UNIWERSYTET ŚLĄSKI
WYDZIAŁ NAUK ŚCISŁYCH I TECHNICZNYCH

Aleksander Jaworski

313998

TWORZENIE APLIKACJI INTERNETOWYCH NA PRZYKŁADZIE APLIKACJI INTERNETOWEJ LOGO QUIZ WEB

PRACA DYPLOMOWA INŻYNIERSKA

Promotor:
dr inż. Anna Gorczyca-Goraj

Chorzów 2020

Abstrakt

Moim projektem inżynierskim jest aplikacja internetowa Logo Quiz Web. W przedłożonej pracy opisuję podstawy tworzenia takich aplikacji. Na początku wprowadzam podstawowe zwroty, które pojawiają się w dalszej części pracy, oraz narzędzia potrzebne do tworzenia nowoczesnych aplikacji. Następnie opisuję oraz pokazuję działanie niektórych bibliotek, których użyłem do stworzenia mojego projektu. Na końcu omawiam, jakie problemy napotkałem podczas tworzenia aplikacji, oraz jakimi sposobami je rozwiązałem.

Słowa kluczowe: Aplikacja internetowa, React.JS, Redux, Firebase, Firestore, Cloud functions

Oświadczenie autora pracy

Ja, niżej podpisany, autor pracy dyplomowej pt. Tworzenie aplikacji internetowych na podstawie aplikacji internetowej Logo Quiz Web, o numerze albumu: 313998, student Wydziału Nauk Ścisłych i Technicznych Uniwersytetu Śląskiego w Katowicach, kierunku studiów Informatyka Stosowana oświadczam, że ww. praca dyplomowa:

- została przygotowana przeze mnie samodzielnie[1],
- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994r. o prawie autorskim i prawach pokrewnych (tekst jednolity Dz. U. z 2006r. Nr 90, poz. 631, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- nie zawiera danych i informacji, które uzyskałem w sposób niedozwolony,
- nie była podstawą nadania dyplomu uczelni wyższej lub tytułu zawodowego ani mnie, ani innej osobie.

Oświadczam również, że treść pracy dyplomowej zamieszczonej przeze mnie w Archiwum Prac Dyplomowych jest identyczna z treścią zawartą w wydrukowanej wersji pracy.

Jestem świadomy odpowiedzialności karnej za złożenie fałszywego oświadczenia.

....

(miejsce i data) (podpis autora pracy)

[1] uwzględniając merytoryczny wkład promotora (w ramach prowadzonego seminarium dyplomowego)

Spis treści

Lista rysunków

1 Wstęp

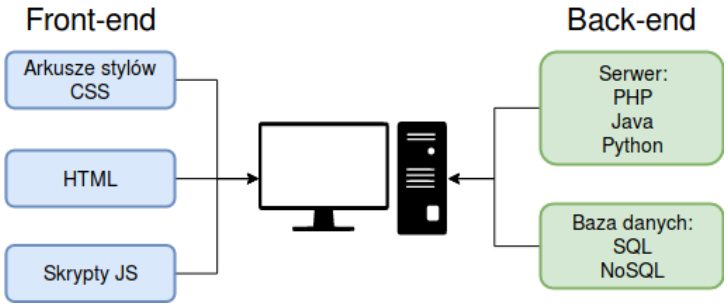
1.1 Skład strony internetowej

Zazwyczaj strona internetowa składa się z pliku HTML, który określa strukturę strony. Każda strona składa tylko z jednego takiego pliku. Do takiej strony mogą również być dołączane arkusze stylów i skrypty. Arkusze stylów CSS (*Cascading Style Sheets*) decydują o tym jak wygląda strona. W celu dokonania zmiany wyglądu strony trzeba napisać regułę, która będzie przestrzegana przez znaczniki w pliku HTML.

Językiem programowania, w jakim pisze się te skrypty, jest Javascript. Kiedyś takie skrypty dodawały ograniczoną funkcjonalność do strony, np. były odpowiedzialne za animacje czy wyświetlanie menu. Lecz z biegiem czasu, wraz z rozwojem komputerów i przeglądarek, skrypty mogły posiadać coraz więcej odpowiedzialności. Aktualnie istnieje wiele stron, w których Javascript kieruje prawie całą logiką; mój projekt inżynierski jest właśnie taką stroną.

1.2 Co to jest aplikacja internetowa

Poprzez aplikację internetową rozumiem stronę internetową, która nie tylko jest udostępniana przez serwer, ale w jakiś sposób się z nim komunikuje. Dzieli się ona na dwie warstwy:



Infografika. 1: Przykładowy podział front-end oraz back-end
Źródło obrazka komputera: https://svgsilh.com/image/2237420.html

1. Front-end to warstwa, którą widzi użytkownik. W jej skład wchodzi strona internetowa, skrypty oraz style do niej dołączone.
2. Back-end to warstwa niewidoczna dla użytkownika. Jest to serwer, który przede wszystkim udostępnia strony internetowe. To on decyduje, jaka strona zostanie wysłana zależnie od adresu URL. Taki serwer może również posiadać API (*application programming interface*), z którym możemy się połączyć. W uproszczonym znaczeniu API to część serwera, która przyjmuje zapytania i wysyła odpowiedzi [1]. Z tak zdefiniowanym API łączymy się za pomocą URL. Najpopularniejsze rodzaje zapytań to GET i POST. Najczęściej interfejs ten jest używany w celu komunikacji z bazą danych.

W aplikacji internetowej front-end może posiadać formularz, który użytkownik musi wypełnić. Po wypełnieniu formularz zostaje wysyłany do back-endu. Następnie jest tam obsługiwany i zazwyczaj użytkownik dostaje odpowiedź z serwera. Taki przepływ informacji daje wiele możliwości np. logowanie się do aplikacji, dodawanie danych do bazy danych lub szukanie informacji w tej bazie danych.

1.3 Aplikacja jednostronowa (*Single page application*)

Jest to aplikacja internetowa, która jest wczytywana tylko raz przez przeglądarkę po wejściu na stronę. W trakcie używania takiej aplikacji użytkownik nie jest przekierowywany na inne strony. Wszystkie zmiany dzieją się dynamicznie poprzez Javascript dołączony do strony. Takie rozwiązanie powoduje, że aplikacja działa bardzo szybko i płynnie ponieważ przeglądarka nie musi odpytywać serwera o nowe strony. Takie rozwiązanie posiada jednak swoje wady.

Jedną z nich jest długi czas ładownia, kiedy użytkownik pierwszy raz wchodzi do aplikacji. Następne wizyty będą szybsze, ponieważ przeglądarka zapamięta (*cache*) pliki dołączone do strony. Druga wada nie dotyczy bezpośrednio aplikacji, ale jej pozycjonowania przez serwisy takie jak Google czy Bing. W uproszczonej formie, serwisy te posiadają swoje "roboty", które wchodzą na stronę i ściągają cały plik źródłowy wraz z treścią strony. Właśnie dzięki tej treści użytkownik może łatwiej odnaleźć stronę. W przypadku aplikacji jednostronowej pojawia się problem. Roboty wejdą do aplikacji, ale będą w stanie jedynie zobaczyć ekran logowania lub głównego menu, co spowoduje, że nie znajdą żadnej wartościowej treści. Jednym z rozwiązań tego problemu byłoby zrobienie oddzielnej strony, na której zamieszczamy opis naszej aplikacji wraz z linkiem do niej. Drugie rozwiązanie wymagałoby, aby strona była dynamicznie tworzona po stronie serwera. Rozwiązanie to nie jest popularne, ponieważ obciąża serwer bardziej niż tradycyjny sposób z przekierowywaniem na inne strony.

1.4 Responsywność strony

W uproszczonej formie pojęcie to oznacza, że strona dobrze wygląda na ekranie o dowolnym rozmiarze. Przy początku wzrostu popularności smartfon duża ilość stron posiadała dwie wersje: komputerową oraz mobilną. Czasami strony pytały, jaką wersję użytkownik chce odwiedzić. Często serwer sam decydował, którą wersję zwrócić dzięki nagłówkom, które wysyłała przeglądarka. W przypadku, kiedy serwer otrzyma nagłówek z systemu operacyjnego telefonu lub jego przeglądarki, wysyła on wersję mobilną, a w innym przypadku - wersję komputerową. Razem z rozwojem przeglądarek najpopularniejszym rozwiązaniem stały się responsywne strony internetowe. Dzięki temu posiadamy tylko jedną stronę, która jest używana na komputerach i na urządzeniach mobilnych. Powoduje to, że utrzymanie takiej strony przez developera staje się łatwiejsze.

1.5 Punkty przerwania (*breakpoints*) strony

Te punkty wyznaczają rozmiary, w których zmienia się wygląd strony: w większości przypadków skupiają się tylko na szerokości okna przeglądarki. Najpopularniejsze punkty szerokości, z którymi się spotkałem to: 1200px, 960px, 768px, 480px [2]. Mogą one odpowiadać urządzeniom: komputer, tablet, duży smartfon, mniejszy smartfon. Punkty te nie są określane w skrypcie, ale w arkuszu stylów strony, który zawsze określa jej wygląd.

1.6 Kompatybilność (*compatibility*) przeglądarek

Pojęcie to odnosi się do technologii, które wspierane są przez przeglądarki. Poprzez technologie rozumiem nowe wersje Javascript oraz nowe reguły w arkuszach stylów. Niektóre nowoczesne przeglądarki posiadają dużo takich technologii, a niektóre prawie żadnych. Dlatego tworząc nowoczesną stronę internetową musimy określić jaką najstarszą przeglądarkę wspierać.Warto też przetestować stronę na wszystkich popularnych przeglądarkach. Niestety samo wspieranie technologii przez przeglądarkę nie oznacza, że wszystko będzie działało poprawnie. Zazwyczaj największym problemem jest przeglądarka safari, ponieważ często na innych przeglądarkach strona wygląda dobrze, ale na safari nie wygląda poprawnie.

1.7 Kompilatory i transpilatory (*compilators/transpilers*)

JavaScript jest językiem ciągle rozwijającym się, ale jak wspominałem wcześniej te nowości nie są zawsze dostępne w przeglądarkach. W rozwiązaniu tego problemu pomagają nam transpilatory kodu (czasem nazywane kompilatorami). Dzięki nim możemy pisać kod w nowych wersjach JavaScript, ponieważ są one później transpilowane do starszych wersji, które są wspierane przez wszystkie przeglądarki.

1.8 Bundlery modułów

JavaScript nie jest językiem obiektowym, co prawda nowe wersje posiadają klasy, lecz są one w początkowej fazie implementacji. W środowisku przeglądarkowym JavaScript nie ma wbudowanej możliwości importowania innych plików skryptowych lub modułów. Można to obejść tworząc wiele plików skryptowych, które muszą być załączone do przeglądarki, i używając globalnych zmiennych, do których będą się one odnosić. Lepszym rozwiązaniem okazują się tzw. bundlery modułów (*module bundlers*), które pozwalają łączyć wiele plików skryptowych w jeden, finalnie zamieszczony w przeglądarce, bez konieczności używania globalnych zmiennych.

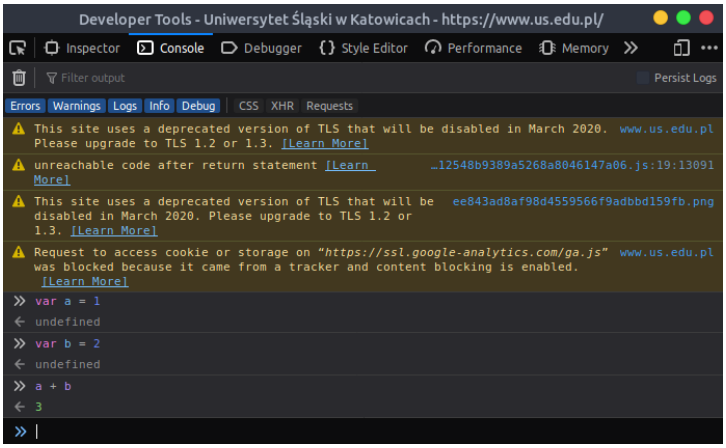
1.9 Biblioteki do budowy aplikacji (*Frameworks*)

Przy tworzeniu aplikacji internetowych pomagają deweloperom frameworki, które znacznie ułatwiają pisanie w JavaScript. Na chwilę obecną takich bibliotek są setki. Często bywa tak, że biblioteki narzucają swoje własne składnie i reguły, które mogą nie przypominać JavaScript. Używanie takich bibliotek znacznie przyśpiesza tworzenie aplikacji, która staje się również łatwiejsza w utrzymaniu, oraz pozwala deweloperom na uniknięcie wielu błędów.

1.10 Konsola dewelopera

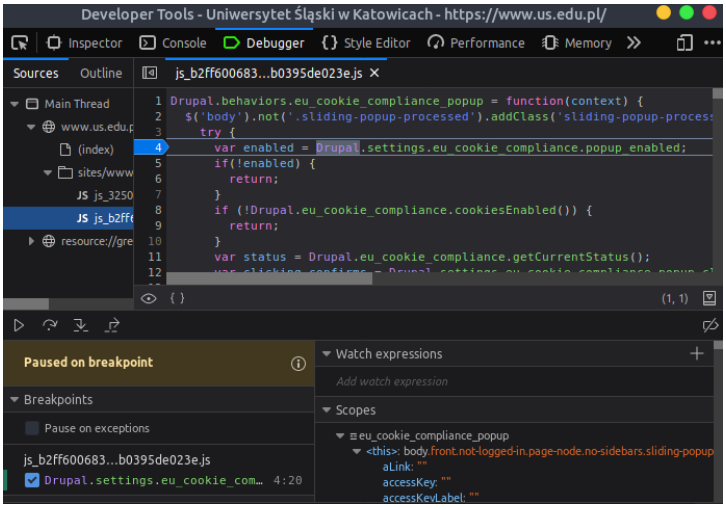
Większość przeglądarek internetowych zainstalowanych na komputerze posiada swoje własne konsole. Ja osobiście używam konsoli która jest dołączona do przeglądarki Firefox Developer Edition [3]. Bez tego narzędzia tworzenie aplikacji byłoby praktycznie niemożliwe. Posiada ono wiele funkcji, a do najważniejszych należą:

1. Konsola przeglądarki - w niej pojawiają się wszystkie błędy lub informacje, które wypisaliśmy (*log*). Dzięki niej mamy również możliwość wykonywania wszystkich poleceń JavaScript. Zazwyczaj używa się jej, żeby zobaczyć wartość jakichś zmiennych.
2. Debugger, który pozwala ustawiać breakpoint'y (o innym znaczeniu niż poprzednio opisane w sekcji 1.5) na wybraną linię w skrypcie - spowoduje to zatrzymanie działania strony po wejściu na zaznaczoną linię. Mamy wtedy podgląd na wszystkie zmienne, więc możemy łatwo prześledzić działanie aplikacji.
3. Możliwość zobaczenia wszystkich zapytań jakie przeglądarka wysyła do serwerów oraz odpowiedzi, jakie otrzymuje. W ten sposób możemy łatwo sprawdzać czy zapytania do API i jego odpowiedzi są poprawne.
4. Możliwość zmiany stylu każdego elementu na stronie (tylko do podglądu).



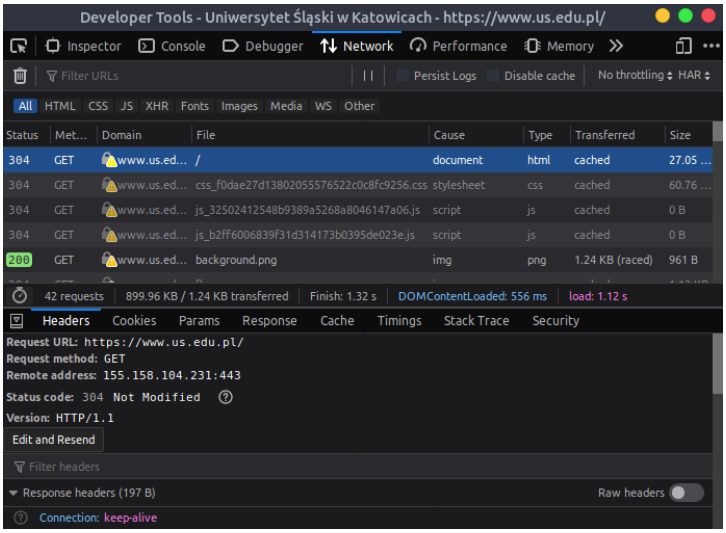
Infografika. 2: Konsola dewelopera w przeglądarce.

Wszystkie logi wypisane w konsoli przez stronę uniwersytetu. W tym wypadku dostajemy informacje, że uniwersytet do marca 2020 powinien przejść na nowszą wersję TLS, w przeciwnym wypadku firefox będzie gorzej wspierał stronę. Znajduje się tam również informacja, że w skrypcie dołączonym do strony znajduje się jakiś błąd logiczny.



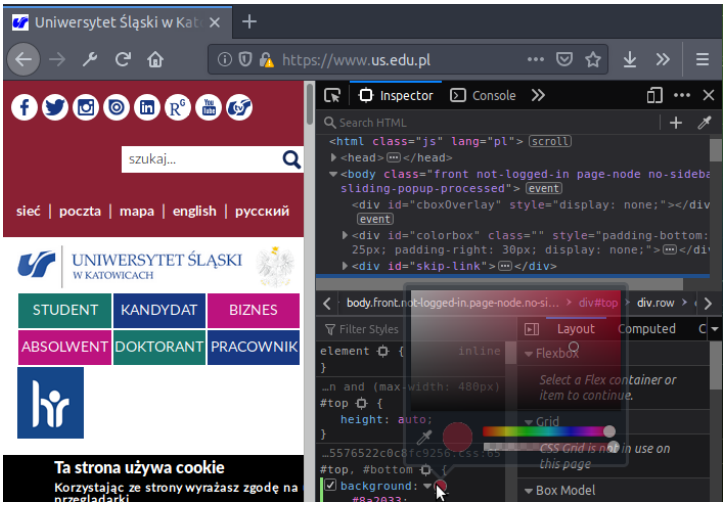
Infografika. 3: Debugger w przeglądarce.

W tym przypadku zatrzymałem stronę uczelni w momencie, gdy tworzy okienko informujące o ciasteczkach używanych na stronie. W czasie, gdy strona jest zatrzymana na jakiejś linii skryptu, możemy zobaczyć z jakich zmiennych korzysta.



Infografika. 4: Monitor sieci w przeglądarce.

Podgląd na zapytania, które wykonuje strona uczelni. Zawsze pierwsze zapytanie będzie odnosiło się do pliku HTML strony za pomocą metody GET.



Infografika. 5: Edytor stylu w przeglądarce.

Podgląd zmiany górnego menu na stronie uniwersytetu (na kolor czerwony).

2 Aplikacja

2.1 Prezentacja danych

W celu wyświetlenia danych w przeglądarce używam framework **"React"** [4]. Pozwala ona rozdzielić aplikację na tak zwane **komponenty**. Mogą być one małe jak zwykły przycisk, lub duże jak cała aplikacja. Takie podejście umożliwia nam utrzymanie zasady DRY (*Don't repeat yourself*): zamiast tworzyć 3 przyciski o różnych kolorach, możemy stworzyć jeden, do którego będziemy przekazywali kolor.

Jedną z największych zalet platformy React jest jej abstrakcja. Dzięki niej programista nie musi dokonywać ręcznej aktualizacji wybranego komponentu aplikacji, gdy zajdzie jakaś zmiana w stanie aplikacji. React wykonuje takie zadanie automatycznie. W sprytny sposób porównuje zmieniony stan z tym, który jest aktualnie w komponentach. Jeżeli stan używany przez komponent został zmieniony, sprawdza czy ta zmiana powinna powodować aktualizację komponentu. Metodę, której do tego używa możemy zaimplementować własnoręcznie w każdym komponencie, ale nie musimy. Wbudowana metoda przy każdej zachodzącej zmianie powoduje aktualizację komponentu.

Poniżej zamieszczam przykład opisanych własności platformy React.

```
class ColorfulButton extends React.Component {
  constructor(props){
    super(props);
    this.state = {licznik: 0};
    this.zwiekszIlosc = this.zwiekszIlosc.bind(this);
  }

  zwiekszIlosc(){
    this.setState( {
      licznik: this.state.licznik + 1;
    } );
  }

  render() {
    var color = this.props.color;
    var text = this.props.text;
    return (
        <button
            style={{backgroundColor: color}}
            onClick={this.zwiekszIlosc}
        >
            {text} {this.state.licznik}
        </button>
    );
  }
}

ReactDOM.render(
  <ColorfulButton
    color="#fff"
```

```
    text="Ilosc klikniec: "
  />,
  document.getElementById('container')
);
```

Infografika. 6: Aplikacja licząca ilość kliknięć użytkownika.

W powyższej aplikacji tworzymy prosty przycisk, który po każdym kliknięciu zwiększa licznik kliknięć. Jedynym zadaniem programisty jest zadbanie o to, gdzie używany jest komponent i w jaki sposób zmienia on stan aplikacji. Prezentacja aplikacji podczas zmian dokonywana jest automatycznie poprzez działanie React.

2.2 Architektura stanu

2.2.1 Podstawowe terminologie

Do zarządzania stanem aplikacji używam biblioteki **"Redux"** [5].

Poniżej przedstawiam terminologię i stosowane konwencje potrzebne do zarządzania stanem przy użyciu tej biblioteki. Poprzez obiekt rozumiem prostą strukturę danych, która składa się z kluczy i przypisanych im wartości.

Akcje - są to obiekty, które używane są do zmian w stanie aplikacji.

```
{
  type: "ADD_TODO",
  payload: "Wyniesc smieci"
}
```

type - Prosta nazwa opisująca, co chcemy zmienić w stanie aplikacji; w naszym wypadku jest to dodanie rzeczy do listy do zrobienia (TODO)

payload - Dane, które chcemy użyć do zmiany stanu. Przyjęte nazewnictwo dla tych danych jest zwykłą konwencją. Oczywiście dane nie muszą być tylko w postaci zwykłego tekstu; może to być zagnieżdżony obiekt lub lista.

Kreator akcji - Funkcja zwracająca akcje. Tego typu funkcje pomagają zmniejszyć ilość kodu i postępować zgodnie z zasadą DRY.

```
function addTodo(text) {
  return {
    type: "ADD_TODO",
    payload: text
  }
}
```

text - Parametr, przez który przekazujemy dane do reducera.

Reducer - Funkcja, w której zamieszczamy warunki i logikę, po wykonaniu których ma zmieniać się stan aplikacji. Sprawdza ona typ przesłanej akcji, a następnie wykonuje polecenia przypisane do tej akcji. W aplikacji może być wiele takich funkcji, ale z reguły każda odpowiada za inną część stanu np. *Użytkownik*, *Lista todo*. Można powiedzieć, że funkcje Reducer są zwykłymi instrukcjami decyzyjnymi, które zwracają nowy stan zależnie od wykonanej akcji.

```
function todos(state = [], action) {
  switch (action.type) {
    case "ADD_TODO":
      let newState = state.slice(0); //Tworzymy kopie
      let newTodo = action.payload;
      newState.push(newTodo);
      return newState
    case "DELETE_TODO":
      let newState = state.slice(0); //Tworzymy kopie
      let index = action.payload;
       //Taka zmiana jest dozwolona poniewaz operujemy na kopii
      newState.pop(index);
      return newState;
    default:
      return state
  }
}
```

state - Obecny stan aplikacji (w momencie wywołania funkcji). Zmieniając ten stan, musimy dopilnować, żeby operować na jego kopii, a nie oryginale. Jeśli operowalibyśmy na oryginale, do aplikacji mogłyby wkraść się błędy. Na przykład jakaś część aplikacji zapisuje do zmiennej referencję do listy, następnie zmienna ta jest przesyłana do stanu. Jeżeli zapisując do stanu nie wykonamy kopii listy, do której odnosi się ta referencja, to aplikacja nadal może działać poprawnie. Jednakże po jakimś czasie oryginalna lista, do której odnosi się referencja może ulec zmianie, wówczas lista znajdująca się w stanie aplikacji również się zmienia, ponieważ posiada ona tę samą referencję. Oznacza to, że zmiana w stanie aplikacji nie została wykonana za pomocą akcji, ale poprzez efekt uboczny działania na oryginalnej referencji listy.

action - Zwykły obiekt akcji, który zawsze posiada pole **type**. Dzięki temu polu aplikacja wie, jakie operacje wykonać na swoim stanie. Czasami akcja posiada pole **payload**, w którym przesyłane są dane potrzebne do wykonania danej operacji.

1. Użytkownik wpisuje do pola swoje zadanie TODO. W tym wypadku jest to “Wynieś śmieci”
2. Klika przycisk “DODAJ”
3. Przycisk wywołuje funkcję addTODO, do której jako argument przekazuje tekst wpisany przez użytkownika (“Wynieś śmieci”)
4. Funkcja zwraca **Akcję**, która oznacza, że chcemy zmienić stan aplikacji. Typ tej akcji wskazuje, że do listy chcemy dodać nowe zadanie.
5. **Akcja** trafia do funkcji **Reducer**, która sprawdza, co chcemy zrobić. Bazując na **typie** akcji ta funkcja stwierdza, że chcemy dodać TODO. W tym celu wykonuje ona następujące czynności:

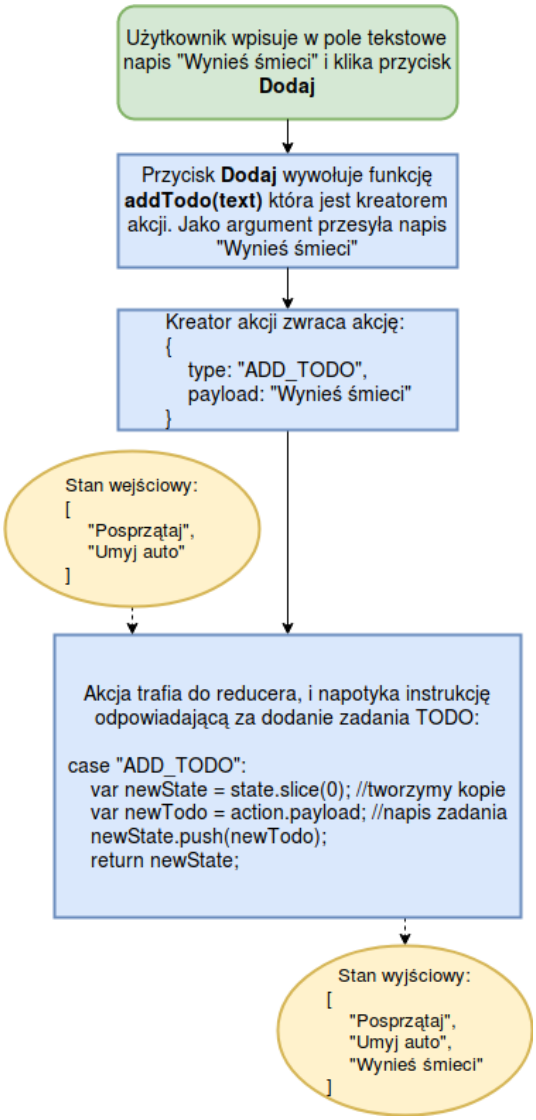
   I Tworzy kopię obecnego stanu aplikacji.

II Do kopii stanu dodaje wprowadzony przez użytkownika tekst.

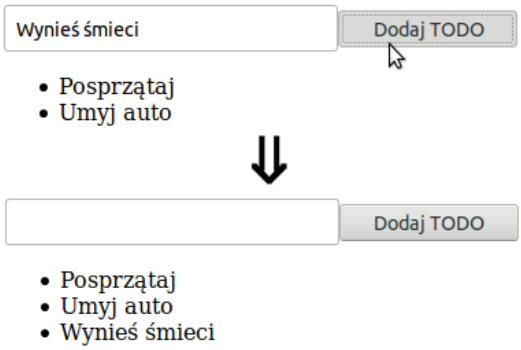
III Zwraca zmienioną kopię naszego stanu.

Zwrócona kopia zostaje nowym stanem aplikacji. Jeśli pominiemy krok tworzenia kopii stanu i będziemy operować na aktualnym stanie, złamiemy wtedy najważniejszą zasadę Redux, którą opiszę w następnym etapie.

Infografika 7 przedstawia przepływ w aplikacji, której zadaniem jest dodawanie do listy TODO nowych zadań. Mamy do dyspozycji pole do wpisania ciągu znaków oraz przycisk. Po wpisaniu jakiegoś zadania klikamy przycisk *Dodaj* i do listy jest dodawane nowe zadanie.



Infografika. 7: Przepływ zdarzeń w aplikacji służącej do planowania zadań.
Przepływ zdarzeń w aplikacji, której zadaniem jest dodawanie zadań TODO do listy.
Zielony - Interakcja użytkownika
Niebieski - Logika wykonywana przez aplikację
Żółty - Stan aplikacji



Infografika. 8: Przykładowa aplikacja służąca do planowania zadań.
Aplikacja, która wykonuje przepływ danych opisany w sekcji 2.2.1.

2.2.2 3 zasady Redux

1. Stan aplikacji znajduje się tylko w jednym **store**, czyli w obiekcie, który składa się z kluczy i wartości. Dzięki temu programista może mieć łatwy podgląd w cały stan aplikacji w jednym miejscu. Taka implementacja ułatwia wypełnianie stanu aplikacji przez źródła zewnętrzne np. bazę danych, ponieważ wystarczy ściągnąć dane, zamienić je w obiekt, a następnie zastąpić nimi pusty stan. Powrót do poprzedniego stanu aplikacji jest przez to bardzo ułatwiony, wystarczy że w jakimś kluczu będziemy trzymali historię stanu aplikacji. Powrót będzie polegał na zwykłym zastąpieniu obecnego stanu tym z historii.
2. Stan jest tylko do odczytu. Oznacza to, że wszystkie zmiany w stanie są wykonywane poprzez **akcje**, a nie przez zewnętrzną edycję obiektu store. W ten sposób mamy pewność skąd pochodzi zmiana, ponieważ do każdej akcji dołączona jest informacja o tej zmianie. Niestety przez asynchroniczne zapytania do zewnętrznych źródeł, programista nigdy nie może być pewny, które zapytanie zostanie ukończone wcześniej. Akcje umożliwiają dodatkowo stworzenie historii, w jakiej kolejności zostały wykonane, a co za tym idzie, w jaki sposób został zmieniony stan aplikacji i w jakiej kolejności.
3. Wszystkie zmiany w stanie, są wykonywane poprzez **pure functions**. Są to funkcje działające wyłącznie na zmiennych, które zostały stworzone podczas działania tej funkcji, lub te przekazane jako parametry. Nie korzystają one z żadnych globalnych zmiennych, ponieważ gdyby tak było, cały stan aplikacji przestałby być przewidywalny. Programista miałby kłopot z określeniem pochodzenia zmiany, która nie wynika z żadnej z wykonanych akcji.

2.2.3 Asynchroniczność w JavaScript

JavaScript jest językiem jednowątkowym. Oznacza to, że nie możemy stworzyć nowych wątków, żeby wykonać jakieś asynchroniczne zadanie, czyli takie, którego czas działania nie jest określony np. zapytanie serwera. JavaScript widząc asynchronicze zadania dodaje je do kolejki wydarzeń (*Event queue*), i wykonuje program dalej, aż do napotkania końca bloku. Następnie sprawdzane jest czy jakieś zadanie z tej kolejki zostało ukończone. W takim wypadku kolejka zwraca dane, które otrzymała po ukończeniu asynchronicznego zadania. Dzięki temu wszystkie asynchroniczne bloki kodu, które mogłyby się wykonywać długi czas, nie zatrzymują przeglądarki na czas jej działania.

Redux obsługuje tylko synchroniczny przepływ danych, lecz w aplikacji Logo Quiz Web prawie wszystkie potrzebne dane znajdują się w bazie danych. Do rozwiązania tego problemu użyłem biblioteki **"Redux Thunk"** [6], dzięki której oprócz zwykłych akcji możemy również wysyłać funkcje do **store**. Żeby biblioteka

poprawnie działała musimy ją dodać do store jako **middleware**, czyli oprogramowanie pośredniczące. W uproszczeniu, middleware odpowiada za to, co się dzieję pomiędzy wysłaniem akcji, a jej dotarciem do reducera. W wypadku redux-thunk sprawdza czy przesłana akcja to funkcja. Jeśli tak jest, dodaje ona do niej wywołanie zwrotne (*callback*), które zostanie wykonane np. w momencie uzyskania odpowiedzi z serwera. Po otrzymaniu tych danych callback tworzy akcje i przesyła je do reducera. Są one zwykłym obiektem, więc redux-thunk przepuszcza je jako zwykłe akcje.

2.3 Baza danych

2.3.1 Firestore

Jako bazę danych używam **"Cloud Firestore"** [7] od firmy **Firebase** [8] (obecnie należącej do Google). Jest to baza danych typu NoSQL (czyli taka, która nie jest oparta na SQL tzn. nie opiera się na relacjach w tabelach) ze wsparciem na wielu platformach i językach programowania. Głównym powodem dla którego wybrałem tę bazę jest brak potrzeby utworzenia własnego serwera, który komunikowałby się z bazą danych. Uwierzytelnianie użytkowników jest również w głównej mierze zapewnione dzięki **Firebase Authentication**.

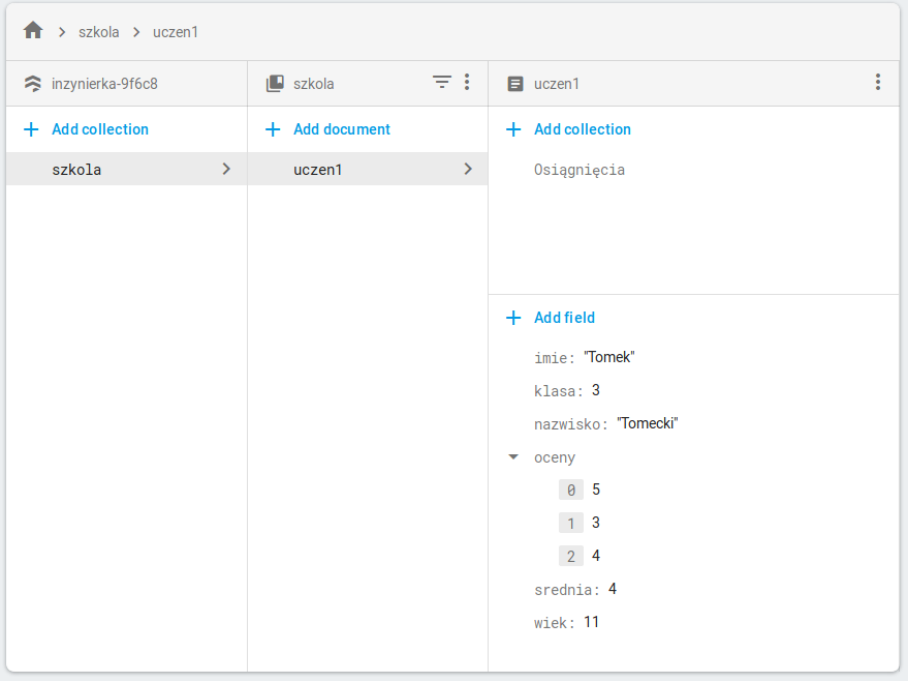
Przez to, że baza danych nie jest oparta na SQL, strukturyzacja danych mocno odbiega od typowych baz danych. Zamiast tabel i wierszy w skład bazy wchodzą:
Kolekcje - przy przechowywaniu danych służą jedynie jako pojemnik na inne dokumenty.
Dokumenty - posiadają pola, które przechowują dane za pomocą kluczy i wartości. Typ danych musi zostać określony. Mogą być to min. liczby, zmienne tekstowe, zmienne logiczne, tablice oraz zagnieżdżone zbiory kluczy i wartości (np. mapy). Poza danymi dokumenty mogą również przechowywać zagnieżdżone kolekcje.

W typowych bazach danych wiersze muszą znajdować się w tabelach. W bazie Firestore również trzeba przestrzegać podobnej reguły. Wszystkie dokumenty muszą znajdować się w kolekcjach, a wszystkie zagnieżdżone kolekcje muszą być umieszczona w dokumentach. Oznacza to, że kolekcja nie może posiadać bezpośrednio innej kolekcji, a dokument nie może innego dokumentu. Taka hierarchia pozwala na indeksowanie bazy danych, dzięki temu odpowiedzi z bazy są szybkie. Prędkość przetwarzania zapytania nie zależy od ilości dokumentów w kolekcji, ale jedynie od wielkości zwróconych dokumentów. Jest to przedstawione na infografikach 9 oraz 10.

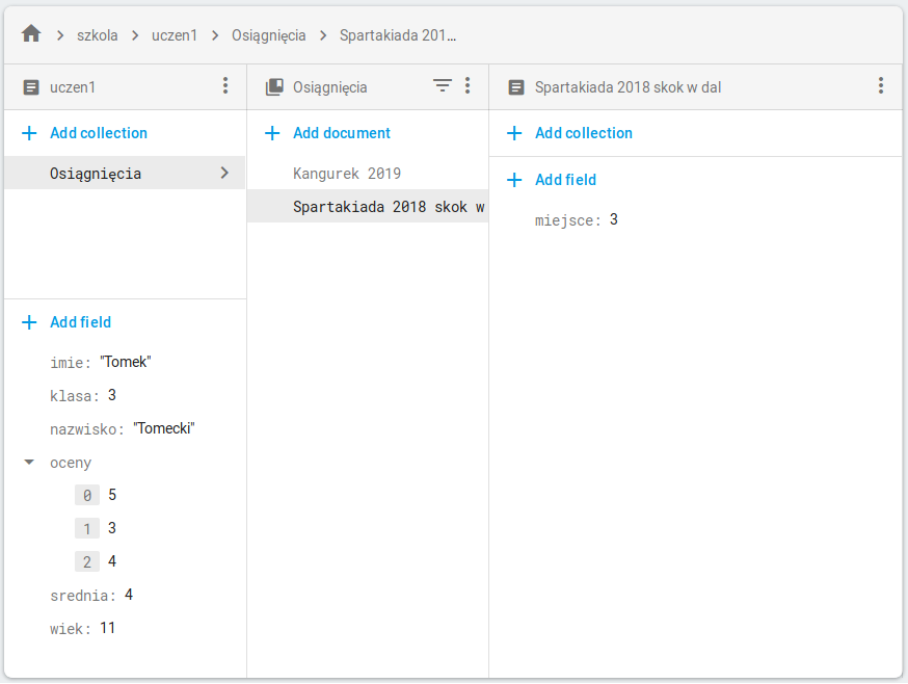
Na pierwszy rzut oka kolekcje mogą się wydawać trywialne, ponieważ tylko przechowują dokumenty. Lecz dzięki temu umożliwiają one tworzenie złożonych zapytań do bazy danych. Baza SQL pozwala zwrócić wiersze, w których spełnione jest jakieś kryterium, kolekcje również pozwalają na to. Przypuśćmy, że istnieje kolekcja o nazwie **Szkoła**, której dokumenty to uczniowie szkoły. Posiadają oni pola z imieniem, nazwiskiem itd. Żeby zwrócić wszystkich uczniów, którzy mają na imię Tomek, musimy wykonać następujące zapytanie (*query*).

```
1 db.collection("szkola").where("imie", "==", "Tomek")
```

Jak widać zapytanie posiada podobną strukturę do tych z baz SQL. W funkcji warunkowej **where** pierwszy argument to pole w dokumencie, które ma spełnić kryterium, drugi argument to porównanie, jakie ma zostać wykonane, a trzeci to wartość, która jest używana do porównania. Wszystkie dokumenty, które spełnią takie kryterium, zostają zwrócone. Jeśli jest taka potrzeba, możemy użyć wiele takich funkcji; wystarczy, że po zakończeniu **where** dopiszemy następną.



Infografika. 9: Widok z panelu Firestore przedstawiający przykładowy dokument ucznia. Pola (*fields*) użytkownika pokazują, że baza wspiera wiele typów danych; w tym wypadku są to dane tekstowe, liczbowe, oraz lista.



Infografika. 10: Widok z panelu Firestore pokazujący zagnieżdżoną kolekcję (*collection*) osiągnięć, która znajduje się w dokumencie ucznia.

Jeśli potrzebujemy informacji o konkretnym uczniu, możemy wykonać zapytanie bezpośrednio na jego dokumencie. Warunkiem tego jest znajomość jego identyfikatora ("uczen1").

```
db.collection("szkola").doc("uczen1")
```

Ograniczenia zapytań:

1. W funkcji **where** porównania typu <, >, >=, <= mogą być użyte tylko na jednym polu. Bazując na poprzednim przykładzie, nie możemy użyć zapytania postaci:

   .where("wiek", ">", "10").where("klasa", "<", "3")

   Oznaczało by to, że szukamy ucznia który ma więcej niż 10 lat i uczęszcza do klasy niższej niż trzecia. Niestety w Firestore takie zapytanie nie jest dozwolone.

2. Zapytanie może obejmować tylko jedną kolekcję.

3. Logiczne **OR** w funkcji **where** nie jest dostępne, czyli w naszym przykładzie zapytanie o ucznia który nazywa się Tomek lub Paweł, jest niemożliwe. Musimy wykonać dwa zapytania i jeśli będzie taka potrzeba, dodać otrzymane wyniki do jakiejś wspólnej struktury danych.

4. W funkcji **where** nie możemy użyć logicznego zaprzeczenia **!=**, ale jako zamiennik możemy użyć dwóch funkcji mniejszości i większości. Jeśli chcielibyśmy, żeby nasze zapytanie zwróciło wszystkich uczniów szkoły oprócz pięcioklasistów, możemy zrobić to w następujący sposób:

   .where("klasa", ">", "5").where("klasa", "<", "5").

   Niestety ten typ zapytania ograniczony jest tylko do typów liczbowych.

Poza funkcją **where** możemy również określić kolejność zwróconej odpowiedzi na zapytania lub określić maksymalną ilość zwróconych dokumentów.
Do określenia kolejności używamy funkcji **order**; wygląda ona następująco:

```
db.collection("szkola").orderBy("nazwisko", "desc")
```

Zwrócone dokumenty będą posortowane według nazwisk uczniów w kolejności od końca alfabetu.
Ograniczenie jest dokonywane funkcją **limit**:

```
db.collection("szkola")
                    .orderBy("srednia", "desc")
                    .limit(5)
```

Powyższe zapytanie zwróci nam 5 uczniów, których średnia jest największa

Ograniczanie ilości zwróconych dokumentów może wydawać się mało użyteczne, ale pozwala ono na stronicowanie danych. Oznacza to, że nie ma potrzeby zwracać od razu wszystkich dokumentów, brakujące później dokumenty można "dociągnąć". Jest to bardzo przydatna rzecz, ponieważ użytkownik nie musi ściągać całej bazy na początku, może ją ściągać etapami w zależności od swoich potrzeb. Odwołując się do naszego wcześniejszego przykładu z ocenami można rozważyć sytuację, kiedy rodzice lub nauczyciele są zainteresowani tylko 3 czy 5 najlepszymi uczniami. Wówczas pobieranie wszystkich uczniów z bazy wydłużyło by działanie zapytania oraz nie potrzebnie zwiększyło zużycie transferu internetowego. W takim wypadku stronicowanie jest bardzo dobrym rozwiązaniem tego problemu: na początku ściągamy 5 najlepszych uczniów, ale jeżeli użytkownik chce zobaczyć następnych 5, może to uczynić.

```
var zapytanie = db.collection("szkola")
                    .orderBy("srednia", "desc").limit(5)

```

```
function pobierzPiecUczniow(){

    zapytanie.get().then(function(dSnaps){
        var index = dSnaps.docs.length-1;
        var ostatniUczenWLiscie = dSnaps.docs[index];

        zapytanie =
            db.collection("szkola")
                .orderBy("srednia", "desc")
                .startAfter(ostatniUczenWLiscie)
                .limit(5);
    })

}
```

Do zmiennej **zapytanie** zostaje zapisane aktualne zapytanie do bazy. Funkcja **pobierzPiecUczniow** jest wywoływana na starcie aplikacji lub gdy użytkownik wyrazi chęć zobaczenia następnych uczniów w liście. Żeby otrzymać dane z zapytania, musimy wywołać funkcję **get()**. Ta funkcja jest obietnicą (*promise*). Oznacza to, że jest ona asynchroniczna, a czas jej ukończenia nie jest znany. Aby operować na zwróconych danych, trzeba dołączyć funkcję **then**, która jest wywoływana po pomyślnym wykonaniu zapytania (w przypadku, gdy możliwe jest, że zapytanie nie przebiegnie pomyślnie, należy dołączyć funkcję **catch**, która przechwyci błąd). Parametr **dSnaps** zostaje zwrócony przez uprzednio zakończoną funkcję get(). Jest to obiekt, który posiada listę zwróconych dokumentów lub pustą listę. Następnie zmienną zapytanie nadpisujemy dokumentem ostatniego ucznia w zwróconej liście w celu stworzenia nowego zapytania. Wykonanie nowego zapytania zwróci listę kolejnych 5 uczniów.

Jako dodatkowe zabezpieczenie, Firestore daje nam możliwość tworzenia własnych "zasad" dostępu do bazy danych. Najłatwiej będzie to przedstawić na przykładzie. Przypuśćmy, że każdy użytkownik w bazie danych posiada własny dokument, w którym zapisane są jego prywatne dane. Przy braku określonych zasad dostępu możliwe jest odczytanie tych danych przez osoby trzecie. W rezultacie każdy użytkownik miałby dostęp do informacji dotyczących wszystkich innych użytkowników. Żeby temu zapobiec, możemy stworzyć następującą zasadę:

```
service cloud.firestore {
  match /users/{userId} {
    allow read, write: if request.auth.uid == userId;

    match /friends/{friendId} {
      allow create: if !exists(/users/{userId}/friends/{friendId});
      allow read: if request.auth.uid != null;
    }
  }
}
```

Powyższa zasada ustala, że dostęp do dokumentu **/users/userId** ma tylko użytkownik, do którego ten dokument należy. Zagnieżdżona zasada odwołuje się do dokumentu w kolekcji **/users/userId/friends**. Pierwsza zawarta w niej reguła pozwala na stworzenie nowego dokumentu, tylko jeśli nie istniał on wcześniej.

Natomiast druga pozwala na odczyt wszystkich dokumentów w kolekcji **friends** tylko tym użytkownikom, którzy są zalogowani w aplikacji.

Firestore w niewielkim stopniu pozwala aplikacji pracować poprawnie nawet w przypadku braku internetu. Użytkownik może wówczas zapisywać swoje dane do bazy danych. Te zmiany są zapisywane w pamięci przeglądarki, a gdy połączenie z internetem znowu jest aktywne zostaną wysłane do bazy w takiej samej kolejności, w jakiej zostały zapisane do pamięci przez użytkownika. Niestety gdy użytkownik odświeży stronę wszystkie te zmiany zostaną stracone.

2.3.2 Funkcje w chmurze (*Cloud functions*)

Jako dodatek do bazy danych, używam również **Cloud functions** od Google. Pozwalają one operować na danych w bazie poprzez funkcje, które są zamieszczone w chmurze (serwisy które są zamieszczone w internecie i są dostępne całodobowo). Korzystam z tych dodatkowych możliwości, ponieważ wszystkie zapytania do bazy danych wykonywane poprzez aplikacje internetowe dzieją się jawnie. Oznacza to, że każdy może przechwytywać dane lub podejrzeć, jakie zapytanie jest wykonywane. Użytkownik może metodą prób i błędów odpytywać bazę danych i dostać informacje, które nie są dla niego przeznaczone. W przypadku aplikacji Logo Quiz Web mogą to być odpowiedzi do zagadek (szczegółowy opis zamieszczam w sekcji 3.1.3). Dlatego wszystkie krytyczne operacje takie jak inicjacja baz danych użytkowników, odblokowywanie poziomów lub usuwanie danych użytkownika, dzieją się za pomocą funkcji zamieszczonej w chmurze. Funkcje w chmurze mają pełny dostęp do bazy. Oznacza to, że nie podlegają one żadnym zasadom i mogą wykonywać na bazie wszystkie operacje.

Funkcje są hostowane w przeznaczonym do tego serwerze od Google. Mogą one wszystkie znajdować się w jednym pliku lub być rozbite na wiele niezależnych plików. Są one hostowane na niepodlegającym nam serwerze. Oznacza to, że funkcje te nie są wybudzone (uruchomione) całą dobę. Wybudzane są tylko w zależności od potrzeby, a usypiane, jeśli przez dłuższy czas nikt nie wysłał do nich zapytania. Niestety takie zachowanie powoduje, że jeśli funkcja nie była długo używana, jej wykonanie zajmie o wiele więcej czasu niż normalnie. Zaletą zaś jest to, że liczba uruchomionych funkcji będzie odpowiadała zapotrzebowaniu użytkowników. Nie znajdziemy się w sytuacji, w której funkcje przestaną działać z powodu zbyt dużej ilości zapytań.

Oprócz wykonywania zapytań do konkretnych funkcji, możemy również ustawiać wyzwalacze (*trigger*). Taki trigger umożliwia wykonywanie automatycznych operacji, które są uruchamiane, jeśli w bazie zostanie wykonana jakaś akcja. Może nią być stworzenie jakiegoś dokumentu, usunięcie lub jego aktualizacja. Wracając do przykładu ze szkołą przypuśćmy, że za każdym razem, gdy uczeń otrzyma ocenę, chcielibyśmy w jego dokumencie przechowywać całkowitą liczbę jego ocen.

```
1  exports.updateUser = functions.firestore
2    .document('szkola/{uczen}')
3    .onUpdate((change, context) => {
4      var staraWartosc = change.before.data();
5      var nowaWartosc = change.after.data();
6
7      if(staraWartosc.oceny === nowaWartosc.oceny){
8          return null;
9      }
10
11     var iloscOcen = nowaWartosc.oceny.length;
```

```
12
13      return change.after.ref.update({
14          iloscOcen: iloscOcen
15      });
16
17      });
```

Powyższy fragment kodu tworzy trigger, który jest uruchamiany przy każdej aktualizacji dokumentu jakiegoś użytkownika w kolekcji szkoła. W triggerze mamy dostęp do wartości pól przed ich aktualizacją oraz po ich zaktualizowaniu. Na początku sprawdzane jest czy oceny ucznia się zmieniły; jeśli nie, to trigger zwraca null. Oznacza to, że trigger nie wykona żadnych zmian, więc kończy swoje działanie. W przypadku gdy oceny się zmieniły, do zmiennej iloscOcen zapisywana jest długość tablicy ocen po aktualizacji. Na końcu zwracana jest asynchroniczna obietnica *promise*, która aktualizuje pole ilości ocen użytkownika w bazie.

2.3.3 Uwierzytelnianie użytkowników dzięki *Firebase Authentication*

Bez żadnej dodatkowej konfiguracji do aplikacji możemy dodać uwierzytelnianie. Dzięki temu nie musimy poświęcić dużej ilości czasu na implementację własnego systemu do uwierzytelniania.

Firebase Authentication zapewnia programiście większość podstawowych funkcjonalności takich jak logowanie czy rejestracja na wiele sposobów. Automatycznie przechowuje zakodowane hasła na swoim serwerze i pozwala łatwo wykonywać takie akcje jak wysyłanie emaila z informacją o konieczności potwierdzenia swojego konta lub emaila umożliwiającego użytkownikowi zmianę hasła.

Authentication daje możliwość różnorodnego uwierzytelniania użytkownika. W dodatku do rejestracji konta przy wykorzystaniu adresu email oraz hasła, pozwala na użycie portali społecznościowych takich jak facebook, google, czy twitter. Jest również możliwość uwierzytelnienia poprzez numer telefonu.

Jeśli użytkownik nie chce podawać żadnych swoich danych, może zalogować się jako użytkownik anonimowy. Oznacza to, że może korzystać z aplikacji w taki sam sposób, jakby stworzył konto. Niestety przy zmianie urządzenia lub w momencie wylogowania się, traci on dostęp do swojego anonimowego konta i zarazem do danych, ponieważ nie będzie się mógł ponownie na to samo anonimowe konto zalogować. Jeśli użytkownik zadecyduje, że chciałby założyć konto, aplikacja pozwoli mu zachować dane z aktualnie wykorzystywanego anonimowego konta. W tym celu jego anonimowe konto zostaje połączone z nowo tworzonym kontem. Programista jedynie musi wywołać odpowiednią metodę tworzenia konta z połączeniem, a wówczas do anonimowego użytkownika zostaną przypisane dane do logowania użytkownika. W rezultacie użytkownik utworzy konto, na którym zachowa swoje dane.

3 Problemy napotkane po drodze

3.1 Baza danych

3.1.1 Struktura

Dużym problemem przy pracy nad aplikacją Logo Quiz Web, okazało się rozplanowanie struktury bazy, ponieważ bardzo odbiegała od standardów SQL. Najtrudniejsza była kwestia, w jaki sposób ukryć przed użytkownikiem dyskretne informacje. Gdyby użytkownik miał możliwość ściągnąć wszystkie informacje o zagadce, mógłby zobaczyć jej rozwiązanie w pamięci przeglądarki lub w odpowiedzi z bazy danych. Dlatego uznałem, że lepszym rozwiązaniem będzie rozdzielić zagadkę na dwa dokumenty. Pierwszy dokument posiada wszystkie informacje dotyczące zagadki, a drugi jej odzwierciedlenia w grze. Pierwszy dokument służy do sprawdzania czy użytkownik poprawnie odgadł zagadkę, ponieważ ten dokument posiada oryginał odpowiedzi. Drugi dokument jest powielany i kopiowany do użytkownika, przy tworzeniu konta, lub za każdym razem, gdy użytkownik odblokuje nowy poziom.

3.1.2 Narzędzia bazy danych

Baza Firestore pozwala na tworzenie nowych dokumentów i kolekcji poprzez panel administracyjny, ale jest to bardzo powolny proces. Importowanie do bazy oraz eksportowanie z niej do pliku jest niemożliwe. Oznacza to, że nie możemy tworzyć kopii zapasowych oraz ich przywracać. Z tego powodu stworzyłem 3 narzędzia do pomocy w zarządzaniu bazą danych:

1. Narzędzie, które czyta pliki JSON z zagadkami oraz poziomami, i zapełnia bazę danych informacjami z tych plików.
2. Narzędzie służące do eksportowania informacji z bazy danych. Podajemy w nim dla jakiego dokumentu chcemy stworzyć kopię zapasową. Zapytania w Firestore są płytkie (*shallow query*), co oznacza, że zapytanie o podany dokument nie zwróci danych w zagnieżdżonych kolekcjach, które ten dokument posiada. Nie wiedząc ile zagnieżdżonych kolekcji może posiadać dokument, musiałem użyć rekurencji. Zaczynając od podanego dokumentu, narzędzie rekurencyjnie odpytuje wszystkie inne zagnieżdżone kolekcje i ich dokumenty, aż nie będzie żadnych innych zagnieżdżonych kolekcji. Wszystkie ściągnięte dane zapisuje do struktury danych złożonej z kluczy i wartości. W celu odróżnienia kolekcji od pól w dokumencie, narzędzie dodaje specjalne pole, w którym zapisuje czy obiekt jest zagnieżdżoną kolekcją czy zwykłym zbiorem kluczy i wartości. Na końcu zapisuje strukturę danych do pliku JSON.
3. Narzędzie, które importuje kopię zapasową stworzoną wcześniej opisanym narzędziem. W tym przypadku wszystkie informacje, które będą importowane są podane, więc zamiast rekurencji użyłem iteracji. Kolejność zapisywania do bazy danych podczas iteracji przebiega w następujący sposób:

   I Iterując po kolekcji, wchodzimy do dokumentu

   II Tworzymy strukturę danych, w której przechowamy pola dokumentu. Następnie iterujemy po dokumencie, sprawdzamy czy wartość to zagnieżdżona kolekcja, czy zwykłe pola dokumentu. W pierwszym przypadku wracamy do punktu I. W drugim, zapisujemy dane do stworzonej struktury danych.

   III Po zakończeniu iteracji po dokumencie, zapisujemy go do bazy danych wraz z jego kolekcjami

   Po pomyślnym przebiegu działania narzędzia, baza danych będzie odzwierciedlać importowany plik JSON.

Wszystkie te narzędzia były pisane w JavaScript, lecz w środowisku serwerowym Node.js. Jest to spowodowane ograniczeniem JavaScript w środowisku przeglądarkowym, które wynika z potrzeby ochrony danych

użytkownika przeglądarki. W przeglądarce nie możemy zapisywać plików na systemie użytkownika, więc musiałem to zrobić w Node.js, żeby mieć pełny dostęp do zasobów komputera.

3.1.3 Szybkość sprawdzania rozwiązania zagadki

Najczęściej wykonywaną akcją w Logo Quiz Web, która wymaga łączności z bazą danych, jest odgadywanie zagadki. Użytkownik spodziewa się, że aplikacja będzie działała szybko, więc po wpisaniu rozwiązania zagadki od razu chce uzyskać odpowiedź czy miał rację. Z powodów wcześniej opisanych problemów nie możemy zapisywać rozwiązań zagadek po stronie użytkownika.

Gdy użytkownik wpisze rozwiązanie i naciśnie przycisk sprawdzenia rozwiązania, wysyłane jest zapytanie do Cloud function. Na początku funkcja ta upewnia się czy użytkownik faktycznie posiada tę zagadkę w swojej bazie danych. Robi to ze względu na użytkowników, którzy próbują oszukiwać. Bez tego zabezpieczenia mogliby rozwiązywać zagadki, których nawet jeszcze nie odblokowali. Później funkcja ta pobiera informacje o zagadce wraz z jej poprawnym rozwiązaniem, następnie porównuje ją z rozwiązaniem użytkownika. Jeśli użytkownik poprawnie odgadł zagadkę, do jego bazy funkcja zapisuje niektóre pobrane wcześniej informacje takie jak opis odgadniętej zagadki. Dodatkowo po sprawdzeniu rozwiązania, niezależnie od jego poprawności, aktualizowane są statystyki użytkownika, który wysłał zapytanie. Na pierwszy rzut oka sprawdzanie czy rozwiązanie zagadki jest poprawne może wydawać się proste, ale jednak w jego skład wchodzi wiele kroków. Z tego powodu jedno zapytanie może wykonywać się długi czas zanim użytkownik otrzyma odpowiedź czy odgadł zagadkę.

Pierwszym rozwiązaniem, jakie zastosowałem, jest trzymanie zagadek w pamięci Cloud function (*cache*). Za pierwszym razem, gdy jakiś użytkownik odgaduje zagadkę jest ona zapisywana do tej pamięci. Efekt jest taki, że gdy Cloud function otrzyma ponowne zapytanie o tę samą zagadkę, nie będzie musiała na nowo ściągać informacji o tej zagadce z bazy danych, ale będzie mogła użyć tej w pamięci. Jednak okazało się, że serwery Cloud function są bardzo dynamiczne. Oznacza to, że są często uruchamiane i zamykane, więc cache nie będzie tak często wykorzystywany. Nawet w przypadku, gdy funkcja wykorzysta informacje z cache, czas który to zaoszczędzi będzie niezauważalny.

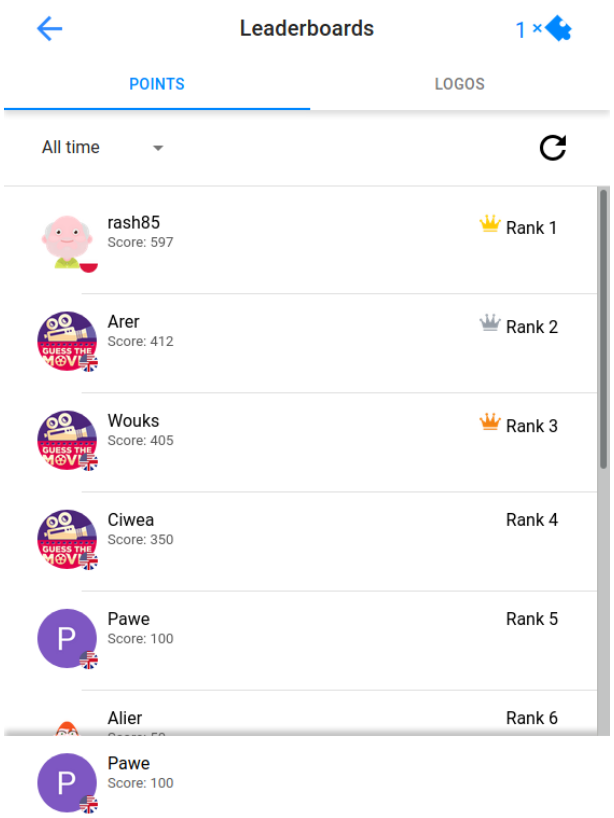
Drugie rozwiązanie używa kryptografii. Jawnych rozwiązań nie można przechowywać w bazie danych użytkownika, ale jeśli je zaszyfrujemy, nikt nie będzie mógł ich odszyfrować. W takim razie nawet jeśli te zaszyfrowane rozwiązania będą w stanie aplikacji, to użytkownik nie będzie mógł wyciągnąć z nich poprawnego rozwiązania zagadki. Przed wysłaniem zapytania do Cloud function, rozwiązanie użytkownika jest upraszczane poprzez usuwanie spacji oraz znaków specjalnych. Po takim uproszczeniu użyty jest algorytm szyfrujący SHA-256. Taką samą procedurę przeszły rozwiązania zmieszczone w bazie danych, które użytkownik ma w stanie aplikacji. Dzięki temu porównanie rozwiązań jest szybkie, ponieważ dzieje się całkowicie po stronie użytkownika i wówczas wie on od razu czy jego rozwiązanie jest poprawne. W międzyczasie wykonywane jest zapytanie do Cloud function w celu upewnienia się czy użytkownik odblokował zagadkę i podał poprawne rozwiązanie. Następnie funkcja aktualizuje bazę danych użytkownika.

Oba te rozwiązania są obecnie użyte w aplikacji Logo Quiz Web.

3.1.4 Pozycja użytkownika w rankingu względem jakiegoś pola

W rankingu znajdują się najlepsi gracze podzieleni pod względem różnych kryteriów, które opiszę w następnej sekcji. W tej sekcji zajmę się pozycją aktualnie zalogowanego gracza, który widnieje na dole infografiki 11. Na infografice można również zauważyć listę graczy posortowanych względem punktów razem z ich określoną pozycją w liście (*Rank*) po prawej stronie.

Docelowym założeniem było, aby aktualnie zalogowany gracz posiadał swoją pozycję rankingu, żeby mógł łatwiej określić jak dobrze mu się powodzi w grze. Okazało się to jednak niemożliwe ze względu na



Infografika. 11: Ekran rankingu w aplikacji Logo Quiz Web z listą najlepszych graczy. Na samym dole znajduje się wynik aktualnie zalogowanego gracza.

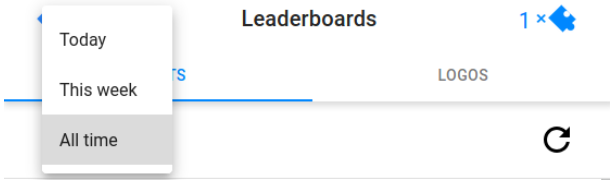
ograniczenia Firestore. W zwykłych bazach danych opartych na SQL znalezienie pozycji rekordu względem jakiegoś pola nie jest problemem, lecz nie jest to możliwe od strony bazy Firestore.

To ograniczenie jest związane ze sposobem indeksowania dokumentów w bazie. Może wydawać się to poważnym problemem, ale dzięki takiemu ograniczeniu baza danych jest w stanie działać bardzo szybko bez względu na ilość dokumentów znajdujących się w bazie. Jak pokazywałem we wcześniejszych sekcjach, aby otrzymać posortowaną listę musimy wykonać polecenie **orderBy()**. W rankingu robię to samo wraz z ograniczeniem ilości wyników do 10 za pomocą polecenia **limit**. Niestety wówczas jestem w stanie pokazać tylko najlepszych lub najgorszych graczy, bez możliwości sprawdzenia jaką pozycję ma aktualnie zalogowany gracz.

Należy pamiętać, że od strony bazy Firestore nie da się określić pozycji aktualnie zalogowanego gracza, jednak od strony klienta lub Cloud function jest to możliwe, ale wiąże się z bardzo dużą ilością przesyłanych danych. Rozwiązanie polega na pobraniu wszystkich dokumentów z kolekcji rankingu, posortowaniu ich i sprawdzeniu jaką pozycję ma aktualnie zalogowany gracz. Jak wspominałem, takie rozwiązanie nie jest optymalne, ponieważ w przyszłości graczy może być parę tysięcy. W takiej sytuacji każdy gracz musiałby pobierać parę tysięcy dokumentów, aby wyświetlić tylko 10 najlepszych graczy oraz swoją pozycję. Można by to usprawnić tworząc Cloud function, która będzie pobierała całą listę graczy i zapisywała do cache. Jednak trzymanie rankingu w cache będzie powodowało, że kolejne zapytania będą dostawały w odpowiedzi przestarzałą listę graczy, która może nie być zgodna z aktualnym stanem rankingu. Dlatego finalnie pozycja aktualnie zalogowanego gracza nie jest pokazywana.

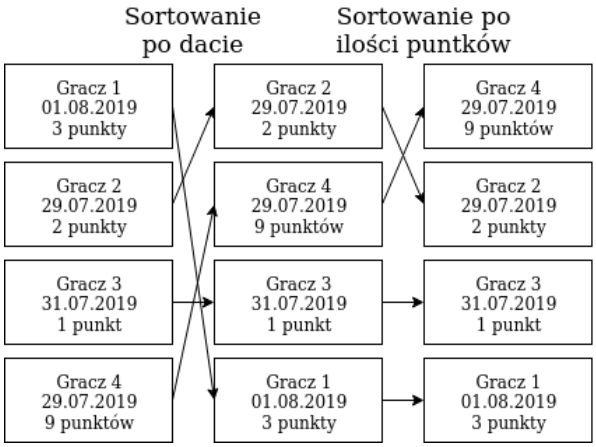
3.1.5 Sortowanie rankingu względem dwóch pól

Jak wspominałem w poprzedniej sekcji, ranking można wyświetlać na 6 sposobów: ranking dzisiejszy, tygodniowy lub cały, oraz dla dwóch kryteriów, punktów oraz rozwiązanych zagadek. Na infografice 12 przedstawiony jest ten podział wraz z dokładnym opisem sposobów podziału.



Infografika. 12: Opcje filtrowania rankingu.
Górna część ekranu rankingu posiada dwa główne podziały, mianowicie *Points*, który odnosi się do uzyskanych punktów, oraz *Logos*, który odpowiada za ilość rozwiązanych przez użytkowników zagadek. Dodatkowo po lewej stronie znajdują się trzy opcje, które określają zakres dat aktualnie wybranego podziału.

Każdy sposób podziału spośród wymienionych 6 posiada swój własny wpis w bazie danych. Ważne jest to, że wpisy te nie są resetowane wraz ze zmianą dnia lub tygodnia. Oznacza to, że przy odpytywaniu bazy danych potrzebne nam jest dodatkowe pole, które będzie określało w jakim dniu użytkownik ostatni raz rozwiązał zagadkę, aby wiedzieć czy pokazać jego pozycję w aktualnym kryterium wyświetlanego rankingu. Dzięki temu polu jesteśmy w stanie określić czy użytkownik rozwiązał zagadkę dzisiaj lub w tym tygodniu. Niestety ograniczenia Firestore wymagają, żeby przy każdym porównaniu pola za pomocą mniejszości lub większości w zapytaniu również znajdowało się sortowanie tego pola. W rezultacie musimy sortować po dacie odgadnięcia oraz ilości punktów lub ilości odgadnięć. Wykonywane w takiej sytuacji podwójne sortowanie wymusza, aby drugie sortowanie odbyło się w grupach, które się utworzyły w wyniku pierwszego sortowania. Przykład realizacji takiego podwójnego sortowania jest przedstawiony na infografice 13. Jak widać wynik nie będzie taki, jakiego użytkownik by się będzie spodziewał.



Infografika. 13: Przebieg sortowania po dwóch polach.
Sortowanie najpierw odbywa się na polu daty i tworzy grupy, które odnoszą się do konkretnej daty. Następnie sortowanie po ilości punktów odbywa się tylko w tych stworzonych grupach, a nie na całości zbioru.

Problem polega na tym, że odpytanie bazy uwzględnia tylko użytkowników, dla których data odgadnięcia jest większa lub równa dzisiejszemu dniu o północy. Jeśli porównanie daty odgadnięcia wykonywane jest na zasadzie większa (>, >=) bądź mniejsza (<, <=), Firestore wymusza sortowana według daty odgadnięcia. Rozwiązaniem tego było stworzenie dwóch dodatkowych pól, w których zapisany jest dzień odgadnięcia zagadki oraz pierwszy dzień tygodnia odgadnięcia wraz z godziną 00:00 dla obu. Dzięki temu zapytanie bazy danych nie porównuje większości bądź mniejszości, ale równości. W rezultacie nie ma wymogu sortowania po dacie, więc zwrócona lista graczy będzie w oczekiwanej przez użytkownika kolejności, która jest posortowana tylko po ilości punktów lub ilości odgadniętych zagadek.

3.2 Aplikacja

3.2.1 Nawigacja po ekranach

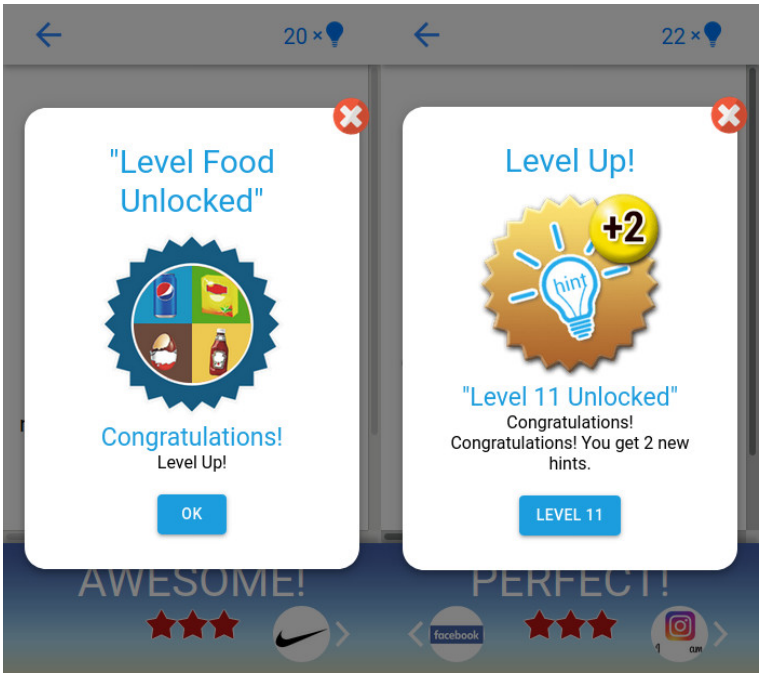
Ze wzgledu na to, że aplikacja musi być szybka i responsywna, zrezygnowałem ze standardowej metody nawigacji poprzez odpytywanie serwera o nową stronę. W jej miejsce zaimplementowałem własny router nawigacyjny, służący do trasowania ekranów aplikacji. Router stworzyłem za pomocą middleware w redux i słuchacza (*listener*) i jest on uruchamiany przy każdorazowej akcji wysyłanej do stanu aplikacji. Router mogłem zaimplementować jako reducer. Jednakże ze względu na fakt, że nawigacja nie zmienia nic w stanie aplikacji, tylko manipuluje historią przeglądarki, uznałem, że najlepszym miejscem dla routera będzie middleware. Zanim przedstawię jego działanie, chcę wyjaśnić, że opisywaną historię przeglądarki dzielę na dwie części. „Historia przed", czyli ta która poprzedza aktualną stronę. Jeśli użytkownik cofnie stronę, to pojawiające się strony są z „historii przed". Drugi rodzaj to „historia po". Jeśli użytkownik wejdzie na stronę us.edu.pl, a następnie cofnie do poprzedniej strony, to w „historii po" będzie znajdować się strona us.edu.pl.

Middleware działa w następujący sposób, bazując na 2 akcjach:

1. PUSH - odpowiada za nawigację w przód. Działa na podobnej zasadzie jak przechodzenie z jednej strony internetowej na drugą. Aktualna strona trafia do „historii przed", a nowa strona staje się aktualną.
2. GO_BACK - odpowiada za nawigację w tył. Aktualna strona jest dodawana do „historii po", a poprzednia strona staje się aktualną.

Są to akcje, które w prosty sposób manipulują historią przeglądarki. Dzięki temu w adresie przeglądarki będą pojawiały się nazwy ekranów. Jednak sam middleware nie będzie w stanie zmieniać ekranów aplikacji. Za zmianę ekranów odpowiedzialny jest listener zamieszczony na stanie aplikacji, który sprawdza czy aktualny adres został zmieniony. Jeśli adres się zmienił, wykonywane jest drzewo decyzyjne, które sprawdza jaki był poprzedni adres i na jaki został zmieniony. Porównując te dwa adresy listener stwierdza jaki ekran powinien pojawić się w aplikacji. Przykładowo, na stronie głównej adresem będzie "/". Użytkownik klika w ikonę ustawień, wykonywana jest akcja PUSH. Listener zostaje o tym powiadomiony. Poprzedni adres to "/", a następnym będzie "/settings". Dzięki temu listener wie, żeby zmienić ekran z głównego na ekran ustawień.

Dodatkowym problemem, moim zdaniem niemożliwym do bezpośredniego rozwiązania, okazała się nawigacja pozioma. W celu wyjaśnienia problemu w skrócie opiszę częściowe działanie aplikacji Logo Quiz Web oraz miejsce, w którym ten problem występował. Na samym początku użytkownik posiada dostęp tylko do klasycznego trybu gry. Wraz z poprawnym rozwiązywaniem zagadek odblokowuje nowe poziomy trybu klasycznego (co określone będzie jako odblokowanie poziomu). Co pewną ilość rozwiązań użytkownik odblokowuje również nowy tryb gry (odblokowanie dodatkowego trybu gry). Wraz z każdym odblokowaniem użytkownik zostaje o tym powiadomiony poprzez wyskakujące okno zilustrowane na infografice 14.



Infografika. 14: Wyskakujące okna po odblokowaniu nowego poziomu.
Po lewej stronie znajduje się odblokowanie poziomu, a po prawej odblokowanie dodatkowego trybu gry.

Docelowo, niebieski przycisk przy odblokowaniu poziomu powinien przenosić do odblokowanego poziomu, a w przypadku odblokowania dodatkowego trybu gry - do ekranu dodatkowych poziomów. Ta nawigacja jest przedstawiona na infografice 15. Nawigacja odblokowania nowego poziomu klasycznego (zielona strzałka na infografice 15) nie jest skomplikowana: wystarczy cofnąć stronę i podmienić dane. W przypadku odblokowania dodatkowego trybu gry nawigacja bardzo się komplikuje (czerwona strzałka na infografice 15): jak widać na infografice wymusza ona nawigację po poziomej linii. Poprawna nawigacja wymagałaby, aby cofnąć się trzykrotnie i przejść do poprawnego ekranu, lecz dla użytkownika taka nawigacja może wydać się błędem w aplikacji. Chciałem to rozwiązać poprzez zwykłe przeniesienie na poprawny ekran bez potrzeby cofania. Jednak okazało się to problematyczne, ponieważ z poziomu JavaScript nie da się usuwać nic z historii przeglądarki. Oznacza to, że gdyby użytkownik został przeniesiony na ekran z dodatkowymi trybami gry, a następnie cofnął stronę, zostałby wówczas przeniesiony z powrotem na początkowy ekran odgadywania zagadki. Ze względów intuicyjnych uznałem, że takie działanie nawigacji będzie miało negatywny wpływ na odbiór użytkownika i zrezygnowałem z tego rozwiązania. Niebieski przycisk na wyskakującym oknie odblokowania dodatkowego trybu gry nie robi nic poza zamknięciem tego okna.

Infografika. 15: Przebieg nawigacji po odblokowaniu poziomu.
Po lewej znajdują się kroki potrzebne, aby dojść do ekranu odgadywania zagadki z trybu klasycznego, a po prawej - kroki potrzebne do przejścia na ekran odgadywania zagadki z trybu dodatkowego. Zielona strzałka przedstawia scenariusz, w którym użytkownik odblokował nowy poziom trybu klasycznego, oraz miejsce, do którego przeniesie go kliknięcie w niebieski przycisk. Czerwona strzałka przedstawia odblokowanie dodatkowego trybu gry oraz przeniesienie po kliknięciu w niebieski przycisk.

3.2.2 Internacjonalizacja

Internacjonalizację wykonałem bez pomocy żadnych zewnętrznych bibliotek. Zrobiłem to za pomocą plików JSON, dlatego w aplikacji Logo Quiz Web każdy język posiada swój własny plik. Implementacja działa na podobnej zasadzie jak internacjonalizacja na systemach Android.

W aplikacji Logo Quiz Web zawsze pobierany jest język angielski, czasem dodatkowo jest pobierany jakiś inny język. Ten inny język jest pobierany za pomocą skryptu PHP, który zależnie od domeny, z której aplikacja wysłała zapytanie,, dostarcza odpowiedni plik językowy. Na przykład, gdy jesteśmy na domenie polskiej, to pobierany jest język angielski oraz plik, w którym znajdują się polskie tłumaczenia. W przypadku domeny angielskiej pobierane jest tylko angielskie tłumaczenie bez dodatkowych plików.

Do wyświetlania tłumaczeń stworzyłem pomocniczą funkcję, która przyjmuje klucz tłumaczenia, i jeśli jest taka potrzeba, dodatkowe zmienne, które pojawią się w tłumaczeniu. Funkcja ta najpierw próbuje uzyskać tłumaczenie z języka dodatkowego. Jeśli jednak taki klucz nie istnieje, pokazuje angielskie tłumaczenie. Dzięki temu nawet, gdy nie wszystko jest przetłumaczone, użytkownik nadal będzie widział napis; w tym wypadku będzie on angielski. Jak wcześniej wspominałem, do funkcji można również przekazywać dodatkowe zmienne. Najprostszym przykładem, który możne to zilustrować, jest powitanie w aplikacji. Każdy użytkownik posiada swoją własną nazwę, więc za każdym razem, gdy uruchomi aplikację, pokazuję mu napis “Witaj nazwa użytkownika”. W tym wypadku dodatkową zmienną będzie nazwa użytkownika.

3.2.3 Obsługa błędów

Obsługa błędów podobnie jak nawigacja, jest oparta na middleware i obsługuje 3 akcje, które można powiązać z poziomem błędu oraz miejscem jego wystąpienia.

Błędy dzielą się na dwa rodzaje, mianowicie błędy przewidziane i nie przewidziane. Błędy przewidziane to takie jak błąd podczas komunikacji z bazą danych lub Cloud function, oraz błędy pojawiające się podczas zmiany stanu aplikacji. Błędy przewidziane dodatkowo dzielą się jeszcze na dwie grupy: błędy w aplikacji oraz błędy w Cloud function. Podział ten ma służyć łatwiejszemu ustaleniu, gdzie leży problem. Błędy nie przewidziane (opisane szczegółowo poniżej), które zostają wychwycone przez przeglądarkę, odróżniają się od powyżej opisanych i zostają przypisane do odrębnego rodzaju, w celu szybkiego wykrycia i stworzenia z nich błędu przewidzianego.

Błędy o poziomie niskim zostają pokazane jako małe, nie intruzywne powiadomienie na dole ekranu, które znika po paru sekundach. Natomiast błędy o wysokim poziomie pojawiają się w wyskakującym okienku, które użytkownik musi sam zamknąć. Do middleware są przekazywane tylko błędy przewidziane; z reguły będą to błędy po niepomyślnym zapytaniu do bazy danych. Zastosowanie tylko tej obsługi spowoduje, że wszystkie błędy, które się wydarzą podczas zmiany stanu aplikacji przez reducer nie zostaną przechwycone, ponieważ nie wiemy jakie dane zostaną przekazane do stanu. Rozwiązaniem takiej sytuacji było zastosowane przeze mnie umieszczenie middleware w bloku *try*, który przechwyci wszystkie inne błędy, które wydarzą się podczas zmiany stanu aplikacji. Tym sposobem większość błędów zostanie obsłużona, lecz nigdy nie jesteśmy w stanie przewidzieć, jakie błędy się wydarzą podczas działania aplikacji. Na szczęście przeglądarki posiadają wbudowany słuchacz błędów (*window.onerror*), który jest uruchamiany za każdym razem, gdy w aplikacji wydarzy się błąd, który nie został przewidziany.

W sytuacji, gdy aplikacja obsługuje błąd, każdorazowo wysyła go do systemu raportowania błędów, który opiszę w następnym paragrafie. Jedno pytanie, które może się nasunąć, to takie dlaczego tworzyć 3 miejsca do obsługi błędów, skoro jeden słuchacz (window.onerror) mógłby obsłużyć je wszystkie. Takie rozwiązanie mogłoby działać, ale tracimy jedną z najważniejszych rzeczy w raportowaniu błędów, mianowicie dane wejściowe. Bez tych danych bardzo trudno stwierdzić, co spowodowało błąd. Gdy błąd wystąpi w jednym z przewidzianych miejsc, aplikacja przesyła dane wejściowe, które go spowodowały. Dzięki temu jest duża szansa, że użycie tych samych danych wejściowych spowoduje ten sam błąd i programista będzie mógł go naprawić.

Każda większa aplikacja potrzebuje systemu raportowania (*log*), dzięki któremu będzie wiadomo, że aplikacja nie działa poprawnie po stronie użytkownika. Jest wiele takich rozwiązań, niestety większość z nich jest płatna. Ale dowiedziałem się że jest inny, darmowy sposób, poprzez który logi dałoby się zaimplementować w aplikacji. Tym sposobem jest Google Analitycs [9]. Jest to narzędzie, które umożliwia właścicielom stron internetowych zbieranie informacji na temat ich stron. Do obsługi błędów używam jedynie zdarzeń (*Event*). Z reguły są one używane do zbierania informacji na temat tego, w jaki sposób użytkownik używa stronę, ale w wypadku Logo Quiz Web zostały użyte do raportowania błędów. Niestety te zdarzenia są bardzo ograniczone, jeśli chodzi o możliwość agregacji danych, ale w celu raportowania błędów wystarczają. Każde takie zdarzenie posiada informację na temat poziomu błędu, danych wejściowych, oraz tzw. *stack trace*, w której skład wchodzą nazwa błędu oraz linijki kodu, w których ten błąd wystąpił.

4 Podsumowanie

Celem mojej pracy było przedstawienie procesu tworzenia nowoczesnych aplikacji internetowych oraz opisanie popularnych narzędzi używanych w tym celu. Większość wiedzy, którą zawarłem w tej pracy, wywodzi się z doświadczenia nabytego przeze mnie podczas tworzenia aplikacji Logo Quiz Web. Niektóre problemy napotkane podczas tworzenia aplikacji opisałem tutaj wraz z moim ich rozwiązaniem. Udało mi się stworzyć aplikację, której funkcjonalności są obecnie w pełni dopracowane i czynią aplikację gotową do opublikowania.

Aplikacja jest ciągle rozwijana: dodawane są funkcjonalności, które istnieją już w oryginalnej wersji mobilnej. Obecnie pracuję nad reklamami w aplikacji oraz nad stworzeniem koła, które użytkownik może zakręcić, aby otrzymać bonusy w grze. Największym potencjalnym zadaniem, które może się pojawić w przyszłości aplikacji, jest stworzenie nowej wersji mobilnej na Android. Ta aplikacja mobilna za pomocą wbudowanych narzędzi będzie wyświetlała opisaną w niniejszym projekcie inżynierskim aplikację internetową tak jak przeglądarka. Dzięki temu nie trzeba będzie zmieniać dużo kodu w aplikacji, ale za to będzie można dodać do niej powiadomienia, reklamy w formie filmów oraz możliwość kupowania bonusów.

Referencje

[1] *What is an API? In English, please.* URL https://www.freecodecamp.org/news/what-is-an-api-in-english-please-b880a3214a82/. 20.04.2019.

[2] *The Most Used Responsive Breakpoints in 2017 Of Mine.* URL https://medium.com/@uiuxlab/the-most-used-responsive-breakpoints-in-2017-of-mine-9588e9bd3a8a. 20.04.2019.

[3] *Firefox Browser Developer Edition.* . URL https://www.mozilla.org/en-US/firefox/developer/. 04.08.2019.

[4] *A JavaScript library for building user interfaces.* URL https://reactjs.org/. 08.05.2019.

[5] *Redux is a predictable state container for JavaScript apps.* . URL https://redux.js.org/. 14.04.2019.

[6] *Redux Thunk middleware allows you to write action creators that return a function instead of an action.* . URL https://github.com/reduxjs/redux-thunk/. 17.04.2019.

[7] *Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud Platform.* . URL https://firebase.google.com/docs/firestore/. 12.08.2019.

[8] *Firebase helps mobile and web app teams succeed.* . URL https://firebase.google.com/. 12.08.2019.

[9] *Google Analytics is a web analytics service offered by Google that tracks and reports website traffic.* URL https://marketingplatform.google.com/about/analytics/. 07.10.2019.