

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

SPIS TREŚCI	3
WSTĘP.....	5
ROZDZIAŁ 1. ZAPOZNANIE Z POJĘCIAMI, TECHNOLOGIAMI ORAZ INSTRUMENTEM DEVELOPERSKIM	7
1.1. Strona Internetowa	7
1.2. Architektura aplikacji webowej	7
1.3. Meteor.....	11
1.4. Struktura folderów projektu.....	12
1.5. Model abstrakcyjny architektury rozwiązania na platformie Meteor	13
1.6. Meteor CLI.....	16
ROZDZIAŁ 2. FRONTEND.....	19
2.1. ES 2015.....	19
2.2. React	20
2.3. Redux.....	23
2.4. Stylizowanie aplikacji webowej.....	25
ROZDZIAŁ 3. BACKEND.....	29
3.1. Wstęp ogólny do platformy Node	29
3.2. Silnik JavaScript oraz specyfikacja ECMAScript	30
3.3. Node oraz V8	31

3.4.	Jakie problemy rozwiązuje Node.js	32
3.5.	Baza danych MongoDB	33
3.6.	Instrumenty zarządzania MongoDB	35
3.7.	Meteor.js MongoDB API CRUD	36
3.8.	Publish & Subscribe.....	38
3.9.	Meteor Methods.....	40
3.10.	Kwestie bezpieczeństwa.....	42
3.11.	Poręczne funkcje standardowe Meteor.js.....	43
3.12.	System zarządzania użytkownikami systemu	44
ROZDZIAŁ 4. TWORZENIE APLIKACJI WEBOWEJ NA PLATFORMIE METEOR.....		47
4.1.	Opis aplikacji.....	47
4.2.	Warstwy aplikacji	57
4.3.	Szczegóły implementacji	59
ZAKOŃCZENIE.....		63
BIBLIOGRAFIA.....		65

Wstęp

Celem mojej pracy jest opisanie procesu tworzenia nowoczesnej strony internetowej z użyciem platformy Meteor.

Czasami nie zdajemy sobie sprawy o tym, jak trudno jest zaimplementować poszczególne części strony internetowej w taki sposób, że działa ona szybko oraz poprawnie na wszystkich urządzeniach docelowych. Dla mnie osobiście największym wyzwaniem było nauczenie się języka funkcyjnego – JavaScript. Przez dłuższy czas miałem okazję pracować w środowiskach obiektowych. Podejścia używane wcześniej trzeba było zastąpić innymi, zmienić sposób myślenia. Tworzenie zaawansowanych aplikacji w języku JavaScript ma swoje zalety oraz wady. Jest jeden aspekt JavaScript, który mi się bardzo podoba – jest to bardzo luźny i zabawny język. Ciekawe jest to, że luźny on jest głównie w miejscach, gdzie na tym mocno zyskujemy. Pisanie aplikacji w JavaScript – to jedna przyjemność. Jeszcze większą przyjemność sprawia użycie platformy Meteor, która zarządza ustawieniem projektu oraz zapewnia możliwość użycia najnowszych funkcji dostępnych w społeczeństwie JavaScript. Meteor rozwiązuje wszystkie problemy, które mogą wstrzymać lub spowolnić napisanie aplikacji: uruchomienie serwera oraz bazy danych, odświeżanie aplikacji w czasie rzeczywistym, aktualizacja pakietów, zaimplementowana ‘reaktywność’ itd.

W pierwszym rozdziale opowiadam głównie o strukturze rozwiązań web oraz rozwiązań na platformie Meteor. Opowiadam o sposobach interakcji z platformą.

W drugim rozdziale skupiono się na części aplikacji, odpalanej w przeglądarce. Opisane możliwości nowej składni JavaScript. Zrobiłem też wprowadzenie w takie technologie jak *React* oraz *Redux*. Opisałem problemy, jakie występują pod czas implementacji stylu aplikacji oraz ich rozwiązanie.

Trzeci rozdział odpowiada za wyjaśnienie części serwerowej na platformie Meteor. Opisałem dokładnie, jak działa środowisko *Node* oraz jak pracować z bazą danych *MongoDB*. Pod koniec rozdziału opisałem kluczowe funkcje Meteor API.

W ostatnim rozdziale przedstawiono aplikację webową, napisaną z użyciem platformy Meteor oraz najbardziej nowoczesnych na daną chwilę technologii i podejść.

Rozdział 1. Zapoznanie z pojęciami, technologiami oraz instrumentem developerskim

1.1. Strona Internetowa

W tej pracy termin „Strona internetowa” (ang. *Website*) będzie używany 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 najprostsze strony internetowe opierają się na zestaw technologii serwerowych / API. Ten aspekt zezwala mi uważać nowoczesną stronę internetową za normalną aplikację, porównywalną z desktopową, 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. Jeśli 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 połączanego z Internetem. Za przechowywanie oraz bezpieczeństwo danych odpowiada firma tworząca web aplikację, co zdejmuje część odpowiedzialności z użytkownika.

1.2. Architektura aplikacji webowej

Architektura aplikacji webowej – to jeden ze sposobów spojrzenia na aplikację z wyższego poziomu. Ja miałem możliwość pracować z dwoma ze trzech architektur przedstawionych poniżej. Ale zanim można będzie 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 narzędzi 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*) – tym pojęciem też 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.

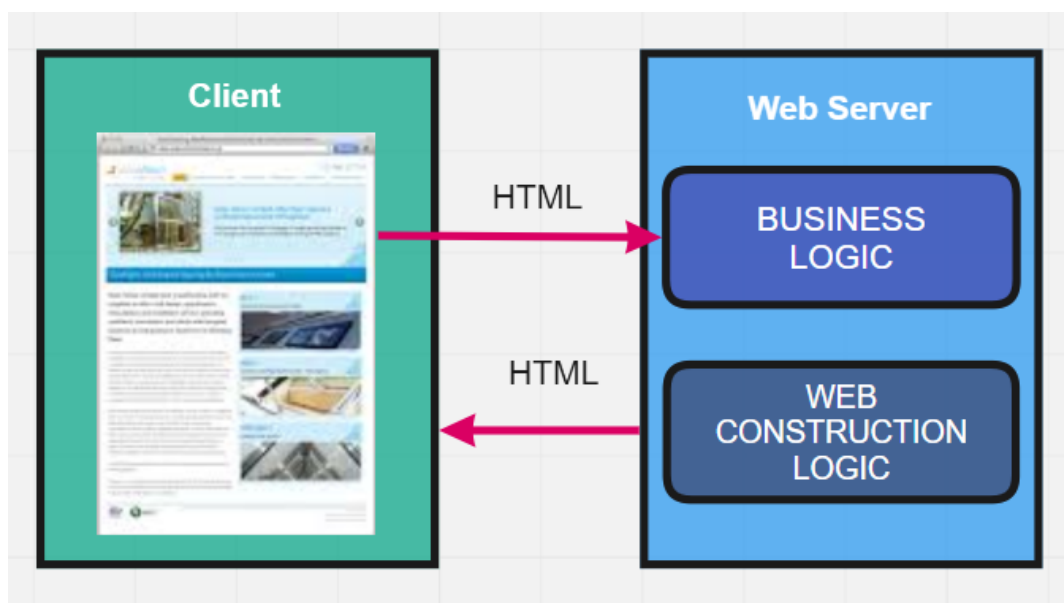
Intrygującym aspektem jest to, że praca nad częścią klienta zajmuje dużo więcej czasu. Jest to związane, że środowiska (przeglądarki) oraz narzędzia od dłuższego czasu nie były wystarczająco standaryzowane. Wielki wpływ to miało na szybkość oraz jakość napisania stylów strony. Przez dłuższy czas nie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

Teraz, kiedy znamy różnicę między stroną klienta oraz serwer, możemy porównać różne 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.

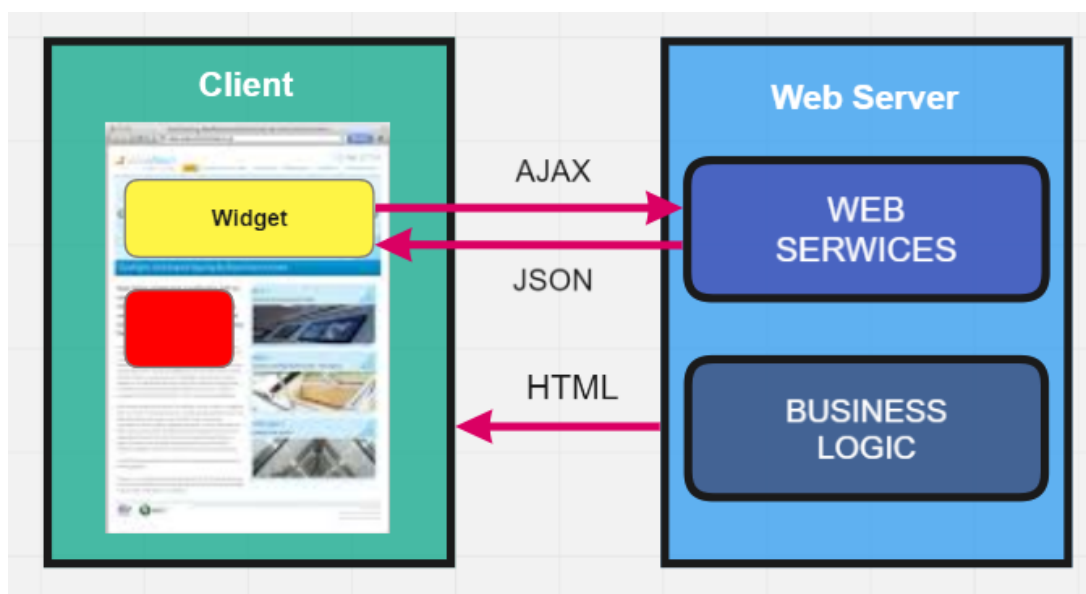


Rysunek 1

- Widget web application

Jest to kolejny krok, jeżeli chodzi o budowę widoku strony internetowej. Strona składa się z bloków o nazwie widżet. 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

rozmiszczono jest więcej logiki po stronie klienta, co wprowadza większe ryzyko związane z bezpieczeństwem.

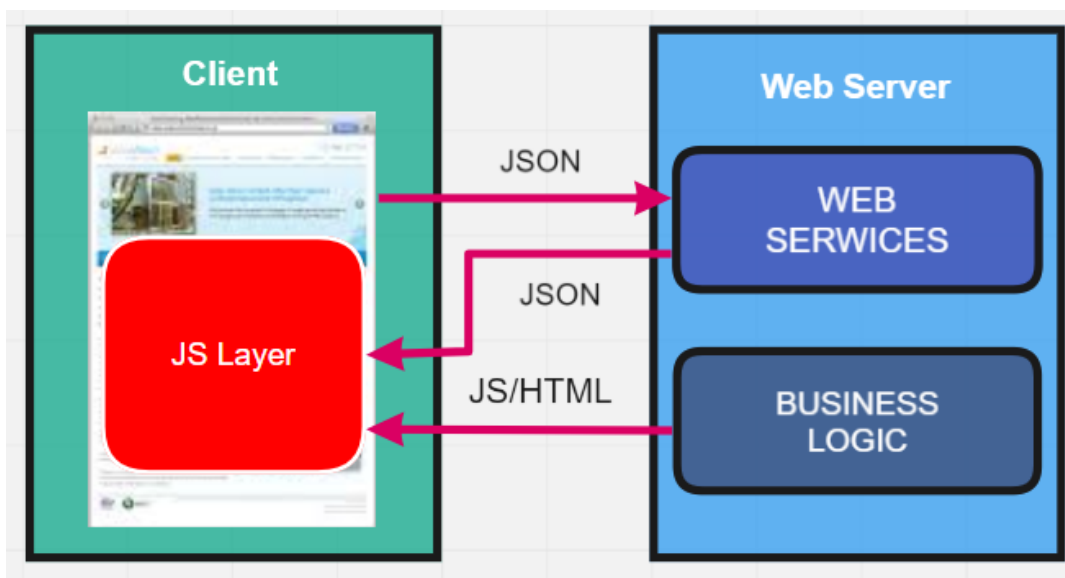


Rysunek 2

- 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 widżetem, 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*.



Rysunek 3

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

1.3. Meteor

Meteor – to platforma najwyższego poziomu, zawierająca cały podzbiór innych bibliotek 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. Jest to narzędzie-aplikacja konsolowa, która mocno wspiera proces tworzenia aplikacji na platformie Meteor. Przy pomocy tego narzędzia mamy możliwość tworzenia bazowego szkieletu aplikacji, budowę aplikacji, instalację pakietów *Atmosphere*, podbijanie wersji Meteor, generowanie hybrydowej aplikacji mobilnej i dużo więcej.

Jednym z ważnych aspektów, na które patrzymy pod czas wyboru platformy lub zestawu narzędzi 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 lepiej udokumentowany 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 społeczność tworzące ten pakiet. W przypadku popularnych

bibliotek większa jest szansa 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 społeczność JavaScript oraz Node.JS. Własne pakiety są dostosowane do platformy Meteor 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*. 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 społeczność.

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).

1.4. Struktura folderów projektu

Decyzja o tym gdzie rozmieszczać pliki oraz foldery w Meteor jak i w innych platformach ma wpływ nie tylko na wygodę pod czas pracy z projektem, ale i na samo działanie aplikacji. Już wiemy, że pisząc aplikację na platformie Meteor 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 różne operacje. Żeby móc kontrolować to, gdzie chcemy odpalać nasze instrukcje, Meteor 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 dostępne 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 agregowany do głównych plików i ładowany do modułów w folderach *'client'* oraz *'server'*.

1.5. Model abstrakcyjny architektury rozwiązania na platformie Meteor

Pod czas projektowania nowego rozwiązania na platformie warto znać jak wygląda połączenie głównych składników Meteor. Każda platforma musi mieć możliwość obsługiwać bazy danych. Meteor 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 one zostały przez społeczność Meteor i są umieszczone na platformie *Atmosphere*.

Meteor używa podejście ‘baza danych wszędzie’ (ang. *database everywhere*), co oznacza to, że mamy możliwość działania na bazie też po stronie klienta. Synchronizacja danych działa za pomocą protokołu DDP, konsumującego technologię Web Sockets.

Web Sockets – nowoczesna technologia, która określa nowy sposób nieprzerwanej 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.



Rysunek 4

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

Wystarczy zainstalować odpowiedni pakiet *Atmosphere*, a kompilacją oraz budowaniem paczki zajmie się sam Meteor. Pomimo tego Meteor jest dostarczany z kompilatorem 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 syntaktyczne, 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 interfejsu użytkownika: tworzenie ładnie stylizowanych komponentów, interakcja z użytkownikiem, przywiązywanie danych do komponentów. Dla tego bardzo ważnym jest dostarczanie najlepszych instrumentów developerskich do pracy z *frontendem*. Meteor zdaje egzamin na 5.

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

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

Chciałbym też opisać, dla czego wybrałem *React*, 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*. Zaletą jest dobra integracja z platformą Meteor. Podejście do renderowania podobne do *Angular*.

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 różni się z *React*. *Angular.js* używa dodatkowe atrybuty 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 kompilowanego *TypeScript*. Też wadą jest rozmiar oraz złożoność technologii – jest to cały *framework*, gdy *React* – to mała biblioteka.

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

Vue – *framework*, stworzony przez jedną osobę a później wspierana przez społeczeństwo. Zawiera najlepsze cechy *Angular.js*, *Angular* oraz *React*. Główną wadą jest to, że *framework* nie

może pochwalić się wsparciem takich gigantów jak Google lub Facebook oraz jest relatywnie nowym na rynku.

1.6. Meteor CLI

Meteor CLI jest głównym interfejsem do pracy z platformą. Przy pomocy Meteor CLI można tworzyć nowe projekty Meteor 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* 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 automatycznie 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*, można użyć zmiennej środowiskowej *'NODE_OPTIONS'*.

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

meteor create nazwa-projektu

To polecenie tworzy nowy projekt szablonowy Meteor 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 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 wersję Meteor 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 dokleając odpowiednie flagi tuż po nazwie pakietu.

meteor remove *nazwa-pakietu*

Usuwa pakiet 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. O ile na maszynie można zainstalować oddzielny *Node* oraz NPM, warto uważać na to, żeby instalować pakiety używając dostarczanych rozwiązań o ile wersje narzędzi mogą się różnić, 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 implementacji aplikacji. Narzędzie jest bardzo wygodne oraz minimalistyczne.

Rozdział 2. Frontend

2.1. ES 2015

Aplikacje Meteor są pisane z użyciem jednego języka programowania – JavaScript. W tej chwili ES 2015 jest najlepszą wersją języka. Meteor zajmuje się wsparciem oraz kompilacją składni ES 2015 dla starszych przeglądarek. Dalej opiszę nowe konstrukcje oraz cechy, dostępne w ES 2015.

let, const

Jest to nowy sposób definiowania zmiennych. Ich zaletą jest to, że oni istnieją w zasięgu bloku. Wcześniej zmienne mieli zasięg globalny lub funkcji, w której były zdefiniowane. Takie zachowanie powodowało niechciane nadpisywania zmiennych, które było trudno wyłapać.

Arrow Functions

W nowej wersji dostaliśmy nowy sposób tworzenia funkcji. Jest to zdecydowanie krótszy zapis. Pomimo tego uruchomienie takiej funkcji nie zmienia wartości słowa kontekstowego *this*, co jest też bardzo wygodne.

Szablonowy string

Szablonowy string (ang. *Template String*) – to nowy sposób na konkatencję zmiennych z tekstem. Taki sposób jest krótszy, czytelniejszy oraz mniej podatny na błędy.

Skrócona definicja obiektów

Nowy, krótszy sposób tworzenia obiektów, które używają zmienne o takiej samej nazwie jak właściwości obiektu.

Natywne funkcje JavaScript do pracy z tablicami

Zestaw gotowych funkcji do pracy z tablicami. Wcześniej oni nie były wspierane i trzeba było używać bibliotek oddzielnych.

Destrukcja obiektów

To jest bardzo poręczny mechanizm, który pozwala zdefiniować listę zmiennych na podstawie pól obiektu. Rozbijać też można obiekty, które są argumentami funkcji.

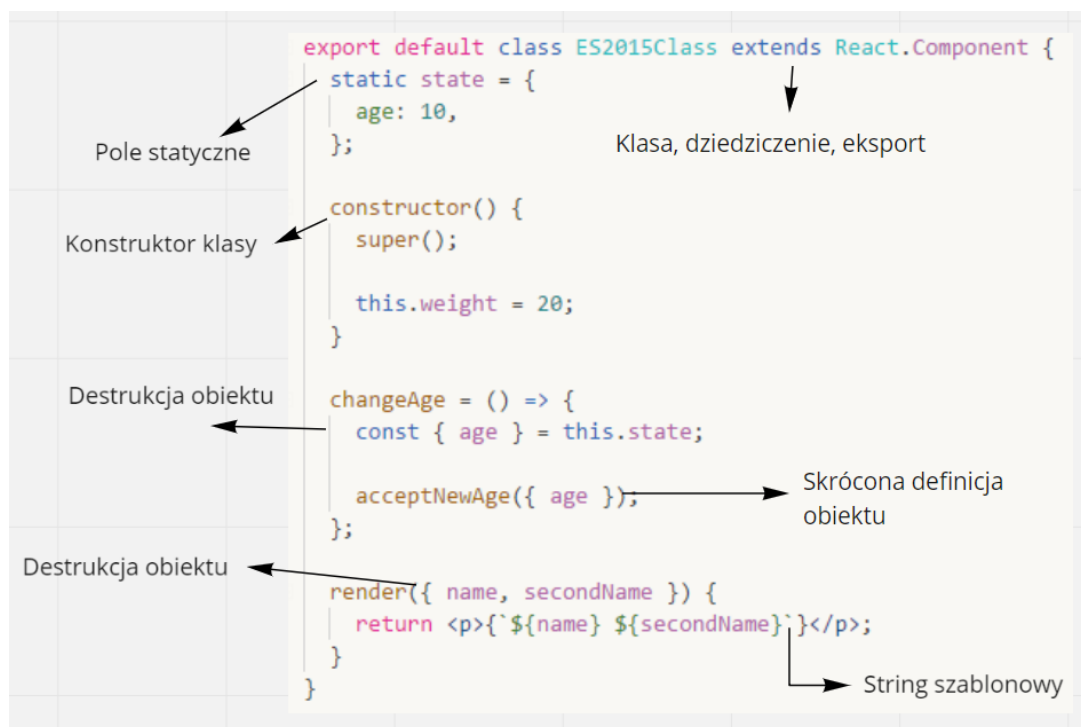
Klasy ES2015

Klasy to nowy syntaks oraz poziom abstrakcji owijający mechanizm prototypów JavaScript. Syntaks jest bardzo podobny do języków obiektowych. Klasa może zawierać pola, metody, statyczne członki oraz może posiadać konstruktor. Klasa też może dziedziczyć po innych klasach.

System importów oraz eksportów ES2015

Jedną tą zmianą robi dużą różnicę. Bardzo ułatwia zarządzanie ładowaniem plików.

Na rysunku 5 pokazane jest użycie nowych konstrukcji ES 2015.



Rysunek 5

2.2. React

Podczas tworzenia aplikacji Meteor, trzeba użyć jednego z dostępnych silników do renderowania. Taki silnik może być dostarczany w formie niewielkiej biblioteki lub całego *frameworku*. Moim wyborem w tej chwili jest React.

React – to biblioteka JavaScript, która zezwala budować interfejs użytkownika.

Biblioteka działa w następujący sposób: funkcja `ReactDOM.render` wstrzykuje drzewo komponentów w DOM strony HTML.

DOM (ang. *Document Object Model*) – to drzewo obiektów JavaScript, które reprezentują strukturę HTML strony internetowej.

Komponent jest jednostką celem, którego jest zwrócenie elementu React. React element może zawierać listę zagnieżdżonych w sobie elementów. React zawiera zestaw funkcji JavaScript, które, generują odpowiednie elementy. Każdy nowoczesny element HTML posiada swój odpowiednik elementu React. Przykładowo, React posiada elementy takie jak *div*, *h1*, *p*, *a* itd.

Za pomocą takiego mechanizmu można tworzyć zagnieżdżone funkcje JavaScript, które będą reprezentować gałąź drzewa DOM.

Na rysunku 6 za pomocą klasy ES 2015 *MyFirstComponent*, tworzymy pierwszy komponent, który zwraca element *div* z zawartością *Hello world*. Dalej wstrzykujemy ten element w DOM za pomocą funkcji *ReactDOM.render*.

```
1 | import React, { Component } from 'react';
2 | import ReactDOM from 'react-dom';
3 |
4 | class MyFirstComponent extends Component {
5 |   render() {
6 |     return React.createElement('div', null, 'Hello world');
7 |   }
8 | }
9 |
10 | ReactDOM.render(
11 |   React.createElement(MyFirstComponent, null, null),
12 |   document.getElementById('body'),
13 | );
```

Rysunek 6

Taka składnia tworzenia elementów ma więcej wad niż zalet, o ile przy tworzeniu złożonego widoku, musielibyśmy zagnieżdżać dużą ilość funkcji *React.createElement*. Dzisiaj standardem, przy tworzeniu komponentów React jest użycie JSX.

JSX – to rozszerzenie syntaktyczne JavaScript, które kompiluje się w elementy React.

Na rysunku X można zobaczyć jak wygląda komponent z użyciem składni JSX. Warto zauważyć, że pod spodem taka składnia generuje ciąg wywołań funkcji *React.createElement*.

```

4 | class MyFirstComponent extends Component {
5 |   render() {
6 |     return <div>Hello world</div>;
7 |   }
8 | }

```

Rysunek 7

Tworzenie elementów HTML nie jest jedynym problemem, który trzeba rozwiązać pod czas tworzenia interfejsu użytkownika aplikacji. Ważną częścią jest mechanizm przepływu oraz przywiązania danych. Przepływ danych do komponentu odbywa się za pomocą *'props'* – to są właściwości komponentu. Komponent w React może zostać stworzony w postaci zwykłej funkcji zwracającej JSX. W takim przypadku *props* – to automatycznie przekazywany argument tej funkcji zawierający dane przekazane zewnątrz. Przykład tworzenia i użycia komponentu funkcyjnego z *props* można zobaczyć na rysunku 8:

```

4 | const HelloWorld = props => <div>Hello {props.name}</div>;
5 |
6 | const MainComponent = () => (
7 |   <div>
8 |     <HelloWorld name="Wladyslaw" />
9 |   </div>
10 | );

```

Rysunek 8

Ważnym aspektem jest to, że komponent jest funkcją czystą (ang. *Pure Function*) – to znaczy, że nie mutuje ona przychodzące argumenty oraz nie posiada efektów ubocznych.

Pomimo zarządzania danymi przychodzącymi, komponent React może posiadać stan własny (ang. *state*). Stan lokalny może posiadać wyłącznie komponent-klasa, dziedzicząca po klasie *React.Component* lub *React.PureComponent*. *State* jest bardzo poręczny do przechowywania danych wejściowych użytkownika. Zmiany w *state* powodują ponowne renderowanie komponentu z użyciem nowych danych.

Na tym można skończyć krótki opis React. Określiłem główne pojęcia i mechanizmy React. Warto zauważyć, że przy pisaniu komponentów React, ich, jakość zależy mocno od znajomości języka JavaScript, szczególnie specyfikacji ES 2015.

2.3. Redux

Redux – to przewidywalny kontener stanu, stworzony dla użycia w aplikacjach JavaScript. Ta biblioteka pozwala lepiej zarządzać stanem komponentów React.

Wcześniej, trzeba było przekazywać stan do komponentów potomnych za pomocą *props*. Teraz jest centralny, kontrolowany schowek (ang. *Store*), który umieszcza stan lub część stanu aplikacji. To bardzo ułatwia zarządzanie stanem aplikacji, ale z innej strony wprowadza dość skomplikowany mechanizm do zarządzania schowkiem. Najwięcej zyskują rozbudowane aplikacje, gdzie bez Redux, stanem komponentów zarządzać jest bardzo ciężko. Niżej spróbuję szybko opisać kluczowe składniki Redux.

store

Jest to obiekt, który zwraca funkcja *createStore*. Ta funkcja przyjmuje jeden lub więcej *reducer*. Obiekt reprezentujący schowek Redux (ang. *Redux Store*), posiada funkcje do wyciągania danych – *getState*, do modyfikacji schowku – *dispatch* oraz do rejestracji słuchaczy (ang. *Listeners*). Aplikacja, która używa Redux posiada tylko jeden schowek. Na rysunku 9 można zobaczyć funkcję, generującą schowek Redux.

```
10 export default () => {
11   const store = createStore(
12     combineReducers({
13       ui: uiReducer,
14       auth: authReducer,
15       stoOrders: stoOrdersReducer,
16     }),
17     composeEnhancers(applyMiddleware(thunk)),
18   );
19
20   return store;
21 }
```

Rysunek 9

Ten schowek zawiera kombinację *reducerów* oraz posiada zarejestrowane rozszerzenie, które zezwala tworzyć akcje Redux (ang. *Redux actions*) w postaci funkcyjnych generatorów.

actions

Akcja Redux – to główne źródło informacji dla schowku globalnego Redux. Akcja jest zwykłym obiektem, który posiada właściwość obowiązkową o nazwie *'type'* oraz dane, związane z tą

akcją. Akcje są aplikowane za pomocą funkcji obiektu schowku *store.dispatch*, gdzie akcja jest przekazywana, jako parametr. Obsługą akcji dalej zajmuje się *reducer*.

reducer

Reducer odpowiada za modyfikację schowku Redux. Po wywołaniu *dispatch*, przekazana w parametrze akcja trafia do *reducera*. Akcja określa, co się stało, gdy *reducer* decyduje, co zrobić z takim wydarzeniem. *Reducer* jest funkcją czystą, która przyjmuje na wejściu początkowy stan schowku oraz akcję i zwraca nową kopię schowku. Na rysunku 10 można zobaczyć przykładowy *reducer*, obsługujący paginację na stronie.

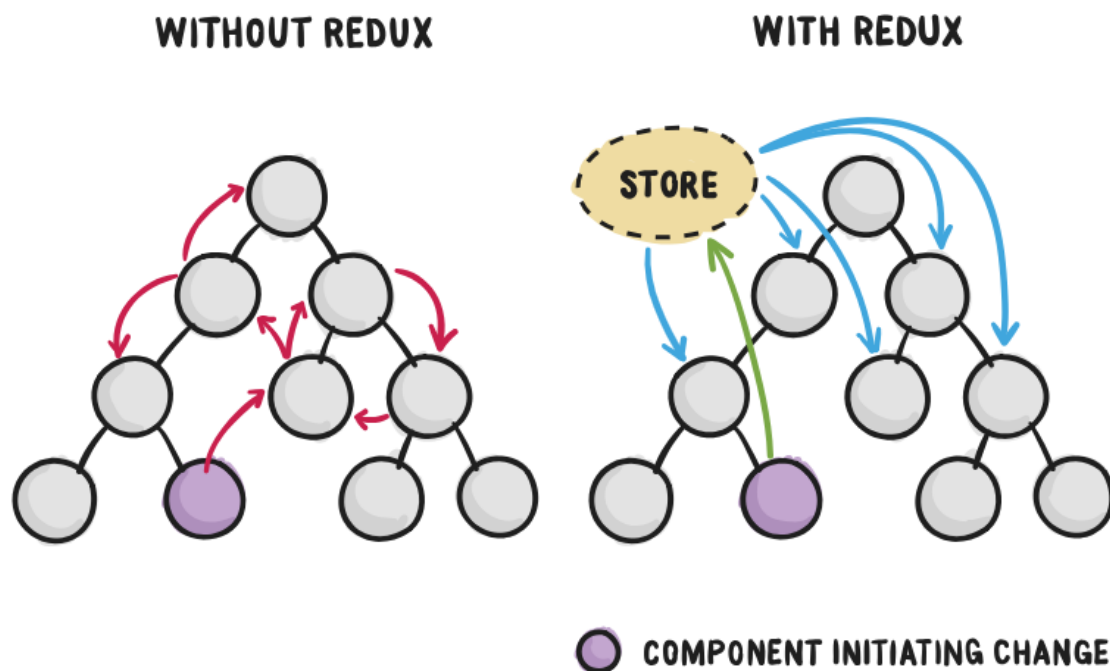
```
7   export default (state = defaultState, action) => {
8     const stateCopy = JSON.parse(JSON.stringify(state));
9
10    switch (action.type) {
11      case 'CHANGE_PAGE':
12        stateCopy.pagesToSkip = action.currentPageNumber * ITEMS_PER_PAGE;
13        return stateCopy;
14      default:
15        return state;
16    }
17  };
```

Rysunek 10

Adaptacja z React

Dla tego, żeby używać Redux z React należy zainstalować pakiet o nazwie *'react-redux'*. Ten pakiet dostarcza funkcję *connect*, która pozwala przekazać do *props* dane schowku a także wywoływać *dispatch* predefiniowanych akcji.

Na rysunku 11 można zobaczyć jak wygląda przepływ danych do komponentu w przypadku Redux oraz React. Także pokazane jest to, jak odbywa się komunikacja zwrotna komponentu.



Rysunek 11

2.4. Stylizowanie aplikacji webowej

Stylizowanie stron internetowych od dawna było dużym wyzwaniem. Nadal jest. Spowodowane jest to wadami języka CSS oraz samym deweloperem. Im większy jest projekt tym trudniej jest kontrolować zachowanie stylu. Niżej określę, w czym jest problem.

W czym jest wina CSS

- Kaskada oraz dziedziczenie. To powoduje to, że każdy kawałek CSS napisany dla jednego elementu ma potencjał wpłynąć na style dla kompletnie innego elementu.
- Jest zbyt luźny. Łatwo jest zacząć pisać niekontrolowane style. Winą CSS jest to, że zezwala na to.
- Wysoka zależność od kolejności ładowania plików.
- Nie wyrazisty. Pod czas analizy kodu źródłowego, trudno jest określić cel osoby oraz co dokładnie ten kawałek kodu robi.
- Specyficzne mechanizmy. Są po prostu dziwne zachowania CSS, o których trzeba wiedzieć.
- Specyficzność. Najgorszy aspekt. W jakiej kolejności piszemy kod? Po czym dziedziczymy? To są pytania, na które w dużych projektach można nie znaleźć odpowiedzi.

W czym jest wina dewelopera

- Brak dokumentacji.
- Brak określonej struktury stylów.
- Nie wystarczająco dobra znajomość projektu oraz języka CSS.
- Różne style pisania kodu.

Każdy kawałek CSS ma potencjał przekazać do lub przyjąć od innej części CSS informacje. W taki sposób CSS staje się dużym bałaganem zależności. Żeby rozwiązać taki worek problemów, używam cały zestaw nowoczesnych narzędzi oraz podejść. Spróbuję je niżej opisać.

SASS

SASS – to preprocesor CSS, który rozszerza składnię języka CSS. Niżej określę nowe możliwości, które daje nam SASS.

- Zmienne.
- Zagnieżdżanie.
- Import plików
- Dziedziczenie
- *Mixins* – odpowiednik funkcji.

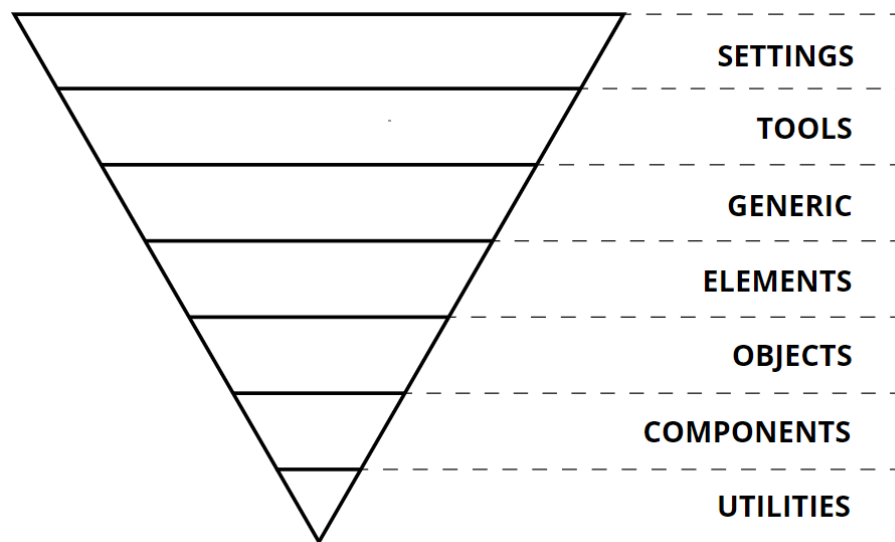
Te funkcje zezwalają pisać bardziej czysty kod. Ale najważniejszą możliwością jest import plików częściowych. Zezwala to tworzyć złożone struktury plików CSS projektu.

ITCSS

ITCSS (ang. *Inverted Triangle CSS*) – to sposób myślenia oraz organizacji plików CSS w projekcie celem, którego jest lepsze zarządzanie kaskadą oraz specyficznością CSS. Głównym pomysłem jest dzielenie CSS projektu na poszczególne warstwy.

Specyficzność CSS – określa szerokość wpływu danego stylu na inne style CSS aplikacji.

Graficzny wygląd takiego podziału można zobaczyć na rysunku 12.



Rysunek 12

Im wyżej jest umieszczony CSS, tym na więcej elementów on będzie miał wpływ. W taki sposób kontrolujemy specyficzność stylów CSS.

Settings

Zawiera globalne zmienne: kolory, rozmiary kontenerów oraz czcionek.

Tools

Zawiera *mixiny* oraz funkcje. Może zawierać generatory *mediaQuery*.

Generic

Style najniższego poziomu. Najlepsze miejsce na wstrzyknięcie *normalize.css*.

Elements

Style, określające bazowy wygląd elementów HTML.

Objects

Style dla abstrakcyjnych obiektów (przycisk, lista itd.)

Components

Style dla specyficznych elementów strony.

Utilities

Kawałki kodu nadpisujące dowolny styl aplikacji. Najwyższa specyficzność.

BEM

BEM (ang. *Block Element Modifier*) – to metodologia, która określa sposób nazewnictwa klas CSS. Także do stylizowania preferowane jest użycie wyłącznie selektorów klas. Na rysunku X umieściłem przykłady HTML z użyciem klas CSS, które używają konwencji BEM.

```
33 <section className="container">
34   <header className="container__header">
35     <h1 className="container__title">Title</h1>
36   </header>
37 </section>
38
39 <section className="container container--rounded">
40   <header className="container__header">
41     <h1 className="container__title--large">Title</h1>
42   </header>
43 </section>
```

Rysunek 13

BEM jest dobry tym, że automatycznie dokumentujemy nasz kod. Na wyjściu mamy CSS, strukturę, którego można określić jednym wzrokiem. Pomimo tego mamy zawsze pewność tego, na który element wpłyną pisane style. BEM także zezwala rozbić CSS na ponownie używalne komponenty.

.block

Jest to pierwsza część nazewnictwa. Określa oddzielny blok strony.

.block__element

Tak wygląda nazewnictwo elementu. Blok może zawierać elementy lub inne bloki.

.block--modifier

Tak wygląda nazewnictwo modyfikatora. Modyfikator jest sposobem na tworzenie alternatywnych stylów dla bloku lub elementu.

.block__element--modifier

Modyfikator też może być zastosowany dla elementu bloku.

Rozdział 3. Backend

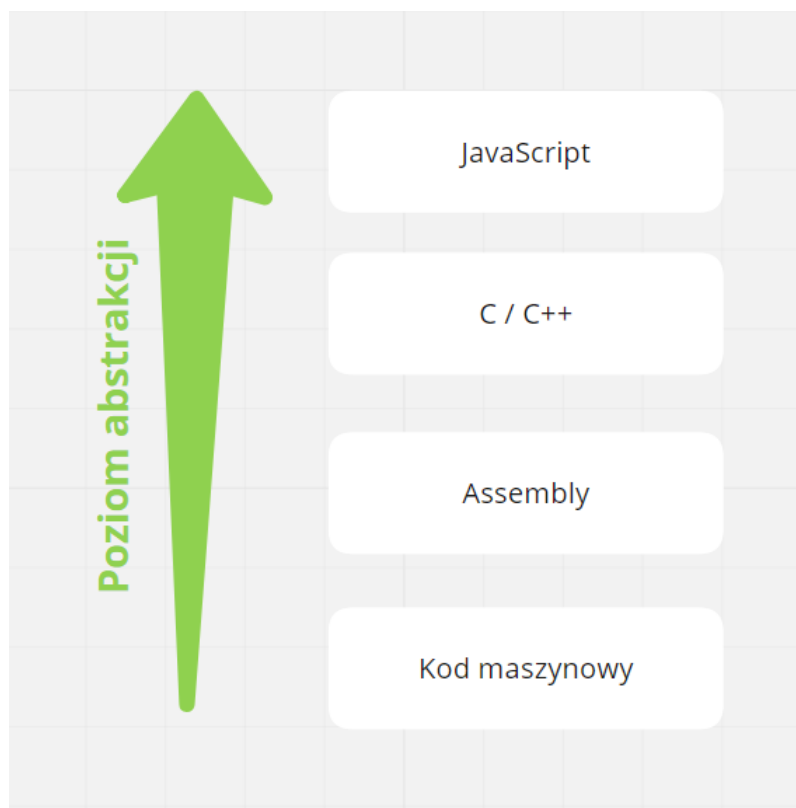
3.1. Wstęp ogólny do platformy Node

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

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

Kod maszynowy – to zestaw instrukcji, które są natywne dla używanej mikro architektury. 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 niskim poziomie abstrakcyjnym.

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



Rysunek 14

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

3.2. 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) zaczęli tworzyć swoje implementacje silników JavaScript, wprowadzając jakieś zmiany. *Ecma International* – to organizacja zajmująca się standaryzacją, celem, której jest określić jak dokładnie ma działać bazowa wersja języka.

Standaryzacja 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 zachowania, określone w odpowiedniej wersji dokumentu *ECMAScript*.

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

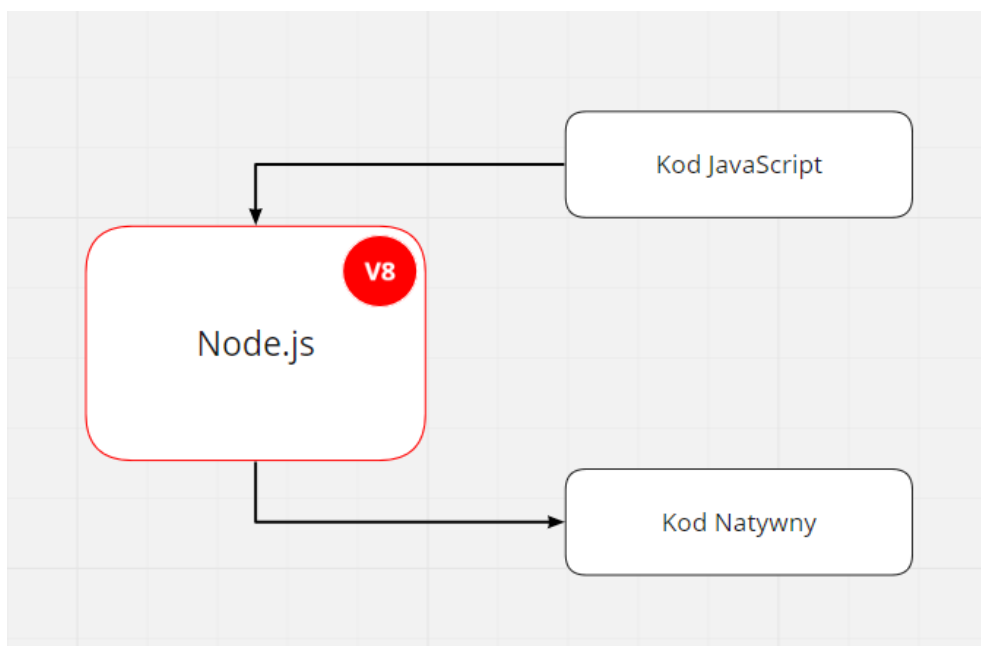
ECMAScript – to standard, na którym jest zbudowany JavaScript.

3.3. Node oraz V8

Głównym celem silnika V8 było użycie go wewnątrz przeglądarki Google Chrome. Jest to produkt z otwartym kodem źródłowym, 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 odpowiedniej 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 niskopoziomowymi 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*.



Rysunek 15

Node – to aplikacja C++, która integruje 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 posiada bibliotekę własną – zestaw kodu JavaScript, który ułatwia wykorzystanie funkcji niskopoziomowych.

3.4. 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, żeby móc wykonywać zadania, jakie wykonuje web server?

Zestaw wymagań umieściłem na poniższą 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. W obu przypadkach możemy zaimportować cały plik, lub obiekty/funkcje/klassy, które zostały wcześniej wyeksportowane.

3.5. Baza danych MongoDB

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

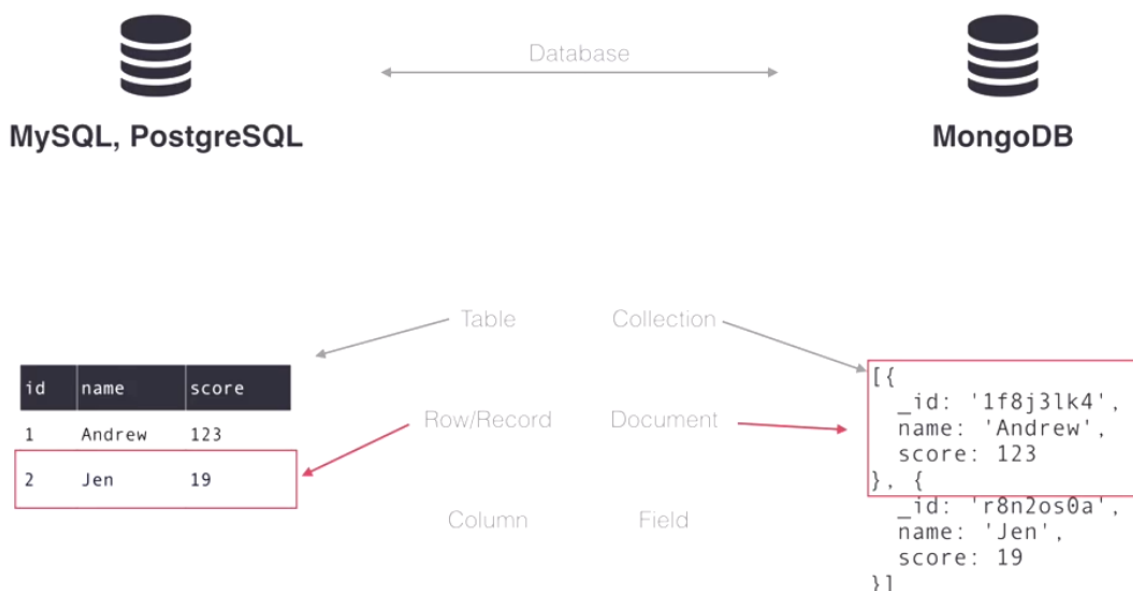
NoSQL – to rodzina baz danych, jakie są alternatywą relacyjnym bazom SQL. Do zbioru baz danych *NoSQL* należy rodzina dokumentnych baz danych. *MongoDB* jest implementacją takiej bazy danych.

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 prostocie, elastyczności przechowywania oraz obsługi danych.

Baza danych *MongoDB* składa się z jednej lub więcej 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.



Rysunek 16

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 kawałek 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, umożliwiającą 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

3.6. Instrumenty zarządzania MongoDB

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

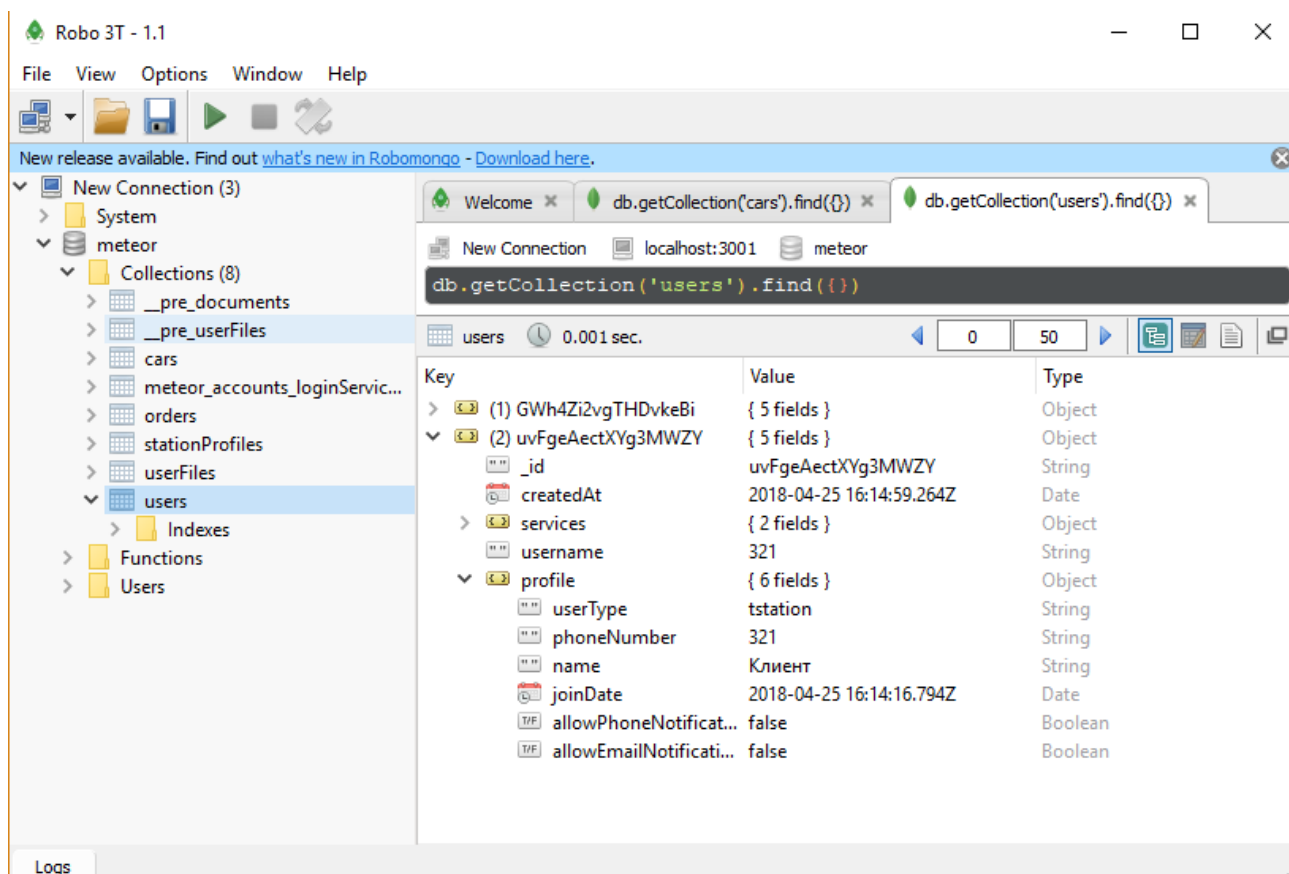
Jednym z tych narzędzi jest *mongo shell*, który jest częścią Meteor CLI. Za pomocą tego w konsoli można odpytywać serwer bazy danych oraz modyfikować dane. Shell można odpalić poleceniem *meteor mongo*.

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.



Rysunek 17

3.7. Meteor.js MongoDB API CRUD

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

Żeby wykonywać CRUD operacje na odpowiedniej kolekcji, musimy posiadać referencje na wcześniej utworzony egzemplarz klasy *Mongo.Collection*, która była zapisywana pod czas tworzenia kolekcji po 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 itd.

Metoda *find* zwraca kursor. To znaczy, że dokumenty nie są zwracane natychmiast. 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 podczas błędu z argumentem *error*.

Jeżeli nie podać funkcji zwrotnej 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 połączyć 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 zastąpione 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.

3.8. Publish & Subscribe

W tej sekcji będzie omówiony sposób kontroli synchronizacji 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 klienta z serwerową. To umożliwia nam korzystanie z danych za pomocą referencji na egzemplarz kolekcji *MiniMongo*, która z kolei zawiera funkcje *find*, *update*, *remove* itd. 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 subskrypcji, 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 subskrypcji. Drugim parametrem jest funkcja, wywoływana za każdym razem, kiedy odbędzie się subskrypcja po stronie klienta. Funkcja *publish* zwraca kursor lub listę kursorów odpowiednich kolekcji.

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

Subskrypcja:

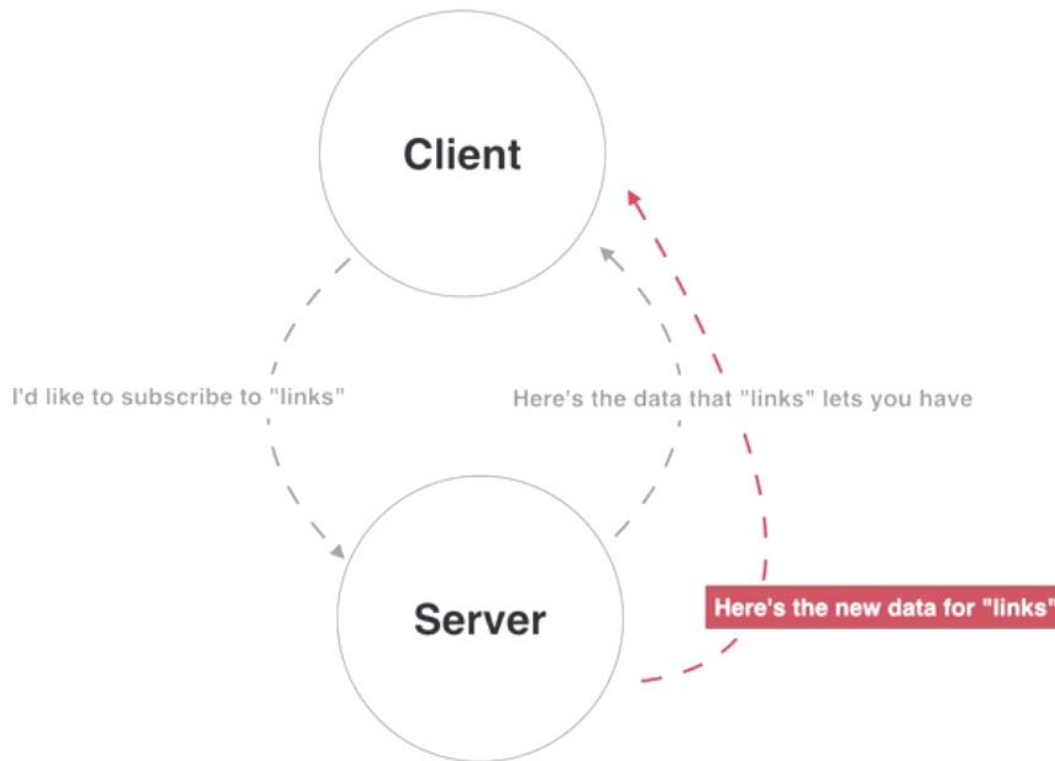
```
Meteor.subscribe(name, [arg1, arg2...], [callbacks])
```

Funkcja jest dostępna wyłącznie po stronie klienta. Pod czas subskrypcji, 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 subskrypcji, gdy *ready()* zezwala sprawdzić czy synchronizacja z serwerem jest ukończona. Pod czas subskrypcji serwer zaczyna proces synchronizacji udostępnionych danych z klientem.

Publications & Subscriptions

Reading Data



Rysunek 18

3.9. Meteor Methods

`Meteor.methods(methods)`

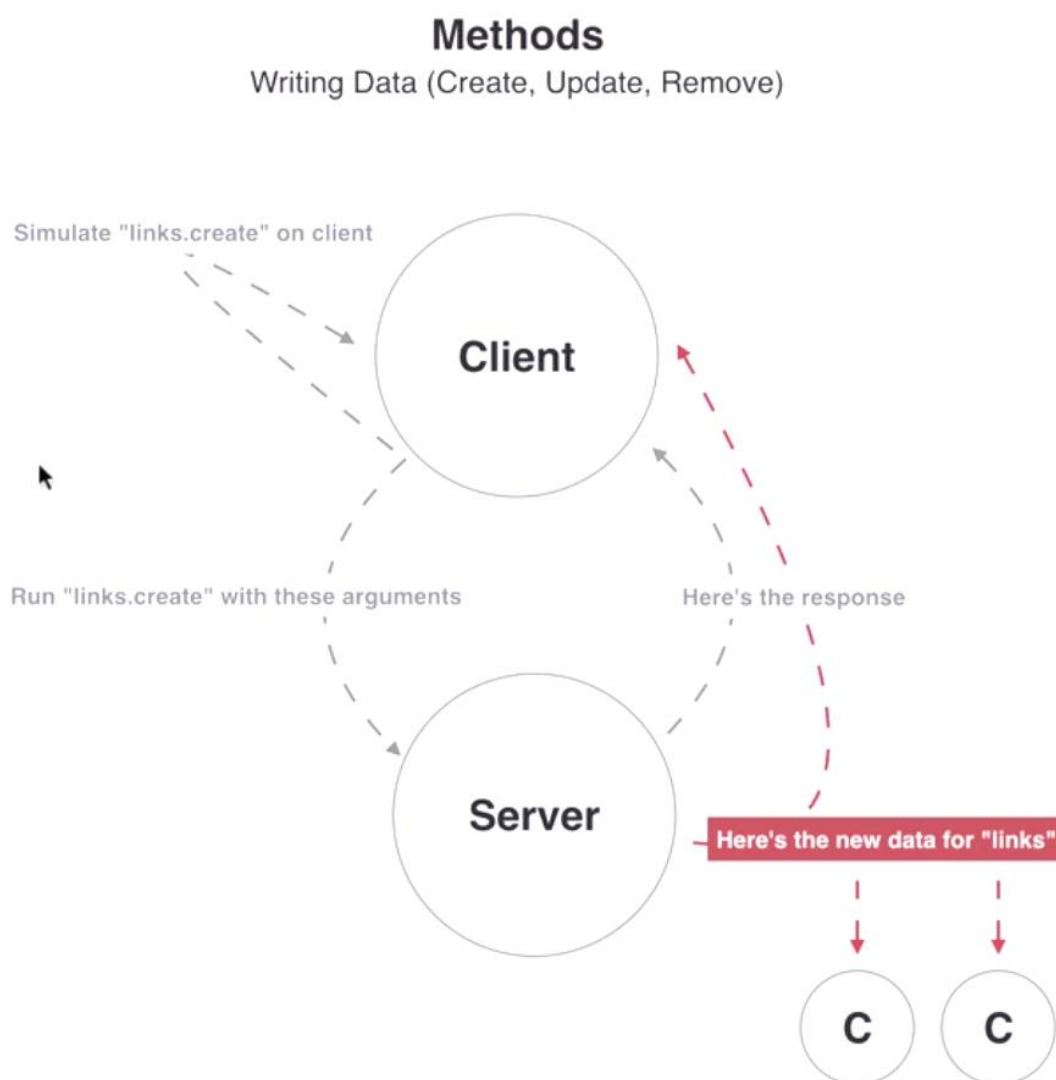
Metody – to są funkcje serwerowe, które można wywołać po stronie klienta. Metody – to sposób na to, żeby bezpiecznie modyfikować dane. O ile metoda jest odpalana na serwerze, jest możliwość zweryfikowania tego, czy działanie wykonuje zalogowany użytkownik oraz czy posiada on uprawnienia do wykonania odpowiedniej akcji. Argumentem jest obiekt, gdzie klucz każdej nowej właściwości – to nazwa metody, a wartością jest funkcja. Takie funkcje mogą przyjmować dowolne parametry. W kontekście każdej funkcji ustawia się słowo kluczowe *this*, które posiada flagi oraz funkcje do zweryfikowania `_id` użytkownika, odblokowania kolejki dla uruchomienia kolejnych metod itd.

`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 renderować 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.



Rysunek 19

3.10. 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.

3.11. 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 obiektu. 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:

```
1  import { check } from 'meteor/check'
2
3  Meteor.methods({
4    addUser(userName, userData) {
5      check(userName, String);
6
7      check(userData, {
8        age: Number,
9        surname: Date,
10       hobby: Match.Maybe([String])
11      });
12    }
13  });
```

Rysunek 20

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:

```

1 Tracker.autorun(function () {
2   if(Meteor.user()) {
3     Meteor.subscribe('LoggedInHomePage');
4   }
5 });

```

Rysunek 21

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:

```

1 Meteor.methods({
2   checkWeather(region) {
3     check(region, String);
4     this.unblock();
5
6     try {
7       const result = HTTP.call('GET', 'http://api.openweathermap.org/data/2.5/forecast?id=524901&APPID={APIKEY}', {
8         params: { region }
9       });
10
11     } catch (e) {
12       registerError(e);
13     }
14     return false;
15   }
16 });

```

Rysunek 22

3.12. System zarządzania użytkownikami systemu

Meteor posiada gotowy system do zarządzania 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 standardowy schemat 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** itd.

Tak przykładowe logowanie do platformy Facebook wygląda następująco:

```
Meteor.loginWithFacebook({
  requestPermissions: ['user_friends', 'public_profile', 'email'],
}, (err) => {
  if (err) {
    // handle error
  } else {
    // successful login!
  }
});
```

Rysunek 23

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

4.1. Opis aplikacji

Aplikacja została napisana głównie w języku JavaScript przy użyciu platformy Meteor oraz zestawu narzędzi 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 skontaktować się z interesującą stacją naprawy, która odpowiada jego kryteriom. 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 priorytetu).
- 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.
- Edytowanie 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 + React / Redux
- MongoDB, MiniMongo, simpl-schema
- Accounts-password, ostrio:files
- ESLint
- AdminLTE

Na poniższych rysunkach prezentują się główne elementy aplikacji.

Na Rysunku 24 widzimy formę do rejestracji oraz logowania nowego użytkownika – właściciela samochodu, dokładnie ten sam element jest używany przy tworzeniu konta właściciela stacji napraw samochodów.

Car owner authorisation and registration

The image shows a web form for car owner registration. At the top, there are two tabs: 'Login' and 'Register'. The 'Register' tab is selected and highlighted in grey. Below the tabs, the form contains three input fields: 'Enter email:' with the value 'tester@gmail.com', 'Enter password:' with three dots indicating a password, and 'Repeat password:' also with three dots. At the bottom of the form is a red button labeled 'Create account'.

Rysunek 24

Na Rysunku 25 widzimy stronę ustawień profilu właściciela samochodu, gdzie użytkownik może zmodyfikować swoje dane, zapoznać się z opłatami, zmienić hasło, numer telefonu oraz adres skrzynki pocztowej. Podobną stronę też posiada część aplikacji, obsługująca właściciela stacji napraw samochodów.

Account settings

Profile and contacts

Your name

Tester aplikacji

Phone*

123321123

E-mail address

tester@gmail.com

Change password

New password

The password must be at least 6 characters long

Change password

Password confirmation

Save

Notifications setup

☒ Enable phone notifications

☐ Enable Email notifications

Save

Service billing

Current plan:

Basic

Valid through:

5 feb 2019

Basic (free):

- normal repair cost estimation mode
- verified hundred
- possibility to make an appointment with one hundred
- chat/email support

PRO 49USD/month:

- all the advantages of a basic fare
- priority evaluation and urgent execution
- there are no restrictions on the number of machines
- individual manager and help

Go PRO

[More information about the possibilities in the PRO Plan](#)

Rysunek 25

Na Rysunku 26 widzimy okno tworzenia nowego zamówienia dla odpowiedniego samochodu. Zamówienia można tworzyć po tym, jak zostanie stworzony / podany samochód użytkownika.

Cars and orders

Mini / 2002 / Cooper

Rear bumper

You have 0 valuations
Geo: Lublin ~95km
Edit

Moderation status:
Not verified

Deadline: 9 June 2018, 11:59

Whole body

You have 0 valuations
Geo: Lublin ~100km
Edit

Moderation status:
Not verified

Deadline: 9 June 2018, 11:59

Add a new order
The application will be created in the context of the current car

BMW / 2004 / 5

perimeter door

You have 0 valuations
Geo: Kiev ~30km
Edit

Moderation status:
Not verified

Deadline: 28 June 2018, 11:59

Add a new order
The application will be created in the context of the current car

Rysunek 27

Na Rysunku 28 widzimy stronę profilu właściciela stacji napraw samochodów. Tu można zobaczyć rankingi oraz status weryfikacji konta. W sekcji Profile Info, można uzupełnić informacje o stacji. Tu można podać pełną nazwę stacji, wyszukać lokalację na mapie, ustawić godziny pracy oraz numer telefonu. Niżej można sformatować opis stacji, który później może być wyświetlony na zewnętrznych serwisach.

52

Rating

Your current rating (in points): 0
Not verified

The rating rises for:

- responses to applications (+1.5 per estimate)
- Verification (+100 and the icon "Verified")
- fullness of profile (from 1 to 100)
- activity on the site (+3 for every 10 ratings)
- customer reviews (+25 for positive customer feedback)
- paid tariff / account (on PRO +250 every month)

Important!

- Rating is one of the most important criteria for a car owner's decision-making price about recording exactly in your service station (after the and location)

Profile Info

Name

SmartSTO

Location

Lavrska St, Kyiv, Ukraine

Phone

111222333

Working hours

8:00 - 16:00

Description

We are the best Tech Station in business.

Rysunek 28

Na Rysunku 29 widzimy komponent do weryfikacji stacji napraw. Zezwala zweryfikować poszczególne dokumenty takie jak: kopia paszportu właściciela, uprawnienia do lokalu, dane firmy. Można łąadować pliki o podanych typach z ograniczeniem rozmiaru 20mb.

Verification

Common status: Not verified

Verification status

Passport.jpg
Not verified

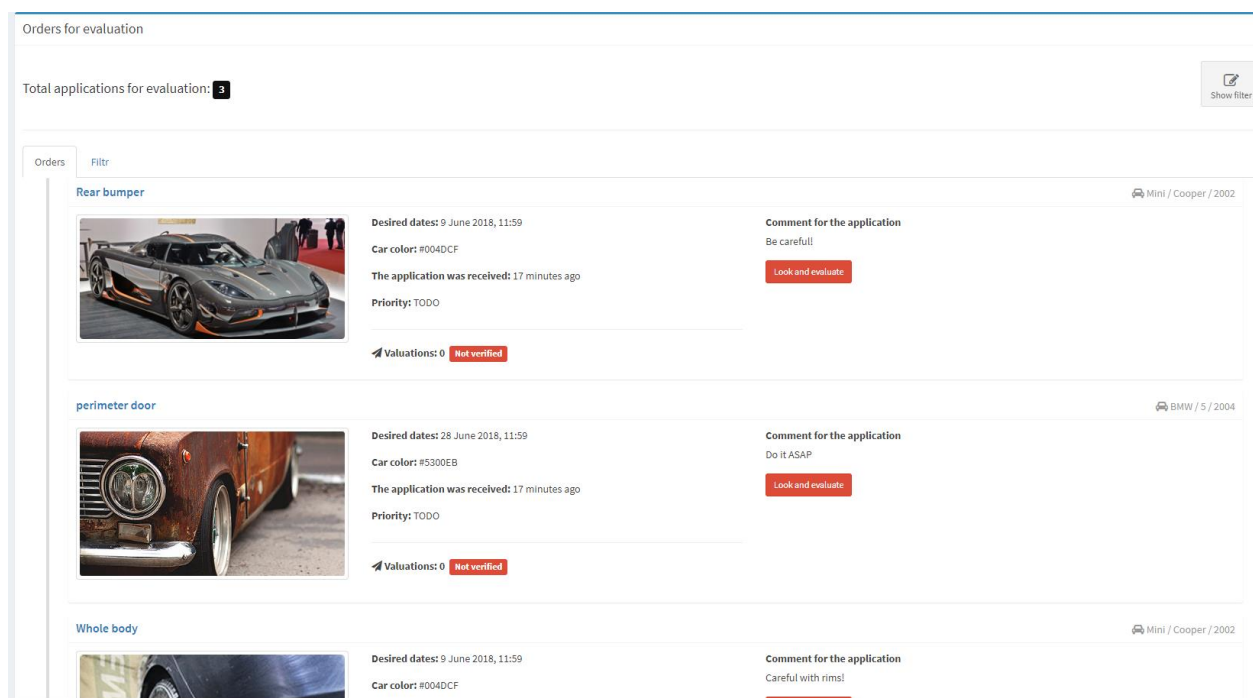
Click to show

Supported formats: png, jpg, jpeg, pdf less than 20 mb

UPLOAD MEDIA

Rysunek 29

Na Rysunku 30 widzimy listę zgłoszeń, które widzi właściciel stacji napraw. Zgłoszenia można filtrować według wielu parametrów. Dalej te zgłoszenia należy wycenić.



Rysunek 30

Na Rysunkach 31 oraz 32 jest pokazane okno wyceny wartości naprawy. Najpierw wyświetlone są dane zgłoszenia: kolor samochodu, model, części korpusu, których dotyczy zgłoszenie itd.

Niżej są wyświetlone pliki media: zdjęcia lub wideo załadowane przez właściciela samochodu.

Także istnieje opcja zostawienia skargi, jeżeli media lub opis nie odpowiada kryteriom stacji.

W sekcji wyceny kosztów naprawy, jest możliwość wyceny kosztu oraz terminu naprawy poszczególnych części samochodu. Po wycenie wszystkich części, można zostawić komentarz oraz wysłać wycenę do użytkownika.

🚗 Mini / Cooper / 2002 - Order evaluation

Order data

Desired dates: 9 June 2018, 11:59

Body parts: Rear bumper

Car color:


Order comment:

The application was received: 21 minutes ago

Be careful!

Priority: High

View photo / video material



1 of 1

Complaint

Rysunek 30

Complaint

Selecting the cause of the complaint:

×

Impossible requirement

×

Distressed photos / videos, numbers:

☒ 1

Comment

It's impossible to do such modification

Send

Costs evaluation

Choose parts:

×

Rear bumper

×

Rear bumper - work cost

200

usd

Rear bumper - terms

5

days

Comment:

Total cost: 200 usd

Total term (days) 5

Cancel

Complete valuation

Rysunek 31

Na Rysunku 33 jest pokazana lista wycen zgłoszenia poprzez poszczególne stacje napraw:

Rysunek 32

4.2. Warstwy aplikacji

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

Po stronie klienta mamy zestaw komponentów React. Jest kilka głównych komponentów, wyświetlanie, których jest kontrolowane poprzez *React Router*. To znaczy, że jak wchodzimy na odpowiedni adres wyświetla się odpowiedni komponent. Taki komponent jest kontenerem dla innych komponentów. Komponenty są stylizowane za pomocą zdefiniowanych stylów SASS.

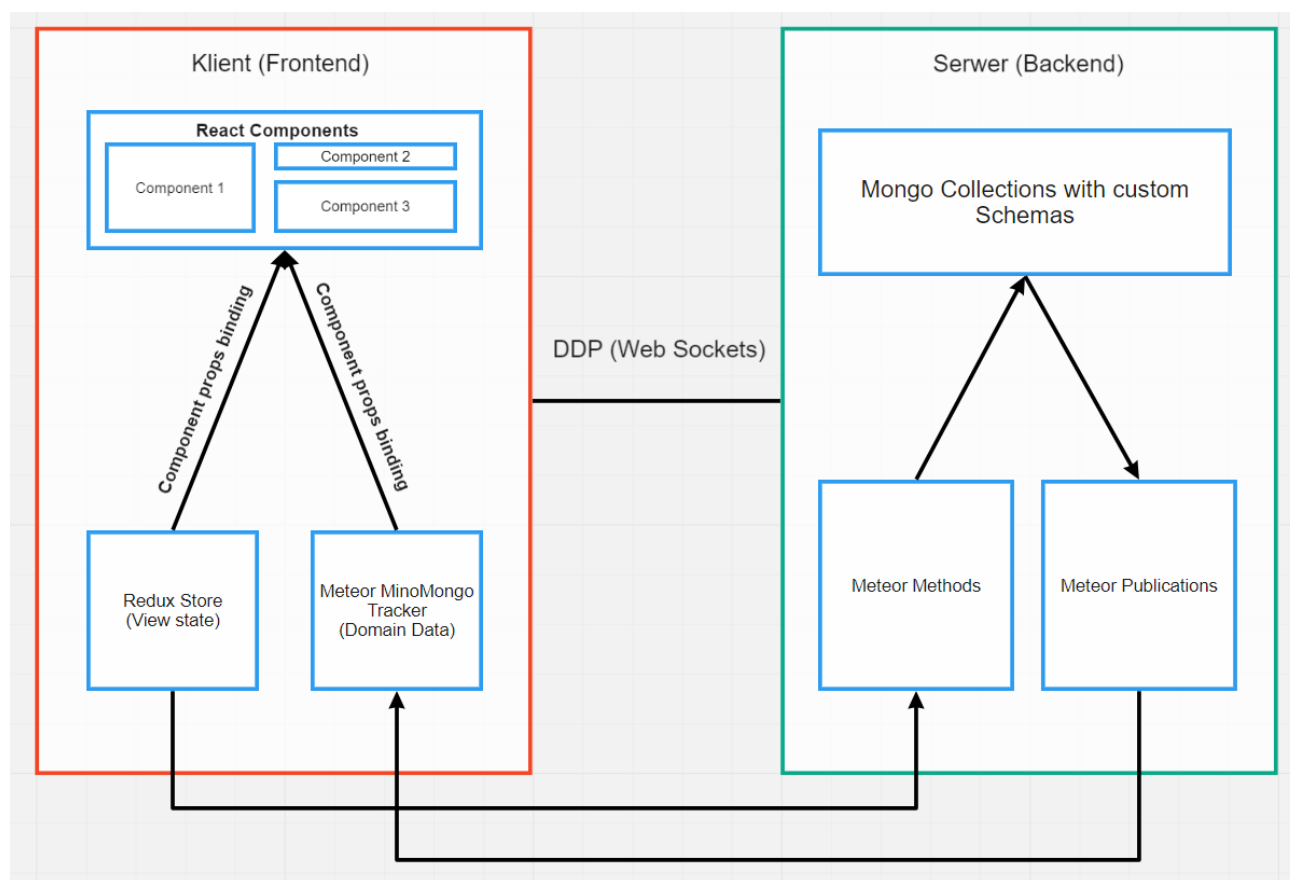
Przepływ danych odbywa się w jedną stronę – do komponentu. Stan komponentu to połączenie dwóch obiektów: *state* oraz *props*. *State* można definiować wewnątrz komponentu. *Props* otrzymujemy zawsze zewnątrz. *Props* mogą być przekazane poprzez atrybuty HTML, przy wywoływaniu tego komponentu lub też mogą zostać powiązane z danymi, które przychodzą z przechowalni *Redux* lub *MiniMongo*. Dane z przechowalni *Redux* – to dane do obsługi widoku: czy jest włączone odpowiednie okno, na której stronie jesteśmy itd. Dane *MiniMongo* – to są wszystkie dane biznesowe, które też są obsługiwane poprzez bazę danych serwerową.

Jeżeli po stronie *Redux* lub *MiniMongo* zmieniają się dane, zbudowany mechanizm zaktualizuje *props* komponentu, co spowoduje odświeżenie komponentu na stronie (ponowne renderowanie). Aktualizacją *MiniMongo* zajmują się mechanizmy Meteor: publikacja oraz subskrypcja. W przypadku

Redux, wszystkie dane są przechowywane w jednym specjalnym schowku po stronie klienta. Dane *Redux* są odcięte od serwera i służą pomocą dla renderowania złożonych komponentów, które mogą mieć kilka stanów.

Użytkownik może wprowadzać zmiany, wypełniać formy itd. Komponent używa akcje, zdefiniowane w *Redux*. W aplikacji podzieliłem akcje na dwa typy: do pracy z metodami Meteor oraz do pracy ze schowkiem *Redux*. Wywołując taką akcję podajemy jej typ oraz przekazujemy potrzebne wartości poprzez parametry. W przypadku akcji działającej z Meteor, wywoła się *Meteor.call* odpowiedniej metody na serwerze. Jeżeli ta akcja ma działać ze schowkiem *Redux*, nią zajmie się *reducer*, po czym zaktualizuje się globalny schowek.

Struktura serwera jest bardzo prosta. Po stronie serwera przechowujemy kolekcje oraz ich schemat. Schemat jest pierwszym etapem kontroli bazy danych i zawiera zestaw oczekiwanych pól oraz ich opis. Po stronie serwera też posiadamy definicje metod oraz publikacji, które są drugim etapem bezpieczeństwa aplikacji.



Rysunek 33

4.3. Szczegóły implementacji

W tym podrozdziale chcę pokazać jak w kodzie wyglądają najbardziej kluczowe mechanizmy przepływu danych.

Na rysunku X można zobaczyć jak do komponentu na raz przekazują się dane *MiniMongo* oraz *Redux*. Najpierw komponent jest owijany w funkcję *withTracker*, która uruchamia się za każdym razem, kiedy zmienia się reaktywne źródło danych. W tym przypadku takim źródłem jest subskrypcja. Dalej, w instrukcji ‘return’ dane są mapowane do *props* komponentu.

Żeby połączyć komponent z *Redux* schowkiem i umożliwić korzystanie z akcji *Redux*, trzeba połączyć komponent za pomocą funkcji *connect*. W tym przypadku komponent używa danych ze schowku i do funkcji *connect* jest przekazywana funkcja *mapStateToProps*, która zajmuje się mapowaniem stanu schowku *Redux* do *props* komponentu, który owijamy w funkcję *connect*.

```
83 = const trackedComponent = withTracker(() => {
84   const carsSub = Meteor.subscribe('getCarsForCurrentUser');
85   const ordersSub = Meteor.subscribe('getOrdersForCurrentUser');
86
87   return {
88     carsSubReady: carsSub.ready(),
89     ordersSubReady: ordersSub.ready(),
90
91     cars: Cars.find().fetch() || [],
92     orders: Orders.find().fetch() || [],
93   };
94 })(carsAndOrdersBox);
95
96 = function mapStateToProps(state) {
97   return {
98     newOrderPopup: state.ui.newOrderPopup,
99     editOrderPopup: state.ui.editOrderPopup,
100    showNewCarPopup: state.ui.showNewCarPopup,
101  };
102 }
103
104 export default connect(mapStateToProps)(trackedComponent);
105
```

Rysunek 34

Na rysunku 36 można zobaczyć jak odbywa się wywoływanie akcji *Redux*. Akcja jest owijana w funkcję *dispatch*, która jest dostępna po połączeniu komponenta z *Redux* poleceniem *connect*.

```

88 |     const cb = (err) => {
89 |       if (err) {
90 |         alert(err);
91 |       }
92 |     };
93 |     if (this.props.orderId === Meteor.userId()) {
94 |       this.props.dispatch(addOrderForCurrentUser(order, cb));
95 |     } else {
96 |       this.props.dispatch(updateOrderForCurrentUser(this.props.orderId, order, cb));
97 |     }
98 |   }

```

Rysunek 35

Na rysunku 37 można zobaczyć jak wyglądają akcje, przekazujące dane metodom Meteor.

```

18 | // UPDATE_CURRENT_USER
19 | export const updateCurrentUser = (updates, cb) => () => {
20 |   Meteor.call('users.updateCurrent', updates, cb);
21 | };
22 |
23 | // UPDATE_CURRENT_USER_NOTIFICATION_SETTINGS
24 | export const updateCurrentUserNotificationSettings = newSettings => () => {
25 |   Meteor.call('users.updateNotificationSettings', newSettings);
26 | };
27 |
28 | // ADD_CAR
29 | export const addCarForCurrentUser = (newCar, cb) => () => {
30 |   Meteor.call('cars.insertForCurrentUser', newCar, cb);
31 | };
32 |

```

Rysunek 36

Na rysunku 38 pokazana została część schematu kolekcji User. Jest możliwość kontroli poszczególnego pola, elementów tablicy lub pól zagnieżdżonych obiektów.

```

40 Schema.User = new SimpleSchema({
41   username: {
42     type: String,
43     optional: true,
44   },
45   emails: {
46     type: Array,
47     optional: true,
48   },
49   'emails.$': {
50     type: Object,
51   },
52   'emails.$.address': {
53     type: String,
54     regex: SimpleSchema.RegEx.Email,
55   },
56   'emails.$.verified': {
57     type: Boolean,
58   },
59   createdAt: {
60     type: Date,
61   },
62   services: {
63     type: Object,
64     optional: true,
65     blackbox: true,
66   },
67   heartbeat: {
68     type: Date,
69     optional: true,
70   },
71   profile: {
72     type: Schema.UserProfile,
73     optional: true,
74   },
75 },
76 });
77
78 Meteor.users.attachSchema(Schema.User);

```

Rysunek 37

Na ostatnim rysunku widzimy część routingu aplikacji. Tu można zobaczyć, że poszczególne komponenty mają przywiązane komponenty. Też został stworzony komponent *PrivateRoutes*, który sprawdza czy użytkownik jest zalogowany.

```

10  const App = () => (
11    <Switch>
12      <Route exact path="/" component={Index} />
13      <Route path="/authPage/:type" component={AuthLayout} />
14      <PrivateRoutes isAuthenticated={!Meteor.userId()}>
15        <Route path="/owner" component={AdminLayout} />
16        <Route path="/tstation" component={AdminLayout} />
17      </PrivateRoutes>
18    </Switch>
19  );
20
21  export default App;

```

Rysunek 38

Zakończenie

Podsumowując swoją pracę, mam nadzieję, że zrealizowałem postawiony na początku cel. W pierwszym rozdziale zrobiłem wprowadzenie w pojęcie oraz architekturę nowoczesnych rozwiązań platformy Web. Także opisałem, czym jest platforma Meteor oraz jej zalety. Rozdział skończyłem opisem interfejsu developerskiego Meteor CLI.

Rozdział drugi poświęciłem opisaniu technik do pracy po stronie przeglądarki. Na początku opisałem nowe funkcje języka JavaScript. Dalej omówiłem biblioteki *React* oraz *Redux*, które zezwalają tworzyć nowoczesne, skomplikowane oraz wydajne interfejsy użytkownika. Pod koniec rozdziału opisałem problemy, z którymi spotyka się developer przy napisaniu stylów aplikacji.

Trzeci rozdział opisuje część serwerową aplikacji na platformie Meteor. Na początku opisane jest środowisko *Node* oraz silnik JavaScript V8. Dalej została opisana warstwa obsługi bazy danych *MongoDB* oraz narzędzia do obsługi. Pod koniec opisałem sposoby transferu oraz synchronizacji danych na platformie Meteor, sposób zabezpieczenia aplikacji oraz główne części Meteor API.

W ostatnim rozdziale zamieściłem opis aplikacji, którą tworzyłem na platformie Meteor. Ten rozdział zawiera także główne warstwy aplikacji oraz ciekawe szczegóły implementacji.

Starałem się, aby w ciągu tej pracy opisać najważniejsze techniki, używane przy napisaniu aplikacji webowych, szczególnie na platformie Meteor. Jestem bardzo wdzięczny za pomoc doktora hab. Ryszarda Kozery, 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] F. Vogelsteller, I. Strack, M. Reyna – *Meteor: Full-Stack Web Application Development*, Packt, 2016.
- [2] K. Simpson – *You Don't Know JS: Up & Going (1st Edition)*, Kindle Edition, 2015.
- [3] K. Simpson – *You Don't Know JS: Scope & Closures (1st Edition)*, Kindle Edition, 2014.
- [4] K. Simpson – *You Don't Know JS: this & Object Prototypes (1st Edition)*, Kindle Edition, 2014.
- [5] K. Simpson – *You Don't Know JS: Async & Performance (1st Edition)*, Kindle Edition, 2015.
- [6] K. Simpson – *You Don't Know JS: ES6 & Beyond (1st Edition)*, Kindle Edition, 2015.
- [7] K. Simpson – *You Don't Know JS: Types & Grammar (1st Edition)*, Kindle Edition, 2015.
- [8] D.Flanagan – *JavaScript: The Definitive Guide (6th Edition)*, O'Reilly, 2011.
- [9] *Meteor Guide*: <https://guide.meteor.com>
- [10] *Meteor Docs*: <https://docs.meteor.com>
- [11] *React.js Docs*: <https://reactjs.org/docs>
- [12] *Redux Docs*: <https://reduxjs.org/basics>
- [13] *Sass Guide*: <https://sass-lang.com/guide>
- [14] *MongoDB Documentation*: <https://docs.mongodb.com>