Wyższa Szkoła Ekonomi i Innowacji
w Lublinie

Wydział Transportu i Informatyki

Kierunek: Informatyka

Specjalność: Inżynieria oprogramowania

Daniel Mariusz Dereziński

22443

Projekt oraz wdrożenie reaktywnego intranetu dla firmy – wykorzystanie technologi Meteor.js, Bootstrap i MongoDB

Praca inżynierska napisana na Wydziale Transportu i Informatyki pod kierunkiem Prof. Grzegorza Marcina Wójcika

Lublin 2015

Spis treści

Spis rysunków

Listings

Rozdział 1

Wstęp

1.1 O aplikacjach sieciowych

opisać z [4]

1.2 Intranet i jego historia

Intranet jest to sieć komputerowa ograniczająca się do komputerów np. w danym przedsiębiorstwie lub innej organizacji, dostępna wyłącznie dla pracowników danej organizacji. Intranet dostarcza szeroki zakres informacji oraz usług z wewnętrznych systemów IT organizacji, które nie są dostępne z publicznego — zewnętrznego — Internetu. Firmowy Intranet dostarcza między innymi centralny punkt wewnętrznej komunikacji, współpracy. Intranet stanowi także pojedynczy punkt dostępu do wewnętrznych jakich zewnętrznych zasobów organizacji. W najprostszej formie intranet budowany jest z wykorzystaniem sieci typu *LAN* (sieć lokalna) oraz *WAN* (rozległa sieć komputerowa) [9].

Coś o historii intranetu....

1.3 Intranet obecnie oraz jego zastosowania

1.4 Cel pracy

W dzisiejszych czasach wiele organizacji / firm wykorzystuje w swojej działalności z jakieś formy intranetu — komunikacja, praca zespołowa. Są to rozwiązania oparte o darmowe systemy CMS, komunikatory, kalendarze, systemy do zarządzania zadaniami. Celem niniejszej pracy było opracowanie oraz wdrożenie systemu intranetowego dla firmy zajmującej się produkcją oprogramowania. Firmy z tej branży pracują w oparciu o

projekty. Jednym z podstawowych celów intranetu jest wymiana wiedzy oraz komunikacja. Zaprojektowana oraz zaprogramowana aplikacja umożliwia dodawanie artykułów, dodawania kategorii, dodawanie projektów, komunikację w obrębie projektów wraz z możliwością dodawania artykułów. Aplikacja pozwala utworzyć profil dla organizacji, zaprosić użytkowników do tak utworzonego profilu w celu podjęcia wspólnej pracy. Możliwe jest także tworzenie kont dla użytkowników nie powiązanych z żadną organizacją.

1.5 Struktura pracy

W rodziale 2 zostanie przedstawione ....

Rozdział 2

Technologia

Rozdział ten przedstawia wykorzystane technologie oraz języki programowania użyte podczas projektowania oraz programowania aplikacji Intranet. Aplikacja powstała z wykorzystaniem *JavaScript*, frameworka aplikacji sieciowych *Meteor.js*, nierelacyjnej bazy danych *MongoDB*, *HTML5*, *CSS3*, *less*[1], frameworka CSS *Bootstrap* oraz gotowego szablon dla panelu administracyjnego *AdminLTE* wykorzystujący Bootstrap.

2.1 JavaScript

Język programowania JavaScript został użyty do zaprogramowania zarówno części serwerowej (*back-end*) jaki i części odpowiedzialnej za interakcje z użytkownikiem (*front-end*) — interfejs użytkownika. Obecne strony WWW a w szczególności aplikacje dostępne przez przeglądarkę (Gmail, Google Docs, Google Maps, Facebook) szeroko korzystają z JavaScript w celu dostarczenie wielofunkcyjnego oraz interaktywnego interfejsu użytkownika. Jednym z powodów wykorzystanie JavaScript była możliwości wykorzystania go po stornie serwera oraz klienta. Najpopularniejszy obecnie sposób tworzenia stron/aplikacji internetowych wyróżnia trzy warstwy — warstwę struktury (HTML), warstwę prezentacji (CSS) oraz warstwę zachowania (JavaScript) [3].

Internet powstał jako zbiór statycznych dokumentów HTML, które były powiązane hiperłączami. Po wzroście popularności oraz rozmiaru sieci, autorom stron przestały wystarczać dostępne narzędzia. Widoczna stała się potrzeba poprawienia interakcji z użytkownikiem. U jej podstaw leżała chęć zmniejszenia ilości połączeń z serwerem w celu realizowania prostych zadań takich jak np. walidacja formularzy. W tym czasie pojawiły się dwie możliwości rozwiązania tego problemu — aplety Javy oraz język *LiveScript*, który został zaproponowany przez firmę Netscape w roku 1995. Został on dołączony do przeglądarki Netscape 2.0 pod nazwą JavaScript [3].

[1] Dynamiczny język arkuszy stylów

Możliwość modyfikacji statycznych elementów stron internetowych bardzo szybko została przyjęta przez rynek. Producenci przeglądarek internetowych szybko dostosowali swoje produktu do obsługi JavaScript'u. Microsoft wyposażył w taką obsługę swoją przeglądarkę Internet Explorer (IE) od wersji 3.0. Była to jednak kopia języka JavaScript — *JScript*, wzbogacona o kilka funkcjonalności przeznaczonych tylko dla IE. W wyniku coraz większych różnić pomiędzy przeglądarkami podjęto próbę standaryzacji różnych implementacji języka. Zadanie to przypadło Europejskiemu Stowarzyszeniu na rzecz Systemów Informatycznych i Komunikacyjnych (ECMA). Tak powstała specyfikacja ECMAScript. Obecnie obowiązuje standard ECMA-262 [6] — jego najpopularniejszą implementacją jest JavaScript.

Wzrost popularności JavaScriptu nastąpił w czasie Pierwszej Wojny Przeglądarek (1996-2001) [3]. Okres ten nazywany jest także okresem bańki internetowej. W tym czasie o udział w rynku walczyli dwaj główni producenci przeglądarek Netscape oraz Microsoft. Firmy kusiły klientów za pomocą coraz to nowych dodatków i ozdóbek wprowadzanych do przeglądarek oraz do stosowanych w nich wersji JavaScriptu. W tym czasie wiele osób wyrobiło sobie negatywną opinie na temat tego języka, który w wyniku wspomnianych działań oraz braku standaryzacji bez przerwy ulegał zmianie. Pisanie programów było koszmarem. Skrypty napisane w oparciu o jedną przeglądarkę nie chciały działać w drugiej. Producenci przeglądarek, skupieni na rozszerzaniu o nowe funkcjonalności, nie dostarczali odpowiednich narzędzi do rozwijania aplikacji [3].

Niespójności pomiędzy przeglądarkami była tylko częścią problemu. Drugą częścią byli sami autorzy stron, którzy upychali w witrynach zbyt wiele zbędnych funkcjonalności. Bardzo często korzystali z wszystkich nowych możliwości dostarczanych przez przeglądarkę, przez co strony były „upiększane" o kwiatki takie jak animacje na pasku stanu, jaskrawe kolory, migające napisy, trzęsące się okna przeglądarek, płatki śniegu, obiekty podążające za kursorem itp., co bardzo często utrudniało korzystanie ze stron. Tego typu nadużycia są także powodem złej reputacji JavaScriptu. Problemy te doprowadziły do traktowania języka JavaScript za niewiele więcej niż zabawkę przeznaczoną dla projektantów interfejsów.

Po zakończeniu Pierwszej Wojny Przeglądarek sposób wytwarzania aplikacji sieciowych uległ zmianie. Zmiany — na lepsze — zostały zapoczątkowane przez kilka procesów [3]:

- Microsoft wygrał wojnę i na okres około pięciu lata wstrzymał się od dodawania nowych funkcjonalności do przeglądarki Internet Explorer oraz do samego JavaScriptu. Dzięki temu producenci innych przeglądarek zyskali czas na dogonienie a czasem nawet przewyższenie możliwości IE.

- Ruch na rzecz standardów sieciowych zyskał przychylność programistów jaki i pro-

ducentów przeglądarek. Standardy chronią programistów od konieczności programowania funkcjonalności dwa (lub więcej) razy na wypadek, gdyby coś nie działało, w którejś z przeglądarek. Co prawda nadal nie istnieje środowisko, które spełniało by wszystkie możliwe standardy.

- technologie i sposoby programowania osiągnęły bardzo dojrzały poziom, na którym można już zajmować się zagadnieniami takim jak użyteczność, dostępność czy też progresywne ulepszanie.

Dzięki nowym, zdrowszym metodologiom programiści zaczęli uczyć się lepszych sposób korzystania z już dostępnych narzędzi. Po wydaniu aplikacji takich jak *Gmail* czy *Google Maps*, które w bardzo szerokim stopniu korzystają z programowania po stronie klienta, stało się oczywiste, że JavaScript to dojrzały, jedyny w swoim rodzaju oraz potężny prototypowy język obiektowy [3]. Dobrym przykładem jego ponownego odkrycia jest szeroka akceptacja funkcjonalności dostarczanej przez obiekt `XMLHttpRequest`, który dawniej był obsługiwany tylko przez przeglądarkę Internet Explorer. Obiekt ten jednak został zaimplementowany przez wiele przeglądarek. `XMLHttpRequest` umożliwia wykonywanie żądań HTTP i pobieranie zawartości serwera w celu aktualizacji pewnych części strony bez konieczności przeładowanie jej całej. Dzięki temu narodził się nowy gatunek aplikacji sieciowych, które przypominają samodzielne aplikacje desktopowe. Takie aplikacje określamy mianem aplikacji *AJAX*.

JavaScript po rewolucji spowodowanej rozwojem technologii AJAX zaczął być używany przez programistów do tworzenia rzeczywistych i ważnych aplikacji. Obecnie mamy wiele aplikacji sieciowych JavaScript począwszy od Twittera, przez Facebook, aż po GitHub [1]. Jedną z ciekawszych cech JavaScriptu jest to, że musi on działać wewnątrz *środowiska*. Najpopularniejszym środowiskiem jest przeglądarka internetowa. Istnieją także inne możliwości — JavaScript może działać po stornie serwera, na pulpicie lub wewnątrz tzw. *rich media*[2].

Od chwili wydania przeglądarki Google Chrome w roku 2008 ciągle i w bardzo szybkim tempie poprawia się wydajność działania JavaScript, co jest wynikiem silnej konkurencji pomiędzy producentami poszczególnych przeglądarek internetowych. Wydajność nowoczesnych maszyn wirtualnych JavaScript zmienia rodzaje aplikacji tworzonych dla sieci. Fantastycznym przykładem jest jslinux[3] — utworzony w JavaScript emulator pozwalający na wczytanie jądra systemu Linux, pracę w powłoce oraz kompilację programów w C.

[2]Aplikacje *rich media* — Flash, Flex — tworzy się przy użyciu ActionScriptu, który jest oparty o ECMAScript

[3]`http://bellard.org/jslinux/`

2.1.1 Przegląd cech JavaScript

JavaScript jest imperatywnym oraz strukturalnym językiem programowania. Wspiera większość składni programowania strukturalnego z języka C, np. pętle `while`, instrukcję wyboru `switch`, pętle `do while` oraz wiele innych. Wyjątkiem jest zasięg zmiennych. W JavaScript zasięg zmiennych z wykorzystaniem słowa kluczowego `var` to zasięg do całego ciała funkcji. Nowy standard ECMAScript 2015 (ES6) wprowadza zakres zmiennych co do bloku instrukcji z wykorzystaniem słowa kluczowego `let` — co oznacza że język ten ma obecnie zakres zmiennych co do ciała funkcji jak i do bloku instrukcji. Podobnie jak C JavaScript rozróżnia wyrażenia oraz instrukcji [10].

Jak większość języków skryptowych także JavaScript jest językiem dynamicznie typowanym. Typy powiązane są z wartościami a nie ze zmiennymi. Na przykład do zmiennej x może być przypisana wartość typu liczbowego a następnie możemy do takiej zmiennej przypisać na przykład łańcuch znaków [10].

JavaScript jest prawie w całości obiektowy. Obiekty w JavaScript są to tablice asocjacyjne rozszerzone o prototypy. Nazwy właściwości obiektu to ciągi znaków. Obiekty wspierają dwie różnoznaczne składnie — z kropką `obj.x = 10` oraz z nawiasami kwadratowymi `obj["x"] = 10`. Właściwości oraz ich wartości mogą być dodawane, zmienianie oraz usuwane w każdej chwili działania programu. Większość właściwości obiektu (oraz te dziedziczone w łańcuch prototypów) mogą być wymienione z wykorzystaniem pętli `for ... in`. JavaScript ma dość skromny wachlarz obiektów wbudowanych, między innymi `Function` (funkcje) oraz obiekty daty — `Date`. JavaScript oferuje również wbudowaną funkcje `eval`, która może wykonywać instrukcje dostarczone w postaci ciągów znaków podczas działania programu [10].

JavaScript jest także językiem funkcyjnym. Funkcję są typu pierwszej klasy. Oznacza to, że JavaScript wspiera przekazywanie funkcji jako argumentów do innych funkcji, zwracania ich jako wartości innych funkcji, przypisywania ich do zmiennych oraz zapisywanie ich w strukturach danych. JavaScript wspiera również funkcje anonimowe [7]. Funkcje w JavaScript jako takie posiadają właściwości oraz metody takie jak `.call()` i `bind`. Funkcje zagnieżdżone to funkcje zdefiniowane wewnątrz innych funkcji. Funkcja taka jest każdorazowo tworzona podczas wywołania funkcji zewnętrznej, w której została ona zdefiniowana. Dodatkowo każda tworzona funkcja tworzy *domknięcie*: zasięg zmiennych funkcji zewnętrznej, włączając w to zmienne lokalne oraz wartości argumentów, stają się częścią wewnętrznego stanu każdego wewnętrznego obiektu funkcji, nawet po zakończeniu wykonywania zewnętrznej funkcji [10].

JavaScript w procesie dziedziczenia wykorzystuje prototypy, w odróżnieniu od innych języków zorientowanych obiektowo wykorzystujących klasy. W JavaScript wykorzystując prototypy można za symulować wiele cech dziedziczenia opartego na klasach [10]. Nowe

wydanie ES6 wprowadza do JavaScript składnie umożliwiającą definiowanie klas.

Funkcje obok swojej typowej roli w JavaScript mogą także tworzyć obiekty. Poprzedzając wywołanie funkcji instrukcją wbudowaną `new` zostanie utworzona instancja prototypu dziedzicząca właściwości oraz metody z konstruktora (włączając w to właściwości z prototypu obiektu wbudowanego `Object`). ECMAScript 5 oferuje metodę `Object.create` pozwalającą na jawne tworzenie instancji bez automatycznego dziedziczenia z prototypu `Object`. Właściwość `prototype` konstruktora określa jaki obiekt zostanie użyty jako wewnętrzny prototyp nowo utworzonego obiektu. Nowe metody mogą zostać dodane poprzez modyfikowanie prototypu funkcji użytej jako konstruktor. Wbudowane konstruktory takie jak `Array` lub `Object`, także posiadają prototyp, który może być modyfikowany. Modyfikowanie prototypy `Object` jest uznawane za złą praktykę ponieważ prawie wszystkie obiekty w JavaScript dziedziczą metody oraz właściwości z prototypu obiektu `Object` [10].

W odróżnieniu od wielu języków zorientowanych obiektowo w JavaScript nie ma różnicy pomiędzy definicją funkcji oraz definicją metody. Różnica pojawia się podczas wywołania funkcji oraz metody. Kiedy funkcja wywoływana jest jako metoda obiektu, słowo kluczowe `this` wewnątrz ciała funkcji jest powiązane z obiektem na rzecz, którego dana metoda została wywołana [10].

JavaScript posiada wbudowane implementacje, oparte na funkcjach, wzorców takich jak *cecha* oraz *domieszka*. Jawna, oparta na funkcjach, delegacja nie wspiera kompozycji, natomiast nie jawna delegacja ujawnia się za każdym razem gdy przechodzony jest łańcuch prototypów np. w celu znalezienia metody, która nie należy bezpośrednio do obiektu. Gdy metoda zostanie odnaleziona wywoływana jest w kontekście danego obiektu. Dlatego dziedziczenie w JavaScript jest realizowane przez delegacje, która polega na przypisaniu właściwości do prototypu konstruktora funkcji [10].

Typowo JavaScript uruchamiany jest w jakimś środowisku np. w środowisku przeglądarki internetowej, które dostarcza obiekty oraz metody, przez które uruchamiany skrypt może oddziaływać na środowisko np. obiekt DOM strony internetowej.

JavaScript przetwarza wiadomości z kolejki — po jednej na raz. Podczas ładowania nowej wiadomości, JavaScript wywołuje funkcję powiązaną z daną wiadomości tworząc ramkę stosu wywołania (są to argumenty funkcji oraz zmienne lokalne). Stos wywołania zmniejsza się oraz powiększa zgodnie z potrzebami wywołanej funkcji. Nazwane jest to pętlą zdarzeń, opisywaną także jako „działaj do ukończenia”, ponieważ każda wiadomość jest całkowicie przetwarzana zanim przejdziemy do kolejnej wiadomości. Jednakże wbudowany model konkurencji nadaje pętli zdarzeń charakter nie blokujący. Operacje wejścia/wyjścia realizowane są z użyciem zdarzeń oraz wywołań zwrotnych (funkcji zwrotnych). Oznacza to, że JavaScript może przetworzyć kliknięcie myszy podczas czekania na dane z zapytania

do bazy danych [10]

Do funkcji może zostać przekazana nie określona ilość argumentów. Funkcja ma do nich dostęp poprzez parametry oraz także przez lokalny obiekt `arguments` [10].

Jak wiele języków skryptowych, tablice jak i obiekty mogą zostać utworzone przez zwięzłą składnie. Te literały stanowią także podstawę formatu *JSON*[4] [10].

JavaScript wspiera również wyrażenia regularne w sposób podobny do *Perl*, z zwięzła oraz bogatą składnią do manipulowania tekstem, znacznie bardziej zaawansowaną niż wbudowane funkcje do obróbki łańcuchów znaków.

2.2 Meteor.js

Meteor.js (Meteor, MeteorJS) jest to otwarto źródłowy framework aplikacji sieciowych napisany w JavaScript z wykorzystaniem *Node.js*. Meteor pozwala na szybkie prototypowanie oraz tworzenie między platformowego kodu (aplikacje sieciowe, Android, iOS). Framework wykorzystuje *MongoDB*, używa protokołu *DDP*[5] oraz wzorca *publikacji i subskrypcji* do automatycznej propagacji zmian w danych do klientów bez potrzeby pisanie kodu synchronizującego taką propagację. Po stronie klienta, Meteor wykorzystuje *jQuery*. Meteor jest rozwijany prze *Meteor Development Group*. Meteor po raz pierwszy został publicznie pokazany w grudniu 2011 pod nazwą *Skybreak* [12].

Meteor jest niepokojącą (w dobrym znaczeniu) technologią. Umożliwia budowanie aplikacji nowego typu, które są szybsze oraz prostsze w tworzeniu. Wykorzystuje nowoczesne techniki takie jak reaktywność po stronie klienta oraz serwera (*Full Stack Reactivity*), kompensacja opóźnień (*Latency Compensation*) oraz *data on the wire* - Meteor nie wysyła danych w postaci HTML, serwer wysyła dane i pozwala klientowi je renderować [4].

Meteor pozostaje wierny następującym zasadom [2]:

- *Data on the Wire* — Meteor nie wysyła HTML przez sieć. Serwer wysyła dane i pozwala klientowi je renderować.
- *Jeden język* — Meteor pozwala pisać stronę klienta jak i serwera w języku JavaScript.
- *Wszechobecna baza danych* — Meteor pozwala na użycie tych samych metod dostępu do danych po stronie klienta oraz serwera.
- *Kompensacja opóźnień* — Po stronie klienta Meteor wstępnie pobiera dane oraz symuluje modele aby wykonywane metody serwerowe zwracały wynik natychmiast.

[4]JavaScript Object Notation - lekki format wymiany danych komputerowych, jest to format tekstowy. Opisany w RFC 4627

[5]Distributed Data Protocol - protokół klient–serwer dla zapytań oraz aktualizacji bazy danych po stronie serwera oraz synchronizacji tych aktualizacji wśród innych klientów

- *Korzysta z ekosystemu* — Meteor jest projektem otwarto źródłowym (open source) oraz integruje się z istniejącymi otwarto źródłowymi narzędziami oraz frameworkami.

- *Prostota to produktywność* — Najlepszym sposobem aby coś wyglądało na proste to zrobić to tak aby było proste. Główne funkcjonalności Meteora mają czyste, klasycznie piękne API.

Koncepcja „danych w kablu“ *Data On the Wire* jest bardzo prosta i powstała z zagnieżdżonego wzorca model – widok – kontroler — *MVC*. Zamiast przetwarzania przez serwer każdego żądania, renderowania treści i wysyłania HTML do klienta, serwer wysyła same dane i pozowana klientowi zdecydować co z nimi zrobić [4].

To rozwiązanie w Meteorze zostało zaimplementowane z użyciem *Distributed Data Protocol* (DDP) — protokołu klient – serwer dla zapytań oraz aktualizacji danych po stronie serwera w bazie danych oraz synchronizacji tych zmian pośród innych klientów. Protokół ten oparty jest o składnie JSON oraz komunikuje się z serwerem w sposób podobny do protokołu *REST*[6]. Dodawanie, usuwanie oraz aktualizację są rozsyłane przez sieć oraz obsługiwane przez docelowe urządzenia, usługi oraz klientów. DDP używa *WebSockets*[7] zamiast HTTP, dane pomogą być wysyłane kiedy tylko dane ulegną modyfikacji [4].

Najważniejszą zaletą protokołu DDP to sposób komunikacji. Nie ma znaczenie jaki system wysyła lub odbiera dane może to być serwer, klient, usługa sieciowa - wszystkie one używają tego samego protokołu. Oznacza to, że żaden z systemów nie wiem czy inne systemy to serwery czy klienci. Z wyjątkiem przeglądarki, każdy z systemów może być serwerem lub pełnić role klienta. Cały ruch generowany przez protokół wygląda tak samo i jest traktowany w ten sam sposób. Tradycyjna koncepcja jednego serwera dla jednego klienta staje się przestarzała. Możliwe jest połączenie wielu serwerów, które pełnią określone role. Klient może być połączony z wieloma serwerami, z którymi może pracować w różny sposób [4].

W skład frameworka wchodzi także ciekawa technologia — *mini baza danych*. Jest to „leka” wersja normalnej bazy danych rezydującej w pamięci po stronie klienta. Klient zamiast wysyłać żądania do serwera może zmieniać dane bezpośrednio w mini bazie danych, które znajduje się po stronie klienta. Po aktualizacji danych w mini bazie danych ta automatycznie synchronizuje się z serwerem, który posiada właściwą bazę danych, wy-

[6]Representational State Transfer - (zmiana stanu poprzez reprezentacje) — styl architektury oprogramowania powstały z doświadczeń podczas opracowywania specyfikacji protokołu HTTP dla systemów rozproszonych. REST wykorzystuje między innymi jednorodny interfejs, bezstanową komunikację, zasoby, reprezentacje, hipermedia [14]

[7]Technologia zapewniająca dwukierunkowy kanał komunikacji za pośrednictwem jednego gniazda TCP. Stworzona głównie jako kanał komunikacji pomiędzy przeglądarką internetową a serwerem internetowym. Może być także stosowana w innych aplikacjach typu klient lub serwer. [15]

korzystując protokół DDP. Meteor w roli mini bazy danych wykorzystuje *Minimongo*, właściwa baza po stronie serwera to *MongoDB* [4].

Aktualizacja danych po stronie klienta, najpierw jest realizowana w instancji Minimongo. Klient pozostawia synchronizację zmian Minimongo (z wykorzystaniem protokołu DDP). Jeżeli serwer zaakceptuje zmiany, rozsyła je do wszystkich połączonych klientów, włączając w to także tego, który dokonał zmian. Natomiast jeżeli serwer odrzuci zmiany, bądź pojawi się nowszy zestaw danych od innego klienta, Minimongo na kliencie zostanie skorygowane, co spowoduje aktualizację wszystkich elementów interfejsu użytkownika powiązanych z tymi danymi. Rozwiązanie to w połączeniu z asynchronicznością (realizowaną z wykorzystaniem DDP) jest przełomowe. Oznacza to, że klient nie musi czekać na odpowiedź z serwera. Klient aktualizuje interfejs użytkownika w oparciu o dane zawarte w instancji Minimongo. W przypadku gdy aktualizacja danych została odrzucona przez serwer lub inne zmiany dotarły z serwera, klient zostanie zaktualizowany zaraz po otrzymaniu takiej informacji z serwera.

W przypadku powolnego połączenia z internetem lub jago całkowitego jego braku Meteor kompensuje to poprzez wysyłanie danych do Minimongo oraz natychmiastowymi zmianami w interfejsie użytkownika. W normalnym środowisku klient – serwer żadna aktualizacja nie została by wykonana, interfejs użytkownika pokazywał by stan ładowania podczas kiedy klient czekał by na odpowiedź z serwera. Wszystkie dokonane zmiany miałby swoje odzwierciedlenie w interfejsie użytkownika w oparciu o Minimongo. Kiedy połączenie z Internetem zostało by przywrócone, wszystkie zakolejkowane zmiany zostaną wysłane na serwer, serwer natomiast prześle autoryzowane zmiany do klienta. Meteor pozwala klientom „brać informacje na wiarę". Jeżeli napotkamy problem to dane otrzymane z serwera naprawią nieścisłości. Jednak przez większość czasu zmiany dokonywane na kliencie są natychmiast wysłane na serwer oraz przez niego rozgłaszane do innych klientów. Aby osiągnąć takie zachowanie należy zdefiniować nową kolekcję w poniższy sposób:

Listing 2.1: Definicja kolekcji

```
1  Articles = new Mongo.Collection("Articles")
```

Ta jedna linijka deklaruje kolekcję `Articles`, której swoją wersję będą posiadały zarówno klient jak i serwer — lecz traktują ją w odmienny sposób. Klient subskrybuje zmiany ogłaszane przez serwer i aktualizuje odpowiednio swoją kolekcję. Serwer natomiast publikuje zmiany oraz nasłuchuje zmian z klientów aktualizując swoją kolekcję.

Jedną z najważniejszych cech Meteora jest reaktywność. Po stornie klienta, Meteor oferuje bibliotekę *Blaze*, która używa szablonów HTML oraz *helper'ów*[8] JavaScript do wykrywania zmian oraz renderowania danych. Kiedy dane zostaną zmienione, helpery same ponowienie się uruchamiają zmieniając, usuwając lub dodając właściwe elementy

[8]Pomocniczy kod JavaScript głównie zwracający wartości do szablonu HTML

interfejsu użytkownika na podstawie struktury zapisanej w szablonach. Funkcje, które same ponownie się uruchamiają zaliczane są do *reaktywnych obliczeń* [4].

Meteor oferuje reaktywne obliczenia zarówno po stronie serwera jak i klienta, bez potrzeby odwoływania się do interfejsu użytkownika. Wchodząca w skład framework'a biblioteka *Tracker* oraz jej helpery także wykrywają zmiany w danych oraz potrafią się same ponownie uruchomić. Ponieważ JavaScript używane jest po stronie serwera oraz klienta, bibliotekę Tracker można używać po obu stronach. Ponieważ używamy tego samego języka po stronie klienta (oraz możemy używać tego samego kod) jak i serwera nazywamy to *full stack reactivity* - reaktywność po stronie klienta jak i serwera.

Ponowne uruchamianie funkcji gdy dane ulegną zmianie ma bardzo dużą zaletę dla programisty. Kod pisany jest deklaratywnie, Meteor sam zajmuję się reaktywnością. Programista określa jak dane mają byś wyświetlane a Meteor sam zajmuję się zmianami w danych. Ten deklaratywny sposób pisania, zazwyczaj osiągany jest z wykorzystaniem szablonów. Szablony działają w oparciu o wiązanie danych z widokiem[9] — są to współdzielone dane, które w zależności od ich zmian będą przedstawiane w różny sposób [4].

2.3 MongoDB

Meteor po stronie serwera używa MongoDB, po stronie klienta używana jest Minimongo — wersja MongoDB. Meteor może używać innych baz NoSQL[10] lub baz zorientowanych na dokumenty. Dzięki użyciu MongoDB programy stają się prostsze, łatwiejsze w tworzeniu. MongoDB doskonale sprawdza się jako szybki oraz lekki magazyn danych.

Tradycyjnie większość danych przechowywana jest z użyciem modelu relacyjnego z wykorzystanie relacyjnych baz danych. Relacyjny model wraz ze wszystkimi powiązanymi z nim zasadami, relacjami, logiką oraz składaniom jest integralną oraz nieocenioną częścią współczesnej informatyki. Sztywna struktura baz relacyjnych, z dokładnymi wymaganiami co do każdego rekordu, relacje oraz asocjacje umożliwiają szybkie wyszukiwanie, skalowalność oraz dostarczają potencjał do głębszej analizy danych.

Taka dokładność nie jest zawsze potrzebna. Dla prostych aplikacji, pełno wymiarowa relacyjna baza była by nad wymiarowa. W niektórych sytuacjach bardzie efektywne jest wykorzystanie elastycznego mechanizmu przechowywania danych, który umożliwia łatwe rozszerzanie schematu danych bez potrzeby znaczących zmian w tworzonym programie.

Jeżeli byśmy chcieli dodać nową właściwość do obiektu `Article` o wiele łatwiejsze było by dodanie takiej właściwości do obiektu i zrzucenie tego na bazę danych niż przepisanie kodu dotyczącego baz danych, dodanie kolumny, aktualizacja wszystkich zapytań

[9] View data bindings
[10] Not Only SQL

SQL oraz upewnienie się, że wszystkie poprzednie wpisy mają nową właściwość.

W takich sytuacjach bardzo dobrze sprawdzają się bazy danych zorientowane na dokument. W tym typie baz danych dane zapisywane są w postaci dokumentów złożonych z par klucz – wartość. Tak przechowywane dokumenty nie muszą mieć z góry ustalonej struktury. Dokumenty mogą przyjmować różne struktury, w kluczach mogą być przechowywane różne wartości — dla bazy danych nie jest to żadne problem. Jednak każdy dokument musi posiadać unikalny klucz za pomocą, którego może on zostać pobrany z bazy danych [4]. Dla przykładu, jeden dokument może mieć bardzo prostą postać:

```
{name: phone_number}
```

Następny dokument może przybrać formę bardziej skomplikowaną (wewnątrz tej samej bazy danych oraz kolekcji) — składający się z zagnieżdżonych list, obiektów oraz innych elementów [4]:

```
{ people: [
  {
    firstname:"STEVE",
    lastname:"Scuba",
    phones :[
      {type:cell, number:8888675309},
      {type:home, number:8005322002}
    ]
  },
    {
      firstname:...
...
  }]
}
```

Dokumenty mogą przybierać dowolną strukturę, o ile każdy z nich posiada unikalny klucz, umożliwiający ich pobranie. Brak struktury wpływa negatywnie na wydajność zapytań, sortowania oraz aktualizowania dokumentów. Podczas tworzenia aplikacji możemy zidentyfikować najczęściej pojawiające się zapytania i zmodyfikować tak strukturę dokumentów by baza była w niektórych zastosowaniach znacznie wydajniejsza od rozwiązań relacyjnych. Dodatkową zaletą wykorzystania takich baz jest szybkość oraz łatwość pisania aplikacji [4]. Duża elastyczność baz zorientowanych na dokumenty ułatwia szybkie i łatwe zmiany a dostarczane wraz z Meteorem biblioteki pozwalają nie martwić się o połączenie oraz strukturę bazy danych. Jedynie to co jest potrzebne to wysoko poziomowe zrozumienie jak wyszukać, dodać oraz zmodyfikować dokumenty, resztą zajmie się sam Meteor.

MongoDB to otwarto źródłowa baz danych typu NoSQL. Jest to baz danych zorientowana na dokumenty z zaawansowanymi funkcjonalnościami takim jak indeksy, replikacje, zapytania ad-hoc, agregacja danych, zapytania do zagnieżdżonych dokumentów, równoważenie obciążenia, możliwość zapisywania plików. Cechuję ją duża skalowalność, wydajność, brak określonej struktury. Została napisana w języku C++. Dane zapisywane są jako dokumenty w stylu JSON. Taki sposób umożliwia aplikacją bardziej naturalne ich przetwarzanie, przy zachowaniu możliwości tworzenia hierarchii oraz indeksowania. Wewnętrzny język do definiowania zapytań oraz funkcji agregujących to JavaScript wykonywany bezpośrednio przez serwer MongoDB. Dokumenty zapisywane są w MongoDB w logicznych grupach nazywanych kolekcjami.

MongoDB posiada ograniczone wsparcie dla transakcji — zasięg jest ograniczony do jednego dokumentu. Zmiany w tym dokumencie mogą być bardzo skomplikowane. Z tego powodu część wdrożeń ogranicza zastosowanie MongoDB do niekrytycznych danych informacyjnych, powierzając obsługę krytycznych operacji relacyjnym odpowiednikom. MongoDB wspiera w niewielkim stopniu kodowanie UTF-8, co jest problemem w przechowywaniu tekstu w języku innym niż angielski. Do sortowania łańcuchów znaków używana jest funkcja *memcmp*, która nie obsługuje poprawnie danych w UTF-8 w różnych ustawieniach regionalnych. Wprowadzenie zmian jest planowane, niestety wiąże się to – według twórców – z wprowadzeniem szeregu poważnych zmian. Z tego powodu termin wydania wersji z poprawną obsługą UTF-8 jest nieokreślony [13].

2.4 Warstwa prezentacji

2.4.1 HTML5

Do opisu struktury szablonów używanych przez Meteora użyto *HTML5*. Wersja HTML — HTML5 jest rozwinięciem języka HTML4 oraz jego XML-owej odmiany XHTML 1. Został on opracowany w ramach pracy grupy roboczej WHATWG[11] oraz W3C. Prace nad specyfikacją zostały ukończone w 2014 roku a 28 października 2014 roku został oficjalnie wydany jako rekomendacja W3C. HTML5 poza dodaniem nowych elementów, usprawniających tworzenie serwisów oraz aplikacji internetowych, doprecyzowuje niejasności w specyfikacji HTML4, które przede wszystkim dotyczą sposobu obsługi błędów. Niejasności co do sposobu, w jaki przeglądarki powinny obsługiwać błędy w kodzie HTML są jedną z podstawowych przyczyn, dla której wiele serwisów internetowych, napisanych z naruszeniem specyfikacji, w różnych przeglądarkach działa w inny sposób – w niektórych działając, w innych nie. Dzięki poprawnej obsłudze błędów przez przeglądarki, zły element

[11] Web Hypertext Application Technology Working Group

będzie działać w każdej przeglądarce bądź w żadnej.

HTML5 stawia także na semantykę. Element `<div>` tracji na znaczeniu na rzecz elementów `<header>`, `<main>`,`<article>`, `<aside>`, `<footer>`, `<nav>`. Dodane zostały także następujące elementy `<canvas>`, `<figure>`, `<details>`, `<summary>`. Element `<input>` zyskał dodatkowe typy: *tel, search, url, email, datetime, date, month, week, time, datetime-local, number, range, color*. Dodano nowe atrybuty do elementów formularza: *autofocus, required, autocomplete, min, max, multiple, pattern, step*. HTML5 ma możliwość osadzania *MathML* i *SVG* bezpośrednio w dokumencie [8].

Oprócz dodania nowych elementów oraz atrybutów W3C zaproponowało większy nacisk na modułowość, określając specyficzne cechy oraz rozwój ich jako oddzielnych specyfikacji. Niektóre technologie, które pierwotnie zostały zdefiniowane w samym HTML5 są obecnie określone w odrębnych specyfikacjach. Do takich technologi należą miedzy innymi: *WebGL*, *Web Sockets*, rysowanie 2D z nowym elementem `<canvas>`, geolokalizacja, baza danych SQL, komunikacja między stronami (np. można wysyłać informację do strony znajdującej się w ramce), *microdata* (przechowywanie danych w atrybutach – prefiks: data-), API do odtwarzania wideo i audio, API dla aplikacji offline. DOM[12] zyskał dodatkowe metody: `getElementsByClassName`, `activeElement`, `hasFocus`,`getSelection`, `classList` `relList` [8].

Przeglądarki obsługujące HTML5 zostaną dostosowane do obsługi błędów w składni. HTML5 został zaprojektowany tak, by starsze przeglądarki bez problemu mogły ignorować nowe konstrukcje. W przeciwieństwie do starszego HTML 4.01 specyfikacja tej wersji zawiera szczegółowe instrukcje jak postępować z niepoprawną składnią, przez co strony z błędami będą wyświetlane w ten sam sposób w różnych przeglądarkach.

2.4.2 CSS3 oraz less

Kaskadowe arkusze stylów — CSS to język służący do opisu formy prezentacji stron WWW. CSS został opracowany przez W3C w 1996 roku. Język ten jest potomkiem języka *DSSSL*. Pierwszy szkic specyfikacji CSS został zaproponowany w 1994 roku.

CSS to lista dyrektyw – reguł – ustalających w jaki sposób ma zostać wyświetlona przez przeglądarkę internetową zawartość wybranego elementu (X)HTML lub XML. Można w ten sposób opisać wszystkie pojęcia odpowiedzialne za prezentację elementów dokumentu strony internetowej, takie jak rodzina czcionek, kolor tekstu, marginesy, odstępy międzywierszowe, pozycja danego elementu względem innych elementów bądź okna przeglądarki. Zastosowanie arkuszy stylów daje znacznie większe możliwości pozycjonowania elementów na stronie, niż oferuje sam HTML [11].

[12] Obiektowy model dokumentu

CSS został stworzony w celu oddzielenia struktury dokumentu od formy jego prezentacji. Separacja zmniejsza zawiłość dokumentów, ułatwia wprowadzanie zmian w strukturze dokumentu. Arkusze stylów ułatwiają także zmiany w renderowaniu strony w zależności od obsługiwanego medium (ekran, telefon, tablet, dokument do druku). Stosowanie zewnętrznych arkuszy CSS daje możliwość zmiany wyglądu wielu stron naraz bez ingerowania w sam kod HTML, ponieważ arkusze mogą być wspólne dla wielu dokumentów.

Pierwotnie HTML był językiem wyłącznie do opisu struktury dokumentu. Jednak z czasem zrodziła się potrzeba ożywienie wyglądu takich dokumentów. Powoli dodawano nowe znaczniki do HTML pozwalające kontrolować kolory, typografię, dodawać nowe media. Te niestandardowe rozszerzenia implementowane były przez producentów przeglądarek bez porozumienia z innymi producentami. Taka sytuacja doprowadziła do zaimplementowania nowych znaczników działających w konkretnej grupie przeglądarek i nie działających w innych przeglądarkach. Twórcy stron internetowych byli zmuszeni do wysyłania do klienta różnych wersji tej samej witryny w zależności od użytej przeglądarki. Uzyskanie identycznego wyglądu w różnych przeglądarkach było praktycznie niemożliwe. Håkon Wium Lie jako pierwszy zaproponował CHSS (Cascading HTML Style Sheets) w październiku 1994 roku. Później Lie i Bert Bos pracowali wspólnie nad standardem CSS (literka H została usunięta ze względu na możliwość stosowania stylów do innych podobnych do HTML języków). Po przejęciu prac przez, dopiero co utworzoną, organizację W3C w 1996 roku wydano oficjalną dokumentację CSS, poziomu 1 [11].

Przed pojawieniem się CSS wszystkie informację dotyczące wyglądu dokumentów HTML (wygląd czcionek, ułożenie, margines itp.) zawarte były w znacznikach HTML. Język CSS umożliwił przeniesie tych informacji do osobnego pliku. Zabieg ten upraszcza oraz zwiększa przejrzystość samego dokumentu HTML. Bez użycie CSS w przypadku definiowania stylu dla danego elementu np. `h1` jego definicja musiałby zostać powtórzona w każdym miejscu użycia danego elementu. Doprowadziło by to zmniejszenia czytelności dokumentu i jego odporności na błędy oraz zwiększyło by trudność jego utrzymania. Wprowadzenie zmiany w definicji stylu danego elementu wymagało by zmian w wielu miejscach dokumentu HTML. CSS umożliwia rozdzielenie warstwy prezentacji od warstwy struktury. CSS umożliwia definicję kolorów, czcionek, układu, rozmiarów, marginesów oraz wielu innych cech związanych z warstwą prezentacji.

Specyfikacja CSS3 w odróżnieniu od jej poprzedniej wersji CSS2 została podzielona na niezależne moduły. Każdy moduł zawiera nowe możliwości i rozszerza elementy zdefiniowane w CSS2, tak aby zachować kompatybilność z wersjami wcześniejszymi. Prace nad trzecim poziomem CSS rozpoczęły się krótko po oficjalnej publikacji CSS2. Najwcześniejsza wersja CSS3 pojawiła się w czerwcu 1999 roku. W wyniku podzielenia na moduły poszczególne elementy CSS3 mają różny status czy poziom stabilności. Dzięki temu po-

działowi poszczególne moduły mogą być publikowane jako obowiązujące niezależnie od etapu prac nad pozostałymi elementami. CSS3 zyskał nowe znacznik i właściwości. W czerwcu 2012 roku *CSS Working Group* miało opublikowanych ponad 50 różnych modułów a kilka z nich zostało oficjalnie zarekomendowanych jako standardy przez W3C np: typy mediów (*media queries*, przestrzenie nazw (*namespaces*, selektory (*selectors*), kolory [11].

Niektóre z modułów, na przykład tła i obramowania czy układ wielokolumnowy, mają nadany status CR (*Candidate Recommendation*), który uważany jest za raczej stabilny. Dostawcy przeglądarek internetowych powinni zatem sukcesywnie poprawiać dotychczasowe implementacje w celu usuwania tzw. prefiksów dostawców (*vendor prefix*) w nazwach właściwości. Czyli zamiast `-moz-border-radius` dla silnika *Gecko*, `-webkit-boerder-radius` dla sinika *WebKit*, powinno zostać zaimplementowane `border-radius`.

Less (*Leaner CSS*) to dynamiczny język arkuszy stylów stworzony przez Alexis Salliera. Został stowrzony w odpowiedzi na język *Sass* oraz dał początek nowszej wersji Sass - *SCSS*, która zapożyczyła cześć jego składni. Less było początkowo oprogramowaniem open source opartym na licencji MIT, którą zmieniono później na Apache License 2.0. Pierwsza implementacja napisana została w Ruby, późnej została ona zastąpiona wersją napisaną w JavaScript. Less jest zagnieżdżonym metajęzykiem – poprawny kod CSS jest również poprawnym kodem Less. Less dostarcza takie mechanizmy jak zmienne, zagnieżdżanie, mixiny, operatory oraz funkcje. Less może działać zarówno po stronie klienta, jak i serwera, jak również jego kod może być skompilowany do czystego CSS.

Meteor po zainstalowaniu paczki `less` automatycznie kompiluje pliki `.less` do CSS a rezultat jest dołączany do pozostałych plików CSS.

2.4.3 Bootstrap

Bootstrap jest to wolna oraz otwarto źródłowa kolekcja narzędzi do tworzenia stron internetowych oraz aplikacji sieciowych. Zawiera szablony projektowe oparte o HTML oraz CSS dla typografii, formularzy, nawigacji, oraz innych elementów interfejsu użytkownika zawiera ona także opcjonalne rozszerzenia napisane w JavaScript. Głównym celem frameworka jest ułatwienie tworzenia aplikacji sieciowych jaki i dynamicznych stron internetowych. Jest to framework front-endowy[13] stanowi on podstawę interfejsu użytkownika [5].

Bootstrap został stowrzony przez Marka Otta oraz Jacoba Thortona jako framework na potrzeby wewnętrznych projektów w Twitterze, które były budowane z wykorzystaniem różnych bibliotek na potrzeby interfejsu użytkownika. Początkowo nazywany był

[13]Front-end jest odpowiedzialny za pobieranie danych od użytkownika oraz przekazanie ich do back-endu. Następnie back-end na podstawie tych danych wykonuje określone zadanie.

jako *Twitter Blueprint*. Bootstrap został wydany 19 sierpnia 2011 jako projekt otwarto źródłowy. Projekt jest utrzymywany oraz rozwijany przez Marka Otta, Jacoba Thortona z małą grupą programistów rdzenia frameworka oraz przez dużą społeczność programistów. Następna wersja Bootstrap 2 został wydana 31 stycznia 2012. Obecna wersja Bootstrap 3 została wydana 19 sierpnia 2013 roku. W tej wersji położono nacisk na urządzenia mobilne oraz płaskie projektowanie[14] [5].

Bootstrap jest kompatybilny z najnowszymi wersjami przeglądarek Google Chrome, Firefox, Internet Explorer, Opera oraz Safari. Od wersji 2.0 framework wspiera także *responsive web design* — responsywność[15] — wygląd strony internetowej dynamicznie zmienia się w zależności od charakterystyki użytego urządzenia (komputer, tablet, telefon).

2.4.4 AdminLTE

AdminLTE jest to otwarto źródłowy szablon dla aplikacji sieciowych do tworzenia paneli administracyjnych lub paneli kontrolnych. Szablon oparty jest o framework Bootstrap 3. Wykorzystuje wszystkie komponenty zawarte w frameworku Bootstrap wraz z ich wygląd oraz przeprojektowując wygląd wielu często używanych wtyczek w celu stworzenia spójnego wyglądu interfejsu użytkownika. AdminLTE zbudowany jest w oparciu o moduły, co pozwala na jego łatwe dostosowanie do wymagań oraz rozszerzanie o nowe funkcjonalności.

2.5 System kontroli wersji

Jako system kontroli wersji został użyty *Git*. Git jest to rozproszony system kontroli wersji. Został stworzony przez Linusa Torvaldsa jako narzędzie wspierające rozwój jądra Linux. Git jest to wolne oprogramowanie i został opublikowany na licencji GNU GPL w wersji 2. Najważniejsze cechy Git to:

- dobre wsparcie dla rozgałęzionego procesu tworzenia oprogramowania — jest dostępnych kilka algorytmów łączenie zmian z dwóch gałęzi, a także można dodawać własne algorytmy;
- praca off-line — każdy z programistów posiada własną kopię repozytorium, do której może zapisywać zmiany bez połączenia z siecią, następnie zmiany mogą być

[14] Styl projektowania grafiki zakładający wyeliminowanie takich elementów jak gradienty i cieniowanie, ograniczenie liczby kolorów i stosowanie tylko prostych kształtów i typografii

[15] Technika projektowania strony www, tak aby jej wygląd i układ dostosowywał się automatycznie do rozmiaru okna urządzenia, na którym jest wyświetlany np. przeglądarki, smartfonów czy tabletów[1]. Strona tworzona w takiej technice jest uniwersalna i wyświetla się dobrze zarówno na dużych ekranach, jak i na smartfonach czy tabletach.

wymieniane między lokalnymi repozytoriami;

- wsparcie dla istniejących protokołów sieciowych — dane można wymieniać przez HTTP/HTTPS, FTP, rsync, SSH;
- efektywna praca z dużymi projektami — system Git według zapewnień Torvaldsa, a także według testów fundacji Mozilla, jest o rzędy wielkości szybszy niż niektóre konkurencyjne rozwiązania;
- każda rewizja to obraz całego projektu — w przeciwieństwie do innych systemów kontroli wersji, Git nie zapamiętuje zmian między kolejnymi rewizjami, lecz kompletne obrazy, wymaga to nieco więcej pracy aby porównać dwie rewizje, lecz pozwala na przykład na automatyczną obsługę zmian nazw plików.

Jako hosting dla repozytorium projektu został wykorzystany *GitHub*[16]. GitHub to hostingowy serwis internetowy przeznaczony dla projektów programistycznych wykorzystujących system kontroli wersji Git. Stworzony został przy wykorzystaniu frameworka Ruby on Rails i języka Erlang. Serwis działa od kwietnia 2008 roku. W kwietniu 2011 roku ogłoszono, iż GitHub obsługuję 2 miliony repozytoriów. GitHub udostępnia darmowy hosting programów otwarto źródłowych oraz płatne prywatne repozytoria. Repozytorium dla projektu napisanego na potrzeby niniejszej pracy dyplomowej znajduje się pod adresem `https://github.com/yp2/intranet`.

[16] `https://github.com`

Rozdział 3

Aplikacja Intranet

Poniższy rozdział omawia implementację aplikacji służącej jako intranet dla firmy z sektora IT, opisane zostaną założenia oraz –wymagania stawanie dla strony serwerowej oraz dla klientów–. W rozdziale poruszone zostaną najistotniejsze oraz kluczowe elementy aplikacji. Aplikacja została napisana z wykorzystaniem frameworka Meteor.js. Część serwera jak i kliencka została za programowana z użyciem JavaScriptu. Warstwa prezentacji została napisana z użyciem HTML5, CSS3/less, Bootstrap oraz AdminLTE. Bazę danych dla aplikacji stanowi nierelacyjna baza danych MongoDB wraz z jej kliencką implementacją Minimongo.

3.1 Założenia

Jak już wspomniono wcześniej do najważniejszych funkcjonalności aplikacji intranetowych zaliczamy dzielenie wiedzy oraz informacji, komunikację pomiędzy pracownikami danej organizacji oraz wspomaganie pracy zespołowej. W oparciu o ten najważniejsze funkcjonalności budowana aplikacja powinna spełniać następujące założenia:

- tworzenie oraz rejestracja organizacji;
- tworzenie oraz rejestracja użytkowników;
- nazwa użytkownika oraz organizacji to jego emial;
- dodawanie utworzonego użytkownika do organizacji;
- tworzenie oraz rejestracja użytkowników z wykorzystaniem zaproszeń powiązanych z organizacją;

- organizacja w swoim zakresie, powinna posiadać główny zbiór artykułów — główne wiki[1];
- główne wiki dla organizacji ma być one widoczne dla wszystkich użytkowników należących do danej organizacji;
- główne wiki ma posiadać główną kategorię, której nie można usunąć;
- tworzenie, edycja, usuwanie kategorii artykułów w głównej wiki;
- tworzenie, edycja, usuwanie artykułów w głównej wiki;
- artykuły mogą być dodawana do głównej kategorii wiki lub do utworzonych przez użytkownika kategorii;
- edycja oraz prezentacja artykułów na obsługiwać język znaczników *markdown*[2] w szczególności jego implementację *GitHub Flavored Markdown* — GFM;
- artykuły mogą mieć stan opublikowany oraz do publikacji;
- użytkownik nie będący twórcą danej organizacji ma dostęp tylko do artykułów opublikowanych przez innych użytkowników oraz do wszystkich swoich artykułów niezależnie od ich stanu;
- organizacja ma dostęp w swoim zakresie do wszystkich artykułów, wszystkich użytkowników należących do danej organizacji niezależnie od stanu ich publikacji;
- kategoria, w której są artykuły nie może zostać usunięta;
- artykuł może należeć tylko do jednej kategorii;
- użytkownik w swoim zakresie posiada główną wiki;
- główna wiki dla użytkownika jest widoczna tylko dla niego w jego zakresie;
- główna wiki dla użytkownika ma takie same funkcjonalności jak główna wiki dla organizacji;

[1] Typ serwisu internetowego, w którym treść można tworzyć i zmieniać z poziomu przeglądarki internetowej, za pomocą języka znaczników lub edytora WYSIWYG. Strony wiki, ze względu na swoją specyfikę, są przede wszystkim wykorzystywane do pracy nad wspólnymi projektami, takimi jak repozytoria wiedzy na wybrany temat lub projekty różnych grup społecznych

[2] Język znaczników przeznaczony do formatowania tekstu zaprojektowany przez Johna Grubera i Aarona Swartza. Został stworzony w celu jak najbardziej uproszczenia tworzenia i formatowania tekstu. Markdown został oryginalnie stworzony w Perlu, później dostępny w wielu innych. Jest rozpowszechniany na licencji BSD i jest dostępny jako wtyczka do kilku systemów zarządzania treścią.

- prezentacja, tworzenie, edycja oraz usuwanie artykułów dla głównego wiki dla użytkownika ma spełniać opisane cechy jak dla artykułów dla organizacji;
- użytkownicy mogą zmieniać zakres w pomiędzy zakresami organizacji, do których należą i swoim zakresem;
- organizacja oraz użytkownicy mogą tworzyć, usuwać projekty w swoich zakresach lub zakresach, do których należą;
- projekty powinny posiadać stronę z podsumowaniem;
- projekty powinny posiadać niezależne wiki;
- projekty powinny mieć możliwością dodawania — zapraszanie — użytkowników oraz ich usuwanie;
- wiki dla projektów ma mieć te same cechy jak główne wiki;
- projekty powinny posiadać możliwość komunikacji pomiędzy użytkownikami przynależącymi do danego projektu;
- twórca projektu może dodawać oraz usuwać użytkowników, którzy mają do niego dostęp;
- użytkownicy widzą tylko projekty, do których zostali zaproszeni;
- organizacja ma dostęp do wszystkich projektów utworzonych w jej zakresie;
- komunikacja w obrębie projektu mam mieć możliwość dodania wiadomości wraz z jej tytułem oraz możliwością dodana komentarzy;
- wysyłanie zaproszeń do projektów oraz organizacji — jeżeli użytkownik z podanym adresem email istnieje w systemie ma być automatycznie dodany do projektu lub organizacji;
- aplikacja ma być dostępna na wszystkich znaczących platformach oraz jak największej ilości urządzeń.

3.2 Wymagania

Niezależność od platformy oraz dostępność na jak największej ilości urządzeń narzuciła wybór typu aplikacji. Budowana aplikacja będzie typu sieciowego. Klienci będą uzyskiwać do niej dostęp poprzez przeglądarki internetowe kontaktując się z serwerem, na którym będzie dostępna budowana aplikacja.

W oparciu o założenia jakie ma spełniać aplikacja *Intranet* oraz o obecne trendy w rozwoju aplikacji sieciowych do budowy aplikacji został wybrany framework Meteor.js, a jako język programowania JavaScript. Wybór tych technologii był podyktowany także szybkością oraz prostotą budowania aplikacji. Meteor.js używa JavaScriptu zarówno po stornie klienta oraz serwera co skraca czas nauki samego framework'a eliminując potrzebę nauki dodatkowego języka programowania. Tematyka oraz założenia projektu nie stanowią problemu dla przyjętego przez framework Meteor.js nierelacyjnego rozwiązania bazodanowego. Aplikacja nie posiada krytycznych elementów takich jak transakcje finansowe, zamówienia oraz innych elementów wymagających transakcji operacji na całej bazie danych. Dane są typowo informacyjne w związku z tym wykorzystanie bazy MongoDB nie stanowi żadnego zagrożenia dla użytkowników oraz samej aplikacji.

Aplikacja ma być dostępna na wszystkich znaczących platformach oraz jak największej ilości urządzeń. Aby to zapewnić w warstwie prezentacji wykorzystano HTML5, CSS3 wraz z kompilatorem less oraz domyślnie wykorzystywaną przez Meteor.js bibliotekę JQuery. W celu zachowania responsywności interfejsu użytkownika wykorzystano szablon AdminLTE, który oparty jest o framework Bootstrap. Wykorzystanie Bootstrapa dostarcza prosty oraz szybki sposób na dostosowanie układu i wyglądu interfejsu użytkownika w zależności od używanego przez niego urządzenia. Rozwiązanie to znaczenie zwiększa użyteczność aplikacji podczas używania urządzeń o różnych wielkościach oraz rozdzielczościach ekranu.

3.3 Struktura aplikacji

Aplikacje napisane z wykorzystaniem frameworka Meteor.js to zestaw plików JavaScript, które działają w przeglądarce internetowej i w kontenerze Node.js po stronie serwera, i innych plików pomocniczych takich jak szablony HTML, kaskadowe arkusze stylów oraz statycznych zasobów. Meteor.js automatycznie pakuje oraz zajmuję się przesyłaniem tych różnych elementów. Framework jest dość elastyczny jeżeli chodzi o strukturę katalogów w jakiej będą umieszczone poszczególne elementy aplikacji. Nazwy plików jaki i katalogów mogą mieć wpływ na to w jakiej kolejności są one ładowane oraz gdzie są one ładowane. Meteor.js niektóre katalogi traktuje w specjalny sposób.

Katalogi o nazwie `client` nie są ładowane po stronie serwera. Wszystkie zasoby znajdujące się w tym katalogu gdy aplikacja nie jest uruchomiona w trybie produkcyjnym są łączone oraz minimalizowane. W trybie developerskim plik JavaScript oraz CSS nie są minimalizowane aby ułatwić znajdowanie błędów w aplikacji. Jednakże pliki CSS nadal są łączone w jeden plik. Pliki HTML po stronie klienta są skanowane w poszukiwaniu trzech elementów najwyższego poziomu `<head>`, `<body>`, `<template>`. Elementy head oraz

body łączone są w jeden element i następnie przesyłane do klienta podczas pierwszego ładowania strony.

Zasoby znajdujące się w katalogach o nazwie `server` nie są ładowane po stronie klienta. Krytyczne oraz wrażliwe elementy aplikacji takie jak zarządzenia hasłami, proces autoryzacji powinny znajdować się w tych katalogach. Meteor zbiera wszystkie pliki JavaScript za wyjątkiem tych znajdujących się w katalogach `public`, `client` i `private` oraz ich podkatalogach i ładuje je do instancje Node.js znajdującej się na serwerze. Kod na serwerze nie jest uruchamiany jako kod asynchroniczny typowy dla aplikacji opartych o Node.js. Meteor uruchamia kod pojedynczy wątek na żądanie

Wszystkie pliki znajdujące w katalogu `public` najwyższego poziomu serwowane są dla klientów takim jakimi są. Jest to właściwe miejsce dla plików takich jak `favicon.ico`, `robots.txt` oraz innych podobnych plików.

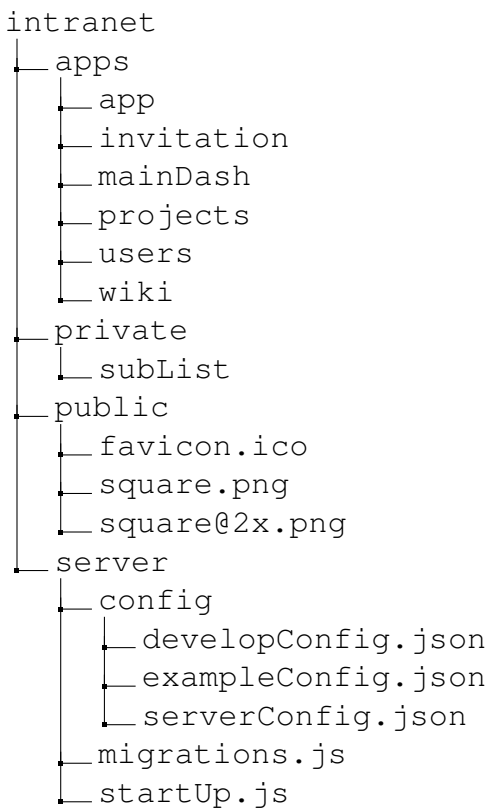
Pliki znajdujące się w katalogu najwyższego poziomu o nazwie `private` dostępne są tylko po stronie serwera z wykorzystaniem API `Assets`. Katalog ten przeznaczony jest dla takich plików jak prywatne dane oraz plików, które nie mają być dostępne z zewnątrz.

Podczas ładowanie plików przez Meteor.js obowiązują następujące reguły:

1. szablony HTML ładowane są zawsze jako pierwsze;
2. pliki zaczynające się na `main.` ładowane są zawsze jako ostanie;
3. następnie ładowane są pliki znajdujące się w jakimkolwiek katalogu `lib/`;
4. następnie ładowane są plik z głębszą ścieżką;
5. następnie pliki ładowane są w kolejności alfabetycznej uwzględniającej całą ścieżkę katalogów.

Wszystkie pliki JavaScript nie znajdujące się w specjalnych katalogach ładowane są zarówno po stronie klienta jaki i serwera. Jest to dobre miejsce na definicje modeli oraz innych funkcji, które mają być dostępne zarówno dla klienta jak i serwera. Meteor.js dostarcza zmienne `Meteor.isClient` oraz `Meteor.isServer`, które można użyć do zmiany zachowania w zależności czy kod jest uruchomiony na serwerze czy kliencie. Pliki CSS oraz HTML poza specjalnymi katalogami ładowane są tylko po stornie klienta i nie mogą zostać użyte po stronie serwera.

Rysunek 3.1 przedstawia główny katalog aplikacji *Intranet*. Skład się on z następujących podkatalogów: `apps`, `private`, `public`, `server`. W katalogu `apps` znajdują się poszczególne moduły aplikacji. Katalog `private` zawiera zasoby nie dostępne z zewnątrz, znajduje się w nim opis subskrypcji wykorzystywanych przez aplikację. Katalog `public` zawiera pliki graficzne używane przez aplikację. W katalogu `server` znajduje się plik

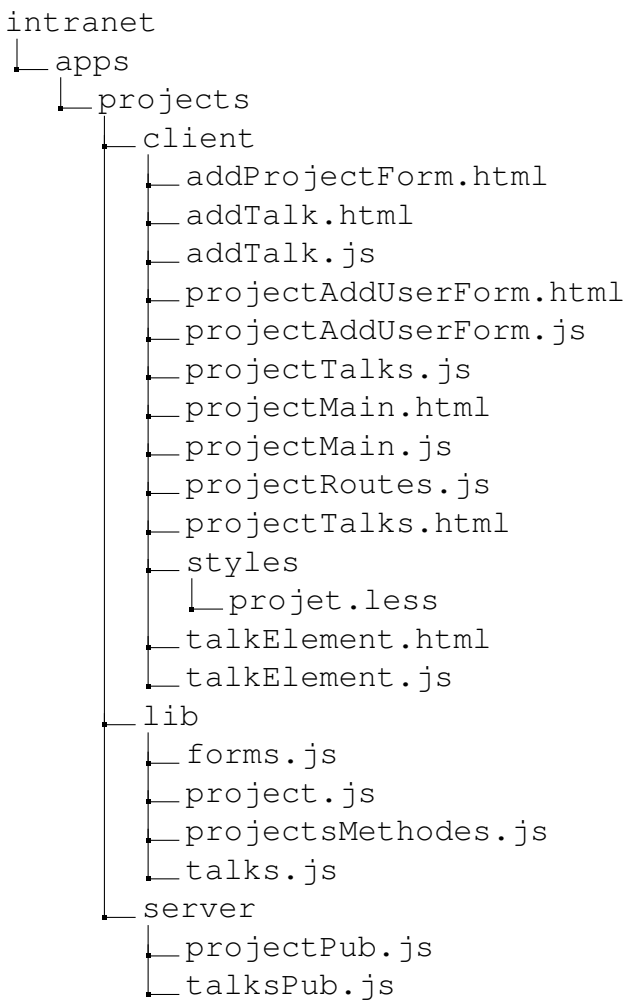


Rysunek 3.1: Główny katalog aplikacji *Intranet*

odpowiedzialny za migrację `migrations.js` oraz plik `startUp.js` zawierający kod uruchamiany na samym początku startu strony serwerowej. W podkatalogu `config` zanajdują się pliki konfiguracyjne zapisane w postaci JSON. Plik `developConfig.json` używany jest podczas pracy rozwojowych nad aplikacją. Plik `serverConfig.json` używany jest w środowisku produkcyjnym.

Katalog `apps` zawiera poszczególne moduły aplikacji. Podkatalog `app` zawiera kod wykorzystywany w całej aplikacji, konfigurację używanych pluginów, globalne definicje *helperów* — funkcji pomocniczych używanych w szablonach HTML, pliki CSS wspólne dla całej aplikacji, pomocnicze szablony HTML oraz rozszerzenie wbudowanego obiektu szablonu o nową właściwość `parentTemplate`.

Rysunek 3.2 przedstawia strukturę modułu *projects* — projekty. Widać w niej charakterystyczny podział aplikacji Meteor.js z wykorzystaniem katalogów specjalnych `client` oraz `server`. Katalog `lib` zawiera kod wykorzystywany zarówno po stronie serwera jaki i klienta. Pliki zawarte w tym katalogu muszą zostać załadowane wcześniej niż kod znajdujący się w katalogach specjalnych. Kod ten zawiera definicje kolekcji projekty – `project.js` i konwersacji – `talks.js`, metody odpowiedzialne za operacje na danych w kolekcjach – `projectsMethodes.js` oraz definicje używanych przez moduł formularzy – `forms.js`. W katalogu `client` znajdują się poszczególne widoki używane przez moduł oraz szablony poszczególnych elementów włączane przez inne szablony pod-



Rysunek 3.2: Struktura przykładowego modułu *projects*

czas renderowania ich zawartości. W pliku `projectRoutes.js` zdefiniowane są trasy — *routing* — dla modułu. Strona serwerowa składa się z dwu plików, które zawierają publikacje dla kolekcji jakich dostarcza moduł. Publikacje mogą być używane także poza modułem, w którym zostały zdefiniowane. Podkatalog `styles` katalogu `client` zawiera plik dynamicznego arkusza stylów less – `project.less`.

3.4 Wspólna część dla strony serwerowej oraz klienckiej

3.4.1 Kolekcje

Dane, w bazie danych, wykorzystywane przez aplikację przechowane są w postaci dokumentów te natomiast grupowe są w logiczne zbiory nazywane kolekcjami. Klient oraz serwer używają tego samego API do dostępu do bazy danych. Klasa `Mongo.Collection` używana jest do definiowania oraz manipulowania kolekcjami. Listing 3.1 przedstawia sposób definicji kolekcji dla projektów — `Project`. Definicja jest umieszczona w katalogu

`lib` modułu `projects`, który ładowany jest po stronie klienta jak i serwera przed plikami znajdującymi się w katalogach `client` oraz `server`. Tak zdefiniowaną kolekcję możemy używać po stronie serwera jak i klienta.

Listing 3.1: Definicja kolekcji dla projektów *Project*

```
1 Project = new Meteor.Collection('project');
```

Wykorzystując dodatkowy pakiet `collection-hooks`[3] możemy rozszerzyć klasę `Mongo.Collection` o dodatkowe właściwości dla operacji dodawania, modyfikowania oraz usuwania dokumentów w kolekcji wykonywane przed — `before` oraz po — `after` wykonaniu tych operacji. Przykładowe zastosowanie przedstawia listing 3.2. Rozszerzeniu ulegają operacje dodawania oraz modyfikacji. Przed wykonaniem operacji dodawania (`insert`) dokumentu do kolekcji dodawane jest pole `createdAt` z wartością ustawianą na dzisiejszą datę. Przed wykonywaniem operacji aktualizacji (`update`) do modyfikatora jaki zostanie użyty podczas operacji aktualizacji dokumentu dodawane są pola `modifiedAt` z wartością ustawioną na dzisiejszą datę oraz pole `modifiedBy` z wartością jednoznacznie identyfikującą dokument użytkownika tak zwane `_id` dokumentu. Linia 12 listingu 3.2 przedstawia użycie operatora *lub* – || – do przypisywania zmiennym, w tym przypadku argumentowi funkcji, wartości. Jeżeli argument ma wartość jego wartość się nie zmienia — ulega ponownemu przypisaniu. Jeżeli natomiast wartość argumentu jest niezdefiniowana (`undefined`) argumentowi przypisany zostanie pusty obiekt — wyrażenie `{}` jest literałem nowego obiektu.

Listing 3.2: Rozszerzanie kolekcji *Project* o dodatkowe właściwości

```
1  Project.before.insert(function (userId, doc) {
2      "use strict";
3
4      let createDate = new Date();
5      doc.createdAt = createDate;
6  });
7
8  Project.before.update(function (userId, doc, fieldNames, modifier, options)
       {
9      "use strict";
10
11     let modDate = new Date();
12     modifier.$set = modifier.$set || {};
13     modifier.$set.modifiedAt = modDate;
14     modifier.$set.modifiedBy = userId
15 });
```

[3] https://github.com/matb33/meteor-collection-hooks

3.4.2 Metody

Meteor.js oferuje mechanizm zdalnie wywoływanych metod. Klient może wywołać zdefiniowane metody zdalnie na serwerze. Metody są definiowane przez wywołanie `Meteor.methods`. Wywołanie `Meteor.methods` po stornie serwera tworzy funkcje, które mogą zostać zdalnie wywoływane przez klienta. Funkcje takie powinny zwracać wartość, która może zostać sparsowana do obiektu *EJSON* lub zwracać błąd. Natomiast wywołanie `methods` po stornie klienta definiuje funkcje typu *stub* — funkcja symulująca zachowanie metody o takiej samej nazwie zdefiniowanej na serwerze. Jeżeli *stub* zostanie zdefiniowany a klient wywoła metodę z nim powiązaną *stub* zostanie wywołany równolegle. Po stronie klienta wartość zwracana przez *stub* jest ignorowana. Funkcje `stub` są uruchamiane dla ich skutków ubocznych — ich celem jest symulacja wyników funkcji wywoływanych na serwerze bez konieczności oczekiwania na ich ukończenie oraz przesłanie ich wyniku z serwera.

Listing 3.3: Wywołanie *Meteor.methods* oraz definiowanie metod

```
1  Meteor.methods({
2      removeUserFromProject (data) {
3          check(data, {
4              userId: String,
5              projectId: String
6          });
7
8          let project = Project.findOne(data.projectId);
9          if (Meteor.isServer) {
10
11             if (!project || (!_.includes(project.secure.allowedUsers, this.
       userId) && project.secure.admin.id !== this.userId) ) {
12                 console.log('cant delete');
13                 throw new Meteor.Error(403, "Can't remove user from project
       ");
14             }
15             Project.update({_id: project._id}, {$pull :{"secure.
       allowedUsers": data.userId, allowedUsers: data.userId}});
16             return true;
17         }
18
19
20     },
21     ...$
22 })
```

Wykorzystując `Meteor.isServer` oraz `Meteor.isClient` możemy rozdzielić kod metod na część wykonywaną tylko na serwerze, kliencie lub wspólnie dla obu przypadków. Definiując metody w części wspólnej dla serwera oraz klienta z wykorzystaniem powyższych elementów API możemy pominąć definiowanie *stubów* dla metod po stronie klienta. Listing 3.3 przedstawia tak zdefiniowaną metodę umieszczoną w pliku `projectsMethodes.js` znajdującym się w katalogu `lib`. Metody mogą przyjmować argumenty. Do sprawdzenia poprawność argumentów metody można skorzystać z funk-

cji sprawdzających wzorce takich jak `check` oraz `Match.test`. Funkcje te dostarczane są przez pakiet `check`. Listing przedstawia usuwanie użytkownika z projektu. Linia 15 przedstawia aktualizację dokumentu w kolekcji `Project` o podanym `_id`. Z tablicy `allowedUsers` usuwany jest ciąg znaków zawierających id danego użytkownika przekazane do metody w argumencie `data`. Pierwszym atrybutem przejmowanym przez metodę `update` wykonywaną na rzecz kolekcji to selektor określający jaki dokument lub dokumenty mają zostać zmodyfikowane. Drugi argument to modyfikator określający sposób modyfikacji danego dokumentu.

3.5 Cześć serwerowa

3.5.1 Baza danych

Relacje

Jako już wspomniano jako bazę danych wykorzystano MongoDB. Jest to baza nierelacyjna zorientowana na dokumenty, które logicznie grupowane są w kolekcje. Struktura dokumentów jest bardzo elastyczna a same kolekcje dokumentów nie narzucają jej. Aby odwzorować relacje pomiędzy dokumentami możemy użyć następujących wzorców: dokumenty wbudowane lub referencji.
Wzorzec dokumentów wbudowanych polega na dodaniu całego dokumentu od jakiego się odnosimy jako pola w innym dokumencie. Tak dodany dokument będzie odpowiadał relacji jeden do jednego. Listing 3.4 przedstawia dwa dokumenty. Pierwszy reprezentuje osobę, drugi reprezentuje adres. Po dodaniu pola `address` i przeniesieniu do niego całego dokumentu adresu uzyskujemy relację jeden do jednego z wykorzystaniem wzorca wbudowanych dokumentów. Wadą tego rozwiązania jest powielanie danych w bazie, zaletą natomiast to, że chcąc odczytać adres danej osoby pobieramy z bazy tylko jeden rekord.

Listing 3.4: Dokumenty wbudowane – relacja jeden do jednego

```
1  // Dokumenty przed utworzeniem relacji
2  {
3     _id: "joe",
4     name: "Joe Bookreader"
5  }
6
7  {
8     patron_id: "joe",
9     street: "123 Fake Street",
10    city: "Faketon",
11    state: "MA",
12    zip: "12345"
13 }
14
15 // Wzorzec dokumenty wbudowane relacja jeden do jednego
```

```
16
17 {
18    _id: "joe",
19    name: "Joe Bookreader",
20    address: {
21                street: "123 Fake Street",
22                city: "Faketon",
23                state: "MA",
24                zip: "12345"
25              }
26 }
```

Następny listing (3.5) przedstawia wykorzystanie tego samego wzorca w celu uzyskania relacji jeden do wielu. Jak widać na listingu pole `address` ma postać tablicy złożonej z dokumentów.

Listing 3.5: Dokumenty wbudowane – relacja jeden do wielu

```
1  {
2     _id: "joe",
3     name: "Joe Bookreader",
4     addresses: [
5                   {
6                     street: "123 Fake Street",
7                     city: "Faketon",
8                     state: "MA",
9                     zip: "12345"
10                  },
11                  {
12                    street: "1 Some Other Street",
13                    city: "Boston",
14                    state: "MA",
15                    zip: "12345"
16                  }
17                ]
18 }
```

Wzorzec referencji unikamy powielania danych poprzez podanie referencji do innego dokumentu w innej lub tej samej kolekcji. Referencją najczęściej jest `_id` dokumentu. Wadą tego wzorca jest konieczność pobierania kolejnych dokumentów z bazy w celu odczytania danych z dokumentu powiązanego. Wykorzystując ten wzorzec otrzymujemy relację jeden do wielu. Listing 3.6 przedstawia wykorzystaniem podlegającej zmianie oraz rosnącej tablicy z `_id` poszczególnych dokumentów.

Listing 3.6: Referencje (tablica) – relacja jeden do wielu

```
1  {
2     name: "O'Reilly Media",
3     founded: 1980,
4     location: "CA",
5     books: [12346789, 234567890, ...] // _id of books for this publisher
6  }
7
8  {
9      _id: 123456789,
```

```
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English"
}
```

Inny sposób tworzenie referencji został pokazany na listingu 3.7, w którym to `_id` dokumentu powiązanego jest przechowywane w polu innego dokumentu.

Listing 3.7: Referencje – relacja jeden do wielu

```
{
   _id: "oreilly",
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA"
}

{
   _id: 123456789,
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher_id: "oreilly"  // reference
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher_id: "oreilly"  // reference
}
```

Do modelowania relacji w bazie danych wykorzystywanej przez aplikacje Intranet wykorzystano wzorzec dokumentów wbudowanych oraz zmodyfikowany wzorzec referencji. Listing 3.8 przedstawia dokument z kolekcji `Wiki` reprezentujący wiki — zbiór artykułów — dla organizacji, użytkownika bądź projektu.

Pole `categories` przedstawia wykorzystanie wzorca dokumentów wbudowanych do przechowywania kategorii dla tej wiki. W polach `admin` oraz `scope` wykorzystano zmodyfikowany wzorzec referencji. Pola te przechowują dokument z referencją do innego do-

kumentu w bazie oraz część najczęściej wykorzystywanych danych z dokumentu powiązanego, w ten sposób jeżeli chcemy uzyskać tylko te dane nie musimy pobierać dodatkowego dokumentu z bazy.

Listing 3.8: Przykładowy dokument z kolekcji *Wiki*

```
{
        "_id" : "e6RhHrwxtznXF7s97",
        "type" : "org",
        "admin" : {
                "name" : "e@e.pl",
                "id" : "xZxtZ9JqoWzzZ7f7g"
        },
        "scope" : {
                "name" : "e.pl",
                "id" : "Csobds6psMftt4Eqw"
        },
        "categories" : [
                {
                        "title" : "main",
                        "titleSlug" : "main"
                },
                {
                        "title" : "Cosie",
                        "titleSlug" : "cosie"
                },
                {
                        "title" : "Cos",
                        "titleSlug" : "cos"
                }
        ],
        "secure" : {
                "type" : "org",
                "admin" : {
                        "name" : "e@e.pl",
                        "id" : "xZxtZ9JqoWzzZ7f7g"
                },
                "scope" : {
                        "name" : "e.pl",
                        "id" : "Csobds6psMftt4Eqw"
                },
                "categories" : [
                        {
                                "title" : "main",
                                "titleSlug" : "main"
                        },
                        {
                                "title" : "Cosie",
                                "titleSlug" : "cosie"
                        },
                        {
                                "title" : "Cos",
                                "titleSlug" : "cos"
                        }
                ]
        },
        "createdAt" : ISODate("2015-09-06T14:38:43.041Z"),
        "modifiedAt" : ISODate("2015-09-17T18:06:56.370Z")
```

```
53 }
```

Kolekcje

Aplikacja Intranet używa następujących kolekcji:

- `Invitation` — dokumenty dotyczące zaproszeń wysyłanych do użytkowników gdy są zapraszaniu do organizacji lub do udziału w projektach;
- `Project` — dokumenty dotyczące projektów tworzonych przez organizację jaki użytkowników;
- `Talks` — konwersacje prowadzone w ramach projektu;
- `Meteor.users` — kolekcja przechowująca dane użytkowników, jest to kolekcja specjalna dostarczana przez framework Metero.js;
- `UserScope` — dokumenty przechowujące dane o zakresach dla organizacji oraz użytkowników;
- `Wiki` — dane dotyczące wiki tworzonych na potrzeby organizacji i użytkowników oraz ich zakresów;
- `WikiArticle` — kolekcja zawierająca dokumenty dotyczące artykuły, które dodawane są do wiki.

Kolekcje definiowana są w osobnych plikach w katalogu `lib` na moduł aplikacji. Wraz z definicjami kolekcji umieszczone są ich rozszerzenia wykonane z wykorzystaniem już wspomnianego pakietu `collection-hooks`.

Publikacje

Dostęp do zestawu danych jakie otrzyma klient jest realizowany za pomocą *publikacji*. Publikacje kontrolują jaki zestaw danych otrzyma klient za każdym razem kiedy dokona subskrypcji tej publikacji. Z powodu sposobu realizacji dostępu do danych przez klienta jaki prezentuje framework a w szczególności to że klient posiada kopię wycinka danych i na nim przeprowadza wszystkie operacje, które następnie są synchronizowane z serwerem a następnie z innymi klientami, bardzo istotne jest poprawne sformowanie publikacji. Źle zakodowana publikacja w ostateczności może doprowadzić do całkowitego „zabicia" przeglądarki klienta. W łatwy sposób możemy przesłać do klienta zestaw np. 100 000 dokumentów, bez znaczącego obciążenia serwera, co całkowicie zatrzyma aplikacje po stronie klienta. Publikacje to także jedyne miejsce, w którym może realizować zasady

dostępu — np. uprawnienia — co do zestawu danych, ich pól, ilości jakie klient ma otrzymać. W publikacja mamy dostęp do `id` użytkownika, który dokonuje subskrypcji. W oparciu o `id` możemy odszukać dokument reprezentujący użytkownik i na jego podstawie zrealizować zasady dostępu. Tu warto wspomnieć o braku ograniczenia co do zestawu danych dla zapytań oraz operacji realizowanych po stornie serwera.

Listing 3.9: Publikacja *„talks”* dla kolekcji Talks

```
"use strict";

Meteor.publish('talks', function (selector, options) {
    let sel = selector || {},
        opt = options || {};

    if (this.userId && sel['project.id']) {
        let draft = Talks.findOne({
            'secure.status': "draft",
            'secure.author.id': this.userId,
            'secure.project.id': sel['project.id']
        });
        _.assign(opt, {sort: {createdAt: -1}, fields: {secure: 0}});

        if (!draft) {
            let user = Meteor.users.findOne(this.userId);
            let insObj = {
                title: "",
                content: "",
                status: "draft",
                author: {
                    id: this.userId,
                    username: user.username
                },
                project: {
                    id:  sel['project.id']
                }
            };

            let taskSecure = {secure: {}};

            _.assign(taskSecure.secure, insObj);

            insObj['secure'] = taskSecure.secure;

            Talks.insert(insObj);
        }

        let result = Talks.find(sel, opt);

        console.log('pub talks', result.count(), sel, opt);
        return result
    }
});
```

Na listingu 3.9 przedstawiono przykładową publikację dla dokumentów z kolekcji reprezentującej konwersacje — kolekcja `Talks`. Funkcja `Meteor.publish` przyjmuje dwa

parametry, pierwszy to łańcuch znaków reprezentujący nazwę publikacji, drugi parametr to funkcja, która będzie wywoływana za każdym razem gdy klient dokona subskrypcji tej publikacji. Funkcja ta, jeżeli klient może otrzymać dane, powinna zwracać obiekt `Collection.Cursor` — kursor dla kolekcji lub tablicę takich kursorów. Przedstawiona publikacja nie tylko zwraca kursor dla kolekcji `Talks` ale także przeprowadza operacje dodania dokumentu do kolekcji. Podczas subskrypcji przez klienta publikacja sprawdza czy w bazie istnieje szkic dla konwersacji jeżeli taki szkic nie istnieje jest on tworzony (linia 36). Następnie tworzony jest kursor — linia 39 — który zawiera szkic dla konwersacji. Następnie taki kursor jest zwracany — linia 42. Kursor powstaje poprzez wywołanie metody `find` na kolekcji i ma postać `Talks.find(selektor, opcje)`. W kodzie widoczny jest `consol.log` w linii 41. Jest on bardzo przydatny w ustaleniu czy subskrypcja realizowana przez klienta przebiegła tak jak należy.

Operacje CRUD

Operacje *CRUD*[4] — tworzenie (*create*), odczytywanie (*read*), aktualizacja (*update*) oraz usuwanie (*delete*) — realizowane jest przez wywoływanie odpowiednich metod na rzecz zdefiniowanych obiektów kolekcji. Tworzenie dokumentów w kolekcji realizowane jest z wykorzystaniem metody `insert`, odczyt danych realizuje metoda `find`, aktualizacja odbywa się z wykorzystaniem metody `update` a metoda `remove` usuwa dokumenty z kolekcji. Listing 3.10 przedstawia przykładowe operacje CRUD.

Listing 3.10: Operacje *CRUD*

```
1  // Tworzenie dokumentu
2  Posts.insert({title: "Hello world", body: "First post"});
3
4  // Odczyt danych
5  Messages.find({userId: Session.get('myUserId')});
6
7  // Aktualizacja dokumentu
8  Messages.update(myMessages[0]._id, {$set: {important: true}});
9
10 // Usuwanie dokumentu strona kliencka
11 Messages.remove({_id: this._id});
12
13 // Usuwanie dokumentu strona serwerowa
14 Players.remove({karma: {$lt: -2}});
```

Powyższy listing pokazuje także różnice pomiędzy operacjami wykonywanymi na kolekcjach po stornie klienta jak i serwera. Po stronie klienta aby usunąć dokument musi podać jego `id`, po stronie serwera takie ograniczenie nie występuje.

Wcześniej wspomniane publikacje regulują dostęp do danych podczas ich odczytu. Aby klient mógł odczytać dane te muszą zostać opublikowane przez serwer i następnie

[4]akronim z języka angielskiego powstały ze słów Create, Read, Update, Delete

przez niego za subskrybowane. Inne operacje regulowane są przez metody `allow` oraz `deny` wywoływane na rzecz kolekcji. Listing 3.11 obrazuje przykładowe definicje reguł dostępu.

Listing 3.11: Definicje *allow* oraz *deny*

```
1  Posts = new Mongo.Collection("posts");
2
3  Posts.allow({
4    insert: function (userId, doc) {
5      // użytkownik musi być zalogowany oraz dokument musi do niego należeć
6      return (userId && doc.owner === userId);
7    },
8    update: function (userId, doc, fields, modifier) {
9      // może modyfikować tylko swoje dokumenty
10     return doc.owner === userId;
11   },
12   remove: function (userId, doc) {
13     // może usuwać tylko swoje dokumenty
14     return doc.owner === userId;
15   }
16 });
17
18 Posts.deny({
19   update: function (userId, doc, fields, modifier) {
20     // nie może zmienić pola właściciel
21     return _.contains(fields, 'owner');
22   },
23   remove: function (userId, doc) {
24     // nie może usunąć dokumentów zablokowanych - locked
25     return doc.locked;
26   }
27 });
```

Gdy klient wywołuje, którąś z metod `insert`, `update` lub `remove` po stornie serwera wywoływane są funkcje zwrotne zdefiniowane w `allow` oraz `deny` po to aby określić czy dana operacja zapisu jest możliwa. Aby dopuścić operacje na kolekcji przynajmniej jedna reguła zdefiniowana w `allow` musi zwrócić prawdę i żadna reguła w `deny` nie zwraca prawdy. Zasady te obowiązują tylko po stronie klienta gdyż nie mamy do końca zweryfikowanego kodu aplikacji. Po stornie serwera nie obowiązują zasady dotyczące dostępu do danych, gdyż mamy pełną kontrole oraz pewność, że nikt nie zmodyfikował naszej aplikacji. JavaScript to język skryptowy z tego powodu aplikacja dostarcza do przeglądarki klienta może być łatwo zmodyfikowana.

Utrzymanie reguł zdefiniowanych z wykorzystaniem `allow` oraz `deny` jest bardzo skomplikowane oraz czasochłonne. Dodając do tego łatwą modyfikację kodu części klienckiej naszej aplikacji nie jest dobrym pomysłem pozwalać klientom na bezpośredni dostęp do operacji na kolekcjach. Kolejnym elementem, który może znacznie utrudnić takie operacje to sam dostęp do danych — klient aby dokonać modyfikacji danych musi mieć do nich dostęp, co może okazać się problemem podczas optymalizacji aplikacji. Wyobraźmy sobie

widok wszystkich powiadomień jakie dostaje użytkownik. Powiadomienia mogą pochodzić ze wszystkich modułów aplikacji. Sprawa nie wydaja się skomplikowana — wystarczy publikacja oraz subskrypcja na jedną kolekcję dotyczącą powiadomień. Sytuacja zaczyna się komplikować gdy z pojedynczego powiadomienia możemy dokonywać akcji dotyczących innych części systemu np. akceptowanie urlopu. Urlopy przechowywane są w innej kolekcji niż powiadomienia. Teraz aby umożliwić taką operację musimy mieć dostęp do kolekcji dotyczącej urlopów. Rodzą się kolejne pytania do jakiej części tej kolekcji ma mieć dostęp dany użytkownik, jak ograniczyć zestaw danych tak aby akceptacja urlopów była możliwa oraz znacząco nie obciążać ani nie spowalniać strony klienckiej. Przedstawiony wycinek to tylko część problemu podejścia do operacji CRUD po stronie klienta. Z tego powodu oficjalne przewodniki odradzają takiego podejścia na rzecz stosowania `Meteor.methods`. Stosujące metody mamy pewność co do poprawności kodu — kod umieszczony w części wspólnej nie może ulec modyfikacji po stronie klienta. Dzieląc umiejętnie kod metod z wykorzystaniem zmiennych `Meteor.isServer` oraz `Meteor.isClient` możemy dowolnie modyfikować zachowanie metod od tego czy kod jest uruchomiony po stronie serwera lub klienta. Drugą zaletą użycia metod to brak ograniczeń w dostępie do danych oraz ograniczeń narzuconych przez reguły `allow` i `deny` gdy metoda jest uruchamiana po stronie serwera. Stosując metody rozwiązanie powyższego problemu staje się trywialnie proste, bezpiecznie — mamy pełną kontrolę nad kodem — a implementacja reguł dostępu do operacji modyfikacji kolekcji jest o wiele prostsza. Utrzymanie tych reguł jest o wiele mniej pracochłonne oraz mniej podatne na błędy. Użycie metod ma jeszcze jedną zaletę — chyba najważniejszą — łatwo napisać kod testujący daną metodę.

3.5.2 Konfiguracja

Każda aplikacja potrzebuje pliku z danymi konfiguracyjnymi. Listing ... przedstawia plik *exampleConfig.json*. Jest to plik przykładowy w formacie *JSON* zwierający klucze używane przez aplikację Intranet, wartości są przykładowe lub zawierają puste ciągi znaków.

Listing 3.12: Przykładowa konfiguracja - *exampleConfig.json*

```
{
  "debug": true,
  "email": {
    "SMTPCreds": "",
    "invitationFrom": "admin@domain.com"
  },
  "hijackEmail": true,
  "hijackEmailAddress": "a@a.pl"
}
```

Aplikacja w środowisku produkcyjnym używa pliku *serverConfig.json*, który jest uzupełniony danymi właściwymi dla tego środowiska. Podczas pisanie aplikacji i jej testowania używany jest inny plik – *developConfig.json*, który zawiera dane właściwe dla tego cyklu tworzenia aplikacji. Plik te przechowywane są po stronie serwerowej po to aby klient nie miał dostępu do wrażliwych danych, które mogą znajdować się w takich plikach. Uruchamianie aplikacji z parametrem `--settings` pozwala na podanie właściwego pliku konfiguracyjnego. Z przyczyn bezpieczeństwa pliki używane podczas rozwoju aplikacji oraz w środowisku produkcyjnym nie są zamieszczone w repozytorium kodu.

3.6 Cześć kliencka

3.6.1 Routing

Aplikacja Intranet jest to aplikacji typu *Single-page Application (SPA)* — aplikacja, która składa się z jednej strony internetowej. Aplikacje tego typu ładują się w całości do pamięci przeglądarki a akcje użytkownika powodują zmiany tylko określonych fragmentów strony bez jej całkowitego przeładowywania. Routing — *trasowanie* — w przypadku aplikacji SPA określa tylko stan aplikacji. Po skierowaniu aplikacji na inny *url* strona nie ulega przeładowaniu — tylko właściwy dla stanu, określonego przez ten *url*, element strony internetowej ulega zmianie. W aplikacji Intranet wykorzystywany jest `FlowRouter` dostarczany przez pakiet `kadira:flow-router`. Routing odbywa się w całości tylko po stronie klienta.

Aplikacja pisane z wykorzystaniem Meteor.js może posiadać w kodzie HTML tylko po jednym elemencie `<body>` oraz `<head>`. Wszystkie szablony będą renderowane w elemencie `<body>` a w elemencie `<head>` będą dołączane wszystkie zależności aplikacji – to jest pliki css oraz js. Do poprawne obsługi routing z wykorzystaniem `FlowRouter` musimy zdefiniować w jakim elemencie będą renderowane szablony obrazujące zmianę stanu aplikacji. Na listingu 3.13 przedstawiono główny plik HTML z tagami `<body>` oraz `<head>`.

Listing 3.13: Główny plik HTML

```
1 <head>
2     <title>IntranetApp</title>
3     <meta content="width=device-width, initial-scale=1, maximum-scale=1,
    user-scalable=no" name="viewport">
4 </head>
5
6 <body>
7 {{> sAlert}}
8 </body>
```

Następny listing (3.14) przedstawia wskazanie elementu HTML w jaki router ma renderować szablony.

Listing 3.14: Wskazanie elementu do renderowania szablonów

```
BlazeLayout.setRoot('body');
```

Kolejny element niezbędny podczas renderowania to podstawowy układ — *layout*, który będzie używany przez aplikację. Aplikacja może posiadać kilka layoutów w zależności od potrzeb np. ekran logowania, widok dla nie zalogowanych użytkowników mogą mieć inne układy. Listing 3.15 przedstawia layout dla aplikacji Intranet dla zalogowanych użytkowników.

Listing 3.15: Główny layout dla aplikacji

```
<template name="mainDashLayout">
    <div class="wrapper">
        {{# if authInProcess }}
            {{> authInProcessLoading}}
        {{else}}
            {{#if canShow }}
                    {{> Template.dynamic template=header}}
                    {{> Template.dynamic template=sideBar}}

                    {{> Template.dynamic template=content}}

                    {{> Template.dynamic template=footer}}
                    {{> Template.dynamic template=controlSideBar}}
                    {{>inviteUserDialog}}
            {{/if}}
        {{/if}}
    </div>
</template>
```

Na listingu widać kilka nietypowych dla HTML znaczników użytych szablonie określającym layout. Znacznik otoczone podwójnymi nawiasami `{{...}}` są to znacznik języka szablonów *Spacebars*[5]. Język ten zostanie omówiony w późniejszej części pracy. Z punktu widzenie routera istotne są znaczniki `Template.dynamic`. W miejsca oznaczone tymi znacznikami zostaną wstrzyknięte odpowiednie szablony. Za to będzie odpowiedzialny sam router, a dokładanie `BlazeLayout.render`. Listing 3.16 przedstawia definicję przykładowego routingu dla aplikacji.

Listing 3.16: Definicja przykładowego routingu

```
FlowRouter.route('/projectSummary/:projectId', {
    action: function (params, queryParams) {
        BlazeLayout.render('mainDashLayout', MyApp.mainDashRegions('
    projectMain'));
    },
    name: 'projectMain'
});

```

[5]https://github.com/meteor/meteor/tree/devel/packages/spacebars

```
8  FlowRouter.route('/projectWiki/:projectId', {
9      action: function (params, queryParams) {
10         BlazeLayout.render('mainDashLayout', MyApp.mainDashRegions('
       mainWiki'));
11     },
12     name: 'projectWiki'
13 });
14
15 FlowRouter.route('/projectWiki/:projectId/:category/:articleId', {
16     action: function (params, queryParams) {
17         BlazeLayout.render('mainDashLayout', MyApp.mainDashRegions('
       wikiArticle'))
18     },
19     name: 'projectWikiArticle'
20 });
21
22 FlowRouter.route('/projectWiki/:projectId/:category', {
23     action: function (params, queryParams) {
24         BlazeLayout.render('mainDashLayout', MyApp.mainDashRegions('
       wikiCategory'));
25     },
26     name: 'projectWikiCategory'
27 });
28
29 FlowRouter.route('/projectConversation/:projectId', {
30     action: function (params, queryParams) {
31         BlazeLayout.render('mainDashLayout', MyApp.mainDashRegions('
       projectTalks'));
32     },
33     name: 'projectTalks'
34 });
```

Poprzez wywołanie funkcji `FlowRouter.route` definiujemy trasę. Funkcja przyjmuje dwa parametry, pierwszy z nich to ciąg znaków określający *url* danej ścieżki, drugi to obiekt konfiguracyjny dla trasy. W tym przypadku obiekt ten ma dwie właściwości: `action` — określa co się stanie po wejściu na daną ścieżkę oraz `name` — używany np. do określenia aktywnej ścieżki. Wewnątrz ciała funkcji, która jest zdefiniowana we właściwości `action` do renderowania poszczególnych szablonów używana jest funkcja `BlazeLayout.render`. Jako pierwszy parametr przyjmuje ona ciąg znaków określający jaki layout ma zostać użyty drugi parametr to obiekt tworzony przez funkcję pomocniczą `MyApp.mainDashRegions`. Jako parametr przyjmuje ona ciąg znaków określający nazwę szablonu, który zostanie wstrzyknięty w miejsce określone przez wyrażenie `{{> Template.dynamic template=content}}` w layoucie (szablonie) `mainDashLayout` — listing 3.15. Reszta regionów definiowanych przez inne wyrażenia `Template.dynamic` zostanie utworzona przez funkcję pomocniczą przedstawioną na listingu 3.17.

Listing 3.17: Funkcja pomocnicza — definicja regionów

```
1 MyApp._mainDashRegions = {
2     header: "mainDashHeader",
```

```
    sideBar: "mainDashSideBar",
    footer: "mainDashFooter",
    controlSideBar: "mainDashControlSideBar"
};
MyApp.mainDashRegions = function (contentTemplate) {
    var regions;
    if (contentTemplate) {
        check(contentTemplate, String);
        regions = _.extend({content: contentTemplate}, this.
    _mainDashRegions)
    } else {
        regions = _.clone(this._mainDashRegions)
    }
    return regions
};
```

Na listingu 3.16 widzimy także użycie parametrów – wyrazy poprzedzone „:” — w ciągach znaków użytych do definicji *url* w funkcji `FlowRouter.route`. Router traktuje je w specjalny sposób przy dopasowywaniu ścieżek. Router umożliwia także pobranie jego wartości. Listing 3.18 przedstawia przykładowe sposoby pobierania takich wartości.

Listing 3.18: Pobieranie parametrów z routera

```
// ścieżka /projectSummary/:projectId
self.project = function () {
    FlowRouter.watchPathChange();
    let context = FlowRouter.current();
    return Project.findOne({_id: context.params.projectId})
};

// lub:
let projectId = FlowRouter.getParam("projectId");
console.log(projectId);
```

3.6.2 Subskrypcje

Jak już wspomniano aby klient miał dostęp do zestawu danych musi być zdefiniowana publikacja określająca dany zestaw oraz klient musi dokonać subskrypcja takiej publikacji. Listing 3.19 obrazuje przykładowe subskrypcje.

Listing 3.19: Podstawowe subskrypcje

```
// publikacja "chat"
Meteor.subscribe("chat", {room: Session.get("current-room")});

// publikacja "privateMessages"
Meteor.subscribe("privateMessages");
```

Jak widać na listingu do subskrypcji możemy użyć metody `Meteor.subscribe`, która przyjmuje jako pierwszy parametr nazwę publikacji a następne parametry zostaną przekazane do funkcji zwrotnej zdefiniowanej w danej publikacji. Inny sposobem jest wykorzystanie obiektu `Blaze.TemplateInstance`. Listing 3.20 pokazuje wykorzystanie drugiego sposobu.

Listing 3.20: Subskrypcje na poziomie szablonu

```
Template.projectTalks.onCreated(function () {
    let self = this;

    self.autorun(() => {

        self.projectId = () => {
            FlowRouter.watchPathChange();
            let context = FlowRouter.current();
            return context.params.projectId
        };

        self.subscribe('userProjects');
        self.subscribe('talks', {'project.id': self.projectId()});


        if (self.subscriptionsReady()) {
            self.project = function () {
                return Project.findOne({_id: self.projectId()})
            };

            if (!self.project()) {
                FlowRouter.go('mainDash');
            }
            self.talks = () => {
                return Talks.find({'project.id': self.project()._id, status
    : "publish"}, {sort: {createdAt: -1}})
            };
            self.draftTalk = () => {
                return Talks.findOne({
                    'author.id': Meteor.user()._id,
                    'project.id': self.project()._id,
                    status: "draft"
                })
            };
        }


    })
});
```

Wykorzystując ten sposób subskrypcji musimy dokonać podczas tworzenia szablonu z wykorzystaniem wywołania funkcji `onCreated`. Jako parametr przekazujemy funkcję zwrotną która zostanie wywołana podczas tworzenia szablonu. Jak widzimy na listingu 3.20 tak samo wykorzystujemy `subscribe` będą właściwością obiektu `this`. W tym kontekście w trakcie tworzenia szablonu będzie to obiekt `Blaze.TemplateInstance`. W szablonie `projectTalks` subskrybowane są dwie publikacje: `userProjects` oraz `talks`. W przedstawiony kodzie wykorzystujemy jeszcze jedną ciekawą funkcję `autorun` związaną z obiektem `Blaze.TemplateInstance`. Funkcja te jest pochodną funkcji `Tracker.autorun`. Mechanizm użyty w tych funkcjach odpowiada za reaktywność aplikacji i jest dostarczany przez bibliotekę `Deps`. Funkcja `autorun` przyjmuje jeden parametr funkcję która zostanie przeliczona za każdym razem gdy zmieni się jakakolwiek

zależność umieszczona w jej ciele. Przez zależności rozumiemy *reaktywne źródła danych*. W przypadku pokazanym w listingu za takie reaktywne źródła danych uznajemy subskrypcje (linie 12 i 13) oraz wywołanie `FlowRouter.watchPathChange()` (linia 7), które to tworzy takie źródło. Każda zmiana subskrypcji oraz zmiana parametru w adresie `url` spowoduje ponowne przeliczenie funkcji przekazanej do `autorun`. Ten mechanizm jest jedną z najważniejszych zalet tego frameworka.

3.6.3 Szablony

Szablony wykorzystywane przez framework składają się z dwu części: cześć z kodem HTML umieszczonym pomiędzy elementami `<template>` oraz opcjonalnej część JavaScriptowej. W części HTML możemy umieszczać znaczniki języka szablonów `Spacebars`. Szablony mogą być używane wielokrotnie w tym samy kodzie HTML. W ostatnim czasie prężnie rozwija się intergracja Meteora z innymi frameworkami JavaScriptowymi takimi jak *AngularJS*[6] oraz *React*[7].

HTML

W części HTML szablonów możemy wykorzystywać dowolne elementy HTML oraz dodatkowo możemy skorzystać z języka szablonów `Spacebars`, które zostały stworzone na podstawie języka szablonów `Handlebars`. `Spacbars` są domyślnie instalowane przy tworzeniu nowej aplikacji. Dostarczają nam dodatkowych znaczników umieszczanych w podwójnych nawiasach wąsatych `{{ }}`. Listing 3.21 przedstawia przykładowe użycie `Spacebars`.

Listing 3.21: *Spacebars*

```
1  <template name="myPage">
2    <h1>{{pageTitle}}</h1>
3
4    {{> nav}}
5
6    {{#each posts}}
7      <div class="post">
8        <h3>{{title}}</h3>
9        <div class="post-content">
10         {{{content}}}
11       </div>
12     </div>
13   {{/each}}
14 </template>
```

W listingu pokazano wykorzystanie kilku znaczników:

[6] http://www.angular-meteor.com
[7] https://github.com/reactjs/react-meteor

- `{{pageTitle}}` — podwójne nawiasy wąsate używane są do wstawiania ciągów znaków, tak wstawiany tekst jest oznaczany jako bezpieczny — czyli pomimo obecności znaków < w tekście niezostanie stworzony żaden znacznik HTML;
- `{{> nav}}` — tag służący do wstawiania innych szablonów rozpoznawanych po nazwie;
- `{{#each}}` — tag blokowy posiadający blok treści, tagi `#if`, `#each`, `#with` oraz `#unless` są wbudowane, możliwe jest także definiowanie swoich własnych tagów blokowych, tagi `#each` oraz `#with` tworzą nowy kontekst danych dla swoich zawartości, w przykładzie `{{tile}}` i `{{content}}` odnoszą się do właściwości pojedynczego postu;
- `{{{content}}}` — potrójne nawiasy wąsate używane są do wstawiania treści HTML, które nie są określane jako bezpieczne — wstawiany jest surowy kod HTML.

JS

autorun

3.6.4 Eventy

3.6.5 Helpery

3.6.6 Pliki less

że są zbierane w całość i łączone w jeden duży plik.

3.7 Objekt MyApp

po co go się stosuje

3.8 Paczki powtstałe na potrzeby aplikacji

yp2:admin-lte@2.3.1 yp2:confirm-modal-bs3@1.1.1 yp2:hijack-email@1.0.0 yp2:yfform@0.3.10 plus informacje że są one dostępne na Atmosfer i githubie

Rozdział 4

Wdrożenie

Rozdział 5

Podsumowanie

Bibliografia

[1] Mike Cantelon i Marc Harter i TJ Holowaychuk i Nathan Rajlich. *Node.js w akcji*. Wydawnictwo Helion, Gliwice, 2014.

[2] Meteor. Meteor Docs. `http://docs.meteor.com/#/full/quickstart`, Grudzień 2015.

[3] Stoyan Stefanov. *JavaScript programowanie obiektowe*. Wydawnictwo Helion, Gliwice, 2010.

[4] Isaac Strack. *Getting Started with Meteor.js JavaScript Framework*. Packt Publishing Ltd., Birmingham, United Kingdom, 2015.

[5] Wikipedia. Bootstrap (front-end framework). `https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)`, Grudzień 2015.

[6] Wikipedia. ECMAScript. `https://en.wikipedia.org/wiki/ECMAScript`, Listopad 2015.

[7] Wikipedia. First-class function. `https://en.wikipedia.org/wiki/First-class_function`, Listopad 2015.

[8] Wikipedia. HTML5. `https://pl.wikipedia.org/wiki/HTML5`, Grudzień 2015.

[9] Wikipedia. Intranet. `https://en.wikipedia.org/wiki/Intranet`, Listopad 2015.

[10] Wikipedia. JavaScript. `https://en.wikipedia.org/wiki/JavaScript`, Listopad 2015.

[11] Wikipedia. Kaskadowe arkusze stylów. `https://pl.wikipedia.org/wiki/Kaskadowe_arkusze_stylów#CSS_3`, Grudzień 2015.

[12] Wikipedia. Meteor (web framework). `https://en.wikipedia.org/wiki/Meteor_(web_framework)`, Grudzień 2015.

[13] Wikipedia. MongoDB. https://pl.wikipedia.org/wiki/MongoDB, Grudzień 2015.

[14] Wikipedia. Representational State Transfer. https://pl.wikipedia.org/wiki/Representational_State_Transfer, Grudzień 2015.

[15] Wikipedia. WebSocket. https://pl.wikipedia.org/wiki/WebSocket, Grudzień 2015.