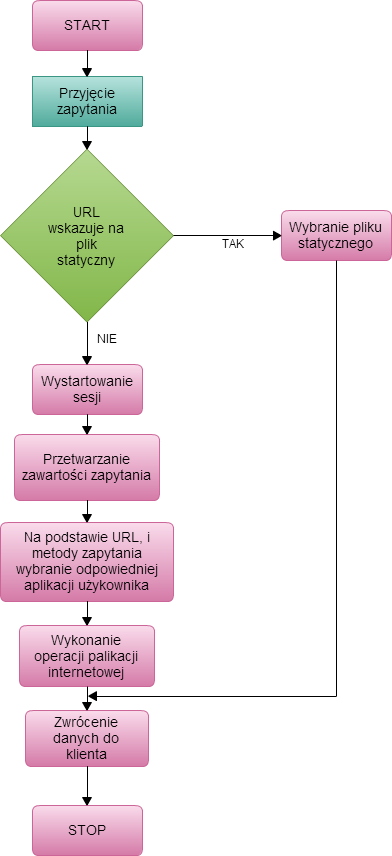
Aplikacja pisana przez użytkownika nie ma dostępu do serwera aplikacji. Warstwą pośredniczącą jest SDK.

* 1. Server Aplikacji

Pozwala on na komunikacje przy pomocy protokołu HTTP bądź FastCGI. Zajmuje się obsługą zapytań otrzymywanych za pomocą któregoś z wyżej wymienionych protokołów. Po otrzymaniu zapytania przetwarza informacje w nim zapisane i przekazuje informacje do wybranej przez użytkownika funkcji obsługującej odpowiedź na zapytanie.

Serwer startowany jest jako aplikacja główna zajmuje się wyszukaniem wszystkich aplikacji użytkownika oraz inicjalizacją ich. Aby inicjalizacja przebiegła pomyślnie wymagana jest przynajmniej jedna aplikacja. Serwer przetwarza także ustawienia zapisane przez użytkownika i w zależności od nich startuje serwer HTTP albo FastCGI. Ustawienia odpowiadają także za zainicjalizowanie modułu ORM. Przy odbiorze zapytania, serwer decyduje czy ma wysłać plik statyczny (np. Obrazek), czy też zająć się przetwarzaniem zapytania i skierowania go do aplikacji użytkownika.

Zapytanie przetwarzane jest w kilku krokach, i w zależności od potrzeby przekazane lub nie do aplikacji użytkownika. Diagram opisujący działanie framework'a przedstawiono poniżej:



Rys. Diagram przedstawiający schemat obsługi zapytania

* 1. SDK

Zawiera zestaw klas oraz funkcji, które pozwalają na komunikacje z serwerem oraz swobodne tworzenie aplikacji internetowych. Wewnątrz SDK jest przechowywana klasa ustawień która statycznie przechowuje ustawienia całego framework'a.

Ustawienia aplikacji zawieraj:

* Listę URL wraz z funkcjami które zostaną wywołane w momencie odebrania zapytania o danym adresie jak i metodzie.
* Ścieżkę do folderu z przygotowanymi *template'ami*
* Ścieżkę do folderu w którym będą przechowywane pliki statyczne, nie przetwarzane przez framework
* Informacja czy pliki statyczne mają być cache'owane w pamięci ram.
* Ustawienia bazy danych oraz klasa obsługująca połączenie
* Informacja o trybie uruchomienia aplikacji
* Lista modułów aplikacji, z których składa się aplikacja
* Słownik, który pozwala na przekazywanie ustawień pomiędzy modułami aplikacji
  1. ORM

Moduł obsługujący dostęp do bazy danych oraz pozwalający na mapowanie relacyjno obiektowe.

Podczas inicjalizacji aplikacji użytkownika przeszukiwane są one aby znaleźć wszystkie klasy modeli aplikacji. Na podstawie klas oraz właściwości które się w nich znajdują budowany jest model relacyjny a następnie wdrażany w wybranym typie bazy danych.

Budowa modułu pozwala na wykorzystanie i zaimplementowanie obsługi własnej bazy danych, poprzez zaimplementowanie interfejsu DatabaseHelper. W aktualnej wersji dostarczone są klasy do obsługi *SQLite* oraz *PostgreSQL*, wybrałem akurat te bazy danych ponieważ *SQLite* nie wymaga instalacji serwera bazy danych. *PostgreSQL* zostało natomiast wybrane dlatego że jest szybką bazą danych oraz dlatego, iż znacząco różni się od *SQLite*. Jego restrykcyjność pozwoliła wyeliminować wiele błędów, które *SQLite* dopuszczał.

Moduł ORM pozwala na swobodne operowanie na informacjach przechowywanych w bazach danych bez konieczności wykorzystania języka zapytań SQL. Przy wykonywaniu zapytań posługujemy sie klasami, które wcześniej oznaczyliśmy jako model. Pobierając jakieś dane z bazy, operujemy na nazwach pól i klas a nie na nazwach kolumn i tabel, a te często się różnią. Nazwy klas są poprzedzane prefiksami.

Podczas tworzenia relacji tworzone są klucze obce, z poziomu użytkownika wykorzystanie ich jest przeźroczyste oznacza, to że użytkownik pyta o nie przez nazwę klasy a następnie przez nazwę pola. Przy pobieraniu tabel z bazy danych gdzie w zapytaniu odwołujemy sie do połączonych relacyjnie obiektów wykonywane są *join'y* w zapytaniach SQL. Przy zwracaniu wyników wykorzystany jest mechanizm *lazy reltaionship*, który cechuję się tym, że dane powiązanych relacyjnie obiektów pobierane są dopiero w momencie odwołania się do nich.

1. Szczegóły implementacyjne
   1. Modularność

Framework udostępnia użytkownikowi możliwość tworzenia aplikacji w kilku modułach. Za ładowanie i inicjalizacje modułów odpowiada serwer. Każda aplikacja musi posiadać przynjmniej jeden moduł *Site*, który może wdrażać inne moduły. Każdy z modułów musi zawierać tylko jedną metodę statyczną o nazwie *Main* jest ona wywołana w momencie inicjalizacji, każdy z modułów może dołączać kolejne pod warunkiem, że zostaną one umieszczone na końcu listy modułów. Każdy z modułów jest tak na prawdę biblioteką *Assembly* platformy *.Net* i posiada rozszerzenie \*.dll. Biblioteki te ładowane są metodą *runtime* do programu, ich nazwy przechowywane są w ustawieniach aplikacji. Poniższy kod przedstawia sposób ładowania modułów do serwera.

Src. Ładowanie modułów do serwera i wywoływanie funkcji inicjalizacyjnej

Assembly assembly = Assembly.LoadFrom(@"Site.dll");

getMainMethod(assembly).Invoke(null, null);

modulesAsseblies.Add(assembly);

for (int i = 0; i < Settings.Modules.Count; i++)

{

var str = Settings.Modules[i];

Assembly moduleAssembly = Assembly.LoadFrom(str + @".dll");

getMainMethod(moduleAssembly).Invoke(null, null);

modulesAsseblies.Add(moduleAssembly);

}

Oraz sposób wyszukiwania statycznej funkcji *Main* w bibliotekach *Assembly*.

Src. Wyszukiwanie statycznej metody Main w bibliotece

public static MethodInfo getMainMethod(Assembly assembly)

{

var types = assembly.GetTypes();

foreach (var t in types)

{

var method = t.GetMethod("Main", BindingFlags.Static | BindingFlags.Public);

if (method != null)

return method;

}

return null;

}

* 1. Serwer Http/FastCGI

Na potrzeby framework'a został napisany serwer HTTP. Jest on wielowątkowy. Jako serwer FastCGI zastosowano bibliotekę *SharpCGI* w obu przypadkach zaimplementowano takie samo przetwarzanie danych.

Zaimplementowana obsługa:

* Pliki statyczne
* Ustawianie nagłówka *Content-Type* na podstawie rozpoznawanego na podstawie rozszerzenia pliku *MimeType*.
* Przetwarzanie zawartości zapytania ze względu na typ przesyłanej treści
  + *application/json*
  + *multipart/form-data*
  + tradycyjne formularze
* Rozpoznawanie metody zapytania
  + GET
  + POST
  + PUT
  + DELETE
* Rozpoznawanie metody obsługującej na podstawie zapisanego w ustawieniach URL (zapisane w wyrażeniach regularnych)
* Startowanie sesji
* Ustawianie ciasteczek
* Obsługa błędów 404 i 503
  1. URL Dispatcher

Na początku sprawdzane jest czy URL prowadzi do pliku statycznego, w wypadku gdy nie jest rozpoznawanie URL jest rozwiązywane na podstawie wyrażeń regularnych, oraz metody z jaką zostało wysłane zapytanie. Najpierw porównywane z regexp'em są zapisane przez aplikacje URL, a w momencie potwierdzenia zgodności sprawdzana jest metoda requestu.

Przykładowy URL w aplikacji użytkownika zawiera trzy informacje:

* Wyrażenie regularne, z którym porównywany jest URL dostarczony w zapytaniu
* Metoda przy której ma być wywołany (GET, POST, PUT, DELETE)
* Funkcja obsługująca zapytanie

Src. Dodanie URL obsługującego zapytanie

Settings.Urls.Add(new Url(@"^/projects/(?<project>[0-9]+)/tasks/(?<task>[0-9]+)/$", Tasks.DeleteTask, RequestMethod.DELETE));

W przedstawionym powyżej przykładzie, na podstawie analizowanego URL i wyrażenia regularnego dobierane są dwa parametry w tym przypadku o nazwie *project* i *task* oby dwa są liczbami nieujemnymi. Po zanalizowaniu przez dispatcher zostają przekazane do funkcji odpowiadającej na zapytanie.

Src. URL dispatcher

private FeintSDK.Url UrlDispatcher(FeintSDK.Request req, FeintSDK.RequestMethod actualMethod)

{

FeintSDK.Url urlApp = null;

for (int i = 0; i < FeintSDK.Settings.Urls.Count; i++)

{

var match = Regex.Match(req.Url.ToString(), FeintSDK.Settings.Urls[i].UrlMatch);

if (match.Success && (FeintSDK.Settings.Urls[i].Method == FeintSDK.RequestMethod.ALL || actualMethod == FeintSDK.Settings.Urls[i].Method))

{

urlApp = FeintSDK.Settings.Urls[i];

setNonPublicSetProperty(req, req.GetType(), "variables", match.Groups);

break;

}

}

return urlApp;

}

* 1. Request Handler i Programowanie aspektowe

Po wybraniu odpowiedniej funkcji, która ma odpowiedzieć na zapytanie pobierany jest parametr View, który tak na prawdę wskazuje na funkcje, która przechwytuje zapytanie. Sama funkcja pozwala na wygenerowanie odpowiedzi. Aby ułatwić pracę tworzenia stron przygotowane zostało rozszerzenie aspektowe. Udało się to uzyskać dzięki atrybutom i refleksji.

Programowanie aspektowe (aspect-oriented programming, AOP) jest to paradygmat programowania polegający na odseparowaniu niezależnych od siebie zagadnień. Przykład który zazwyczaj jest podawany to przelew bankowy.

Src. Programowanie nieaspektowe przykład

public void transferMoney(int accId1, int accId2, int amount)

{

log("transfer started " + amount + " from account " + accId1 + " to " + accId2);

accounts[accId1] -= amount;

accounts[accId2] += amount;

log("transfer ended " + amount + " from account " + accId1 + " to " + accId2);

}

Widać że zapisywanie logów jest logicznie niezwiązane z operacjami na kontach więc w paradygmacie AOP powyższy przykład wyglądał by tak;

Src. Programowanie aspektowe przykład

[Log]

public void transferMoney(int accId1, int accId2, int amount)

{

accounts[accId1] -= amount;

accounts[accId2] += amount;

}

W tym wypadku można atrybut Log mówi, że przed wywołaniem funkcji i po jej wywołaniu nastąpi logowanie informacji o przebiegu tej operacji. Jest to rozwiązanie prostsze i bardziej czytelne.

Zaimplementowane zostało to przy pomocy abstrakcyjnej klasy atrybutu. Zawiera ona prototypy dwóch metod *PreRequest* wywoływana przed i *PostRequest* wywoływana po. Metoda *PreRequest* może dodatkowo spowodować że funkcja główna nie zostanie wywołana. Jeżeli zwróci *null* funkcja główna wykonywana jest w sposób normalny, jeżeli *Response* funkcja główna jest pomijana.

Src. Klasa AOPAttribute

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]

public abstract class AOPAttribute:Attribute

{

public abstract Response PreRequest(Request req);

public abstract void PostRequest(Request req);

}

Poniżej przedstawiony jest fragment kodu odpowiedzialny za wywołanie odpowiednich metod przed wywołaniem funkcji główne a także po jej wywołaniu.

Src. Funkcja RunApplication obsuługa AOP

private static FeintSDK.Response RunApplication(Response response, FeintSDK.Response res, FeintSDK.Request req, FeintSDK.Url urlApp)

{

MethodInfo mi = urlApp.View.GetMethodInfo();

FeintSDK.AOPAttribute[] aops = (FeintSDK.AOPAttribute[])mi.GetCustomAttributes(typeof(FeintSDK.AOPAttribute), true);

res = null;

foreach (var aop in aops)

{

res = aop.PreRequest(req);

}

if (res == null)

{

res = urlApp.View(req);

if (res.MimeType != null)

{

response.SetHeader(ResponseHeader.HttpContentType, res.MimeType + "; charset=utf-8");

}

}

foreach (var aop in aops)

{

aop.PostRequest(req);

}

return res;

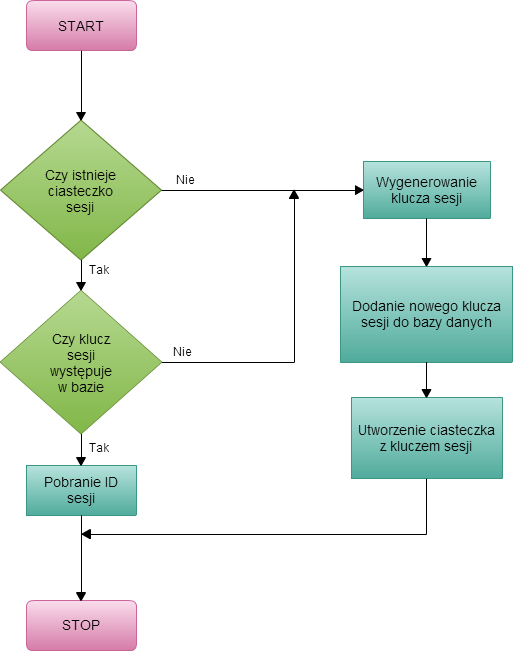
}

W aplikacji internetowej od razu narzuca się możliwość wykorzystania takiego rozwiązania. Gdy chcemy sprawdzić czy użytkownik jest zalogowany i ma uprawnienia do oglądania strony i w momencie gdy takowych nie posiada wyświetlić informacje o błędzie.

* 1. Sesja

Sesja została zaimplementowana przy pomocy Bazy danych SQLite oraz ciasteczek przechowywanych w przeglądarce.

Przy obsłudze requestu w momencie rozpoznania że nie odwołuję się on do pliku statycznego startowana jest sesja.



Rys. Diagram startowania sesji

Do tworzenia klucza użyto algorytmu hashującego SHA1 który zwraca 160-bitowy skrót wiadomości. Skrót wyliczany jest na podstawie aktualnego czasu w milisekundach 32 bitowej liczby wygenerowanej przy pomocy wbudowanego generatora liczb pseudolosowych i id ostatniego dodanej sesji.

* 1. Przekazywanie informacji do funkcji aplikacji internetowej

SDK Dostarcza klasę Request i Response. Klasa request przekazywana jest w parametrze funkcji odpowiadającej na zapytanie HTTP i zawiera informacje związane z zapytaniem.

public class Request

{

public String Url { get; set; }

public String Method { get; set; }

public Dictionary<String, String> FormData=new Dictionary<string,string>();

public Session Session { get; set; }

public GroupCollection Variables { get; private set; }

public Request(String url)

{

this.Url = url;

}

}

Src. Klasa Request

* *Url* zawiera ścieżkę z jaką zostało wysłane zapytanie
* *Method* zawiera metode zapytanie
* *FormData* sparsowane informacje przekazane w kontencie zapytania
* *Session* obiekt sesji (już ustanowionej)
* *Variables* parametry wyciągnięte z adresu URL przy pomocy wyrażeń regularnych

Klasa Response jest zwracana przez funkcje obsługującą zapytania HTTP, i przechowuje w sobie zawartość wysyłaną do klienta. Po wywołaniu odpowiedniej metody można wymusić przekserowanie na inny adres.

Konstruktory pozwalają przyjąć :

* Dane binarne
* Tekst
* Ścieżkę prowadzącą do templatu oraz obiekt przechowujący informacje o paramtrach przekazanych to templatu

Jest także statyczna metoda *Redirect*,która zwraca obiekt *Response* i pozwala wywołać przekierowanie.

Klasa zawiera także publiczne właściwości:

* *Status* - kod statusu zwracany do klienta
* *MimeType* - typ odpowiedzi wysłanej do klienta

Src. Klasa Response

public class Response

{

public byte[] Data { get; set; }

private string redirectUrl;

public int Status { get; set; }

public string MimeType { get; set; }

private Response()

{

Status = 200;

}

public Response(byte[] data)

{

Status = 200;

this.Data = data;

}

public Response(string data)

{

Status = 200;

this.Data = System.Text.Encoding.GetEncoding("iso-8859-2").GetBytes(data);

}

public Response(String name, Hash parameters)

{

Status = 200;

FileStream fs = new FileStream("FeintSite/" + Settings.ViewsFolder + name, FileMode.Open, FileAccess.Read);

StreamReader reader = new StreamReader(fs, Encoding.UTF8);

var contents = reader.ReadToEnd();

fs.Close();

Template template = Template.Parse(contents);

String renderedTemplate = template.Render(parameters);

this.Data = System.Text.Encoding.GetEncoding("iso-8859-2").GetBytes(renderedTemplate);

}

public static Response Redirect(String url)

{

Response r = new Response();

r.redirectUrl = url;

return r;

}

}

* 1. ORM

Na ORM składa się kilka ważnych klas

* *FeintORM* klasa która odpowiada z tworzenie tabel na podstawie klas modelu
* *DBModel* klasa po której dziedziczyć musi każda klasa której model ma zostać odwzorowany w bazie danych
* *DBProperty* klasa atrybut który musi zostać dodany do każdej właściwości, której odwzorowanie ma pojawić się w tabeli
* *DBForeginKey* klasa przechowująca informacje o relacjach
* *WhereBuilder* klasa odpowiedzialna za wykonywanie zapytań oraz mapowanie otrzymanych danych do obiektów
  1. FeintORM

Przy każdej inicjalizacji *FeintORM* następuje tworzenie nieistniejących tabel. Podczas tworzenia tabeli należy zachować odpowiednią kolejność ponieważ aby utworzyć relacje musi występować tabela, do której się odnosi. Aby to uzyskać została utworzona specjalna list utworzonych do tej pory klas i w momencie gdy ORM próbuje utworzyć tabelę przy każdej relacji sprawdzane jest czy tabela, do której się odnosi istnieje, gdy nie została ona jeszcze utworzona zostaje wywołana rekursywnie funkcja tworząca potrzebną tabele.

Src. Funkcja wywołująca funkcje tworzącą rekursywnie tabele

public void CreateTablesFromModel()

{

Helper.CreateDatabase(settings.Name);

var types = getAllModelClass();

tablesCreated = new List<string>();

foreach (var t in types)

{

if(tablesCreated.IndexOf(t.Name)<0)

createTable(t);

}

}

Src. Funkcja tworzące tabele z relacjami

private void createTable(Type t)

{

var tableName = Prefix+t.Name;

List<Column> collumns = new List<Column>();

List<Foreign> foreigners = new List<Foreign>();

var properties = getPropertiesFromClass(t);

foreach (var p in properties)

{

var attr = (DBProperty)p.GetCustomAttribute(typeof(DBProperty));

var collumn = new Column(p.Name, Helper.getDBType(p.PropertyType), attr.PrimaryKey, attr.AutoIncrement, attr.Unique, attr.AllowNull);

collumns.Add(collumn);

}

var foreignersTypes = getForeignersFromClass(t);

foreach (var f in foreignersTypes)

{

//sprawdzenie czy została już utworzona tabela która jest zależnością

if (tablesCreated.IndexOf(f.PropertyType.GenericTypeArguments[0].Name) < 0)

createTable(f.PropertyType.GenericTypeArguments[0]); //rekursywne wywołanie funkcji

var collumn = new Column("fk\_" + f.Name, "INTEGER");

var foreignKey = new Foreign("fk\_" + f.Name, Prefix + f.PropertyType.GenericTypeArguments[0].Name, "id") { Col = collumn };

collumns.Add(collumn);

foreigners.Add(foreignKey);

}

this.Helper.CreateTable(tableName, collumns, foreigners);

tablesCreated.Add(t.Name);

}

Do pobrania informacji o modelach wykonywana jest refleksja. Najpierw wyszukiwane są wszystkie modele z każdego modułu aplikacji.

Src. Wyszukiwanie modeli

public IEnumerable<Type> getAllModelClass()

{

foreach (var assembly in assemblies)

{

foreach (var t in assembly.GetTypes())

if (t.BaseType == typeof(DBModel))

yield return t;

}

}

W powyższym przykładzie został wykorzystany mechanizm *yield return* co pozwala nam na rozpoczęcie przetwarzania pierwszych elementów kolekcji gdy cała funkcja jeszcze nie skończyła się wykonywać.

Aby pobrać informacje o właściwościach modelu na podstawie, których mają zostać utworzone kolumny w tabelach zostały stworzone dwie funkcje. Jedna odpowiada za pobranie wszystkich informacji o kolumnach a druga o relacjach do innych tabel.

Src. Pobieranie informacji o kolumnach i relacjach

public IEnumerable<PropertyInfo> getPropertiesFromClass(Type t)

{

foreach (var p in t.GetProperties())

if (p.GetCustomAttributes(typeof(DBProperty), false).Length > 0)

yield return p;

}

public IEnumerable<PropertyInfo> getForeignersFromClass(Type t)

{

foreach (var p in t.GetProperties())

if (p.PropertyType.IsGenericType && p.PropertyType.GetGenericTypeDefinition() == typeof(DBForeignKey<>))

yield return p;

}

Klasa FeintORM przy inicjalizacji pobiera wybranego w ustawieniach *DatabaseHelper'a* i na podstawie danych także zapisanych w ustawieniach wykonuje połączenie do bazy danych.

* 1. DBModel

Klasa po której dziedziczyć musi każdy obiekt przechowywany w bazie danych. Każdy model posiada już istniejącą właściwość Id, która później zamieniana jest na klucz główny w tabeli. Posiada też zestaw metod które pozwalają na wygodne wykonywanie operacji na rekordach tabeli. Tworzenie nowego rekordu polega na stworzeniu nowego obiektu i wypełnieniu go damami. Jedyną daną, która nie może zostać zmieniona jest Id. Aby stworzony obiekt został zapisany w bazie danych należy wywołać funkcje *Save* bądź też wywołać statyczną metodę *Add* przekazując jej w parametrze obiekt modelu.

Funkcja statyczna *Add* na początku sprawdza czy nie jest to już istniejący model, jeżeli jego *Id* jest różne od zera operacja dodawania zostaje przerwana. Aby dodać rekord pobierane są wszystkie właściwości modelu oznaczone jak *DBProperities* a następnie przetwarzane tak aby uzyskać informacje o kolumnie. Następnie przetwarzane są relacje jeżeli obiekt do którego prowadzi relacja nie istnieje wykonywana jest rekurencyjna funkcja *Add* dla jego modelu.

Src. Dodawanie nowych obiektów jako rekord do bazy

Funkcja Save pozwala zarówno na stworzenie nowego rekordu jak i jego modyfikacje. Na podstawie *Id* podejmowana jest decyzja czy obiekt jest nowym obiektem bazy danych czy też jest modyfikowany. W momencie modyfikacji wszystkie pola rekordu są aktualizowane zgodnie z tymi zapisanymi w obiekcie.

Do usuwania rekordów wykorzystywana jest funkcja *Remove* na podstawie *Id* wybierany jest rekord, który zostanie poddany usunięciu.

Do pobrania pojedynczego obiektu jeśli posiadamy jego id wykorzystywana jest funkcja *Ref*, która jako parametr przyjmuje *Id* obiektu, który chcemy pobrać.

Src. Pobieranie pojedyńczego rekordu

public static T Ref<T>(long id)

{

return Find<T>().Where().Eq("Id",id).Execute()[0];

}

Aby pobrać więcej niż jeden rekord należy wywołać funkcje *Find* zwraca ona obiekt typu *QueryBuilder* z którego mamy dostęp do obiektu *WhereBuilder*.

Src. Pobieranie wielu rekordów

public static T Ref<T>(long id)

{

return Find<T>().Where().Eq("Id",id).Execute()[0];

}

* 1. WhereBuilder

WhereBuilder jest klasą która pozwala pobranie wielu rekordów z bazy z warunkami jakie chcę wykorzystać klient posiada także możliwość wybrania sposobu sortowania a także wykorzystać klauzule limit.

Funkcja *convertParameters* odpowiada za przetworzenie parametrów z typu obiektowego do typu relacyjnego odpowiada za dodanie prefiksów do nazw kolumn analizę relacji i dodanie odpowiednich join'ów pomiędzy tabelami.

Src. Przetwarzanie parametrów przed wysłaniem zapytania

Mapowanie na obiekty odbywa sie przy pomocy refleksji. Pobierane są wszystkie właściwości, które zostały zapisane jako modele, które zostaną zapisane w bazie danych. Następnie wypełniane są na podstawie typów jakie właściwości posiadają.

Src. Mapowanie rekordu na obiekt

foreach (PropertyInfo p in pr)

{

int i = columns[p.Name];

if (p.PropertyType != dateType)

{

if (p.PropertyType == int32Type&&row.ItemArray[i].GetType()==typeof(Int64))

{

p.SetValue(obj, Convert.ToInt32((Int64)row.ItemArray[i]), null);

}

else

{

p.SetValue(obj, row.ItemArray[i], null);

}

}

else

{

DateTime dt = new DateTime (1970, 1, 1, 0, 0, 0, 0);

dt= dt.AddSeconds (((long)row.ItemArray [i])/1000);

p.SetValue(obj,dt, null);

}

}

Mapowanie relacji następuje w podobny sposób przy czym cała zawartość obiektu nie jest pobierana w tym samym momencie. W trakcie pobierania rekordów uzupełniany jest tylko id relacji i w momencie pierwszego odwołania do obiektu następuje jego pobranie.

Src. Mapowanie relacji

foreach (PropertyInfo f in fr)

{

Object fobj = Activator.CreateInstance(f.PropertyType, true);

var value=row.ItemArray[columns["fk\_" + f.Name]];

if (value!=null&&value.GetType() != typeof(DBNull))

f.PropertyType.GetProperty("Id", BindingFlags.Instance|BindingFlags.NonPublic|BindingFlags.Public).SetValue(fobj, value);

f.SetValue(obj, fobj);

}

Z pomocą *WhereBuilder'a* można także pobrać ilość rekordów spełniającą dane zapytanie.

* 1. DBProperty i DBForginKey

*DProperty* jest klasą atrybutem stosowanym do oznaczania właściwości w klasach modelu. W atrybucie tym ustawiamy specyficzne dla kolumn włsćiwości.

* PrimaryKey
* AutoIncrement
* Unique
* AllowNull

*DBForeginKey* jest klasą przechowującą informacje o tworzonych relacjach posiada ona parametr typu generycznego który musi dziedziczyć po *DBModel*.

Src. Nagłówek klasy DBForeginKey

public class DBForeignKey<T> where T:DBModel

Przy pobieraniu tabeli pobierany jest tylko *Id* relacji a wartość obiekt pobierany jest dopiero w momencie odwołania się do *Value*

Src. Właściwość Value DBForeginKey

1. Tutorial