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

Rozdział 1
WSTĘP

1.1 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.

Rozdział 2

ARCHITEKTURA FRAMEWORK'A

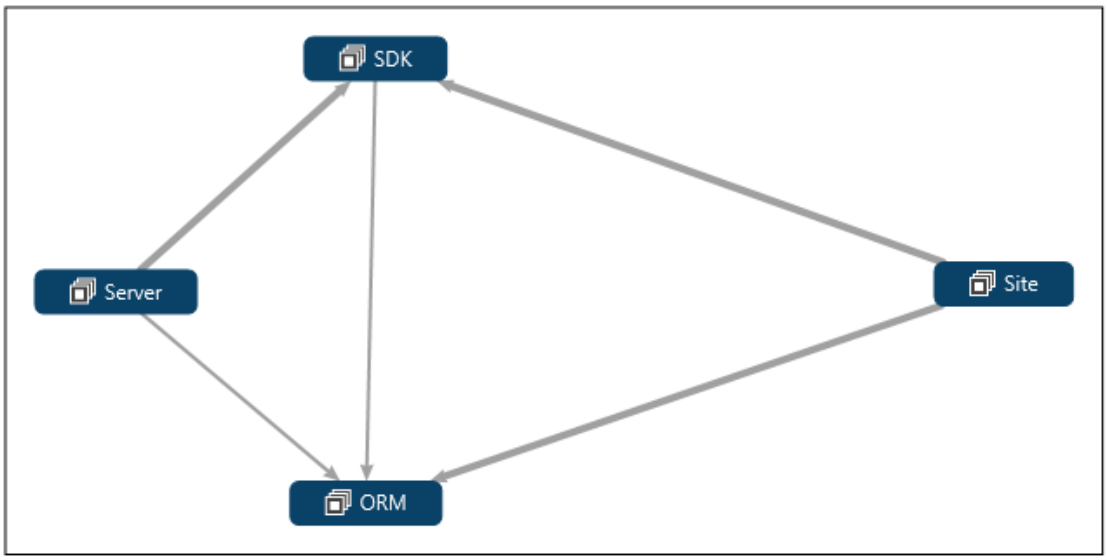
2.1 Główne moduły aplikacji.

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

2.2 Sposób połączenia pomiędzy modułami

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



Rys. 1 Graf przedstawiający zależności pomiędzy głównymi modułami framework'a

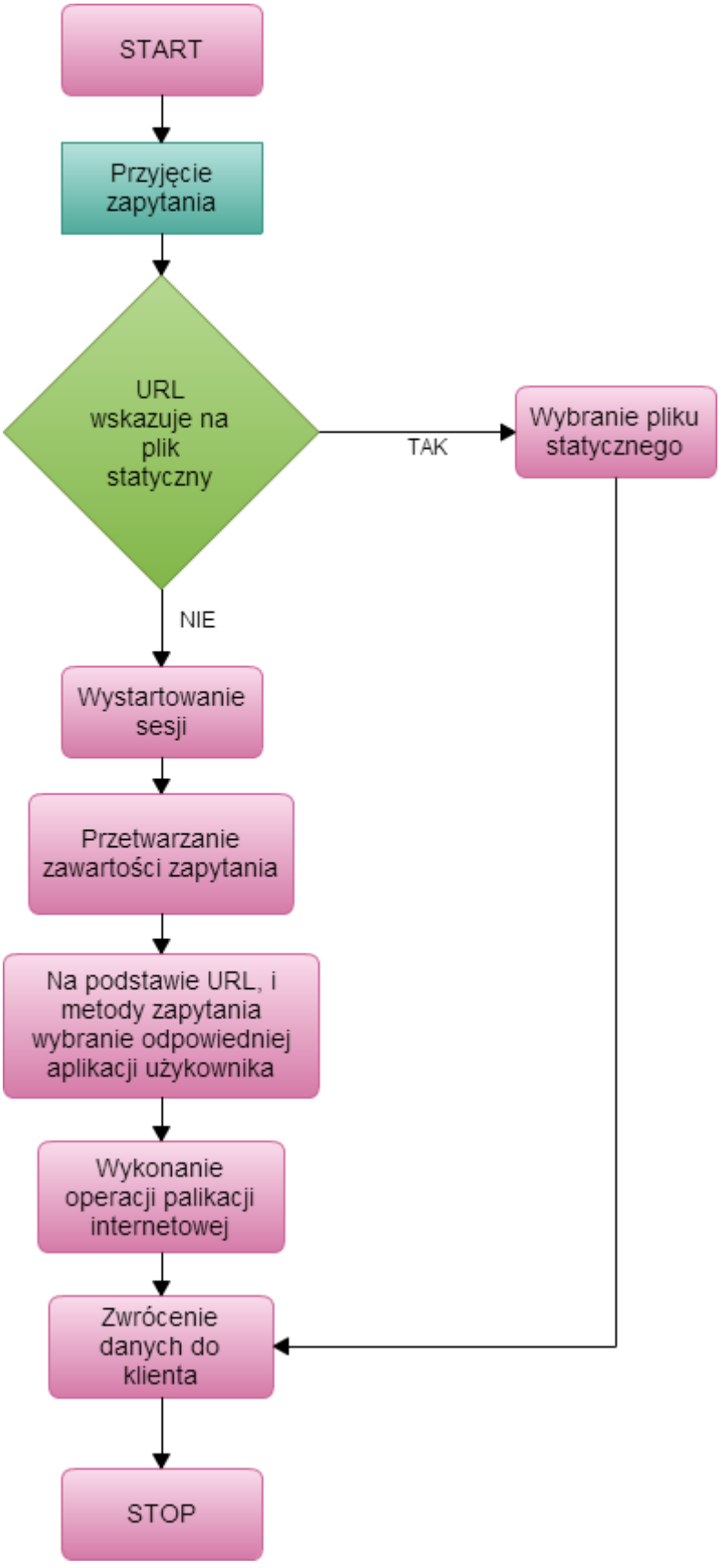
Aplikacja pisana przez użytkownika nie ma dostępu do serwera aplikacji. Warstwą pośredniczącą jest SDK.

2.3 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. 2 Diagram przedstawiający schemat obsługi zapytania

2.4 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

2.5 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.

Rozdział 3

SZCZEGÓŁY IMPLEMENTACYJNE

3.1 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.

```
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);
}
```

Src. 1 Ładowanie modułów do serwera i wywoływanie funkcji inicjalizacyjnej

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

```
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;
}
```

Src. 2 Wyszukiwanie statycznej metody Main w bibliotece

3.2 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

3.3 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

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

Src. 3 Dodanie URL obsługującego zapytanie

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.

```
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;
}
```

Src. 4 URL dispatcher

3.4 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.

```
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);
}
```

Src. 5 Programowanie nieaspektowe przykład

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

```
[Log]
public void transferMoney(int accId1, int accId2, int amount)
{
	accounts[accId1] -= amount;
	accounts[accId2] += amount;
}
```

Src. 6 Programowanie aspektowe przykład

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

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.