

UNIWERSYTET GDAŃSKI
Wydział Matematyki, Fizyki i Informatyki

Oskar Plichta

nr albumu: 195009

Budowa aplikacji modularnej do udostępniania fotografii w web 3.0

Praca magisterska na kierunku:

INFORMATYKA

Promotor:

dr W. Bzyl

Gdańsk 2015

Streszczenie

W pracy zostanie przedstawiony program do udostępniania fotografii na kilka serwisów społecznościowych jednocześnie z intuicyjnym interfejsem Material Design. Dzięki temu w prosty i szybki sposób można podzielić się swoimi zdjęciami z innymi użytkownikami kilku sieci społecznościowych. Aplikacja będzie wykonana z dwóch modułów - części serwerowej i wizualnej. Część serwerowa jest oparta o *Ruby on Rails* a wizualna wykonana w *EmberJS* oraz *Materialize*. Aplikacje mobilna dla systemu Android wykonane zostały dzięki aplikacji *Cordova*, która pozwala przekształcić aplikację opartą o *EmberJS* oraz *Ember CLI* w natywną aplikację dla danego systemu mobilnego. Ostatecznie aplikacja została wykonana zgodnie z założeniami i spełnia wyznaczone cele.

Słowa kluczowe

User Interface, Material Design, Ruby on Rails, EmberJS, PostgreSQL, RSpec, QUnit

Spis treści

Wprowadzenie	5
1. Budowa aplikacji modularnej do udostępniania fotografii w web 3.0	7
1.1. Porównanie dostępnych rozwiązań	7
1.2. Możliwości zastosowania praktycznego	8
2. Projekt i analiza	9
2.1. Aktorzy i przypadki użycia, wymagania funkcjonalne i niefunkcjonalne	9
2.2. Diagram klas	10
2.3. Diagram modelu danych	10
2.4. Projekt interfejsu użytkownika w oparciu o framework Materialize	10
2.4.1. Wytyczne odnośnie UI w Material Design	11
3. Architektura aplikacji PicDrop	12
3.1. Użyte technologie	12
3.2. API aplikacji	13
3.2.1. Budowa API w oparciu o Ruby on Rails	13
3.2.2. Połączenie z bazą danych PostgreSQL	15
3.2.3. Dodawanie gemów w Ruby on Rails	16
3.2.4. Autoryzacja użytkowników	17
3.2.5. Połączenie z API różnych sieci społecznościowych	18
3.3. Front-end aplikacji	20
3.3.1. Budowa Front-endu w oparciu o framework EmberJS	20
3.3.2. Połączenie z API	21
3.3.3. Opis narzędzia Ember-CLI	22
3.3.4. Dodawanie wtyczek do aplikacji w EmberJS	22
3.3.5. Same origin policy oraz Cross-origin resource sharing	22

4. Testy	25
4.1. Testowanie API przy użyciu RSpec	25
4.1.1. Raport z testów API	25
4.2. Testowanie Front-end przy użyciu QUnit	25
4.2.1. Raport z testów Front-end	26
5. Wkład własny	27
Zakończenie	28
A. Tytuł załącznika jeden	29
B. Tytuł załącznika dwa	30
Bibliografia	31
Spis tabel	32
Spis rysunków	33
Oświadczenie	34

Wprowadzenie

Fotografie są jednym z najczęstszych typów danych przesyłanych w Web 3.0. Gwałtowny rozrost sieci społecznościowych spowodował, że prawie każdy udostępnia zdjęcia aby podzielić się nimi z rodziną i przyjaciółmi. Temat wysyłania zdjęć do kilku serwisów jednocześnie, tak aby nie jeszcze raz nie powtarzać tej samej czynności na innym serwisie społecznościowym, pozostaje otwarty i dlatego postanowiłem go zgłębić.

Portale takie jak Facebook, Flickr czy Twitter prześcigają się w tym aby wysyłanie zdjęć na ich serwer było jak najprostsze. Większość z nich pozwala na tzw. *drag and drop*¹ fotografii oraz na wysyłanie ich do innych serwisów. Zakładając, że mamy zdjęcia z wakacji i chcemy je udostępnić na Facebooku oraz umieścić na naszym koncie Flickr w celu archiwizacji musimy zalogować się na Facebooka, następnie wysłać zdjęcia na serwer Facebooka, ewentualnie dopisać opis i kliknąć w przycisk do udostępniania, po czym całość powtórzyć na serwisie Flickr. Aplikacja, którą opisuje w tej pracy pozwala na jednoczesne wysyłanie zdjęć na kilka serwisów społecznościowych za pomocą kilku kliknięć. Wystarczy wybrać zdjęcia dzięki *drag and drop*, zalogować się na wybrane przez nas serwisy a następnie kliknąć wyślij. Wszystko przebiega szybko i sprawnie a my oszczędzamy nasz czas. Aplikacja ta jest tzw. aplikacją modułową co znaczy, że w przeciwieństwie do aplikacji monolitycznej, składa się z niezależnych od siebie części tj. serwerowej oraz wizualnej. Obie części komunikują się ze sobą za pomocą wiadomości JSON. Część serwerowa odpowiada za komunikację z bazą danych, komunikację z serwerami zewnętrznymi oraz autoryzację użytkowników i wysyłanie danych do front-endu czyli aplikacji wizualnej. Zostanie ona wykonana w języku Ruby i frameworku *Ruby on Rails*. Aplikacja wizualna, która zostanie wykonana w języku JavaScript i frameworku *EmberJS*, ma za zadanie wyświetlanie danych w przystępnej formie dla użytkownika poprzez tzw. interfejs. Omówię zagadnienie aplikacji modularnej i monolitycznej bardziej szczegółowo w jednym z kolejnych rozdziałów. Interfejs

¹ang. *drag and drop* - przeciągnij i upuść

użytkownika² jest podstawowym sposobem komunikacji pomiędzy człowiekiem a maszyną dlatego tak ważne jest, aby był on intuicyjny i przyjazny dla użytkownika. Postaram się pokazać dlaczego UI w mojej aplikacji jest przyjazny, intuicyjny dla użytkownika i pozwala mu na wydajną pracę a wszystko dzięki wytycznym Google Material Design, które jest nowym designem dla aplikacji od firmy Google. Więcej na temat Material Design oraz interfejsu aplikacji PicDrop w kolejnych rozdziałach.

Większość smartphonów posiada dobrej jakości aparaty, zarówno z przodu jak i z tyłu urządzenia, dlatego w każdej chwili możemy wysłać zdjęcia z dowolnego miejsca do rodziny i przyjaciół poprzez sieci społecznościowe, za pomocą natywnej aplikacji na dany system mobilny. Dzięki temu, że moja aplikacja jest modułowa, jej część serwerowa czyli API pozwala na połączenie z nią różnych części wizualnych tzw. front-end z systemów mobilnych takich jak np. iOS lub Android. Jest to możliwe dzięki aplikacji Cordova, która zamienia aplikację *EmberJS* w natywną aplikację na dany system mobilny przez co jest ona łatwiejsza w obsłudze i lepiej wykorzystuje mały wyświetlacz smartphona niż aplikacja przeglądarkowa. Opiszę dokładnie ten proces w odpowiednim rozdziale oraz podam inne zalety aplikacji natywnej. Wskażę również z jakimi problemami musi się uporać *developer* aplikacji webowych, aby jego aplikacja była intuicyjna i funkcjonalna. Wyjaśnię, które elementy mojej aplikacji testuję i w jaki sposób. Opierając się na doświadczeniach innych badaczy między innymi Roberta Hoekmana jr [1] oraz Jenifer Tidwell [2], którzy opisali swoje spostrzeżenia w ich książkach, postaram się napisać aplikację PicDrop, która będzie miała przyjazne UI i pozwoli na łatwe udostępnianie treści. Opiszę dlaczego wybrałem *EmberJS*, *MaterializeCSS* oraz *Ruby on Rails* do stworzenia tej aplikacji i dlaczego te technologie uważam za najlepszy wybór.

²ang. *User Interface* - UI

Budowa aplikacji modularnej do udostępniania fotografii w web 3.0

1.1. Porównanie dostępnych rozwiązań

Każdy serwis społecznościowy posiada swoje możliwości udostępniania zdjęć, które ograniczają się do jednego serwisu. Tak więc jeśli chcemy wgrać swoje zdjęcia jednocześnie na Facebooka, żeby zobaczyli je nasi znajomi oraz Flickr w celu archiwizacji to musimy je wgrać na jeden z tych dwóch serwisów a następnie na kolejny. Takie rozwiązanie zajmuje dużo czasu i jest niekorzystne dla użytkownika. Serwis Facebook, który jest najpopularniejszym serwisem społecznościowym, zrzesza ponad 500 milionów użytkowników. Udostępnianie zdjęć poprzez ten serwis polega na kliknięciu w ikonkę zdjęcia i wybraniu danego pliku, bądź przeciągnięciu i upuszczeniu pliku w odpowiednim miejscu. Możemy dopisać, krótki tekst i po naciśnięciu guzika wyślij, post pojawi się na naszej tablicy. Facebook nie pozwala na udostępnianie zdjęć poza swoją stroną, także jeżeli mamy znajomych nie posiadających konta na tym serwisie to nie zobaczą oni naszych zdjęć.

Flickr jest serwisem dla miłośników fotografii, który daje nam, aż 1 TB na przechowywanie naszych zdjęć. Pozwala on również na udostępnianie plików także do innych serwisów społecznościowych. Umieszczanie zdjęć w tym serwisie jest podobne do rozwiązania stosowanego w Facebooku, to znaczy albo wybieramy pliki albo przeciągamy je do okna przeglądarki. Po wgraniu ich na serwer możemy je udostępnić dalej poprzez wybranie zdjęć. Aktualnie nie ma serwisu, który by pozwalał na wysyłanie zdjęć do dwóch serwisów jednocześnie. Z tego powodu postanowiłem stworzyć aplikację, która zmienia ten stan rzeczy.

1.2. Możliwości zastosowania praktycznego

Głównym celem aplikacji *PicDrop* jest proste i intuicyjne udostępnianie fotografii. Pierwszym krokiem jest założenie konta w aplikacji lub zalogowanie przy użyciu adresu email oraz hasła. Następnie użytkownik może wgrać własne fotografie poprzez przeciągnięcie do okna przeglądarki, gdzie zobaczy miłą dla oka animację pulsującego koła. Dostępny jest również estetycznie wyglądający guzik w kształcie koła z plusem na środku. Gdy wybierze zdjęcia pojawią się ich miniatury. Następnie wystarczy, że użytkownik kliknie guzik z nazwą portalu gdzie chce udostępnić swoje fotografie i zaloguje się na wybrany portal społecznościowy. Później jedno kliknięcie wysła asynchronicznie wszystkie zdjęcia na serwery *Flickr* i *Facebooka*. To wszystko. Obsługa jest szybka i intuicyjna. Gdybyśmy stwierdzili, że nie chcemy wysłać jednego ze zdjęć możemy je usunąć klikając ikonę kosza. Zdjęcia można powiększać, klikając na nie. Funkcja ta może być szczególnie przydatna gdy zdjęcie ma dużo detali a my chcemy mu się dokładniej przyjrzeć. Można także udostępnić pojedyncze zdjęcie klikając na odpowiednią ikonę. Aplikacja ma piękny design oparty o *Material Design* od firmy *Google* dlatego korzystanie z niej jest miłe dla oka. Dzięki tej aplikacji możemy zaoszczędzić czas gdy chcemy szybko udostępnić zdjęcia znajomym a jednocześnie zapisać je na serwerach dla archiwizacji. Portal *Flickr* od firmy *Yahoo* zapewnia każdemu użytkownikowi 1 TB czyli 1024 GB miejsca na zdjęcia za darmo. Taka ilość przestrzeni powinna zaspokoić potrzeby najbardziej wymagających użytkowników.

Projekt i analiza

2.1. Aktorzy i przypadki użycia, wymagania funkcjonalne i нефункционалне

Aplikacja wczytuje zdjęcia z bazy danych PostgreSQL, wysyła je przez JSON do klienta i tam EmberJS odpowiednio obrabia dane pokazując je w formie przyjaznej użytkownikowi. Następnie, gdy chcemy udostępnić jakiś plik to jest to obsługiwane także przez EmberJS, tak aby komunikacja była szybka i niezawodna. Jednakże jeśli użytkownik wczytuje własne zdjęcia, które chce udostępnić to są one zamieniane na ciąg znaków w standardzie Base64Image i wysyłane wiadomością JSON do API. Tam back-end łączy się z wybranym portalem społecznościowym i przesyła do niego zdjęcia.

Wymagania funkcjonalne aplikacji to:

- Użytkownik będzie mógł udostępniać fotografie.
- Użytkownik będzie mógł się zalogować.
- Użytkownik będzie mógł wybrać serwis na jaki fotografie zostaną wysłane.
- Użytkownik będzie mógł usunąć wybrane fotografie

Wymagania нефункционалне aplikacji to:

- Wszelka niezbędna komunikacja między serwerem a aplikacją kliencką powinna dać się wyrazić za pomocą wiadomości *JSON*
- Interfejs aplikacji powinien być zgodny z wytycznymi *Material Design*.
- Aplikacja nie powinna udostępniać osobom trzecim danych użytkowników zapisanych w bazie danych.

2.2. Diagram klas

2.3. Diagram modelu danych

PicDrop posiada trzy modele danych. Są to *User*, *Photo* oraz *Authorization*. Pierwszy z nich służy do obsługi danych użytkownika i logowania do aplikacji. Zawiera on pola do zapisywania email, hasła, potwierdzenia hasła, tokenu użytkownika oraz jego nazwę. Drugi natomiast jest modelem dla danych o wysyłanych zdjęciach, czyli na przykład nazwa, adres url oraz nazwa właściciela. Ostatni model przechowuje dane z kont społecznościowych. Zapisywane są w nim loginy, hasła oraz tokeny autoryzacji użytkowników.

2.4. Projekt interfejsu użytkownika w oparciu o framework Materialize

Interfejs użytkownika musi być czytelny i łatwy w obsłudze a jednocześnie pozwalać na wydajną pracę. W aplikacji *PicDrop* główny nacisk położyłem na intuicyjność i szybkość działania. Do stworzenia designu aplikacji użyłem *frameworka MaterializeCSS* opartym o *Material Design* o którym więcej w kolejnym podrozdziale. *MaterializeCSS* pozwala na używanie komponentów webowych, z których chcemy zbudować naszą aplikację. Do wyboru mamy wiele różnych typów menu, kart, guzików, formularzy, paneli nawigacyjnych. Proponowana jest także paleta barw, tak aby strona spełniała założenia *Material Design*. Framework ten odpowiada także za łatwe rozmieszczenie komponentów na stronie. Dzieje się tak poprzez tak zwany *Grid*, który dzieli stronę na 12 kolumn co pozwala na zawijanie wierszy na urządzeniach mobilnych bez utraty treści i konieczności przewijania strony. *UI* aplikacji jest także funkcjonalne. Dobrym przykładem jest wybieranie fotografii za pomocą upuszczenia ich w oknie przeglądarki internetowej. Głównymi komponentami użytymi w mojej aplikacji jest panel nawigacyjny u góry oraz karty z miniaturkami zdjęć. Poza tym użyłem pól formularzy oraz guzików, które po naciśnięciu wyświetlają tak zwaną falę, przez co użytkownik wie, że dany guzik został naciśnięty.

2.4.1. Wytyczne odnośnie UI w Material Design

Google Material Design został zaprezentowany pierwszy raz na Google I/O w 2014 roku wraz z systemem Android i jego wersją Lollipop. Dzięki temu wszystkie aplikacje na platformę od *Google'a* posiadają spójny wygląd. Wytyczne przedstawiają paletę barw, typografię, ikony, bardziej spójną hierarchię interfejsu oraz wszelkie zasady co i jak powinno się tworzyć. Mimo, że od tamtego wydarzenia minęło już trochę czasu to jest on używany przez Google w kolejnych edycjach systemu Android. Głównymi założeniami tego designu jest to że składa on się z kilku nałożonych na siebie warstw papieru w różnych kolorach. Dzięki temu mamy wrażenie głębi a wszystko jest przejrzyste i intuicyjne. Dodatkowo animacje w *Material Design* dopasowują się do kolejnej wytycznej czyli do takiego animowania elementów na ekranie, żeby użytkownik miał najważniejsze elementy cały czas na widoku, żeby te nie pojawiały się znikąd.

ROZDZIAŁ 3

Architektura aplikacji PicDrop

Aplikacja składa się z dwóch części. Pierwszą z nich jest serwer API, które zostało stworzone w *frameworku Ruby on Rails*. API jest pośrednikiem między bazą danych *PostgreSQL* a drugą częścią aplikacji czyli klientem stworzonym w *EmberJS*, który jest odpowiedzialny za warstwę wizualną aplikacji. Obie części komunikują się poprzez *JSON*. Dzięki takiemu rozwiązaniu można stworzyć dodatkowych klientów, na przykład do aplikacji mobilnych.

3.1. Użyte technologie

Część serwerowa aplikacji może zostać wykonana w wielu językach programowania takich jak *Ruby*, *Python*, *JavaScript* czy *PHP*. Wybrałem język *Ruby*, ponieważ jest on szybkim językiem skryptowym oraz posiada stabilny i sprawdzony *framework Ruby on Rails*, który pozwala na szybkie tworzenie aplikacji internetowych oraz dzięki narzędziu *RSpec* na łatwe testowanie kodu. Spośród mnóstwa technologii do tworzenia interfejsów użytkownika, najbardziej przodujące są oparte te na języku *JavaScript* takie jak *Bootstrap*, *jQuery* czy *AngularJS*. Opierając się na artykułach [1] oraz [2] postanowiłem wybrać *EmberJS*, *Materialize* oraz *Ruby on Rails*. UI zostanie wykonane w *EmberJS*, jest to biblioteka *open-source* języka *JavaScript* stworzona przez *Yehuda Katz’a* i *Tom’a Dale’a*, która jest cały czas usprawniana przez społeczność. Posiada ona szereg mechanizmów ułatwiających developerom tworzenie *UI* na jej podstawie, ma także czytelną i przejrzystą dokumentację. UI aplikacji zostanie dodatkowo upiękkszzone poprzez *framework Materialize*, a *QUnit* pozwala na testowanie kodu *JavaScript* w *Emberze*.

3.2. API aplikacji

3.2.1. Budowa API w oparciu o Ruby on Rails

API czyli *Application Programming Interface* to ściśle określony zestaw reguł i ich opisów, w jaki programy komputerowe komunikują się między sobą. W przypadku *Ruby on Rails* główną regułą jest komunikacja przy pomocy *JSON*, czyli formatem tekstowym, bazującym na podzbiorze języka *JavaScript*, niezależnym od konkretnego języka. Jest używany między innymi w językach *C++*, *Java*, *JavaScript*, *Ruby*, *PHP* i *Phyton*. Przykładowa wiadomość *JSON* wygląda następująco:

```
{
  :provider => 'facebook',
  :uid => '1234567',
  :info => {
    :nickname => 'jbloggs',
    :email => 'joe@bloggs.com',
    :name => 'Joe Bloggs'
  },
  :credentials => {
    :token => 'ABCDEF',
    :expires_at => 1321747205,
    :expires => true
  }
}
```

Ruby on Rails opiera się na modelu MVC¹ jak sama nazwa wskazuje składa się z 3 elementów.

- Modele

Model reprezentuje dane aplikacji i reguły do manipulowania tymi danymi. W przypadku *Ruby on Rails*, modele są głównie wykorzystywane do zarządzania zasadami interakcji z odpowiednią tabelą bazy danych. W większości przypadków jednej tabeli bazy danych odpowiada jeden model. Większość logiki biznesowej aplikacji będzie skoncentrowana w modelach.

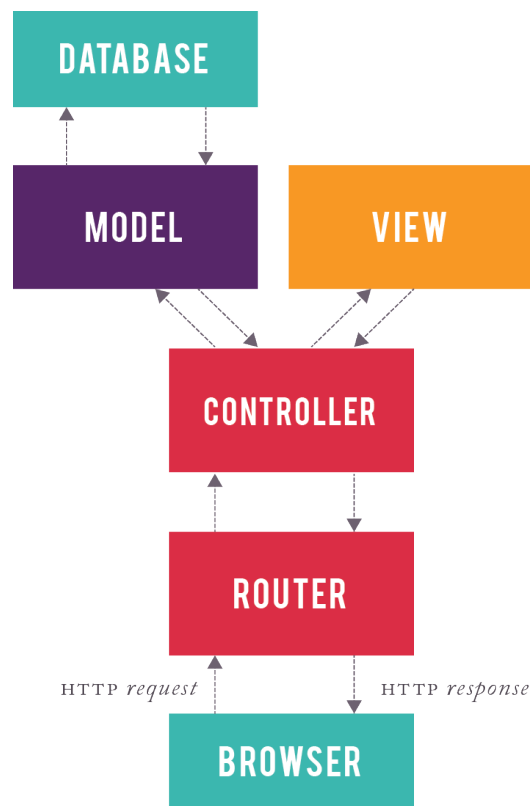
- Widoki

Widoki tworzą interfejs użytkownika aplikacji. W *Ruby on Rails* widoki są często plikami HTML zawierającymi kod w języku Ruby, wykonujący zadania związane wyłącznie z prezentacją danych. Przede wszystkim są one odpowiedzialne za dostarczanie danych do przeglądarki internetowej lub innego narzędzia, które jest używane do oglądania efektów działania aplikacji. W przypadku gdy nasza aplikacja jest *API* widoki nie występują, gdyż zamiast nich kontroler wysyła wiadomości *JSON* do komunikacji z *Front-end*, czyli *Ember.js*

- Kontrolery

Kontrolery łączą modele i widoki. W *Ruby on Rails* odpowiadają za przetwarzanie żądań przychodzących z przeglądarki internetowej, pozyskiwanie danych z modeli i przekazywanie ich do widoków w celu ich prezentacji.

¹ang. *Model-View-Controller* - Model-Widok-Kontroler



Rysunek 3.1: Html-reporter

Źródło: <https://robots.thoughtbot.com/shared-terminology-yet-different-concepts-between-emberjs-and-rails>

3.2.2. Połączenie z bazą danych PostgreSQL

PostgreSQL jest SQL-ową bazą danych, rekomendowaną dla *Ruby on Rails*, przez swoją ścisłą integrację z wcześniej wymienionym *frameworkem* oraz posiada dobre narzędzia i metody zarządzania. Standardową bazą danych w aplikacjach stworzonych w *Ruby on Rails* jest *SQLite*, jednakże nie jest ona wspierana przez *Heroku*, który jest najpopularniejszym darmowym serwerem dla aplikacji w *Ruby on Rails*. Brak wsparcia jest spowodowany tym, że *SQLite* działa głównie w pamięci, i tylko od czasu do czasu zapisuje dane do pliku. O ile takie rozwiązanie jest dobre dla małych lokalnych aplikacji, których dane mogą być łatwo przenoszone, nie sprawdza

się na serwerach *Heroku*, gdyż w darmowej wersji, nasza aplikacja może działać maksymalnie przez 18 godzin na dobę po czym jest wyłączana. Takie nagłe wyłączenie spowodowałoby utratę całej bazy danych. Drugim ważnym aspektem jest skalowanie aplikacji. Większy program może zostać uruchomiony na większej ilości zasobów sprzętowych tak zwanych *dyno*, a baza danych *SQLite* może pracować tylko na jednym procesie, tak więc jeśli będzie więcej procesów każdy z nich będzie posiadać własną bazę danych a dane nie będą synchronizowane prawidłowo. W związku z powyższym powinniśmy używać *PostgreSQL* na etapie produkcyjnym naszej aplikacji.

Zmiana bazy danych z *SQLite* na *PostgreSQL* nie jest zbyt trudna. Pierwszym krokiem jest zainstalowanie bazy na komputerze, na którym ma być uruchomiona aplikacja. Na oficjalnej stronie można znaleźć linki do pobrania. Po zainstalowaniu, musimy dodać do naszej aplikacji gem 'pg' który zapewnia kompatybilność z bazą *PostgreSQL*, a usuwamy gem 'sqlite3'. Następnie modyfikujemy plik *database.yml*, który jest plikiem konfiguracyjnym naszej aplikacji. Ostatnim krokiem jest utworzenie bazy danych na nowym środowisku przy pomocy komendy `rake db:create db:migrate`

3.2.3. Dodawanie gemów w Ruby on Rails

Jedną z największych zalet Ruby on Rails jest dopasowanie się frameworka do naszych potrzeb za pomocą minibibliotek tzw. gemów, tworzonych przez developerów z całego świata. W sieci można znaleźć tysiące różnych gemów, a każdy z nich posiada przeważnie tylko jedno specyficzne zadanie. Gemy, które są w ciągłym użytku są na bieżąco aktualizowane, dzięki czemu są poprawiane błędy i dodawane nowe funkcje. Przykładowo jeżeli chcemy testować naszą aplikację, możemy to zrobić dzięki gemowi *RSpec*. Pozwala on na testowanie różnorodnych elementów naszej aplikacji od walidacji pól w formularzach do kontrolerów i zapisywania danych w bazie.

Aby dodać nowy gem do naszej aplikacji wystarczy dodać jego nazwę do pliku *Gemfile* w głównym katalogu naszej aplikacji go zapisać. Następnie wystarczy w terminalu otworzyć folder z naszą aplikacją i wpisać komendę "bundle install".

Ruby on Rails automatycznie zainstaluje najnowszą wersję danego gemu, chyba, że wpisaliśmy konkretną wersję w pliku Gemfile. Całość przebiega szybko i sprawnie.

3.2.4. Autoryzacja użytkowników

Autoryzacja użytkowników jest jednym z najważniejszych zagadnień w aplikacji PicDrop. Dzięki niej logujemy się do aplikacji jak i do serwisów społecznościowych, na które chcemy umieścić zdjęcia. Bez niej ktoś obcy mógłby korzystać z naszych danych logowania z Facebooka i Flickr bez naszej wiedzy. Cały proces autoryzacji jest dość skomplikowany i przebiega w kilku etapach. Gdy chcemy się zalogować wpisujemy nasz login i hasło a następnie klikamy na guzik Log in. Nasz login jest adresem email i musi być unikatowy, dlatego mamy pewność, że nie będzie dwóch takich samych kont ale z różnymi hasłami, dzięki czemu nikt się nie zaloguje na nasze konto. *Ember.js* pobiera dane z naszego formularza i wysyła je używając wiadomości *JSON* do naszego serwera. Tam aplikacja generuje dla danego użytkownika klucz tzw. token, który jest przechowywany w bazie danych i na nim opiera się tzw. sesja dzięki, której jesteśmy zalogowani nawet jeśli odświeżymy stronę. Token jest odsyłany do *Ember.js* i jesteśmy zalogowani.

Jak pisałem we wcześniejszym rozdziale, aplikacja na serwerze posiada trzy modele - *User*, *Photo* oraz *Authorization*. Model *User* jest zarządzany przez gem *Devise*. Jest to najpopularniejszy gem do uwierzytelniania w Ruby on Rails. Ma wiele funkcji i składa się z 10 modułów dzięki czemu może być dostosowany do potrzeb developera. Wymienię tylko najważniejsze z nich dla mojej aplikacji. Pierwszy z nich to *Database Authenticatable*, który szyfruje i przechowuje hasło w bazie danych to autoryzacji użytkownika podczas logowania. Drugi to *Omniauthable* dodaje on wsparcie dla OmniAuth, który jest gemem do logowania się w aplikacjach *Ruby on Rails* na różne serwisy społecznościowe. Omówię go dokładniej w następnym rozdziale. Następny to *Timeoutable*, który porzuca sesję dla nieaktywnego użytkownika przez sprecyzowany okres czasu. Takie rozwiązanie znacznie zwiększa bezpieczeństwo naszych danych, ponieważ nawet jeśli zostawimy zalogowany komputer to po czasie wygaśnięcia sesji musimy się ponownie zalogować co nie pozwala użytkować naszego konta przez osoby trzecie.

Ember.js do autoryzacji używa biblioteki *Ember Simple Auth* i jej modułu kom-

patybilnego z *Devise*, który nosi nazwę *Ember Simple Auth Devise*. Biblioteka ta zapewnia kompleksowe zarządzanie sesją i logowaniem do aplikacji opartych na *Ember.js*, gdyż posiada wiele modułów dla różnych typów uwierzytelniania. Są to między innymi *simple-auth-oauth2*, który pozwala na logowanie w standardzie *OAuth2*. Standard ten jest używany przez Facebooka i zostanie dokładniej omówiony w jednym z kolejnych rozdziałów. Kolejny moduł to *simple-auth-torii*. Jest to moduł kompatybilny z biblioteką *Torii*, która odpowiada za logowanie w standardzie *OAuth 1a*. Standard ten jest wykorzystywany między innymi przez *Twittera* i *Flickr*.

3.2.5. Połączenie z API różnych sieci społecznościowych

I User Authorization URL: Access Token URL: wysyłanie zdjęć API aplikacji łączy się z różnymi sieciami społecznościowymi przez gem *Omniauth*. Dzięki niemu możemy po zalogowaniu, wysłać nasze zdjęcia na dwa serwisy jednocześnie. W zależności od serwisu społecznościowego, używany jest inny standard protokołu *OAuth*. Pierwszym z nich jest *OAuth 1.0a*, używany przez *Twittera* i *Flickr*. Jest on starszym i bardziej skomplikowanym standardem logowania. Logowanie przebiega w 3 krokach. Pierwszy krok to wysłanie prośby o tak zwany *Request Token*, który zawiera *request oauth token* oraz *request oauth secret* a nasze zapytanie wysła m.in. identyfikator i sekretny kod naszej aplikacji oraz datę i czas zapytania. Wygląda mniej więcej tak:

```
https://www.flickr.com/services/oauth/request_token
?oauth_nonce=89601180
&oauth_timestamp=1305583298
&oauth_consumer_key=653e7a6ecc1d528c516cc8f92cf98611
&oauth_signature_method=HMAC-SHA1
&oauth_version=1.0
&oauth_callback=http%3A%2F%2Fwww.example.com
```

Gdy prośba zostaje przetworzona, serwer odsyła *Request Token*

```
oauth_callback_confirmed=true  
&oauth_token=72157626737672178-022bbd2f4c2f3432  
&oauth_token_secret=fccb68c4e6103197
```

Następnie użytkownik zostaje przekierowany do strony logowania gdzie podaje swoje dane. Po poprawnym zalogowaniu serwer odsyła *oauth verifier*.

```
http://www.example.com/  
?oauth_token=72157626737672178-022bbd2f4c2f3432  
&oauth_verifier=5d1b96a26b494074
```

Ostatnim krokiem jest wysłanie przez naszą aplikację *Request Tokena* oraz *Oauth verifiera* do serwerów autoryzacji w celu uzyskania *Access Tokena* dzięki któremu możemy się zalogować.

```
https://www.flickr.com/services/oauth/access_token  
?oauth_nonce=37026218  
&oauth_timestamp=1305586309  
&oauth_verifier=5d1b96a26b494074  
&oauth_consumer_key=653e7a6ecc1d528c516cc8f92cf98611  
&oauth_signature_method=HMAC-SHA1  
&oauth_version=1.0  
&oauth_token=72157626737672178-022bbd2f4c2f3432  
&oauth_signature=UD9TGXzrvLIb0Ar5ynqvzatM58U%3D
```

W standardzie OAuth 2.0, używany przez między innymi *Facebooka*. Gdy chcemy się zalogować do tego serwisu wystarczy nacisnąć guzik logowania, który przekierowuje nas do adresu naszego API tj. `/users/auth/facebook`. Następnie *Omniauth* łączy się z serwerami *Facebooka* generując mniej więcej taki adres

```
https://www.facebook.com/auth?response_type=code
&client_id=CLIENT_ID&redirect_uri=REDIRECT_URI&scope=photos
```

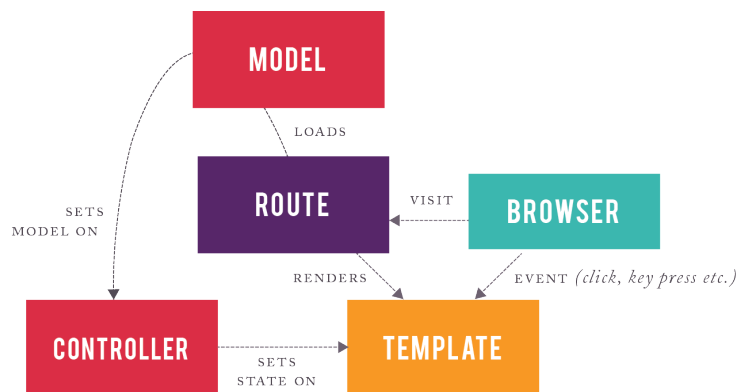
W nim znajduje się prośba o kod autoryzacyjny, identyfikator naszej aplikacji, adres do przekierowania po wpisaniu prawidłowych danych z *Facebooka* oraz uprawnienia aplikacji, na które użytkownik musi zezwolić, żeby się zalogować. Po podaniu prawidłowych danych zostaje odesłany kod autoryzacyjny, który zostaje zamieniony przez *Omniauth* na tak zwany *Access Token* a użytkownik zostaje zalogowany do aplikacji. Aby użytkownicy mogli się logować do naszej aplikacji należy wcześniej ją zarejestrować na stronach dla *web developerów Facebooka* oraz *Flickr*a. Tam podając dane nasze oraz aplikacji zostaje wygenerowany identyfikator. Wpisujemy go w pliku konfiguracyjnym *Omniuth*, aby serwisy wiedziały jaka aplikacja chce się zalogować.

3.3. Front-end aplikacji

3.3.1. Budowa Front-endu w oparciu o framework EmberJS

Warstwa wizualna aplikacji czyli tak zwany front-end został wykonany przy użyciu frameworka EmberJS. Dzięki temu łatwiej zarządzać tą częścią aplikacji. *EmberJS* opiera się na wzorcu *Model-View-Controller* ² co oznacza, że składa się z trzech podstawowych części odpowiedzialnych za różne akcje.

²ang. Model-Widok-Kontroler



Rysunek 3.2: Diagram budowy Ember.js

Źródło: <https://robots.thoughtbot.com/shared-terminology-yet-different-concepts-between-emberjs-and-rails>

Model jest pewną reprezentacją problemu bądź logiki aplikacji. W naszym przypadku model jest szablonem z opisanymi typami danych danego zasobu np. zdjęcia. Model również komunikuje się z Ember Data czyli warstwą aplikacji bezpośrednio komunikującą się z serwerem. Widok opisuje, jak wyświetlić pewną część modelu w ramach interfejsu użytkownika. W EmberJS widok składa się z tak zwanych templates czyli kodem HTML z aktywnie zmieniającymi się częściami. Kontroler przyjmuje dane wejściowe od użytkownika i reaguje na jego akcje, zarządzając aktualizacje modelu oraz odświeżenie widoków. Kontroler w EmberJS pobiera dane z modelu oraz zarządza akcjami w widoku.

3.3.2. Połączenie z API

Połączenie z serwerem aplikacji jest jedną z najważniejszych rzeczy w całej aplikacji. To dzięki niemu możemy się komunikować z serwerem oraz przetwarzać dane z bazy danych. EmberJS do pobierania i przetwarzania danych korzysta z biblioteki Ember Data. Ma ona kilka wbudowanych tzw. adapterów do połączenia z różnymi typami serwerów napisanych w różnych językach i technologiach. W mojej aplikacji używam ActiveModelAdapter, który działa bezproblemowo z ActiveModel w

Ruby on Rails przeprowadzając serializację i deserializację danych z EmberJS do wiadomości JSON.

3.3.3. Opis narzędzia Ember-CLI

Ember-CLI (Command Line Interface) to program narzędziowy, który zarządza aplikacją napisaną we frameworku EmberJS. Głównymi zaletami korzystania z tego narzędzia jest zarządzanie plikami, zależnościami, proste dodawanie wtyczek, uruchamianie serwera, generowanie konkretnych części programu jak np. model lub kontroler.

3.3.4. Dodawanie wtyczek do aplikacji w EmberJS

Wtyczki są to programy dodające jakąś funkcjonalność do naszego programu, dzięki czemu nie musimy wszystkich elementów pisać od podstaw. Wystarczy jak poszukamy wtyczki zapewniającej nam funkcjonalność, której szukamy np. `t17-ember-upload` pozwala na wgrywanie zdjęć do naszej aplikacji poprzez proste *Drag & Drop*³. Aby dodać wtyczkę wystarczy zainstalować ją przy użyciu NPM lub bowera po czym dodać `app.import` w pliku `Brocfile.js`. Następnie należy zaimportować konkretny moduł z wtyczki do naszej części aplikacji i już można używać widoku oraz akcji z wtyczki.

3.3.5. Same origin policy oraz Cross-origin resource sharing

Same Origin Policy uniemożliwia dwóm osobnym kontekstom Javascript modyfikacji swoich drzew DOM. Dzięki temu strona Agresora `aggressive.com` nie może modyfikować DOM strony Banku `bank.com`, w sytuacji kiedy np. te dwie strony są otwarte w kartach przeglądarki. Jednakże jeśli chcemy aby skrypt z jednego kontekstu modyfikowały inny takie zmiany zostaną wykonane. Wszystko dlatego, że strona ma ten sam *Origin* czyli pochodzenie. Na pochodzenie ma wpływ:

- protokół
- host

³pol. Przeciągnij i upuść

- port

Zasada *SOP* jest jednocześnie zbawieniem i przekleństwem. Z jednej strony, jest to mechanizm skutecznie chroniący każdą stronę internetową, z drugiej – ogranicza developerów, którzy chcą np. asynchronicznie pobrać zasób z innej domeny. Aby pomóc developerom firma W3C wprowadziła *Cross-Origin Resource Sharing*⁴. Jest to metoda omijania *SOP* polegająca na odpowiednim ustawianiu nagłówków HTTP. Przeglądarka wysyła do serwera nagłówek "Origin". Mówi w nim z jakiej strony chcemy się dostać do zasobu. Serwer musi zgodzić się na stronę źródłową. Załóżmy, że strona jest pod adresem `http://moja-strona.pl` a API ma adres `http://moje-api.pl/api`. Żądanie przeglądarki będzie wyglądać tak:

```
GET /api HTTP/1.1
Origin: http://moja-strona.pl
Host: moje-api.pl
Accept-Language: en-US
User-Agent: Mozilla/5.0
```

Odpowiedź serwera to:

```
Access-Control-Allow-Origin: http://moja-strona.pl
Content-Type: text/html; charset=utf-8
```

Nagłówek `Access-Control-Allow-Origin` mówi przeglądarce, że serwer zgadza się, żeby `moja-strona.pl` sięgała do naszego API. Można w tym miejscu ustawić gwiazdkę, wtedy serwer zgadza się na wszystkie strony. Aby móc wykorzystywać *Cross-Origin Resource Sharing* w swoim serwisie muszą zostać spełnione dwa warunki:

- przeglądarka musi obsługiwać CORS
- serwer, do którego się odwołujemy musi być odpowiednio skonfigurowany

Wszystkie obecne przeglądarki obsługują CORS także punkt pierwszy nie jest problemem. Natomiast konfiguracja serwera czyli naszego API przebiega następująco. Po pierwsze musimy zainstalować gem `'rack-cors'`, następnie dodajemy następujący kod do pliku `application.rb`

⁴and. dzielenie zasobów różnego pochodzenia

```
config.middleware.use Rack::Cors do
  allow do
    origins "*"
    resource "*", headers: :any, methods: [:get, :post,
      ↪ :put, :delete, :options]
  end
end
```

Po ponownym uruchomieniu serwera jest on odpowiednio skonfigurowany.

W *Ember.js* nie potrzebujemy dodawać nowych modułów, wystarczy dodać do pliku `enviroment.js` kod:

```
ENV.contentSecurityPolicy = {
  'default-src': "http://www.facebook.com/",
  'script-src': "'self' 'unsafe-inline' 'unsafe-eval' ",
  'font-src': "'self' http://fonts.gstatic.com",
  'connect-src': "'self' http://localhost:3000/
  ↪ http://localhost:3000/photos http://127.0.0.1:3000
  ↪ http://localhost:3000/users/sign_in ",
  'img-src': "'self' http://localhost:3000
  ↪ http://localhost:4200/category" ,
  'report-uri': "'localhost'",
  'style-src': "'self' 'unsafe-inline'
  ↪ http://fonts.googleapis.com",
  'media-src': "'self'"
}
```


ROZDZIAŁ 4

Testy

4.1. Testowanie API przy użyciu RSpec

RSpec jest najpopularniejszą biblioteką używaną do testowania aplikacji napisanych w *Ruby on Rails*. Testy dzielą się na 3 typy. Pierwszy z nich to testy kontrolera, które sprawdzają poprawność zapisywania i odczytywania danych w bazie. Dzięki temu wiadomo czy aplikacja prawidłowo komunikuje się z bazą danych. Drugi typ testów to testy modelu danych. Testy te głównie sprawdzają czy walidatory, czyli elementy sprawdzające poprawność i format danych wpisywanych przez użytkownika, działają prawidłowo. Gdyby nie było walidacji, można by było na przykład zamiast imienia wpisać dowolny ciąg znaków o dowolnej długości co może powodować konflikty i błędy podczas przetwarzania danych. Trzeci typ testów to testy funkcjonalne, czyli testy zachowania interfejsu użytkownika na konkretne akcje, jak na przykład wpisanie w pole formularza lub kliknięcie na guzik. Do testów funkcjonalnych często używana jest dodatkowa biblioteka *Capybara*, która współpracuje z *Rspec*.

4.1.1. Raport z testów API

W API aplikacji zostało przetestowane głównie łączenie się aplikacji ze sieciami społecznościowymi tj. *Facebook* oraz *Flickr*.

4.2. Testowanie Front-end przy użyciu QUnit

QUnit jest standardową biblioteką używaną do testowania aplikacji napisanych w *Ember.js*. Pozwala na pisanie zarówno *Unit tests* ¹ jak i *Acceptance tests* ². Testy

¹ang. Testy jednostkowe

²ang. Testy akceptujące

jednostkowe to wyizolowane części funkcjonalności, bez ich zależności od innych elementów. Przykładowo:

- Użytkownik ma imię.
- Użytkownik ma imię i nazwisko, które nie może przekraczać 50 znaków.
- Zdjęcie ma nazwę.

Testy akceptacyjne są używane do testowania interakcji użytkownika z aplikacją. Przykładowo:

- Użytkownik jest w stanie zalogować się przez formularz logowania.
- Użytkownik może utworzyć nowe konto.
- Użytkownik po zalogowaniu jest przekierowany do strony głównej.

Uruchamianie testów może przebiegać na dwa sposoby. Pierwszy z nich to wpisanie komendy `ember test`. Aplikacja uruchomi wszystkie testy w konsoli i ewentualne błędy zostaną również tam wyświetlone. Drugim sposobem jest normalne uruchomienie serwera i wpisanie adresu serwera z końcówką `/tests`. W oknie przeglądarki zostaną pokazane wszystkie testy.

Warto wspomnieć również o tzw. *helperach* czyli plikach zawierających nie-standardowe metody pomagające testować kod.

4.2.1. Raport z testów Front-end

W *front-endzie* główny nacisk został położony na przetestowanie logowania i uwierzytelniania użytkowników i te elementy zostały przetestowane automatycznie. Reszta aplikacji została przetestowana ręcznie a znalezione błędy zostały usunięte.

ROZDZIAŁ 5

Wkład własny

Zakończenie

W niniejszej pracy starałem się pokazać budowę aplikacji modularnej, zalety i wady takiego rozwiązania oraz szereg różnych problemów z jakimi musi się uporać developer podczas pisania takowej aplikacji.

DODATEK A

Tytuł załącznika jeden

Treść załącznika jeden.

DODATEK B

Tytuł załącznika dwa

Treść załącznika dwa.

Bibliografia

- [1] Robert Hoekman jr. *Magia interfejsu. Praktyczne metody projektowania aplikacji internetowych*. Helion, 2010.
- [2] Jenifer Tidwell. *Projektowanie interfejsów. Sprawdzone wzorce projektowe*. Helion, 2012.
- [3] Don Norman. *The Design of Everyday Things*. Basic Books, 2002.
- [4] Steve Krug. *Nie każ mi myśleć! O życiowym podejściu do funkcjonalności stron internetowych*. Helion, 2012.
- [5] DouglasCrockford. *JavaScript - Mocne Strony*. Helion, 2011.
- [6] Joe Fiorini. *User Interface Thinking in Rails: An Example*. 2012.
- [7] Rolf Hennicker Nora Koch. *Modeling the User Interface of Web Applications*. 2001.
- [8] [RSpec Docs](#) dostęp 2015-05-12.
- [9] [Ruby on Rails Docs](#) dostęp 2015-05-12.
- [10] [EmberJs Docs](#) dostęp 2015-05-12.
- [11] [Ember CLI Docs](#) dostęp 2015-05-12.
- [12] [Ember Rails](#) dostęp 2015-05-12.
- [13] [EmberJS Tutorial](#) dostęp 2015-05-12.
- [14] [Introduction to Ember](#) dostęp 2015-05-12.

Spis tabel

Spis rysunków

3.1. Html-reporter	15
3.2. Diagram budowy Ember.js	21

Oświadczenie

Ja, niżej podpisany(a) oświadczam, iż przedłożona praca dyplomowa została wykonana przeze mnie samodzielnie, nie narusza praw autorskich, interesów prawnych i materialnych innych osób.

.....

data

.....

podpis