POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Przemysław Walerianczyk
Nr albumu 7143

Różne podejścia do projektowania aplikacji internetowych

Praca magisterska napisana pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, czerwiec 2015

Streszczenie

Praca dotyczy istotnego problemu, jakim jest wybór odpowiedniego podejścia do projektowania aplikacji internetowych. Mnogość rozwiązań sprawia, że coraz trudniej podjąć właściwą decyzję. Dokumentacja opisuje trzy możliwe sposoby implementacji. Praca nie ma na celu odpowiedzieć na pytanie, które podejście jest najlepsze, a jedynie ułatwić dokonanie właściwego dla zadanych wymagań wyboru. Praca przedstawia fazę analizy omawianego prototypu, przebieg implementacji we wszystkich trzech podejściach oraz wnioski z tego wynikające. Prototypy zostały stworzone w następujący sposób:

- Wizualne projektowanie – w tym celu wykorzystano technologię ASP.NET WebForms
- Wzorzec architektoniczny MVC – w tym celu wykorzystano technologię ASP.NET MVC
- Wzorzec architektoniczny MVC z wykorzystaniem biblioteki JavaScript – w tym celu wykorzystano połączenie technologii ASP.NET MVC z biblioteką AngularJS

Wszystkie trzy projekty wykorzystują .NET Framework w wersji 4.5.1, językiem programowania jest C#. Dodatkowo została użyta biblioteka CSS Bootstrap oraz mechanizmy ASP.NET Identity 2 i Microsoft Entity Framework.

Spis treści

1. Wstęp

Programowanie aplikacji internetowych staje się coraz bardziej popularne. Powoli zaczyna wypierać ono tworzenie tzw. „grubych klientów”. Dostarczają one więcej problemów nie tylko w kwestii utrzymania, ale także w kwestii aktualizacji danego oprogramowania. Aplikacje webowe umożliwiają łatwiejszy dostęp, wystarczy, że klient zna adres aplikacji i bez problemu może się do niej dostać. Łatwość propagowania, aktualizacji oraz utrzymania sprawia, że aplikacje internetowe są pierwszym wyborem przy tworzeniu nowych projektów.

Jak zatem napisać dobrą aplikację webową? Jaki framework wykorzystać? Które podejście pozwoli najszybciej stworzyć taką aplikację, a które sprawi, że rozwijanie jej będzie proste i przyjemne? Niniejsza praca na pewno jednoznacznie nie odpowie na powyższe pytania, jednak przedstawi dostępne możliwości i pomoże w dokonaniu odpowiedniego do potrzeb wyboru.

1.1. Cel pracy

Celem niniejszej pracy było przedstawienie poszczególnych podejść do tworzenia aplikacji internetowych. Analiza miała posłużyć do wyciągnięcia pewnych wniosków i ewentualnej pomocy w wyborze określonego podejścia. Dokumentacja opisuje poszczególne implementacje, jak również etapy powstawania aplikacji tworzonej na potrzeby pracy.

W celu zbadania stawianego przed pracą problemu powstał prototyp aplikacji - sklep internetowy. Odpowiedzialny jest on za sprzedaż desek snowboardowych. Poszczególne podejścia do tworzenia prototypu można podzielić na:

- Wizualne projektowanie – w tym celu wykorzystano technologię ASP.NET WebForms
- Wzorzec architektoniczny MVC – w tym celu wykorzystano technologię ASP.NET MVC
- Wzorzec architektoniczny MVC z wykorzystaniem biblioteki JavaScript – w tym celu wykorzystano połączenie technologii ASP.NET MVC z biblioteką AngularJS

1.2. Rozwiązania przyjęte w pracy

Z uwagi na fakt, że, na co dzień zajmuję się programowaniem aplikacji internetowych w technologiach Microsoftu praca będzie oparta o rozwiązania dostarczane właśnie przez tę firmę. Aby w pewnym stopniu oddać relację klient-wykonawca system będzie tworzony na podstawie przygotowanej już wcześniej bazy danych z odpowiednimi tabelami.

Wszystkie trzy podejścia zostały napisane z wykorzystaniem .NET Framework, języka programowania C#, biblioteki CSS Bootstrap oraz bazy danych Microsoft SQL Server.

1.3. Rezultat pracy

Głównym wynikiem pracy jest analiza poszczególnych podejść do programowania aplikacji internetowych. Ułatwia ona podjęcie decyzji w kontekście wyboru podejścia do tworzenia aplikacji internetowych.

Podczas pisania pracy powstały trzy odrębne aplikacje stanowiące prototypy do sporządzonych wcześniej wymagań jak również struktury bazy danych.

1.4. Organizacja pracy

Na początku pracy zostanie przedstawione każde z trzech podejść do programowania aplikacji internetowych. Stanowić to będzie swego rodzaju teoretyczny wstęp i zaznajomienie z technologią.

Następnym etapem będzie przedstawienie wymagań funkcjonalnych, jakie powinny spełniać tworzone prototypy rozwiązań. Dodatkowo pokazana zostanie struktura bazy danych, która powinna być przełożona na obiekty biznesowe występujące w aplikacjach.

Kolejną rzeczą będzie pokazanie implementacji poszczególnych rozwiązań. Każde rozwiązanie będzie opatrzone swego rodzaju notatkami z przebiegu pracy. Zawarte tam będą problemy, na jakie

można napotkać przy wykorzystaniu konkretnego podejścia jak również ciekawostki ułatwiające tworzenie aplikacji.

Ostatni etap będzie składał się z analizy i wniosków. Przedstawione zostaną testy i porównania wszystkich podejść w kontekście prędkości działania (wczytywania w przeglądarce internetowej), wielkości zajmowanego miejsca oraz obciążenia procesora jak i pamięci operacyjnej.

2. Przedstawienie omawianych technologii

Poniższy rozdział służy do teoretycznego przedstawienia technologii, dzięki którym powstały prototypy aplikacji.

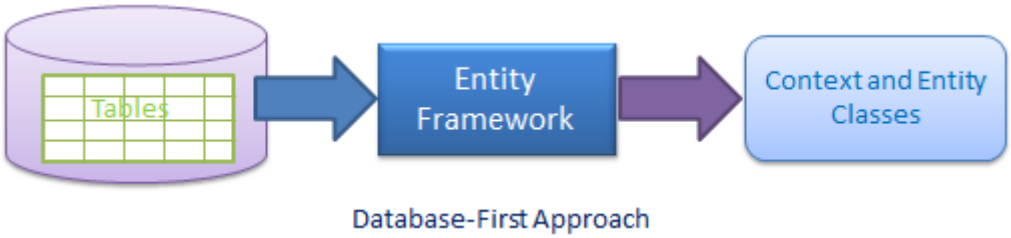
2.1. Microsoft Entity Framework

Entity Framework to technologia, która wspiera programistów podczas tworzenia aplikacji. Technologię wykorzystuje się w momencie korzystania z danych zewnętrznych, zazwyczaj baz danych. Entity Framework pozwala przekształcić model relacyjnej bazy danych do świata obiektów, na których programista w prosty sposób może wykonywać rozmaite operacje. Technologia ta jest najczęściej wykorzystywana, kiedy mamy do czynienia z aplikacjami wielowarstwowymi, gdzie dostęp do danych stanowi odrębna warstwa, odseparowana od warstwy prezentacyjnej [1].

Entity Framework opiera się na modelu danych, który może być stworzony na trzy sposoby:

- *Database First* – jak sama nazwa wskazuje, pierwszym etapem jest stworzenie bazy danych, a na jej podstawie generowany jest model, w którym znajdą się istniejące w bazie tabele, widoki oraz procedury składowane (Rysunek 1).
- *Model First* – nie wymaga tworzenia bazy danych za pomocą kodu SQL. Model danych tworzony jest w specjalnym designerze, a następnie na jego podstawie generowana jest struktura bazy.
- *Code First* - w pierwszej kolejności piszemy gotowe klasy. Muszą one być zaprojektowane w określony sposób, a poszczególne właściwości dodatkowo okraszone odpowiednimi atrybutami. Na podstawie definicji klas tworzona jest struktura bazy danych.

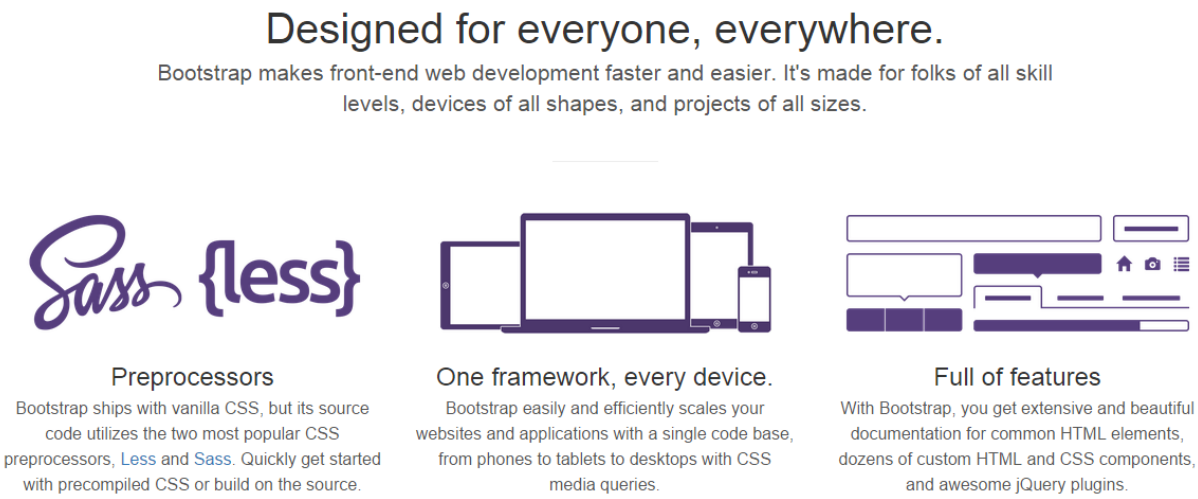
Podczas implementacji prototypów wykorzystano podejście *Database-First*, aby w pewnym stopniu zaprezentować relację klient-wykonawca. Podejście to wymusza dostosowanie się programisty do istniejącej bazy danych. Z drugiej strony nie musi on przejmować się relacjami oraz atrybutami poszczególnych obiektów, wszystko będzie utworzone automatycznie na podstawie bazy danych. Jest to bardzo prosty i szybki sposób, a w przypadku dobrze zamodelowanej bazy nie potrzeba dodatkowych działań ze strony programisty.



Rysunek 1 Schemat *Database-First*.
Źródło: [2]

2.2. Bootstrap

Coraz więcej osób przegląda strony internetowe korzystając z różnych urządzeń takich jak: telefony, tablety, komputery osobiste. W związku z tym tworzenie witryn zaczęło wymagać, aby wyświetlały się one poprawnie na jak największej ilości różnych nośników. Bootstrap jest biblioteką CSS, który ułatwia tworzenie takich stron internetowych. Został stworzony przez programistów Twittera. Aktualna wersja (3.3.4) została użyta do zbudowania milionów stron na świecie.



Rysunek 2 Bootstrap według jego twórców. Źródło: [3]

Bootstrap sprawia, że tworzenie stron internetowych jest szybsze i łatwiejsze. Zawiera gotowe zestawy styli między innymi dla tabel, formularzy, przycisków, ikon. Mogą go używać ludzie na wszystkich poziomach zaawansowania. Ogromną zaletą jest fakt, że za pomocą jednego kodu bazowego jesteśmy w stanie łatwo i skutecznie przeskalować aplikacje na urządzenia o różnych kształtach i o różnej rozdzielczości. Warto wspomnieć, że to właśnie Bootstrap bierze na siebie odpowiedzialność zgodności na różnych przeglądarkach. Twórca stron nie musi już się tym przejmować, a to powoduje znaczne przyspieszenie implementacji rozwiązań [3].

Bootstrap nie wymusza od projektantów używania tylko i wyłącznie jednego stylu. Istnieje możliwość dostosowania go do swoich potrzeb. Zmiana kolorów, czcionki, wykluczenie niepotrzebnych komponentów – to wszystko jest możliwe za pośrednictwem strony http://getbootstrap.com/customize. Oprócz tego istnieje mnóstwo motywów gotowych do ściągnięcia i użycia w interesującym nas projekcie. Wielu projektantów specjalizuje się w przygotowywania motywów dla Bootstrap. Dostępność różnych opcji jest naprawdę spora.

Visual Studio wykorzystuje szablony do tworzenia projektów webowych. Szablon taki może tworzyć pliki, struktury folderów oraz dołączać biblioteki potrzebne do nowego projektu. Szablony zostały stworzone po to, aby wdrożyć najnowsze standardy internetowe oraz pokazać najlepsze praktyki, jak korzystać z technologii ASP.NET, a także dać ułatwić i przyspieszyć tworzenie nowych aplikacji internetowych. Wraz z wersją Visual Studio 2013 Microsoft dostarczył programistom szablonów, które domyślnie zawierają framework Bootstrap. Zestawy czcionek, styli, skryptów JavaScript – wszystko podłączone i gotowe do działania. Dzięki temu szablony ASP.NET pozwalają zbudować responsywne witryny, które dobrze wyglądają na urządzeniach mobilnych, tabletach i komputerach, a implementacja takich rozwiązań jest łatwa i przyjemna [4].

2.3. ASP.NET Identity 2

Mechanizm ASP.NET Identity 2 został wydany 20 marca 2014. Zawierał długo oczekiwane właściwości związane z bezpieczeństwem oraz zarządzaniem kontami. Rozszerzono możliwości zabezpieczania i autoryzacji wszystkich typów aplikacji ASP.NET. ASP.NET Identity został początkowo zaprezentowany w 2013 roku, jako kontynuacja poprzedniego (mającego już swoje lata) rozwiązania – ASP.NET Membership. ASP.NET Identity w wersji pierwszej prezentował przydatne funkcje, jednak API było nieco minimalistyczne. Wersja druga przyniosła sporo udoskonaleń między innymi w obszarze integracji logowania za pomocą serwisów społecznościowych, a także rozszerzania definicji modelu użytkownika [5].

Obecnie najnowszą wersje stanowi ASP.NET Identity 2.2.1 wydaną 7 kwietnia 2015 roku. Aktualnie trwają prace nad wersją ASP.NET Identity 3.0, która będzie częścią nowego frameworku ASP.NET 5 [6].

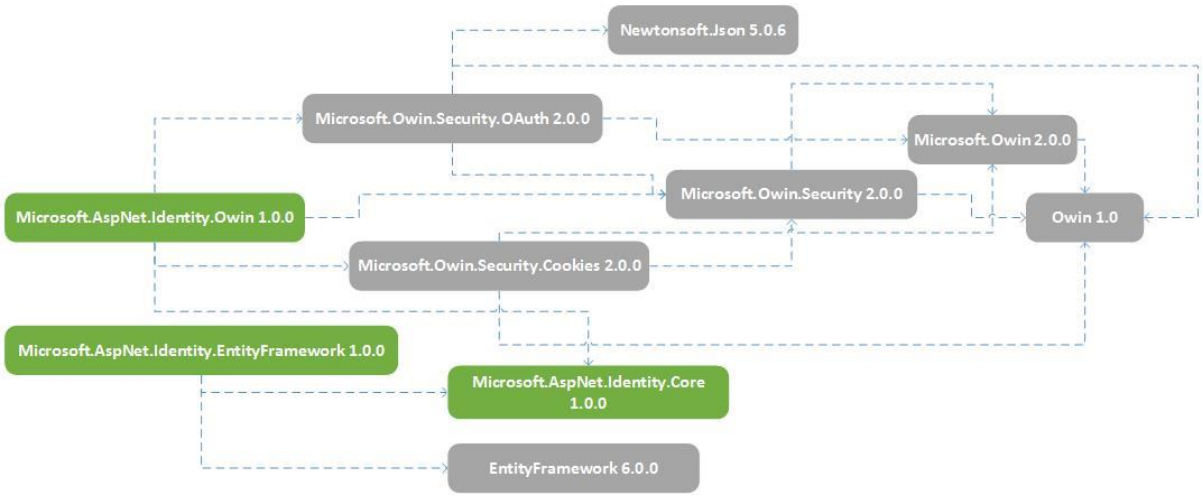
2.3.1. Nowe funkcje

Poniżej przedstawiono kilka nowych funkcji wprowadzonych w mechanizmie ASP.NET Identity 2:

- Rozszerzenie definicji konta użytkownika, w tym adresu e-mail oraz danych kontaktowych
- Dwuskładnikowe uwierzytelnianie poprzez adres e-mail lub wiadomość SMS – funkcjonalność podobno do tej stosowanej przez Google, Microsoft
- Potwierdzenie założenia konta poprzez e-mail
- Zarządzanie administracyjne użytkownikami i rolami
- Blokowanie kont w momencie nieprawidłowej próby logowania
- Ulepszona obsługa logowań za pomocą portali społecznościowych

2.3.2. Komponenty

Rysunek 3 przedstawia schemat komponentów systemu ASP.NET Identity. Pakiety zaznaczone na zielony kolor tworzą system Identity. Pozostałe stanowią zależności niezbędne do korzystania z mechanizmu w aplikacjach ASP.NET.



Rysunek 3 Komponenty ASP.NET Identity. Źródło: Opracowanie własne na podstawie [7]

2.3.3. Opcje uwierzytelniania

Visual Studio 2013 oferuje kilka opcji uwierzytelniania dostępnych dla aplikacji tworzonych podejściem ASP.NET WebForms lub ASP.NET MVC. Są to:

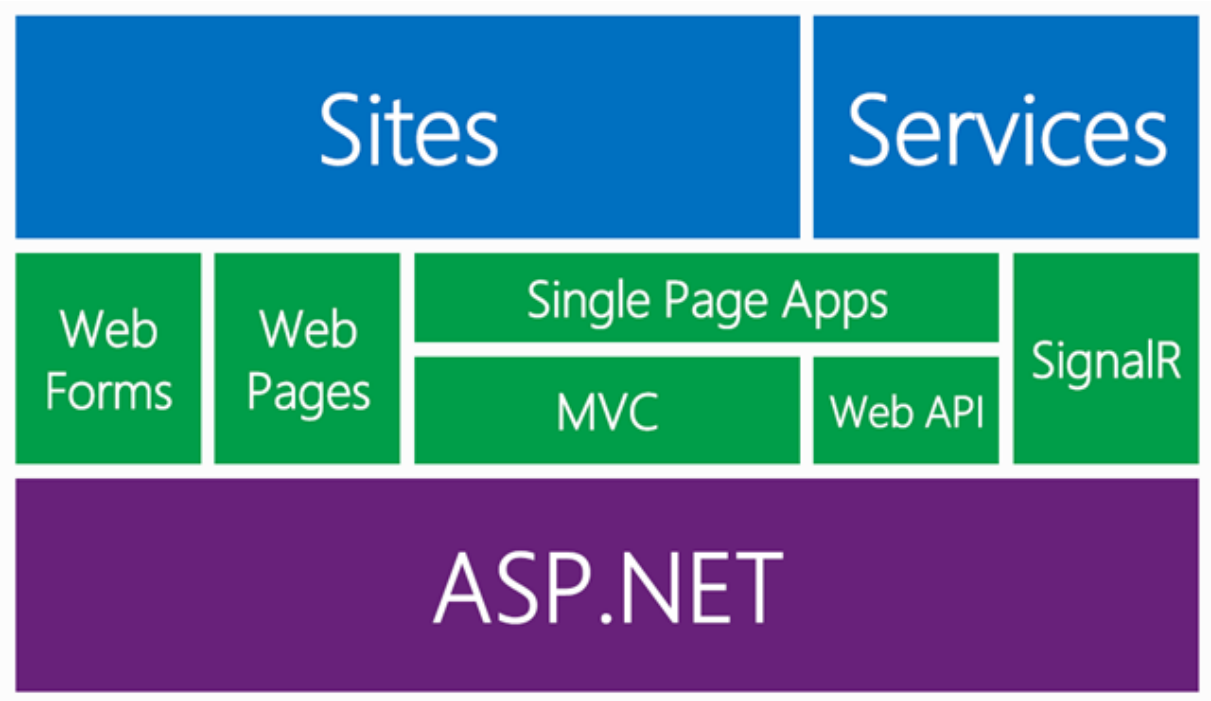
- Brak uwierzytelniania
- Indywidualne konta użytkownika (ASP.NET Identity)
- Konta organizacyjne (Windows Server Active Directory lub Azure Active Directory)
- Uwierzytelnianie Windows

Podczas tworzenia nowego projektu w momencie wyboru szablonu (Rysunek 12 na stronie 26), programista ma możliwości wyboru interesującej go opcji uwierzytelniania (Rysunek 13 na stronie 27).

2.4. ASP.NET MVC

Microsoft ASP.NET MVC jest frameworkiem do budowania aplikacji webowych stworzonym przez firmę Microsoft jako część platformy .NET Framework. Głównie opiera się on na wzorcu projektowych Model-View-Controller (MVC), a jako główną zaletę wymienia się łatwość zarządzania

tworzonego kodu. Pierwsza wersja powstała w 2008 roku [8]. Rysunek 4 pokazuje, w którym miejscu w aktualnej wersji .NET Framework znajduje się ASP.NET MVC.



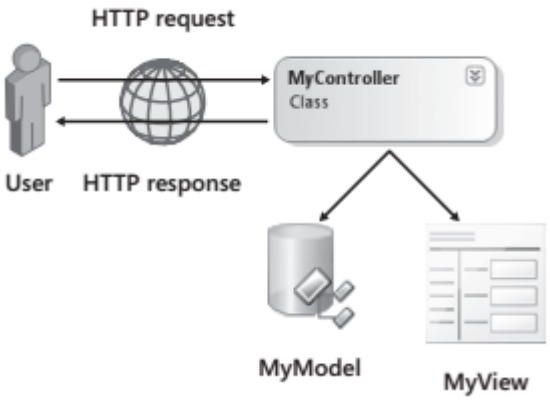
Rysunek 4 Struktura ASP.NET 4.5.
Źródło: [9]

Jako główne zalety ASP.NET MVC można wymienić:

- Łatwość zarządzania kodem, poprzez podział aplikacji na model, widok i kontroler
- Kontrola nad generowanym kodem HTML
- Nie występuje mechanizm ViewState
- Łatwość testowania
- Zaawansowany system routingu
- Kontrola pomiędzy żądaniami pomiędzy przeglądarką a serwerem

2.4.1. Wzorzec architektoniczny MVC

MVC (*Model-View-Controller*) jest wzorcem architektonicznym używanym podczas projektowania aplikacji z interfejsem użytkownika. Polega na wyraźnym odseparowaniu warstwy danych, logiki biznesowej oraz prezentacji. Rysunek 5 przedstawia architekturę poszczególnych komponentów MVC. Żądania są kierowane do kontrolerów, które współpracują z modelem danych oraz wybierają odpowiedni widok. Użytkownik poprzez przeglądarkę wywołuje metodę klasy kontrolera.



Rysunek 5 Architektura MVC.
Źródło: [10]

Model stanowią dane oraz związana z nimi logika biznesowa. Do tego dochodzą zasady walidacji po stronie aplikacji. Na modelu możemy wykonywać operacje CRUD (*Create-Retrieve-Update-Delete*), które będą odzwierciedlane na źródle danych, na przykład bazie SQL Server. Model może stanowić kolekcja obiektów, pojedynczy obiekt, a nawet pojedyncza zmienna. Jest to twór, który zostanie przekazy przez kontroler do widoku i wyświetlony użytkownikowi.

Widok reprezentuje dane oraz decyduje jak będą one wyświetlane po stronie warstwy prezentacyjnej. W aplikacjach internetowych widok jest odpowiedzialny za generowanie kodu HTML w przeglądarce użytkownika końcowego. Może być to również inna postać, jak XML, JSON czy zwykły tekst.

Kontroler jest odpowiedzialny za komunikację z użytkownikiem. Przyjmuje dane wejściowe, aktualizuje model oraz odświeża widoki. Może wykonywać różne czynności związane z użytkownikiem, jak na przykład autoryzacja. To kontroler pobiera dane z bazy danych, aktualizuje je. Można powiedzieć, że głównym zadaniem kontrolera jest sterowanie logiką aplikacji.

2.4.2. Routing

Routing ASP.NET umożliwia korzystać z adresów URL, które nie wskazują na fizyczny plik w witrynie internetowej. Pozwala to na używanie linków, które opisują swoje działanie użytkownikowi, a przez to są dla niego bardziej zrozumiałe. System routing w ogólnej perspektywie odpowiedzialny jest za dwie funkcje:

- Badanie przychodzącego adresu URL w celu wykonania odpowiedniej akcji kontrolera, którego żądanie dotyczy
- Generowanie linków wychodzących, które wyświetlane są w widokach dla użytkowników końcowych [11]

System Routing ASP.NET opiera się na zestawie tak zwanych *route*. Nie ma potrzeby definiowania *route* dla każdego adresu, które chcemy obsłużyć. Zamiast tego każda *route* może zawierać wzór URL, który jest przyrównywany do przychodzącego żądania. Jeśli wzór pasuje do adresu URL wtedy system routingu przetwarza taki URL. Adres URL mogą być podzielone na segmenty. Stanowią one części adresu bez nazwy hosta i oddzielone są znakiem /. Na przykład adres *http://localhost:38596/Store/Board/1* składa się z trzech segmentów:

- Store
- Board
- 1

Pierwszy segment wskazuje na kontroler, drugi na akcję kontrolera, natomiast trzeci jest to parametr przekazywany do metody. Wzór URL, który przechwyci takie żądanie wygląda następująco:

{controller}/{action}/{id}

Natomiast definicję *route* w aplikacji pokazano na listingu 1.

Listing 1 Definicja route. Źródło: Opracowanie własne.

```
routes.MapRoute(
     name: "Default",
     url: "{controller}/{action}/{id}",
     defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
);
```

Wszystkie zdefiniowane *route* są rejestrowane w aplikacji w momencie jej uruchomienia.

2.4.3. Razor

Silnik Razor jest językiem znaczników używanym po stronie warstwy prezentacyjnej. Pozwala na osadzanie kodu serwerowego w widokach. Dzięki temu można tworzyć dynamiczną zawartość stron. Są one przetwarzane przez silnik ASP.NET w locie, a następnie wysyłane do przeglądarki. Kod wykonywany po stronie serwera może zrobić zadania, których nie jest w stanie zrobić przeglądarka, jak na przykład dostęp do bazy danych.

Widoki są kompilowane przez silnik Razor w celu zwiększenia wydajności. Są one tłumaczone na język C#. Dzięki temu możemy używać fragmentów kodu C# w widokach. Strony są kompilowane dopiero w momencie uruchomienia aplikacji, więc aby podejrzeć, co wygenerował silnik Razor należy zainicjować pierwsze żądanie do aplikacji. Wywoła to proces kompilacji dla wszystkich widoków. Tak wygenerowane pliki z kodem C# są zapisywane na dysku, domyślnie pod lokalizacją: „C:\Users\<NazwaUżytkownika>\AppData\Local\Temp\Temporary ASP.NET Fileson Windows”. Niestety nazwy plików nie odpowiadają nazwom klas, które zawierają. Znalezienie interesującego nas pliku może być uciążliwe. Wygenerowane klasy dziedziczą po *WebViewPage<T>*, gdzie *T* jest to typ modelu podpięty do widoku. Nazwa klasy nawiązuje do ścieżki pliku [11]. Listing 2 przedstawia skompilowany widok Index.cshtml wyświetlany na głównej (Home) stronie aplikacji.

Listing 2 Wygenerowana klasa widoku Home/Index. Źródło: Opracowanie własne.

```
#pragma checksum "C:\Users\Johny\Personal\Uczelnia\Praca
Magisterska\projekt\code\PJWSTK.MGR.SnowShop\PJWSTK.MGR.SnowShop.mvc
\Views\Home\Index.cshtml" "{ff1816ec-aa5e-4d10-87f7-6f4963833460}"
"09C7D37DA19F91EDB6F18A0BA1D0D3D0B9F6B25B"

//------------------------------------------------------------------
------------

// <auto-generated>

//     Ten kod został wygenerowany przez narzędzie.

//     Wersja wykonawcza:4.0.30319.34014

//

//     Zmiany w tym pliku mogą spowodować nieprawidłowe zachowanie i
zostaną utracone, jeśli

//     kod zostanie ponownie wygenerowany.
```

```
// </auto-generated>
//------------------------------------------------------------------------
------------

namespace ASP {

    using System;

    using System.Collections.Generic;

    using System.IO;

    using System.Linq;

    using System.Net;

    using System.Web;

    using System.Web.Helpers;

    using System.Web.Security;

    using System.Web.UI;

    using System.Web.WebPages;

    using System.Web.Mvc;

    using System.Web.Mvc.Ajax;

    using System.Web.Mvc.Html;

    using System.Web.Optimization;

    using System.Web.Routing;

    using PJWSTK.MGR.SnowShop.mvc;

    public class _Page_Views_Home_Index_cshtml :
System.Web.Mvc.WebViewPage<dynamic> {

#line hidden

        public _Page_Views_Home_Index_cshtml() {

        }

        protected ASP.global_asax ApplicationInstance {

            get {

                return
((ASP.global_asax)(Context.ApplicationInstance));

            }

        }

        public override void Execute() {

            #line 1 "C:\Users\Johny\Personal\Uczelnia\Praca
Magisterska\projekt\code\PJWSTK.MGR.SnowShop\PJWSTK.MGR.SnowShop.mvc
\Views\Home\Index.cshtml"

    ViewBag.Title = "Home Page";
```

```
            #line default
            #line hidden
BeginContext("~/Views/Home/Index.cshtml", 39, 8, true);
WriteLiteral("\r\n\r\n<div");
EndContext("~/Views/Home/Index.cshtml", 39, 8, true);
BeginContext("~/Views/Home/Index.cshtml", 47, 18, true);
WriteLiteral(" class=\"jumbotron\"");
EndContext("~/Views/Home/Index.cshtml", 47, 18, true);
BeginContext("~/Views/Home/Index.cshtml", 65, 28, true);
WriteLiteral(" style=\"text-align: center;\"");
EndContext("~/Views/Home/Index.cshtml", 65, 28, true);
BeginContext("~/Views/Home/Index.cshtml", 93, 35, true);
WriteLiteral(">\r\n    <h1>SnB Store</h1>\r\n    <img");


EndContext("~/Views/Home/Index.cshtml", 93, 35, true);
WriteAttribute("src", Tuple.Create(" src=\"", 128),
Tuple.Create("\"", 154)
, Tuple.Create(Tuple.Create("", 134), Tuple.Create<System.Object,
System.Int32>(Href("~/Content/banner.jpg")
, 134), false)
);
BeginContext("~/Views/Home/Index.cshtml", 155, 11, true);
WriteLiteral(" />\r\n</div>");
EndContext("~/Views/Home/Index.cshtml", 155, 11, true);
        }
    }
}
```

Dla zobrazowania jak wiele robi za nas silnik Razor listing 3 przedstawia zawartość pliku widoku, który odpowiada wygenerowanej klasie powyżej.

Listing 3 Widok Home/Index. Źródło: Opracowanie własne.

```
@{
    ViewBag.Title = "Home Page";
}
<div class="jumbotron" style="text-align: center;">
    <h1>SnB Store</h1>
```

```
    <img src="~/Content/banner.jpg" />
</div>
```

Zasady używania silnika Razor:

- Bloki kodu zawierają się w @{ … }
- Wyrażenia w pojedynczej (zmienne i funkcje) linii zaczynają się od @
- Deklaracje kodu zakończone są średnikiem – ‘;’
- Zmiennie można deklarować słowem kluczowym var
- Obiekty typu String są ujęte w cudzysłów
- Pliki mają rozszerzenie .cshtml

Podczas używania silnika Razor możemy wykorzystać kilka bardzo przydatnych funkcji:

- Blok *If-Else*
- Czytanie danych wprowadzonych przez użytkownika za pomocą funkcji *Request[]*
- Właściwość *IsPost*, która mówi czy strona została załadowana z powrotem
- Deklarowanie zmiennych
- Operatory wykorzystywane w wyrażeniach
- Konwertery typów
- Pętle *For*, *For Each* oraz *While*
- Koszystanie z obiektu modelu za pomoca wyrażenia *@Model*
- Dostęp do obiektu *DataTime*

Te i wiele innych funkcji pozwalają na tworzenie bardzo zaawansowanych widoków z dynamicznie generowaną treścią. Konstrukcje warunkowe (*If-Else*) umożliwiają dostosowywanie treści do wartości danych otrzymanych z kontrolera. Na dodatek składnia języka Razor jest prosta do przyswojenia.

2.5. ASP.NET WebForms

ASP.NET Web Forms jest częścią .NET Framework i służy do tworzenia aplikacji internetowych. Web Forms są to strony, które za pomocą przeglądarki są prezentowane użytkownikowi. Strony te mogą być napisane z wykorzystaniem kodu HTML, skryptów po stronie klienta, kontrolek oraz kodu po stronie serwerowej. W momencie żądania przez użytkownika strony, jest ona kompilowana i wykonywana przez framework na serwerze. Następnie generowany jest kod HTML, który wyświetla przeglądarka internetowa [12].

2.5.1. Kontrolki

ASP.NET dostarcza zestawu kontrolek, które są obiektami umieszczanymi na stronie. W momencie żądania danej strony, kod kontrolki jest uruchamiany, a wynik wyświetlany jest przez przeglądarkę. Wiele kontrolek przypomina elementy HTML, takie jak przyciski czy pola tekstowe. Inne zawierają bardziej zaawansowaną funkcjonalność jak na przykład kalendarz lub kontrolki łączące się z bazą danych [12].

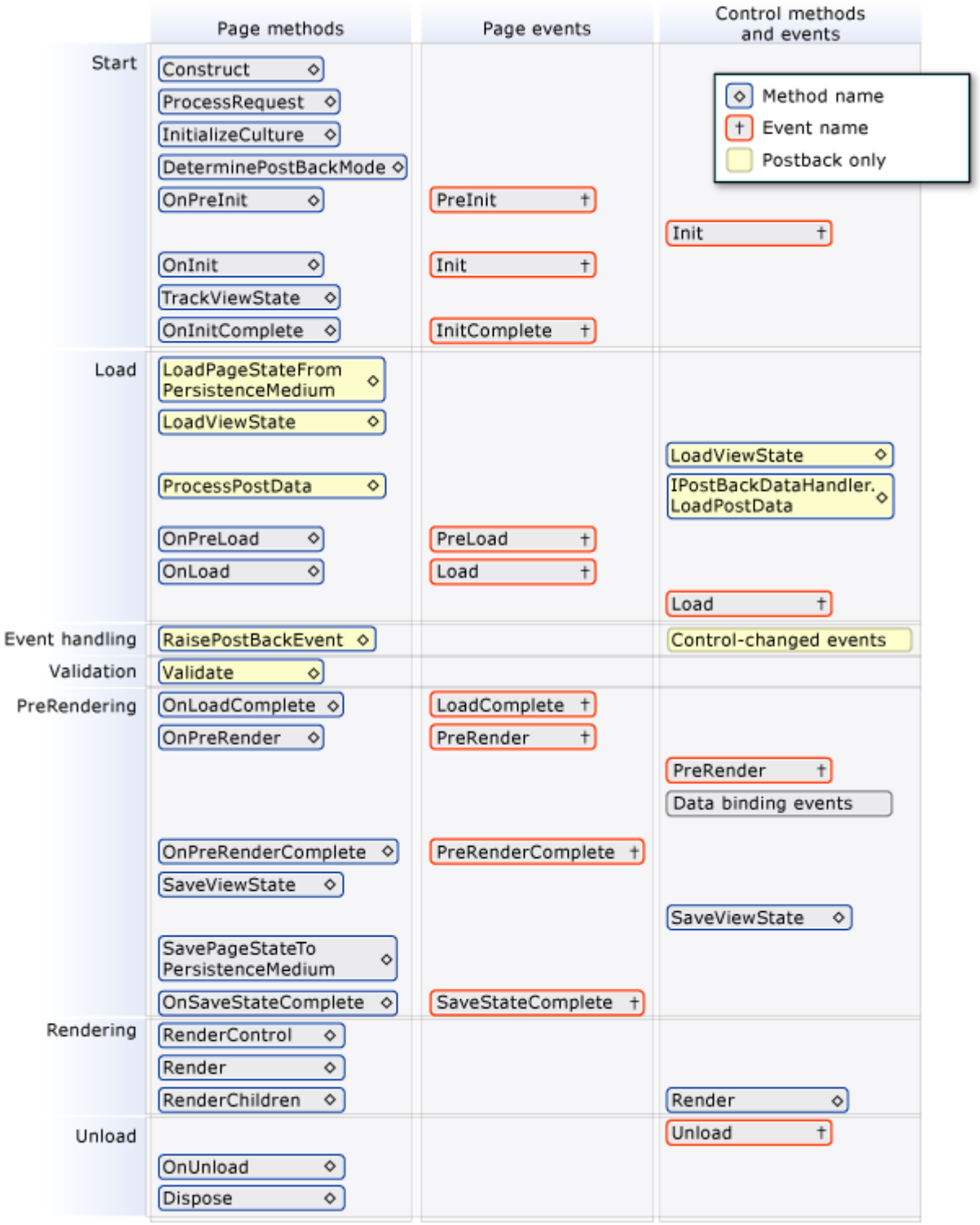
Kontrolki możemy podzielić na kilka kategorii:

- Kontrolki HTML – kontrolki, które odpowiadają elementom języka HTML.
- Kontrolki webowe – kontrolki tworzone i uruchamiane po stronie serwera. Po wykonaniu operacji na serwerze w przeglądarce wyświetlany jest odpowiedni kod HTML.
- Kontrolki walidacyjne – zestaw kontrolek odpowiedzialny za walidacje danych wprowadzonych przez użytkownika. Walidacja ta odbywa się po stronie przeglądarki przed wysłaniem danych do serwera.
- Kontrolki użytkownika – kontrolki stworzone przez programistę z wykorzystaniem pozostałych kontrolek

Tak jak kontrolki użytkownika tak i kontrolki webowe mogą być tworzone przez programistę. Różnica polega na tym, że kontrolki webowe mogą być przenoszone pomiędzy projektami w postaci bibliotek dll.

2.5.2. Cykl życia strony

Kiedy następuje żądanie strony przechodzi ona przez cykl życia, w którym wykonuje się szereg etapów przetwarzania. Należą do nich między innymi inicjalizacja instancji kontrolek, przywrócenie i utrzymanie stanu, uruchomienie kod obsługi zdarzenia oraz renderowanie. Tworząc aplikacje w ASP.NET WebForms bardzo istotne jest zrozumienie cyklu życia strony, aby w pełni świadomie pisać kod w momencie, w którym jest on odpowiedni. Na każdym etapie cyklu życia strona wywołuje zdarzenia, które programista może przechwycić, a następnie uruchomić w nich swój kod [13]. Rysunek 6 przedstawia cykl życia strony oraz metody, jakie są dostępne.



Rysunek 6 Cykl życia strony.
Źródło: [14]

2.5.3. Stan

Protokół HTTP jest protokołem bezstanowym. Kiedy klient odłączy się od serwera silnik ASP.NET wyrzuca wszystkie obiekty z pamięci. Wobec tego potrzebny jest mechanizm, który przechowuje informacje pomiędzy żądaniami. Odpowiedzialny jest za to tak zwany stan, który możemy podzielić na:

- View State – jest to stan strony i wszystkich jej kontrolek. Jest utrzymywany automatycznie w całej aplikacji. Wszystkie zmiany dokonane we właściwościach strony przez użytkownika są zapisywane do ukrytego pola _VIEWSTATE, a następnie przesyłane do serwera za pomocą żądania HTTP.
- Control State – został zaprojektowany, aby przechowywać ważne dane kontrolek, które muszą być dostępne po odświeżeniu strony.
- Session State – tworzony jest w momencie, kiedy użytkownik uruchomi stronę ASP.NET. Pozwala na przechowywanie informacji pomiędzy żądaniami. Jest tworzony indywidualnie dla każdego użytkownika.
- Application State – w momencie pierwszego uruchomienia aplikacji tworzony jest stan, który jest wspólny dla wszystkich użytkowników. Wykorzystywany jest do przechowywania informacji podczas całego okresu działania aplikacji [15].

2.6. AngularJS

AngularJS jest frameworkiem napisanym na potrzeby tworzenia aplikacji internetowych. Został skonstruowany w języku JavaScript, aby rozszerzyć możliwości HTML. Cała funkcjonalność wykonuje się po stronie użytkownika. Angular powstał w 2009 roku w wyniku pracy programistów firmy Google. Podstawowym celem przyświecającym twórcom było wdrożenie wzorca MVC (Model-View-Controller). Angular pozwala tworzyć zaawansowane aplikacje internetowe przy znajomości tylko i wyłącznie JavaScript i HTML [16].

Jako zalety frameworka możemy wymienić:

- AngularJS jest biblioteką *Single Page Application* (SPA)
- Użycie AngularJS pozwala na osiągnięcie zadania mniejszym kosztem (pod ilością linii kody) niż rozwiązanie korzystające z czystego JavaScript lub biblioteki jQuery.
- Z uwagi na to, że logika biznesowa jest odseparowana, aplikacje stworzone przy pomocy AngularJS są łatwiejsze do ostylowania za pomocą CSS.
- Niezależność od back-end'u
- Nie ma potrzeby stosowania AngularJS w kontekście całej aplikacji [17]

2.6.1. Unit testing

Bardzo istotną kwestią w świecie Angular są testy jednostkowe. Z uwagi na fakt, że JavaScript nie jest językiem typowanym, nie posiada własnego kompilatora wymusza to na programistach odpowiedzialność za pisanie kodu działającego zgodnie z jego przeznaczeniem. To właśnie testy jednostkowe stanowią sposób na tworzenie kodu takiego, jakiego oczekujemy, a ich wykonywanie w sposób automatyczny sprawia, że nie musimy się martwić o nieoczekiwaną zmianę zachowania.

Testy jednostkowe jest to koncepcja, która testuje pojedynczą funkcję lub fragment kodu, aby zapewnić, że działa on zgodnie z przeznaczeniem. Jest to bardzo powszechne podejście, dla kodu wykonującego się po stronie serwera. Oto trzy powody, dla których powinniśmy pisać testy jednostkowe, kiedy pracujemy z językiem JavaScript:

1. Dowód poprawności
   Testy jednostkowe stanowią dowód, że nasza funkcja działa poprawnie. Sprawdzają, czy to, co stworzyliśmy naprawdę robi to, co powinno robić. Pozwalają sprawdzić wszystkie przypadki brzegowe.
2. Brak kompilatora
   W JavaScript nie występuje kompilator, który powie nam, że popełniliśmy semantyczny błąd. To przeglądarka internetowa poinformuje nas czy coś jest zepsute. Musimy pamiętać, że każda przeglądarka może generować własne błędy. Testy jednostkowe mogą być uruchamiane zanim aplikacja zostanie wygenerowana przez przeglądarkę, a tym samym ostrzec nas przed ewentualnymi błędami.
3. Wcześniejsze wyłapywanie błędów

Bez testów jednostkowych o ewentualnych błędach dowiemy się, dopiero w momencie odświeżenia strony w przeglądarce. Testy pomagają wyłapać błędy wcześniej, a tym samym zwiększy szybkość programowania.

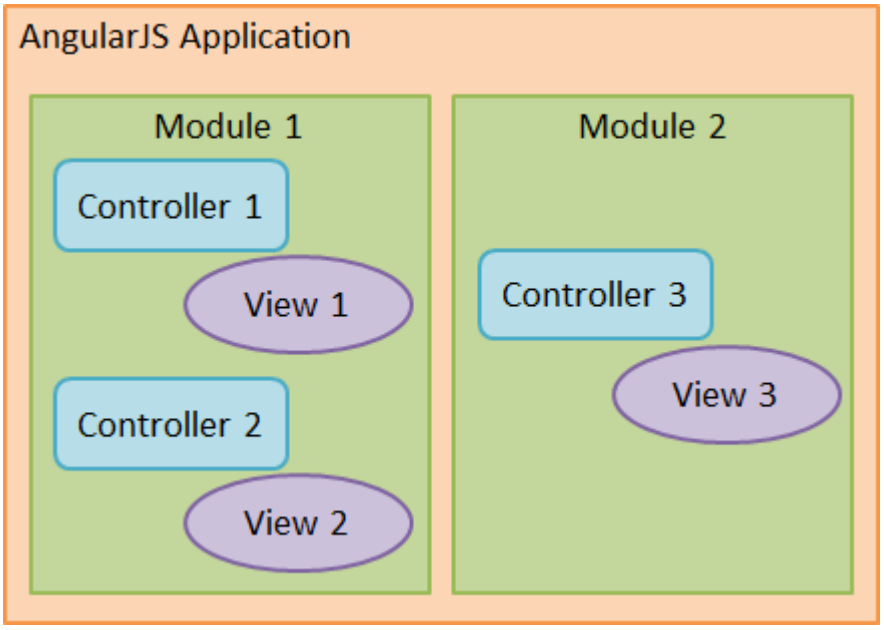
2.6.2. Moduły

Moduły służą do opakowywania kodu aplikacji pod pojedynczą nazwą. Jest to swego rodzaju podobieństwo do *namespace* w C# oraz *packages* w Java. Moduły zawierają definicję przynależących do nich kontrolerów, usług, fabryk oraz dyrektyw, które następnie mogą być używane w obrębie aplikacji. Aby zadeklarować moduł należy podać jego nazwę oraz tablicę komponentów, od których danym moduł jest zależny:

Listing 4 Deklaracja modułu w AngularJS. Źródło: Opracowanie własne.

```
var app = angular.module("TestModule", []);
```

Jak pokazano na Rysunek 2, każdy moduł może zawierać kilka kontrolerów oraz widoków zarządzanych przez odpowiednie kontrolery. Wszystkie moduły wraz z ich elementami składają się na aplikację AngularJS.



Rysunek 7 Podział aplikacji Angular na moduły, kontrolery i widoki.
Źródło: [18]

2.6.3. Kontrolery

Kontrolery w AngularJS są funkcjami JavaScript, które są odpowiedzialne za kontrolowanie przepływu informacji w aplikacji. Za ich działanie odpowiada silnik AngularJS – oznacza to tyle, że funkcje te, nie są jawnie uruchamiane. Podstawowe zadania, za które odpowiedzialny jest kontroler to:

- Pobieranie odpowiednich danych z serwera
- Decydowanie, które dane pokazać użytkownikowi
- Zarządzanie logiką prezentacyjną
- Zarządzanie interakcjami

Kontrolery zawsze są związane z widokiem HTML, który przedstawia dane z powiązanym modelem. Służą do zarządzania pewnym obszarem HTML, w obrębie, którego zostały wczytane. Definicja kontrolera wygląda w sposób następujący:

Listing 5 Deklaracja kontrolera w AngularJS. Źródło: Opracowanie własne.

```
var TestController = function ($scope) {
      $scope.testText = „Test”;
}
```

Tworząc kontroler, do funkcji musimy dodać parametr $scope, który będzie obiektem-kontenerem, do którego możemy przypisać zmienne. Przypisane rzeczy do obiektu $scope staną się modelem, który możemy następnie wykorzystać w widoku.

2.6.4. Dyrektywy

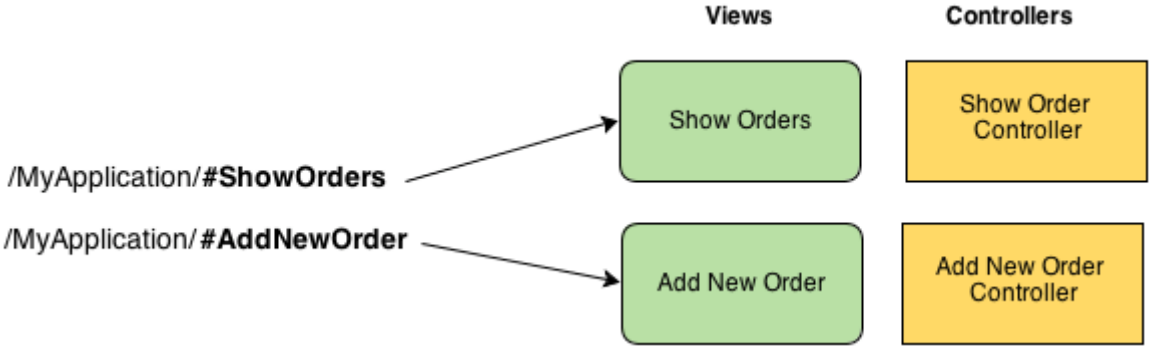
Dyrektywy w Angular są rozszerzeniami atrybutów HTML z prefiksem ng-. Korzystamy z nich w widoku, aby na przykład przypisać kontroler do pewnego obszaru HTML, który następnie będzie sterowany przez ten kontroler [19].

Wyrażenie {{ }} jest tak zwanym *binding expression*, które stanowi specjalną dyrektywę ngBind służącą do wyświetlania danych otrzymanych przez kontroler w postaci modelu. Możemy wyróżnić kilka ważniejszych dyrektyw:

- ng-app – główny element aplikacji Angular
- ng-init – definiuje wartości startowe dla aplikacji Angular
- ng-model – wiąże wartości kontrolek HTML z danymi aplikacji
- ng-repeat – klonuje elementy HTML dla każdego elementu w kolekcji

2.6.5. Routing

Z uwagi na fakt, że AngularJS korzysta z wzorca MVC nie mogło zabraknąć w nim mechanizmu routingu. Funkcjonalność ta, pozwala parsować adres podany w przeglądarce a następnie, na jego podstawie wyświetlić odpowiednią treść. Treść jest zarządzana przez konkretny kontroler. Schemat takiego działania został pokazany na Rysunek 5.



Rysunek 8 Schemat routingu w Angular.
Źródło: [20]

2.6.6. $http

Większość aplikacji opiera się na danych zewnętrznych. Angular do pobierania danych z serwera wykorzystuje wyrażenie $http. Służy ono do komunikacji z serwerem przez obiekt JavaScript XMLHttpRequest lub poprzez obiekty w formacie JSON. Usługa $http jest niejako funkcją JavaScript, która jako parametr przyjmuje obiekt konfiguracyjny służący do wygenerowania zapytania HTTP [21].

Angular posiada funkcjonalność pozwalającą na wywołanie poniższych metod:

- GET - $http.get
- POST - $http.post
- PUT - $http.put
- DELETE - $http.delete

2.6.7. Użyteczność AngularJS

jQuery jest domyślą biblioteką dodawaną do projektu w momencie jego tworzenia. Od wersji Visual Studio 2010 programiści używali głównie tego narzędzia do implementacji skryptów JavaScript po stronie warstwy prezentacji. Jednak wiele rzeczy jesteśmy w stanie osiągnąć w znacznie prostszy sposób wykorzystując do tego AngularJS.

Moc AngularJS można przedstawić w bardzo prostym przykładzie. Poniższe fragmenty kodu będą odpowiedzialne za wyświetlanie napisu, w czasie rzeczywistym, który został wprowadzony do pola tekstowego. Aby osiągnąć taki efekt za pomocą popularnej biblioteki JavaScript jQuery należy nadać odpowiednim elementom na stronie identyfikatory, następnie podpiąć do nich odpowiednie funkcje i wykonać interesującą nas operację, w naszym przypadku przypisanie wartości elementu H2 pobranej z pola tekstowego:

Listing 6 AngularJS kontra jQuery. Przykład jQuery. Źródło: Opracowanie własne.

```
<html>
<head>
    <script src="jquery.js"></script>
    <script type="text/javascript">
        $(function() {
            var nazwa = $('#nazwa');
            var powitanie = $('#powitanie');

            nazwa.keyup(function () {
                powitanie.text('Witaj ' + nazwa.val());
            });
        })
    </script>
</head>
<body>
    <input id="nazwa" type="text">
    <h2 id="powitanie"></h2>
</body>
</html>
```

Natomiast za pomocą Angular wystarczy skorzystać z trzech dyrektyw:
- ng-app
- ng-model

- ng-bind, które reprezentuje wyrażenie {{ }}

Listing 7 AngularJS kontra jQuery. Przykład AngularJS. Źródło: Opracowanie własne.

```
<html ng-app>
    <head>
        <script src="angular.js"></script>
    </head>
    <body>
        <input type="text" ng-model="nazwa">
        <h2>Witaj {{ nazwa }}</h2>
    </body>
</html>
```

Jak widać, kod Angular nie tylko zajmuje mniej miejsca, co również jest bardziej intuicyjny. Sprawia to, że pisanie aplikacji jest szybsze, łatwiejsze w utrzymania jak również ma ogromny wpływ na wydajność.

2.6.8. TypeScript

TypeScript został opracowany przez firmę Microsoft w roku 2012. Język ten, miał przenieść JavaScript w świat obiektowości. Dostarcza mechanizmów do tworzenia między innymi klas, interfejsów, czy typowanych zmiennych. Jako, że TypeScript stanowi nadzbiór języka JavaScript wszystkie skrypty mogą być wykonywane przez przeglądarki internetowe [22].

Nowa wersja AngularJS 2.0 będzie powstawać właśnie na bazie TypeScript, co sprawi, że biblioteka stanie się jeszcze bardziej atrakcyjna. Pozwoli na tworzenie znacznie czytelniejszego kodu, co będzie skutkowało zmniejszeniem kosztów utrzymania istniejących rozwiązań. Sama implementacja kodu będzie łatwiejsza i przyjemniejsza, zwłaszcza dla programistów zaznajomionych z obiektowymi językami programowania [23].

3. Projekt przykładowych aplikacji

Powodem powstania aplikacji było zaprezentowanie różnych podejść w procesie implementacji systemu. System informatyczny jest prostym sklepem internetowym stworzonym w technologii webowej. Asortymentem sklepu internetowego są deski snowboardowe. Do stworzenia aplikacji została wykorzystana biblioteka Bootstrap. Aby w pewnym stopniu oddać relację klient-wykonawca system został stworzony na podstawie przygotowanej już wcześniej bazy danych z odpowiednimi tabelami.

3.1. Funkcjonalność systemu

Podstawową funkcjonalnością systemu jest możliwość dokonywania zakupów w sklepie internetowych z asortymentem desek snowboardowych.

1. System ma przechowywać informacje o klientach:
    a. Imię
    b. Nazwisko
    c. Adres
    d. Telefon
    e. Email (login)
    f. Hasło
2. System ma przechować informacje o producentach:
    a. Nazwa
    b. Data powstania
    c. Opis
3. System ma przechowywać informacje o produktach:
    a. Producent
    b. Model
    c. Nazwa
    d. Rok produkcji
    e. Opis
    f. Rozmiar
    g. Cena
4. System ma przechowywać informacje o zamówieniach:
    a. Dane zamówionych produktów
    b. Ilość zamówionych produktów
    c. Całkowita cena zamówienia
    d. Dane kupującego (w tym adres wysyłki)
    e. Status zamówienia
    f. Data zamówienia

3.2. Wymagania funkcjonalne

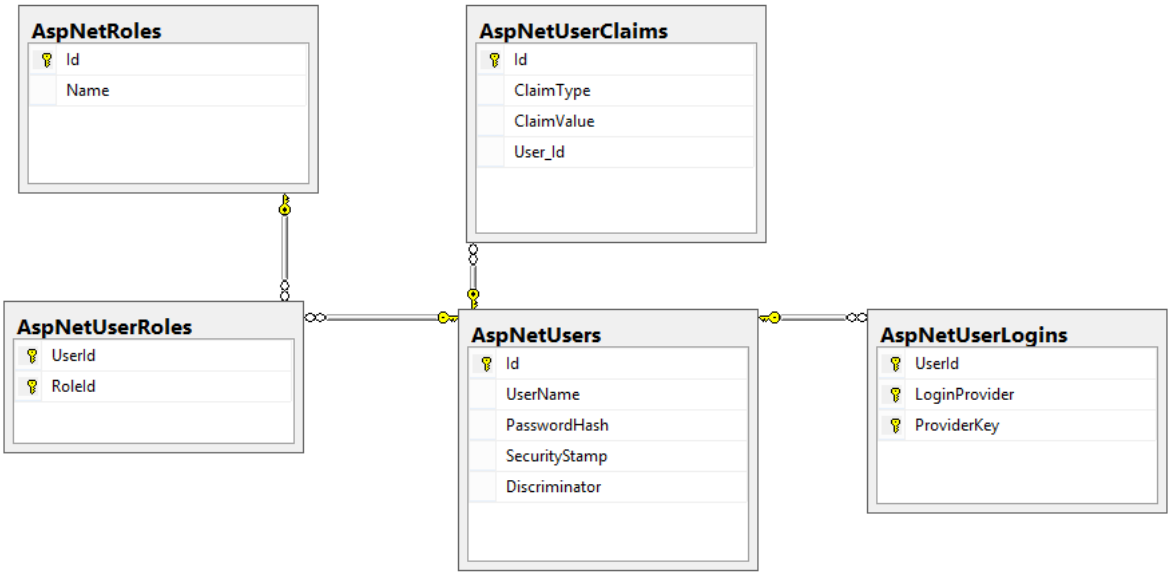
Poniższa tabela przedstawia wymagania funkcjonalne stawiane przed systemem.

L.p.	Nazwa wymagania	Opis wymagania	Ograniczenia	Aktorzy
1.0	CRUD producentów	Administrator serwisu musi mieć możliwość dodawania, usuwania, edycji oraz przeglądania danych producentów	Nie można usuwać producentów, którzy mają „przypisany” produkt	Administrator
2.0	CRUD produktów	Administrator serwisu musi mieć możliwość dodawania, usuwania, edycji oraz	Nie można usuwać produktów, który został wcześniej zamówiony.	Administrator

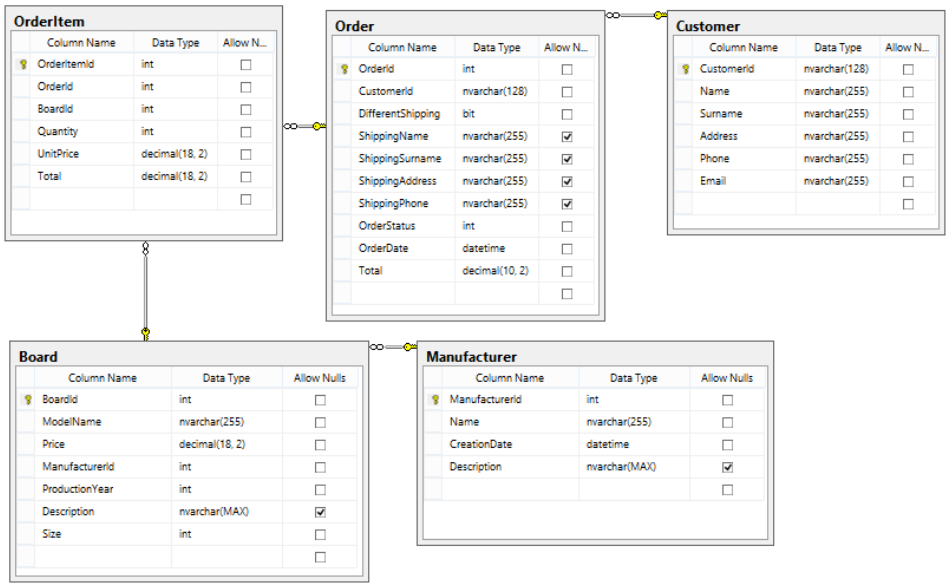
		przeglądania danych produktów.		
3.0	Lista zamówień	Administrator ma możliwość wyświetlenia listy zamówień z możliwością filtrowania oraz sortowania.	Brak	Administrator
4.0	Lista klientów	Administrator ma możliwość wyświetlenia listy klientów	Brak	Administrator
5.0	Kupno produktu	Klient serwisu może dokonać zakupu wybranego produktu	Brak	Klient
5.1	Wyszukiwanie produktu	Klient ma możliwość znalezienia produktu po producencie, rozmiarze oraz zakresie cenowym	Brak	Klient
5.2	Sortowanie listy produktów	Klient może sortować listę produktów po nazwie, cenie rosnącej oraz malejącej	Brak	Klient
5.3	Zmiana ilości w koszyku	Klient może w widoku koszyku zmienić ilość zamówionych produktów	Brak	Klient
6.0	Rejestracja klientów	Zamówienia może dokonać tylko zarejestrowany użytkownik	Podczas próby dokonania zamówienia należy przekierować klienta do strony logowania/rejestracji.	
7.0	Przyjazne linki	Adresy url powinny być w przyjaznej dla użytkownika formie		

3.3. Diagram bazy danych

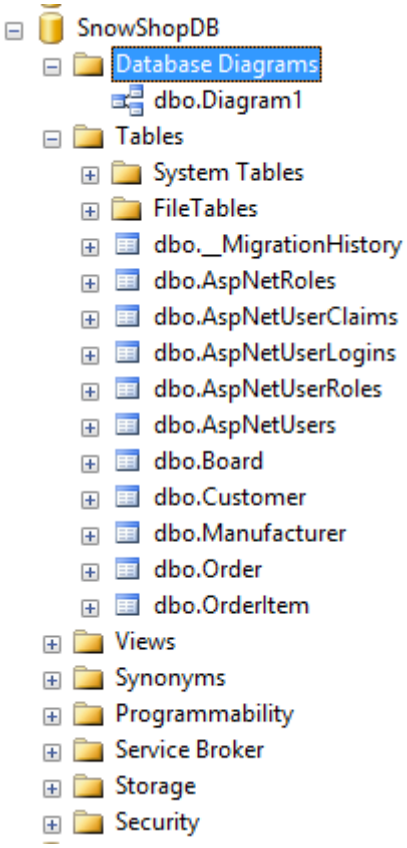
Diagram bazy danych, zaprezentowany na rysunkach 9 i 10, został stworzony w Microsoft SQL Management Studio. Następnie na jego podstawie zostały wygenerowane tabele bazy danych widoczne na Rysunek 11. Przy pomocy skryptu dostarczanego przez firmę Microsoft zostały również utworzone tabele potrzebne do mechanizmu ASP.NET Identity [24].



Rysunek 9 Schemat tabel skryptu ASP.NET Identity w Microsoft SQL Management Studio.
Źródło: Opracowanie własne.

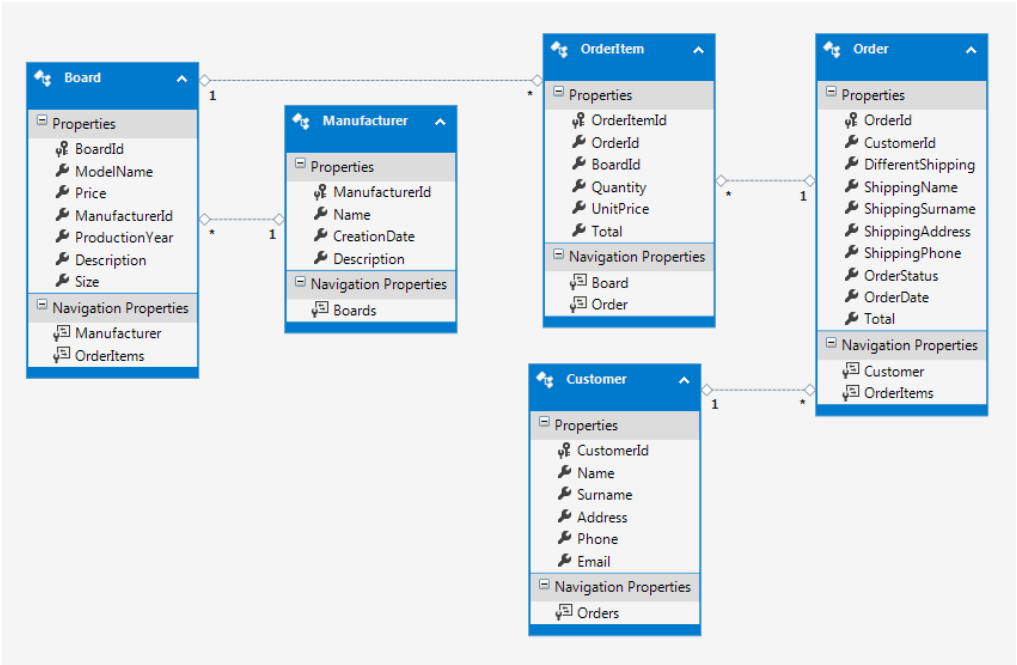


Rysunek 10. Diagram bazy danych w Microsoft SQL Management Studio.
Źródło: Opracowanie własne.



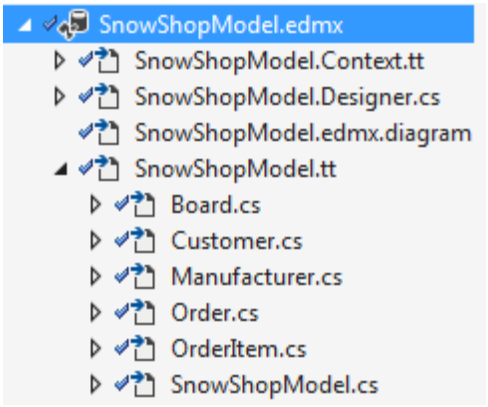
Rysunek 11 Struktura tabel w Microsoft SQL Management Studio.
Źródło: Opracowanie własne.

Kiedy cała struktura bazy danych została już wygenerowana, kolejną czynnością było zaimportowanie odpowiednich tabel do Visual Studio i utworzenie niezbędnych klas reprezentujących odpowiednie tabele (patrz rysunek 12).



Rysunek 12. Diagram bazy danych w Visual Studio 2013.
Źródło: Opracowanie własne.

Poprzez funkcjonalność Entity Framework we wszystkich trzech podejściach, udało się sprawnie wygenerować odpowiednie klasy reprezentujące tabele w bazie danych (patrz rysunek 13).



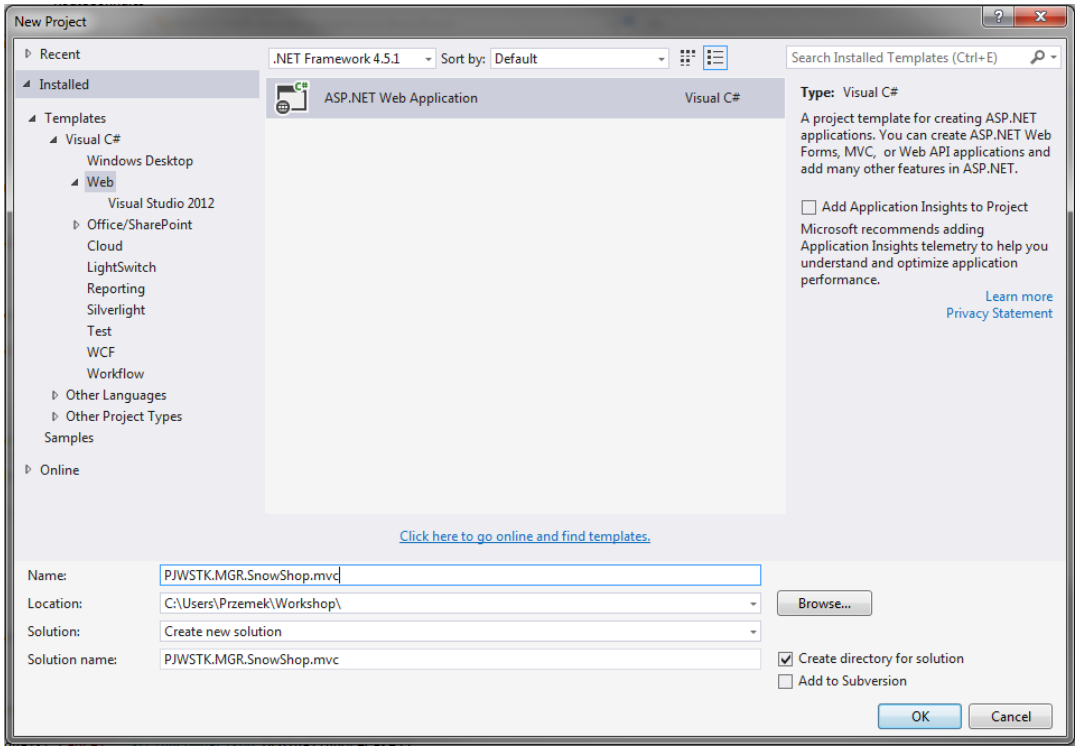
Rysunek 13 Wygenerowane klasy na podstawie bazy danych.
Źródło: Opracowanie własne.

3.4. Implementacja ASP.NET MVC

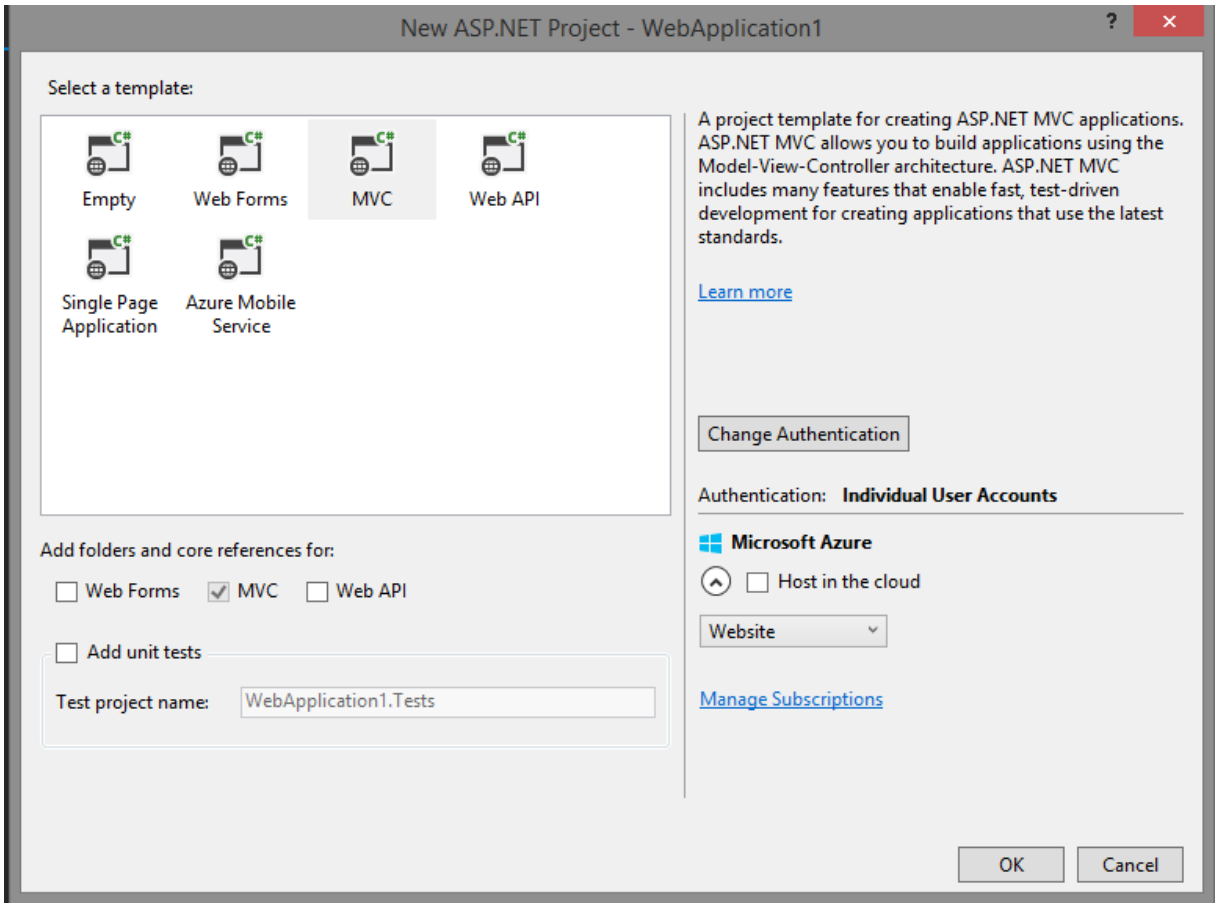
Rozdział ten zawiera opis implementacji prototypu podejściem ASP.NET MVC.

3.4.1. Tworzenie projektu

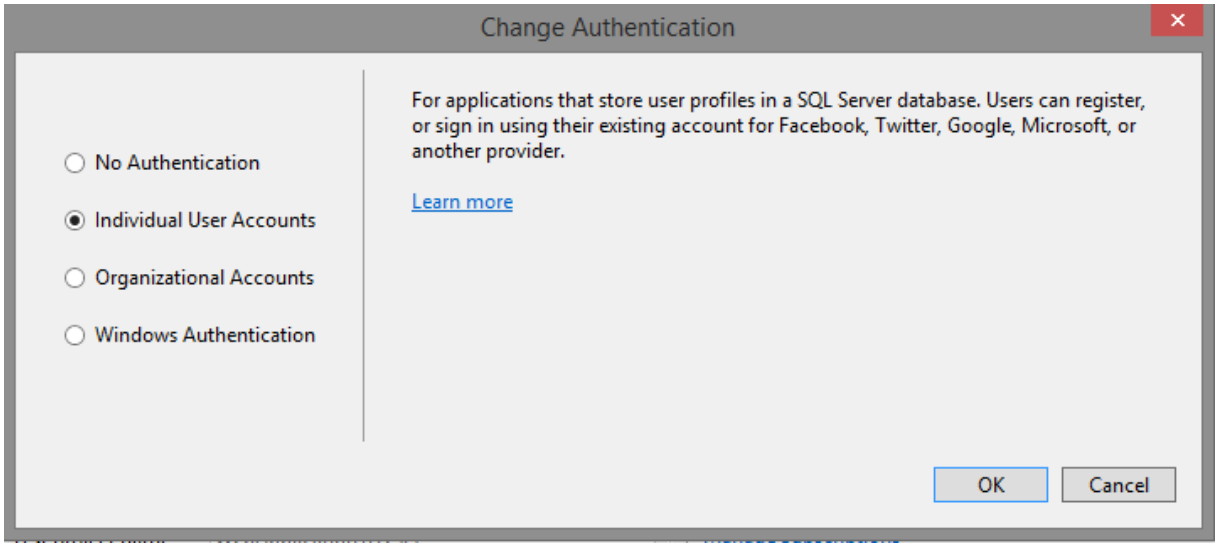
Dzięki wykorzystaniu bardzo zaawanasowanego narzędzia, jakim jest Visual Studio 2013 stworzenie projektu w ASP.NET MVC sprowadziło się do kilku kliknięć myszką. Frameworkiem użytym na potrzeby prototypu był .NET Framework 4.5.1. Program Visual Studio dostarcza szablonu, który idealnie wpasował się w początkową fazę implementacji. Szablon zawierał podłączoną bibliotekę bootstrap, funkcjonalność Entity Framework oraz narzędzie ASP.NET Identity 2. Kolejne kroki przedstawione są na rysunkach 14, 15 oraz 16.



Rysunek 14 Tworzenie projektu w Visual Studio 2013.
Źródło: Opracowanie własne.

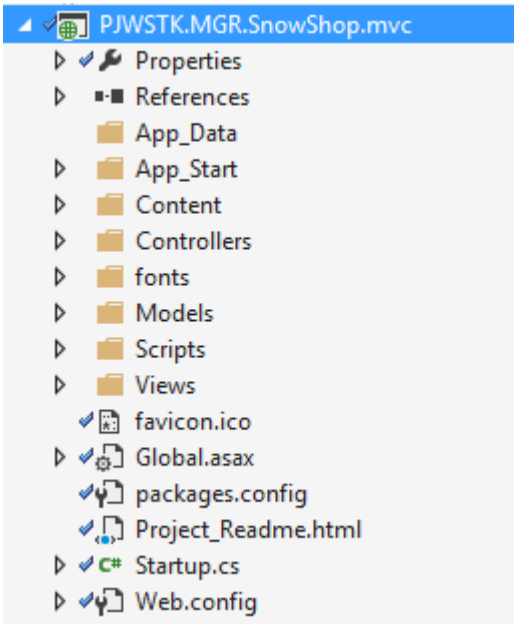


Rysunek 15 Wybór szablonu - ASP.NET MVC.
Źródło: opracowanie własne.



Rysunek 16 Podpięcie narzędzia autoryzacji użytkowników.
Źródło: Opracowanie własne.

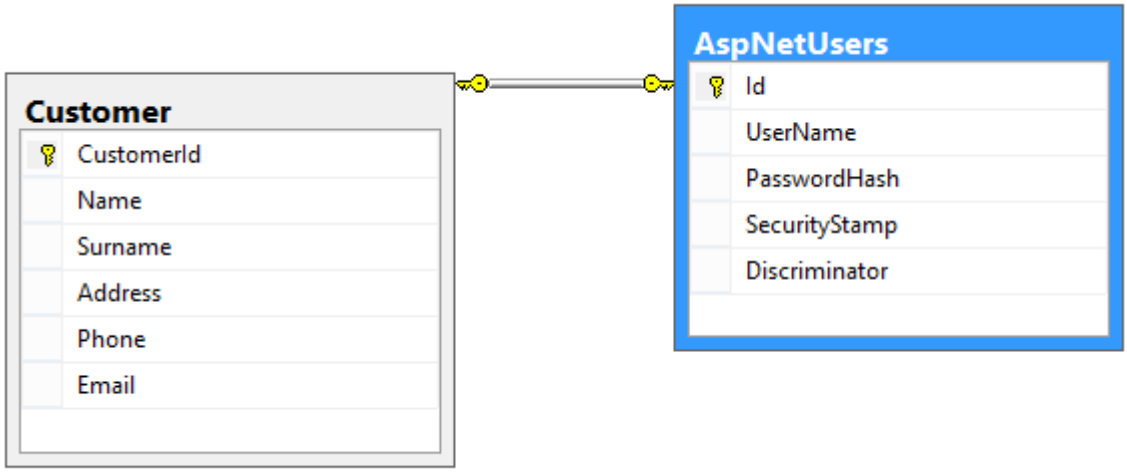
Dzięki powyższym krokom Visual Studio przygotował całą strukturę niezbędnych do działania katalogów (patrz rysunek 17).



Rysunek 17 Schemat katalogów w projekcie – ASP.NET MVC.
Źródło: Opracowanie własne.

3.4.2. Użytkownicy

W prototypie wykorzystano mechanizm ASP.NET Identity 2, jednak został on zmodyfikowany, aby dostosować go do potrzeb aplikacji. Konieczne było powiązanie tabeli przechowującej użytkowników *AspNetUsers* z tabelą gromadzącą informacje o klientach sklepu (*Customer*).



Rysunek 18 Relacja Customer - AspNetUsers.
Źródło: Opracowanie własne.

Powyższa relacja (Rysunek 18) została stworzona na podstawie kolumny *Id* w tabeli *AspNetUsers*, która jest przechowywana po stronie tabeli *Customer*, jako kolumna *CustomerId*. Mając przygotowany model można było przystąpić do dostosowywania funkcjonalności już obecnych w projekcie.

Modyfikacji wymagała klasa *RegisterViewModel*, wykorzystywana podczas procesu rejestracji nowych użytkowników. Należało ją rozszerzyć o właściwości, które następnie byłby przekazywane do tabeli *Customer*.

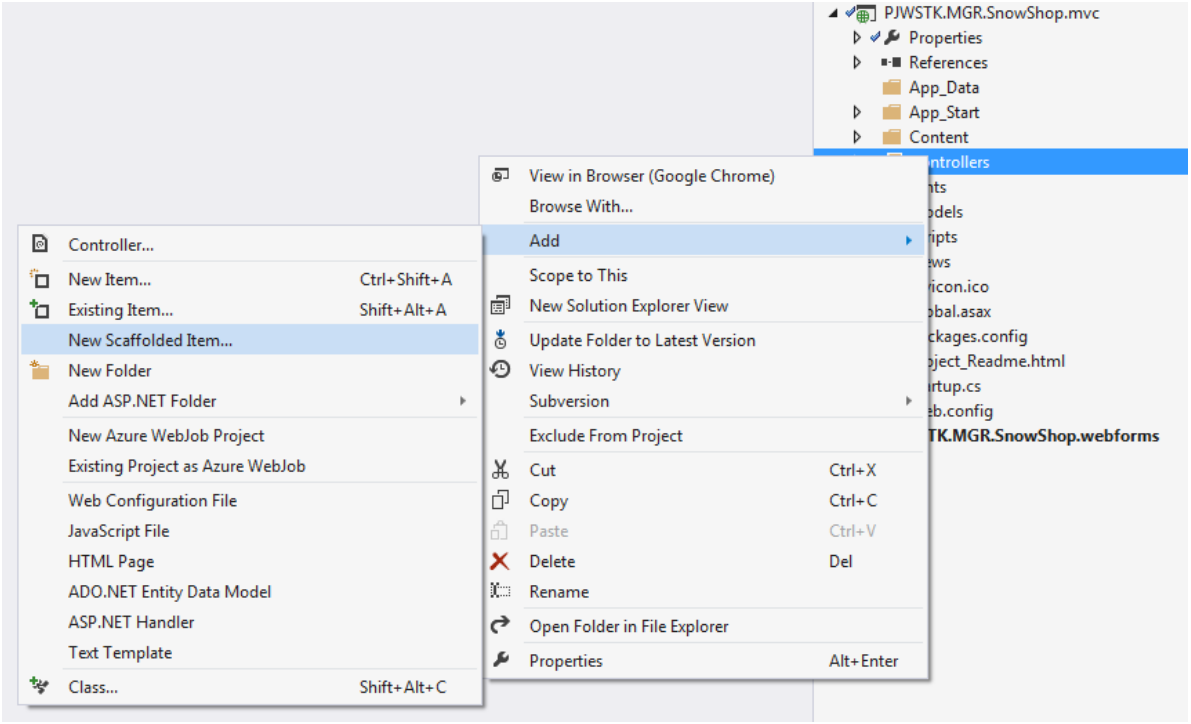
Listing 8 Deklaracja dodatkowych właściwości klasy RegisterViewModel. Źródło: Opracowanie własne.

```
[Required]
[Display(Name = "Name")]
public string Name { get; set; }
[Required]
[Display(Name = "Surname")]
public string Surname { get; set; }
[Required]
[Display(Name = "Address")]
public string Address { get; set; }
[Required]
[Display(Name = "Phone")]
public string Phone { get; set; }
```

Kolejnym etapem było dostosowanie widoku, strony HTML służącej do rejestracji użytkownika. Po tym procesie pozostało już tylko w metodzie tworzącej nowego użytkownika *AspNetUsers* utworzyć i powiązać obiekt *Customer*, a następnie dodać go do bazy danych. Dzięki temu małym kosztem zaimplementowano mechanizm autoryzacji i autentykacji użytkowników wraz z przechowywaniem niezbędnych dla projektu danych klientów.

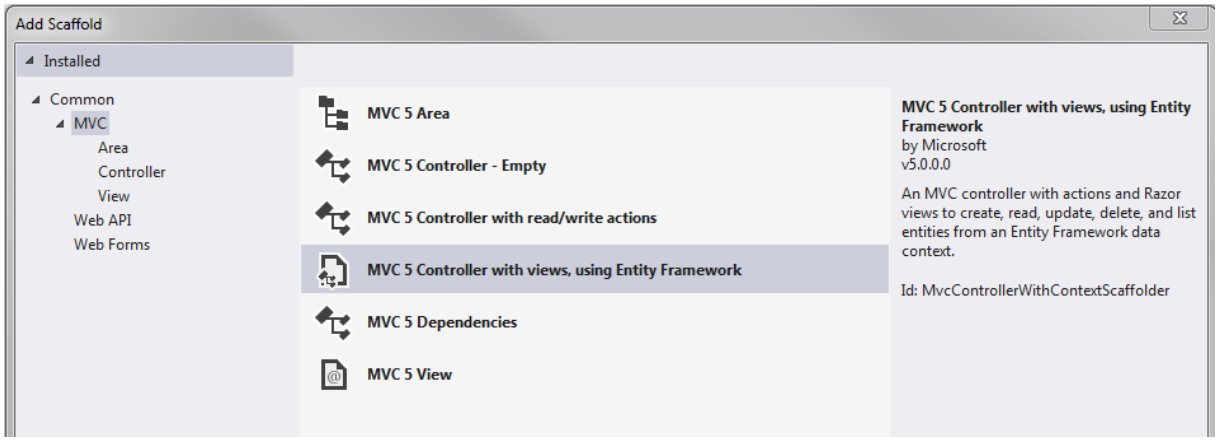
3.4.3. Scaffolding

Visual Studio dostarcza mechanizm *scaffolding*, który jest odpowiedzialny za generowanie kodu dla aplikacji internetowych. Używany jest w momencie, kiedy chcemy szybko dodać do projektu funkcjonalność, która współdziała z modelem. W odniesieniu do pracy narzędzie to zostało wykorzystane w kilku miejscach, między innymi do spełnienia wymagania o zarządzaniu producentami oraz deskami. Dodawanie nowego obiektu *scaffolded* przedstawione jest na Rysunek 19.



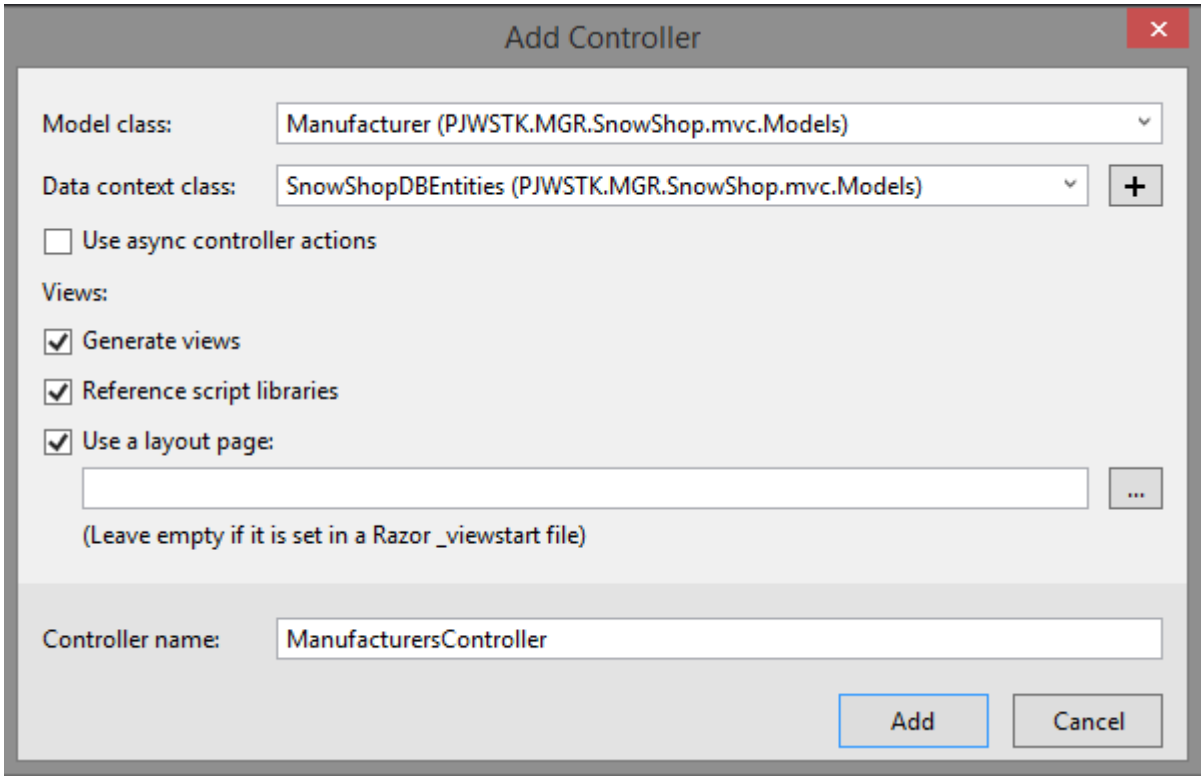
Rysunek 19 Dodawanie nowego obiektu Scaffolded.
Źródło: Opracowanie własne.

W oknie *Add Scaffold* wybieramy typ, jaki chcemy dodać (Rysunek 20).



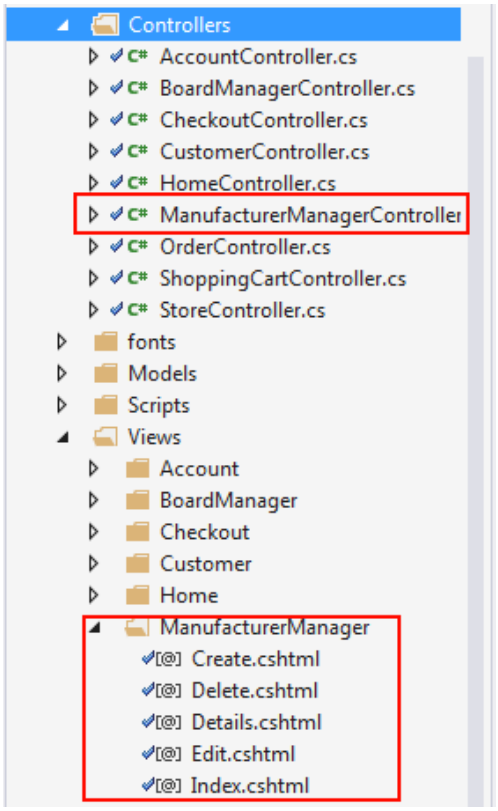
Rysunek 20 Scaffolding - wybór typu.
Źródło: Opracowanie własne.

Kolejny krokiem jest wybranie klasy modelu, dla którego chcemy stworzyć obiekt Scaffold, kontekstu Entity Framework oraz kilku dodatkowych opcji (patrz Rysunek 21).



Rysunek 21 Scaffolding - Model, kontekst, dodatkowe opcje.
Źródło: Opracowanie własne.

Cała procedura trwa kilka sekund, a dostarcza nam kontroler wraz z widokami (patrz Rysunek 22) i podstawową funkcjonalnością operacji CRUD. Spełnia to wymagania stawianie przed prototypem i oszczędza mnóstwo czasu w porównaniu do ręcznej implementacji tych samych funkcji. Programista może skupić się na dostosowywaniu warstwy prezentacyjnej.



Rysunek 22 Scaffolding - wygenerowany kontroler wraz z widokami.
Źródło: Opracowanie własne.

Większość obszarów aplikacji została stworzona za pomocą narzędzia Scaffoldingu, pozwoliło to przyspieszyć proces implementacji. W pracy wystarczyło użycie domyślnych szablonów, jednak podejście MVC pozwala na stworzenie własnych, jak również modyfikację istniejących generatorów widoków oraz kontrolerów [25].

3.4.4. Panel administracyjny

Sklep musiał posiadać panel do zarządzania producentami, deskami oraz wyświetlać wszystkich klientów i zamówienia. Panel ten nie mógł być ogólnie dostępny, potrzebne było zabezpieczenie przed nieuprawnionymi użytkownikami. ASP.NET Identity 2 dostarcza mechanizmów, które pozwalają ograniczyć dostęp do części aplikacji poprzez nadanie użytkownikom odpowiedni ról [26]. Na potrzeby aplikacji powstała jedna rola *Administrator*, która pozwalała na dostęp do panelu administracyjnego. Atrybut *Authorize* dostępny w MVC określa, że dostęp do kontrolera lub jego pojedynczych metod jest ograniczony do użytkowników, którzy spełniają wymóg autoryzacji. Połączenie tych dwóch mechanizmów pozwoliło w prosty sposób osiągnąć zamierzony cel.

Listing 9 Deklaracja atrybutu Authorize w kontrolerze ManufacturerManager. Źródło: Opracowanie własne.

```
[Authorize(Roles = "Administrator")]

public class ManufacturerManagerController : Controller
```

Powyższy fragment kodu przedstawia ograniczenie dostępu do całego kontrolera odpowiedzialnego za zarządzanie producentami, dla użytkowników posiadających rolę *Administrator*. Podejście to zostało wykorzystane w następujących obszarach:

- Lista zamówień (*OrderController*)

- Lista klientów (*CustomerController*)
- Zarządzanie producentami (*ManufacturerManagerController*)
- Zarządzanie deskami (*BoardManagerController*)

Aby ułatwić administratorom aplikacji poruszanie się po stronie, odpowiednie odnośniki zostały dodane do menu. Nie mogły one być widoczne dla zwykłych użytkowników, zatem skorzystano z mechanizmu wyświetlającego zawartość na podstawie roli.

Listing 10 Deklaracja nawigacji - podejście ASP.NET MVC. Źródło: Opracowanie własne.

```
<ul class="nav navbar-nav">
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("Store", "Index", "Store")</li>
@if (ViewContext.HttpContext.User.IsInRole("Administrator"))
{
<li>@Html.ActionLink("ManufacturerAdm",
"Index","ManufacturerManager")</li>
<li>@Html.ActionLink("BoardAdm", "Index", "BoardManager")</li>
<li>@Html.ActionLink("Customers", "Index", "Customer")</li>
<li>@Html.ActionLink("Orders", "Index", "Order")</li>
}
</ul>
```

W powyższym fragmencie kodu cztery odnośniki będą widoczne tylko dla użytkowników należących do roli *Administrator*. Oczywiście istnieje możliwość ręcznego przejścia do tych obszarów, jednak atrybutu *Authorize* zabroni nieuprawnionym użytkownikom dostępu.

3.4.5. Koszyk

Funkcjonalność koszyka została zrealizowana z wykorzystaniem obiektu *Session*, w którym przechowywane są informacje na temat dodanych do koszyka produktów jak również tymczasowy obiekt zamówienia.

Listing 11 Deklaracja klasy ShoppingCart. Źródło: Opracowanie własne.

```
public class ShoppingCart
{
      public CartVM Cart { get; private set; }
      public Order CurrentOrder { get; private set; }
}
```

Obiekt zamówienia oprócz informacji potrzebny do wypełnienia tabeli *Orders* zawiera również informacje o kliencie dokonującym zamówienie oraz listę produktów.

Listing 12 Deklaracja klasy Order. Źródło: Opracowanie własne.

```
public partial class Order
{
        public int OrderId { get; set; }
        public string CustomerId { get; set; }
        public bool DifferentShipping { get; set; }
        public string ShippingName { get; set; }
        public string ShippingSurname { get; set; }
        public string ShippingAddress { get; set; }
        public string ShippingPhone { get; set; }
        public int OrderStatus { get; set; }
        public System.DateTime OrderDate { get; set; }
        public decimal Total { get; set; }
        public virtual Customer Customer { get; set; }
        public virtual ICollection<OrderItem> OrderItems { get; set;
}
}
```

Klasa *CartVM*, służy do przechowywania pozycji w koszyku, całkowitej ilości produktów oraz całkowitej kwoty do zapłaty. Oprócz tego, klasa została wykorzystana do wygenerowania widoku koszyka.

Listing 13 Deklaracja klasy CartVM. Źródło: Opracowanie własne.

```
public class CartVM
{
        public List<CartItemVM> Items { get; set; }
        public int TotalCount
        {
            get
            {
                int res = 0;
                if (this.Items != null && this.Items.Count > 0)
                {
                    res = this.Items.Sum(o => o.Quantity);
                }
                return res;
            }
```

```
        }
        public decimal TotalPrice
        {
            get
            {
                decimal res = 0.0M;
                if (this.Items != null && this.Items.Count > 0)
                {
                    res = this.Items.Sum(o => o.Quantity *
o.UnitPrice);
                }
                return res;
            }
        }
}
```

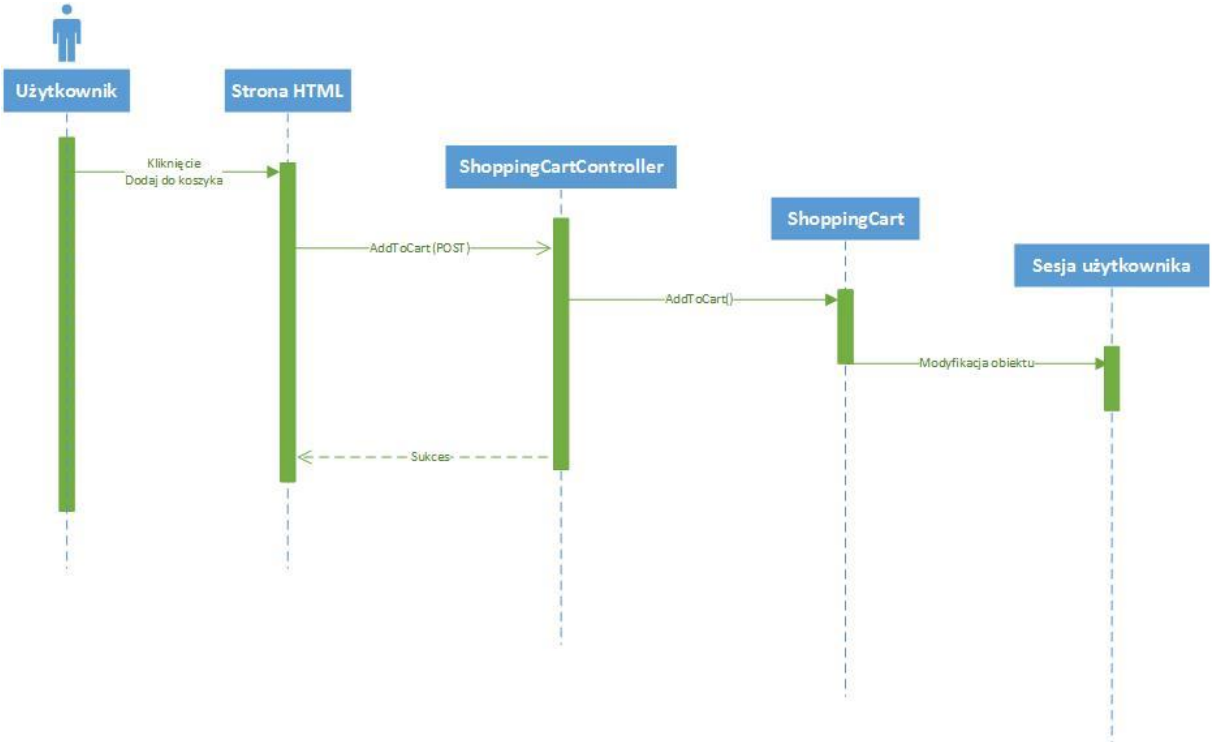
Wszystkie operacje wykonywane na koszyku są wykonywane za pośrednictwem *ShoppingCartController*, który następnie odwołuje się do odpowiednich metod udostępnionych przez klasę *ShoppingCart*. Dostęp do kontrolera wykonywany jest za pomocą zapytania HTTP metodą POST. Poniżej znajduje się przykład, w którym to dokonywane jest dodanie nowego produktu do koszyka po stronie warstwy prezentacji:

Listing 14 Implementacja wywołania metody kontrolera AddToCart. Źródło: Opracowanie własne.

```
$.post("/ShoppingCart/AddToCart", { "boardId": boardToAdd },function
(data) {
      var msgpnl = $('#update-message');
      msgpnl.attr('class', 'alert alert-success');
      msgpnl.html(data.Message);
      msgpnl.show();
});
```

Następnie takie zapytanie jest przechwytywane przez *ShoppingCartController* i wykonywana jest metoda *AddToCart*, w której to następuje modyfikacja danych w sesji, w tym przypadku dodanie nowe produktu, bądź zwiększenie jego ilość, jeśli został dodany już wcześniej.

Rysunek 23 przedstawia diagram sekwencji omawianego procesu dodawania produktu do koszyka.



Rysunek 23 Diagram sekwencji dodawania do koszyka - ASP.NET MVC.
Źródło: Opracowane własne.

3.4.6. PartialView i Layout

PartialView są to specjalne fragmenty widoku przechowywane w osobnym pliku używane w celu wyświetlenia ich wewnątrz innych widoków. Koncept ten powstał, aby umożliwić wielokrotne użycie danej sekcji w wielu miejscach [8]. PartialView został wykorzystany, aby wyświetlić status aktualnego użytkownika. W zależności czy użytkownik jest zalogowany do aplikacji prezentowane są inne informacje, m.in. przycisk zaloguj/wyloguj.

Layout jest to specyficzny widok tworzony na potrzeby zdefiniowania struktury oraz układu strony. Pozwala stworzyć HTML, który będzie obecny wokół treści wszystkich pozostałych stron aplikacji [8]. W prototypie został wykonany tylko jeden Layout służący do wyświetlania panelu nawigacyjnego, stopki strony oraz wczytywania niezbędnych plików styli i skryptów.

Dzięki powyższym mechanizmom kod jest znacznie łatwiejszy w utrzymaniu. W przypadku zmian w panelu nawigacyjnego, programista nie musi dokonywać modyfikować kodu w każdej ze stron, wystarczy, że zrobi to tylko w widoku Layout. Podczas implementacji PartialView oraz Layout odgrywały istotną rolę z punktu widzenia szybkości implementacji. Czas poświęcony na modyfikacje jednego pliku jest znacznie mniejszy niż zmiany dokonywane we wszystkich stronach aplikacji.

3.4.7. Linki

Mechanizm Routingu dostępny w ASP.NET MVC pozwolił w łatwy sposób spełnić wymaganie funkcjonalne „Przyjazne linki”. Umożliwia on korzystanie z adresów URL, które nie muszą wskazywać na fizyczny plik w witrynie. Dzięki temu można użyć linków, które opisują swoje działanie, a tym samym są bardzie zrozumiałe dla użytkownika końcowego. Cała konfiguracja odbywa się w jednym pliku *RouteConfig.cs*. Poniżej znajduje się deklaracja route, dla lokalizacji koszyka. W oryginał formie adres wyglądałby w sposób następujący: http://snowshop.pl/ShoppingCart dzięki mechanizmowi Routingu możliwe było osiągnięcie postaci: http://snowshop.pl/Basket, która jest bardziej czytelna i przekazuje więcej informacji użytkownikowi.

Listing 15 Deklaracja klasy RouteConfig. Źródło: Opracowane własne.

```
public class RouteConfig
{
      public static void RegisterRoutes(RouteCollection routes)
      {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
            name: "Basket",
            url: "Basket",
            defaults: new { controller = "ShoppingCart", action =
"Index"});
      }
}
```

3.4.8. Podsumowanie

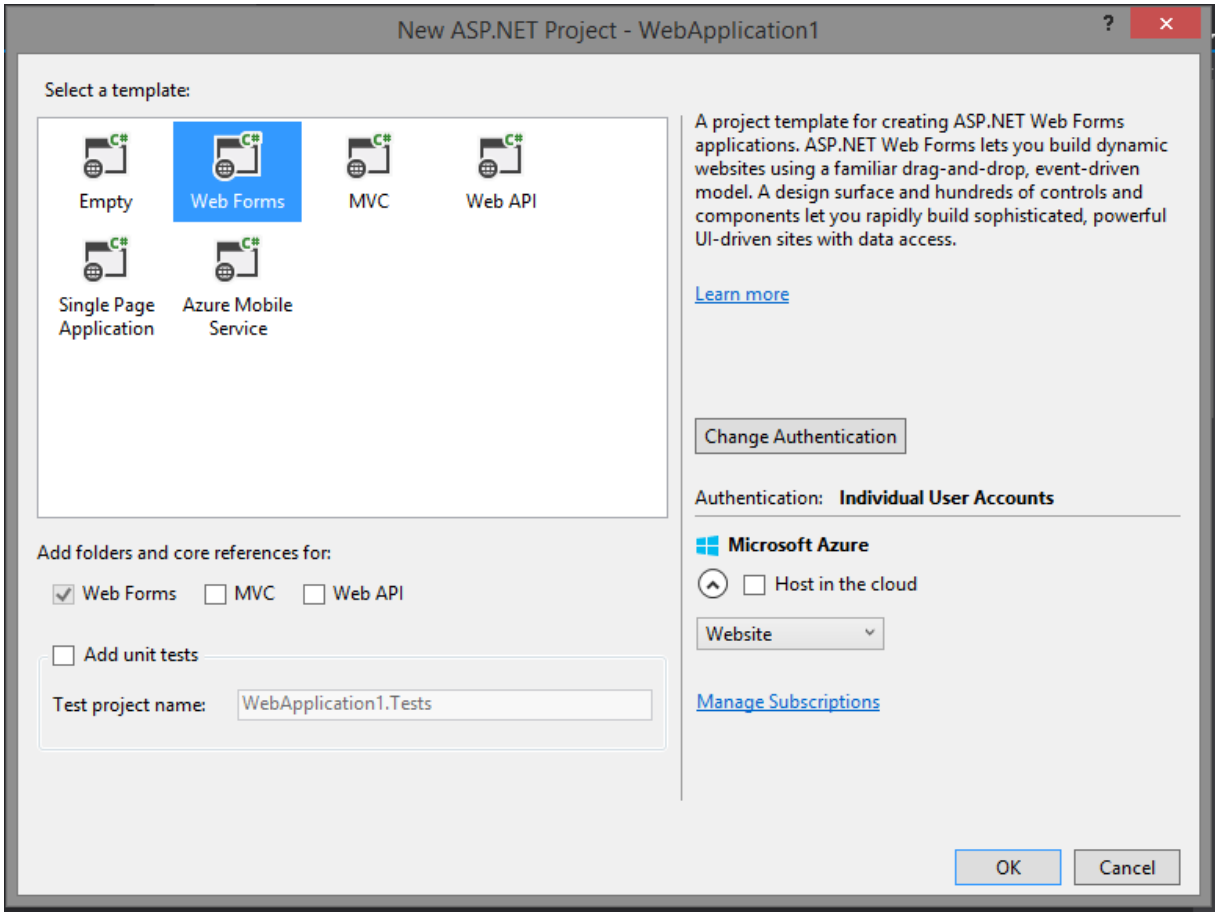
Implementacja w podejściu ASP.NET MVC była niezwykle intuicyjna. Wzorzec projektowy MVC sprawdził się doskonale w przypadku realizowanego projektu. Najważniejszą rzeczą było odpowiednie przygotowanie modelu. To pozwalało na wykorzystaniu mechanizmu Scaffolding, który znacznie przyspiesza pracę i generuje duże ilości kodu za programistę. W przypadku bardziej skomplikowanych projektów istnieje możliwość modyfikacji generatorów, co jeszcze bardziej zwiększa możliwości przyspieszenia implementacji. Wykorzystanie ASP.NET Identity 2 sprawiło, że zarządzanie użytkownikami, autoryzacja i autentykacja wymagała od programisty tylko i wyłącznie odpowiedniej konfiguracji. Cały mechanizm był gotowy i spełniał wymagania. Dodatkowo dzięki wykorzystaniu adnotacji realizacja uprawnień została osiągnięta w bardzo prosty i szybki sposób. Wykorzystanie kontrolera za zarządzania koszykiem również była istotnym plusem. Wszystkie niezbędne funkcje znajdują się w jednym miejscu i wystarczy tylko odwołanie do konkretnego kontrolera. Dzięki PartialView oraz Layout zarządzanie kodem oraz modyfikacja jest bardzo prosta i szybka. Zatem utrzymanie aplikacji lub też dalsza jej rozbudowa jest stosunkowo łatwa i mało problematyczna.

3.5. Implementacja ASP.NET WebForms

Rozdział ten zawiera opis implementacji prototypu podejściem ASP.NET Webforms.

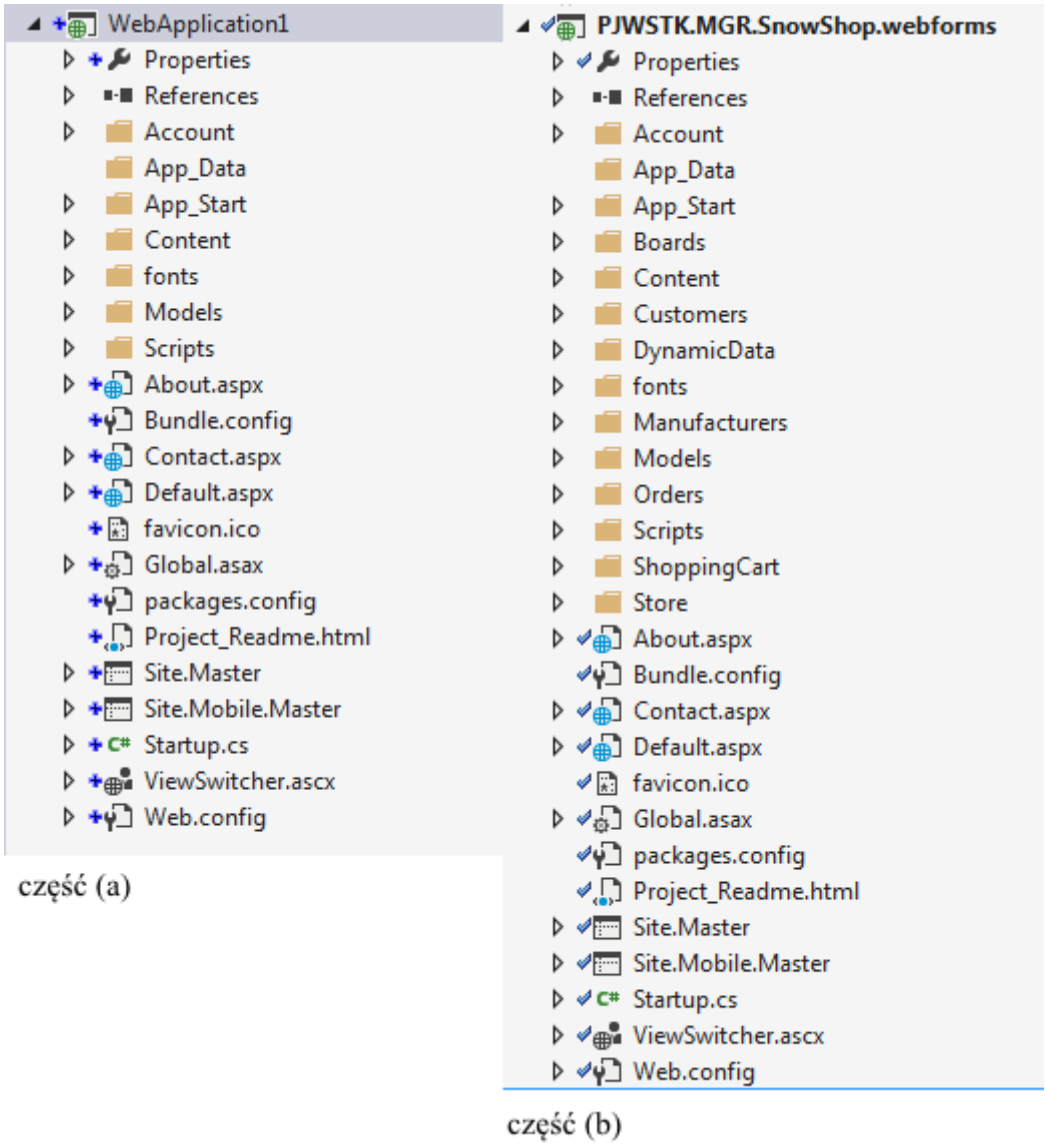
3.5.1. Tworzenie projektu

Do stworzenia projektu w podejściu ASP.NET WebForms również został wykorzystany program Visual Studio 2013 oraz .NET Framework 4.5.1. Różnica w podejściu ASP.NET MVC była tylko w jednym kroku, mianowicie wyboru szablonu widocznym na rysunku 24. Pozostałe elementy takie jak biblioteka bootstrap, funkcjonalność Entity Framework oraz mechanizm ASP.NET Identity 2 były obecne w omawianym szablonie.



Rysunek 24 Wybór szablonu - ASP.NET WebForms.
Źródło: Opracowanie własne.

Schemat katalogów, przed rozpoczęciem implementacji wyglądał w sposób pokazany na rysunku 25 część a. Jednak po zakończeniu implementacji, struktura katalogów nabrała dużo bardziej skomplikowanej formy (patrz rysunek 25 część b).



Rysunek 25 Schemat katalogów w projekcie - ASP.NET WebForms.
Źródło: Opracowanie własne.

3.5.2. Użytkownicy

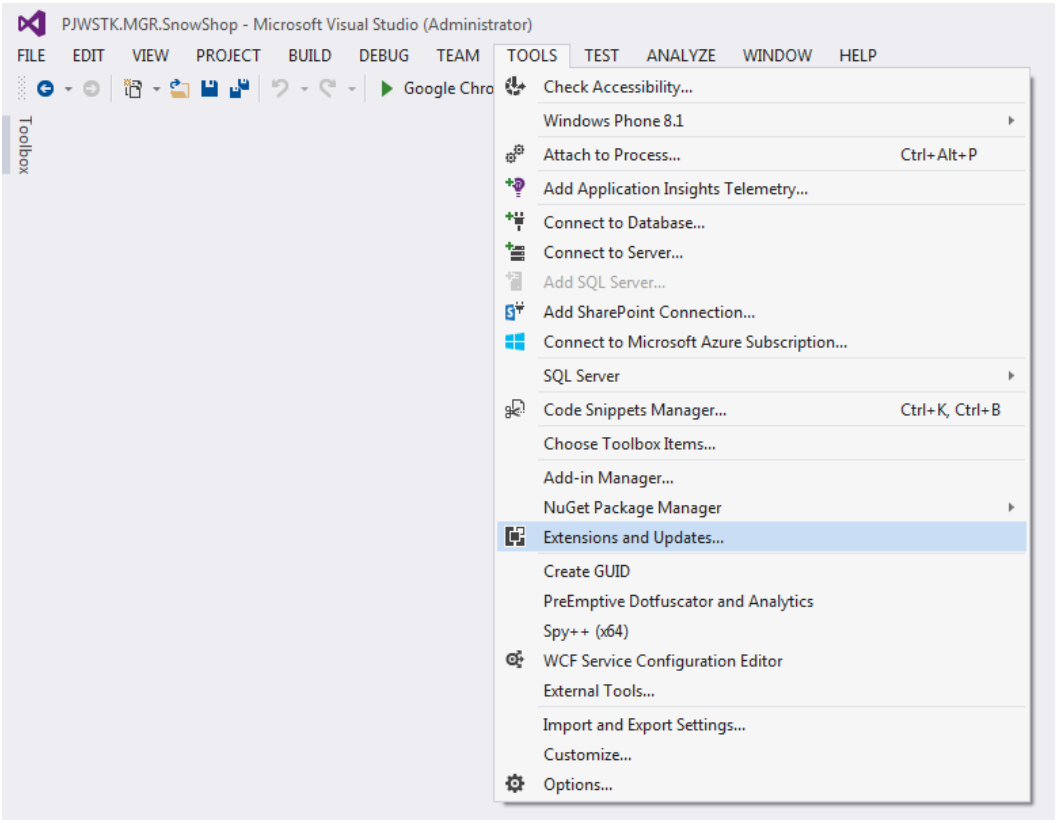
Tak samo jak w podejściu ASP.NET MVC, tak w ASP.NET Webforms zostało wykorzystane narzędzie ASP.NET Identity 2. W tym przypadku należało zmodyfikować stronę rejestracji użytkownika. Z racji tego, że Webforms nie korzysta z modelu, wystarczyło dodać odpowiednie kontrolki odpowiedzialne za przyjęcie danych o imieniu, nazwisku, adresie oraz numerze telefonu, a następnie po stronie code behind wyciągnąć te dane i utworzyć obiekt *Customer*. Koszt czasowy jest porównywalny z podejściem ASP.NET MVC. Jednak w podejściu Webform wszystkie dane trzeba wyciągać ręcznie odwołując się do kontrolek po stronie prezentacyjnej. W MVC natomiast po stronie kontrolera otrzymujemy cały model ze wszystkimi danymi.

3.5.3. Scaffolding

Mechanizm *scaffolding* był jednym z głównych przyczyn bardzo szybkiej implementacji w podejściu ASP.NET MVC. Na pierwszy rzut oka w ASP.NET Webform wszystkie formularze należy tworzyć ręcznie, jednak dzięki narzędziu Web Forms Scaffolding, dostępnym za darmo dodatku do

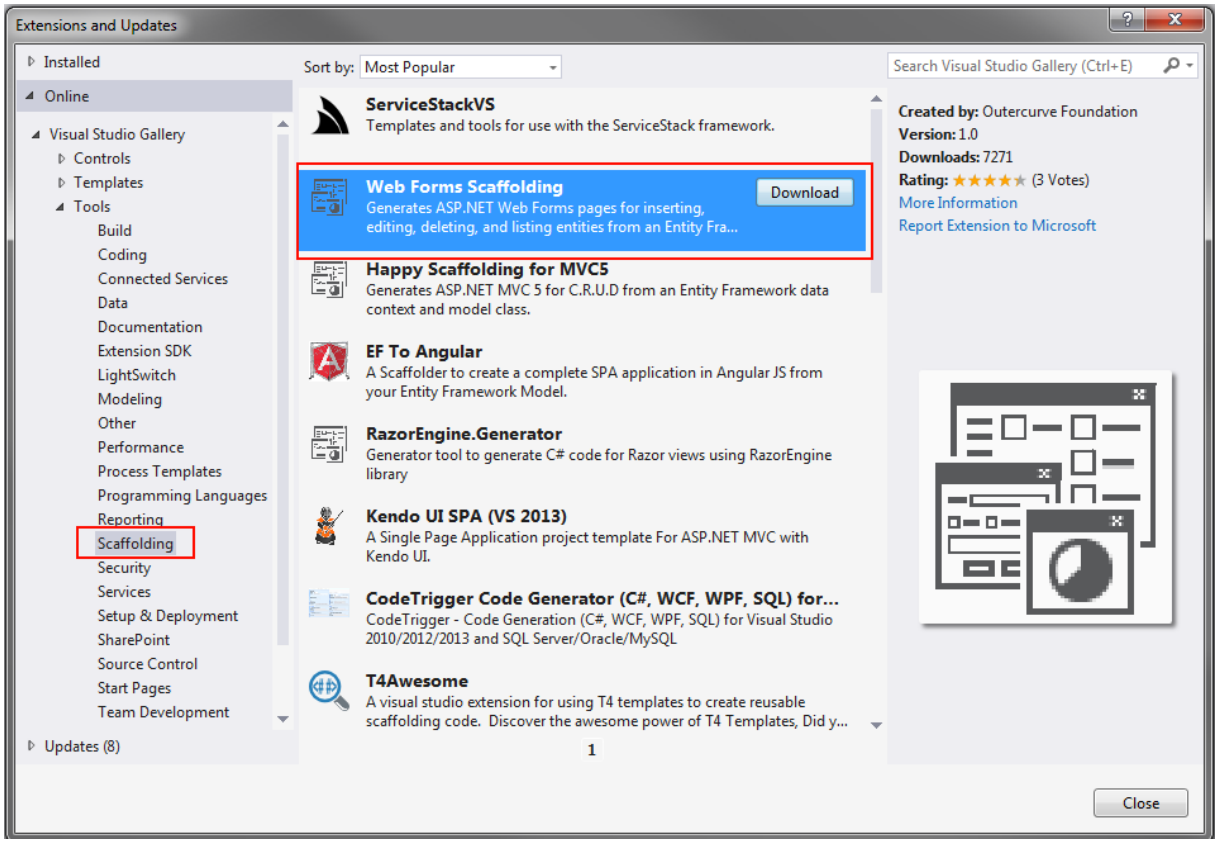
Visual Studio, który generuje strony dla dodawania, edycji, usuwania oraz wylistowania danych z Entity Framework, sprawa wygląda zupełnie inaczej [27].

Aby skorzystać z możliwości Web Forms Scaffolding należy doinstalować dodatek z poziomu Visual Studio przez opcje menu Tools – Extensions and Updates (Rysunek 26):



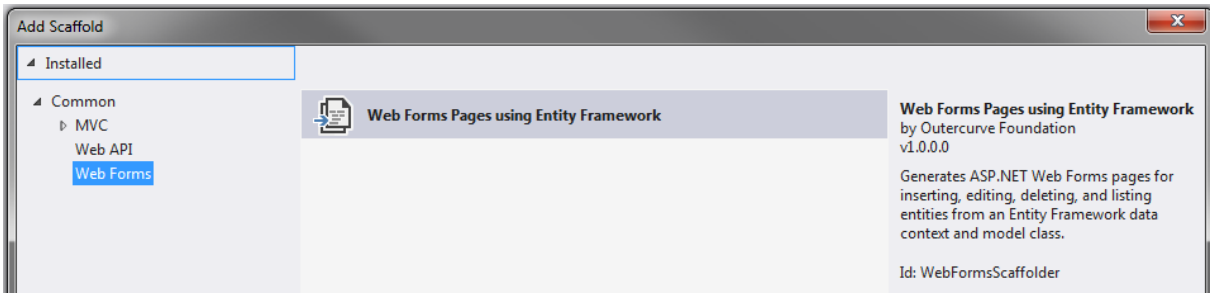
Rysunek 26 Instalowanie Web Forms Scaffolder - krok 1.
Źródło: Opracowanie własne.

Następnie należy wybrać Online – Visual Studio Gallery – Tools – Scaffolding i z wyświetlonej listy wybrać interesujący nas dodatek, czyli Web Forms Scaffolding (patrz Rysunek 27).



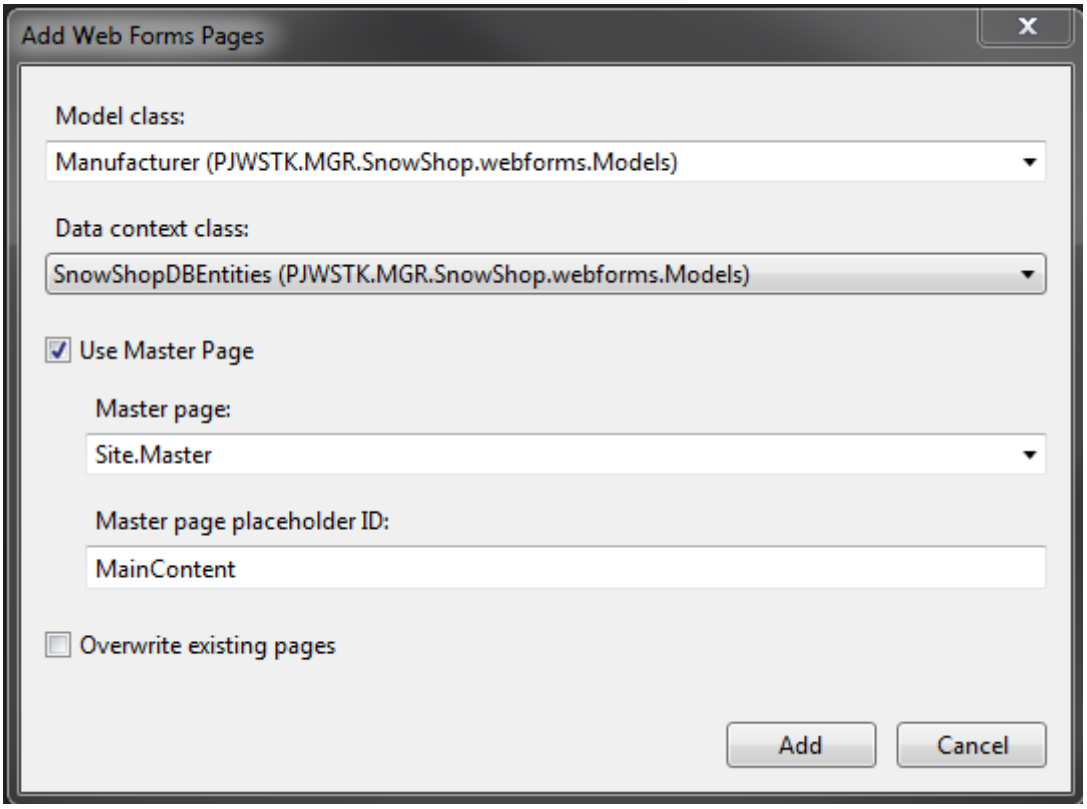
Rysunek 27 Instalowanie Web Forms Scaffolder - krok 2.
Źródło: Opracowanie własne.

Dodawanie nowego obiektu Scaffolded nieznacznie różni się od podejścia ASP.NET MVC, w momencie wyboru typu Scaffoldingu (patrz Rysunek 28).



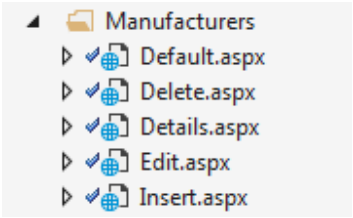
Rysunek 28 Scaffolding - wybór typu Web Forms.
Źródło: Opracowanie własne.

Kolejny krok polega już tylko na wyborze modelu, dla którego chcemy wygenerować odpowiednie strony do dodawania, usuwania, edycji oraz listowania. Podpinamy również kontekst danych oraz wybieramy stronę nadrzędną (patrz Rysunek 29).



Rysunek 29 Scaffolding - Model, kontekst danych oraz strona nadrzędna. Źródło: Opracowanie własne.

Tak jak w przypadku ASP.NET MVC procedura trwa bardzo krótko i generuje wszystkie potrzebne strony, grupując je w jednym folderze widoczne na Rysunek 30.

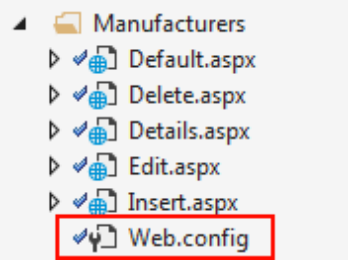


Rysunek 30 Scaffolding - wygenerowane strony dla modelu.Źródło: Opracowanie własne.

W bardzo prosty sposób możemy osiągnąć podobną funkcjonalność, która występuje w ASP.NET MVC, a tym samym zwiększyć szybkość implementowania aplikacji. Jedynym minusem tego rozwiązania jest konieczność instalacji dodatku Web Forms Scaffolding na każdej stacji roboczej, która uczestniczy w procesie implementacji. Opcja ta nie jest domyślna w programie Visual Studio i fakt ten może być nieco problematyczny.

3.5.4. Panel administracyjny

Pomimo tego, że tak jak w przypadku podejścia ASP.NET MVC został wykorzystany mechanizm ASP.NET Identity 2, sprawa ograniczenia dostępu do niektórych obszarów dla użytkowników wcale nie była taka prosta. Jako, że w ASP.NET Webforms nie występują kontrolery nie można było ustawić ograniczenia dostępu poprzez atrybut *Authorize*. Aby osiągnąć zamierzony cel, należało dodać plik konfiguracyjny *web.config* do poszczególnych lokalizacji, zawierających strony, które mogą być wyświetlane tylko przez użytkowników należących do roli *Administrator* (patrz Rysunek 31).



Rysunek 31 Plik konfiguracyjny z ograniczeniem dostępu do lokalizacji manufactures. Źródło: Opracowanie własne.

Listing 16 Definicja pliku web.config - autoryzacja całej lokalizacji. Źródło: Opracowanie własne.

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Administrator"/>
      <deny users="*"/>
    </authorization>
  </system.web>
</configuration>
```

Tak przygotowany plik, umożliwia dostępu do stron znajdujących się w lokalizacji *Manufactures* tylko i wyłącznie użytkownikom posiadającym rolę *Administrator*. Ten sam mechanizm został wykorzystanie w stosunku do lokalizacji *Customers* oraz *Boards*. Sprawa komplikuje się w przypadku *Orders*, które zawiera zarówno stronę dostępną dla wszystkich użytkowników (*Insert.aspx*), jak również taką, która jest ograniczona do roli *Administrator*. W takim przypadku plik konfiguracyjny musiał być nieznacznie zmodyfikowany, aby zabronić dostępu tylko do jednej strony pod tą samą lokalizacją.

Listing 17 Definicja pliku web.config - autoryzacja pojedynczej strony. Źródło: Opracowanie własne.

```
<?xml version="1.0"?>
<configuration>
  <location path="Default.aspx">
    <system.web>
      <authorization>
        <allow roles="Administrator"/>
        <deny users="*"/>
      </authorization>
    </system.web>
  </location>
```

```
</configuration>
```

Zawierzony cel został osiągnięty, jednak musimy pamiętać o utrzymaniu kilku (w przypadku tego projektu) plików konfiguracyjnych.

Kolejna rzeczą była konieczność ukrycia odpowiednich elementów menu przed użytkownikami, którzy nie mają dostępu do danych obszarów. W przypadku ASP.NET Webforms konieczne było wykorzystanie kontrolki LoginView, która pozwala na wyświetlanie różnych informacji dla anonimowych oraz zalogowanych użytkowników [28].

Listing 18 Deklaracja kontrolki LoginView. Źródło: Opracowanie własne.

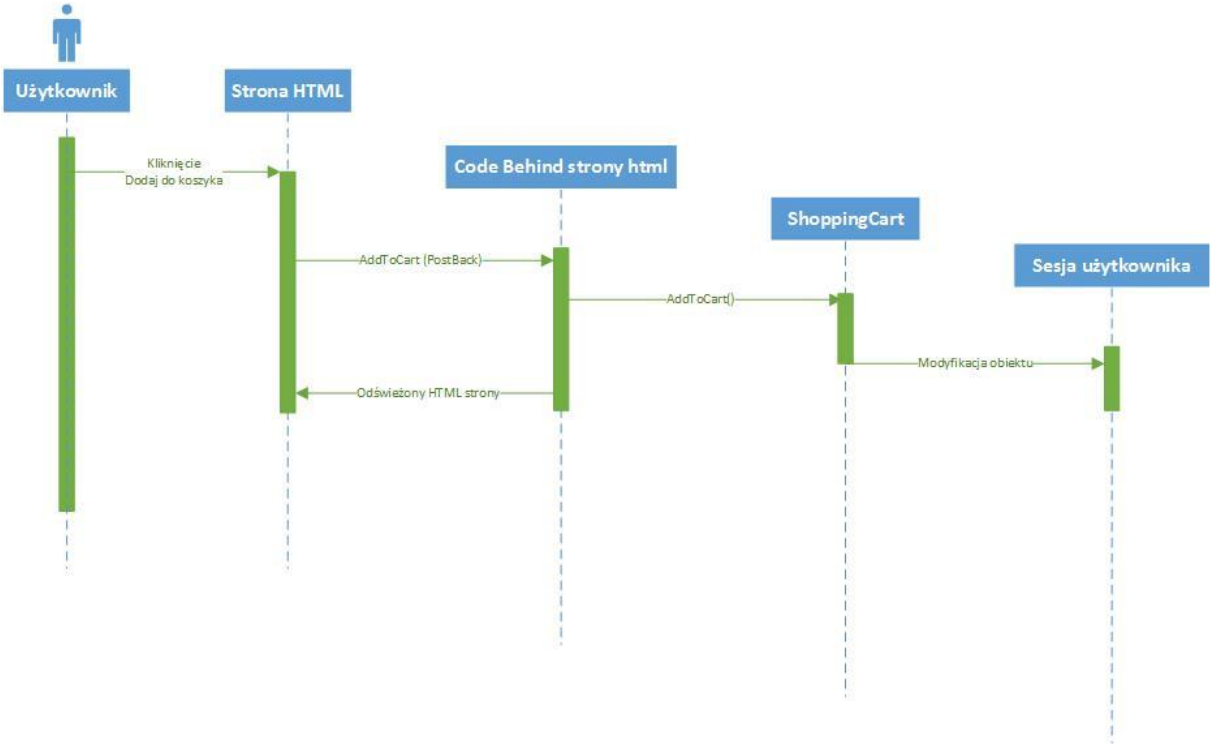
```
<asp:LoginView runat="server" ViewStateMode="Disabled">
<RoleGroups>
<asp:RoleGroup Roles="Administrator">
      <ContentTemplate>
      <li><a runat="server"
href="~/Manufacturers/Default">ManufacturerAdm</a></li>
      <li><a runat="server" href="~/Boards/Default">BoardAdm</a></li>
      <li><a runat="server"
href="~/Customers/Default">Customers</a></li>
      <li><a runat="server" href="~/Orders/Default">Orders</a></li>
      </ContentTemplate>
</asp:RoleGroup>
</RoleGroups>
</asp:LoginView>
```

Powyższy fragment kodu znajduję się na stronie nadrzędnej w sekcji deklaracji menu. Kontrolka wyświetla cztery linki użytkownikom, którzy należą do roli *Administrator*.

3.5.5. Koszyk

Do implementacji funkcjonalności koszyka została wykorzystana ta sama klasa, jak w podejściu ASP.NET MVC. Inny jest natomiast sposób dostępu do klasy *ShoppingCart*. W przypadku ASP.NET MVC, każda operacja wykonywana była za pośrednictwem kontrolera, w podejściu ASP.NET WebForms nie było to możliwe. Należało wykonać takie operacje poprzez wywołanie PostBack i przekazanie działania do odpowiedniej metody w CodeBehind strony, na której dane zdarzenie zostało, poprzez kliknięcie w przycisk, wywołane przez użytkownika. Zatem wszystkie modyfikacje na obiekcie przechowującym zarówno dane koszyka jak i zamówienia są porozrzucane po różnych miejscach w kodzie aplikacji.

Rysunek 32 przedstawia diagram sekwencji dodawania produktu do koszyka.



Rysunek 32 Diagram sekwencji dodawania do koszyka - ASP.NET WebForms.
Źródło: Opracowanie własne.

3.5.6. UserControl i MasterPage

Odpowiednikiem PartialView i Layout w podejściu ASP.NET Webforms są kontrolki użytkownika (*UserControl*) oraz strony nadrzędne (*MasterPage*). *MasterPage* są używane w celu stworzenia wspólnej struktury dla wszystkich stron w witrynie. Pozwalają izolować wspólne elementy interfejsu danej witryny, takie jak logo, menu, wyszukiwarka z zachowaniem unikalności treści każdej ze strony. Poszczególne strony ASPX są „wstrzykiwane" wewnątrz strony nadrzędnej, dzięki temu, kiedy pojawi się potrzeba aktualizacji powtarzającej się sekcji jak menu, stopka strony, nie trzeba przerabiać każdej ze stron z osobna. *UserControl* są to fragmenty widoku przechowywane w oddzielnych plikach, co pozwala programiście na podział widoku na małe części, a następnie łączenie ich w czasie wykonywania aplikacji. *UserControl* oferują doskonały sposób, aby wielokrotnie korzystać daną sekcję w wielu miejscach [12].

Powyższe mechanizmy pozwalają uzyskać niemal identyczne wyniki jak w przypadku podejścia ASP.NET MVC. Jednak w przypadku ASP.NET Webform należy pamiętać o cyklu życia strony i kolejności, w jakiej wykonywane są poszczególne zdarzenia stron nadrzędnych, kontroler użytkownika. Bez tej wiedzy programista może się bardzo łatwo zagubić, co może doprowadzić do błędów rzutujących na warstwę prezentacyjną.

3.5.7. Linki

W początkowym etapie tworzenia prototypu podejściem ASP.NET Webforms wydawało się, że przyjazne linki będą jednym z trudniejszych wyzwań. Jak się okazuje z .NET Framwork 4.0 (a nawet z Service Pack 1 do .NET Framework 3.5) Microsoft dostarcza ten sam silnik routingu, co w ASP.NET MVC. Mechanizm ten pozwala oddzielić związek pomiędzy adresem URL w żądaniu HTTP i fizycznym plikiem w witrynie. Umożliwia to budowanie przyjaznych linków dla aplikacji internetowej [29].

Deklaracja interesującej nas route wygląda bardzo podobnie jak w ASP.NET MVC.

Listing 19 Deklaracja route - podejście ASP.NET Webforms. Źródło: Opracowanie własne.

```
routes.MapPageRoute(
      "Basket",
      "Basket",
      "~/ShoppingCart/Default.aspx");
```

W celu uniknięcia mapowania wszystkich fizycznych plików na witrynie warto skorzystać z dodatku ASP.NET FriendlyUrls wydanym w lutym 2013 roku. Umożliwia on automatyczne zmapowanie wszystkich fizycznych plików do postaci bez rozszerzeń (takich jak .aspx lub .ashx) [30].

Dzięki temu zamiast http://snowshop.pl/Store/Board.aspx?BoardId=1 mechanizm przekształca to do postaci http://snowshop.pl/Store/Board/1 co jest bardziej czytelne.

3.5.8. Podsumowanie

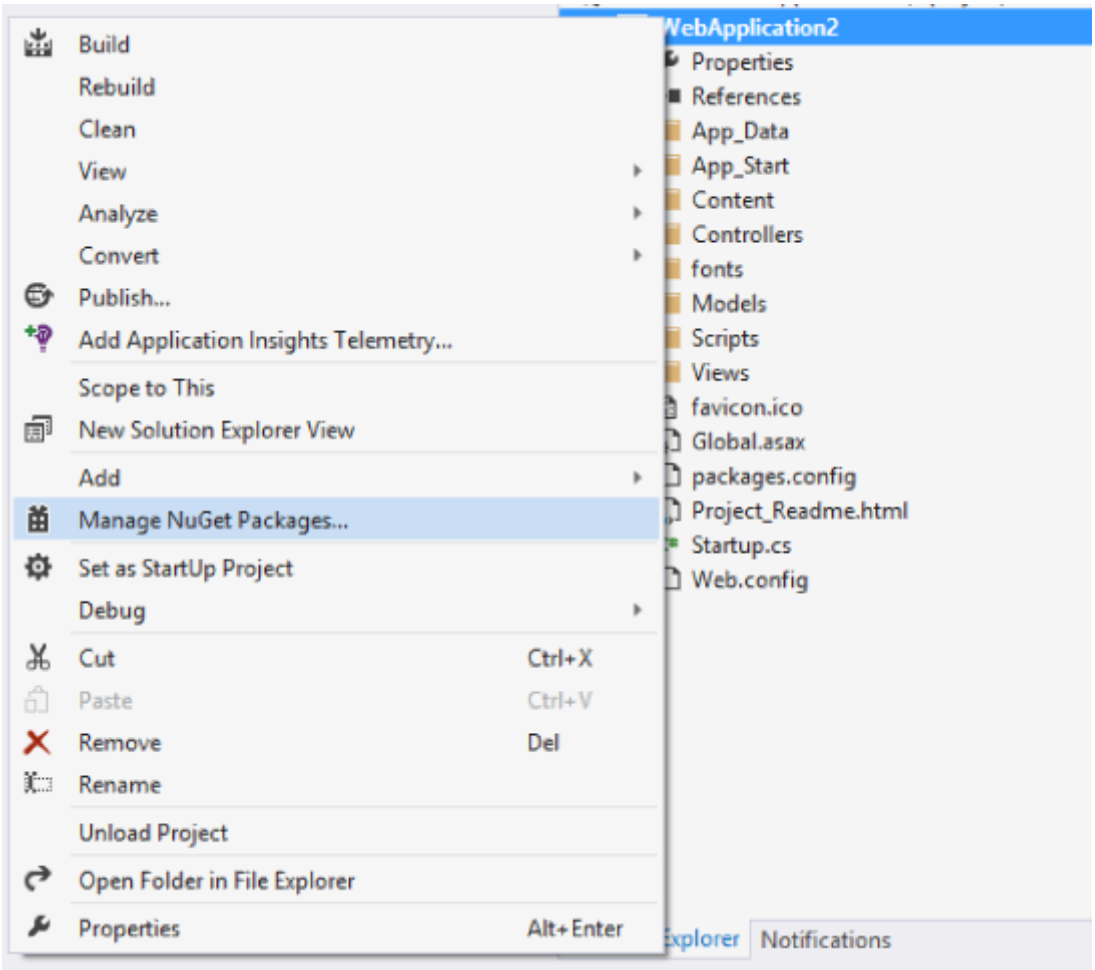
Implementacja podejściem ASP.NET Webforms z początku wydawała się problematyczna. Jednak przy pomocy kilku dodatków (Web Forms Scaffolding, ASP.NET FriendlyUrls) można było w dość szybki i nieskomplikowany sposób osiągnąć zamierzone cele. Mechanizm autoryzacji i autentykacji użytkowników był dostarczony w postaci ASP.NET Identity 2, konfiguracji natomiast wymagało ustawienie dostępu do poszczególnych lokalizacji na witrynie. Mimo, że było to kwestia dość niewygodna, to spełniała swoją funkcję. Jednak obecność wielu plików konfiguracyjnych web.config (nie ma możliwości zmiany nazwy) może być kłopotliwa i sprawia trudności podczas utrzymywania aplikacji. Konieczne, a zarazem problematyczne, okazało się instalowanie dodatków na wszystkich maszynach, które brały udział w implementacji rozwiązania. Każda stacja robocza, aby w pełni wykorzystać potencjał Scaffoldingu lub też FriendlyUrls musiała być w pierwszej kolejności zaopatrzona w powyższe funkcje dostarczane z zewnętrznych bibliotek. Końcowa struktura plików i katalogów stała się zawiła, a mamy tu do czynienia z mało skomplikowanym prototypem. Funkcjonalność koszyka spełnia swoje założenia, jednak dalsze utrzymanie może być kłopotliwe. Wynika to z tego, że metody odpowiedzialne za modyfikację koszyka nie znajdują się w jednym miejscu, a są porozrzucane po całej aplikacji.

3.6. Implementacja ASP.NET MVC + AngularJS

Rozdział ten zawiera opis implementacji prototypu podejściem ASP.NET MVC + AngularJS.

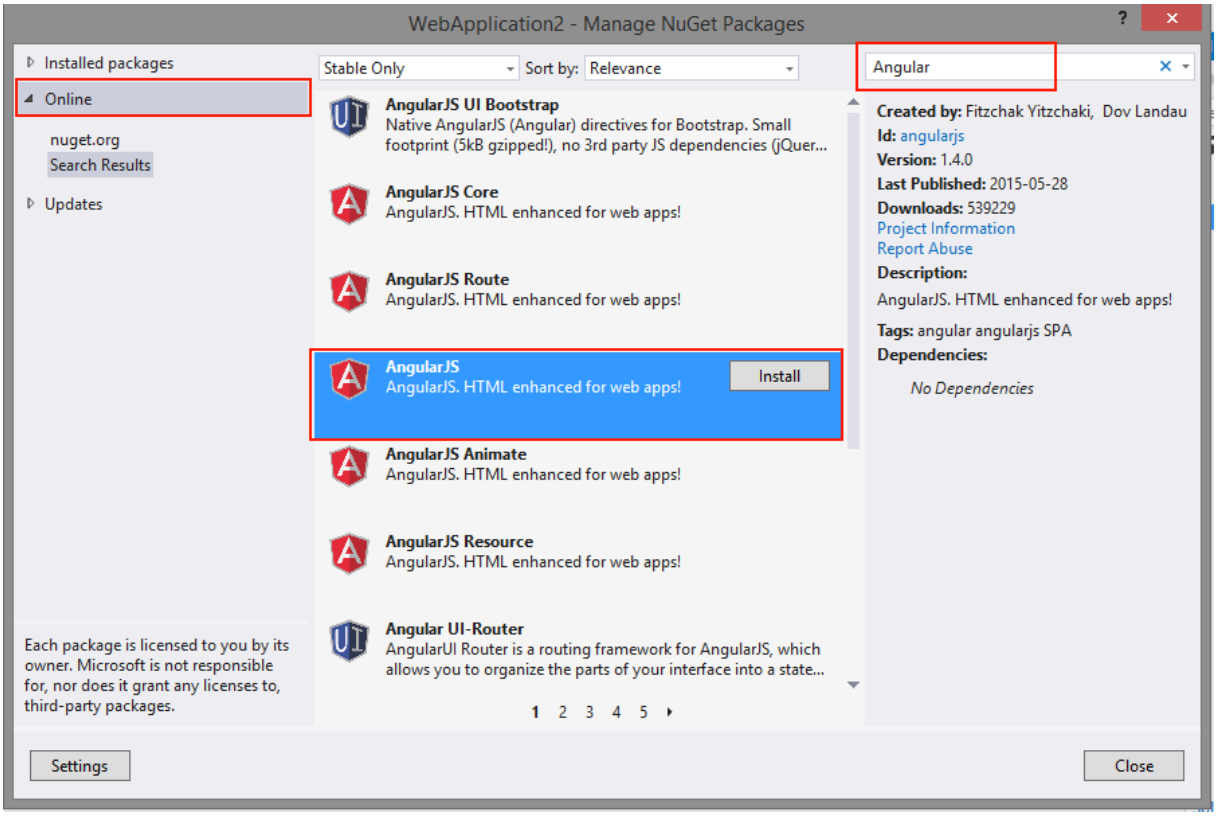
3.6.1. Tworzenie projektu

Początkowa faza tworzenia projektu podejście ASP.NET MVC + AngularJS wygląda niemal identycznie jak w przypadku ASP.NET MVC. Wykorzystano to samo narzędzie (Visual Studio 2013), ten sam framework (4.5.1), jak również identyczny szablon zawierający bibliotekę Bootstrap, funkcjonalność Entity Framework oraz narzędzie ASP.NET Identity 2. Struktura katalogów pozostała taka sama. Funkcją, która musiała być dodatkowo zainstalowana była biblioteka AngularJS. Można tego dokonać wykorzystując mechanizm zarządzania pakietami. Niezbędne do wykonania operacje zostały przedstawione na Rysunek 33 oraz Rysunek 34.



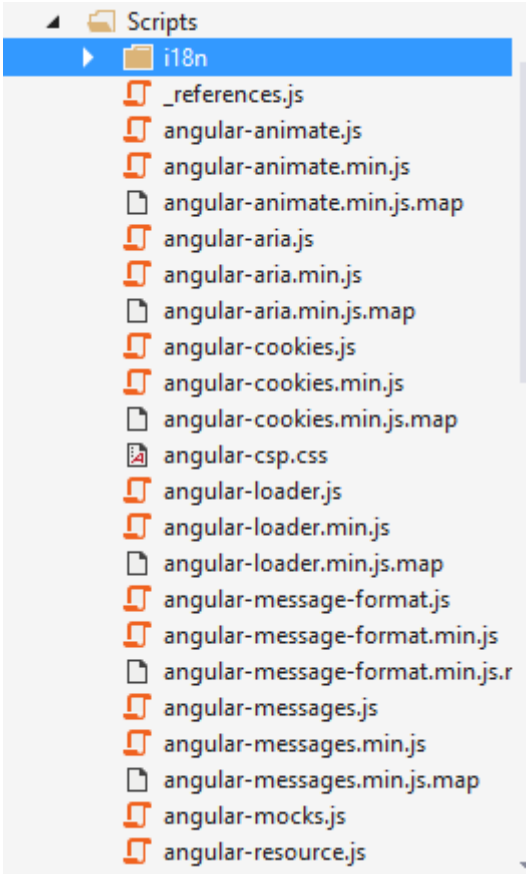
Rysunek 33 Instalowanie AngularJS - krok 1.
Źródło: Opracowanie własne.

W pierwszym kroku należy wybrać opcję *Manage NuGet Packages* dostępną w menu kontekstowym interesującego nas projektu.



Rysunek 34 Instalowanie AngularJS - krok 2.
Źródło: Opracowanie własne.

Kolejny krok polega na wyszukaniu interesującej nas biblioteki z katalogu Online. Po instalacji, do naszego projektu zostaną dodane niezbędne do działania pliki przedstawione na Rysunek 35. Dodatkowo dodatek instaluje również pliki lokalizacyjne odpowiedzialne za tłumaczenia tekstów użytych w silniku AngularJS na język zgodny z tym, jaki używa użytkownik.



Rysunek 35 Struktura plików AngularJS.
Źródło: Opracowanie własne.

3.6.2. Użytkownicy

Pomimo użycia narzędzia ASP.NET Identity 2 kwestia użytkowników wcale nie było taka trywialna. Dostęp do bazy, metody dodawania użytkowników mogły być wykorzystane, jednak przebudować należało warstwę prezentacyjną oraz komunikację z kontrolerem po stronie serwerowej. Tak jak w przypadku ASP.NET MVC należało zmodyfikować klasę *RegisterViewModel* o brakujące dane odzwierciedlające pola w tabeli *Customer*.

Aby umożliwić rejestrację użytkowników należało stworzyć kontroler AngularJS. Jak pokazano na Listing 20 zawiera on definicję danych przekazywanych z formularza oraz metodę wykonywaną w momencie potwierdzenia formularza rejestracyjnego.

Listing 20 Deklaracja RegisterController w AngularJS. Źródło: Opracowanie własne.

```
var RegisterController = function ($scope, $location,
RegistrationFactory) {
    $scope.registerForm = {
        emailAddress: '',
        password: '',
        confirmPassword: '',
        name: '',
        surname: '',
```

```
        address: '',
        phone: '',
        failure: false
    };
    $scope.register = function () {
        var result =
RegistrationFactory($scope.registerForm.emailAddress,
$scope.registerForm.password, $scope.registerForm.confirmPassword,
$scope.registerForm.name, $scope.registerForm.surname,
$scope.registerForm.address, $scope.registerForm.phone);
        result.then(function (result) {
            if (result.success) {
                $location.path('/home');
            } else {
                $scope.registerForm.failure = true;
            }
        });
    }
}
RegisterController.$inject = ['$scope', '$location',
'RegistrationFactory'];
```

Metoda *register* przekazuje dane otrzymane z formularza i przesyła je do nowo powstałej fabryki *RegistrationFactory*, gdzie odbywa się połączenie z kontrolerem po stronie serwera. W zależności od rezultatu rejestracji, użytkownik jest przekierowywany na stronę domową lub też, wyświetlany jest komunikat błędu. Listingu 21 przedstawia deklarację fabryki RegistrationFactory.

Listing 21 Deklaracja RegistrationFactory w AngularJS. Źródło: Opracowanie własne.

```
var RegistrationFactory = function ($http, $q) {
    return function (emailAddress, password, confirmPassword, name,
surname, address, phone) {
        var deferredObject = $q.defer();
        $http.post(
            '/Account/Register', {
                Email: emailAddress,
                Password: password,
                ConfirmPassword: confirmPassword,
                Name: name,
                Surname: surname,
```

```
                Address: address,
                Phone: phone
            }
        ).
        success(function (data) {
            if (data == "True") {
                deferredObject.resolve({ success: true });
            } else {
                deferredObject.resolve({ success: false });
            }
        }).
        error(function () {
            deferredObject.resolve({ success: false });
        });
        return deferredObject.promise;
    }
}
RegistrationFactory.$inject = ['$http', '$q'];
```

RegistrationFactory otrzymane dane z kontrolera Angular przekazuje metodą POST do kontrolera po stronie serwerowej. Dopiero w tym momencie wykorzystywany jest mechanizm ASP.NET Identity 2, który tworzy użytkownika w bazie danych, dodaje nowy wpis w tabeli *Customer* oraz loguje nowo stworzonego użytkownika do aplikacji.

Logowanie użytkowników wygląda w ten sam sposób. Należało stworzyć nowy kontroler Angular oraz nową fabrykę łączącą się z kontrolerem po stronie serwerowej. Warstwa prezentacji poprzez wywołanie metody kontrolera Angular przekazuje dane zebrane z formularze, które następnie otrzymuje fabryka, żeby połączyć się z kontrolerem po stronie serwera i dokonać już właściwego zalogowania użytkownika.

Na pierwszy rzut oka oba procesy nie wymagały zbyt wiele wysiłku. Łatwo jednak sobie wyobrazić sytuację, gdzie modyfikacji ulega model użytkownika i należy podczas rejestracji zapisać nowe dane, na przykład pesel. W takim przypadku nakład pracy jest bardzo duży. Należy zmodyfikować klasę *RegisterViewModel*, dokonać niezbędnych zmian w kontrolerze oraz fabryce Angular. Poza tym warstwa prezentacji musi być zaktualizowana o nowe dane jak również kontroler po stronie serwera wymaga dostosowania. W takim przypadku uzasadnione wydaje się pozostanie przy standardowym podejściu ASP.NET MVC i nie komplikowaniu mechanizmów logowania oraz rejestracji użytkowników.

3.6.3. Scaffolding

Na dzień pisania pracy nie istniało rozwiązanie pozwalające na użycie mechanizmu *scaffolding* w kontekście tworzonego prototypu podejściem ASP.NET MVC + AngularJS. Rozwiązaniem godnym uwagi było wykorzystanie obecnego dla ASP.NET MVC narzędzia, a następnie ręczne modyfikowanie widoków. Jest to jednak pół-środek. Często pisanie kodu od zera było szybsze niż przechodzenie po

wygenerowanej stronie i dokonywanie zmian. Tak jak w przypadku *Użytkowników* należało stworzyć nowe kontrolery Angular dostarczające funkcjonalności zbierania danych oraz przekazywania ich do strony serwerowej, gdzie mogły być fizycznie (w bazie danych) modyfikowane.

3.6.4. Panel administracyjny

W celu ograniczenia dostępu do poszczególnych sekcji został wykorzystany ten sam mechanizm, co w podejściu ASP.NET MVC. Odpowiednie kontrolery otrzymały atrybut *Authorize*, co w połączeniu z narzędziem ASP.NET Identity praktycznie spełniło zamierzony cel. Do tego momentu proces autoryzacji wygląda w tej sposób, że gdy użytkownik próbuje dostać się na stronę, do której nie ma dostępu zostaje przekierowany na stronę logowania. Sprawia to, że cała strona jest wyświetlana od nowa, a nasza aplikacja przestaje być *Single Page Application*. Zadanie polegało na tym, aby to AngularJS był odpowiedzialny za kod odpowiedzi 401 - nieautoryzowany dostęp.

W tym celu powstała nowa fabryka odpowiedzialna za przechwytywanie odpowiedzi, sprawdzanie ich kodu, następnie wyświetlanie widoku z panelem logowania w przypadku nieautoryzowanej próby dostępu.

Listing 22 Deklaracja fabyrki AuthHttpResponseInterceptor. Źródło: [31]

```
var AuthHttpResponseInterceptor = function ($q, $location) {
    return {
        response: function (response) {
            if (response.status === 401) {
                console.log("Response 401");
            }
            return response || $q.when(response);
        },
        responseError: function (rejection) {
            if (rejection.status === 401) {
                console.log("Response Error 401", rejection);
                $location.path('/login').search('returnUrl',
$location.path());
            }
            return $q.reject(rejection);
        }
    }
}
AuthHttpResponseInterceptor.$inject = ['$q', '$location'];
```

Pozostała jeszcze kwestia ukrycia odpowiednich pozycji w menu przed użytkownikami z ograniczonym dostępem. W tym celu wykorzystano dyrektywę *ng-show*, która odpowiada za pokazywanie lub ukrywanie elementów w zależności od przekazanej zmiennej.

Listing 23 Deklaracja nawigacji - podejście ASP.NET MVC + AngularJS. Źródło: Opracowanie własne.

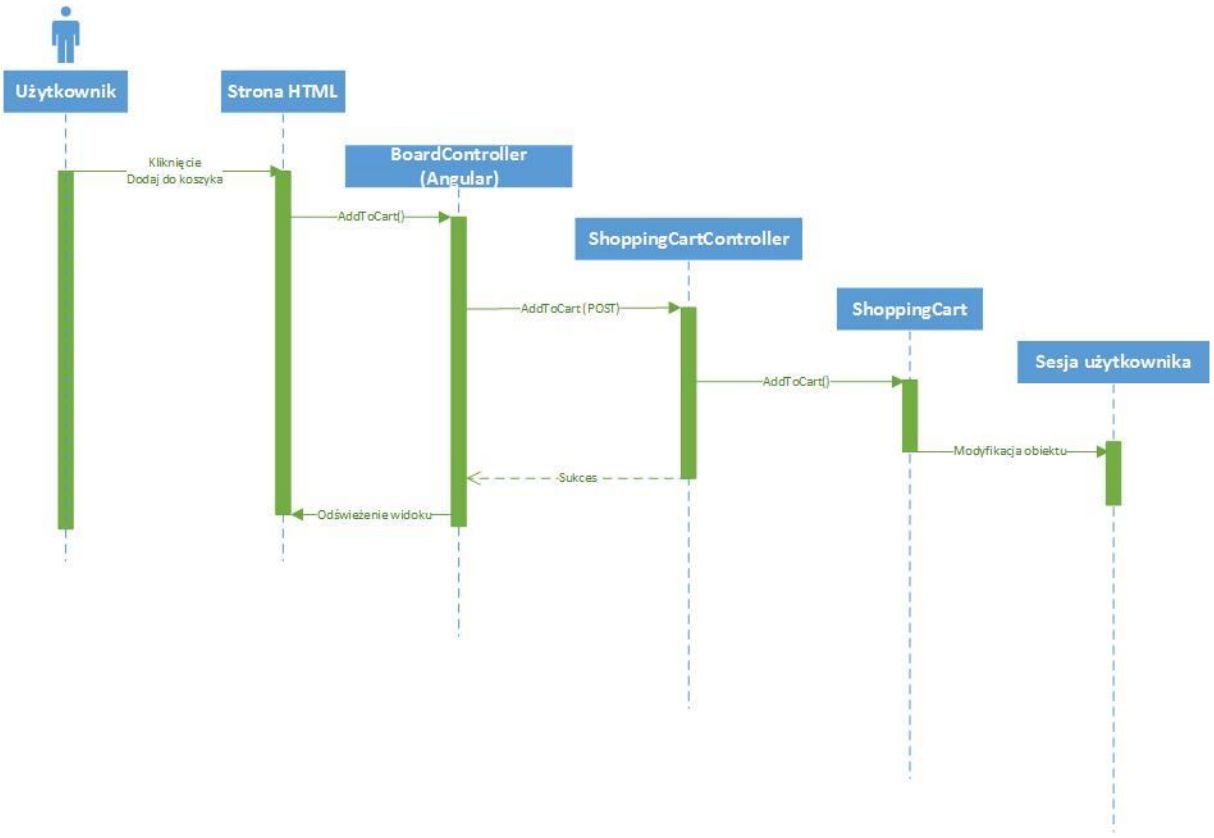
```
<li ng-show="models.isAdministrator"><a
href="/#/ManufacturerManager">ManufacturerManager</a></li>

<li ng-show="models.isAdministrator"><a
href="/#/BoardManager">BoardManager</a></li>

<li ng-show="models.isAdministrator"><a
href="/#/Customers">Customers</a></li>

<li ng-show="models.isAdministrator"><a
href="/#/Orders">Orders</a></li>
```

Powyższy listing przedstawia, jak na podstawie właściwości modelu *isAdministrator* silnik AngularJS za pomocą dyrektywy *ng-show* ukrywa ograniczone dostępem elementy. Właściwość *isAdministrator* jest deklarowana w głównym kontrolerze Angular. Pobiera on informacje o tym, czy aktualny użytkownik należy do roli *Administrator* z kontrolera po stronie serwerowej.

3.6.5. Koszyk

Po stronie serwerowej funkcjonalność koszyka wygląda identycznie jak w przypadku ASP.NET MVC. Cała funkcjonalność jest skupiona w jednym miejscu. Po stronie warstwy prezentacji wygląda to podobnie. Kontroler Angular używany na karcie produktu, gdzie można dokonać dodania elementu do koszyka, wyposażony jest w metodę łączącą się z warstwą serwerową. Przesyła on odpowiednie informacje o produkcie żądaniem POST.

Rysunek 36 przedstawia proces dodawania produktu do koszyka.



Rysunek 36 Diagram sekwencji dodawania do koszyka - ASP.NET MVC + AngularJS. Źródło: Opracowanie własne.

3.6.6. ngView i ngInclude

Wraz z mechanizmem routingu AngularJS wprowadza dyrektywę *ng-view*, która służy do wczytywania plików HTML w miejsce elementów oznaczonego tą dyrektywą. Aplikacja AngularJS może posiadać tylko i wyłącznie jeden element *ng-view*. Odpowiedzialny jest on za „dostrzeganie" zmian w adresie URL i na jego podstawie aktualizację zawartości zgodnie z definicją *route* [17].

Dyrektywa *ng-include*, jako argument przyjmuje ścieżkę do pliku HTML. Plik ten jest następnie ładowany wewnątrz elementu, w którym występuje dyrektywa. Pozwala to, na rozbicie kodu na mniejsze oraz łatwiejsze w utrzymaniu pliki, a co najważniejsze, wielokrotne ich użycie. W przypadku, kiedy mamy do czynienia z dużymi plikami HTML, możemy łatwo wyodrębnić je na mniejsze, łatwiejsze do zarządzania i uczynić naszą aplikację bardziej modułową [17].

Powyższe funkcjonalności bardzo przypominają UserControl i MasterPage w podejściu ASP.NET Webforms, a także PartialView i Layout w ASP.NET MVC. Dyrektywę *ng-view* można „obudować" kodem HTML, który chcemy, aby powtarzał się na każdej stronie, na przykład: menu nawigacyjne, nagłówek, stopkę. Natomiast *ng-include* pozwala wczytywać i wielokrotnie używać zewnętrzne pliki HTML. Obie dyrektywy przyspieszają implementację i ułatwiają utrzymanie aplikacji.

3.6.7. Linki

Dzięki temu, że AngularJS zawiera mechanizm routingu, przygotowanie przyjaznych linków było zadaniem jak najbardziej możliwym. Jednak, aby zachować cechy *Single Page Application* wszystkie możliwe odnośniki musiały posiadać *route*. Co więcej wszystkie *route* musiały zostać zadeklarowane ręcznie. Listing 24 przedstawia fragment deklaracji *route* w głównym module aplikacji AngularJS.

Listing 24 Fragment deklaracji route w AngularJS. Źródło: Opracowanie własne.

```
var configFunction = function ($routeProvider, $httpProvider) {
    $routeProvider.
        when('/home', {
            templateUrl: 'Home/Index'
        })
        .when('/store', {
            templateUrl: 'Store/Index',
            controller: BoardListController
        })
        .when('/store/:id', {
            templateUrl: function (params) { return
'Store/Board?id=' + params.id; },
            controller: BoardController
        })
        .when('/login', {
            templateUrl: '/Account/Login',
            controller: LoginController
        })
```

```
        .when('/register', {
            templateUrl: '/Account/Register',
            controller: RegisterController
        })
        .when('/account', {
            templateUrl: 'Account/Manage'
        })
        .when('/cart', {
            templateUrl: 'ShoppingCart/Index'
        })
        .when('/ManufacturerManager', {
            templateUrl: 'ManufacturerManager/Index'
        })
        .otherwise({
            templateUrl: 'Home/Welcome',
        });
}
```

3.6.8. Podsumowanie

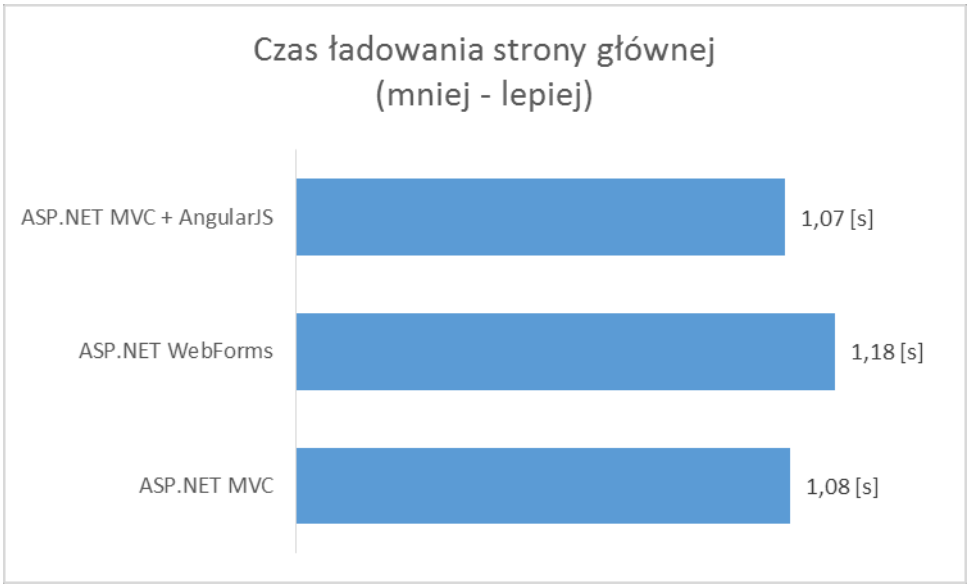
Implementacja podejściem ASP.NET MVC + AngularJS była najbardziej skomplikowana. Angular zawiera wszystkie niezbędne mechanizmy, aby osiągnąć zamierzony cel. Nakład pracy natomiast był dużo większy. Przyczyną nie jest tylko i wyłącznie brak narzędzia scaffoldingu ale również konieczność implementacji dodatkowych kontrolerów po stronie warstwy prezentacyjnej. Tworzenie projektu było analogicznej jak w przypadku ASP.NET MVC. Funkcjonalność logowania oraz rejestracji wymagała stworzenia dodatkowych kontrolerów, a także fabryk, które umożliwiły przechwytywanie nieautoryzowanych żądań. Dyrektywy *ngView* i *ngInclude* pozwoliły na większą modułowość aplikacji oraz łatwiejsze utrzymanie jej w przyszłości. Mechanizm routing w zupełności wystarczył, aby adresy url były w przyjaznej postaci. Pozostaje pytanie, czy cały nakład dodatkowej pracy, warty był osiągniętego efektu, czyli powstania *Single Page Application*. Czy może jednak w przypadku takiego projektu, był to zabieg nadmiarowy.

4. Wnioski w odniesieniu do stworzonych aplikacji

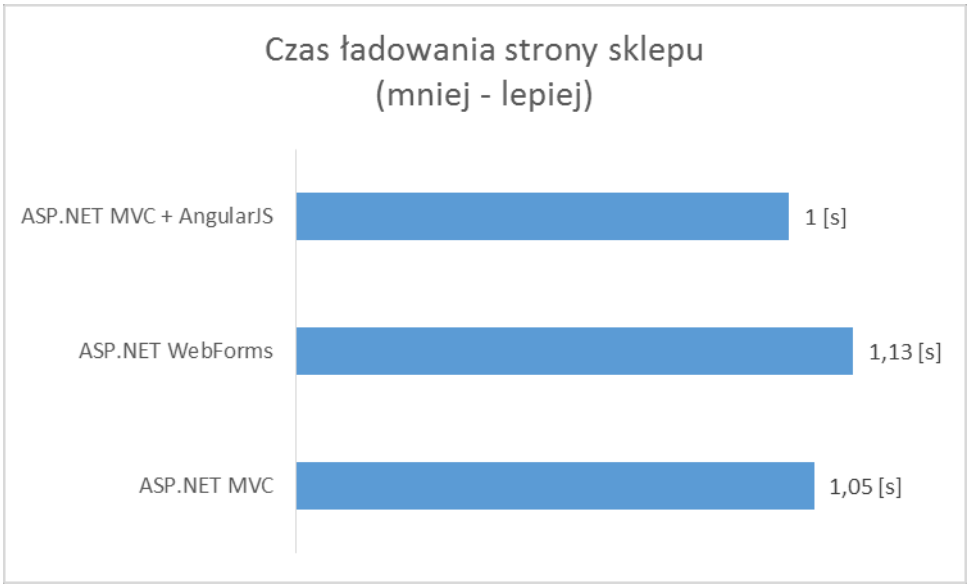
Wszystkie trzy podejścia spełniły swoją funkcję i pozwoliły na stworzenia aplikacji sklepu internetowego. Większość założeń została spełniona. W niektórych przypadkach wymagało to użycia zewnętrznych bibliotek. Pod względem łatwości implementacji najlepiej wypadło podejście ASP.NET MVC. W przeciwieństwie do ASP.NET WebForms oraz AngularJS mechanizm *scaffolding* jest obecny w standardzie. Funkcja ta ogromnie przyspiesza pracę programistom. Po zainstalowaniu odpowiedniego dodatku mechanizm ten był również dostępny w podejściu ASP.NET WebForms. Wymagało to jednak instalacji dodatku na każdej stacji roboczej biorącej udział w procesie implementacji. W podejściu AngularJS wszystkie widoki musiały być implementowane ręcznie. Wydłużyło to proces implementacji.

4.1. Czasy ładowania stron

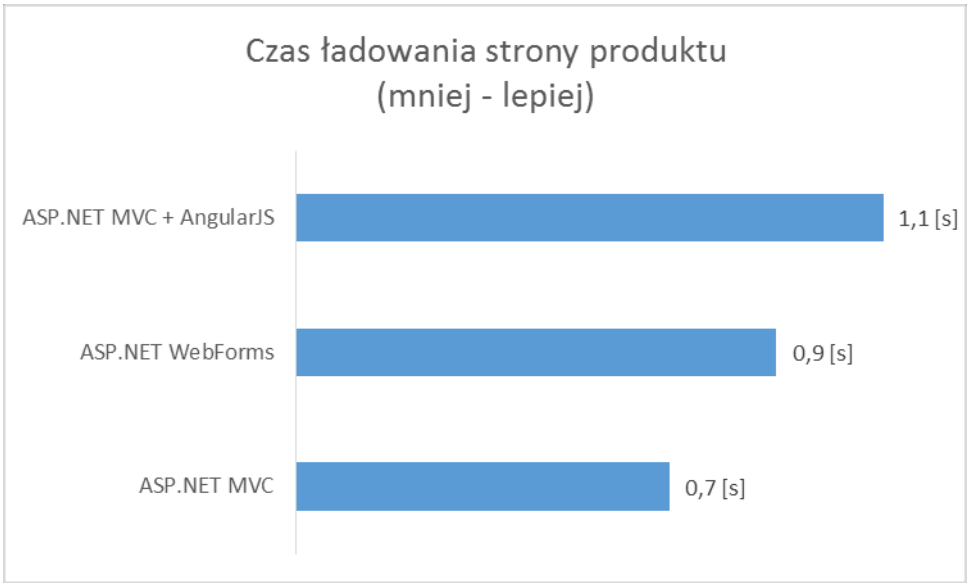
W ramach badania dokonano pomiaru prędkości otwierania się poszczególnych stron. Wykorzystano do tego dodatek *FireBug* dostępny dla przeglądarki Mozilla Firefox. Rysuneki 37, 38 oraz 39 przedstawiają wyniki podane w sekundach dla poszczególnych podejść.



Rysunek 37 Wykres czasu ładowania strony głównej.
Źródło: Opracowanie własne.



Rysunek 38 Wykres czasu ładowania strony sklepu.
Źródło: Opracowanie własne.

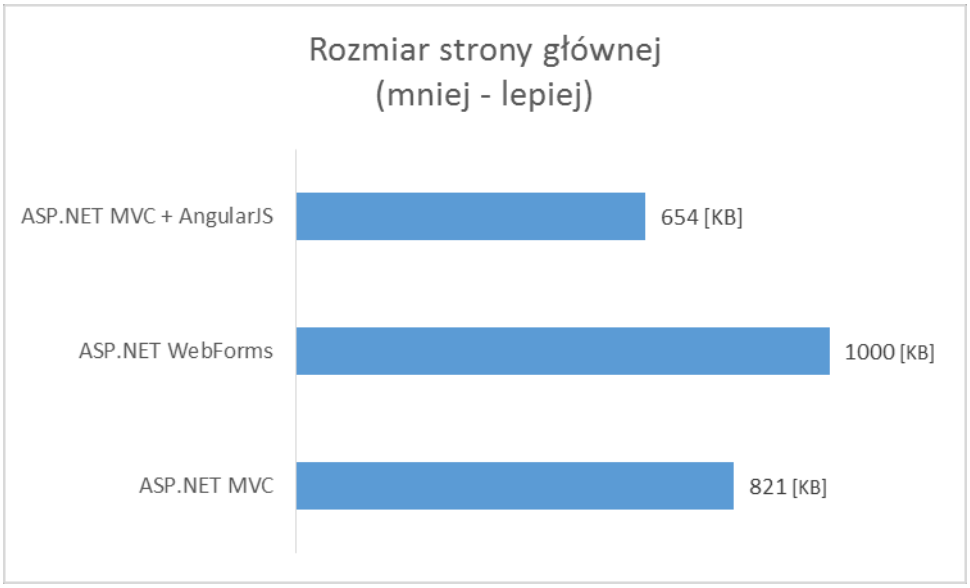


Rysunek 39 Wykres czasu ładowania strony produktu.
Źródło: Opracowanie własne.

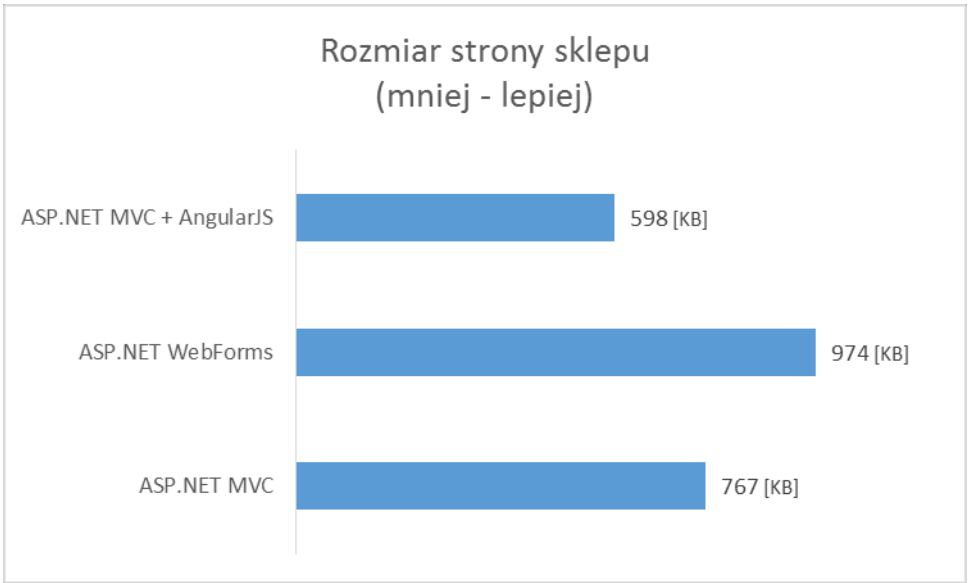
Powyższe wykresy przedstawiają wyniki uśrednione. Dokonano kilku pomiarów a następnie wynik uśredniono. W dwóch na trzech przypadkach podejście ASP.NET WebForms osiąga najsłabszy rezultat. Różnica pomiędzy ASP.NET MVC a ASP.NET MVC + AngularJS jest niewielka i raczej niezauważalna dla użytkownika końcowego.

4.2. Rozmiary stron

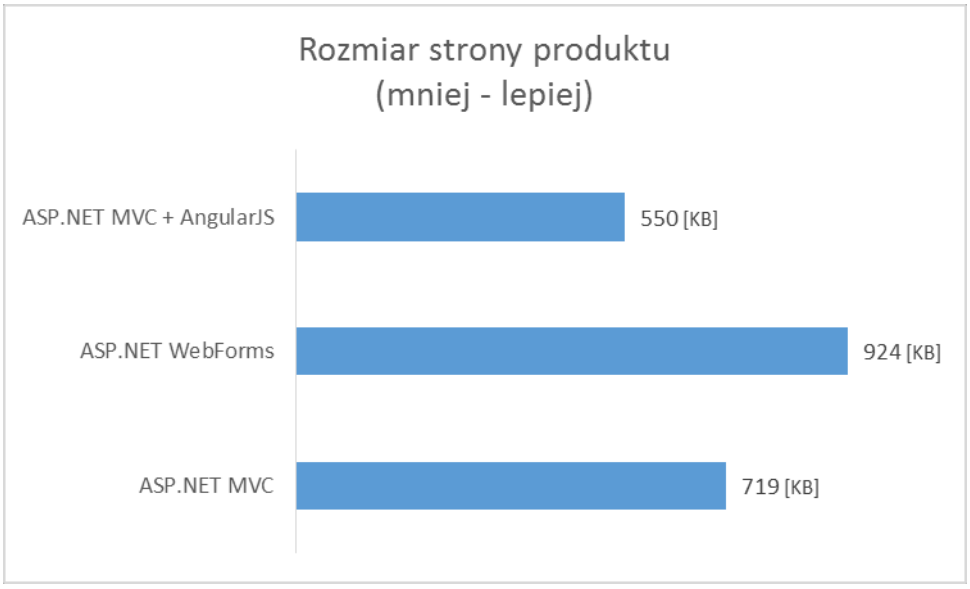
Kolejne badanie polegało na zmierzeniu rozmiaru, jakie zajmują poszczególne strony. Aby tego dokonać dana strona została pobrana i zapisana na dysk twardy. Rysunki 40, 41 oraz 42 przedstawiają wyniki dla poszczególnych podejść w kilobajtach.



Rysunek 40 Wykres rozmiaru strony głównej.
Źródło: Opracowanie własne.

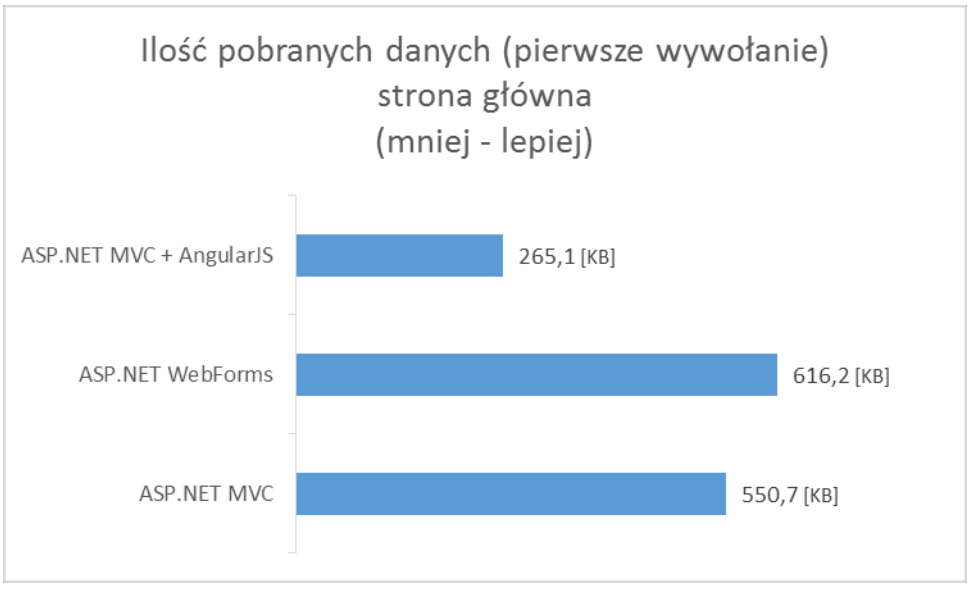


Rysunek 41 Wykres rozmiaru strony sklepu.
Źródło: Opracowanie własne.

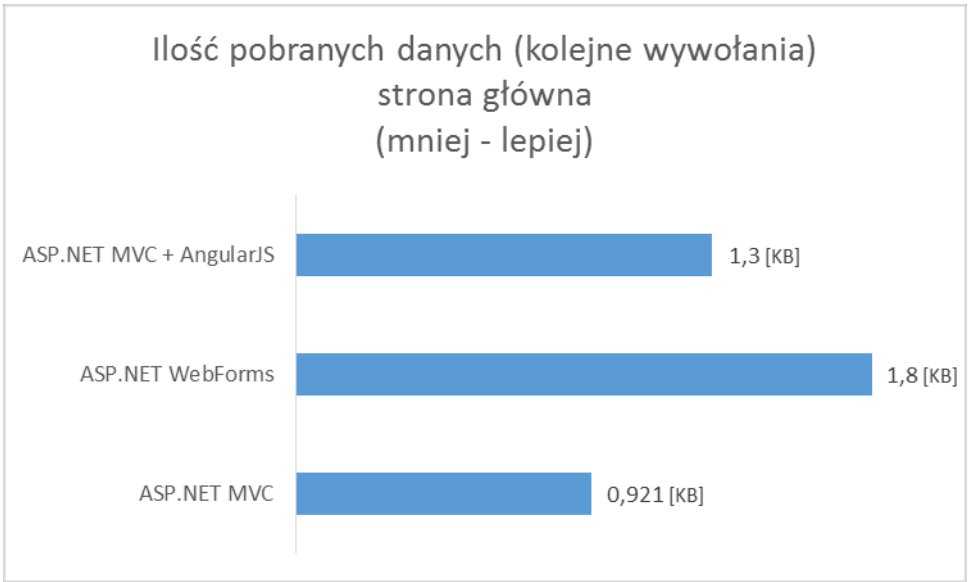


Rysunek 42 Wykres rozmiaru strony produktu.
Źródło: Opracowanie własne.

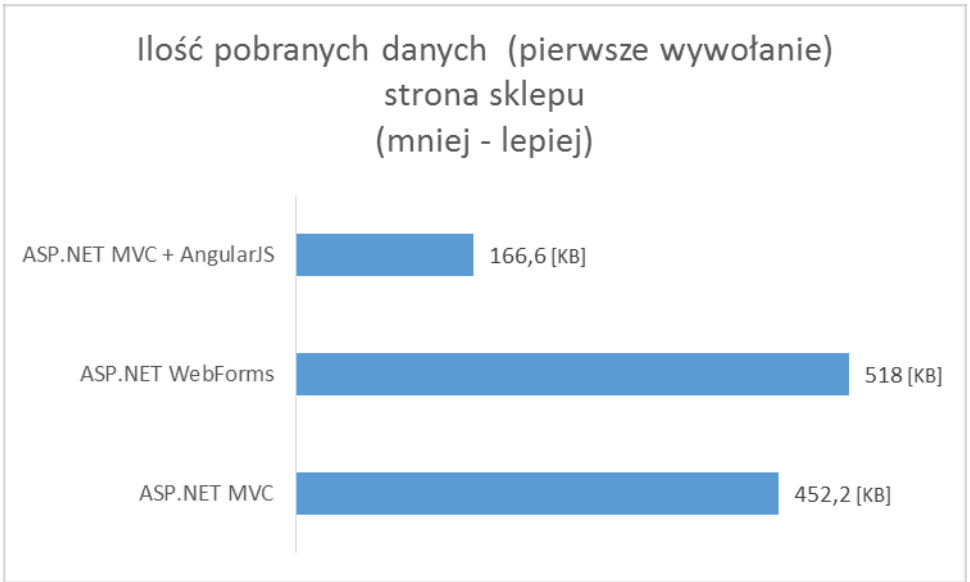
W ramach uzupełniania badań dokonano sprawdzenia faktycznej wielkości pobranych danych. Wykorzystano do tego narzędzie *FireBug*. Rysunki od 43 do 48 przedstawiają wyniki dla poszczególnych stron w kilobajtach zarówno dla pierwszego wywołania jak i kolejnych (po *cache'owaniu* przez przeglądarkę).



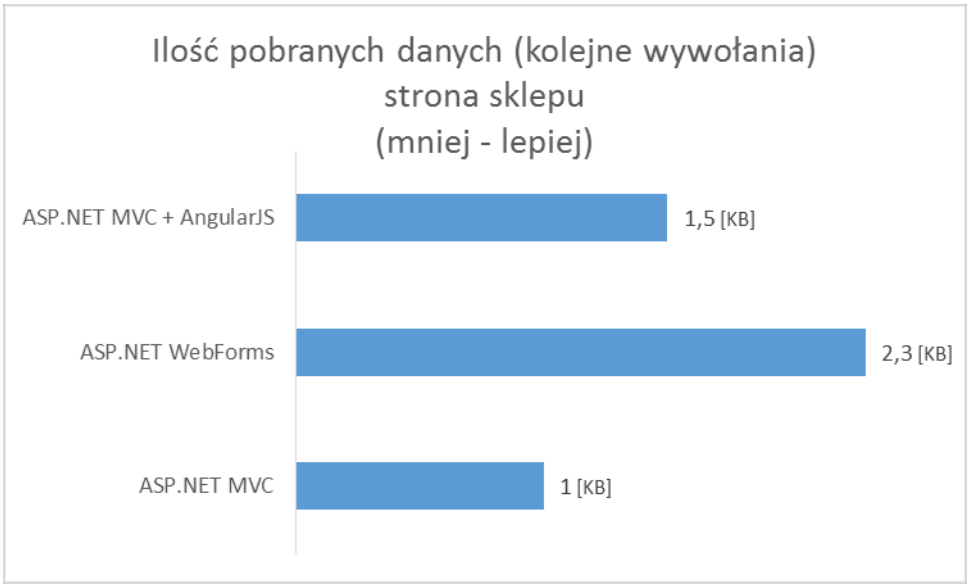
Rysunek 43 Wykres ilości pobranych danych (pierwsze wywołanie) strony głównej.
Źródło: Opracowanie własne.



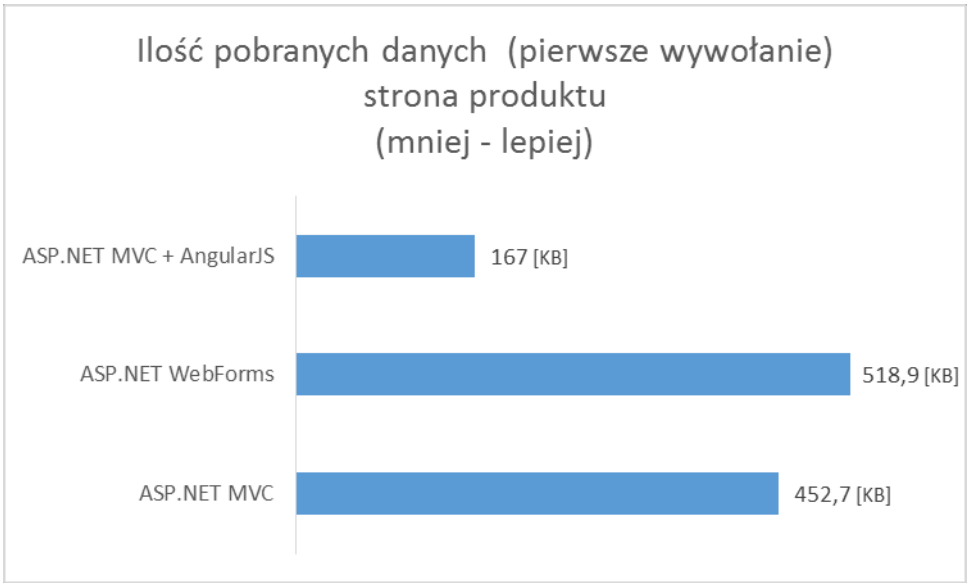
Rysunek 44 Wykres ilości pobranych danych (kolejne wywołania) strony głównej.
Źródło: Opracowanie własne.



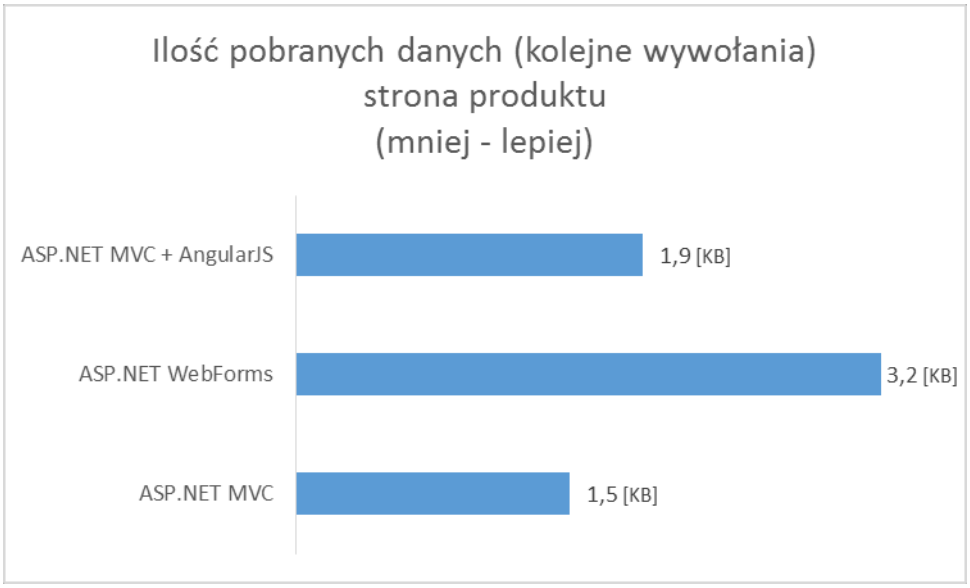
Rysunek 45 Wykres ilości pobranych danych (pierwsze wywołanie) strony sklepu.
Źródło: Opracowanie własne.



Rysunek 46 Wykres ilości pobranych danych (kolejne wywołania) strony sklepu.
Źródło: Opracowanie własne.



Rysunek 47 Wykres ilości pobranych danych (pierwsze wywołanie) strony produktu.
Źródło: Opracowanie własne.



Rysunek 48 Wykres ilości pobranych danych (kolejne wywołania) strony produktu.
Źródło: Opracowanie własne.

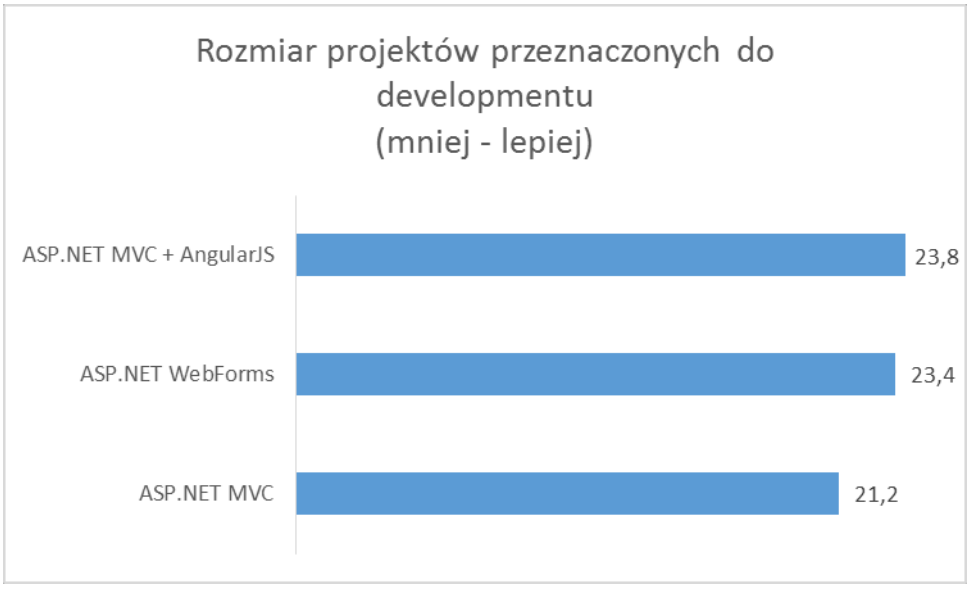
W badaniu na prędkość otwierania się poszczególnych stron mogło dojść do błędu pomiaru wynikającego z wielu przyczyn (na przykład problem z łączem internetowym, opóźnienia w dostępie do bazy). Jednak podczas pomiaru rozmiaru poszczególnych stron nie było mowy o pomyłce. Ewidentnie podejście ASP.NET WebForms wypada najsłabiej w tej kwestii. Ma to na pewno związek z koniecznością wykorzystywania ViewState. Badanie ukazała bardzo duże różnice pomiędzy pierwszym wywołaniem strony, a każdym kolejnym. Powodem tego jest pamięć podręczna. Przeglądarka FireFox tymczasowo przechowuje obrazki, skrypty oraz inne elementy witryny internetowej w pamięci podręcznej, aby przyspieszyć ich przeglądanie.

4.3. Poziom skomplikowania

Jak pokazano na diagramach sekwencji w rozdziałach dotyczących koszyka najbardziej skomplikowaną postać ma podejście z AngularJS. Sprawia to, że rozwiązanie jest bardziej podatne na błędy ze względu na większą ilość etapów, jaki musi pokonać proces dodawania produktu do koszyka. Na pierwszy rzut oka przoduje w tej kwestii podejście ASP.NET Webform, ponieważ w tym przypadku nie występuje żadne wywołanie asynchroniczne. Programista ma pełną kontrolę nad przepływem i bardzo łatwo wychwycić błędy. Na minus przemawia konieczność implementacji takiego przepływu (pierwszych dwóch etapów) na każdej stronie, z której chcemy dodać produkt do koszyka. Co więcej brak asynchroniczności sprawia, że wrażenia użytkownika są znacznie gorsze. Złotym środkiem jest ASP.NET MVC. Nie występuje tu skomplikowany przebieg, nie ma konieczności powtarzania kodu. W przypadku ewentualnych zmian wystarczy zmodyfikować *ShoppingCartController*. Nie trzeba wprowadzać ich w każdym miejscu, z którego dodawany jest produkt do koszyka. Etap wywołania żądania przez użytkownika wykonywany jest asynchronicznie, zatem zyskujemy na wrażeniach i odczuciach użytkownika.

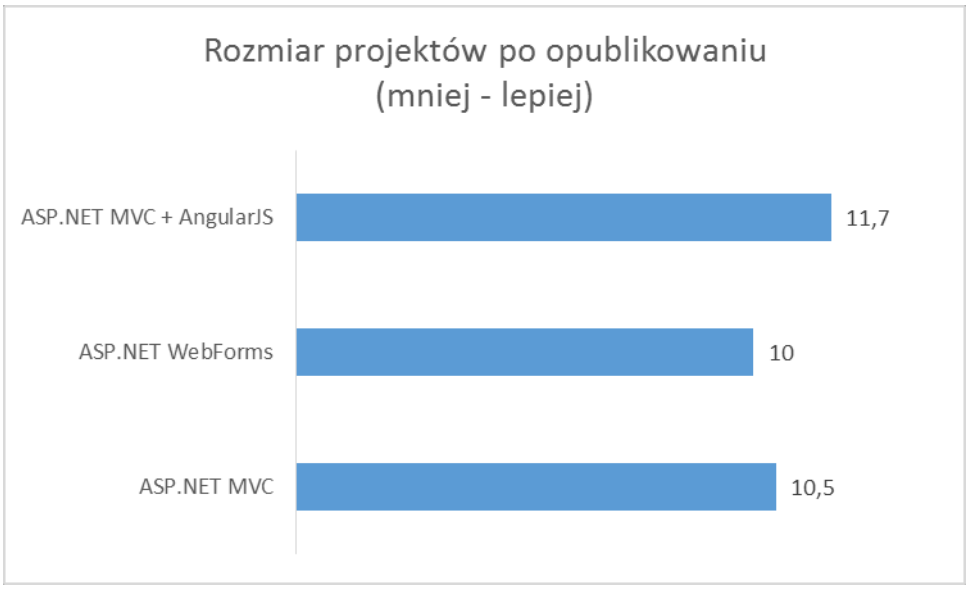
4.4. Struktura katalogów

Struktura katalogów w tworzonym projekcie jest najbardziej czytelna i uporządkowana w podejściu ASP.NET MVC. Mogłoby się wydawać, że w przypadku ASP.NET MVC + AngularJS struktura jest taka sama. Jednak konieczność uporządkowania kontrolerów, fabryk i modułów Angular sprawia, że schemat katalogów się komplikuje. Nie bez znaczenia jest również rozmiar projektów. Aby dokonać pomiaru na koniec implementacji sprawdzono wielkość poszczególnych podejść. Na Rysunek 49 przedstawiono rozmiary projektów w megabajtach.



Rysunek 49 Wykres rozmiaru projektów przeznaczonych do developmentu.
Źródło: Opracowanie własne.

Podejście ASP.NET MVC prezentuje się najlepiej. AngularJS z konieczności przechowywania dodatkowych bibliotek JavaScript prezentuje się najgorzej. Jednak wynik ASP.NET WebForms również nie jest zadowalający. Rozmiar projektu przeznaczonego do developmentu nie ma tak wielkiego znaczenia jak projektu już opublikowanego. Będzie on umieszczony na serwerze, z którego aplikacja będzie udostępniana użytkownikom. Aby zbadać rozmiar dokonano publikacji wszystkich trzech podejść. Rysunek 50 przedstawia wynik badania w megabajtach.



Rysunek 50 Wykres rozmiaru projektów po opublikowaniu.
Źródło: Opracowanie własne.

Najlepszy wynik osiągnęło podejście ASP.NET WebForms. Jest to o tyle zaskakujące, że projekt po publikacji osiągnął ponad połowę mniejszy rozmiar niż projekt przeznaczony do developmentu. Minimalnie słabszy wynik osiągnęło podejście ASP.NET MVC. Tak jak w przypadku poprzedniego badania najsłabiej prezentuje się podejście ASP.NET MVC + AngularJS. Spowodowane jest to dołączeniem do projektu bibliotek AngularJS. Można zaoszczędzić miejsce i skorzystać z wersji udostępnianej przez *Content Delivery Network (CDN)*. Pozwoli to na korzystanie z bibliotek

przechowywanych na zewnętrznych serwerach, a tym samym zmniejszenie obciążenia naszego serwera. Przykładem takiego systemu może być *cdnjs* [32].

4.5. Pozostałe

Biorąc pod uwagę pracę w zespole najlepiej wypada podejście ASP.NET MVC. Nie wymaga instalacji żadnych dodatków. Praca może być podzielona na różne obszary. Poprzez rozdzielenie poszczególnych elementów aplikacji. Jeden programista może zająć się implementacją kontrolera, drugi natomiast w tym samym czasie pracować nad widokami.

WebForms pozwoliło osiągnąć zamierzony cel a przy użyciu odpowiednich dodatków cały proces implementacji był prosty i przyjemny. Jednak jest kilka kwestii, które źle rzutują na to podejście:

- Rozmiar View State – mechanizm, który jest niezbędny w podejściu ASP.NET WebForms, aby utrzymywać stan pomiędzy żądaniami, przerodził się w duże ilości danych przesyłanych pomiędzy klientem a serwerem. Dane te mogą osiągnąć wielkość nawet kilkuset kilobajtów. Transportowane są w jedną i drugą stronę za każdym żądaniem i powodują spowolniony czas odpowiedzi oraz zwiększenie zapotrzebowania przepustowości serwera.
- Cykl życia strony – mechanizm odpowiedzialny za połączenie pomiędzy zdarzeniami po stronie klienta z tymi po stronie serwera. Cykl ten potrafi być bardzo skomplikowany, a to może prowadzić do licznych błędów. Zwłaszcza w przypadku manipulowania hierarchią kontrolek w trakcie wykonywania aplikacji.
- Słaba kontrola nad HTML – kontrolki serwerowe wyświetlają własny kod HTML, który możne znacznie różnic się od tego, który chcemy osiągnąć. ASP.NET generuje skomplikowane wartości ID poszczególnych elementów, a to może utrudnić dostęp do nich z poziomu JavaScript [11].

Powyższe podpunkty stawiają podejście ASP.NET WebForms w bardzo słabym świetle. Dodając do tego nienajlepsze wyniki badań oraz fakt, że WebForms przestaną być częścią ASP.NET 5, podejście to w niedalekiej przyszłości nie powinno być brane pod uwagę. Oczywiście będzie istniała możliwość tworzenia aplikacji pod Framework .NET 4.6 jednak nie pozwoli to na wykorzystanie nowych cech ASP.NET 5 [33].

Dużym minusem AngularJS jest konieczność ręcznej implementacji większości widoków. Brak mechanizmu *scaffolding* ma ogromny wpływ na czas implementacji projektu. W Visual Studio 2015 pojawią się szablony do tworzenia modułów, kontrolerów, dyrektyw i fabryk AngularJS, czyli można powiedzieć, że Angular stanie się niejako częścią frameworka .NET 5, a to stawia go na wyższej pozycji niż WebFormy. Natywne wsparcie sprawi, że programowanie podejściem łączonym ASP.NET MVC + AngularJS będzie łatwiejsze i szybsze [33].

4.6. Wnioski końcowe

Biorąc pod uwagę wnioski zawarte w powyższym rozdziale oraz podsumowania do implementacji poszczególnych podejść można mieć wrażenie, że podejście ASP.NET MVC wypada najlepiej. Należy jednak pamiętać, że dotyczy to omawianego prototypu. W przypadku innego rodzaju systemu wnioski mogłyby być zupełnie inne. Mimo wszystko wydaje się, że to właśnie ASP.NET MVC jest właściwym wyborem. ASP.NEW WebForms ma już swoje lata, a fakt, że od najnowszej wersji .NET Framework nie będzie wspierany sprawia, że jest to najsłabszy wybór. AngularJS jest świetnym rozwiązaniem dla *Single Page Application* jednak nie wszystkie projekty potrzebują tego rozwiązania.

5. Podsumowanie

W wyniku pracy powstały trzy projekty sklepu internetowego z wykorzystaniem różnych podejść do tworzenia aplikacji internetowych. Osiągnięto zamierzony cel, jakim było stworzenie działających prototypów korzystając ze wszystkich podejść. Wszystkie projekty spełniają zakładane wymagania. Zostały przeprowadzone badania mające na celu wskazanie, które podejście jest najlepszym wyborem. Biorąc pod uwagę wyniki badań oraz subiektywne odczucia autora podczas implementacji rozwiązań najlepszym wyborem wydaje się podejście ASP.NET MVC. Dostarcza niezbędnych mechanizmów do łatwej i szybkiej implementacji rozwiązań. Stworzona aplikacja nie zajmuje wiele miejsca na serwerze, a rozmiar stron po stronie użytkownika jest zadowalający. Na dzień pisania pracy wydaje się, że jest to idealne rozwiązanie. Należy jednak pamiętać, że badania jak i wybór rozwiązania były robione na podstawie z góry określonych wymagań. W przypadku innego rodzaju projektu wyniki mogą być zupełnie inne.

Z racji tego, że wszystkie podejścia pozwalały uzyskać zamierzony efekt nie należy odrzucać żadnego z możliwych rozwiązań. Mimo wszystko trzeba patrzeć na projekt z szerszej perspektywy. Wizja braku wsparcia przez Microsoft podejścia ASP.NET WebForms powinna być jasnym sygnałem, że ta opcja nie jest najlepszym rozwiązaniem. Natomiast natywne wsparcie AngularJS w programie Visual Studio powinno przynieść szereg mechanizmów ułatwiających implementacje projektów.

6. Bibliografia

[1]. *Entity Framework Overview*, https://msdn.microsoft.com/en-us/library/vstudio/bb399567%28v=vs.100%29.aspx, 20.05.2015 r.

[2]. *Database First development with Entity Framework*, http://www.entityframeworktutorial.net/images/EF5/databasefirst.png, 20.05.2015 r.

[3]. *Bootstrap*, http://getbootstrap.com/, 20.05.2015 r.

[4]. *Bootstrap in the Visual Studio 2013 web project templates*, http://www.asp.net/visual-studio/overview/2013/creating-web-projects-in-visual-studio, 20.05.2015 r.

[5]. Pranav Rastogi, *Announcing RTM of ASP.NET Identity 2.0.0*, http://blogs.msdn.com/b/webdev/archive/2014/03/20/test-announcing-rtm-of-asp-net-identity-2-0-0.aspx, 20.05.2015 r.

[6]. Pranav Rastogi, *ASP.NET Identity 2.2.1*, http://blogs.msdn.com/b/webdev/archive/2015/04/07/asp-net-identity-2-2-1.aspx, 20.05.2015 r.

[7]. Pranav Rastogi, Rick Anderson, Tom Dykstra, Jon Galloway, *Introduction to ASP.NET Identity*, http://www.asp.net/identity/overview/getting-started/introduction-to-aspnet-identity, 20.05.2015 r.

[8]. Jess Chadwick, Todd Snyder, Hrusikesh Panda, *Programing ASP.NET MVC 4*, ISBN 978-1-449-32031-7, data wydania 14.09.2012 r.

[9]. Kalyan Bandarupalli, *ASP.NET 4.5 New Features Overview*, http://www.techbubbles.com/aspnet/asp-net-4-5-new-features-overview/, 20.05.2015 r.

[10]. Tony Northrup, Mike Snell, *Exam 70-515: Web Applications Development with Microsoft .NET Framework 4*, ISBN: 978-0-7356-2740-6, data wydania 23.12.2010 r.

[11]. Adam Freeman, *Pro ASP.NET MVC 4*, ISBN 978-1-4302-4236-9, data wydania: 27.12.2012 r.

[12]. Dino Esposito, *Programming Microsoft ASP.NET 4*, ISBN: 978-0-7356-4338-3, data wydania 02.2014 r.

[13]. *ASP.NET Page Life Cycle Overview*, https://msdn.microsoft.com/en-us/library/ms178472(v=vs.100).aspx, 20.05.2015 r.

[14]. *ASP.NET Page Life Cycle Diagram*, http://blogs.msdn.com/b/aspnetue/archive/2010/01/14/asp-net-page-life-cycle-diagram.aspx, 20.05.2015 r.

[15]. *Understanding ASP.NET View State*, https://msdn.microsoft.com/en-us/library/ms972976.aspx, 20.05.2015 r.

[16]. http://pl.wikipedia.org/wiki/AngularJS, 20.05.2015 r.

[17]. Shyam Seshadri, Brad Green *AngularJS Up & Running*, ISBN 978-491-90194-6, data wydania 05.09.2014 r.

[18]. *AngularJS Controller Tutorial With Example*, http://img.viralpatel.net/2013/06/angularjs-app-modules-controllers-view.png, 20.05.2015 r.

[19]. http://www.w3schools.com/angular/default.asp, 20.05.2015 r.

[20]. *AngularJS Routing And Views Tutorial With Example*, http://img.viralpatel.net/2013/07/angularjs-routing-view-controller.png, 20.05.2015 r.

[21]. *AngularJS API Docs*, https://docs.angularjs.org/api/ng/service/$http, 20.05.2015 r.

[22]. http://en.wikipedia.org/wiki/TypeScript, 20.05.2015 r.

[23]. *Angular 2.0 built on TypeScript*, http://blogs.msdn.com/b/typescript/archive/2015/03/05/angular-2-0-built-on-typescript.aspx, 20.05.2015 r.

[24]. *SQL script for creating an ASP.NET Identity Database*, http://www.codeproject.com/Tips/677279/SQL-script-for-creating-an-ASP-NET-Identity-Databa, 20.05.2015 r.

[25]. *Override the Default Scaffold Templates*, https://msdn.microsoft.com/en-us/magazine/dn745864.aspx, 20.05.2015 r.

[26]. *Introduction to ASP.NET Identity*, http://www.asp.net/identity/overview/getting-started/introduction-to-aspnet-identity, 20.05.2015 r.

[27]. *Web Forms Scaffolding*, https://visualstudiogallery.msdn.microsoft.com/a6c3614f-83be-4749-afbc-8da394b6ea86, 20.05.2015 r.

[28]. *ASP.NET login controls overview*, https://msdn.microsoft.com/en-us/library/cc295303.aspx, 20.05.2015 r.

[29]. Scott Allen, *Routing with ASP.NET Web Forms*, https://msdn.microsoft.com/en-us/magazine/dd347546.aspx, 20.05.2015 r.

[30]. Scott Hanselman, *Introducing ASP.NET FriendlyUrls*, http://www.hanselman.com/blog/IntroducingASPNETFriendlyUrlsCleanerURLsEasierRoutingAndMobileViewsForASPNETWebForms.aspx, 20.05.2015 r.

[31]. *AngularJS Interceptors – Globally handle 401*, http://blog.thesparktree.com/post/75952317665/angularjs-interceptors-globally-handle-401-and, 20.05.2015 r.

[32]. cdnjs, *angular.js*, http://cdnjs.com/libraries/angular.js/, 20.05.2015 r.

[33]. Stephan Walther, *Top 10 Changes in ASP.NET 5 and MVC6*, http://stephenwalther.com/archive/2015/02/24/top-10-changes-in-asp-net-5-and-mvc-6, 20.05.2015 r.

7. Wykaz rysunków

8. Wykaz listingów