**Katolicki Uniwersytet Lubelski Jana Pawła II**

**Wydział Matematyki, Informatyki i Architektury Krajobrazu**

**Instytut Matematyki i Informatyki**

Informatyka,

Studia stacjonarne II stopnia

**Władysław Werenicz**

Nr albumu 134439

**Tworzenie stron internetowych przy użyciu Meteor.js**

Praca magisterska

napisana na seminarium sztuczna inteligencja pod kierunkiem

**dr. hab. Ryszarda Kozery**

Lublin 2018

# Spis treści

[Wstęp 5](#_Toc454191634)

[Rozdział 1. Zapoznanie z pojęciami oraz technologiami 7](#_Toc454191635)

[1.1. NET / C# 7](#_Toc454191636)

[1.2. Programowanie obiektowe, paradygmaty 8](#_Toc454191637)

[1.3. SOLID 9](#_Toc454191638)

[1.4. UML 10](#_Toc454191639)

[1.5. Problemy projektowania nowoczesnych systemów informatycznych 13](#_Toc454191640)

[Rozdział 2. Wzorce projektowe 15](#_Toc454191641)

[2.1. Pojęcie wzorca projektowego 15](#_Toc454191643)

[2.2. Klasyfikacja wzorców 15](#_Toc454191644)

[2.3. Factory Method 16](#_Toc454191645)

[2.4. Abstract Factory 18](#_Toc454191646)

[2.5. Singleton 21](#_Toc454191647)

[2.6. Adapter 23](#_Toc454191648)

[2.7. Facade 25](#_Toc454191649)

[2.8. Composite 28](#_Toc454191650)

[2.10. Proxy 31](#_Toc454191651)

[2.11. Command 33](#_Toc454191652)

[2.12. Iterator 35](#_Toc454191653)

[2.13. Observer 37](#_Toc454191654)

[2.14. Strategy 41](#_Toc454191655)

[Rozdział 3. Stworzenie aplikacji webowej z użyciem wzorców 45](#_Toc454191656)

[3.1. Opis aplikacji 45](#_Toc454191658)

[3.2. Warstwy aplikacji 48](#_Toc454191659)

[3.3. Szczegóły implementacji 50](#_Toc454191660)

[Zakończenie 53](#_Toc454191661)

[Bibliografia 55](#_Toc454191662)

# Wstęp

Celem mojej pracy jest przedstawienie najważniejszych oraz najczęściej używanych wzorców projektowych i architektonicznych, stosowanych przy tworzeniu nowoczesnych aplikacji webowych na platformie .NET w języku C#.

Czasami nie zdajemy sobie sprawy o ważności świadomego wykorzystywania takich wzorców w codziennej pracy nad projektem. Bez nich byłoby trudno wyobrazić sobie możliwość tworzenia skomplikowanego nowoczesnego oprogramowania. Dla mnie osobiście zaskoczeniem było to, że niektórzy moi koledzy nie mogli nazwać przynajmniej trzech wzorców z katalogu GoF. Jest to bardzo zła tendencja, jako że brak znajomości wzorców projektowych w nowoczesnym projektowaniu rozwiązań informatycznych można przyrównać do braku znajomości całek w nowoczesnej matematyce oraz fizyce. Wzorce projektowe są bowiem nowym poziomem abstrakcji w informatyce, tak samo, jak programowanie obiektowe. Abstrakcja to pojęcie, które określa postęp nie tylko technologiczny, ale też całej ludzkości. Im większy jest poziom abstrakcji, tym bardziej złożony system można zbudować.

W rozdziale pierwszym dokonano wprowadzenia do pozostałych części pracy. Przedstawiono kroki, niezbędne do zrozumienia wzorców projektowych w języku programowania C#. W kolejnych podrozdziałach opisano platformę .NET i wyjaśniono główne stereotypy języka C#. Zaprezentowano pojęcia programowania obiektowego oraz SOLID, ponadto został dokonany wstęp do języka modelowania UML oraz przeanalizowano problemy, z którymi się spotyka architekt przy projektowaniu systemu informatycznego.

W drugim rozdziale skupiono się na wzorcach projektowych. Dokładnie wyjaśniono, czym jest wzorzec, a następnie sklasyfikowano wszystkie wzorce projektowe. Dalej przeanalizujowano 11 najczęściej używanych wzorców projektowych w .NET oraz 2 wzorce architektoniczne.

W ostatnim rozdziale przedstawiono aplikację webową, napisaną z użyciem wzorców oraz najbardziej nowoczesnych na daną chwile technologii.

# Rozdział 1. Zapoznanie z pojęciami, technologiami oraz instrumentem developerskim

## Strona Internetowa

W tej pracy termin „Strona internetowa” (ang. *Website*) będzie używane w tym samym kontekście jak i termin „Aplikacja internetowa” (ang. *Web application, Web app*). Związane jest to z szybkim rozwojem możliwości oraz wymagań produktów webowych. Dzisiaj nawiat najprostsze strony internetowe opierają się na zestaw technologii backendowych / API. Ten aspekt zezwala mi uważać nowoczesną stronę internetową normalną aplikacją, porównywalną z dekstopową ale o pewnych specyficznych właściwościach. Warto dodać, że dzisiaj są używane technologie, pozwalające tworzyć natywne desktopowe aplikacje w oparciu o Node.JS, o którym opowiem później w pracy. Przykładem takiej technologii jest Electron. Przykładem aplikacji jest IDE Visual Studio Code, który został użyty do napisania aplikacji, wspierającą pracę magisterską. W przypadku takiej aplikacji, użytkownika nie dotyka specyfika web aplikacji – potrzeba połączenia internetowego oraz użycia przeglądarki internetowej. Jeżeli zwrócić uwagę na stronę internetową spotkamy pewne aspekty które wymuszają developera myśleć inaczej. Przykładem takich aspektów są: bezstanowy web (ang. Stateless *Web*), ograniczone możliwości oprogramowania 3D grafiki, rozmiar oraz wydajność aplikacji. Główną zaletą aplikacji opartą o technologie webowe jest brak potrzeby instalacji aplikacji, więc istnieje dostęp do niej z dowolnego urządzenia połączonego z internetem. Za przechowywanie oraz bezpieczeństwo danych odpowiada firma tworząca web aplikację, co zdejmuje część odpowiedzialności z użytkownika.

## Architektura aplikacji webowej

Architekura aplikacji webowej – to jeden ze sposobów spojrzenia na aplikację z wyzszego poziomu. Ja miałem możliwość pracować z dwoma ze trzech architektur przedstawionych poniżej. Ale zanim można będzię zanalizować te architektury, potrzebujemy określić pewne pojęcia.

Klient (ang. *Client*) – tym pojęciem możemy oznaczyć część aplikacji która działa po stronie użytkownika aplikacji albo też zestaw narzędź oraz środowisko w którym jest odpalana aplikacja po stronie użytkownika. Kod odpalany po stronie klienta najczęściej jest odpowiedzialny za renderowanie (wyświetlenie) danych biznesowych dostarczanych częścią serwerową oraz obsługę interakcji użytkownika z aplikacją.

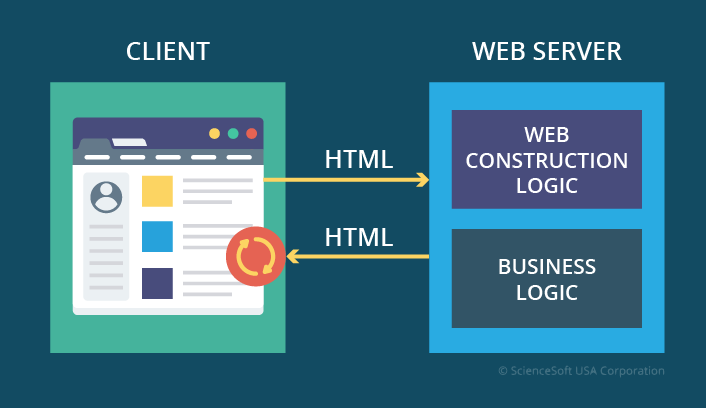
Serwer (ang. *Server*) – tak samo jak wyżej, tym pojęciem można określić jak część implementacji aplikacji tak i środowisko wykonania tego kodu. Implementacja serwerowa może być odpowiedzialna za przygotowanie poszczególnych elementów do renderowania, operacje na bazie danych, implementację logiki biznesowej, pracę z plikami i t.d.

Intrygującym aspektem jest to, że praca nad częścią klienta zajmuje dużo więcej czasu. Jest to związane, że środowiska klientskie (przeglądarki) oraz narzędzia od dłuższego czasu nie były wystarczająco standardyzowane. Wielki wpływ to miało na szybkość oraz jakość napisania stylów strony. Przez dłuższy czas nie było możliwości łatwego wycentrowania elementu, trzeba było używać tak zwanych css-trików. Były tworzone dodatkowe elementy-kontenery, ustawiane właściwości takie jak float, text-align, position. Ale jak rozmiar okna zostawał zmieniony element zmieniał swoją pozycję. Dzisiaj jeżeli nie potrzebujemy wspierać przeglądarki starsze niż Internet Explorer 10 mamy możliwość używać nowe techniki do ustawiania elementów na stronie.

Teraz kiedy znamy różnicę między częścią klientską oraz serwerową, możemy porównać róźne architektury, które były używane do tworzenia aplikacji webowych poprzez ostatnie 15 lat.

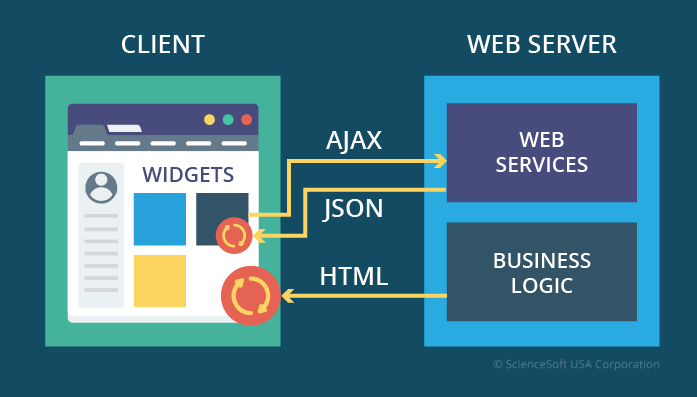
* Legacy web application

To jest pierwsza i podstawowa architektura aplikacji Web. Serwer posiada logikę budowy aplikacji (generuje html, css, js strony) i też jest odpowiedzialny za logikę biznesową oraz obsługę warstwy danych. Interakcja użytkownika powoduje odesłanie żądania http do serwerowej części. Serwer obsługuje to żądanie, zapisuje bieżący stan w bazie danych, tworzy nowy widok strony i wysyła ją do klienta. Po każdej interakcji z systemem aplikacja musi się przeładować o ile renderuje się całkiem nowa strona. Przykładową technologią jest ASP.NET Web Forms, JSP i t.d. Jest to bardzo bezpieczna architektura o ile za wszystko odpowiada serwer gdzie zwykły użytkownik nie posiada dostępu.



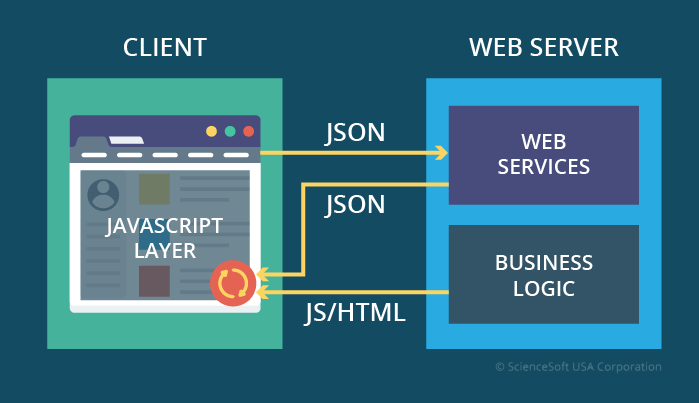
* Widget web application

Jest to kolejny krok, jeżeli chodzi o budowę widoku strony internetowej. Strona składa się z bloków o nazwie ‘widget’. Jeżeli ten blok potrzebuje otrzymać nowe dane, wysyłane jest AJAX żądanie, które zwraca porcję danych w formacie JSON. Takie wysyłanie działa asynchronicznie i nie powoduje potrzeby przeładowania całej strony. W tej architekturze rozmieszczono jest więcej logiki po stronie klienta co wprowadza większe ryzyko związane z bezpieczeństwem.



* Single-page web application

Najbardziej popularna dziś architektura, gdzie pojedyncza strona jest pobierana tylko raz. Po stronie klienta jest warstwa View-Model która komunikując się z serwerem aktualizuje widok w czasie rzeczywistym. Ilość danych przesyłana przez sieć jest dużo mniejsza niż w przypadku z widgetem, jest to bardzo elastyczna i przyjazna użytkownikowi architektura. Oprogramowanie napisane w tej architekturze można łatwo przekształcić w aplikację mobilną za pomocą hybrydowych opakowań takich jak Cordova.



Meteor.js ułatwia oprogramowanie bazujące się na trzecim typie architektury, dostarczając też narzędzia oraz interfejsy do budowy aplikacji pod platformy mobilne z użyciem Cordova.

## Meteor.js

Meteor.js – to platforma najwyższego poziomu, zawierająca cały podzbiór innych frameworków oraz narzędzi celem których jest umożliwienie tworzenia ‘reaktywnych’ aplikacji dla platform web oraz mobile, przy użyciu jednego języka oprogramowania - JavaScript.

Meteor CLI (ang. *Command Line Interface*) – jeden z bazowych składników platformy Meteor.js. Jest to narzędzie-aplikacja konsolowa, która mocno wspiera proces tworzenia aplikacji na platformie Meteor.js. Przy pomocy tego narzędzia mamy możliwość tworzenia bazowego szkieletu aplikacji, budowę aplikacji, insstalację pakietów Atmosphere, podbijanie wersji Meteor.js, generowanie hybrydowej aplikacji mobilnej i dużo więcej.

Jednym z ważnych aspektów na które patrzymy pod czas wyboru platformy lub zestawu narzędź do napisania oprogramowania jest zestaw gotowych bibliotek (API), które są ich ważną częścią. Często jakość tych bibliotek ma ogromny wpływ na sukces platformy. Im większy i lepej odokumentowany jest zestaw tych bibliotek, tym bardziej elastycznie można decydować o rozwiązaniu problemu. Dzisiaj rzadko zdarza się implementować funkcjonalność lub całą bibliotekę samodzielnie. Preferowanym jest użycie gotowego rozwiązania, odpowiedzialność za jakie bierze na siebie autor lub społeczeństwo tworzące ten pakiet. W przypadku popularnych bibliotek większa jest szańsa na to, że będą odnalezione i poprawione błędy jeszcze zanim zaczniemy ich używać w swojej aplikacji.

Meteor API – jest połączeniem pakietów własnych z pakietami, które dostarcza społeczeństwo JavaScript oraz Node.JS. Własne pakiety są dostosowane do platformy Meteor.js i często używają bazowe funkcjonalności platformy, co utrudnia ich przeniesienie do schowku globalnego NPM. Takie pakiety są dostępne w specjalnym globalnym schowku o nazwie Atmosphere.js. Na stronie *atmospherejs.com* możemy znaleźć takie gotowe pakiety i swobodnie je używać w swojej aplikacji.

Warto zwrócić uwagę na to, że można bez ograniczeń korzystać z globalnego schowku NPM, który jest bardziej popularny niż Atmosphere. To znaczy że to rozwiązanie może być używane na innych platformach, a to powoduje lepsze wsparcie tego pakietu poprzez społeczeństwo.

Meteor Build Tool – narzędzie budowy aplikacji Meteor pod przeglądarkę oraz urządzenia mobilne. Jest częścią Meteor CLI. Ułatwia instalację projektu, zawiera swój określony protokół ładowania plików oraz poszczególnych części kodu. Wspiera najnowsze wersje języka JavaScript (ES 2015).

## Struktura folderów projektu

Decyzja o tym gdzie rozmieszczać pliki oraz foldery w Meteor.js jak i w innych platformach ma wpływ nie tylko na wygodę pod czas pracy z projektrem ale i na samo działanie aplikacji. Już wiemy, że pisząc aplikację na platformie Meteor.js musimy orientować się na architekturę SPA (ang. Single Page Application). To znaczy, że większa część plików będzie odpalana po stronie klienta, część plików będzie odpalana po stronie serwera. Ale to nie wszystko. Posługując się podejściem Uniwersal JavaScript, możemy odpalić ten sam kod w obu środowiskach, przy czym ta sama linia kodu może dokonywać róźne operacje. Żeby móc kontrolować to, gdzie chcemy odpalać nasze instrukcje, Meteor.js wprowadza protokół ładowania plików oraz dodatkowe zmienne środowiskowe.

Niżej określona jest strategia ładowania plików w projekcie:

* Najpierw ładowane są pliki HTML.
* Pliki o nazwie ‘main.\*’ ładowane są na końcu.
* Dalej ładują się pliki z folderu ‘/lib’.
* Później pliki ładują się według zagnieżdżenia, później według nazwy alfabetycznie.

Foldery specjalne:

* ‘/imports’ – pliki muszą być załadowane manualnie, za pomocą dyrektywy JavaScript ‘import’.
* ‘/node\_modules’ – pliki muszą być załadowane manualnie, zawiera zainstalowane zależności NPM.
* ‘/client’ – każdy plik w tym folderze automatycznie będzie załadowany po stronie klienta.
* ‘/server’ – każdy plik w tym folderze automatycznie będzie załadowany po stronie serwera.
* ‘/public’ – pliki serwowane klientowi ‘as-is’. Najlepsze miejsce do przechowywania zdjęć, czcionek i t.d.
* ‘/private’ – pliki ładowane po stronie serwera i są dostępnie wyłącznie na serwerze przy pomocy ‘Assets API’.
* ‘/tests’ – pliki nie są ładowane nigdzie.

Meteor także ignoruje podane foldery specjalne:

* .meteor
* .git
* packages
* cordova-build-override
* programs

Pliki, które nie są wrzucone do folderu specjalnego odpalane są jak po stronie klienta tak i po stronie serwera. Jest możliwość imperatywnie wskazać platformie, jaki kawałek kodu chcemy odpalić na poszczególnej stronie. Dokonać tego możemy za pomocą zmiennych środowiskowych: ‘Meteor.isClient’ oraz ‘Meteor.isServer’.

Najbardziej popularnym podejściem ułożenia bazowej struktury projektu jest umieszczenie większości kodu do folderu ‘imports’, gdzie można łatwo strukturalnie poukładać poszczególne warstwy projektu. Później ten kod jest aggregowany do głównych plików i ładowany do modułów w folderach ‘client’ oraz ‘server’.

## Problemy projektowania nowoczesnych systemów informatycznych

Głównym problemem projektowania nowoczesnych systemów informatycznych jest zmniejszenie liczby związków między poszczególnymi klasami [1].

Duże systemy programistyczne są dość trudne w projektowaniu i wspieraniu, dlatego, dla ułatwienia pracy, duże systemy rozbijamy na podsystemy. Głównym zadaniem jest zmniejszenie liczby związków-relacji między klasami, czyli zmniejszenie liczby wzajemnych zależności klas.

Zależność (ang. *dependency*) – to termin techniczny opisujący liczbę istniejących związków-relacji.

Jeżeli na diagramie widzimy, że jakaś klasa ma dużo związków z innymi – mówi się, że taka klasa jest mocno uzależniona od innych. Także można powiedzieć, że praca egzemplarza podobnej klasy będzie zależeć od pracy egzemplarzy innej. Zależności klas powodują tak zwane przywiązania.

Przywiązanie – termin logiczny, który zmusza developera do zastanowienia się nad sensem zależności. Ze strony programowania obiektowego, przywiązania mogą istnieć w dwóch postaciach: dobre – biznesowe przywiązania lub złe – przywiązania techniczne.

Przywiązania biznesowe wyrażają wymagania biznesu. Na przykład klasa Customer jest związana związkiem-relacją asocjacji z klasą Order.

Przywiązania techniczne, z innej strony wyrażają potrzeby systemowe. Na przykład, kiedy klasa Customer jest związana związkiem asocjacji z klasą DataSet.

Niestety bez pewnej liczby przywiązań technicznych nie da się zaprojektować systemu. Przywiązanie techniczne może być nazwane dobrym wtedy i tylko wtedy, kiedy zależne encje są umieszczone w różnych warstwach systemu. Na przykład Customer w Business Layer, DataSet w Data Layer. W takim przypadku przy analizie logiki biznesowej systemu można zignorować związki prowadzące do warstw o niższych poziomach abstrakcji.

Przy projektowaniu systemu musimy zmniejszyć liczbę przywiązań oraz pamiętać o  zachowywaniu logicznej całości sensu procesu, który modelujemy. Moc zależności można określić przy pomocy terminu ‘coupling’.

Coupling jest miarą zależności. Coupling posiada wiele metryk pozwalających na  określenie mocy zależności. Opis tych metryk nie wchodzi w kontekst danej pracy.

# Rozdział 2. Frontend



## Pojęcie wzorca projektowego

W

# Rozdział 3. Backend

## Pojęcie wzorca projektowego

# Rozdział 4. Tworzenie aplikacji webowej na platformie Meteor.js



## Opis aplikacji

Aplikacja została napisana głównie w języku JavaScript przy użyciu platformy meteor.js oraz zestawu narzędź Google oraz Amazon. Aplikacja składa się z dwóch głównych części: części obsługi właściciela samochodu oraz części obsługi właściciela stacji napraw samochodów. Aplikacja zezwala właścicielowi samochodu zgłosić defektywną część blachy samochodu, załączyć materiały pomocnicze (zdjęcia lub wideo miejsca z defektem). Aplikacja agreguje podobne zgłoszenia po czym stacja napraw samochodu może wyszukać zgłoszenie, wystawić ocenę / ofertę i wysłać swoje dane do właściciela samochodu. Właściciel samochodu oczekuje na oceny i ma możliwość zkontaktować się z interesującą stacją naprawy, która odpowiada jego ktyrteriom. Aplikacja posiada system zarządzania kontem użytkownika, możliwość opłaty kartą taryfy PRO (zaślepka ze względu bezpieczeństwa) oraz narzędzia geo.

Zostały wyodrębnione następujące przypadki użycia dla właściciela samochodu:

* Logowanie do systemu oraz wylogowanie się z systemu.
* Rejestracja użytkownika w systemie.
* Edytowanie profilu, ustawień.
* Wybór taryfy oraz zapis karty płatniczej.
* Dodanie nowego samochodu.
* Tworzenie zgłoszenia/zamówienia.
* Ładowanie ograniczonej ilości zdjęć / wideo z defektem, możliwość zarządzania tymi materiałami (usunięcie / wybór prioritetu).
* Wyszukiwanie propozycji stacji napraw samochodów.
* Strona informacyjna aplikacji.

Zostały wyodrębnione następujące przypadki użycia dla właściciela stacji napraw samochodów:

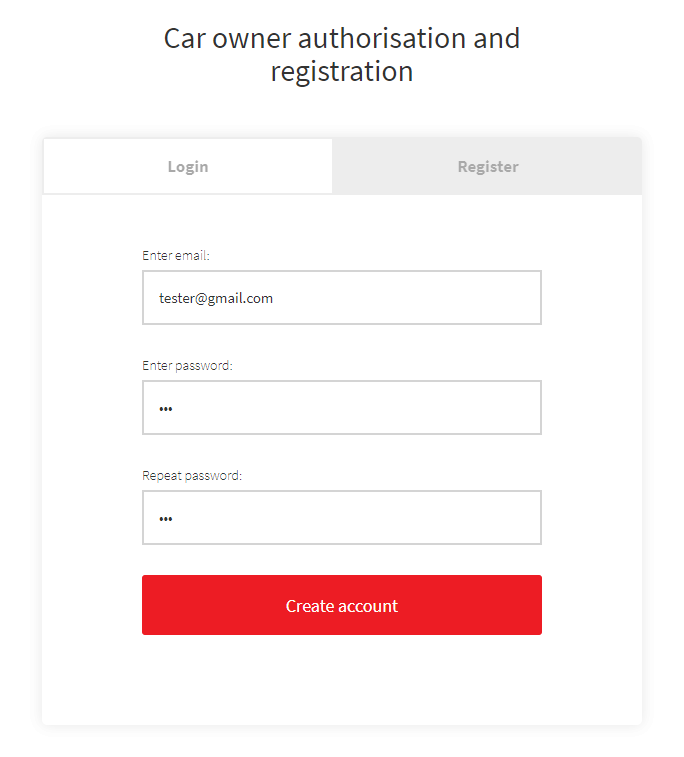
* Logowanie do systemu oraz wylogowanie się z systemu.
* Rejestracja użytkownika w systemie.
* Edytowanie profilu, ustawień.
* Wybór taryfy oraz zapis karty płatniczej.
* Edytwoanie profilu stacji napraw, podanie adresu, wyszukiwanie na mapie, rating.
* Weryfikacja stacji napraw według dokumentów prawnych (prawo do lokalu, zdjęcia paszportu / dowodu osobistego)
* Wyszukiwanie zgłoszeń od właścicieli samochodów.
* Ocena kosztu naprawy części samochodu, według podanych w zgłoszeniu / zamówieniu materiałów.
* Zgłoszenie skargi dotyczącej zamówienia właściciela samochodu.

*Spis użytych technologii:*

* HTML, SASS, JavaScript (ES 2015)
* Meteor.js + React.js / Redux
* MongoDb, MiniMongo, simpl-schema
* Accounts-password, ostrio:files
* ESLint
* AdminLTE

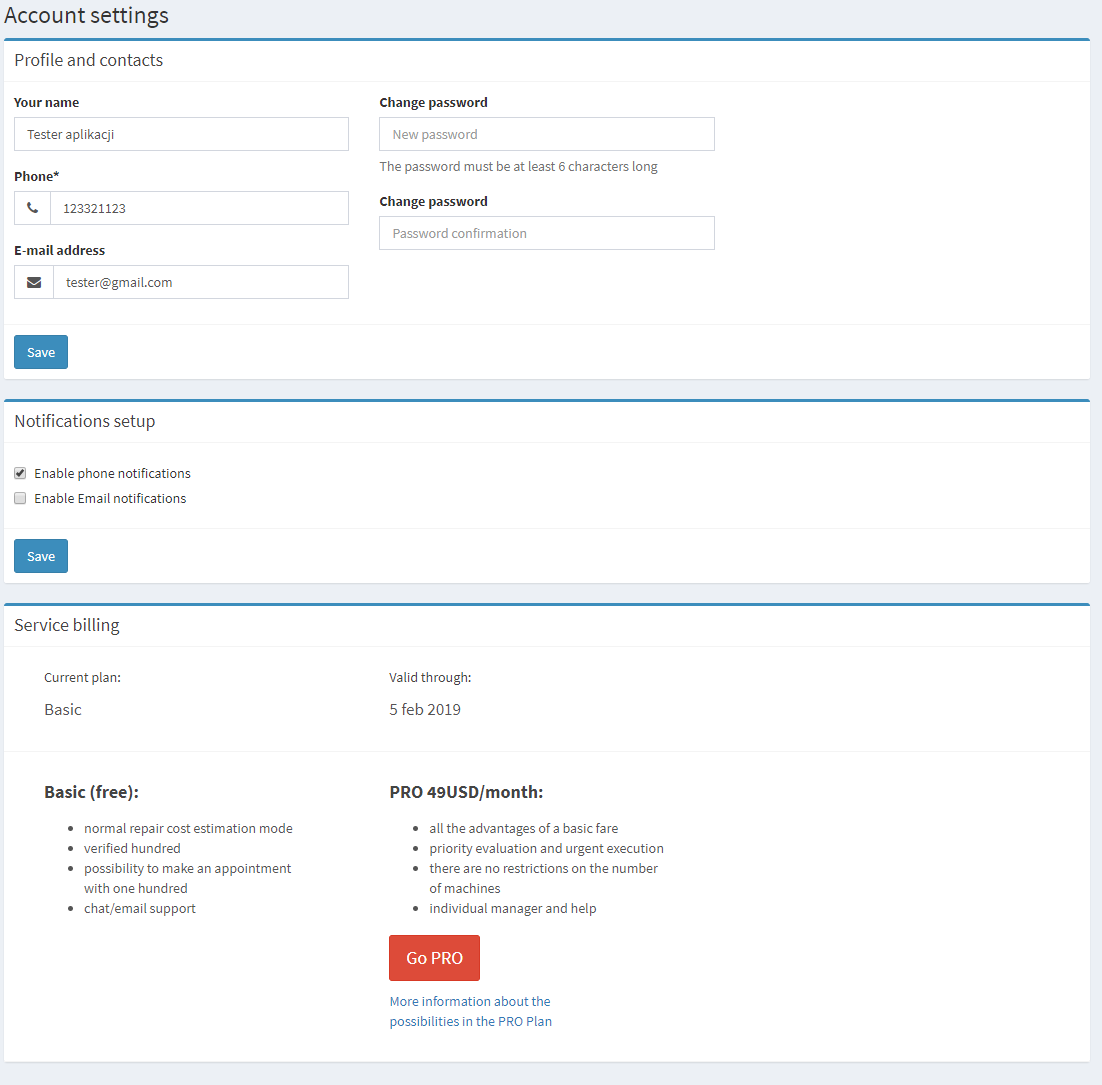
Na poniższych rysunkach prezentują się główne elementy aplikacj.

Na Rysunku 33 widzimy formę do rejestracji oraz logowania nowego użytkownika – właściciela samochodu, dokładnie ten sam element jest używany przy tworzeniu konta właściciela stacji napraw samochodów.



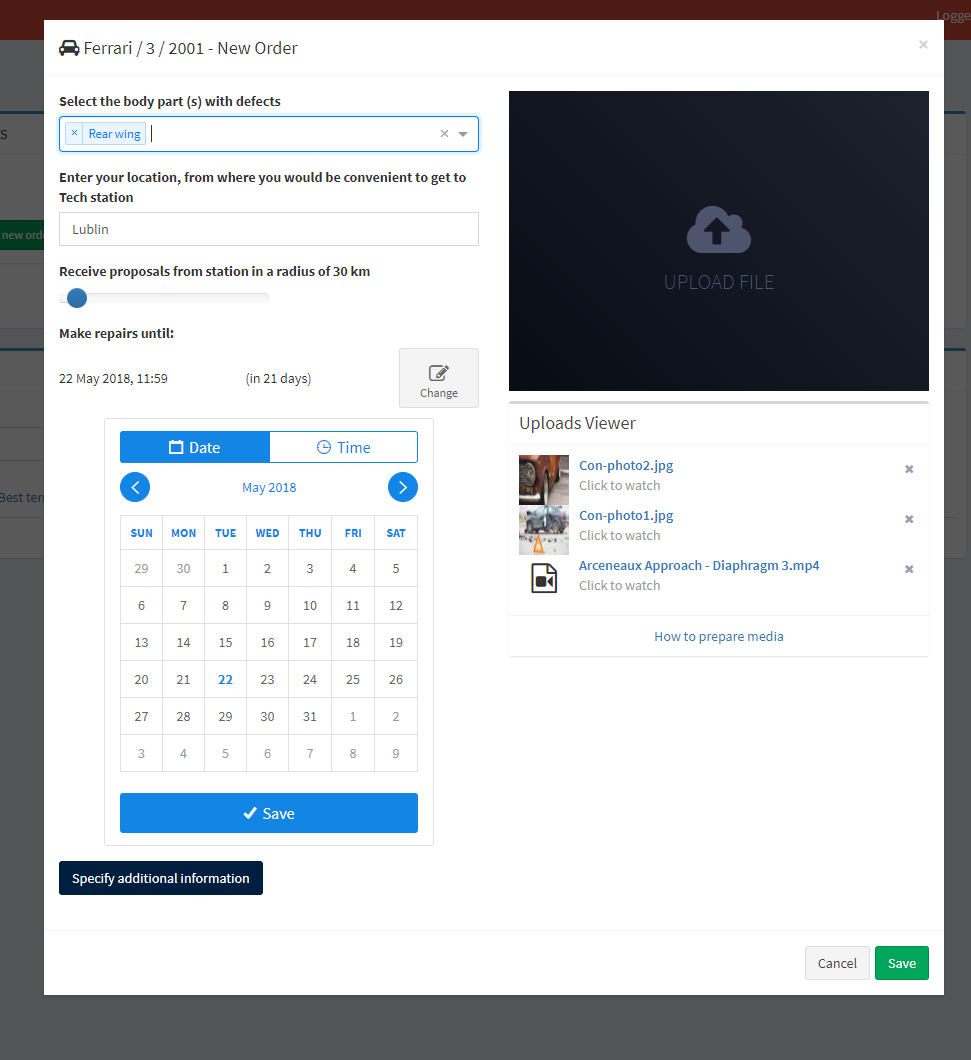
Rysunek 33. Forma rejestracji / logowania nowego użytkownika

Na Rysunku 34 widzimy stronę ustawień profilu właściciela samochodu, gdzie użytkownik może zmodyfikować swoje dane, zapoznać się z opłatami, zmienić hasło, numer telefonu oraz adres skrzyńki pocztowej. Podobną stronę też posiada część aplikacji, obsługująca właściciela stacji napraw samochodów.



Rysunek 34. Profil użytkownika / Billing

Na Rysunku 35 widzimy popup tworzenia nowego zamówienia dla odpowiedniego samochodu. Zamówienia można tworzyć po tym jak zostanie stworzony / podany samochód użytkownika.



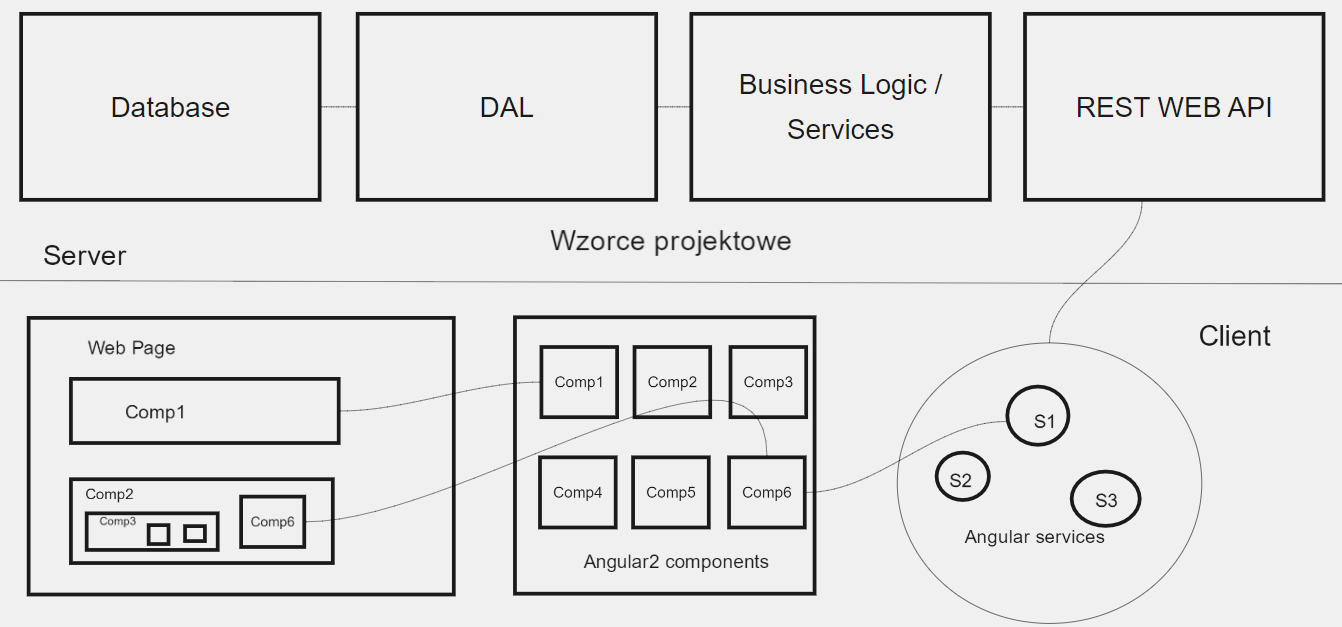
Rysunek 35. Tworzenie zamówienia poprzez właściciela samochodu

Na Rysunku 36 widzimy widok bieżącego miesiąca. Tu na wykresach widzimy wydatki według tygodnia oraz wydatki według tagu. Klikając na odpowiedni tydzień na wykresie, przechodzimy na widok wybranego tygodnia.

## Warstwy aplikacji

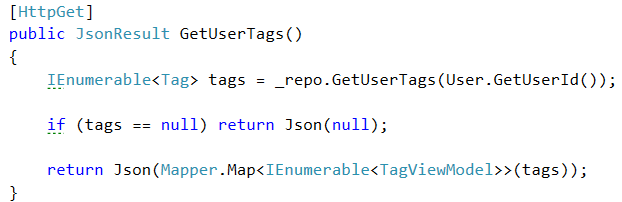
Na Rysunku 38 można zobaczyć warstwy aplikacji Finances. Aplikacja składa się z dwóch części – strony klienta oraz serwerowej.

Po stronie klienta mamy stronę HTML, która składa się z poszczególnych komponentów, do których są podpięte klasy typescriptowe, odpowiadające za logikę danego komponentu. Komponent może zawierać w sobie inne komponenty. Wewnątrz siebie komponent ma możliwość używania serwisów, czyli logiki współdzielonej. W Finances głównym serwisem jest *http service*, celem którego jest wysyłanie danych do serwisów restowych uruchomionych na serwerze.



Rysunek 38. Warstwy aplikacji Finances

Po stronie serwera mamy mechanizm ASP 5 WEB API, który przyjmuje zapytania HTTP ze strony klienta oraz wysyła odpowiednią do zapytania odpowiedź. W najprostszym przypadku (Rysunek 29), po otrzymaniu zapytania mechanizm restowy używa wprost DAL, w sercu którego leży wzorzec repository, który obrabia oraz zwraca dane z bazy danych. Także w serwisie rest odbywa się mapowanie encji modelu bazy danych do view modelu [9].



Rysunek 39. Najprostszy Action w kontrolerze restowym

Obecne są także bardziej złożone ścieżki po stronie serwera, na przykład użycie logiki biznesowej oraz serwisów, które są już odpowiedzialne za komunikację z DAL (Rysunek 40).

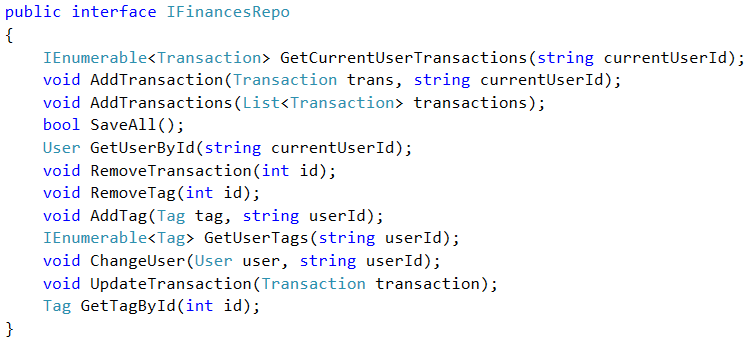


Rysunek 40. Złożony Action w kontrolerze restowym

## Szczegóły implementacji

O ile ASP.NET Core już zawiera wszystkie niezbędne mechanizmy ułatwiające projektowanie aplikacji, dość niełatwym zadaniem okazało się odnalezienie miejsc, gdzie można by było sensownie wstrzyknąć wzorce projektowe. W aplikacji był użyty Facade, żeby przedstawić poręczny interfejs zarządzania bazą danych. W danym przypadku jest to też tak zwany repository pattern, o ile przedstawia sobą jedną klasę do zarządzania danymi.

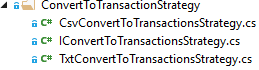
Interfejs klasy wygląda Jak na Rysunku 41:



Rysunek 41. Interfejs IFinancesRepo

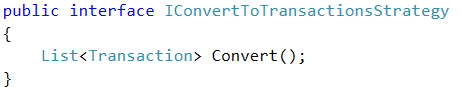
Dany mechanizm wykorzystują rest serwisy, serwisy zwykłe oraz logika biznesowa aplikacji. Także w aplikacji został użyty wzorzec strategy, do zarządzania ładowaniem plików. W taki sposób tworzymy różne strategie dla ładowania plików .csv, .txt oraz innych rozszerzeń.

Do konwertowania używamy oddzielnych klas, które dziedziczą oraz realizują interfejs IConvertToTransactionStrategy (Rysunek 42):



Rysunek 42. Struktura plików według użytego wzorca

Interfejs strategii jest bardzo prosty (Rysunek 43):



Rysunek 43. Interfejs strategii

Także w aplikacji był użyty mechanizm dependency injection, który pozwala nie tworzyć ręcznie egzemplarzy konkretnych klas.

Warto zauważyć, że ciągle podczas tworzenia aplikacji towarzyszyły mi już gotowe, microsoftowe, implementacje wzorców, jako mechanizmy platformy. Przykładem może być standardowy interfejs IEnumerable oraz mechanizm delegatów.

# Zakończenie

Podsumowując swoją pracę, mam nadzieje, że zrealizowałem postawiony na początku cel. W  pierwszym rozdziale zrobiłem krótki opis platformy .Net oraz języka C#, ponadto opisałem podstawy programowania obiektowego oraz SOLID. W końcu pierwszego rozdziału rozpatrzyłem konstrukcje języku UML. Rozdział drugi poświęciłem opisaniu najczęściej używanych w .NET wzorców projektowych. Każdy wzorzec był także opisany w języku C# oraz UML. W ostatnim rozdziale zaprezentowałem swoją aplikację, która używa najbardziej nowoczesnych technologii Google oraz Microsoft na dany moment. Także przedstawiłem wzorce projektowe, których używałem przy tworzeniu aplikacji.

Starałem się, aby z dwudziestu trzech wzorców projektowych opisanych dekady temu wybrać te  najważniejsze, najczęściej używane, które każdy deweloper musi brać pod uwagę podczas pracy nad projektem programistycznym. Jestem bardzo wdzięczny za pomoc doktora Andrzeja Bobyka, który pomógł mi ściśle określić cel pracy. Dzięki tej pracy zdobyłem doświadczenie, którego będę używał na co dzień w pracy nad złożonymi projektami w przyszłości.

# Bibliografia

1. A. Shevchuk, D Okhrimenko, A. Kasyanow - *Design Patterns Via C#*, Ebook, 2015.
2. J. Richter - *CLR Via C# (4ht Edition)*, Microsoft Press, 2012.
3. S. McConnell, *Code complete*, Microsoft Press, 2004.
4. K. Cwalina, B. Abrams, *Framework Design Guidelines*, Microsoft Press, 2005.
5. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns. Elements of Reusable Object-oriented Software*, Addisson-Wesley Proffessional Computing Series, 1995.
6. T.Wright, *Learning JavaScript. A Hands-On Guide to the Fundamentals of Modern JavaScript*, Wydawnictwo Addison-Wesley, 2012.
7. A. Troelsen, *C# 6.0 and the .NET 4.6 Framework*, Apress, 2015.
8. S. Millett, [*Professional ASP.NET Design Patterns*](http://shop.oreilly.com/product/9780470292785.do), O'Reilly, 2010.
9. F.Muhammad, *Real World ASP.NET Best Practices*, Apress, 2003.
10. M.J. Price, [*C# 6 and .NET Core 1.0: Modern Cross-Platform Development*](http://shop.oreilly.com/product/9781785285691.do), Apress, 2016.
11. B. Joshi, *Beginning SOLID Principles and Design Patterns for ASP.NET Developers*, Apress, 2016.
12. R. Ferguson, *Javascript Recipes*, Apress, 2016.
13. A. Prabhu, *Beginning CSS Preprocessors with SASS, Compass and Less*, Apress, 2015.
14. S. Fenton, *Pro TypeScript*, Apress, 2014.
15. *ASP.NET Core Documentation*: https://docs.asp.net/en/latest/
16. D. Esposito, *Modern Web Development: Understanding domains, technologies, and user experience*, Microsoft Press, 2015.