**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 ...

# 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.

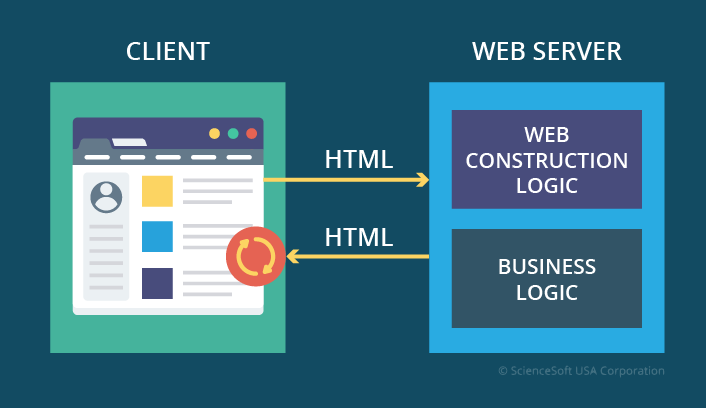
Serwer (ang. *Server*) – tak samo jak wyżej, tym pojęciem można określić jak część implementacji aplikacji tak i środowisko wykonania tego kodu. Implementacja serwerowa może być odpowiedzialna za przygotowanie poszczególnych elementów do renderowania, operacje na bazie danych, implementację logiki biznesowej,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

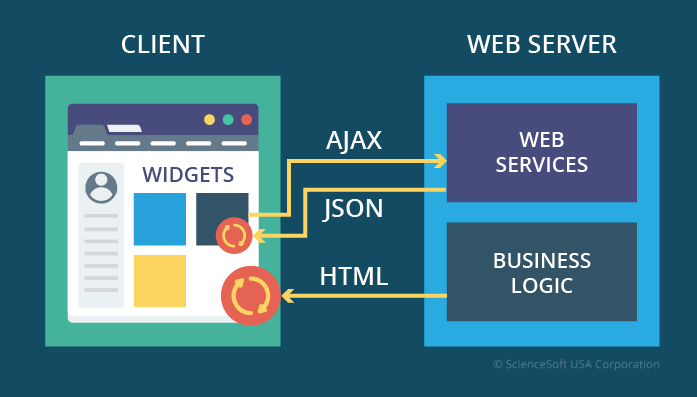
* Legacy web application

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.



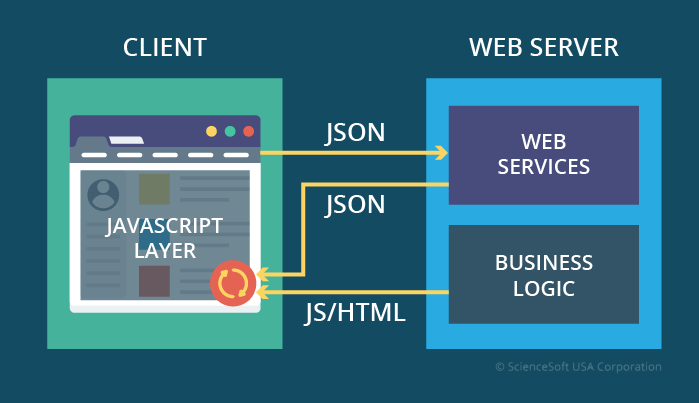
* Widget web application

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’.

## Model abstrakcyjny architektury rozwiązania na platformie Meteor.js

Pod czas projektowania nowego rozwiązania na platformie warto znać jak wygląda połączenie głównych składników Meteor.js. Każda platforma musi mieć możliwość obsługiwać bazy danych. Meteor.js na razie wspiera wyłącznie bazy No-SQL. Zespół w tej chwili pracuje nad sterownikami, które wprowadzą możliwość użycia baz relacyjnych. Warto zauważyć, że takie sterowniki już istnieją. Stworzone oni zostały przez społeczeństwo Meteor.js i są umieszczone na platformie Atmosphere.

Meteor używa podejście *database everywhere*, co oznacza to, że mamy możliwość działania na bazie też po stronie klienta. Synchronizacja danych działa za pomocą protokolu DDP, konsumującego technologię Web Sockets.

Web Sockets – nowoczesna technologia, która określa nowy sposób nieprzerywalnej komunikacji pomiędzy klientem a serwerem. Odpytywanie serwera odbywa się raz, po czym tworzy się interaktywna sesja. Dalej zmiany na serwerze powodują wysyłanie odpowiedzi w postaci zdarzenia.

Serwerowa część aplikacji głównie odpowiada za bezpieczeństwo danych, odczyt oraz ich modyfikację.



Po stronie klienta też zwanej *frontendem* posiadamy zestaw funkcji dla pracy z danymi oraz inne, ułatwiające pracę nad kodem. Dużą zaletą Metoer.js jest swoboda wyboru preprocesorów JavaScript, CSS oraz HTML. Tak zamiast *JavaScript* możemy pisać kod w języku *CoffeeScript*, zamiast *CSS* użyć *SASS*, a zamist *HTML* – *Jade*.

Wystarczy zainstalować odpowiedni pakiet Atmosphere, a kompilacją oraz budowaniem paczki zajmie się sam Meteor.js. Pomimo tego Meteor.js jest dostarczany z transpilerem o nazwie *Babel*, który zezwala używać najnowsze specyfikacje JavaScript w starszych przeglądarkach. To bardzo ułatwia pracę nad projektem, o ile nowsze wersje języka JavaScript wprowadzają nowe konstrukcje syntaksyczne, techniki oraz gotowe funkcje, które zezwalają pisać mniej kodu, a ten co jest napisany staje się bardziej odporny na błędy oraz zrozumiały.

Podczas tworzenia większych aplikacji webowych, używane są tak zwane frameworki frontendowe lub biblioteki, które zajmują się właśnie renderowaniem interfejsu użytkownika. Warto zauważyć, że większość czasu pod czas napisania aplikacji zajmuje obsługa intefejsu użytkownika: tworzenie ładnie wystylowanych komponentów, interakcja z użytkownikiem, przywiązywanie danych do komponentów. Dla tego bardzo ważnym jest dostarczanie najlepszych instrumentów developerskich do pracy z frontendem. Meteor.js zdaje egzamin na 5.

Meteor.js jest dostarczany z systemem do renderowania o nazwie Blaze. Jest to produkt zespołu Meteor.js i jest dostępny do użycia bez konieczności instalacji platformy Meteor.js. Blaze posiada swoje zalety oraz wady, dla tego zespół Meteor.js daje możliwość używać inne popularne rozwiązania do renderowania interfejsów: *React.js, Angular.js, Angular 2/4/5, Vue.js*.

W zakresie tej pracy, będę głównie opierał się na bibliotekę o nazwie *React.js.*

Chciałbym też opisać dla czego wybrałem *React.js* jako system do renderowania. Też chcę porównać między sobą jego alternatywy.

*Blaze* – główną wadą tego rozwiązania jest wydajność. Też wadą jest mniejsze społeczeństwo, które nie produkuje tak dużej ilości gotowych komponentów, jak w przypadku Angular lub React.js. Zaletą jest dobra integracja z platformą Meteor.js. Podejście do renderowania podobne do Angulara.

*Angular.js* – dobrą zaletą jest łatwa integracja z projektem, ilość już gotowych funkcji oraz pakietów, duże społeczeństwo, dostawcą jest Google. Wadą jest wydajność oraz rozmiar. Pomimo tego podejście do renderowania róźni się z React.js. Angular.js używa dodatkowe attrybuty wewnątrz HTML, gdzie umieszczane są powiązania oraz kod imperatywny. Można powiedzieć, że developer pisze JavaScript wewnątrz HTML.

*Angular 2 / 4 / 5* – jest pozbawiony wady związanej z wydajnością, ale integracja jest dużo trudniejsza, preferowane jest użycie języka transpilowanego TypeScript. Też wadą jest rozmiar oraz złożoność technologii – jest to cały framework, gdy React – to mała biblioteka.

*React.js* – biblioteka napisana oraz wspierana przez zespół Facebook, co jest ogromną zaletą. Też zaletą jest rozmiar oraz prostoć użycia. Jest to najbardziej wydajne rozwiązanie na tej liście, aczkolwiek nowsze wersje Angular oraz Vue.js też są wystarczająco wydajne. Podejściem do renderowania jest tworzenie HTML w JavaScript. Więcej informacji o tej bibliotece, będzie umieszczone w rozdziale ‘*Frontend’*.

*Vue.js* – framework, stworzony przez jedną osobę a później wspierana przez społeczeństwo. Wbiera najlepsze cechy Angular.js, Angular oraz React.js. Główną wadą jest to, że framework nie może pochwalić się wsparciem takich gigantów jak Google lub Facebook oraz jest relatywnie nowym na rynku.

## Meteor.js CLI

Meteor CLI jest głównym interfejsem do pracy z platformą. Przy pomocy Meteor CLI można tworzyć nowe projekty Meteor.js używając odpowiedniego szablonu, podbijać wersje zależności, dodawać oraz usuwać pakiety Atmosphere, pracować z bazą danych Mongo, uruchamiać aplikację w trybie debugowania, resetować stan projektu oraz dużo więcej. W tej sekcji chciałbym omówić polecenia które używam najczęściej.

**meteor help *command***

Wyświetla dokumentację dotyczącą określonej komendy. Jeżeli nie podać komendę, wyświetli się liczba najbardziej używanych komend oraz ich krótki opis.

**meteor run**

Uruchamia serwer Node.js dla bieżącego projektu. Każdy projekt posiada specjalny folder o nazwie *.meteor* z plikami konfiguracyjnymi. Aplikacja jest udostępniona pod adresem localhost:3000. Zaimplementowana jest funkcja ‘hot replace’ pod czas której zmiany w plikach projektu awtomatycznie stosują się na uruchomionym serwerze. Port na którym działa aplikacja można zmienić za pomocą flagi *–port.* Wtedy polecenie uruchomienia aplikacji, działającej na odpowiednim porcie wygląda następująco: *meteor run –port 8080.* Żeby przekazać dodatkowe opcje serwerowi Node.js, można użyć zmiennej środowiskowej ‘*NODE\_OPTIONS*’.

W nowszych wersjach Meteor.js debugowanie kodu serwerowego umożliwia dodatkowa flaga *--inspect-brk*, która zatrzymuje proces serwerowy po załadowaniu kodu ale przed jego wykonaniem i umożliwia wpięcie breakpointów.

**meteor create *nazwa-projektu***

To polecenie tworzy nowy projekt szablonowy Meteor.js w nowym folderze o podanej nazwie. Jest też możliwość wyboru standardowego szablonu aplikacji. Za to odpowiadają flagi: *--bare* oraz *--full*. Trzeci szablon jest domyślny. Warto zauważyć że każdy z szablonów posiada swoją listę zainstalowanych pakietów standardowych. Za pomocą flagi *--package*, można stworzyć pakiet, który można będzie użyć w bieżącej aplikacji lub umieścić na platformie Atmosphere.

**meteor update**

Uruchomienie tego polecenia powoduje podbicie Meteor.js oraz pakietów Atmosphere do ostatniej wersji. Update jest zrobiony w taki sposób, że bierze pod uwagę kompatybilność poszczególnych pakietów i nie podbija automatycznie pakiety do wersji, która może złamać aplikację. Jeżeli nie potrzebujemy podbijać release wersję Meteor.js i chcemy podbić tylko wersje zainstalowanych pakietów, można użyć flagi *--packages-only*, lub podać pełne nazwy pakietów.

**meteor add *nazwa-pakietu***

Dodaje nowy pakiet Atmosphere do projektu. Też zezwala dodać ograniczenia wersji doklejając odpowiednie flagi tuż po nazwie pakietu.

**meteor remove *nazwa-pakietu***

Usuwa packiet z projektu.

**meteor list**

Wyświetla listę pakietów dodanych do projektu oraz ich wersje. Także zawiera opisy pakietów i dostępność nowych wersji.

**meteor mongo**

Uruchamia wiersz poleceń do pracy z developerską bazą danych. Działa po uruchomieniu aplikacji poleceniem *meteor run*.

**meteor reset**

Wyczyszcza wszystkie dane dotyczące projektu, usuwa bazę danych Mongo. Polecenie bardzo poręczne pod czas zmian struktury bazy danych.

**meteor npm** oraz **meteor node**

Te polecenia są poręczne kiedy musimy użyć npm lub node, które były dostarczane razem z Meteor.js. O ile na maszyne można zainstalować oddzielny node oraz npm, warto uważać na to, żeby instalować pakiety używając dostarczanych rozwiązań o ile wersje narzędź mogą się róźnić co może spowodować dziwne zachowanie aplikacji po wdrożeniu na produkcję.

Pomimo wymienionych poleceń Meteor CLI dostarcza wiele innych, które są poręczne w bardziej specyficznych sytuacjach: wsparcie wielu platform, logowanie się na konto developerskie Meteor, wdrażanie projektu na platformę Galaxy, rozmieszczenie pakietów na Atmosphere, budowanie produkcyjnej wersji aplikacji, wyszukiwanie pakietów, uruchomienie testów jednostkowych.

Praca z Meteor CLI jest ważną częścią procesu developmentu aplikacji. Narzędzie jest bardzo wygodne oraz minimalistyczne.

# Rozdział 2. Frontend



## Pojęcie wzorca projektowego

W

# Rozdział 3. Backend

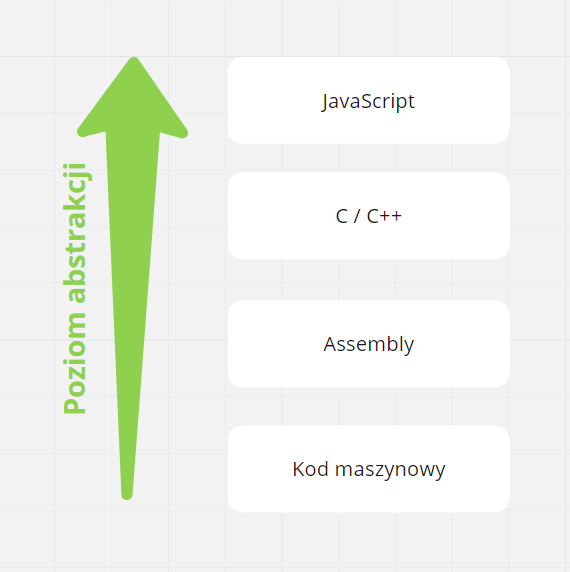
## Wstęp ogólny do platformy Node.js

Aby sformować model mentalny platformy Node.js, należy najpierw zrozumieć jej jądro – silnik JavaScript V8. Znajomość tego, jak działa V8 jest ważnym aspektem pod czas pracy na platformie Node.js.

Zanim można będzie zanurzyć się w platformę Node.js oraz silnik V8, należy zrobić kilka kroków wstecz – do kodu maszynowego.

Kod maszynowy – to zestaw instrukcji, które są natywne dla używanej michroarchitektury. Kiedy piszemy aplikacje w języku nie natywnym, zawsze jest proces, który konwertuje nasz kod w język natywny dla jednej lub więcej platform mikroprocesorów. Dzisiaj nie piszemy oprogramowanie wprost używając kod maszynowy, o ile przywiązujemy się do jednej platformy oraz działamy na zbyt nizkim poziomie abstrakcyjnym.

Rozwój oprogramowania przez dłuższy czas określał się wzrostem poziomu abstrakcji języków oprogramowania. Im większy był ten pozióm tym mniej język przypominał kod maszynowy i tym mniej nizkopoziomowych operacji zezwalał wykonywać.



Otóż pierwszą ważną rzeczą, którą należy zrozumieć o Node.js jest to, że Node.js jest napisany w C++. To zdanie może spowodować nieporozumienie, o ile nazwa Node.js mentalnie wiąże się z językiem JavaScript, ale nie C++. Faktycznie, pisząc aplikacje na platformie Node.js, posługujemy się językiem JavaScript, ale w tym samym czasie Node.js jest aplikacją C++. Powodem tego jest to, że V8, silnik JavaScript, jest napisany w języku C++.

## Silnik JavaScript oraz specyfikacja ECMASCRIPT

Ostatnim pojęciem, na które zwrócimy uwagę zanim zanurzymy się w szczegóły silnika V8 jest specyfikacja ECMASCRIPT.

Najpierw był stworzony JavaScript, ale później dostawcy przeglądarek internetowych (Microsoft, Netscape) zaczeli tworzyć swoje implementacje silników JavaScript, wprowadzając jakieś zmiany. Ecma International – to organizacja zajmująca się standardyzacją, celem której jest określić jak dokładnie ma działać bazowa wersja języka.

Standardyzacja jest bardzo potrzebna, o ile jest wiele silników JavaScript, które musimy umieć obsługiwać. Jednym z nich jest V8. Na dany moment V8 realizuje standard ECMASCRIPT 2015 (alternatywnie ES6). Najnowszym standardem dziś jest ECMASCRIPT 2017. Nowszy standard zwykle określa nowe konstrukcje językowe, które ułatwiają proces napisania oprogramowania. Podsumowując, gdybym miał napisać swój silnik JavaScript, musiałbym orientować się na zachowanis, określone w odpowiedniej wersji documentu ECMASCRIPT.

Silnik JavaScript – to program, który konwertuje kod JavaScript w kod maszynowy oraz realizuje specyfikację ECMASCRIPT.

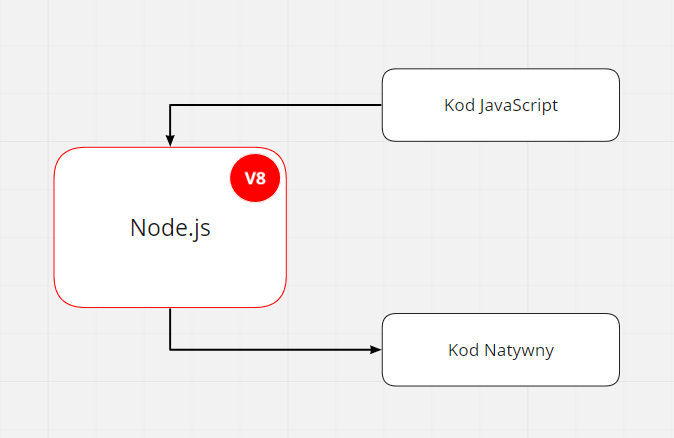
ECMASCRIPT – to standard na którym jest zbudowany JavaScript.

## Node.js oraz V8

Głównym celem silnika V8 było użycie go wewnątrz przeglądarki Google Chrome. Jest to produkt outsorsowy, napisany oraz wspierany przez Google. V8 był zbudowany w sposób, który nie ogranicza jego użycie w innych środowiskach. V8 zawiera zestawy kodu dla każdej wspieranej architektury mikroprocesorów, które zajmują się tłumaczeniem języka JavaScript na kod natywny odpowiednej architektury. V8 jest ogromny, budowany przez lata i bardzo wydajny. Ważnym aspektem jest też to, że on jest darmowy i może być wstrzyknięty w dowolną aplikację C++.

Można napisać nową aplikację C++, która będzie mogła pracować z instrukcjami JavaScript, przepuszczając ich przez V8. Ale to nie wszystko. V8 posiada ‘wpięcia’ które dają możliwość rozszerzyć bazowe właściwości JavaScript. Jest możliwość określić dodatkowe instrukcje JavaScript, które będą powodować uruchomienie odpowiedniego kodu nowej aplikacji . W takie sposób nowa aplikacja rozumie więcej niż to co jest w standardowej specyfikacji ECMASCRIPT.

Jest to bardzo ważne, o ile C++ ma dużo więcej możliwości niż JavaScript, który był zaprojektowany jako język obsługujący przeglądarkę. JavaScript nie był zaprojektowany do pracy z nizkopoziomowymi operacjami, takich jak operacje na fizycznych plikach albo łączenie z bazą danych. Teraz możemy ‘udostępnić’ umiejętności C++ dla kodu, który piszemy w JavaScript, przepuszczając go przez nową aplikację C++. Taką aplikacją jest Node.js.



Node.js – to aplikacja C++, która integruję silnik JavaScript V8 i rozszerza możliwości JavaScript do tego stopnia, że możemy używać funkcje dostępne w C++.

Pomimo części C++, tak samo jak i wiele innych platform, Node.js posiada bibliotekę własną – zestaw kodu Javascript, który ułatwia wykorzystanie funkcji nizkopoziomowych.

## Jakie problemy rozwiązuje Node.js

Tworzenie Node.js było próbą odpowiedzi na jedno ważne pytanie: czego potrzebuje JavaScript, żeby obsługiwać część serwerową? Jakie dodatkowe funkcjonalności były zaimplementowane w Node.js, żeby móc wykonywać zadania, jakie wykonuje web serwer?

Zestaw wymagań umieściłem na ponizszą listę:

1. Sposób organizacji kodu źródłowego oraz plików.
2. Obsługa systemu plików na serwerze.
3. Obsługa baz danych.
4. Umiejętność komunikować się w Internecie.
5. Możliwość przyjmować żądania i wysyłać odpowiedź w standardowym formacie.
6. Obsługa procesów o długim czasie wykonywania.

**Sposób organizacji kodu źródłowego oraz plików:**

Jest realizowany za pomocą systemu modułów. Nowa wersja JavaScript zawiera system importów oraz exportów, którego używam zamiast systemu modułów Node.js. W obu przypadkach możemy zaimportować cały plik, lub obiekty/funkcje/klasy, które zostały wcześniej wyeksportowane.

## Baza danych MongoDB

Meteor.js używa MongoDB do przechowywania danych na serwerze. MongoDB jest bazą danych NoSQL.

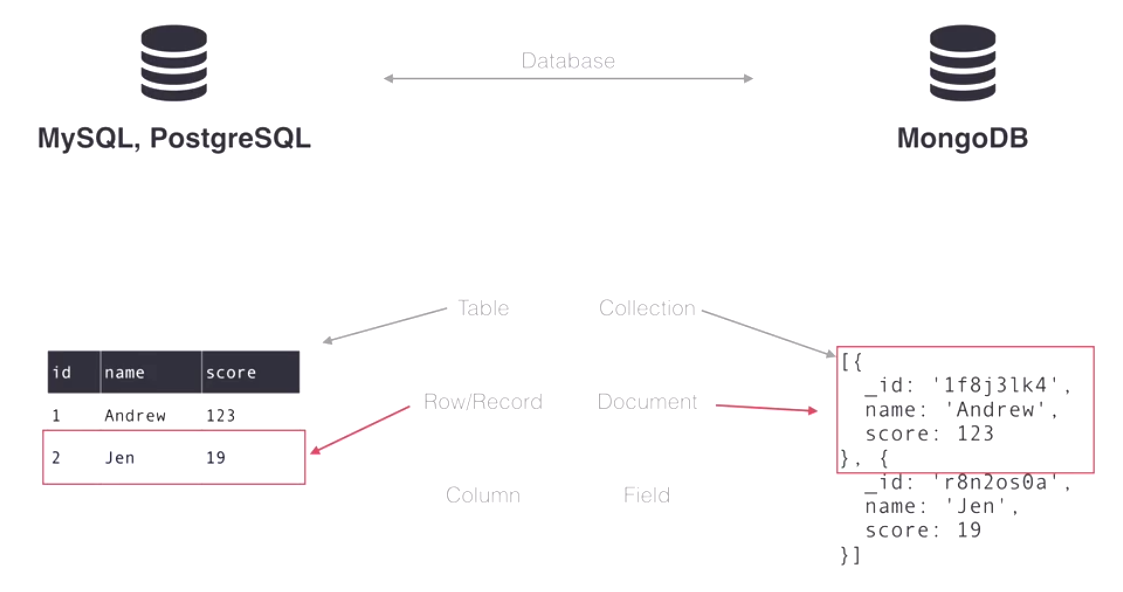
NoSQL – to rodzina baz danych, jakie są alternatywą relacyjnym bazam SQL. Do zbioru baz danych NoSQL należy rodzina dokumentncyh baz danych. MongoDB jest implementacją takiej bazy danych.

Wcześniej powtórzenie danych było wielkim problemem, o ile oznaczało to większe zużycie miejsca na dysku. Relacyjne bazy danych były zaprojektowane, żeby radzić sobie z tym problemem.

Duża ilość czasu minęła z tego czasu kiedy zostały zaprezentowane bazy danych SQL. Dzisiaj ilość dostępnego miejsca na dysku jest znacznie większa, a koszt jednostki pamięci jest mniejszy. Dzisiaj dużo większym problemem jest to, jak często musimy modyfikować strukturę danych. W dzisiejszym świecie, potrzebujemy mieć możliwość sprawnie dodawać i modyfikować funkcjonalność oprogramowania. W bazach danych NoSQL, każdy dokument zawiera zarówno dane oraz strukturę, co zezwala dokonywać pojedynczych zmian tam gdzie jest to potrzebne. W taki sposób tracimy na rozmiarze ale zyskujemy na prostoci, elastyczności przechowywania oraz obsługi danych.

Baza danych MongoDB składa się z jednej lub więcjej kolekcji.

Kolekcja (ang. *Collection*) – ekwiwalent do tabel w relacyjnych bazach danych. Kolekcje zawierają *dokumenty*, gdzie każdy dokument musi posiadać unikalny identyfikator. Dokumenty mają postać obiektów JSON, składają się z pól (ang. *Field*) i mogą posiadać wiele wymiarów.



MongoDB posiada swój język zapytań, bazujący się na obiektach JSON. Taka składnia jest bardzo potężna, szczególnie, kiedy musimy odpytywać instancję Mongo po stronie klienta.

Zobaczmy jak łatwo można tworzyć nowe kolekcje przy pomocy MongoDB API:

const Test = new Mongo.Collection(‘test’);

Ten krótki kawał kodu tworzy nową kolekcję MongoDB lub MiniMongo, w zależności od tego czy jest on odpalony po stronie klienta lub serwera. Pod koniec zwraca referencję na obiekt, umożliwiający pracę z tą kolekcją.

Ten obiekt-API posiada następujące podstawowe metody:

* *insert*: Dodawanie nowych dokumentów do bazy danych (kolekcji)
* *update*: Modyfikacja dokumentów lub ich części
* *upsert*: Dodawanie lub modyfikacja dokumentów
* *remove*: Usunięcie dokumentu z kolekcji
* *find*: Zwraca rezultat wyszukiwania dokumentów w bazie
* *findOne*: Zwraca pierwszy dokument, jako rezultat wyszukiwania

## Instrumenty zarządzania MongoDB

Pod czas uruchomienia aplikacji, Meteor.js automatycznie uruchamia i konfiguruje dla nas serwer bazy danych. Także Meteor zawiera zestaw podstawowych narzędzi do pracy z Mongo.

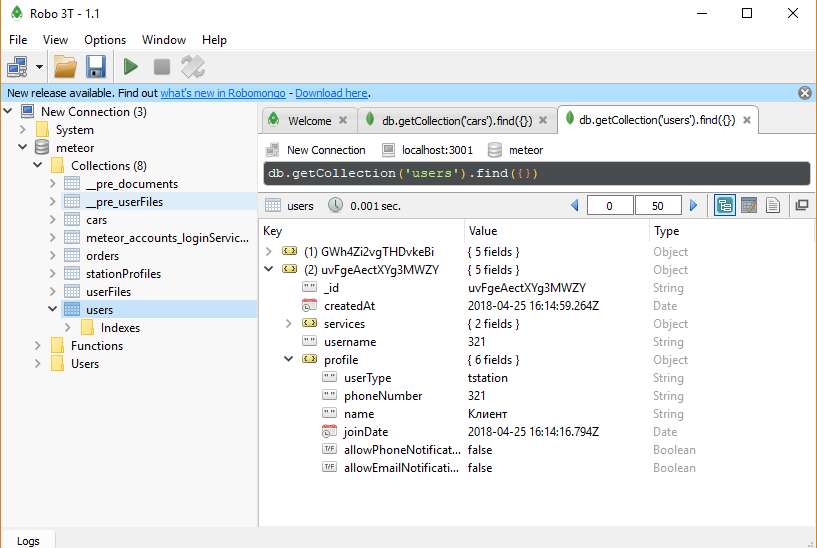
Jednym z tych narzędzi jest mongo shell, który jest częścią Meteor CLI. Za pomocą tego w konsoli można odpytywać serwer bazy danych oraz modyfikować dane. Shell można odpalić poleceniem *meteor mongo*.

Często potrzebne jest wyczyszczenie używanej bazy danych. Opłaca się to zrobić, kiedy zmieniona zostaje struktura używanej kolekcji. Zresetować bazę można jednym poleceniem - *meteor reset*.

Warto zauważyć, że nie zawsze jest poręczne użycie konsoli do pracy z bazą danych. Często potrzebny jest bardziej reprezentatywny podgląd na dane. Idealnym narzędziem do pracy z MongoDB jest Robo 3T.

Robo 3T – to GUI narzędzie dla developerów baz danych MongoDB. To narzędzie jest darmowe i posiada otwarty kod źródłowy.

Niżej można zobaczyć okno narzędzia, gdzie jest zaprezentowana część jego możliwości.



## Meteor.js MongoDB API CRUD

CRUD (ang. *Create, read, update, delete*) – to jest skrót, który odnosi się do czterech podstawowych działań na baize danych: dodanie nowych danych, odczyt danych, modyfikacja danych oraz ich usunięcie.

Żeby wykonywać CRUD opercje na odpowiedniej kolekcji, musimy posiadać referencje na wcześniej utworzony egzemplarz klasy Mongo.Collection, która była zapisywana pod czas tworzenia kolekcji pos stronie klienta (MiniMongo) oraz serwera (MongoDB). Nie wszystkie operacje można bezpiecznie wykonać po stronie klienta. Gdyby tak było, użytkownik za pomocą przeglądarki mógłby usunąć bazę danych na serwerze. Kwestie bezpieczeństwa będą omówione później.

**Odczyt danych:**

Mongo.Collection.find([selector], [options])

Mongo.Collection.findOne([selector], [options])

Te funkcje zezwalają odczytać dokumenty lub dokument kolekcji według podanego selektora. Selektor jest obiektem JavaScript i konfiguruje się według standardów MongoDB. Dodatkowe opcje zezwalają sortować rezultat, pomijać pewną ilość dokumentów, ograniczyć ilość dokumentów, podać listę pól które chcemy posiadać w dokumentach i t.d.

Metoda *find* zwraca kursor. To znaczy że dokumenty nie są zwracane odrazu. Kursor posiada funkcje ***fetch***, która zwraca wszystkie pasujące dokumenty, ***map*** oraz ***foreach*** dla iteracji po dokumentach oraz ***observe***, służący do rejestracji funkcji zwrotnej, uruchamianej wtedy kiedy zostają zmienione pasujące dokumenty.

Obie funkcje mogą być wykorzystywane jak na serwerze tak i na kliencie.

**Dodanie nowych danych:**

Mongo.Collection.insert(document, [callback])

Ta funkcja dodaje dokument do kolekcji i zwraca jego *\_id*. Dokumentem jest zwykły obiekt. Pola mogą zawierać dowolne kombinacje EJSON-kompatybilnych typów danych. *Callback* jest funkcją zwrotną, która jest wykonywana po dodaniu dokumentu do kolekcji lub wywoływana pod czas błędu z argumentem *error*.

Jeżeli nie podać funkcję zwrotną po stronie serwera, działanie wykona się synchronicznie, co wpłynie negatywnie na wydajność aplikacji. Po stronie klienta funkcja zawsze wykonuje się asynchronicznie.

Jeżeli obiekt nie będzie posiadał pola *\_id*, Meteor wygeneruje go automatycznie. Funkcja działa zarówno na kliencie, jak i na serwerze.

**Modyfikacja danych:**

Mongo.Collection.update(selector, modifier, [options], [callback])

Ta funkcja modyfikuje jeden lub więcej dokumentów kolekcji i zwraca liczbę zmodyfikowanych dokumentów. W selektorze filtrujemy, jakie dokumenty chcemy modyfikować. Modyfikator jest najważniejszą częścią polecenia. On określa to jak chcemy zmienić dokumenty dopasowane do selektora.

Przykłady modyfikatora:

{$set: { name: ‘Bob’ }}

{$inc: { age: 2 }}

Dwa poprzednie modyfikatory można połaczyć z sobą:

{$set: { name: ‘Bob’ }, $inc: { age: 2 }}

Jeżeli modyfikator nie zawiera $-operatora, on jest traktowany jako obiekt. Jeżeli modyfikatorem jest obiekt, wszystkie dokumenty dopasowane do selektora będą zastępione tym obiektem.

Funkcja posiada dwie dodatkowe opcje: *multi* – która kontroluje czy ma być zmieniony tylko jeden dokument oraz *upsert*, która z wartością true tworzy nowy dokument, jeżeli nie udało się znaleźć żadnego dopasowania. Funkcja działa zarówno na kliencie, jak i na serwerze.

**Usunięcie danych:**

Mongo.Collection.remove(selector, [callback])

Ta funkcja usuwa z kolekcji wszystkie dokumenty dopasowane do selektora. Jeżeli selektorem jest pusty obiekt, usuną się wszystkie dokumenty kolekcji. Po stronie klienta posiada wiele ograniczeń, związanych z bezpieczeństwem.

## Publish & Subscribe

W tej sekcji będzie omówiony sposób kontroli sycnhronizacji bazy danych MongoDB na serwerze z bazą danych MiniMongo po stronie klienta. Domyślnie przy tworzeniu aplikacji o standardowym szablonie, do projektu jest dodany pakiet o nazwie autopublish, który automatycznie udostępnia dane klientowi. Tak, kiedy dane są udostępnione, Meteor za pomocą protokołu DDP będzie na bieżąco synchronizował dane bazy klientskiej z serwerową. To umożliwia nam korzystanie z danych za pomocą referencji na egzemplarz kolekcji MiniMongo, która z kolei zawiera funkcje find, update, remove i t.d. Warto zauważyć, że praca z MiniMongo odbywa się z dokładnie takiej samej składni jak i na serwerze.

Podejście z użyciem pakietu autopublish ma swoje zalety oraz wady. Posiadając dokładnie te same dane na kliencie nie musimy w ogóle zastanawiać się nad warstwą danych. Jest to bardzo pomocne, kiedy tworzymy prototyp aplikacji. Ale za tym idą też wady: bardziej wydajną będzie synchronizacja tylko potrzebnych danych. Pomimo tego często musimy ograniczyć, do jakich danych ma dostęp użytkownik.

Meteor posiada mechanizm publikacji oraz subskrybcji, który zezwala kontrolować jakie dane będą zsynchronizowane i dostępne po stronie klienta.

**Publikowanie:**

Meteor.publish(name, func)

Implementacje mechanizmu należy zacząć ze strony udostępniania danych. Żeby opublikować podzbiór danych, po stronie serwera należy wykonać funkcję *Meteor.publish().* Funkcja nie jest dostępna po stronie klienta. Pod czas tworzenia publikacji musimy podać jej nazwę, która będzie służyła identyfikatorem pod czas subskrypbcji. Drugim parametrem jest funkcja, wywoływana za każdym razem, kiedy odbędzie się subskrybcja po stronie klienta. Funkcja publish zwraca kursor lub listę kursorów odpowiednich kolekcji.

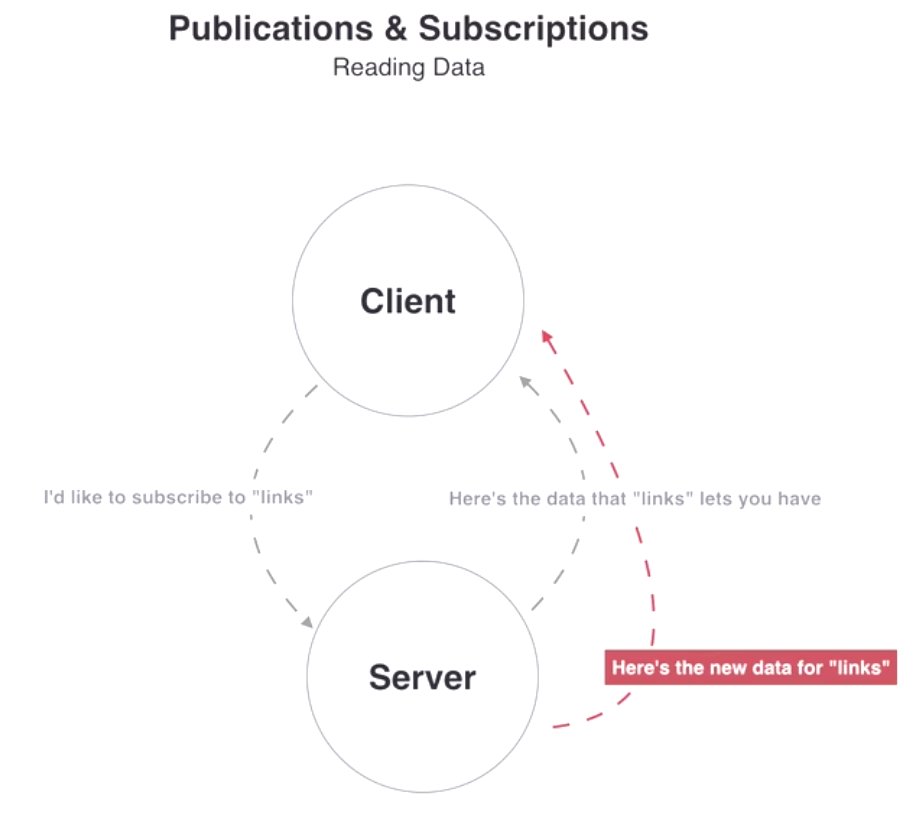
Jeżeli ten sam dokument był opublikowany kilka razy, wersje dokumentów będą złączone.

**Subskrybcja:**

Meteor.subscribe(name, [arg1, arg2…], [callbacks])

Funkcja jest dostępna wyłącznie po stronie klienta. Pod czas subskrybcji, przekazujemy nazwę-identyfikator publikacji, opcjonalne argumenty, oraz dwie funkcje zwrotne wywoływane pod czas wydarzeń: *onStop* oraz *onReady*. Wydarzenie onStop będzie wywołane, jeśli proces na serwerze będzie zakończony lub jeżeli wystąpi błąd. Wydarzenie *onReady* będzie wywołane, jeśli w funkcji publikacji zostanie wywołana funkcja *this.ready().*

Funkcja zwraca obiekt zawierający funkcje *stop()* oraz *ready()*. Funkcja *stop()* zezwala zrezygnować z subskrybcji, gdy *ready()* zezwala sprawdzić czy synchronizacja z serwerem jest ukończona. Pod czas subskrybcji serwer zaczyna proces synchronizacji udostępnionych danych z klientem.



## Meteor Methods

Meteor.methods(methods)

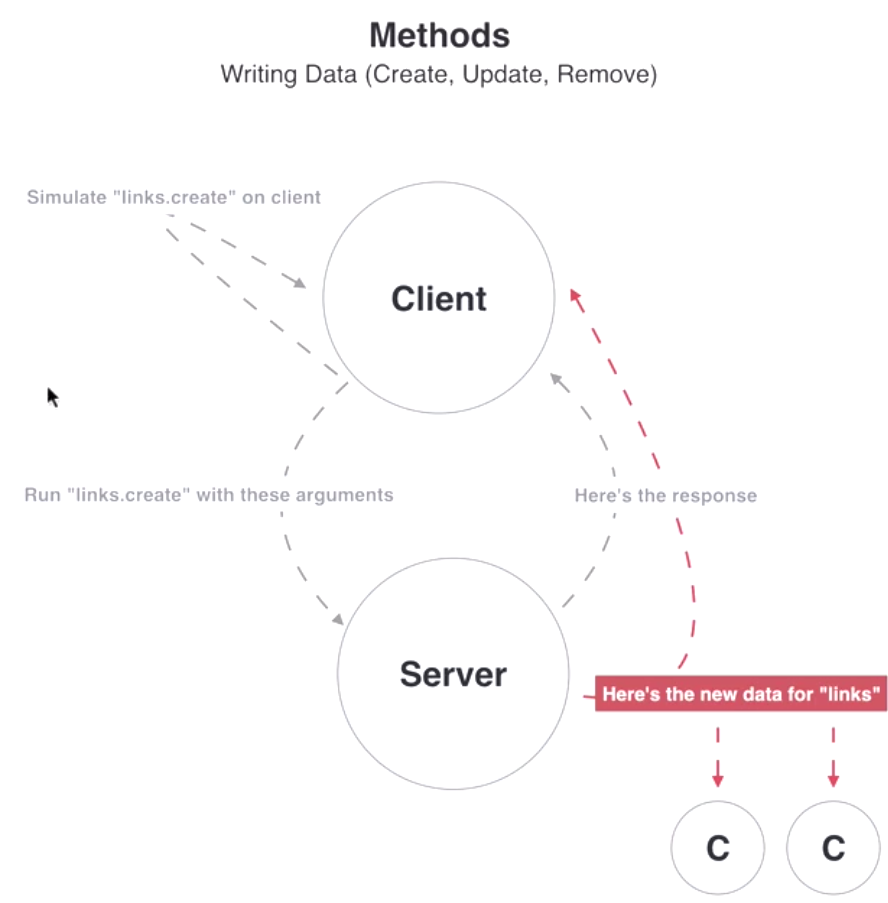
Metody – to są funkcje serwerowe, które można wywołać po stronie klienta. Metody – to sposób na to, żeby bezpiecznie modyfikować dane. O ile metoda jest odpalana na serwerze, jest możliwość zweryfikowania tego, czy działanie wykonuje zalogowany użytkownik oraz czy posiada on uprawnienia do wykonania odpowiedniej akcji. Argumentem jest obiekt, gdzie klucz każdej nowej propercji – to nazwa metody, a wartością jest funkcja. Takie funkcje mogą przyjmować dowolne parametry. W kontekcie każdej funkcji ustawia się słowo kluczowe *‘this’*, które posiada flagi oraz funkcje do zweryfikowania \_id użytkownika, odblokowania kolejki dla uruchomienia kolejnych metod i t.d.

Meteor.call(name, [arg1, arg2…], [asyncCallback])

Pod czas wywoływania metod po stronie klienta zdarzają się dwie rzeczy: żądanie wysyła się na serwer oraz wykonuje się na lokalnym schowku MiniMongo. Takie podejście zezwala momentalnie optymistycznie wyrenderować interfejs użytkownika.

W argumentach podajemy nazwę metody oraz jej parametry. Jeżeli podamy asyncCallback, wywołanie metody odbędzie się asynchronicznie i po zakończeniu wywoła się funkcja zwrotna. Jeżeli nie podać ostatni argument, wywołanie odbędzie się w trybie synchronicznym.

Po modyfikacji danej, ta modyfikacja jest dostępna dla wszystkich subskrypcji, korzystających z tej danej.



## Kwestie bezpieczeństwa

O ile aplikacje na platformie Meteor są pisane w stylu łączącym klient oraz serwer, ważna jest świadomość tego, gdzie będzie uruchamiany kod.

Ważnym jest owinięcie wrażliwych kawałków w metody. Tak każde żądanie, które przychodzi ze świata zewnętrznego, jest weryfikowane po stronie serwera. Jeżeli zaniedbać ten mechanizm, użytkownik będzie mógł wprowadzać niechciane zmiany do bazy danych.

Trzeba pamiętać o tym, żeby się pozbyć pakietów szablonowych: *insecure* oraz *autopublish*.

*Insecure* – to pakiet dostarczany z bazowym szablonem aplikacji, który zezwala modyfikować bazę danych po stronie klienta.

*Autopublish* – to pakiet dostarczany za bazowym szablonem aplikacji, który automatycznie synchronizuje wszystkie dane bazy serwerowej z klientem.

Warto pamiętać o tym, żeby sprawdzać wszystkie argumenty, które przychodzą do metod ze strony klienta.

Nie można wysyłać \_id użytkownika ze strony klienta. Ta wartość jest automatycznie dostępna po stronie serwera i jest zarządzana systemem loginów DDP.

O ile metody są dostępne wszędzie, bardzo łatwo można ubić serwer skryptem złośliwym, wywołującym te metody w pętli. Meteor ma wbudowane ograniczenia ilości wołań przez jednostkę czasu. Także możemy modyfikować te wartości ręcznie.

Dobrym sposobem jest też kontrola pól kolekcji, która jest zwracana na poziomie publikacji.

Klucze do baz danych oraz klucze do zewnętrznych API muszą być przechowywane w plikach konfiguracyjnych lub za pomocą zmiennych środowiskowych. Te podejścia można kombinować. W taki sposób w środowisku developerskim można używać pliku konfiguracyjnego, a po wdrożeniu na serwer produkcyjny skonfigurować zmienne środowiskowe w ustawieniach serwera.

Każda aplikacja produkcyjna, która zarządza danymi użytkownika musi być zabezpieczona za pomocą SSL. Meteor udostępnia pakiet o nazwie *force-ssl*, który będzie wymuszał połączenie SSL na środowisku produkcyjnym.

## Poręczne funkcje standardowe Meteor.js

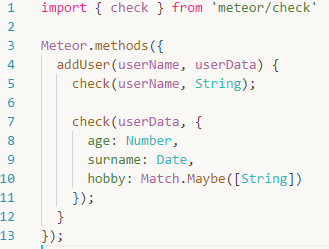
Celem Meteor jest ułatwienie napisania aplikacji. Platforma zawiera dużą ilość drobnych dodatków, które robią dużą różnicę.

**meteor/check**

Pakiet, który zezwala kontrolować strukturę danych obiekta. Pakiet zawiera dwie funkcje: check oraz test. Pierwsza wyrzuca błąd gdy druga zwraca true lub false.

Ten pakiet jest bardzo użyteczny, o ile musimy kontrolować argumenty, które przychodzą z zewnątrz do metod. Tak można sprawdzić czy pola są odpowiedniego typu i czy zawierają dobre wartości.

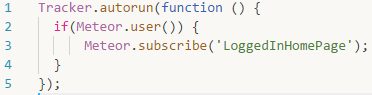
Przykład użycia:



**Tracker.autorun**

Funkcja, która umożliwia uruchomienie kodu, jeżeli zmieniają się jej zależności. Zwraca obiekt, który można użyć do zatrzymywania obserwowania. Ten mechanizm jest bardzo wydajny i często używany. Zależnościami są reaktywne źródła danych (takie jak kolekcje). Nie musimy je jakoś specjalnie zaznaczać, wystarczy użyć je w funkcji.

Przykład użycia:



**HTTP.call**

Funkcja, która daje możliwość wykonać żądanie http do serwerów zewnętrznych. Może być wywołana synchronicznie lub asynchronicznie. Jest użyteczna, kiedy chcemy użyć zewnętrznych API.

W taki sposób możemy dowiedzieć się o bieżącej pogodzie:



## System zarządzania użytkownkami systemu

Meteor posiada gotowy system do zarzadzania użytkownikami systemu. Jest to dużą zaletą tej platformy, o ile realizacja systemu kont użytkownika oraz bezpieczeństwa jest bardzo czasochłonna. Meteor jest zbudowany w taki sposób, że automatycznie udostępnia \_id użytkownika w kontekście Metod oraz Publikacji. Także jest prosty sposób identyfikacji użytkownika po stronie klienta.

Większość funkcjonalności jest zapakowana w pakiecie pod nazwą accounts-base. Ten pakiet zajmuje się inicjalizacją oraz utrzymaniem kolekcji users, tworzy dla niej standardową schemę oraz udostępnia kolekcje za pomocą klasy-singletonu Meteor.users. Ta klasa zawiera dużą ilość generycznych metod.

Pakiet accounts-base odpowiada za mechanizm logowania oraz za szyfrowanie haseł użytkownika.

Meteor posiada cały zestaw pakietów do logowania za pomocą zewnętrznych platform:

* **Facebook** – *accounts-facebook*
* **Google** – *accounts-google*
* **GitHub** – *accounts-github*
* **Twitter** – *accounts-twitter*
* **Meteor Developer** – *accounts-meteor-developer*
* Inne: **Meetup** i t.d.

Tak przykładowe logowanie do platformy Facebook wygląda następująco:



Dalej warto omówić najważniejsze funkcje API systemu do zarządzania użytkownikami:

**Wyciągnięcie obiektu zalogowanego użytkownika:**

Meteor.user()

**Wyciągnięcie identyfikatora zalogowanego użytkownika:**

Meteor.userId()

**Kolekcja użytkowników aplikacji:**

Meteor.users()

**Logowanie użytkownika:**

Meteor.loginWithPassword(user, password, [callback])

Argumentem user może być username lub email.

**Wylogowanie użytkownika:**

Meteor.logout([callback])

**Tworzenie użytkownika:**

Accounts.createUser(options, [callback])

Argument options jest obiektem, który zawiera: username, email, password, profile.

**Dodanie email:**

Accounts.addEmail(userId, newEmail, [verified])

**Usunięcie email:**

Accounts.removeEmail(userId, email)

**Weryfikacja email:**

Accounts.verifyEmail(token, [callback])

Token można otrzymać w emailu weryfikacyjnym

**Wysyłka maila weryfikacyjnego:**

Accounts.sendVerificationEmail(userId, [email], [extraTokenData])

**Zmiana hasła:**

Accounts.changePassword(oldPassword, newPassword, [callback])

Jest to odrobina możliwości API do zarządzania kontem użytkownika. Spróbowałem umieścić najczęściej używane funkcjonalności.

# 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:

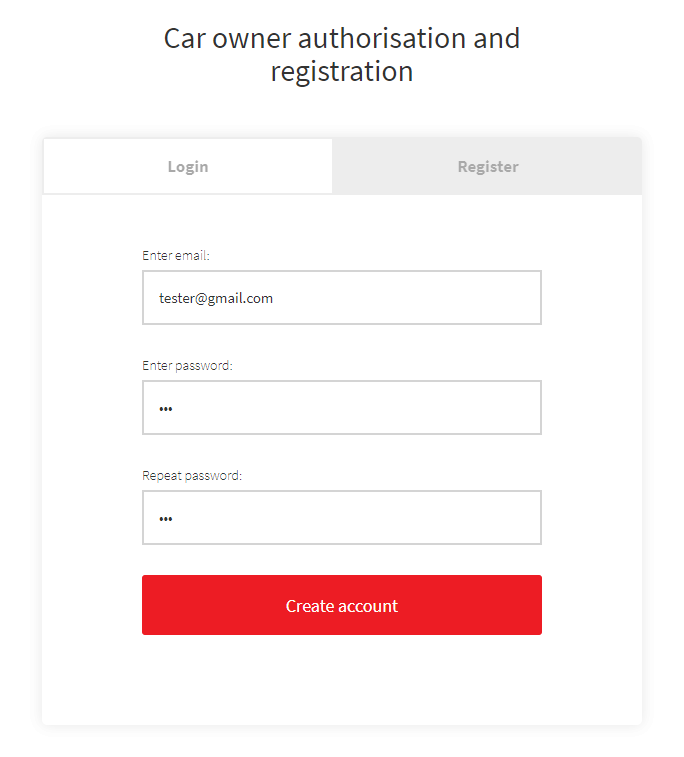
* Logowanie do systemu oraz wylogowanie się z systemu.
* Rejestracja użytkownika w systemie.
* Edytowanie profilu, ustawień.
* Wybór taryfy oraz zapis karty płatniczej.
* Edytwoanie profilu stacji napraw, podanie adresu, wyszukiwanie na mapie, rating.
* Weryfikacja stacji napraw według dokumentów prawnych (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

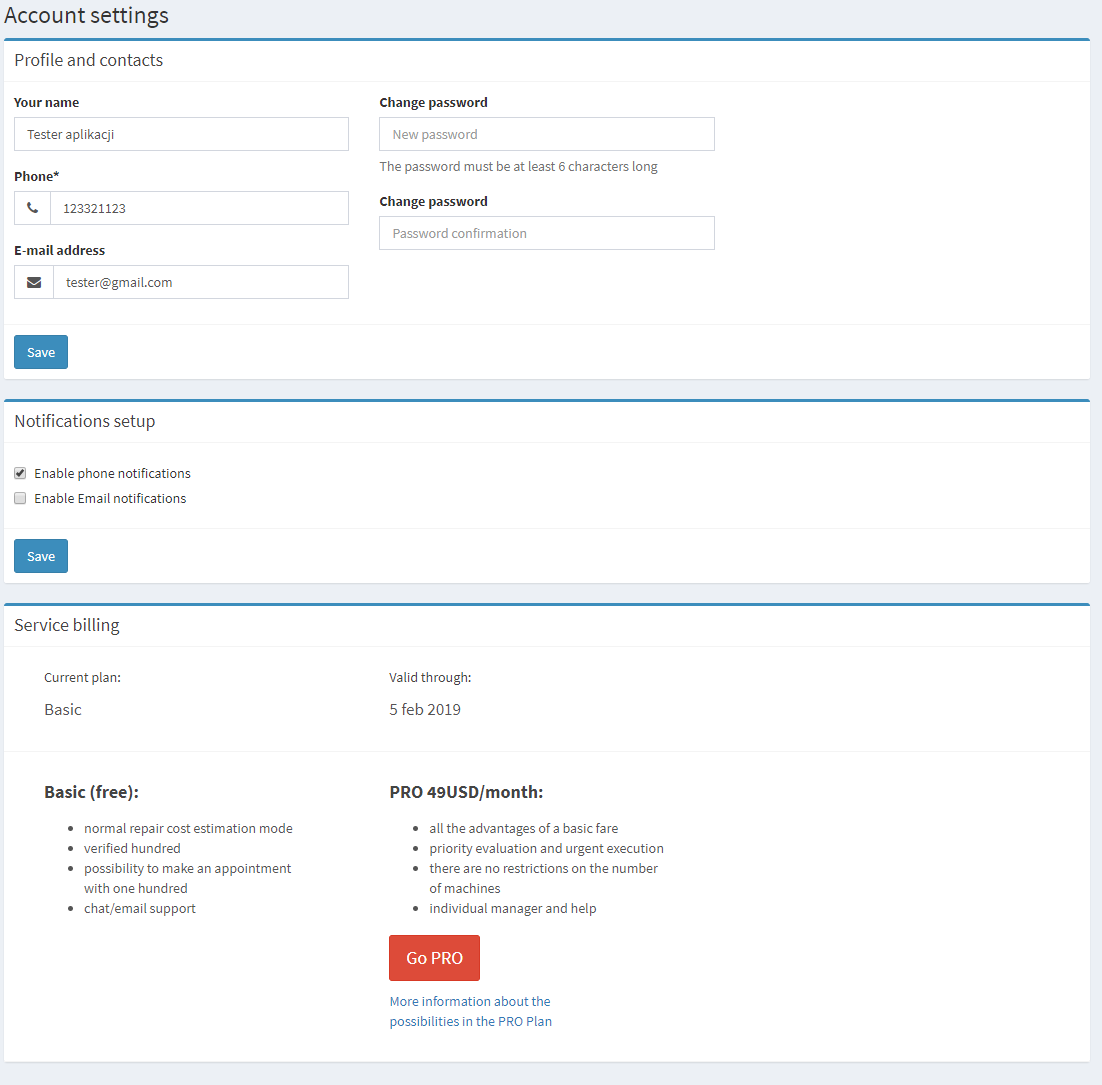
Na poniższych rysunkach prezentują się główne elementy aplikacj.

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.



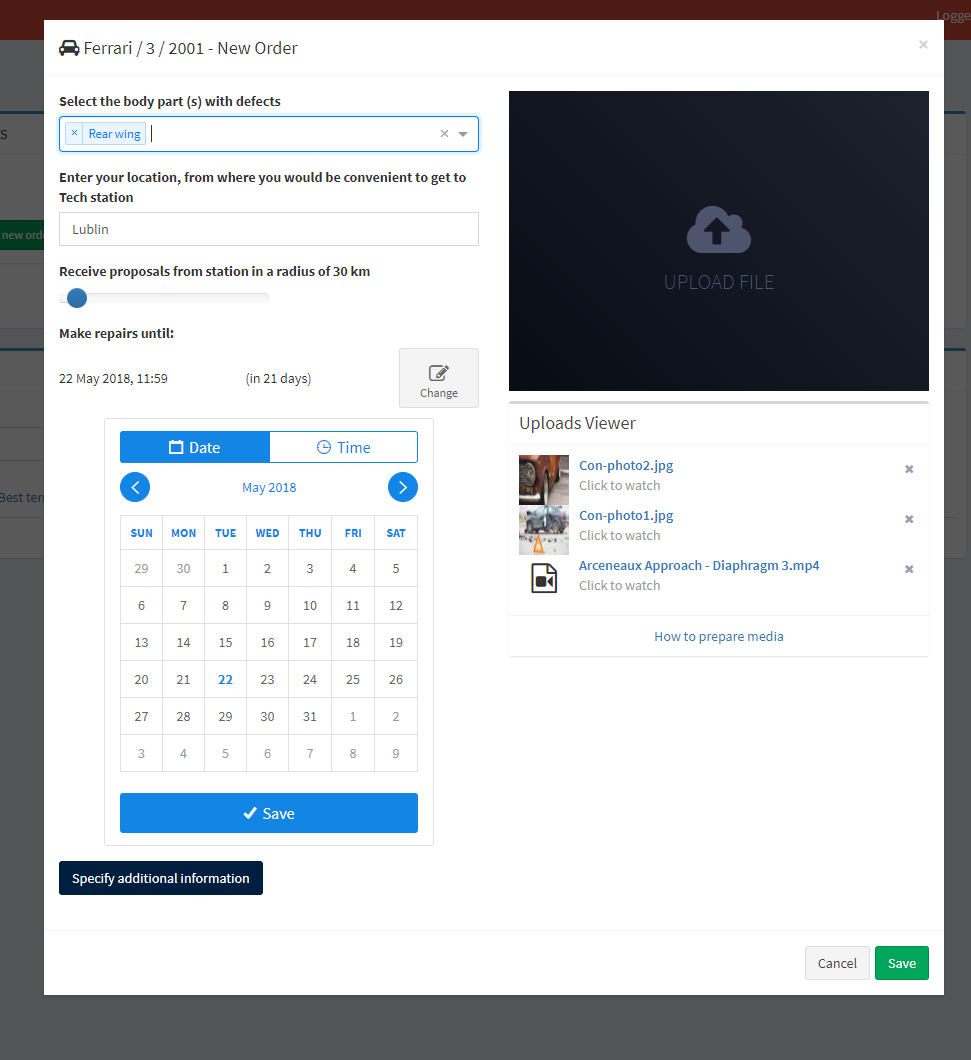
Rysunek 33. Forma rejestracji / logowania nowego użytkownika

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.



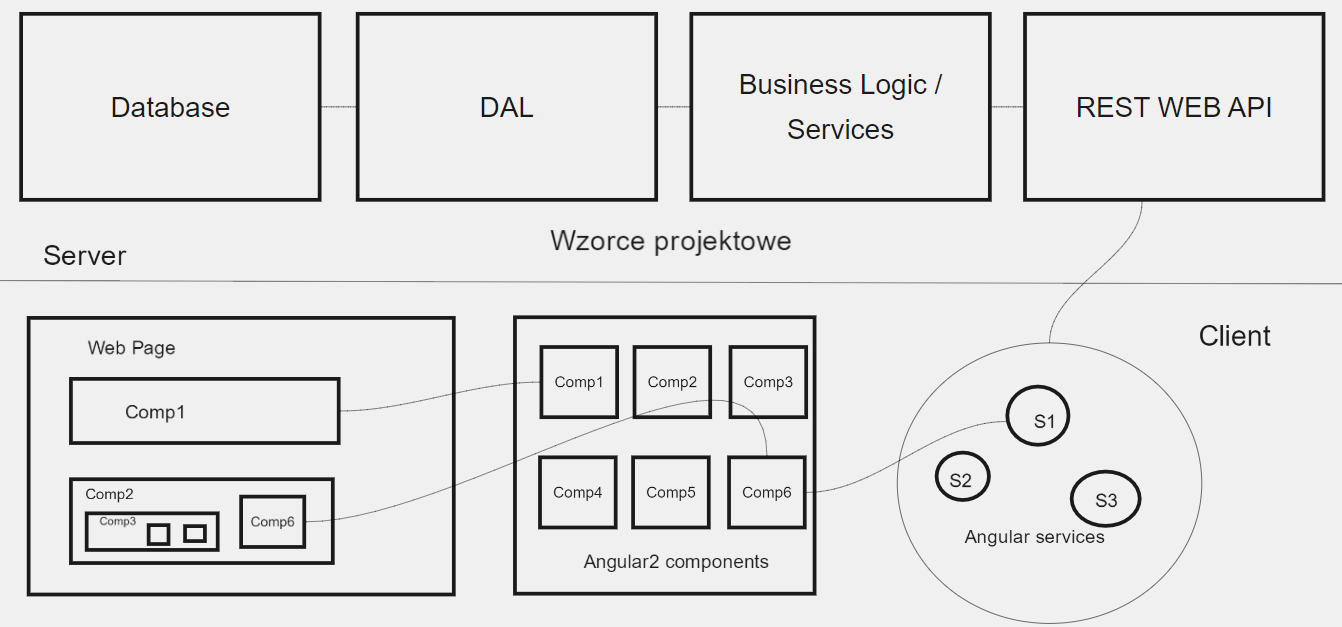
Rysunek 35. Tworzenie zamówienia poprzez właściciela samochodu

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

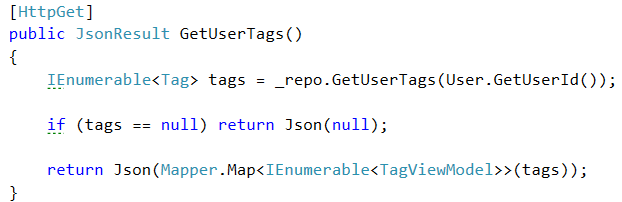
Na Rysunku 38 można zobaczyć warstwy aplikacji Finances. Aplikacja składa się z dwóch części – strony klienta oraz serwerowej.

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).

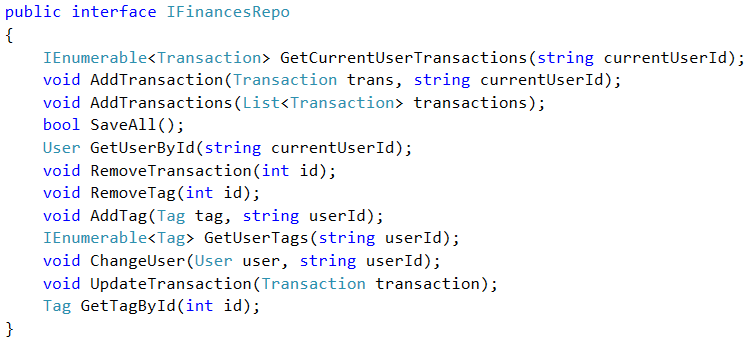


Rysunek 40. Złożony Action w kontrolerze restowym

## 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.

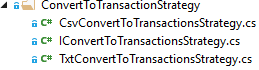
Interfejs klasy wygląda Jak na Rysunku 41:



Rysunek 41. Interfejs IFinancesRepo

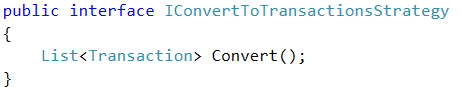
Dany mechanizm wykorzystują rest serwisy, serwisy zwykłe oraz logika biznesowa aplikacji. Także w aplikacji został użyty wzorzec strategy, do zarządzania ładowaniem plików. W taki sposób tworzymy różne strategie dla ładowania plików .csv, .txt oraz innych rozszerzeń.

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.