**POLITECHNIKA KRAKOWSKA im. T. Kościuszki**
Wydział Mechaniczny
Instytut Informatyki Stosowanej

Kierunek studiów : Informatyka
Specjalność : Inżynieria oprogramowania

STUDIA STACJONARNE

# PRACA DYPLOMOWA
## INŻYNIERSKA

**Karol Trybulec**

Internetowy system zarządzania domami studenckimi
z zastosowaniem modelu MVC po stronie klienta i serwera

Online student houses management system using MVC model in client and server architecture

Promotor:
dr inż. **Wojciech Czyżycki**

Kraków, rok akad. 2015/2016

**Autor pracy:** **Karol Trybulec ……..**

**Nr pracy:** **…….**

## OŚWIADCZENIE O SAMODZIELNYM WYKONANIU PRACY DYPLOMOWEJ

Oświadczam, że przedkładana przeze mnie praca dyplomowa magisterska/inżynierska* została napisana przeze mnie samodzielnie. Jednocześnie oświadczam, że ww. praca:

1) nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem/am* w sposób niedozwolony,

2) nie była wcześniej podstawą żadnej innej procedury związanej z nadawaniem tytułów zawodowych, stopni lub tytułów naukowych.

Jednocześnie przyjmuję do wiadomości, że w przypadku stwierdzenia popełnienia przeze mnie czynu polegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzej pracy, lub ustalenia naukowego, właściwy organ stwierdzi nieważność postępowania w sprawie nadania mi tytułu zawodowego (art. 193 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym, Dz.U. z 2012 r. poz. 572, z późn. zm.).

……
data i podpis

* niepotrzebne skreślić

**Uzgodniona ocena pracy: ……..**

**……...** **.....**

podpis promotora podpis recenzenta

# SPIS TREŚCI

# 1. Cel i zakres pracy

Celem niniejszej pracy jest stworzenie systemu internetowego do zarządzania domami studenckimi. Projekt powinien wykorzystywać najnowsze technologie i rozwiązania jakie obecnie panują na rynku. Pomysł na temat narodził się z powodu dostrzeżenia niedoskonałości aktualnych rozwiązań oraz bardzo małego wyboru panującego na rynku.

Interfejs użytkownika powinien skupiać się na responsywności i dostosowaniu aplikacji do wszelkich urządzeń mobilnych. Ten aspekt jest szczególnie ważny, ponieważ obecny poziom informatyzacji jest ogromny i ciągle rośnie. Urządzenia mobilne stanowią coraz większy odsetek urządzeń podłączonych do sieci internetowej. Można śmiało założyć, że trend rosnący zostanie utrzymany. System powinien być przyjazny dla osób niepełnosprawnych, które mogą być potencjalnymi użytkownikami systemu.

Ważnym celem tej pracy jest także zaprezentowanie wzorca MVC w praktyce. Użycie wzorca MVC nie jest żadną nowością w warstwie serwera, jednak warstwa prezentacji przeważnie nie jest w żaden sposób uporządkowana. Projekty gdzie kod HTML i kod JavaScript przeplatają się z treścią, prędzej czy później zaczynają sprawiać problemy w ich dalszej rozbudowie. Wtedy dodatnie jakiejkolwiek funkcjonalności niesie ze sobą szereg komplikacji. Z roku na rok rozbudowa i utrzymanie takich projektów bywa coraz bardziej kłopotliwe i problematyczne, dowodzi to zawodności zastosowania pojedynczego modelu MVC i konieczności wymyślenia lepszego rozwiązania.

W niniejszej pracy użyty zostanie nowatorski sposób na całkowite oddzielenie warstwy logiki od warstwy prezentacji, posługując się wzorcem MVC po stronie serwera i symulując jego działanie po stronie klienta. Rola prezentacji danych zostanie całkowicie wyeliminowana z serwera aplikacji, całość zostaje przeniesiona na stronę klienta.

Takowy zabieg doprowadza do sytuacji, gdzie serwer aplikacji pełni jedynie rolę szyny komunikacyjnej, pośrednika pomiędzy bazą danych a klientem.

# 2. Wstęp

XXI wiek jest przełomowy pod względem rozwoju komputerów i smartfonów. Zaobserwować można ciekawe zjawisko: ludzie w podeszłym wieku są osobami, które w młodości nie miały telewizorów, a w ciągu ich życia zdążyły powstać komputery, później telefony stacjonarne a następnie smartfony. Ludzie, którzy na własną rękę nie interesowali się komputerami znajdują się w złej sytuacji. Informatyka jest tak wszechobecna, że nieznajomość działania prostych urządzeń mobilnych potrafi bardzo zaszkodzić. Na porządku dziennym są informacje o kradzieżach, przestępstwach internetowych czy wyłudzeniach.

Taki sam problem jest ze zdolnościami ludzi do wykorzystania narzędzi jakie dają komputery i sieć internet. W wielu małych firmach, które są sercem polskiej gospodarki, wszystko ciągle załatwiane jest za pomocą kartek i papierowych dokumentów. W firmach zajmujących się transportem osób, zatrudniani są pracownicy którzy przez cały dzień odbierają telefony notując ilość zarezerwowanych miejsc na dane godziny.

W małych jednostkach administracyjnych takich jak akademiki szkół wyższych, rekrutacja ciągle polega na przyniesieniu pliku kartek we wskazane miejsce w ciągu jakiegoś terminu. Temat tej pracy narodził się właśnie w momencie rozważań na temat problemów obecnego rynku. Wielu przedsiębiorców nie zdaje sobie sprawy, że prosty system informatyczny jest w stanie zwiększyć ich zyski, pozwoli zredukować liczbę pracowników a przede wszystkim jest bezpieczniejszy i bardziej skuteczny.

Zastanawiający jest fakt, czy niechęć do informatyzacji drobnych biznesów wynika z faktu braku wiedzy informatycznej, strachu przed trzymaniem danych w postaci cyfrowej, czy z faktu cen jakie obecnie panują na rynku. Chłodne przemyślenia podpowiadają, że cena choć nie mała, nie powinna odstraszać przedsiębiorców. Koszt realizacji systemu dla małej firmy zwróciłby się w kilka miesięcy w przypadku najmniejszej redukcji zatrudnień jaka mogłaby zostać osiągnięta.

# 3. Analiza istniejących rozwiązań

Na rynku nie ma wielu rozwiązań oferujących systemy do zarządzania domami akademickimi. Te, które zostały odnalezione są systemami płatnymi bez jakiejkolwiek możliwości przetestowania wersji próbnej. Są to produkty małych działalności gospodarczych, bez zaawansowanego zaplecza marketingowego. Prawdopodobnie to jest powodem braku jakichkolwiek wersji testowych.

## 3.1. Wirtualny dziekanat Politechniki Krakowskiej

Pod uwagę wzięty został moduł wirtualnego dziekanatu Politechniki Krakowskiej odpowiedzialny za akademiki. Został poddany analizie, ponieważ jest jedynym podsystemem, który był możliwy do fizycznego przetestowania.

Pod względem funkcjonalnym moduł rezerwacji akademików wygląda bardzo dobrze. Studentowi są udostępnione wszelkie niezbędne funkcje potrzebne do rezerwacji miejsca, podglądu rezerwacji, wyświetlenia informacji o zakwaterowaniu a także o historii zameldowania.

Wadą systemu zarządzania akademikami połączonego z wirtualnym dziekanatem jest z pewnością fakt zależności od konkretnej platformy. Całość działa jako funkcjonalność większego systemu a nie jako odrębny byt. Z tego powodu niemożliwe okazuje się złożenie rezerwacji w przypadku studentów z innych uczelni, bądź ludzi, którzy ukończyli studia.

System wypadł fatalnie także pod względem testów interfejsu. Został napisany w technologii HTML bez zdefiniowanego tzw. *doctype*, więc nie można określić jego wersji. Całość zbudowana jest na tabelkach, które nie naddają się do budowania treści stron.

Tabele służą do reprezentacji danych tabelarycznych, wszelka próba opierania na nich szkieletu strony może skończyć się wieloma błędami:

- problem z parsowaniem tabel przez urządzenia mobilne
- określają sztywny szkielet dokumentu, nie da się na niego wpłynąć arkuszem stylów CSS
- niemożność zastosowania responsywnego szablonu za pomocą *media queries*
- współczynnik kodu HTML do treści strony jest bardzo duży, co powoduje wolniejsze ładowanie i problemy z pozycjonowaniem
- wszelkie zmiany w układzie szablonu są później bardzo trudne bądź niemożliwe

Widoczne jest, że zostały podjęte próby wprowadzenia responsywności widoku. Szerokości tabel nie są ustalone na stałe tylko podane w wartościach procentowych więc dopasowują się do kształtu okna. Niestety na urządzeniu mobilnym strona prezentuje się bardzo źle, ponieważ nie został dodany specjalny znacznik *meta* informujący urządzenie aby próbowały wymusić responsywność.

Bardzo negatywnie moduł do zarządzania akademikami prezentuje się pod względem przystosowania interfejsu dla osób niepełnosprawnych lub niedowidzących. Nie spełnia nawet podstawowych norm kolorystycznych dokumentu WCAG, który jest zbiorem zadań mówiących

o tym jak budować przyjazne strony internetowe (szczególnie dla osób niepełnosprawnych). Spełnienie kryterium WCAG AA wymaga kontrastu pomiędzy tłem strony a małą czcionką w stosunku przynajmniej 4.5:1, a w analizowanym systemie jest to stosunek 3.07:1.

## 3.2. Uniwersalny system rezerwacji pokoi hotres.pl

Na rynku można znaleźć systemy przystosowane do obsługi wszelkiego rodzaju rezerwacji noclegów i pokoi. Są to systemy uniwersalne a przez to bardzo rozbudowane. Przy odpowiedniej konfiguracji udałoby się przystosować uniwersalny system do obsługi domu studenckiego.

System hotres.pl stworzony został przy użyciu najnowszych technologii. Doskonale prezentuje się na wszelkich urządzeniach mobilnych i desktopowych. Analizując system w oczy od razu rzuca się profesjonalny interfejs użytkownika, który jest bardzo intuicyjny i dopracowany. Analizując wersję demo można zauważyć, że do pełni idealnego interfejsu brakuje kontrolek powiększania tekstu, zmiany kolorystyki strony lub wielojęzyczności. Autorzy systemu kompletnie nie skupili się na dostosowaniu interfejsu dla osób niepełnosprawnych i niedowidzących. Mimo, że interfejs spełnia wymagania użyteczności (tzw. *user experience, UX*), to niestety nie spełnia wymagań wizualnych.

Kolorystyka linków w menu poziomym nie spełnia norm AAA ani AA dokumentu WCAG 2.0. Czcionka poziomego menu jest bardzo mała, a stosunek kontrastu koloru liter do koloru tła strony wynosi 2.32:1.

## 3.3. Aplikacja desktopowa i mobilna bed-booking.com

Bed-booking.com jest kolejnym uniwersalnym systemem służącym do rezerwacji noclegów. To co odróżnia go od innych rozwiązań to fakt, że udostępnia aplikację dedykowaną na telefony komórkowe i desktopy.

Analizując aplikację można dojść do wniosku, że pisanie dedykowanej aplikacji zależnej od konkretnej platformy rodzi więcej problemów niż zysków. Przez cały cykl używania oprogramowania musi stać za nią szereg programistów, którzy na bieżąco będą w stanie zapewniać kompatybilność aplikacji ze wszelkimi nowymi systemami i urządzeniami. Architektura systemu oparta na aplikacji internetowej wykorzystującej protokół HTTP wydaje się w tym wypadku bardziej uniwersalna i skuteczna. Jest po prostu niezależna od platformy na jakiej pracuje. W końcu każda platforma i architektura wspiera dostęp do sieci internetowej i wspiera chociaż jeden typ przeglądarki internetowej.

Wyżej wymieniony produkt udostępnia kalendarz, możliwy do wstawienia na dowolnej stronie, jednak kalendarz jest zintegrowany z aplikacją na telefonie. Z tego powodu użycie kalendarza nie eliminuje problemu konieczności wykorzystania aplikacji mobilnej lub desktopowej.

Analiza funkcjonalna i architektoniczna na dzień dzisiejszy nie jest możliwa, z powodu braku darmowej wersji próbnej produktu.

## 3.4. Podsumowanie

Problem obsługi domów studenckich jest problemem bardzo złożonym i specjalistycznym. Analizując zagadnienie można natrafić na wiele problemów charakterystycznych tylko dla tej dziedziny.

Gotowe uniwersalne systemy służące do obsługi imprez masowych, hoteli i noclegów są zbyt złożone a przez to ciężkie. Dodatkowo nie zostały w nich przewidziane pewne sytuacje skrajne takie jak np. pytanie studenta o numer legitymacji podczas rejestracji.

Rezerwacje noclegów dla osób prywatnych działają odmiennie i na innych zasadach niż zakwaterowywanie studentów w akademikach. Pojawia się tutaj pytanie, czy przerabianie gotowego systemu byłoby grą wartą świeczki? Czy lepszym rozwiązaniem byłoby zamówienie dedykowanego systemu.

# 4. Koncepcja systemu

Do opracowania koncepcji systemu, wzięte były pod uwagę przede wszystkim wady spotkane w innych projektach poddanych analizie. Opracowanie koncepcji systemu pozwala dobrać odpowiednie narzędzia programistyczne w stosunku do problemów i celów jakie zostaną obrane.

## 4.1. Specyfikacja wymagań systemu

Odpowiednio sformułowana specyfikacja wymagań tworzy jasne wskazówki dla wykonawcy systemu. Jest to język pośredni pomiędzy klientem zamawiającym system a jego wykonawcą. Odpowiednio przygotowana specyfikacja wymagań powinna przede wszystkim usprawnić komunikację.

### 4.1.1. Wymagania funkcjonalne

Wymagania funkcjonalne powinny mówić wprost jakie funkcje zawiera system w celu zidentyfikowania jego zachowania. Każde wymaganie funkcjonalne wyróżnia unikalna nazwa, opis, komplet danych wejściowych i wyjściowych oraz wymagania.

| Nazwa funkcji | **Logowanie do systemu** |
|---|---|
| Opis funkcji | Umożliwia autoryzację użytkownika w systemie. |
| Dane wejściowe | Dane logowania |
| Dane wyjściowe | Potwierdzenie o zalogowaniu lub jego brak. |
| Wymagania | Posiadanie konta w systemie. |

| Nazwa funkcji | **Rejestracja użytkownika** |
|---|---|
| Opis funkcji | Umożliwia późniejszą autoryzację i logowanie. |
| Dane wejściowe | Dane personalne użytkownika |
| Dane wyjściowe | Potwierdzenie o rejestracji lub jego brak. |
| Wymagania | Unikalność wprowadzonych danych. |

| Nazwa funkcji | **Rezerwacja akademika** |
|---|---|
| Opis funkcji | Umożliwia złożenie rezerwacji o miejsce w akademiku. |
| Dane wejściowe | Identyfikator użytkownika. |
| Dane wyjściowe | Potwierdzenie złożenia rezerwacji. |
| Wymagania | Bycie zalogowanym; wolne miejsca w akademiku. |

| Nazwa funkcji | **Podgląd rezerwacji** |
|---|---|
| Opis funkcji | Umożliwia sprawdzenie stanu rezerwacji akademika. |
| Dane wejściowe | Identyfikator użytkownika. |
| Dane wyjściowe | Informacje o stanie rezerwacji. |
| Wymagania | Bycie zalogowanym; wcześniejsze złożenie rezerwacji. |

| Nazwa funkcji | **Oczekujące płatności** |
|---|---|
| Opis funkcji | Sprawdzenie listy oczekujących płatności dla danego użytkownika. |
| Dane wejściowe | Identyfikator użytkownika. |
| Dane wyjściowe | Lista oczekujących płatności. |
| Wymagania | Bycie zalogowanym; bycie mieszkańcem domu studenckiego. |

| Nazwa funkcji | **Rezygnacja z akademika** |
|---|---|
| Opis funkcji | Zrezygnowanie z miejsca w domu studenckim. |
| Dane wejściowe | Identyfikator użytkownika. |
| Dane wyjściowe | Potwierdzenie przyjęcia rezygnacji. |
| Wymagania | Bycie zalogowanym; bycie mieszkańcem domu studenckiego. |

| Nazwa funkcji | **Wylogowanie** |
|---|---|
| Opis funkcji | Wylogowanie z konta i zniszczenie sesji użytkownika. |
| Dane wejściowe | Identyfikator użytkownika. |
| Dane wyjściowe | Potwierdzenie o udanym wylogowaniu. |
| Wymagania | Bycie zalogowanym |

| Nazwa funkcji | **Lista wniosków** |
|---|---|
| Opis funkcji | Lista wniosków o rezerwację miejsca w akademiku. |
| Dane wejściowe | brak |
| Dane wyjściowe | Lista złożonych wniosków o rezerwację. |
| Wymagania | Bycie zalogowanym z rangą administrator. |

| Nazwa funkcji | **Przyjęcie wniosku – przypisanie studenta do pokoju** |
|---|---|
| Opis funkcji | Pozytywne rozpatrzenie rezerwacji i przypisanie studenta do pokoju. |
| Dane wejściowe | Wybranie łóżka, pokoju, piętra i budynku. |
| Dane wyjściowe | Potwierdzenie przyjęcia wniosku. |
| Wymagania | Bycie zalogowanym z rangą administrator; oczekujący wniosek |

| Nazwa funkcji | **Lista mieszkańców** |
|---|---|
| Opis funkcji | Wyświetlenie aktualnej listy wszystkich mieszkańców. |
| Dane wejściowe | brak |
| Dane wyjściowe | Lista mieszkańców akademików. |
| Wymagania | Bycie zalogowanym z rangą administrator |

| Nazwa funkcji | **Wyrzucenie mieszkańca** |
|---|---|
| Opis funkcji | Umożliwia natychmiastowe wykwaterowanie danego mieszkańca. |
| Dane wejściowe | Określony mieszkaniec domu studenckiego. |
| Dane wyjściowe | Potwierdzenie o powodzeniu. |
| Wymagania | Bycie zalogowanym z rangą administrator; istniejący mieszkańcy. |

| Nazwa funkcji | **Akceptacja oczekującej płatności** |
|---|---|
| Opis funkcji | Umożliwia zaakceptowanie danej płatności. |
| Dane wejściowe | Mieszkaniec, któremu akceptujemy płatność. |
| Dane wyjściowe | Potwierdzenie o powodzeniu. |
| Wymagania | Bycie zalogowanym z rangą administrator; istniejące płatności. |

### 4.1.2. Wymagania niefunkcjonalne

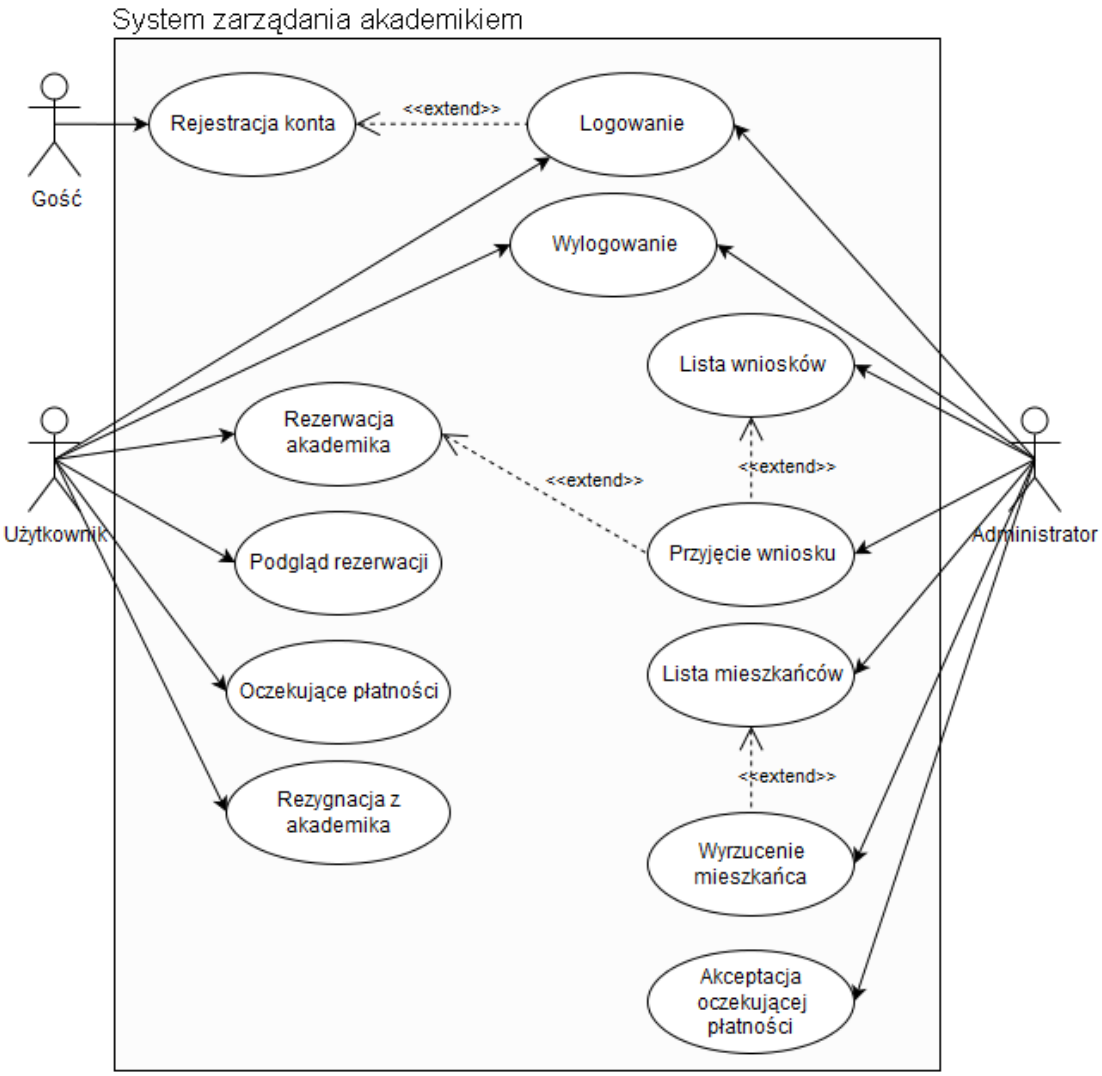
Wymagania niefunkcjonalne określają wszelkie uwagi i wskazówki dotyczące działania systemu. Nie dotyczą jego funkcjonalności bezpośrednio a jedynie mówią o sposobie jego działania, dotyczą systemu jako całości. Opisują kwestie takie jak dostępność, wydajność, niezawodność, bezpieczeństwo, odporność. Wymagania niefunkcjonalne przedstawia tabela 2.1.

| | |
|---|---|
| Dostępność | – Dostępny 24h na dobę<br>– Maksymalnie 1-2 dni niedostępności w miesiącu<br>– Dostęp z wielu urządzeń także mobilnych |
| Wydajność | – Nielimitowana ilość połączeń<br>– Nielimitowana ilość zapytań<br>– Nielimitowana ilość transakcji SQL<br>– Równoczesny dostęp przynajmniej 20 użytkowników<br>– Niskie zużycie transferu podczas łączenia się z telefonu |
| Niezawodność | – Długi czas pomiędzy awariami, nie częściej niż raz w roku<br>– Odzyskiwanie danych z kopii zapasowej w czasie 5-10minut<br>– Cotygodniowe raporty o działaniu systemu dla obsługi tech. |
| Bezpieczeństwo | – Automatyczne zrzuty bazy danych co 2h (kopia zapasowa)<br>– Szyfrowanie wrażliwych danych użytkowników algorytmami niesymetrycznymi<br>– Połączenie przez protokół HTTPS z certyfikatem |
| Odporność | – Natychmiastowy zapis wszelkich danych do bazy danych w celu zabezpieczenia przed brakiem zasilania |

Tabela 2.1. Wymagania niefunkcjonalne projektu.

### 4.1.3. Diagram przypadków użycia

Diagram przypadków użycia jest to diagram obrazujący wymagania funkcjonalne systemu. Daje on spojrzenie na funkcje systemu udostępnione na zewnątrz z podziałem na aktorów, przedstawiających realnych użytkowników systemu. Pojęcie to zostało szeroko opisane w literaturze [2]. Diagram przypadków użycia dla systemu przedstawia rysunek 3.1.



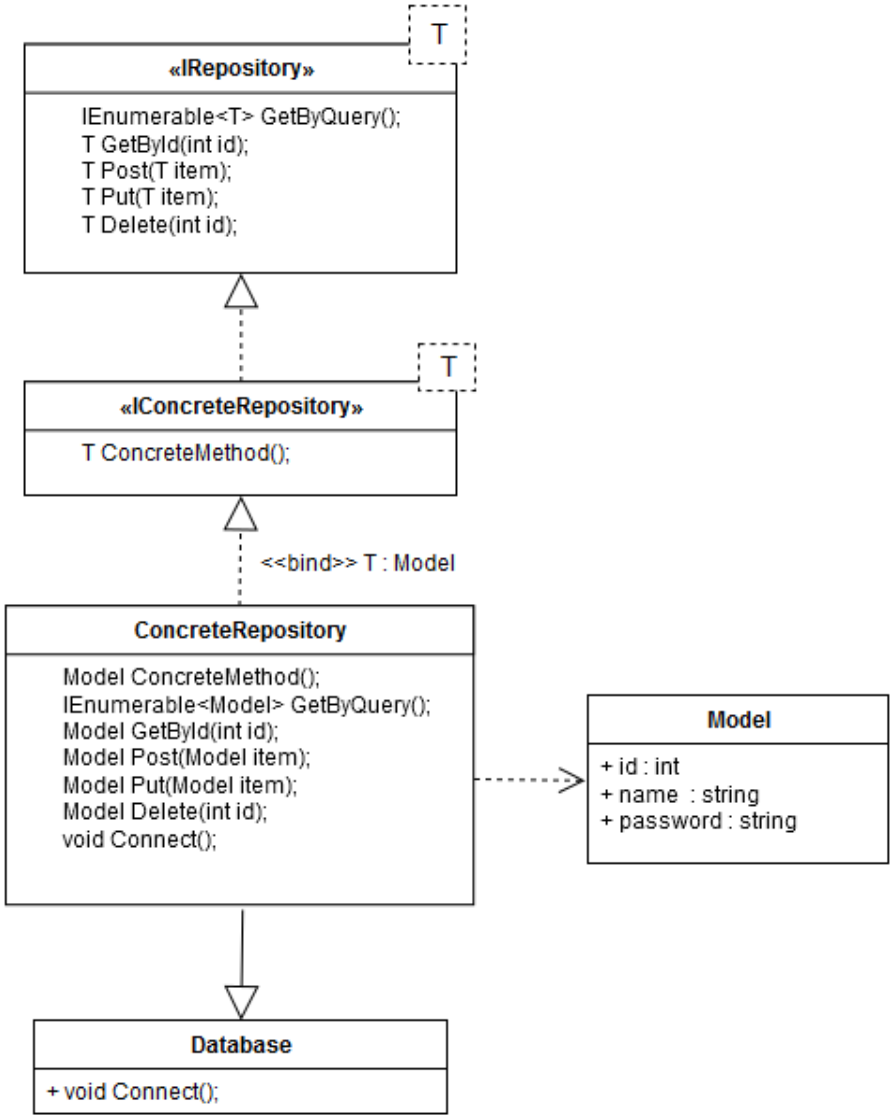

Rys. 3.1. Diagram przypadków użycia systemu

### 4.1.4. Diagram klas UML

Język UML jest językiem służącym do graficznego modelowania różnego rodzaju aplikacji oraz systemów. Diagramy klas UML przedstawiają obiektową strukturę fragmentu systemu oraz związki między tymi obiektami. Definiują jakie metody i jakie pola powinna zawierać dana klasa. Diagram klas pokazuje typy obiektów bez ich instancji, jest o tym napisane w literaturze [1]. Należy pamiętać, że diagramy klas UML są tylko diagramami poglądowymi, nie muszą oddawać prawdziwej implementacji problemu w skali rzeczywistej.

#### 4.1.4.1. Diagram klas wzorca repozytorium

Na poniższym rysunku 1.1 zaprezentowany jest diagram klas UML wzorca repozytorium.




Rys. 1.1. Diagram klas UML wzorca repozytorium

## 4.2. Użyte technologie

Projekt został stworzony przy użyciu najnowszych technologii dostępnych na rynku. Przy ich wyborze brany pod uwagę był procentowy udział w rynku oraz trend popularności. Korzystanie z najbardziej aktualnych technologii jest w informatyce bardzo ważne, ponieważ jest to prężnie rozwijająca się gałąź przemysłu. Korzystanie z rynkowych standardów daje twórcom systemów komputerowych gwarancję konkurencyjności.

Technologie stare oraz niewspierane zawsze z biegiem czasu zaczynają sprawiać nieprzewidziane problemy np. z kompatybilnością z nowymi platformami.

### 4.2.1. ASP.NET MVC 5

Biblioteka .NET jest tworzona przez Microsoft. Przy jej tworzeniu Microsoft opierał się na błędach i sukcesach wyniesionych z języka Java. Tak jak w przypadku Javy aplikacje kompilowane w środowisku .NET działają w wewnętrznej piaskownicy i są kompilowane dwuetapowo.

W starych językach takich jak C++ kompilator tworzy plik wykonywalny zbudowany z kodu maszynowego, jest on bezpośrednio związany z architekturą, na której zostanie skompilowany. Programy napisane w języku C++ są zbudowane z kodu zarządzalnego a więc podlegają normalnym procedurom systemu. Dla procesu alokowana jest pamięć, tworzona jest sterta dla procesu i stos dla każdego z wątków osobno. Stosem i stertą zarządza kernel systemu i wszystkie skomplikowane funkcje (*alloc, malloc, free* itd.) wystawione wprost z *WindowsAPI*. Programista sam musi dbać o porządek w pamięci aplikacji, szczególnie uważać na błędy typu wyciek pamięci (*memory leak*).

Na platformie .NET programy kompilowane są do kodu IL, jest to kod pośredni, nie da się go uruchomić na żadnej architekturze. Próba uruchomienia programu skompilowanego do języka IL oznacza „kompilację w czasie realnym" do języka JIT (z ang. *Just-in-time)*. Program taki zostaje przetworzony i uruchomiony przez bibliotekę .NET w jej wewnętrznej piaskownicy, niezależnie od architektury na jakim ta piaskownica została uruchomiona. Jest to jedna z największych zalet języków wysokiego poziomu nad językami starszej generacji takimi jak C++.

ASP.NET jest zbiorem technologii podległych pod bibliotekę .NET umożliwiających tworzenie stron oraz usług internetowych. Jej nowszą odmianą jest ASP.NET MVC, który narzuca na programistę stosowanie wzorca *Model-View-Controller*. Potęga tej technologii polega na szybkości tworzonych rozwiązań oraz na wsparciu jakie zapewnia nam technologia ASP.NET. Programista może liczyć na setki gotowych klas i rozwiązań, całość oparta jest na interfejsach i jest bardzo łatwo skalowalna. Systemy stworzone w ASP.NET bardzo łatwo współpracują z usługami sieciowymi. Sam ASP.NET umożliwia wiele gotowych, zaimplementowanych mechanizmów gotowych do wykorzystania takich jak: silnik połączenia do bazy danych (ADO.NET), mechanizm wspierania wersji językowych, mechanizmy autoryzacji, ORM itd.

Architektura .NET została bardzo dobrze przedstawiona i opisana w książce [3].

### 4.2.2. AngularJS 1.4

AngularJS jest darmową platformą programistyczną języka skryptowego JavaScript rozwijaną przez firmę Google, jest szeroko opisana w literaturze [4]. Stanowi potężne narzędzie służące do budowy aplikacji internetowych. Fenomenem działania Angulara i jego główną zaletą jest tzw. mechanizm *two-way data binding*. Dzięki temu mechanizmowi wszelkie zmiany naniesione w widoku zostają automatycznie przekazane do modelu, zależność występuje też w drugą stronę. Efektem tego mechanizmu jest natychmiastowa asynchroniczna synchronizacja wszelkich danych.

AngularJS służy do tworzenia aplikacji w pełni asynchronicznych. Raz załadowane konkretne moduły Angulara nie wymagają jakichkolwiek dodatkowych zapytań do serwera. Nawet fakt użycia złożonych routingów uruchamiających odrębne kontrolery i odrębne widoki, nie wymaga żadnego przeładowania. To wszystko daje użytkownikowi efekt płynności i lekkości aplikacji.

Opisywana platforma zdobywa coraz większą popularność i na dobre zadomowiła się na rynku wytwarzania oprogramowania jako bardzo silne, a co najważniejsze darmowe, narzędzie. Służy do budowania aplikacji po stronie klienta a więc skupia się na warstwie prezentacji.

Mimo ogromnych możliwości jakie posiada, serwer aplikacji musi zostać napisany w innej technologii. W moim przypadku jest to oczywiście ASP.NET MVC.

Wadą Angulara jest to, że nie naddaje się do każdego projektu. Doskonale sprawdza się w projektach o skomplikowanym interfejsie użytkownika, gdzie niezbędne jest silne rozróżnienie warstwy prezentacji od warstwy biznesowej. AngularJS symuluje pośrednio działanie wzorca MVC więc znakomicie pomaga oddzielić kod HTML od logiki. Dzieje się to poprzez rozszerzenie funkcjonalności zwykłego HTMLa przez dostarczenie różnego rodzaju dyrektyw i wczytywanie danych z zewnętrznych modeli.

Ponadto AngularJS nie naddaje się do budowania aplikacji, które będą pozycjonowane w sieci internet. Kod strony, która zbudowana jest w tej technologii to hierarchia przeróżnych dyrektyw. Dyrektywa zostaje zastąpiona wartościową treścią dopiero w momencie uruchomienia silnika Angulara a dokładniej pierwszego uruchomienia pętli *digest cycle*.

Biorąc pod uwagę dwie powyższe wady AngularJS może być idealnym rozwiązaniem do zbudowania warstwy prezentacji złożonego systemu. Przeważnie jest wykorzystywany do wszelakich systemów bankowych, paneli administratorskich, paneli zarządzania itd. Chcąc użyć go na zwykłej stronie typu blog, powinniśmy raczej rozważyć użycie zwykłego JavaScript lub JQuery.

### 4.2.3. Oracle Database

Baza danych dostarczona przez Oracle jest najbardziej popularnym silnikiem baz danych na świecie. Naddaje się zarówno do małych jak i dużych projektów przechowujących setki terabajtów danych. Jej oczywistym konkurentem na rynku jest silnik dostarczony przez Microsoft czyli SQL Serwer. Obydwa rozwiązania mają podobne możliwości, mimo tego w projekcie została użyta baza danych Oracla. Wiele informacji na temat tejże bazy dostarcza literatura [5].

Oracle wyznaczył swoje standardy, od których ciężko się odzwyczaić. Inżynierowie Oracla stworzyli produkt, który pod wieloma względami jest lepszy od rozwiązania Microsoftu.

Bazy danych Oracle posiadają lepszą i wygodniejszą kontrolę transakcji. Każde połączenie jest osobną transakcją, której efekty można łatwo cofnąć w przypadku niepowodzenia pojedynczej operacji. W bazach danych SQL Serwer trzeba operować dodatkowymi rozkazami bezpośrednio w ciele procedury, co jest ewidentnie mniej wygodne.

Ważnym aspektem przemawiającym na korzyść Oracle jest możliwość wygodnego grupowania procedur w tzw. paczki. Jest to mechanizm niezwykle skuteczny, znajdujący szerokie zastosowanie we wszystkich dużych projektach. Oprócz tego, że zapewnia porządek i bardzo dobrą organizację, umożliwia wystawianie danej paczki na zewnątrz, jako funkcjonalny interfejs dla innych systemów. Można to osiągnąć np. poprzez utworzenie db-linków pomiędzy dwoma różnymi bazami.

Opisywana baza danych potrafi współpracować z wieloma innymi technologiami w tym także z platformą .NET. Jest znakomicie obsługiwana przez silnik ADO.NET.

### 4.2.4. Bootstrap3, HTML5 i CSS3

Czasy tworzenia stron za pomocą czystego HTMLa kończą się. Obecnym niekwestionowanym standardem jest używanie biblioteki Bootstrap. Jest to gotowy plik arkusza

stylów, który zawiera zestaw własnych zdefiniowanych klas. Projektując strony należy znać i umieć korzystać z Bootstrapa. Technologia ta została opisana w literaturze [6] a także jej najlepsze połączenie z AngularJS [7].

Korzystanie z Bootstrapa niesie ze sobą szereg korzyści. Jego powstanie stało się początkiem ustandaryzowania tworzenia stron internetowych. Każdy szablon stworzony w tej technologii jest zbudowany na tzw. siatce (grid). Główną korzyścią płynącą z tego faktu jest całkowita responsywność na każdym urządzeniu mobilnym niezależnie od jego rozdzielczości.

Responsywność Bootstrapa działa na zasadzie kombinacji *media queries* będących funkcjonalnością CSS3. Webmaster nie jest już zmuszony tworzyć wielu skomplikowanych reguł dostosowanych do konkretnych rozdzielczości, wszystko jest już gotowe i zaimplementowane.

Aby umieć Bootstrapa nie wystarczy znać zasad pisania arkuszy stylów, a znać pewne koncepcje narzucone przez autorów tej biblioteki.

Co ciekawe twórcami Bootstrapa są inżynierowie Twittera. Stworzyli dla siebie wygodny komplet arkusza stylów, który mógłby być wielokrotnie używany bez konieczności tworzenia podstaw od początku. Idea okazała się tak trafna, że teraz z ich rozwiązania korzysta każdy.

## 4.3. Rozważania na temat wzorców architektonicznych

Zaprezentowanie podziału aplikacji od strony klienta i serwera na logiczne warstwy wzorca MVC jest ważną częścią niniejszej pracy inżynierskiej. W wielu źródłach pojawiają się informacje, że zarówno aplikacje ASP.NET MVC jak i odpowiednie użycie Angulara pozwolą osiągnąć architekturę MVC.

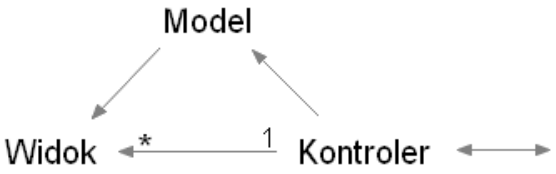
Słuszność tych stwierdzeń zależy od sposobu implementacji danego wzorca projektowego. Warto przeanalizować kilka scenariuszy.

### 4.3.1. Architektura MVC w ASP.NET MVC

Wzorzec MVC to skrót od angielskiej nazwy Model View Controller. Jest to wzorzec projektowy, wykorzystywany przez wielu programistów. Polega on na podzieleniu naszej aplikacji na poszczególne warstwy. Każda warstwa odpowiada za poszczególne zadania. Dzięki podzieleniu aplikacji na osobne warstwy kod nie przeplata się z treścią. Niesie to ze sobą ogromne plusy.

Zalety wzorca MVC jest wiele. Dzięki podzieleniu projektu na warstwy każda osoba w drużynie może pracować we własnej warstwie. Grafik tworzy interfejs użytkownika a programista kontrolery odpowiedzialne za logikę.

Źródłem interakcji we wzorcu MVC jest kontroler. Przychodzące żądanie zostaje obsłużone przez kontroler, ten za pomocą routingu wybiera odpowiednią akcję łącząc model z widokiem. Jeden kontroler może zwracać wiele różnych modeli, tak samo jak wiele modeli może odnosić się do tego samego kontrolera.




Rys 3.1 Schemat wzorca MVC

Za synchronizację danych z widoku do modelu odpowiada kontroler. Wszelkie zmiany wprowadzone w widoku nie zostają automatycznie przeniesione do modelu, stanie się to dopiero po (ponownie) wywołaniu akcji kontrolera. Upraszczając kontroler pełni rolę pośrednika pomiędzy widokiem a modelem.

Ważny jest fakt, że kontroler jako serce jest absolutnie bezstanowy. Stan danych zawierają modele, reprezentację widoki, komunikacja odbywa się poprzez kontroler, który odpowiada tylko za logikę biznesową.

Źródłem danych dla widoku jest model, a nie kontroler. Kontroler nie przechowuje żadnych informacji.

### 4.3.2. Architektura MVW w AngularJS

AngularJS na początku swojego istnienia architektonicznie bardzo przypominał użycie wzorca MVC (szeroko opisany w literaturze [8] oraz [9]). Sytuacja zmieniła się z biegiem czasu wraz z powstawaniem kolejnych wersji tej platformy i obsługą np. mechanizmu wstrzykiwania zależności.

Zagłębiając się w różne źródła można znaleźć setki informacji o tym jak sklasyfikować architektonicznie aplikacje stworzone w Angularze. Wielu inżynierów podaje swoje własne rozwiązania i interpretacje, co zaskutkowało powstaniem pojęcia MVW (z ang. model widok cokolwiek).

Analizując sytuację na dzień dzisiejszy aplikacjom Angularowym zaprojektowanym w specjalny sposób najbliżej jest do zasymulowania wzorca o nazwie MVVM (z ang. model widok widok-model).

Wzorzec architektoniczny MVVM jest pochodnym dla wzorca MVC, jednak występują między nimi pewne różnice. Pierwszą z nich jest źródło interakcji, które pochodzi bezpośrednio z widoku. Użytkownik systemu poprzez wykonywanie dowolnych operacji na widoku powoduje poinformowanie warstwy widok-model o zachodzących zmianach. Warstwa widok-model komunikuje się natomiast z modelami manipulując danymi w odpowiedni sposób. Kontroler ze wzorca MVC został zastąpiony elementem widok-model, który to element jest pośrednikiem pomiędzy widokiem a modelem.


Rys. 2.1 Schemat wzorca MVVM

MVVM cechuje fakt, że widok nie wie nic o istnieniu modelu, nie ma na jego temat żadnych informacji, taka sama relacja występuje w drugą stronę. W modelu MVC źródłem danych dla widoku jest model, jednak widok jest związany z modelem. Związanie występuje podczas wywołania akcji kontrolera. Dodatkowo widok zawiera referencję do modelu, więc zna jego strukturę. W architekturze MVVM wiązanie pomiędzy widokiem a widokiem-modelem występuje ciągle przez mechanizm *two-way data binding*, ale nie występuje pomiędzy widokiem a modelem. Widok-model przechowuje stan widoku i wszelkie zachodzące na nim zmiany, relacja działa też w drugą stronę. To kiedy model zostanie wypełniony danymi (np. po kliknięciu w przycisk) zależy od logiki aplikacji, która także zawarta jest w widok-modelu.

Głównym celem MVVM jest odcięcie się od widoku aplikacji, poprzez „wystawienie" jej danych za pomocą mechanizmu dwustronnego bindowania. Model-widok, którym w Angularze jest po prostu *scope* aplikacji, jest interfejsem wystawiającym dane. Jednak widok, który z tych danych skorzysta, nie ma pojęcia o modelu danych, nie jest z nim w żaden sposób związany. Dzięki temu uzyskujemy pewną warstwę dodatkowej abstrakcji, dodatkową skalowalność i responsywność.

### 4.3.3. Skąd rozwój w stronę MVVM?

Należy odpowiedzieć sobie na pytanie, dlaczego nastąpił zwrot ze wzorca MVC na wzorzec MVVT (szeroko opisany w książce [10])? Dawniej aby zbudować solidny interfejs użytkownika wystarczał projekt ASP.NET z warstwą widoków, tworzonych przy użyciu platformy Razor. Technologia ta, choć skuteczna, szybko przestała być wystarczająca. Minęły już czasy kiedy front-end aplikacji był tylko zlepkiem HTMLa przeplatanego wartościami pobieranymi z modeli. Front-end aplikacji przestał być pasywną, ubogą stroną niepodatną na jakiekolwiek interakcje aż do momentu przeładowania widoku przez nowe zapytanie HTTP.

Rozwój front-endu spowodowany jest wielką globalną informatyzacją. Interfejsy internetowe stawały się coraz ładniejsze i coraz bardziej funkcjonalne. Inżynierowie oprogramowania odkryli potencjał jaki ma w sobie przeglądarka internetowa klienta, gotowa parsować język skryptowy JavaScript posiadając w zapleczu całkiem dużą ilość pamięci operacyjnej.

Ciągle istniało jednak ograniczenie, komunikacja pomiędzy klientem a serwerem odbywa się przez protokół HTTP a jest to protokół bezstanowy, polegający na żądaniach i odpowiedziach. W takiej architekturze mógł działać tylko wzorzec MVC, który jest bezstanowy i wywołuje odpowiednie modele i widoku podczas określonych odwołań do routingów. Co prawda możliwe było ratowanie się asynchronicznym AJAX-em, jednak pisanie funkcji zwrotnych w AJAX było czasochłonne i nieskalowalne. W tym momencie zrodziła się koncepcja wymienienia warstwy bezstanowego kontrolera na warstwę widok-modelu, (w Angularze jest to *scope*), który byłby w stanie zapamiętywać dane, a także je przetwarzać.

Warstwa kontrolera nie znikła dosłownie, jednak teraz kontroler jest częścią większej całości jaką jest model-widok a więc *scope*. Kontrolery są w obiekcie *scope* symulowane jako funkcje (anonimowe lub nie). Użytkownik nie musi się już komunikować z kontrolerami, tylko bezpośrednio z widokiem, który resztę przekazuje dalej poprzez dwustronne bindowanie danych.

# 5. **Projekt systemu**

W tym dziale pokazany będzie ogólny projekt systemu. Zaprezentowane zostaną wszelkie wykorzystane mechanizmy zarówno po stronie serwera jak i klienta.

## 5.1. Warstwa serwera

Strona serwera to warstwa aplikacji napisana w technologii ASP.NET MVC5. Jest odpowiedzialna przede wszystkim za pobieranie danych z bazy danych i wystawianie ich dla klienta w postaci REST API.

Serwer został skonfigurowany tak, aby wszelkie dane były wyświetlane w postaci JSON. Ponieważ klient oparty jest o AngularJS – framework JavaScript – format JSON będzie najbardziej przyjazny i łatwy do przetworzenia.

Serwer aplikacji pełni rolę szyny komunikacyjnej, przekazuje i odbiera dane z front-endu do bazy danych. Jedyny moment kiedy użyte będą widoki ASP.NET MVC to utworzenie kontenera dla klienta AngularJS oraz podstrony logowania, rejestracji i błędów.

### 5.1.1. Autoryzacja

ASP.NET udostępnia wiele gotowych mechanizmów służących do autoryzacji. W projekcie zarządzania domami studenckimi autoryzacja została oparta na sesjach i klasie statycznej przechowującej model zalogowanego użytkownika.

Po wpisaniu danych logowania uruchomiona zostaje metoda kontrolera *AccountController* wywołująca odpowiednią funkcję z repozytorium *UserRepository*. Poprawność danych sprawdzana jest z danymi zawartymi w bazie danych. Jeżeli wpisane dane są poprawne zostaje ustawiona odpowiednia flaga w zmiennej sesyjnej, a model zalogowanego użytkownika zwrócony przez repozytorium zostaje zapisany w klasie statycznej.

Dzięki temu w każdej chwili działania systemu i aktywności danej sesji mamy globalny dostęp do danych użytkownika.

#### 5.1.1.1. Filtr autoryzacji

W projekcie istnieje zaimplementowany filtr o nazwie *MyAuthorize*. Jest on dołączony do każdego kontrolera, który ma być zabezpieczony hasłem. Dotyczy to zarówno kontrolerów zwykłych generujących startowe widoki pod Angulara jak i kontrolerów WinAPI odpowiedzialnych za wystawienie danych za pomocą RESTa.

Nie możliwe okazało się globalne nałożenie filtra autoryzacji w pliku *FilterConfig.cs* w metodzie *RegisterGlobalFilters* ponieważ skutkowało to brakiem dostępu do kontrolera *AccountController*, zawierającego akcje logowania i rejestracji konta. To samo dotyczy kontrolera *ErrorController* odpowiedzialnego za mechanizm błędów.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    AllowMultiple = false, Inherited = true)]
public class MyAuthorize : ActionFilterAttribute
{

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!UserHelper.IsLogged())
        {
            context.HttpContext.Response.Redirect("~/Account/Home/Login", true)
            context.Result = new RedirectResult("~/Account/Home/Login");
        }
    }
}
```

Rys. 1.1. Budowa filtra uwierzytelniającego

Budowa filtra jest bardzo prosta, skupia się na odpytywaniu klasy statycznej *UserHelper* czy nastąpiło logowanie.

#### 5.1.1.2. Filtr roli

W systemie jest zaimplementowany mechanizm roli. Każdy użytkownik w bazie danych posiada zdefiniowane określone role. Role posiadają wartości liczbowe będące potęgami liczby 2. Dzięki temu określone role można łączyć poprzez zsumowanie ich wartości.

```
public enum Role
{
    Admin = 1,
    Pracownik = 2,
    Student = 4,
    Gość = 8,
}
```

Rys. 2.1. Role w systemie

Tak samo jak w przypadku filtra odpowiedzialnego za autoryzację, w systemie istnieje filtr o nazwie *MyRoles*. Został on dodany jako atrybut do każdego kontrolera, gdzie dostęp powinni dostać użytkownicy w zależności od danej roli.

### 5.1.2. Protokół komunikacyjny HTTP

W projekcie protokołem komunikacyjnym jest protokół HTTP. Jest to protokół bezstanowy składający się z żądań i odpowiedzi. Została podjęta decyzja o użyciu protokołu HTTP ze względu na jego prostotę, skuteczność i niezawodność.

Dawniej budując API aplikacji także korzystało się np. z protokołu HTTP, jednak nie było tam podejścia RESTowego. Wywołując jakąś akcję z API pod adresem *api/deleteUser/5* otrzymywaliśmy odpowiedź w formacie json lub xml zawierającą odpowiedź czy zabieg się udał. Kod odpowiedzi był ustawiony na 200 (sukces) i oznaczał sukces dostarczenia odpowiedzi a nie sukces operacji usunięcia użytkownika.

#### 5.1.2.1. Styl REST

REST jest to sposób komunikacji oparty na protokole HTTP. Wiele interesujących informacji znajduje się w literaturze [11]. Cechują go prosta budowa i łatwość implementacji. Polega on na ujednoliceniu sposobu wymiany danych pomiędzy klientem a serwerem. Systemy informatyczne używające RESTa są zrozumiałe i nie wymagają tak szczegółowej dokumentacji jak API budowane w innych standardach.

Potęga RESTa polega na tym, że do zbudowania API wykorzystuje on jedynie metody dostępne w protokole HTTP, są to: *Post, Get, Put, Delete*. Dzięki temu użytkownik danego repozytorium podświadomie rozumie działanie systemu bez konieczności długiego wdrażania się. Po pierwsze powstaje uporządkowane API więc nie trzeba domyślać się pod jaki adres wysłać POSTa żeby usunąć użytkownika. Gdyby nie REST usunięcie użytkownika mogłoby się odbyć przez żądanie POST na adres *api/deleteUser*. Dodatkowo nie trzeba sprawdzać treści odpowiedzi od serwera, wszystkiego dowiadujemy się już po kodzie odpowiedzi. Sukces (200) zostaje zwrócony tylko w przypadku powodzenia operacji.

REST jest szeroko wykorzystywany ponieważ jest prosty i intuicyjny. Bardzo dobrze sprawdza się pod względem wieloplatformowości gdzie nie sprawdza się już np. SOAP. Serwer postawiony w NodeJS korzystający z podejścia RESTfull API zajmuje zaledwie kilka linijek kodu. Funkcjonalność REST jest automatycznie wbudowana w protokół HTTP i nie wymaga żadnych dodatkowych nakładów pracy. Dla porównania wdrożenie SOAP jest już bardziej pracochłonne np. ze względu na konieczność definiowania XML Schemas.

REST jest wygodny zarówno dla programisty piszącego kod jak i dla odbiorcy. Narzuca na obie strony pewne umowne standardy, dzięki czemu programista nie musi się zastanawiać od jakiej strony ugryźć problem lub jakie metody wystawić w nowym fragmencie API. Podejście REST stało się wygodnym standardem implementowanym wszędzie tam, gdzie nie potrzeba bardziej zaawansowanych narzędzi, choć czasem prowadzi to do nadmiarowości.

### 5.1.3. Interfejs WebAPI

Do budowania API oraz repozytoriów aplikacji najlepiej użyć kontrolerów WebAPI. Są to wszelkie kontrolery dziedziczące jawnie z klasy *ApiController*. Różnica pomiędzy zwykłym kontrolerem a WebAPI kontrolerem jest taka, że zwykłe kontrolery zwracają widoki a WebAPI kontrolery zwracają dane (w postaci JSON lub XML).

WebAPI został stworzony z myślą o standardzie REST. W przypadku chęci użycia np. SOAP należałoby zamiast WebAPI użyć serwisów z platformy WCF. Serwisy są bardziej rozbudowane i zapewniają większe możliwości, ale wymagają też większego nakładu pracy. Do budowania prostego API aplikacji najlepszym wyborem są kontrolery WebAPI.

W pliku *WebApiConfig.cs* została wprowadzona zmiana, dzięki której wszystkie zwracane dane mają postać JSON. Aby osiągnąć ten efekt należało usunąć z kolekcji *SupportedMediaTypes* wartość *application/xml*.

#### 5.1.3.1. Bindowanie parametrów

Przekazywanie danych do metod wystawionych w kontrolerach nosi nazwę bindowania parametrów. Bindowanie parametrów polega na dopasowaniu parametrów podanych wewnątrz zapytania do parametrów podanych jako argumenty akcji kontrolera.

Podczas bindowania parametrów typów prostych kontroler WebAPI próbuje automatycznie zbindować je z adresu URI. Typami prostymi są wszelkie te wywodzące się z przestrzeni *System*. Chcąc przesłać typ prosty przez treść zapytania należy w parametrach użyć atrybutu *[FromBody]* i wymusić bindowanie z treści.

Parametry typów złożonych są automatycznie przekazywane przez treść więc żaden problem nie wystąpił.

#### 5.1.3.2. Walidacja danych

Platforma .NET udostępnia ciekawy mechanizm walidacji modeli za pomocą tzw. atrybutów. Atrybuty dodajemy w modelach przed nazwą właściwości, której mają dotyczyć. Dzięki temu możemy zakładać różne warunki np. określić że pole jest wymagane, lub jaką długość znaków ma mieć.

Walidacja następuje poprzez sprawdzenie atrybutu *ModelState.IsValid*. W przypadku wartości *false* model nie został wypełniony zgodnie z atrybutami.

W tym projekcie wszystkie modele są walidowane za pomocą atrybutów.

### 5.1.4. Zabezpieczenie poprzez AntiForgeryToken

W aplikacji o architekturze MVC używającej do komunikacji stylu REST i protokołu HTTP, wrażliwym fragmentem systemu jest API wystawione przez stronę serwera. Odbieranie i zapisywanie wszelkich danych odbywa się za pomocą zapytań POST lub GET wysyłanych na poszczególne adresy API dołączając do nich odpowiednie wartości. Wartości zapytań są parsowane przez kontroler i zapisywane w bazie danych.

Potencjalny haker może bez żadnego problemu podejrzeć treść, typ i nagłówki wysyłanych zapytań a następnie je spreparować w celu osiągnięcia korzyści, co skutkuje złamaniem aplikacji. Przykładowe zapytanie POST może wyglądać następująco:

```
-- zapytanie --
POST /api/usunUzytkownika
Content-type: application/json

{"id" : 51}

-- odpowiedź --
200 OK
```

Rys. 4.1. Przykładowe żądanie POST

Za pomocą dowolnego programu umożliwiającego wysyłanie żądań HTTP (tzw. REST Client) można spreparować treść zapytania zmieniając ID użytkownika na jakikolwiek inny istniejący w systemie. Jest to przykład ataku CSRF charakteryzujący się właśnie podrobieniem parametrów zapytań. W odróżnieniu od XSS atakujący jest użytkownikiem posiadającym

autoryzowany dostęp do systemu, podatność na atak polega na zaufaniu systemu do użytkownika i braku walidacji, a nie na naiwności użytkownika klikającego w spreparowany link jak w ataku XSS.

Zabezpieczeniem przed tego typu atakiem jest użyciem mechanizmu AntyForgeryToken. Jest to prosty ale bardzo skuteczny mechanizm. Po jego wdrożeniu każde zapytanie wysłane do serwera posiada nagłówek z ciągiem losowych znaków. Ten sam ciąg losowych znaków jest zapisany w ciasteczku po stronie klienta i w sesji po stronie serwera. Jeżeli ciąg znaków nie zgadza się z ciasteczkiem lub z wartością sesji na serwerze, żądanie zostaje odrzucone.

Istnieje jednak pewna ważna przeszkoda, która musiała zostać rozwiązana. Domyślnie token generowany jest w widoku poprzez funkcję udostępnioną przez Razora o nazwie *@.Html.AntiForgeryToken()*. Zostaje dołączony do formularza w postaci pola typu *hidden* i wysłany żądaniem POST. Funkcja generuje także ciastko i zapisuje token do sesji. Zmiana jakiejkolwiek wartości (sesji, ciastka lub nagłówka żądania z pola *hidden*) spowoduje zablokowanie dostępu do kontrolera. Aby zabezpieczyć określony kontroler tokenem, należy użyć filtra *[ValidateAntiForgeryToken]*.

#### 5.1.4.1. AntiForgeryToken i AngularJS

Na potrzeby komunikacji asynchronicznej jaką udostępnia AngularJS niezbędna jest modyfikacja mechanizmu tokenu. Token musimy za każdym razem pobierać z widoku Razora, po załadowani modułów Angulara i dołączać do żądań, które AngularJS wysyła do API serwera. W tym celu najlepiej stworzyć własny interceptor żądania i dołączać do globalnie do każdego zapytania.

```
function antiForgeryTokenInterceptor() {
    return {
        request: function (config) {
            var token = $("input:hidden[name=\"__RequestVerificationToken\"]").val();

            if (token.length > 0) {
                config.headers.__RequestVerificationToken = token;
            }

            return config;
        }
    };
}
```

Rys. 1.1. Interceptor odpowiedzialny za dodawanie AntiForgeryTokena do zapytań.

Ostatnim prostym krokiem jest dodanie interceptora w sekcji *config* modułu.

```
$httpProvider.interceptors.push("antiForgeryTokenInterceptor");
```

Rys. 1.2. Globalne uruchomienie interceptora

Ostatnim krokiem w celu osiągnięcia pełnej funkcjonalności jest napisanie własnego filtra, walidującego token. Walidujemy wartość z ciastka oraz z nagłówka żądania.

```
public sealed class MyAntiForgeryValidate : ActionFilterAttribute
{
    public override void OnActionExecuting(System.Web.Http.Controllers.HttpActionContext actionContext)
    {
        var headers = actionContext.Request.Headers;

        var headerToken = headers.GetValues("__RequestVerificationToken").FirstOrDefault();
        var cookieToken = headers.GetCookies()
            .Select(x => x[AntiForgeryConfig.CookieName]).FirstOrDefault();

        AntiForgery.Validate(cookieToken.Value, headerToken);

        base.OnActionExecuting(actionContext);
    }
}
```

Rys. 1.3. Filtr do walidacji tokena po stronie serwera

Teraz każde żądanie wysłane od klienta jest zabezpieczone tokenem. Istnieje drobna wada tego rozwiązania, token nie jest odświeżany co każde zapytanie, a dopiero w momencie przeładowania widoku Razora a więc w momencie odświeżenia akcji kontrolera. Mimo tego jakakolwiek próba podrobienia żądania zostanie szybko wychwycona.

Każdy kontroler WebAPI jest oznaczony atrybutem *[MyAntiForgeryValidate]*. Warto dodać, że AntiFogreryToken zabezpiecza tylko metody POST, co jest nawet logiczne, ponieważ metody GET nie powinny być odpowiedzialne za zapisywanie żadnych danych.

### 5.1.5. Silnik ADO.NET do komunikacji z bazą

Do połączenia z bazą danych użyty został silnik ADO.NET udostępniony przez platformę .NET. Według statystyk wydajności jest on nieznacznie szybszy od konkurencyjnego JDBC znanego programistom Javy.

#### 5.1.5.1. Mikro ORM Dapper

W projekcie został użyty mikro ORM o nazwie Dapper. Ponieważ jest to mikro ORM nie mapuje on encji bazy danych na obiekty typu POCO, a jedynie odpowiada za bindowanie danych zwracanych z bazy danych do modeli.

Użycie zwykłego ORMa takiego jak *NHibernate* lub *EntityFramework* jest pod pewnym względem wygodne, jednak zapewnia małą skalowalność co zaprzeczałoby celom tego projektu.

```
IEnumerable<Reservation> list = connection.Query<Reservation>("Reservations.Reservation_GetAll",
    Parms, commandType: CommandType.StoredProcedure).ToList();
```

Rys 1.1. Przykładowe użycie Dappera

Aby móc użyć Dappera niezbędne jest zwrócenie parametrów nazwanych w ten sam sposób co właściwości znajdujące się w modelu. W przypadku niezgodności ilości parametrów, ich typów albo liczby zostanie zwrócony wyjątek *Wrong numer or types of arguments*.

Dapper jest bardzo funkcjonalny, umożliwia zwracanie pojedynczych obiektów jak i całych kolekcji. Możliwe jest także zwrócenie bardziej złożonych konstrukcji np. kolekcji z

zagnieżdżonymi modelami. Mikro ORM znacząco skraca ilość wymaganego kodu pozostawiając programiście maksimum elastyczności.

### 5.1.6. Wzorzec repozytorium

Sercem serwera jest użycie wzorca repozytorium. Jest to wzorzec, który doskonale pasuje do architektury MVC. Kompletny diagram klas UML wzorca został przedstawiony w dziale 4.1.4.1 na rysunku 1.1.

Każde repozytorium dziedziczy jawnie z generycznego interfejsu *IRepository*, który definiuje podstawowe metody odpowiadającego RESTowi.

```
public interface IRepository<T> where T : class
{
    IEnumerable<T> GetByQuery();
    T GetById(int id);
    T Post(T item);
    T Put(T item);
    T Delete(int id);
}
```

Rys 6.1 Główny interfejs wzorca repozytorium

Oznacza to, że każde utworzone repozytorium musi implementować metody interfejsu. W przypadku gdy implementacja którejś z metod nie jest wymagana, repozytorium rzuca wyjątek *NotImplementedException*.

W przypadku chęci rozszerzenia repozytorium o dodatkowe metody należy dostarczyć dodatkową warstwę interfejsu czyli interfejs rozszerzający, zawierający dodatkowe metody.

```
public interface IPaymentRepository<T>
    : IRepository<T> where T : class
{
    IEnumerable<T> GetByQuery(int paid);
}
```

Rys 6.2 Rozszerzający interfejs repozytorium

Interfejs rozszerzający jest następnie implementowany przez główną klasę repozytorium. W projekcie założone zostało, że interfejs rozszerzający jest dołączony do każdego repozytorium nawet w przypadku jeżeli jest pusty.

Ponieważ projekt korzysta z mikro ORM nie ma w nim warstwy serwisu. Po dłuższej analizie okazało się, że byłaby ona nie używana i stanowiłaby dodatkową zbędną warstwę abstrakcji.

#### 5.1.6.1. Różnice pomiędzy Repozytorium a DAO

Wiele osób zastanawia się, jaka jest różnica pomiędzy wzorcem repozytorium a wzorcem DAO (ang. Data Access Object). Odpowiedź na to pytanie jest dość prosta.

Wzorzec repozytorium skupia się na obiektach POCO, czyli tych zdefiniowanych po stronie serwera. Zapewnia prosty, czysty i zrozumiały interfejs operacji na danych. Argumenty metod w repozytorium są dziedziczone z interfejsu bazowego *IRepository*.

DAO służy do operowania na obiektach bazy danych czyli potocznie encjach. DAO może mieć wiele różnorodnych metod w zależności od zapotrzebowania. W zaawansowanych projektach dobrze sprawdza się mechanizm łączenia wzorca repozytorium ze wzorcem DAO. Repozytorium odpytuje się wtedy poszczególnych metod DAO w celu przetwarzania informacji.

Czasem spotyka się implementację DAO korzystając jednocześnie z ORMa, co jest błędem. ORM jest typową warstwą, która operuje na encjach bazy danych i dołączanie do tego DAO wnosi tylko kolejną niepotrzebną warstwę.

Przykładem konieczności użycia DAO jest sytuacja, kiedy w projekcie znajduje się byt o przykładowej nazwie *Działka*. Do klasy reprezentującej działkę utworzone zostaje repozytorium z podstawowymi operacjami CRUD. Mimo tego obiekt działki jest w bazie danych przechowywany w wielu różnych tabelach, ponieważ w skład działki wchodzą budynki i rośliny. W tej sytuacji repozytorium powinno operować na obiekcie POCO od strony serwerowej, a DAO na encjach bazy danych.

### 5.1.7. Wstrzykiwanie zależności

Użycia wzorca repozytorium znacznie zwiększa skalowalność projektu. Możliwe jest odpinanie i podmienianie poszczególnych repozytoriów, np. w zależności od bazy danych na której odbywa się praca. Można posiadać osobne repozytoria dla baz testowych i osobne dla baz produkcyjnych.

Do uzyskania pełnej skalowalności w projekcie użyty został mechanizm wstrzykiwania zależności. Jest on realizowany za pomocą biblioteki open-source o nazwie *Ninject*.

```
private static void RegisterServices(IKernel kernel)
{
    kernel.Bind<IUserRepository<User>>().To<UserRepository>();
    kernel.Bind<IReservationRepository<Reservation>>().To<ReservationRepository>();
    kernel.Bind<IProposalRepository<Proposal>>().To<ProposalRepository>();
    kernel.Bind<IPaymentRepository<Payment>>().To<PaymentRepository>();
    kernel.Bind<IResidentsRepository<Residents>>().To<ResidentsRepository>();
    kernel.Bind<IPaymentListRepository<Payment>>().To<PaymentListRepository>();
}
```

Rys. 7.1. Zależności zdefiniowane w bibliotece Ninject

Na rys. 7.1. został przedstawiony fragment kodu, na którym zostały zdefiniowane zależności wstrzyknięcia zależności. Ninject szuka we wszystkich klasach projektu konstruktorów przyjmujących parametry o typach podanych w konfiguracji. Następnie do konstruktorów o określonych typach parametrów zostają wstrzyknięte instancje klas do nich przypisane.

Dzięki temu zabiegowi w projekcie utworzone zostały tylko odpowiednie konstruktory, bez tworzenia instancji obiektu. Efektem tego jest wygodnictwo w przypadku podmiany repozytorium. Jeżeli nowe repozytorium będzie rozszerzać interfejs swojego poprzednika wtedy aby przełączyć repozytoria wystarczy tylko podmienić plik w projekcie, bez jakichkolwiek zmian w kodzie.

### 5.1.8. Wielojęzyczność

Wielojęzyczność projektu to mechanizm złożony z dwóch części. Wersje językowe będą działały inaczej po stronie klienta i inaczej po stronie serwera.

Odsyłacze umieszczone w widokach służące do zmiany języka mają za zadanie wywołanie określonej metody z kontrolera odpowiedzialnej za zapisanie po stronie klienta ciasteczka. Zawartością ciasteczka jest język aplikacji.

Kolejnym krokiem jest odczytanie wartości ciastka i ustawienie języka będącego jego treścią. Kod odpowiedzialny za odczytywanie ciastka został umieszczony w pliku *Global.asax*. Do wyboru są dwa zdarzenia wywoływane podczas przyjścia każdego żądania do serwera, są to: *Application_AcquireRequestState* oraz *Application_BeginRequest*.

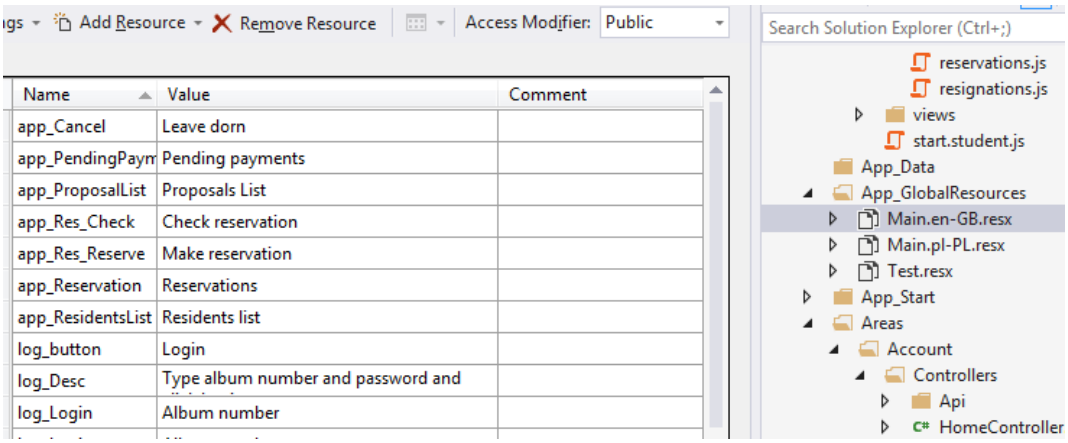
- Application_BeginRequest – jest to zdarzenie wywoływane gdy do serwera aplikacji przychodzi dowolne żądanie.
- Application_AcquireRequestState – jest to zdarzenie wywoływane podczas przypisania kontekstu sesji do danego żądania.

To w którym zdarzeniu obsłużymy mechanizm językowy nie ma w tym projekcie żadnego zdarzenia, ponieważ nie operujemy na obiekcie sesji. Gdyby nasze tłumaczenia korzystały ze zmiennych sesyjnych, nie moglibyśmy użyć Application_BeginRequest.

```
void Application_AcquireRequestState(object sender, EventArgs e)
{
    HttpContext context = HttpContext.Current;

    var cookie = context.Request.Cookies.Get("Language");
    if (cookie != null)
    {
        try
        {
            CultureInfo ci = new CultureInfo(cookie.Value);
            Thread.CurrentThread.CurrentCulture = ci;
        }
        catch
        {
            throw new Exception("Wrong language value");
        }
    }
}
```

Rys. 8.1. Ustawianie wersji językowej dla danego wątku

ASP.NET dostarcza ciekawy mechanizm operowania na plikach zasobów **.res*. Podczas zdefiniowania języka w klasie *CultureInfo* w projekcie zachodzą pewne z góry zaimplementowane zmiany np. zmienia się format daty na charakterystyczny dla danej kultury. Kolejną ciekawostką jest próba dopasowania plików zasobów w zależności od wybranego języka. Aby mechanizm zadziałał plik z zasobami musi posiadać w swojej nazwie symbol języka.




Rys. 8.2. Podgląd plików zasobów z wersjami językowymi

Na załączonym rysunku 8.2 widać strukturę pliku zasobów. Składa się on z par klucz-wartość. Po prawej stronie obrazka widać dwa pliki nazwane tak samo z małą różnicą dotyczącą języka. Teraz w przypadku wybrania wersji angielskiej lub polskiej, ASP.NET sam wybierze odpowiedni plik odwołując się do niego jedynie po nazwie z przed kropki.

Odczytane wartości zostają w prosty sposób wyświetlone w odpowiednich miejscach w widokach Razor. Odpowiada za to kod:

```
@HttpContext.GetGlobalResourceObject("Main", "utils_Welcome", Ci)
```

Parametrem Ci jest aktualna instancja klasy *CurrentThread.CurrentCulture*. W głównym widoku aplikacji, w którym zostaje załadowana cała aplikacja klienta napisana w AngularJS, zostaje dodana globalna zmienna JavaScript *lang*. Jest w niej zapisany symbol aktualnego języka. Dzięki temu po załadowaniu aplikacji po stronie klienta do języku wybranego na serwerze zostaje szybko dopasowana pasująca wersja i związany z nią serwis.

## 5.2. Warstwa klienta

W tym akapicie zostały opisane wszelkie niezbędne aspekty dotyczące działania aplikacji po stronie klienta. Całość oparta jest o platformę programistyczną AngularJS opartą na języku skryptowym JavaScript.

### 5.2.1. Działanie biblioteki AngularJS

AngularJS jest użyteczną biblioteką przeznaczoną bardziej dla developerów niż projektantów. Zapewnia asynchroniczną komunikację z dowolnym źródłem danych z jakiego korzysta. Biblioteka została wyposażona w mechanizm wstrzykiwania zależności, dzięki temu tworzony kod jest skalowalny i łatwo testowalny. Dzięki wstrzykiwaniu zależności stworzony zostaje alias dla klasy, której używamy. W przypadku chęci testowania kontrolera wystarczy zmienić miejsce odpowiedzialne za wstrzykniecie zależności a nie logikę kontrolera.

Działanie AngularJS opiera się na trzech fazach:

– Faza ładowania – następuje podczas uruchomienia w oknie przeglądarki podstrony, na której załadowana jest biblioteka AngularJS. Podczas tej fazy załadowane

zostają wszelkie moduły i komponenty. Zostają także wstrzyknięte instancje odpowiednich klas za pomocą mechanizmu wstrzykiwania zależności.

- Faza kompilacji – statyczne drzewo dokumentu DOM wygenerowane przez przeglądarkę zostaje zamienione na dynamiczne. W tym momencie wszelkie dyrektywy zostają wypełnione danymi pochodzącymi z zasięgu danego kontrolera. Zostaje uruchomiony mechanizm *two-way data binding*. Podczas skanowania statycznego drzewa DOM do każdej napotkanej dyrektywy zostaje podpięty obserwator (watcher).
- Faza działania – jeżeli jakikolwiek obserwator wykryje zmianę danych po stronie widoku lub kontrolera, uruchomiona zostanie tzw. *digest cycle* czyli pętla odpowiedzialna za odświeżenie danych. Faza ta trwa aż do momentu zamknięcia okna przeglądarki.
-

#### 5.2.1.1. Ładowanie biblioteki AngularJS

W realizowanym projekcie AngularJS zostaje wczytany za pomocą ładowania programowalnego. W drzewie DOM nie znajduje się nigdzie dyrektywa *ng-app* co powoduje, że Angular nie zostanie automatycznie uruchomiony. W celu uruchomienia AngularJS, w pliku głównego modułu została dołączona linijka rozkazująca załadować platformę AngularJS (tzw. bootstraping Angulara).

Dzięki temu biblioteka zostanie załadowana dopiero w momencie gdy cała hierarchia drzewa DOM zostanie już załadowana, podczas wystąpienia zdarzenia *(document).ready()*.

### 5.2.2. Symulowanie działania wzorca MVC

W poprzednich rozdziałach niniejszej pracy omówione zostały aspekty dotyczące architektury aplikacji napisanych w AngularJS. Nazywane one zostają po prostu wzorcem MVW (z ang. model widok cokolwiek). Mimo tego programista posiada kontrolę nad tym, w jaki sposób zaprojektuje aplikację. W zależności od sposobu implementacji poszczególnych warstw, architektura może być bardziej zbliżona do wzorca MVC lub MVVM. Ze względu na dwustronne bindowanie danych najbardziej korzystnym rozwiązaniem wydaje się korzystanie z architektury MVVM.

#### 5.2.2.1. Moduły

W aplikacji zarządzania akademikami zostaje utworzony główny moduł aplikacji odpowiedzialny za zdefiniowanie aplikacji. Za pomocą mechanizmu wstrzyknięcia zależności zostają do niego przekazane referencje do innych modułów pobocznych. Poboczne moduły mają na celu grupowanie kontrolerów związanych z danym aspektem. Dla przykładu moduł *payments* zawiera w sobie kilka kontrolerów związanych z płatnościami takich jak: *paymentsListController, paymentsPeroidController, paymentCheckController*. Oprócz tego każdy moduł definiuje routingi jakie muszą zostać wywołane w celu uruchomienia danego kontrolera i widoku.

```
(function (angular) {
    'use strict';

    angular
        .module('reservations', ['reservationsResource', 'mgcrea.ngStrap'])
        .config(config)
        .controller('MakeReservation', MakeReservationController)
        .controller('CheckReservation', CheckReservationController)
    ;
```

Rys. 1.1. Definicja modułu rezerwacji

Nieodłącznym elementem modułu jest sekcja ustawień zawierających określony routing. Określa on jakie kontrolery mają zostać załadowane dla poszczególnych definicji routingu, oraz jakich widoków mają one używać.

```
function config($routeProvider) {

    $routeProvider
      .when("/reservations", {
          redirectTo: 'reservations/make'
      })

        .when('/reservations/make', {
            controller: 'MakeReservation',
            controllerAs: 'vm',
            templateUrl: template('reservations/make'),
            resolve: {
                reservation: function (reservationFactory) {
                    return reservationFactory.check().$promise;
                }
            },
      })
```

Rys. 1.2. Fragment definicji routingów dla modułu

### 5.2.2.2. Kontrolery

Kontrolery budują wnętrze modułów aplikacji. Są odpowiedzialne za podstawową logikę a przede wszystkim jest do nich podpięty *scope*. Kontrolery są warstwą logiczną pomiędzy widokiem a modelem. *scope* w kontrolerze pełni rolę warstwy widok-model (czyli VM we wzorcu MVVM). *scope* jest częścią kontrolera, pośrednikiem pomiędzy widokiem a kontrolerem. Kontroler definiuje z jakiego modelu i widoku będzie korzystała warstwa widok-model.

```
PaymentPeroidController.$inject = ['$scope', 'paymentFactory', 'payments'];
function PaymentPeroidController($scope, paymentFactory, payments) {

    var vm = this;
    vm.months = ["styczeń", "luty", "marzec", "kwiecień", "maj", "czerwiec",
        "lipiec", "sierpień", "wrzesień", "październik", "listopad", "grudzień"]

    vm.payments = payments;

    angular.forEach(vm.payments, function (item) {
        var nr = moment(item.Datev).format('M');
        item.Datevv = vm.months[parseInt(nr)-1];
    });
}
```

Rys. 2.1. Przykładowy kontroler wykorzystany w aplikacji

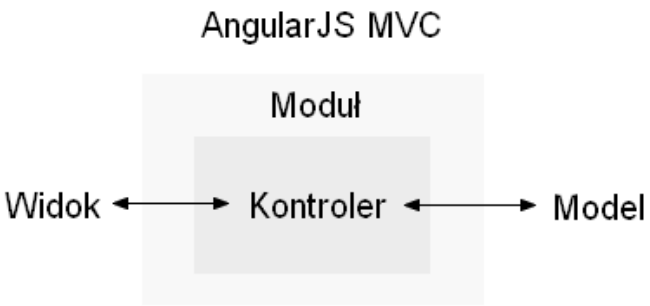
#### 5.2.2.3. Serwisy i fabryki

Serwisy i fabryki w AngularJS tworzą warstwę modelu. Dodatkowo są wyposażone w metody pobierania i odczytywania danych z API. Nigdzie nie jest powiedziane, że we wzorcu MVC lub MVVT model musi przypominać prostą klasę znaną z C#. Zaimplementowanie w nich drobnej logiki nie jest żadnym nadużyciem.

Różnice pomiędzy serwisami i fabrykami są subtelne. Serwis zwraca instancję obiektu funkcji gotową do użycia. Serwis jest singletonem, natomiast fabryka zwraca zwykły obiekt, który został wewnątrz niej zdefiniowany.

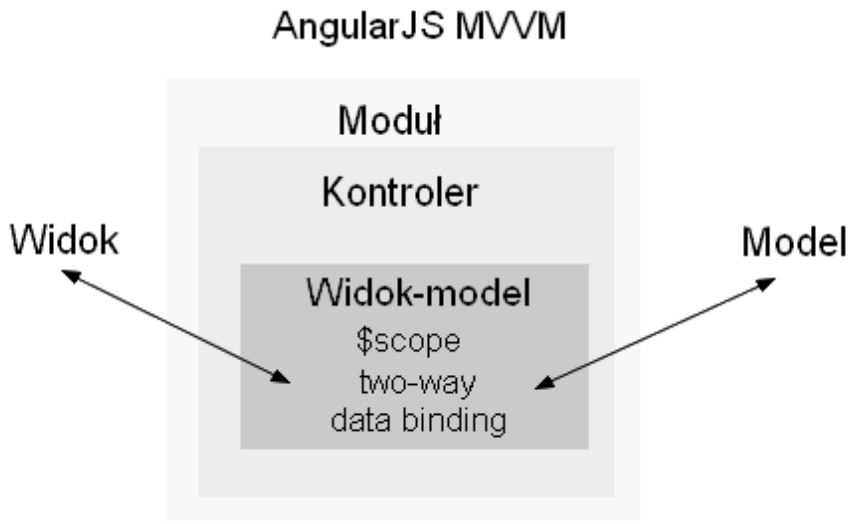
#### 5.2.2.4. MVC czy MVVM?

Analizując aplikację powierzchownie, realizuje ona wzorzec MVC. Struktura zaprezentowana jest na rysunku niżej:




Rys. 4.1. Model MVC aplikacji Angular

Przyglądając się dogłębniej sposobowi działania AngularJS, spod warstwy kontrolera wyłania się kolejna warstwa. Pozwala to sklasyfikować architekturę do MVVM.




Rys. 4.2. Model MVVM aplikacji Angular

### 5.2.3. Wspieranie REST

Platforma AngularJS jest bardzo przyjazna stylowi REST. W module *ngResource* znajduje się serwis o nazwie *resource*. Pozwala on na szybkie i łatwe stworzenie komunikacji z serwerem wykorzystując REST.

Serwis ten wspiera domyślnie wszystkie rodzaje żądań HTTP. Dodatkowo można go rozszerzyć o własne elementy i skonfigurować według potrzeb.

W niniejszym projekcie obiekty *resource* są zwracane z fabryk dołączonych za pomocą wstrzykiwania zależności. Przykładowy *resource* wygląda następująco:

```
.factory('dornFactory', ['$resource', function ($resource) {
    return $resource('api/proposal/', {}, {
        dornlist: {
            method: 'GET',
            isArray: true,
            url: 'api/proposal/dornlist'
        },
        floorlist: {
            method: 'GET',
            isArray: true,
            url: 'api/proposal/floorlist'
        }
    })
}]);
```

Rys. 3.1. Przykładowa implementacja *resource*

### 5.2.4. Responsywność

Responsywność jest bardzo ważnym aspektem systemów internetowych. Wiele informacji na jej temat opisuje literatura [12]. Od czasu pojawienia się systemu Android procent urządzeń mobilnych podłączonych do sieci jest ogromny. Stworzenie interfejsu użytkownika, który nie jest responsywny wiąże się ze stratą potencjalnych użytkowników/klientów. Otworzenie strony

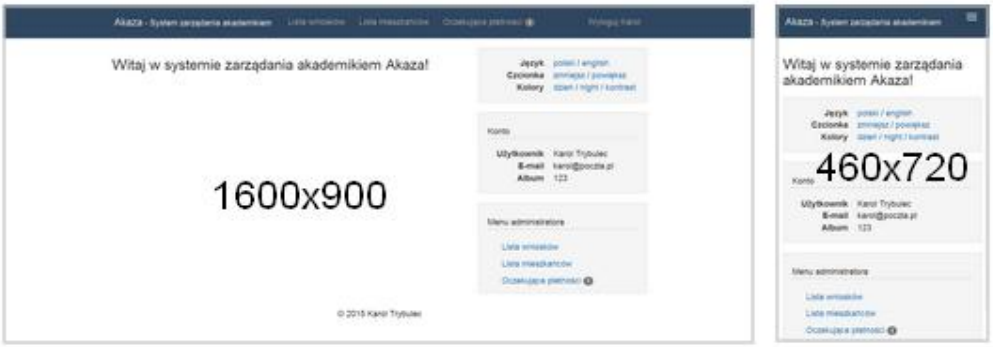
przystosowanej do rozdzielczości przeciętnego laptopa na małym wyświetlaczu tableta lub telefonu jest po prostu niemożliwe.

Tworząc strony responsywne należy trzymać się zasady aby nie ustalać stałej szerokości (atrybut *width*) dla żadnego elementu interfejsu. Można posługiwać się stałą, statyczną wysokością ale szerokość może być podana tylko w jednostkach procentowych. Użycie jednostki procentowej w arkuszach stylów dla atrybutów wielkości powoduje obliczenie rozmiaru elementu względem rozmiarów okna przeglądarki.

Dawnym sposobem na tworzenie stron przystosowanych do telefonów komórkowych było tworzenie osobnego, „małego" szablonu serwisu dostępnego w subdomenie *m.mojastrona.pl*. Kolejnym krokiem była analiza tablicy superglobalnej $_SERVER zawierającej różne informacje o środowisku. Konkretnej należało przeanalizować element *HTTP_USER_AGENT* z którego można odczytać system operacyjny użytkownika wysyłającego żądanie. Było to rozwiązanie skuteczne ale niewygodne i obecnie odchodzi w zapomnienie.

Responsywność pojawiła się w arkuszach stylów od wersji CSS3 wraz z rozbudowaniem możliwości *Media queries*. Zostały one poszerzone o nowe możliwości *min-width* oraz *max-width* pozwalające na tworzenie reguł wyświetlania poszczególnych elementów DOM w zależności od wykrytej szerokości okna w czasie rzeczywistym. Zalet korzystania z responsywności opartej o *media queries* jest wiele:

- brak konieczności aktualizacji treści w dwóch różnych miejscach (istnieje tylko jedna wersja serwisu)
- brak zamieszania z przekierowaniem domen, przełączaniem szablonów oraz detekcją urządzeń
- nie trzeba się martwić o blokadę *duplicate-content*
- jest to sposób oficjalnie rekomendowany przez Google
- dostosowuje się nie tylko do różnych urządzeń, ale także do różnych rozdzielczości ekranu (jest najbardziej elastycznym rozwiązaniem)




Rys. 4.1. Widok dla różnych rozdzielczości

W projekcie wykorzystany został Bootstrap CSS będący zbiorem gotowych klas gotowych do wykorzystania. Podejście projektowania interfejsu korzystając z bootstrapa należy trzymać się koncepcji opracowanej przez samych jego twórców czyli tzw. siatki (z ang. bootstrap grid). Siatkę tworzymy poprzez odpowiednią manipulację klasy *row* reprezentującej przestrzeń

poziomą oraz podzieleniu jej na kolumny używając klas *col-md-12*. Za ciąg znaków *md* można podstawić inny identyfikator mówiący dla jakiej rozdzielczości będzie dedykowane rozwiązanie (xs, md, lg czyli small, medium i large). Za numer *12* można podstawić jakąkolwiek liczbę, która informuje o szerokości kolumny. Łączna szerokość musi wynosić 12 jednostek. Oznacza to, że możemy zbudować szablon z dwoma kolumnami na treść i menu z proporcją np. *col-md-9* i *col-md-3*. Ważne żeby ich suma wynosiła 12. Manipulacja tą proporcją jest jedyną możliwością wpłynięcia na szerokość poszczególnych pojemników i jest przeliczana automatycznie.

```
@media (min-width: 992px) and (max-width: 1199px) {
  .visible-md-block {
    display: block !important;
  }
}
@media (min-width: 992px) and (max-width: 1199px) {
  .visible-md-inline {
    display: inline !important;
  }
}
@media (min-width: 992px) and (max-width: 1199px) {
  .visible-md-inline-block {
    display: inline-block !important;
  }
}
```

Rys. 4.2. Przykładowe reguły *media-queries* wprost z bootstrapa

### 5.2.5. Przystosowanie interfejsu dla niepełnosprawnych

Każdy poważny projekt, którego odbiorcami jest duża grupa użytkowników, powinien być jak najbardziej przyjazny dla osób niepełnosprawnych. Istnieją specjalne wytyczne dla osób zajmujących się projektowaniem interfejsów systemów internetowych, niestety ciągle mało stron zostaje zoptymalizowana zgodnie z ich treścią. W stworzonym systemie zarządzania akademikami podjętych zostało wiele czynności, aby był to system jak najbardziej przyjazny użytkownikom.

Zbiór wytycznych dotyczących użyteczności i dostępności witryn internetowych został opisany w dokumencie WCAG 2.0. Autorem dokumentu jest konsorcjum W3C (ang. Word Wide Web Consorcium) zajmujące się ustandaryzowaniem tworzenia stron internetowych. Jest to grupa zrzeszonych ludzi i organizacji, które pracowały m.in. nad standardami HTML, XHTML, CSS, DOM, HTTP więc warto im ufać.

Dokument WCAG opisuje 3 poziomy użyteczności witryny

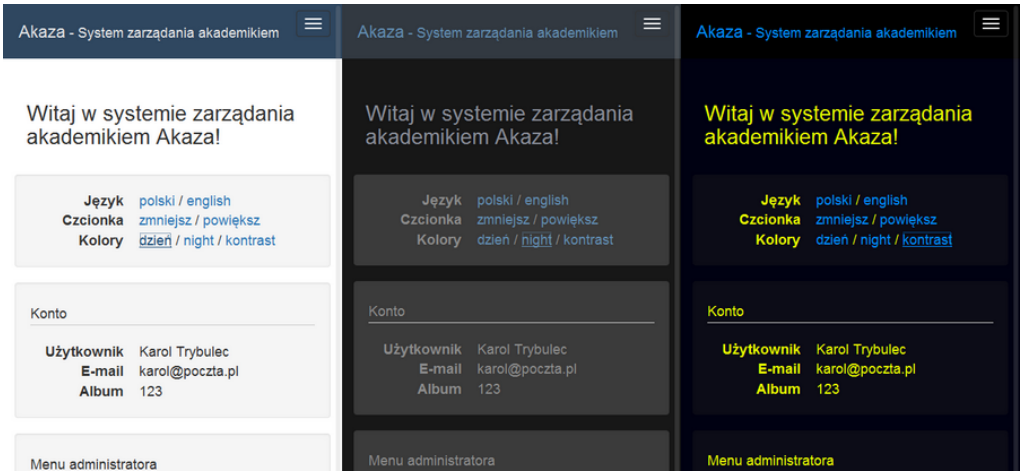
- Poziom A – musi zostać spełniony
- Poziom AA – powinien zostać spełniony
- Poziom AAA – będzie bardzo dobrze, gdy będzie spełniony

#### 5.2.5.1. Zmiana kolorystyki strony

Według standardu WCAG poziom AA stosunek kontrastu tła i tekstu powinien wynosić minimum 4.5:1 dla małego tekstu poniżej 18px wielkości. Dla dużego tekstu od 18px w górę

stosunek wynosić musi minimum 3:1. Dla poziomu AAA stosunek kontrastu dla małego tekstu to aż 7:1 a dla dużego 4.5:1.

Niniejszy projekt spełnia te standardy a dodatkowo udostępnia każdemu użytkownikowi prosty panel, umożliwiający szybką zmianę kolorystyki tła strony. Istnieją 3 wersje arkusza stylów: jasny przystosowany do dnia, ciemny przystosowany do nocy oraz jeden z bardzo silnym kontrastem.




Rys. 1.1. Trzy wersje kolorystyczne interfejsu

Mechanizm działa w bardzo prosty sposób. Do strony dołączone są 3 osobne arkusze stylów, każdy posiada atrybut *rel=”alternate stylesheet”*. Dzięki temu przeglądarka daje nam wybór, jakiego stylu ma używać. Styl można zmienić klikając w menu przeglądarki „Widok” > „Styl strony”. Dodatkowo zaimplementowana została zmiana stylu z poziomu strony. Po kliknięciu w nazwę odpowiedniego stylu kolorystycznego wywoływana zostaje funkcja JavaScript z parametrem nazwy stylu jaki nas interesuje. Uzyskuje ona uchwyt do wszystkich znaczników *<link rel…>* za pomocą których dołączane są style. Następnie tablica zostaje zbadana w pętli w poszukiwaniu stylu o atrybucie *name* równym temu przekazanemu do funkcji JavaScript. W momencie znalezienia stylu jego atrybut *disabled* zostaje ustawiony jako false, co automatycznie powoduje jego aktywowanie. Dodatkowo zostaje zapisana informacja w pliku COOKIE o aktualnie aktywnym stylu.

```
function switchStyle(styleName) {

    var i, linkTag;

    for (i = 0, linkTag = document.getElementsByTagName("link") ; i < linkTag.length; i++) {
        if ((linkTag[i].rel.indexOf("stylesheet") != -1)) {
            linkTag[i].disabled = true;
            if (linkTag[i].title == styleName) {
                linkTag[i].disabled = false;
            }
        }
        set_cookie("actualStyleCookie", styleName, 1, "localhost");
    }
}
```

Rys. 1.2. Funkcja aktywująca arkusz stylów o nazwie przekazanej w parametrze

Dodatkowo, podczas generowania widoku strony silnik Razor sprawdza treść ciastka, w którym zapisana jest nazwa aktywnego stylu. Jeżeli jest to styl *night* wtedy podczas dołączania stylu znacznikiem *link* nie zostaje dodany do niego parametr *alternative*. Dzięki temu prostemu zabiegowi nawet odświeżanie strony będzie uwzględniać wybrany przez użytkownika styl, bez najmniejszego mignięcia lub widocznego przełączania.

#### 5.2.5.2. Zmiana wielkości tekstu strony

Bardzo ważnym elementem usprawniającym użytkowanie systemu osobom niepełnosprawnym i słabowidzącym jest możliwość manipulowania rozmiarem liter. Mechanizm jaki został zastosowany w celu osiągnięcia efektu jest prosty i skuteczny. Rozmiary czcionki wszelkich tekstów, akapitów i nagłówków zostały ustalone atrybutem *font-size* z jednostką *em*. Jednostka ta powoduje, że rozmiar tekstu zostaje związany proporcjonalnie z rozmiarem tekstu rodzica. Oznacza to, że 1em może wynosić np. 20px, a 1.5em będzie wynosił wtedy 30px. Cały dokument zbudowany jest w hierarchii dziedziczenia rozmiaru czcionki, więc rozmiar czcionki zostaje ustalony tylko w jednym miejscu znacznikowi *body*, który jest naturalnym rodzicem każdego elementu w hierarchii DOM.

Kliknięcie na przycisk zmiany rozmiar czcionki zwiększa bądź zmniejsza czcionkę dla znacznika *body* w prosty sposób dodając do niego styl *inline*.

Dodatkowo informacja zostaje zapisana w ciastku, które jest sprawdzane podczas ładowania strony. Jeżeli wartość ciastka nie jest pusta, zostaje automatycznie przez silnik Razor dodana jako rozmiar czcionki dla *body*. Dzięki temu po wybraniu rozmiaru czcionki odpowiedniego dla danego użytkownika, zostaje on aktywny aż do momentu wygaśnięcia ciastka niezależnie od odświeżania strony lub przechodzenia na inne podstrony.

### 5.2.6. Wielojęzyczność

Aplikacja po stronie klienta szuka zmiennej globalnej JavaScript zdefiniowanej przez silnik Razor oznaczającą aktualnie aktywną wersję językową. Jej wartość zostaje przekazana jako argument do serwisu odpowiedzialnego za zwrócenie modelu, zawierającego tłumaczenia dla danego języka.

Model jest następnie przekazywany do zakresu kontrolera danego modułu i używany w widoku.

## 5.3. Warstwa bazy danych

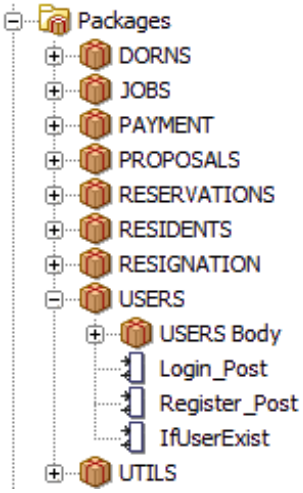
W projekcie użyta została baza danych Oracle Database. Naturalnym wyborem bazy danych dla technologii .NET wydaje się być baza SQL Server dostarczana przez Microsoft. Wybór bazy Oracle nie jest przypadkowy, baza ta została wybrana po kilku analizach. Okazało się, że lepiej spełnia wymagania w pełni skalowalnego systemu.

W bazie danych Oracle lepiej zorganizowanych jest kilka kluczowych kwestii. Lepiej radzi sobie ona z transakcjami. Każde połączenie do bazy danych jest osobną transakcją, którą można wygodnie cofnąć z poziomu kodu serwera. Jest to bardzo ważne w rozbudowanych systemach, gdzie wywołanie poszczególnych operacji prawie zawsze wiąże się z wykonaniem kilku

zapytań do bazy danych. W bazie SQL Server transakcje działają na zasadzie rozkazów wstawianych bezpośrednio do ciała funkcji/procedury.

### 5.3.1. Podział na paczki

Silnym argumentem na rzecz Oracle jest mechanizm paczek, który pomoże logicznie zorganizować strukturę po stronie bazy. SQL Server nie udostępnia paczek ani żadnego mechanizmu zastępczego. Jedynym sensownym rozwiązaniem wydaje się bardzo staranne nazywanie procedur a konkretniej ich przedrostków. W Oracle paczki pełnią nie tylko funkcję organizacyjną ale także zabezpieczają nas przed nieprzewidzianymi sytuacjami. W przypadku przypadkowej zmiany jakiejkolwiek procedury rozkompilowana zostaje cała paczka. Jest to dobry sygnał dla programisty, że celowo lub nieumyślnie zaszły w paczce zmiany. Problem wydaje się błachy, ale jest naprawdę poważny. Tworząc zaawansowaną logikę po stronie bazy danych łatwo o błędy. Wynika to z faktu braku idealnego mechanizmu kontroli wersji.




Rys. 1.1. Schemat paczek wraz z podglądem paczki *Users*

Paczki to coś więcej niż wygodna organizacja procedur i funkcji. Zapewniają modularność tworzonego przez programistę kodu. Każda paczka składa się z definicji oraz ciała paczki. Ciało paczki jest dokumentem zawierającym zbiór procedur i funkcji a jej definicja jest interfejsem dającym informacje o dostępnych procedurach, ich typach zwracanych i parametrach.

Kolejną ważną funkcjonalnością jest hermetyzacja. Paczki umożliwiają definiowanie procedur prywatnych lub publicznych.

### 5.3.2. Procedury i funkcje

W większości przypadków w projekcie użyte zostały procedury ze zdefiniowanym parametrem wyjściowym będącym typu *SYS_REFCURSOR*. Zdefiniowana paczka odpowiada nazwą modułowi AngularJS po stronie klienta oraz WebAPI kontrolerowi znajdującemu się po stronie serwera. W paczce znajdują się procedury charakterystyczne dla danego repozytorium.

```
PROCEDURE Payment_GBQ(userid IN number, paid IN number,
  refCursor OUT SYS_REFCURSOR) AS
BEGIN
  OPEN refCursor FOR
    SELECT
      PAY_ID Id,
      PAY_DATE Datev,
      PAY_AMOUNT Amount
    FROM AZ_USERS
      LEFT JOIN AZ_RESERVATIONS ON RES_USE_ID = USE_ID
      LEFT JOIN AZ_PAYMENTS ON PAY_RES_ID = RES_ID
    WHERE PAY_RES_ID IS NOT NULL AND USE_ID = userid AND PAY_PAID = paid;
END Payment_GBQ;
```

Rys. 2.1. Przykładowa procedura zwracająca listę płatności dla danego użytkownika

Dane zwracane są refkursorem. Różnica między kursorem a refkursorem jest taka, że te drugie mogą być przekazywane jako parametry do innych procedur. Zapewnia to dodatkową elastyczność.

Nazwy publicznych procedur zostały ustandaryzowane. Pierwszy człon jest słowem kluczowym mówiącym z czym związana jest procedura. Kolejnym członem jest typ operacji powiązany z RESTem np. POST, GET, DELETE itd. Wyjątkiem jest widoczny na załączonym rysunku 2.1. człon GBQ (z ang. Get by query). W stylu REST jest on tożsamy ze zwykłym bezparametrowym GETem, czyli pobraniem wszystkich rekordów. Pobranie określonego rekordu to już GET z parametrem *id*.

### 5.3.3. Zadania cykliczne (jobs)

W każdym rozbudowanym systemie pojawia się konieczność uruchamiania cyklicznych zadań, które wykonują określone operacja. Często są to zadania migracyjne danych. W systemie zarządzania akademikiem zdefiniowane zostało zadanie cykliczne, które raz w miesiącu zajmuje się naliczeniem płatności dla wszystkich zarejestrowanych użytkowników.

Do tego celu wykorzystany został mechanizm zadań udostępniany przez Oracle. Umożliwia on zdefiniowanie cyklicznego zadania wywołującego określoną proedurę. Wywołanie zadania ustalone zostało na końcówkę miesiąca i nastąpi w godzinach nocnych, wtedy gdy serwer bazy danych jest najmniej obciążony.

```
LOOP
  FETCH vRows into resId;
  EXIT WHEN vRows%notfound;

  BEGIN
    SELECT * INTO paymentsRow FROM
      (SELECT * FROM AZ_PAYMENTS WHERE PAY_RES_ID=resId ORDER BY PAY_DATE DESC)
    WHERE ROWNUM <= 1;

    rowYear:=to_char(paymentsRow.PAY_DATE,'YY');
    rowMonth:=to_char(paymentsRow.PAY_DATE,'MM');

    IF (rowYear!=actualYear OR rowMonth!=actualMonth) THEN
        dbms_output.put_line('Generowanie platnosci dla biezacego miesiaca');
        INSERT INTO AZ_PAYMENTS (PAY_RES_ID, PAY_DATE, PAY_PAID, PAY_AMOUNT)
          VALUES (resId, sysdate, 0, 500);
    ELSE
        dbms_output.put_line('Platnosc dla biezacego miesiaca juz istnieje');
    END IF;

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      dbms_output.put_line('Generowanie pierwszej platnosci dla tej rezerwacji');
      INSERT INTO AZ_PAYMENTS (PAY_RES_ID, PAY_DATE, PAY_PAID, PAY_AMOUNT)
        VALUES (resId, sysdate, 0, 500);
  END;

END LOOP;
```

Rys. 3.1. Fragment (główna pętla) zadania cyklicznego

Schemat działania zadania cyklicznego odpowiedzialnego za płatności wygląda następująco. W pierwszym kroku do kursora *vRows* zostają pobrane wszystkie rezerwacje w systemie z tabeli *AZ_RESERVATIONS*. Następnie uruchomiona zostaje pętla przetwarzająca każdą rezerwacje. Następuje sprawdzenie, czy dla danej rezerwacji są zdefiniowane jakieś płatności. Jeżeli nie ma płatności zostaje dodana płatność za bieżący miesiąc. W przypadku wykrycia płatności procedura analizuje tą wystawioną jako ostatnią. Jeżeli nie jest to płatność za obecny miesiąc wtedy zostaje takowa wygenerowana. Umieszczenie takiego warunku było konieczne, ponieważ płatność zostaje dodana do konta także w momencie zakwaterowania (akceptacji wniosku) studenta, po to aby nie czekać kilka dni lub tygodni na uruchomienie zadania cyklicznego.

# 6. Implementacja systemu

Implementacja systemu jest procesem polegającym na uzyskaniu działającego systemu informatycznego i wdrożeniu go w określonej fromie. W tym akapicie opisana zostanie ogólna koncepcja działania projektu wraz z informacjami o testach bezpieczeństwa jakie zostały na nim wykonane.

## 6.1. Mechanizm działania roli

Role w projekcie są reprezentowane za pomocą liczb i są kolejnymi potęgami liczby 2. Im większy wartość liczby przypisana do roli tym mniejsze prawa ma rola.

Role można ze sobą łączyć sumując kilka liczb ze sobą. Istnieje tutaj warunek, jeden użytkownik może być przypisany do kilku roli, jednak muszą być to role występujące po sobie w przypadku hierarchicznego ułożenia liczb. Mechanizm ten jest sprawdzony i skuteczny ponieważ dla uporządkowanego ciągu liczb będących potęgami liczby 2, suma liczb *x0..xn* nigdy nie będzie większa niż wartość liczby *xn+1*.

Aby używać mechanizmu roli wystarczy dowolny kontroler udekorować filtrem *[MyRoles]* wraz z rolami wylistowanymi po przecinku. Sprawdza on czy wartość roli dla danego użytkownika pobrana z bazy danych podczas procesu logowania zgadza się tymi ustawionymi w filtrze.

Jest to sposób dość naiwny. O jego implementacji zdecydował fakt małej złożoności systemu. W zależności od potrzeb mechanizm roli powinien zostać bardziej rozwinięty i bazować na operacjach bitowych. Dostęp do danego zasobu (lub rola) byłby reprezentowany przez bit na konkretnej pozycji. Dzięki takiemu rozwiązaniu proste operacje bitowe XOR, AND, OR pozwoliłby przetwarzać role w wygodniejszy sposób.

## 6.2. Struktura katalogów projektu

Struktura katalogów jest bardzo ważna ponieważ tworzy logiczną reprezentację projektu. Im bardziej skomplikowany projekt i im więcej osób nad nim pracuje, tym większą staranność trzeba przyłożyć do struktury katalogów.

### 6.2.1. Po stronie serwera – obszary

Głównym założeniem realizowanym po stronie serwera jest podzielenie projektu na obszary (z ang. areas). Jest to mechanizm udostępniany przez ASP.NET. Obszary mogą mieć nawet rangę osobnych podprojektów.

W niniejszym projekcie istnieją 3 obszary. Jeden odpowiedzialny za zwykłego użytkownika, drugi za administratora i trzeci za konto użytkownika w pojęciu ogólnym w tym rejestrację i logowanie. Każdy obszar posiada swój własny zestaw kontrolerów, repozytoriów, widoków i własną konfigurację routingów. Izolacja wrażliwych danych jest dodatkowym zabezpieczeniem. Użytkownik bez roli administratora nie ma żadnego dostępu do obszaru administratora itd.

Oprócz obszarów istnieje też gałąź główna projektu, do której wyciągnięte są podstawowe pliki odpowiedzialne za działanie aplikacji. Są to oczywiście:

- katalog kontrolerów
- widoków
- modelów
- repozytoriów
- ustawień startowych
- skryptów i treści

Do tego standardowego zestawu dołączone zostały także katalogi globalnych zasobów, klas pomocniczych (z ang. helpers) i filtrów.

Dodatkowym najważniejszym katalogiem jest katalog o nazwie *Angular*, przechowujący całą aplikację od strony warstwy klienta.

### 6.2.2. Po stronie klienta – moduły

Biblioteka AngularJS zostaje załadowana podczas wczytania plików **.js* zawierających informacje o bootstrapie bibliteki.

Logicznie cała aplikacja od strony klienta znajduje się w katalogu *~/Angular/* w głównym drzewie aplikacji. W tym katalogu znajdują się 2 katalogi odpowiadającego obszarom z podziału logicznego serwera. Trzeci katalog dotyczący autoryzacji nie został utworzony, ponieważ całość autoryzacji przejął na siebie serwer (wraz z widokami).

W obydwóch katalogach znajdują się informacje o załadowaniu (bootstrapingu) aplikacji Angulara. To który katalog reprezentujący obszar zostanie uruchomiony zależy od tzw. mechanizmu bundlingu, ponieważ dołączona on pliki **.js* tylko z katalogu związanego z danym obszarem. W przypadku wybrania katalogu uruchomiona zostaje aplikacja Angulara wraz z utworzeniem głównego zakresu aplikacji (z ang. *$rootScope*). Podczas ładowania aplikacji Angulara, do głównego modułu wstrzyknięte zostają zależności do wszystkich podmodułów, które znajdują się w podkatalogach, które z kolei grupują w całość poszczególne zestawy kontrolerów. Każdy katalog w głównym katalogu aplikacji AngularJS posiada:

- główny plik startowy aplikacji – uruchamia Angulara, tworzy aplikację dołącza zależności do pozostałych modułów
- katalog z pozostałymi modułami
- katalog z modelami (fabryki, serwisy)
- katalog z dyrektywami
- katalog z widokami dla kontrolerów i dyrektyw

## 6.3. Bundling i minifikacja czyli optymalizacja

Mechanizm bundlingu jest w ASP.NET bardzo pożądaną funkcjonalnością. Polega on na tworzeniu reguł dotyczących styli i skryptów ładowanych na poszczególnych podstronach serwisu. Konkretniej rzecz biorąc, programista definiuje własne zestawy bundla, które dzielą się osobno na zestawy stylów i osobno na zestawy skryptów. Następnie zestaw może być wczytany w określonym widoku za pomocą jednej linijki kodu.

Dzięki temu mechanizmowi w niniejszym projekcie AngularJS wie, który moduł startowy aplikacji powinien zostać włączony. Pliki *index.cshtml* dla poszczególnych obszarów (z ang. areas) mają w nagłówkach dołączone różne zestawy bundli, potrzebne do działania danego obszaru. Oto zestawy bundla użyte w projekcie:

- ~/bundles/scripts – zestaw odpowiedzialny za ładowanie wszystkich niezbędnych skryptów, m.in. bibliotek AngularJS, jQuery, AngularStrap, Moment.js, bootstrap.js i innych pomniejszych skryptów np. do powiększania czcionki i zmieniania tła strony
- ~/bundles/styles – zestaw odpowiedzialny za załadowanie wszelkich stylów używanych w projekcie
- ~/bundles/admin i ~bundles/student – dwa bardzo ważne zestawy odpowiedzialne za załadowanie plików startowych aplikacji AngularJS (zawierających w sobie regułę bootstrapa Angulara) oraz wszelkich modułów AngularJS dla danego obszaru

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bundles)
    {
        BundleTable.EnableOptimizations = true;

        bundles.Add(new ScriptBundle("~/bundles/student").Include(
                "~/Angular/student/start.student.js",
                "~/Angular/student/modules/*.js",
                "~/Angular/student/resources/*.js"
        ));

        bundles.Add(new ScriptBundle("~/bundles/admin").Include(
                "~/Angular/Admin/start.admin.js",
                "~/Angular/Admin/modules/*.js",
                "~/Angular/Admin/resources/*.js"
        ));
```

Rys. 3.1. Fragment konfiguracji mechanizmu bundlingu

Na załączonych obrazku 3.1. bardzo wyraźnie widać w jaki sposób uruchamiana jest aplikacja AngularJS. W zależności od modułu, który należy uruchomić, używana jest odpowiednia reguła bundlingu. Reguła wczytuje wszystkie pliki **.js* charakterystyczne dla danego obszaru. Najważniejszy wczytywanym plikiem jest plik startowy aplikacji AngularJS czyli *start.nazwa_obszaru.js*. Uruchamia on bibliotekę AngularJS, tworzy nową aplikację czyli moduł główny i podpina do niego wszystkie zależności wstrzyknięte za pomocą mechanizmu wstrzykiwania zależności, czyli wszystkie moduły wczytany z podkatalogów */modules/* i */resource/*.

Bundling posiada także cechę zwaną minifikacją. Domyślnie minifikacja jest wyłączona podczas kompilowania projektu w trybie debugowania, i jest włączona w wersji produkcyjnej. Wpis umieszczony w pliku konfiguracji bundlingu wymusza włączenie mechanizmu minifikacji za każdym razem.

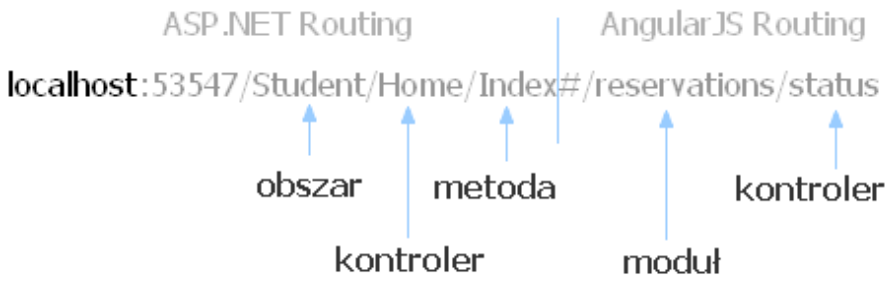
Minifikacja odpowiedzialna jest za maksymalne skondensowanie i zoptymalizowanie wczytywanych plików arkusza stylów oraz skryptów JavaScript. Wszystkie pliki arkusza stylów i skryptów łączone są w jeden plik i wczytywane za jednym razem. Dzięki minifikacji jest dostrzegalny spadek zużycia transferu oraz szybszy czas wczytywania plików (ponieważ wszystko jest wczytywane jednym zapytaniem zamiast dwudziestoma).

## 6.4. Konfiguracja routingów

W projekcie zaimplementowany jest skomplikowany układ routingów. Wyróżnić można następujące rodzaje routingów:

- Globalny routing kontrolerów ASP.NET
- Globalny routing kontrolerów WebAPI ASP.NET
- Lokalny routing określonego obszaru ASP.NET
- Routing kontrolerów określonego modułu AngularJS

Globalny routing ASP.NET zaczyna się po ukośniku od razu po nazwie domeny pod jaką znajduje się aplikacja. Standardowo wyróżnia się kontroler i jego metodę, jednak zostały one poprzedzone także nazwą obszaru, w którym występują. Routing AngularJS zaczyna się po #. Jest to znak nieprzypadkowy, ponieważ wszystko co znajduje się po znaku hasha nie jest brane pod uwagę przez żaden silnik HTTP.




Rys. 4.1. Struktura routingu projektu

W routingu ASP.NET zmienia się tylko nazwa danego obszaru, w zależności od roli zalogowanego użytkownika. Kontroler jest nie posiada żadnych innych metod oprócz metody *Index*. Metoda ta zwraca główny plik widoku zawierający dyrektywę *ng-view*, w którym zagnieżdżona zostaje aplikacja AngularJS.

Możliwe jest opracowanie adresu bardziej przyjaznego dla użytkownika, wystarczy usunąć z adresu nazwę kontrolera i nazwę metody. Efekt taki możliwy jest do osiągnięcia modyfikując lokalny routing danego obszaru. Modyfikacja polega na podaniu domyślnej nazwy dla kontrolera i metody:

```
public override void RegisterArea(AreaRegistrationContext context)
{
    context.MapRoute(
        "Admin_default",
        "Admin/{controller}/{action}/{id}",
        new { controller = "Home", action = "Index",
            id = UrlParameter.Optional }
    );
}
```

Rys. 4.2. Modyfikacja routingu dla obszaru znacznie skracająca strukturę linków

Dzięki temu prostemu zabiegowi, osiągnięty został krótszy, bardziej przyjazny i łatwiejszy do zapamiętania adres:

- przed localhost:53547/Student/Home/Index#/reservations/status
- po localhost:53547/Student#/reservations/status

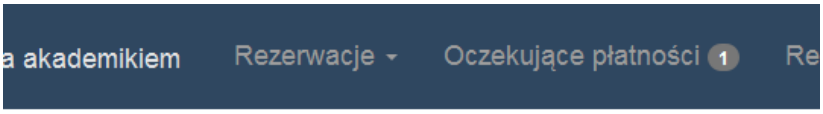
Routingi kontrolerów WebAPI określają pod jakim adresem dostępne będzie API aplikacji, służące jako interfejs wejścia/wyjścia dla danych. Nie został zmodyfikowany, nazwę kontrolera poprzedza człon *api/*, na drugim miejscu jest odpowiedni kontroler i metoda.

## 6.5. Widoki Razor i system powiadomień

Warstwa prezentacji danych została prawie w całości napisana przy użyciu biblioteki AngularJS, jednak w systemie mimo wszystko obecne są podstrony generowane przez silnik Razor. Są to ekrany logowania i rejestracji użytkowników oraz główny plik szablonu.

Główny plik szablonu definiuje ogólny szkielet interfejsu. Jego dodatkowym zadaniem jest wczytanie głównego zestawu bundli zawierającego najpotrzebniejsze skrypty i style. Sprawdza on także ciasteczka dotyczące wielkości czcionek i wybranej kolorystyki strony. W głównym widoku szablonu zdefiniowane są także główne menu poziome i pionowe. W miejscu wystąpienia treści zdefiniowana jest funkcja *@RenderBody()* pobierająca widok z poszczególnych obszarów, a te są już skonfigurowane do wystartowania aplikacji AngularJS.

Główny widok aplikacji zawiera referencję do modelu, z którego korzysta. Jest to prosty model reprezentujący liczbę powiadomień w danej zakładce. Przed zwróceniem widoku przez główny kontroler wysłane zostaje zapytanie zwracające ilość powiadomień dla poszczególnych elementów menu.




Rys. 5.1. Liczba oczekujących operacji dla danych pozycji w menu

Dzięki temu prostemu zabiegowi zyskujemy dużą funkcjonalność. Zarówno użytkownik systemu jak i jego administrator zostaje od razu po zalogowaniu wizualnie poinformowany gdzie czeka na niego jakaś operacja.

## 6.6. Testy aplikacji

Testowanie aplikacji jest bardzo ważne. Aspekt testowania opisuje literatura [13]. Wielu programistów pomija pisanie testów aby zaoszczędzić czas. Brak zawsze przypomina o sobie w przyszłości, gdy nie sposób pamiętać już o wszystkich zakamarkach systemu. Pozornie zaoszczędzony czas ucieka z nawiązką, ponieważ często błędy pojawiają się jeden po drugim, przy implementacji rzeczy pozornie nie związanych ze źródłem błędu.

Pisanie testów jest pracą żmudną i długą. Wymaga takiego samego nakładu pracy jak pisanie samej aplikacji. Z tego powodu testy powinny być pisane równolegle do powstawania kolejnych części systemu.

Należy pamiętać, testy stanowią dobre źródło informacji o skuteczności działania aplikacji, ale nigdy nie sprawdzają skuteczności jednego rozwiązania nad innym. Testy nie potwierdzą, czy aplikacja została zaprojektowana w sposób optymalny, czy użyte zostały odpowiednie wzorce i algorytmy – sprawdzony zostanie sam wynik.

Wykresy metryk zawarte w niniejszym dziale są wykresami punktowymi. Przedstawiają wartość metryk dla wszystkich funkcji w projekcie. Dane wykresu są zbiorami dyskretnymi a nie ciągłymi, jednak punkty zostały połączone bladymi liniami w celu zachowania czytelności.

### 6.6.1. Testy jednostkowe kontrolerów ASP.NET

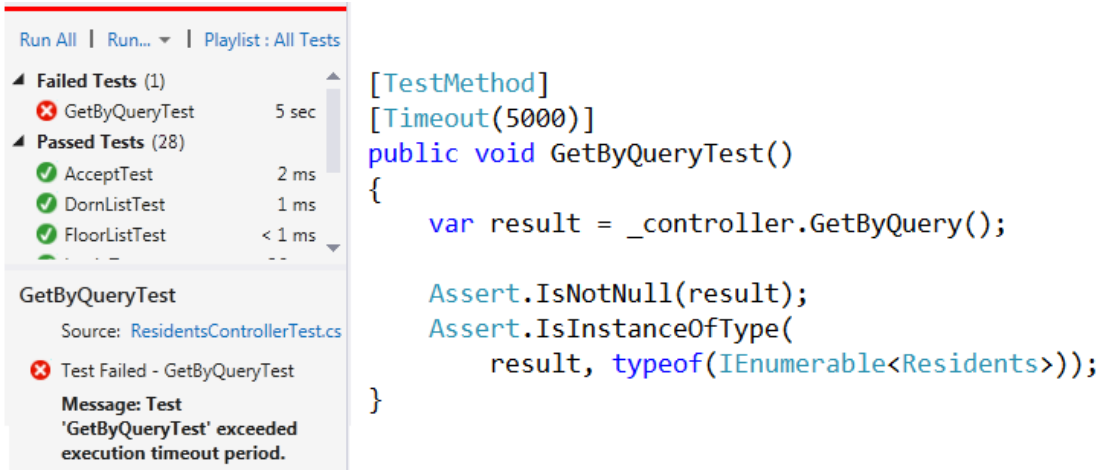
W celu pokrycia projektu testami utworzony został nowy projekt o nazwie *AkazaTesting*. Aby możliwe było testowanie kontrolerów WebAPI należy dodać do niego referencję do ASP.NET WebAPI2 Core. Dodatkowo należy dodać referencję do projektu *Akaza* w celu dostępu do jego kontrolerów. Dla każdego kontrolera utworzony został plik z testami dla każdej z jego metod.

```
[TestClass]
public class ProposalControllerTest
{
    private ProposalController _controller;

    public ProposalControllerTest()
    {
        _controller = new ProposalController(new ProposalRepository());
    }

    [TestMethod]
    [ExpectedException(typeof(ArgumentNullException))]
    public void AcceptTest()
    {
        _controller.Accept(null);
    }

    [TestMethod]
    public void DornListTest()
    {
        var result = _controller.DornList();

        Assert.IsInstanceOfType(result, typeof(IEnumerable<DornList>));
    }
```

Rys. 1.1. Fragment testów jednostkowych metod kontrolera ProposalController

W celu napisania testów, każdy kontroler z testami posiada prywatną referencję do testowanego kontrolera. Zostaje ona zainicjalizowana w konstruktorze. Tutaj bardzo dobrą robotę odgrywa nam mechanizm wstrzykiwania zależności, ponieważ dzięki niemu możemy w łatwy sposób przepiąć repozytoria kontrolerów. Po przepięciu repozytoriów można utworzyć repozytorium testowe, zwracające statyczne dane.

ASP.NET udostępnia nam bardzo różnorodne metody testowania danych. Dostępne są różne funkcjonalności z klasy *Assert* ale oprócz tego możliwe jest np. oznaczenie metody atrybutem *ExpectedException* i zdefiniowanie jakiego rodzaju błędu spodziewamy się po wywołaniu określonej metody.

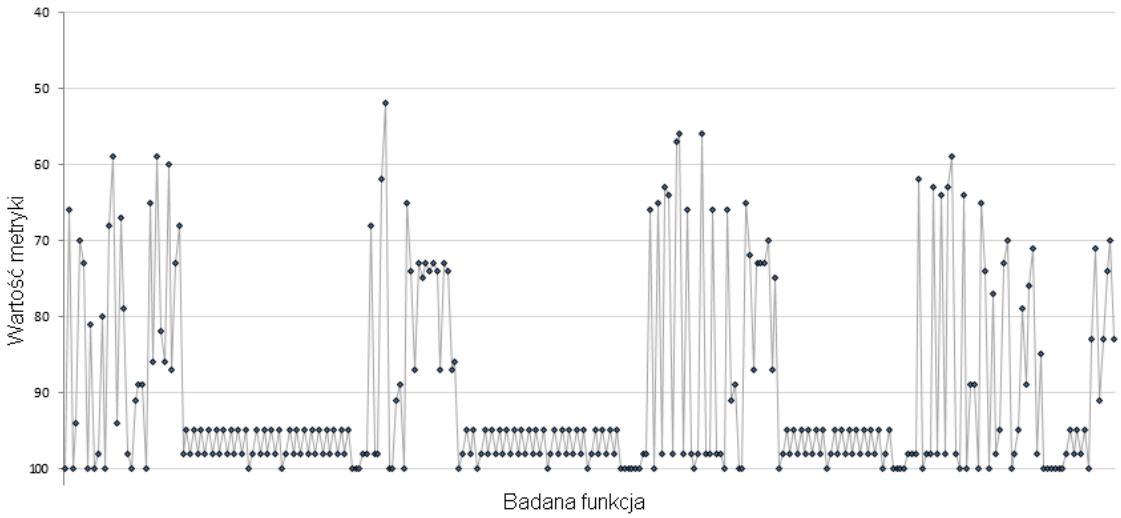



Rys. 1.2. Test nie przeszedł z powodu przekroczenia limitu czasu

Na rysunku 1.2. przedstawiony został bardzo ciekawy sposób definiowania maksymalnego czasu trwania testu. W przypadku przekroczenia wartości podanej w milisekundach w atrybucie *Timeout*, test zgłosi niepowodzenie. Limity czasowe mogą być bardzo pomocne w wykrywaniu problemów z połączeniem z bazą danych oraz przy źle skonstruowanych pętlach.

### 6.6.2. Metryka współczynnik utrzymywalności

Współczynnik utrzymywalności (ang. maintainability index) [14] jest wartością od 1 do 100 zaprezentowaną w sposób liczbowy. Pośrednio określa łatwość utrzymania i rozbudowy kodu. Im wyższa wartość tym lepiej. Wartość poniżej 20 punktów uznawana jest za kod złej jakości.




Rys. 2.1. Współczynnik utrzymywalności

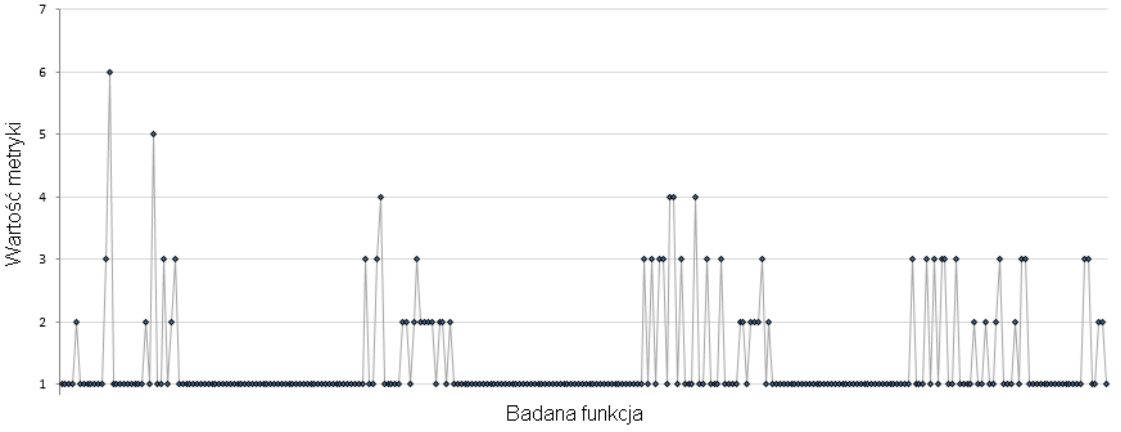
Aplikacja wiele razy osiąga poziom 100 a więc ten optymalny. W najgorszym momencie osiągnięta zostaje wartość 52. Odpowiada ona metodzie odpowiedzialnej za logowanie do systemu. Kolejną niską wartością są obarczone metody odpowiedzialne za pozyskiwanie listy akademików, listy pięter i listy pokoi. Są to operacje złożone wymagające wielu obliczeń i zapytań do bazy danych.

Wynik średni współczynnika utrzymywalności dla warstwy serwera aplikacji to 86.

### 6.6.3. Metryka złożoność cyklomatycznej

Złożoność cyklomatyczna [14] jest miarą złożoności kodu. Aby obliczyć tę metrykę należy narysować graf przepływu informacji dla danej metody. Na szczęście środowisko Visual Studio posiada podsumowanie, które pozwala wyliczyć metryki dla każdej metody danej klasy bez konieczności ręcznego przygotowywania grafów.

Wzór na złożoność cyklomatyczną to: M = E − N + 2P. Kolejno symbole oznaczają M – złożoność cyklomatyczna, E – liczba krawędzi grafu, N – liczba węzłów grafu, P – liczba grafów spójnych. Graf spójny to graf w którym każda para wierzchołków jest połączona ścieżką. W przypadku sporządzenia grafu dla jednej metody, wartość P zawsze będzie wynosić 1.



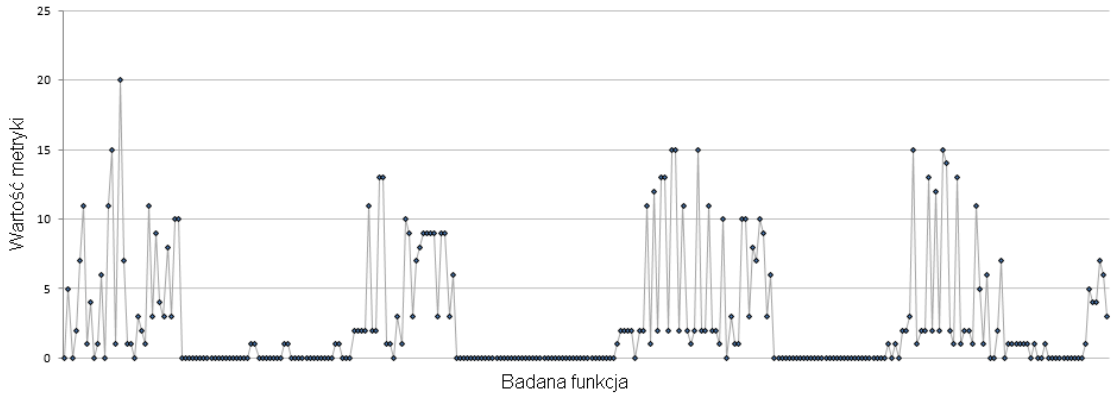

Rys. 3.1. Złożoność cyklomatyczna kodu

Najwyższą złożoność cyklomatyczną osiągnęła metoda *CreateKernel()*. Jest to metoda związana z mechanizmem wstrzykiwania zależności repozytoriów do kontrolerów. Posługiwanie się tym mechanizmem jest bardzo wygodne, dzięki niemu projekt zyskuje na skalowalności oraz kod jest łatwiejszy do testowania.

Niestety wszelkie udogodnienia często bywają obarczone pewnymi stratami wydajnościowi. Idea stosowania wzorców projektowych polega na lekkiej stracie wydajności względem zachowania porządku architektonicznego.

### 6.6.4. Metryka sprzężenia klas

Sprzężenie klas informuje o unikalnych powiązaniach między poszczególnymi klasami w obrębie pojedynczej metody. Brane pod uwagę są: typy zwracane, parametry, implementacje interfejsów, wystąpienia generyczne a także dekorowanie metod atrybutami i filtrami.




Rys. 3.1. Sprzężenie klas kodu

W projekcie ponownie największą wartość osiąga metoda *CreateKernel()*. Brzmi to dosyć paradoksalnie, ponieważ mechanizm wstrzykiwania zależności miał znacznie poprawić skalowalność projektu. Zgodnie z danymi odczytanymi z metryk dzieje się zupełnie na odwrót.

Kod dobrej jakości powinien mieć jak najmniejsze współczynniki sprzężenia klas, wtedy jest to kod łatwy do rozbudowy, poszczególne metody nie są od siebie zależne. Jest to założenie zgodne z jedną z zasad SOLID a konkretniej z zasadą odwrócenia zależności (szeroko opisana na Wikipedii [15]). Należy jednak przyjąć poprawkę. Testy i metryki pomagają analizować kod aplikacji jednak jest to analiza naiwna. Klasa platformy *Ninject* została uznana za najbardziej obciążającą, jednak dzięki użyciu tego narzędzia reszta projektu wypadła znacznie lepiej niż powinna.

Ponosząc świadomie pewne straty zyskujemy korzyści w wielu innych wymiarach. Jest to dla projektu zabieg opłacalny i pozytywny.

# Wnioski

Celem pracy było stworzenie systemu zarządzania domami studenckimi. Pomysł na projekt narodził się w chwili przeanalizowania istniejących rozwiązań, które są rozwiązaniami słabej jakości. Wiele z istniejących systemów tej kategorii nie jest dedykowana do obsługi akademików co powoduje, że posiadają nadmiar funkcjonalności i nie przewidują sytuacji wyjątkowych. Istniejące systemy dedykowane, są napisane w starych technologiach, można śmiało założyć, że nie są już rozwijane. Gdyby były – na pewno wspierałyby najnowsze trendy i standardy panujące na rynku. Ważnym aspektem tej pracy było także zaprezentowanie wzorca MVC zarówno po stronie klienta i serwera.

Wszystkie obrane cele zostały z sukcesem zrealizowane. Udało się stworzyć w pełni funkcjonalny system do zarządzania domami studenckimi, ze wszystkimi niezbędnymi funkcjonalnościami. Serwer działa w technologii ASP.NET MVC i pełni rolę szyny komunikacyjnej pomiędzy bazą danych a front-endem. Po stronie klienta znajduje się cały podsystem napisany w technologii AngularJS zajmujący się przetwarzaniem i manipulacją danych. Baza danych działa na silniku Oracle Database, która okazała się lepsza od konkurencji.

Podczas projektowania i budowy systemu duży nacisk położono na dopracowanie interfejsu użytkownika.. Użycie najnowszych technologii na rynku pozwoliło osiągnąć aplikację profesjonalną, elastyczną, dostosowaną do wszelkich rozdzielczości ekranu a także przyjazną osobom niepełnosprawnym i słabowidzącym. Podczas realizacji projektu były brane pod uwagę standardy opisane w międzynarodowych dokumentach WCAG, nad którymi czuwa konsorcjum W3C. Wymienione konsorcjum zajmuje się ustandaryzowaniem internetu i zrzesza wiele firm z branży IT z całego świata. To właśnie ich wiedza i doświadczenie jakie niewątpliwie posiadają, powinna ostatecznie przekonać każdego inżyniera do dążenia do ustalonych przez nich standardów.

Po stronie serwera stworzona i zaimplementowana została struktura bardzo dobrze reagująca na rozbudowę. Dzięki wstrzykiwaniu zależności system jest skalowalny, w każdym momencie można dopisać do niego dowolny kontroler zwracający konkretne dane. Zmianą jednej linijki kodu można w dowolnym celu przepiąć repozytoria zwracające dane konkretnego modułu. Jest to aspekt szczególnie ważny podczas testowania – nie jest problemem przepięcie repozytorium głównego na testowe zwracające statyczne dane (bez korzystania z bazy danych). Systemy oparte na architekturze MVC są przyjazne programistom, wdrożenie do istniejącego projektu jest prostsze tak samo jak zrozumienie zasad jego działania.

Szeroko został omówiony wzorzec MVC a także zależności pomiędzy wzorcami dla niego pochodnymi np. MVVM. Na konkretnych przykładach pokazane zostały różnice pomiędzy architekturą MVC a jego odmianą MVVM. Szeroka analiza tego tematu daje dobre zobrazowanie zasad, jakimi powinni kierować się architekci oprogramowania, chcący dobrze rozplanować warstwę klienta. Ta, mimo że dawniej zaniedbywana i mieszcząca się jedynie w pojedynczym pliku widoku HTML, dzisiaj rozmiarem i złożonością kodu niczym nie odbiega od warstwy serwera. Aplikacje z rozbudowanym front-endem są coraz bardziej potrzebne i zaczynają dominować na rynku. Rozumienie architektury po stronie serwera jak i klienta powinno być dla każdego programisty kwestią kluczową.

Mimo, że używanie wielu współgrających wzorców projektowych tworzących szkielet całego systemu niesie ze sobą pewną redundancję, to jest to efekt nieunikniony. Zalety jakie przynosi takie rozwiązanie i tak są większe. Straty wynikłe z zastosowania wzorców projektowych widać dobrze w rozdziale poświęconym testowaniu.

# Literatura


[1] Marcinkowski B., Wrycza S., Wyrzykowski K.: *Język UML 2.0 w modelowaniu systemów informatycznych*, 2006.

[2] Stevens P., *Using UML: Software Engineering with Objects and Components (2nd Edition)*, 2007.

[3] Troelsen A., *C# 2015 and the .NET 4 Platform*, 2011

[4] Freeman A., *Pro AngularJS*, 2015

[5] Walczak T.: *Oracle Database 12c. Programowanie w języku PL/SQL*, 2015.

[6] Rahman S. F., *Jump Start Bootstrap*, 2015

[7] Radford S., *Learning Web Development with Bootstrap and AngularJS*, 2015

[8] Gamma E., Helm R., Johnson R., Vlissides J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*, 1997.

[9] Metsker S. J., *Design Patterns in C#*, 2005

[10] Bukowska E., Fryźlewicz Z., Nikończuk D., *ASP.NET MVC 4. Programowanie aplikacji webowych*, 2013

[11] Mehta B., *RESTful Java Patterns and Best Practices*, 2015

[12] Barker T., *High Performance Responsive Design: Building Faster Sites Across Devices*, 2015

[13] Beck K., *Test Driven Development: By Example*, 2014

[14] Dokumentacja MSDN "Metryki testowania", *https://msdn.microsoft.com/pl-pl/library/bb385914.aspx*, dostęp dnia: 2016-01-11

[15] Wikipedia "Solid principles", *https://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29*, dostęp dnia: 2016-01-11

# Summary

Main goal of this thesis was to make a student houses management system. Systems which are actually existing on the market contains a lot of errors and inconveniences, they have low quality. Many of this systems are not dedicated to support student houses, but they are universal tools for many other purposes. Thereby, they contains a lot of functionalities which won't be used.

Existing systems are wrote in old technologies and they are not developed anymore. Because if they were, they would implement latest solutions and technologies actually available. Important part of this thesis is to implement MVC pattern both in client and server side.

Completely working system with all required functionalities has been wrote without any problems. All goals have been realized successfully. Server side application is working in ASP.NET MVC5 technology. Server acts communication bus role. It is responsible for retrieving data from database and sending it to the front-end. It is also retrieving data from front-end and saving into a database.

By client side there is an subsystem wrote in AngularJS platform. Its main goal is to process data from user interface. Project using Oracle Database, which seems to be the best available database at this moment. Oracle database provides multiple solutions for many problems like oracle database packages and great transitions control. Packages helping to keep good project structure, and transactions are easy to handle.

Created system is very good elaborated from user interface side. Using newest technologies gives great results. Application is adjusted for both mobile devices and desktops. Application is disabled-friendly. It provides a lot of additional functions like text-size changing, background and text color change for achieve better contrast, and multilingualism. During the project realization WCAG document standards and guidelines were taken into account. WCAG is a document with set of standards that should be followed during making websites. WCAG is a product of W3C consortium, they knowledge and experience is very important and valuable.

By server-side it is implemented structure which is very well to expand. By using dependency injection mechanism project is really scalable. In every moment it is possible to add any controller or repository into it. Systems which are using MVC and repository pattern are friendly to programmers and easy to learn to.

In this thesis MVC pattern has been discussed and compared to the MVVM variety. Differences between MVC and MVVM are shown on attached pictures.