Wprowadzenie do aplikacji Internetowych

laboratorium 5

Cel zajęć:

Celem laboratorium jest przećwiczenie zagadnień związanych z obsługą rzeczywistych danych produkcyjnych pochodzących z serwera z danymi. Nasze dane będą w sposób trwały przechowywane po stronie Backendu.

Drugim tematem poruszonym w trakcie zajęć będzie kwestia uporządkowania naszej aplikacji klienckiej pod katem wyświetlanych komponentów. Aktualnie nasza aplikacja kliencka wyświetla wszystkie powstałe komponenty na jednym ekranie. Jest to niewygodne i sprawia wrażenie chaosu w aplikacji. Dzisiejsze laboratorium wykorzystując koncepcje routingu ma za zadanie uporządkowanie tego zagadnienia.

Routing

Angular pozwala nam na przechodzenie z jednego widoku do drugiego w miarę jak użytkownik podejmuje odpowiednie działania na stronie. Przeglądarka jest takim modelem w którym możemy nawigować. Po wpisaniu jakiegoś adresu możemy na niego przejść, po wybraniu jakiegoś odnośnika również. Router Angulara korzysta z tej koncepcji. Może traktować konkretny adres jako polecenie przejścia do konkretnego widoku. Może przekazywać do niego opcjonalnie różne parametry w zależności od naszych potrzeb które pomogą w tym widoku określić co konkretnie użytkownik chce wyświetlić. Możemy podstawić router do linków na naszej stronie tak, żeby kliknięcie w nie powodowało przeniesienie nas do konkretnego widoku.

Routing pozwala zawrzeć pewne aspekty stanu aplikacji w adresie URL. Dla aplikacji front-end jest to opcjonalne - możemy zbudować pełną aplikację bez zmiany adresu URL. Dodanie routingu pozwala jednak użytkownikowi przejść od razu do pewnych funkcji aplikacji. Dzięki temu aplikacja jest łatwiej przenośna i dostępna dla zakładek oraz umożliwi użytkownikom dzielenie się linkami z innymi.

Routing ułatwia:

- Utrzymanie stanu aplikacji
- Wdrażanie aplikacji modułowych
- Stosowanie ról w aplikacji (niektóre role mają dostęp do określonych adresów URL)

Konfiguracja Routingu sprowadza się do:

1. Importu odpowiednich modułów `import { RouterModule, Routes } from '@angular/router';`

Router jest opcjonalnym serwisem który pokazuje określony widok dla zdefiniowanego adresu. Nie jest częścią biblioteki *core* Angulara.

2. *Konfiguracja*

Aplikacje wykorzystujące routing mają jedną instancję serwisu routera. Nie ma on jednak określonych adresów, czy też ścieżek, po których powinien nawigować. Musimy mu je więc skonfigurować jeśli chcemy żeby cokolwiek robił. Dokonujemy tego w pliku *app.module.ts*.

Przykładowy wpis wyglądałby tak:

```
const appRoutes: Routes = [
  { path: 'glowna', component: AComponent },
  { path: 'test/:id',      component: BComponent },
  { path: 'testy',   component: CComponent,   data: { key: 'ABC' }   },
  { path: '',    redirectTo: '/testy',     pathMatch: 'full'    },
  { path: '**', component: PageNotFoundComponent }
```

Tablica *appRoutes* określa do jakich komponentów nawigować po określonych ścieżkach. Przekazujemy ją do metody *RouterModule.forRoot()* żeby router ją zaimportował i wprowadził w życie.

Ścieżki które prowadzą do określonych komponentów nie są poprzedzone znakiem /. Router buduje ostateczne ścieżki pozwalając odnosić się do relatywnych i konkretnie określonych ścieżek kiedy nawigujesz pomiędzy widokami swojej aplikacji.

Paramter *:id* w drugiej ścieżce pozwala na podstawienie w to miejsce jakieś wartości i późniejsze odniesienie się do niej właśnie przez *id*. Kiedy wpiszesz */test/42* komponent *BComponent* będzie wiedział, że dostał *id* o wartości 42.

Właściwość *data* w trzeciej ścieżce zawiera w sobie jakieś specyficzne dane przypisane dla konkretnej ścieżki. Również będą one dostępne dla widoku do którego nawiguje. Powinno się jej używać do takich rzeczy jak tytuły podstron, tekstów do nawigacji czy innych statycznych danych, tylko do odczytu.

Czwarta, pusta ścieżka prezentuje domyślną lokalizację aplikacji do której użytkownik powinien zostać przekierowany zaraz po jej odpaleniu.

Ostatnia ścieżka gdzie znajduje się podwójny znak * zostanie użyta kiedy wprowadzony przez użytkownika adres nie będzie pasował do żadnej podanej do tej pory ścieżki w naszym routerze. Jest to więc najzwyczajniej odpowiednik przekierowania do strony informującej o błędzie 404.

Kolejność podania tych ścieżek ma znaczenie. Router użyje pierwszej pasującej ścieżki żeby odnieść się do podanego mu adresu. Dlatego też te bardziej precyzyjne, skomplikowane ścieżki powinny być umieszczone poniżej tych bardziej ogólnych.

3. Określenie miejsca umieszczenia Router outlet

Kiedy przekażesz taką konfigurację do routera w swojej aplikacji i wpiszesz URL */testy* wyświetlony zostanie *CComponent* w miejscu po tagu *<router-outlet>* który należy umieścisz w swojej aplikacji. Zazwyczaj wygląda to tak, że umieszcza się go w *app.component.html* gdyż jest to główny jej komponent.

4. Definiowanie połączeń między trasami

Masz już skonfigurowane i użyte ścieżki do których router ma się odnosić. Musisz jednak dodać do swojej strony jeszcze możliwość nawigowania po nich. Oczywiście ktoś może przechodzić na konkretne podstrony poprzez wpisywanie konkretnego adresu w przeglądarce ale raczej dużo bardziej normalnym rozwiązaniem z którego korzysta większość użytkowników będzie klikanie w odpowiednie odnośniki na stronie.

Dodaj linki do tras za pomocą dyrektywy RouterLink .
Na przykład poniższy kod definiuje łącze do trasy w ścieżce component-one .

```
<a routerLink="/component-pierwszy">PierwszyKomponent</a>
```

Alternatywnie możesz nawigować do trasy, wywołując funkcję navigate na routerze:

```
this.router.navigate(['/ component-pierwszy']);
```

Przykład:

```
<h1>Angular Router</h1>
 <nav>
  <a routerLink="/glowna" routerLinkActive="active">Strona Głowna </a>
  <a routerLink="/testy" routerLinkActive="active">Testy</a>
 </nav>
 <router-outlet></router-outlet>
```

Jeśli chcesz żeby twoje linki były bardziej dynamiczne powinieneś tak jak powyżej pokazałem zastosować routerLinkActive który pozwoli na nadanie klasy active i wizualne wyróżnienie aktywnych ścieżek.

Stan

Po każdym udanym cyklu nawigacji Angular buduje drzewko obiektów ActivatedRoute które określają obecny stan routera. Możesz odwołać się do obecnego RouterState, czyli właśnie tego stanu, z dowolnego miejsca w twojej aplikacji używając serwisu Router i jego właściwości: routerState. Każdy obiekt ActivatedRoute w RouterState dostarcza metody, żeby odnieść się do nadrzędnego elementu, elementów potomnych czy sąsiadujących w hierarchii nawigacji.

ActivatedRoute

Dzięki temu serwisowi możemy odnieść się do wielu, czasem bardzo przydatnych informacji odnośnie lokalizacji w której obecnie się znajdujesz. Są to między innymi url który jest tablicą składającą się z poszczególnych członów ścieżki w której się znajdujesz, data czyli obiekt z dodatkowymi danymi dostarczony do ścieżki, paramMap – mapa pozwalającą na odwoływanie się do parametrów dostarczonych do ścieżki. parent to oczywiście element ActivatedRoute będący rodzicem, firstChild z kolei zawiera pierwszy element potomny a children je wszystkie.

Tworzenie backendu:

Potrzebujemy wiec jakiegoś mechanizmu zapisu danych w sposób trwały. W tym celu musimy skorzystać z funkcjonalności serwera aplikacyjnego (backend) z którym będziemy komunikować się za pomocą wywołań REST. Interesuje nas oczywiście tylko rozwiązanie bazujące na stosie JS.

Są trzy sposoby na realizacje tego zagadnienia:

- Skorzystanie z znanego Ci już json-server i modyfikacje lokalnego pliku JSON.
- Skorzystanie z gotowej platformy zbudowanej w oparciu o NodeJS – Google Firebase (ścieżka łatwiejsza)
- Samodzielne zbudowanie serwera aplikacyjnego w oparciu o NodeJS i moduł Express. (ścieżka trudniejsza)

Ścieżka łatwiejsza.

Do tego celu użyjemy gotowe środowisko backendowe – Firebase. W zasadzie cała nasza aktywność sprowadzi się do stworzenia konta oraz przygotowania danych w dostępnej bazie danych. Do wyboru mamy bazę typu RealTime lub FireStore gwarantujące aktualizacje naszego Frontendu w przypadku modyfikacji danych w bazie.

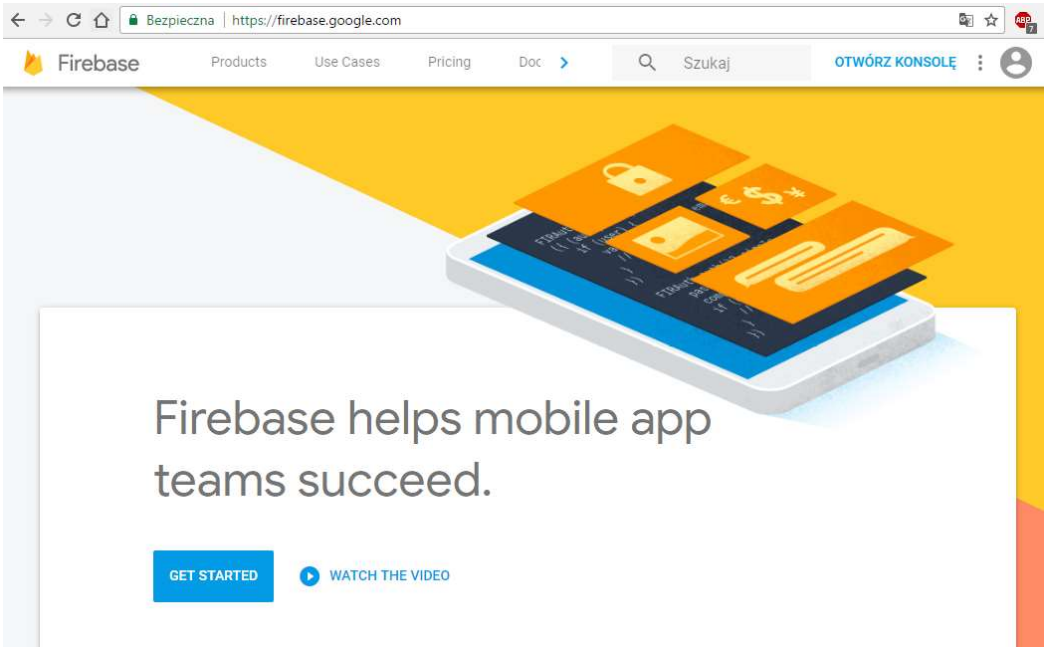
Ze względu na chęć przećwiczenia współpracy z obiektem httpClient jeden z projektów zrealizujemy w oparciu o współprace z fejkowym serwerem jasonplaceholder znajdującym się pod adresem https://jsonplaceholder.typicode.com/, a który dostarcza kilka przykładowych grup danych.

Informacje wstępne na temat Firebase.

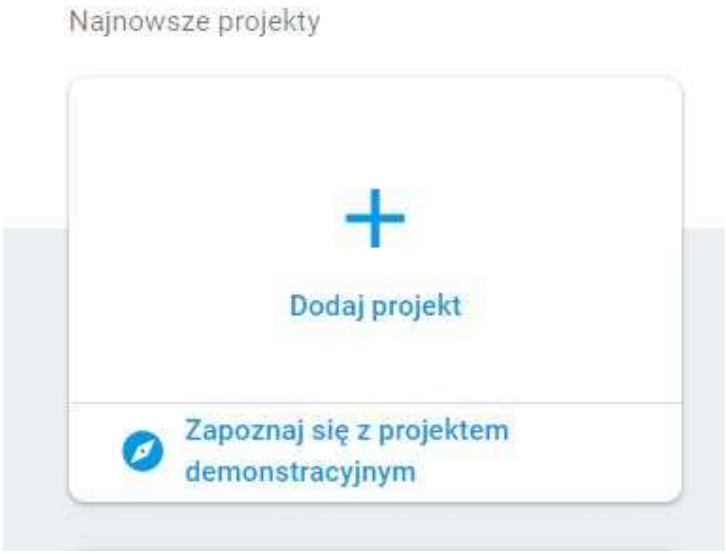
Firebase to tzw. **BaaS** (Backend as a Service), który umożliwia min. przechowywanie danych w formacie JSON oraz plików binarnych (np. jpg, mp4). Firebase dostarcza nam dwie bazy danych typu real-time database, gdzie komunikacja z serwerem jest oparta o Websockets, dzięki czemu po aktualizacji danych, klient automatycznie dostaje najświeższe dane. Obsługa jest banalnie prosta, w podstawowym zakresie można niemalże wszystko wygenerować z

panelu użytkownika! Dobre rozwiązanie na początek, gdy chce się postawić w szybkim czasie serwer z danymi a nie ma się czasu lub umiejętności aby zrobić to samodzielnie.

Zanim rozpoczniemy komunikację z serwerem, musimy oczywiście sobie go wcześniej przygotować. Rozpoczynamy od odwiedzenia strony www.firebase.com, założenia konta, a następnie po zalogowaniu klikamy przycisk „**OTWÓRZ KONSOLĘ**” w prawym górnym rogu:



Następnie w kolejnym widoku, klikamy „UTWÓRZ PROJEKT”:



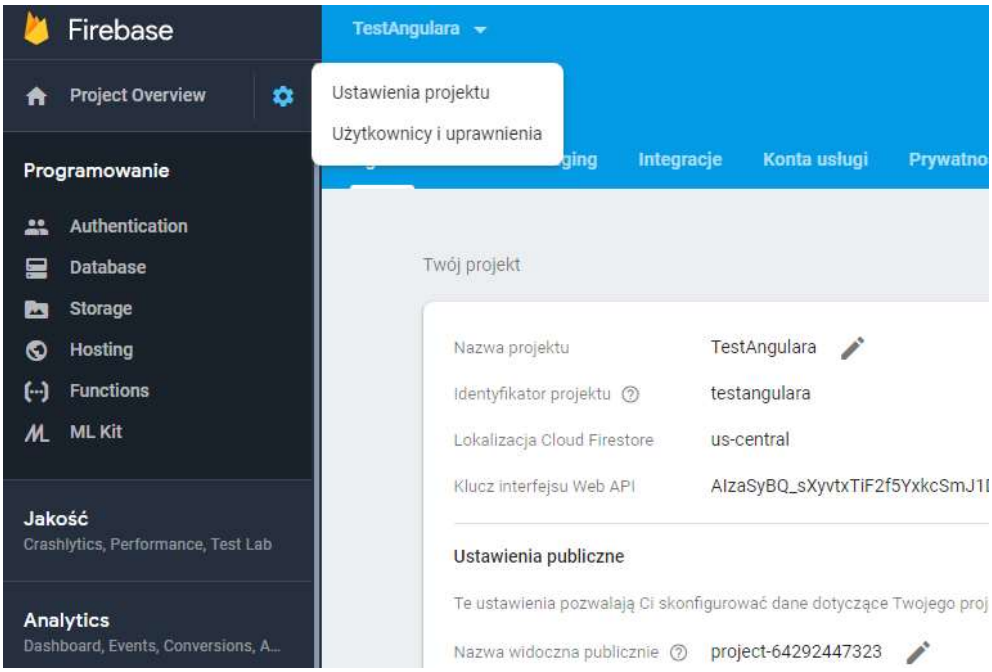
W okienku podajemy nazwę projektu, akceptujemy regulamin i klikamy „UTWÓRZ PROJEKT” i czekamy cierpliwie na powstanie projektu.

AngularFire

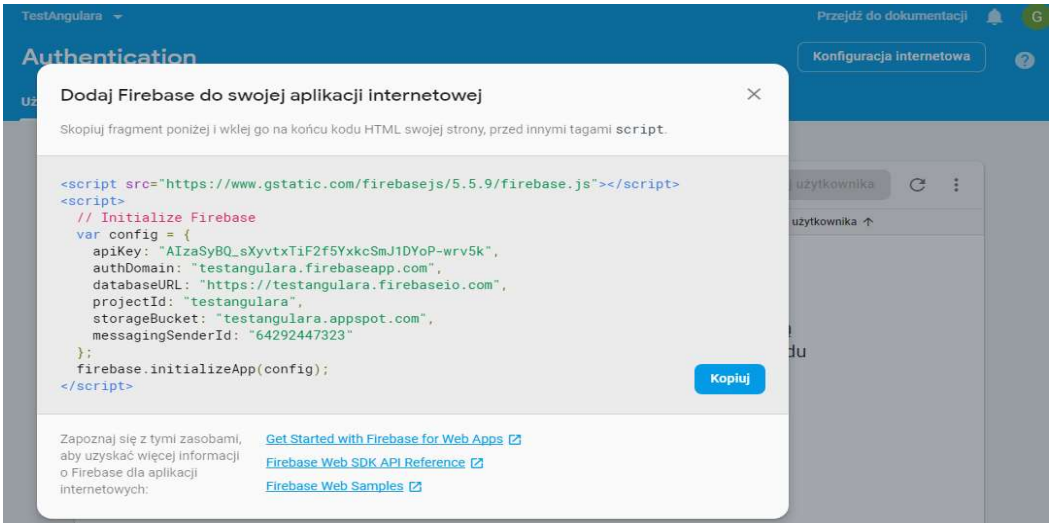
W celu komunikacji z naszym kontem w Firebase z poziomu tworzonej aplikacji webowej należy użyć dedykowanej biblioteki – **AngularFire**, który jest wrapperem na bliblioteke **firebase.js.** Pozwoli nam się zsynchronizować z danymi w czasie rzeczywistym, oraz dostarcza bardzo dobre API do logowania i monitorowania uwierzytelniania użytkownika. Rozpoczynamy od instalacji paczki:

```
npm install firebase @angular/fire –save
```

Po instalacji, wracamy do panelu Firebase i przechodzimy do ustawień projektu, klikając na zębatkę obok „PROJECT OVERVIEW” w lewym menu:



Scrollujemy nieco w dół i klikamy „Dodaj Firebase do swojej aplikacji internetowej”. Pojawia się okienko z danymi projektu:



Kopiujemy sam obiekt przypisany do „var config" i zapisujemy np. w jakimś pliku tekstowym. Reszta nas nie interesuje.

Podpięcie się pod Firebase

Wracamy do aplikacji angularowej i wklejamy plik konfiguracyjny do plików **environment.ts oraz environment.prod.ts** w katalogu src/environments. Jest to dobre miejsce na trzymanie takich globalnych ustawień. Obiekt możemy przypisać np. do pola „firebaseConfig":

```
// The file contents for the current environment will overwrite these during build.
// The build system defaults to the dev environment which uses `environment.ts`, but if you do
// `ng build --env=prod` then `environment.prod.ts` will be used instead.
// The list of which env maps to which file can be found in `.angular-cli.json`.

export const environment = {
    production: false,
    firebaseConfig: {
        apiKey: "AIzaSyBQ_sXyvtxTiF2f5YxkcSmJ1DYoP-wrv5k",
        authDomain: "testangulara.firebaseapp.com",
        databaseURL: "https://testangulara.firebaseio.com",
        projectId: "testangulara",
        storageBucket: "testangulara.appspot.com",
        messagingSenderId: "64292447323"
  };
};
```

Następnie przechodzimy do app.module.ts i importujemy następujące moduły **AngularFireModule)**.

```
import { AngularFireModule } from "@angular/fire";
import { environment } from '../environments/environment';
```

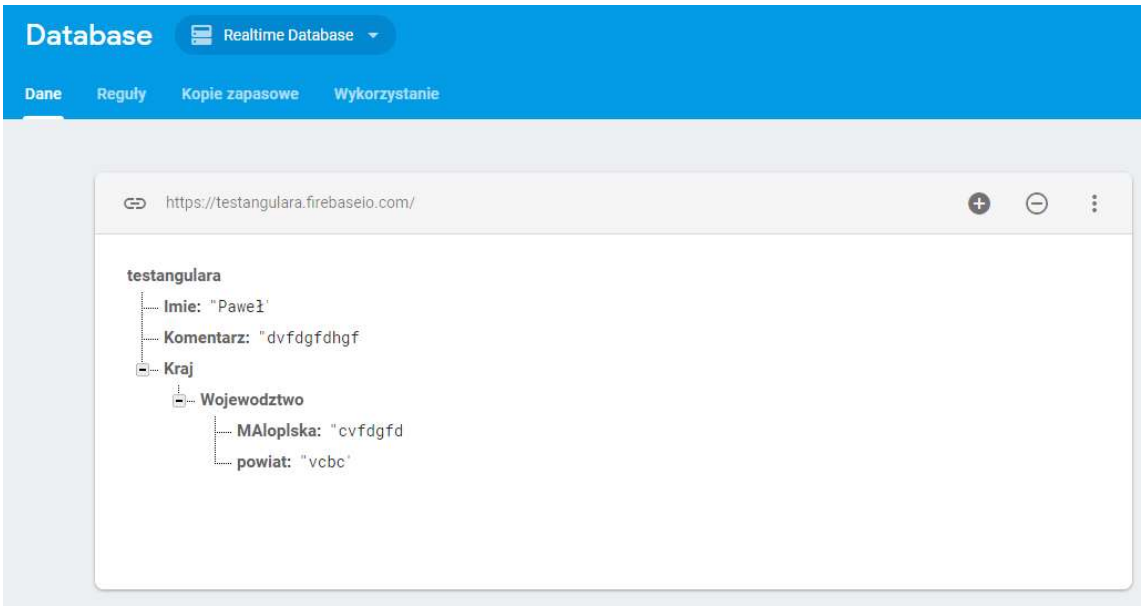
```
……..
imports: [
   AngularFireModule.initializeApp(environment.firebase),
],
```

Zwróć uwagę na wywołanie metody initializeApp z obiektem konfiguracyjnym, który zapisaliśmy w pliku environment.ts i environment.prod.ts. Teraz nasza aplikacja staje się świadoma backendu dostarczonego przez Firebase.

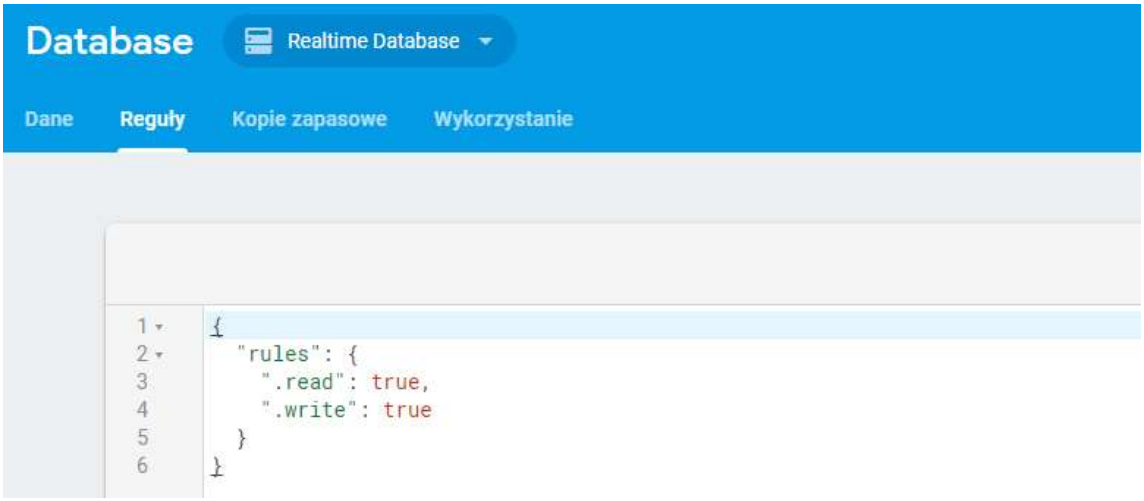
Obsługa bazy danych

Głównym celem użycia FireBase było skorzystanie z wbudowanej bazy danej, która miała za zadanie wspierać operacje CRUD na obiektach JASON.

Pierwszym krokiem jest stworzenie w bazie przykładowych danych. Możemy użyć RealTime DataBase lub Cloud Firestore. W obu przypadkach tworzenie danych w bazie jest banalnie proste.



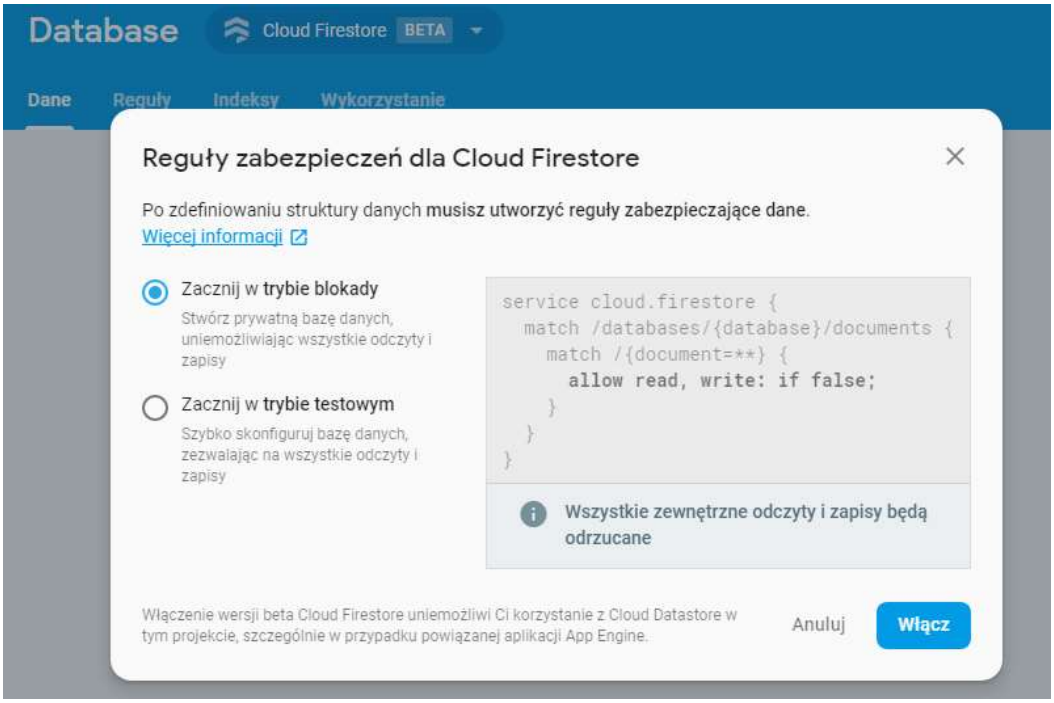
Nie wolno zapomnieć o ustawieni reguł dostępowych – domyślnie obie wartości są ustawione na false.



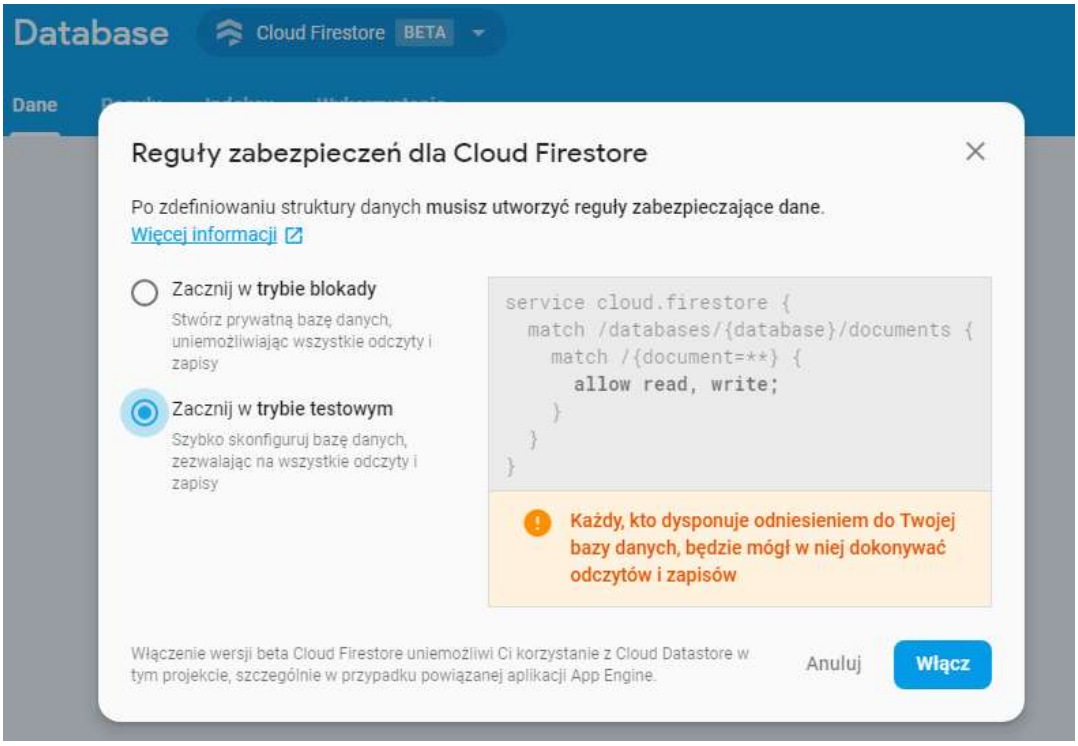
lub tak gdy zrezygnowaliśmy z autoryzacji

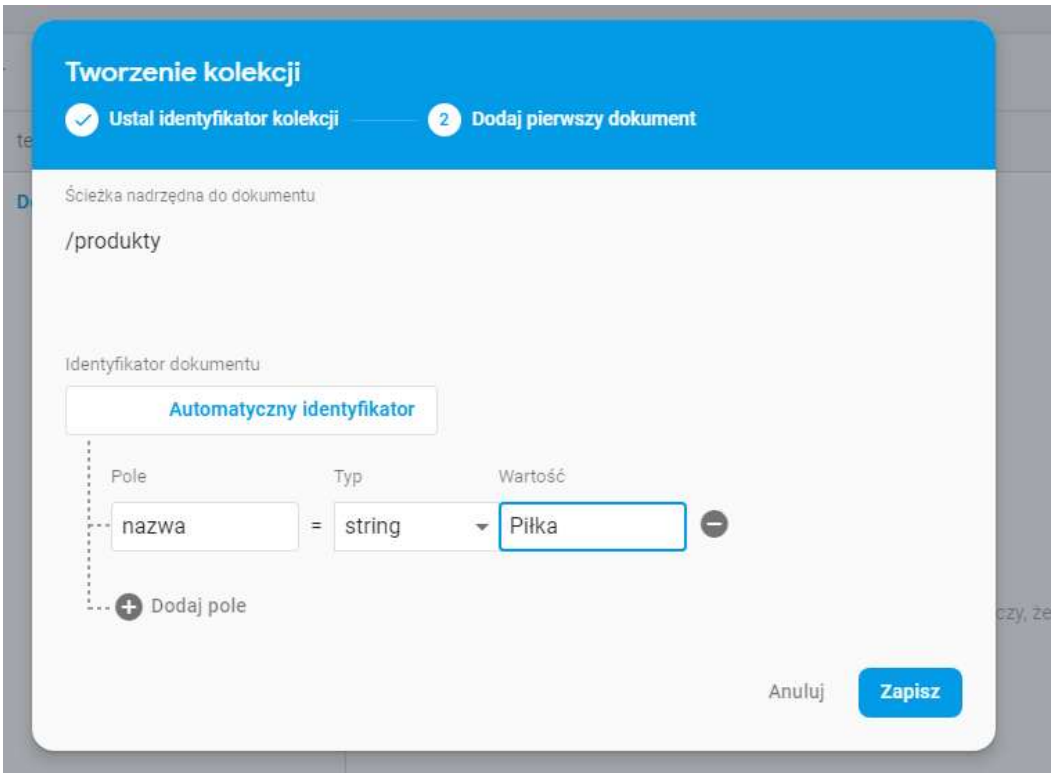
```
{
  "rules": {
    ".read": "auth == null",
    ".write": "auth == null"
  }
}
```

W przypadku wybrania wersji Cloud



Lub





Po stworzeniu bazy w Firebase przechodzimy do Angulara.

W zależności od użytej bazy :

W module app.module.ts importujemy albo

```
import { AngularFireStore } from '@angular/fire/store';
```

albo

```
import { AngularFireDatabase} from '@angular/fire/database';
```

w zależności od tego, z której z baz chcemy korzystać.

```
 ...

 @NgModule({
  imports: [
    AngularFireModule.initializeApp(environment.firebaseConfig),
    BrowserModule,
    AppRoutingModule,
    AngularFireDatabaseModule   lub  AngularFirestoreModule
  ],
 ...
 })
```

Teraz w odpowiednim komponencie należy poprzez dedykowaną usługę dostarczyć zawartość bazy danych.

Przykładowa implementacja

```
import { AngularFireDatabase} from '@angular/fire/database';

export class GRFireStoreService {
  constructor(private db: AngularFireDatabase) { }
}
```

AngularFireDatabase

do obsługi pojedynczego Objektu

– wiazanie referencji z obiektem w bazie danych - odczyt pojedy

```
daneRef: AngularFireObject<any>;
this.daneRef = db.object('student');
        // lub tak
Observable<any> dane = db.object('student').valueChanges();
```

– tworzenie obiektu w bazie:

```
const daneRef = db.object('student’);
 daneRef.set({ name: 'GR'});
```

– edycja danych w bazie

```
const daneRef = db.object(' student’);
daneRef.update({ item2: 'cos tam' });
```

– Usuniecie obiektu:

```
const daneRef = db.object('student');
daneRef.remove();
```

obsługa Listy (dane jako lista elementów)

Odczyt danych :
+ pobieranie danych jako tablica obiektów JSON bez metadanych (tylko same dane)

```
daneRef: Observable<any[]>;
this.daneRef = db.list('students').valueChanges();
```

+ pobieranie danych jako tablica obiektów JSON wraz z metadata
(DatabaseReference oraz dokument id, lub index tablicy):

```
daneRef: Observable<any[]>;
this.daneRef  = db.list('students').snapshotChanges();
```

Dodanie nowego elelmentu do listy:

```
const daneRef = db.list('students');
daneRef.push({ name: 'GR', item: 'cos tam dodaje' });
```

Aktualizacja listy:

```
const daneRef = db.list('students');
daneRef.set('key', { name: 'ktos inny', item: 'cos innego' });
```

+ lub zmiana tylko specyficznej wartosci

```
const daneRef = db.list('students');
daneRef.update('key', { name: 'kolejny ktos' });
```

Usuniecie obiektu z listy:

```
const daneRef = db.list('students');
daneRef.remove('key');
```

Usuniecie calej listy:

```
const daneRef = db.list('students');
daneRef.remove();
```

AngularFireStore

dla documentu

odczyt danych:

```
daneRef: AngularFirestoreDocument<any>;
this.daneRef = db.doc('students');
        // lub
Observable<any> daneRef = db.doc('students').valueChanges();
```

– Dodanie/utworzenie dokumentu :

```
const daneRef = db.doc('students');

daneRef.set({ name: 'GR'});
```

– Aktualizacja

```
const daneRef = db.doc('students');
daneRef.update({ item2: 'cos tam' });
```

– Usuniecie:

```
const daneRef = db.doc('students');
daneRef.delete();
```

AngularFireStore

Kolekcja (Collection)

Odczyt danych :
+ pobieranie danych jako tablica obiektów JSON bez metadanych (tylko same dane)

```
daneRef: Observable<any[]>;
this.daneRef = db.collection('students').valueChanges();
```

+ pobieranie danych jako tablica obiektów JSON wraz z metadata
(DatabaseReference oraz dokument id, lub index tablicy):

```
daneRef: Observable<any[]>;
this.daneRef  = db.collection('students').snapshotChanges();
```

Dodanie nowego elementu do kolekcji:

```
const daneRef = db.collection('students');
const dane = { name: 'GR', item: 'cos tam dodaje' };
daneRef.add({ ... dane });
```

Aktualizacja kolekcji:

```
const daneRef = db.collection('students');
daneRef.doc('id').set({ name: 'ktos inny', item: 'cos innego' });
```

+ lub zmiana tylko specyficznej wartosci

```
const daneRef = db.collection('students');
daneRef. doc('id').update({ name: 'kolejny ktos' });
```

Usuniecie obiektu z kolekcji:

```
const daneRef = db.collection('students');
daneRef.doc('id').delete();
```

Zadanie 1. Pod adresem https://jsonplaceholder.typicode.com/ znajduje się fejkowy serwer aplikacyjny udostępniający swoje dane za pomocą REST API. Zapoznaj się z jego zawartością a następnie napisz aplikację frontonową, która pobierze i wyświetli zawartość dwóch endpointów serwera /posts oraz /photos. Do tego celu wykorzystaj httpClient (a nie FetchAPI znane Ci z wcześniejszych zajęć). Dla posts dodaj również możliwość wysyłania posta na serwer. Niech zawartość każdego z endpointów będzie wyświetlana na oddzielnej podstronie. W tym celu użyj mechanizmu Routingu. Przygotuj menu nawigacyjne zawierające 3 pozycje: Home (strona głowna z informacjami ogólnymi, Posty (zawierająca kolekcje pobranych postów) oraz Zdjecia (wyświetlająca informacje o pobranych zdjęciach).

Dla zdjęcia dodaj możliwość wyświetlania wybranego zdjęcia na oddzielnym widoku. Zastanów się jak powinien wyglądać adres takiego widoku.

Niech menu będzie niezmienne na każdej z podstron (wyświetlane na samej górze ekranu).
(2 pkt)

Zadanie 2. Biuro Turystyczne – uporządkowanie widoków.

Wracamy do naszej główne aplikacji BiuroTurystyczne-Wycieczki, którą zaczęliśmy rozwijać na lab 4. Wykorzystując nabytą w zadaniu 1 umiejętność zarządzania nawigacja po stronie – uporządkuj zawartość poszczególnych ekranów aplikacji. Niech będą jako niezależne ekrany wyświetlające: widok startowy naszego biura, widok z ofertą wycieczek (zawierający listę

wycieczek filtrami), dodawanie nowej wycieczki, podgląd zawartości koszyka oraz wiod z historia zakupionych i odbytych wycieczek. Oczywiście lista nawigacyjne zrealizowana jako menu responsywne widziane na każdym widoku tak aby w łatwy sposób nawigować po stronie. Proponuje zastosowanie adresacji opartej na REST API (omawianej na wykładzie 7). Skonfiguruj moduł Routingu tak aby możliwe było przemieszczanie się pomiędzy poszczególnymi widokami. Oprócz listy nawigacyjnej wyświetl także nagłówek zawierający dane z koszyka (ilość i suma wybranych wycieczek), symbol powiadomienia o zbliżajcie się wycieczce oraz w przyszłości dane o uzytkowniku **(2 pkt)**

Widok startowy biuraTurytycznego powinien zawierać przygotowaną „artystycznie" wizytówkę naszego Biura + dane kontaktowe + mapa z lokalizacją. **(1 pkt)**

Zadanie 3. Wycieczka – widok szczegółowy. (3 pkt)

Aktualnie wszystkie informacje o wycieczce dostępne są w widoku wyświetlającym listę wycieczek. Nie jest to najwygodniejsze rozwiązanie dlatego proponuje stwórz nowy komponent wyświetlające szczegóły dotyczące wycieczki. Przejście do szczegółów potrawy odbywać się będzie z poziomu widoku listy potraw.

Po kliknij na zdjęcie wycieczki powinniśmy zostać przekierowani do widoku konkretnej wycieczki. Widok szczegółowy powinien w atrakcyjny sposób prezentować wszystkie szczegóły związane z informacją o wycieczce. Dodatkowo tylko z poziomu szczegółów wycieczki ma być możliwość oceniania potrawy. Przenosimy więc tę funkcjonalność z poziomu listy głownej do widoku szczegółowego. Duplikujemy również na tym poziomie możliwość rezerwacji i rezygnacji z wycieczki.

Z widoku pojedynczej wycieczki, gdzie powinniśmy zobaczyć:

- nazwę wycieczki
- lokalizacje
- lista zdjęć powiązanych z wycieczka (np. w postaci slajdera lub karuzeli)
- Opis wycieczki
- Cena £/USD/zł w zależności od ustaleń globalnego ustawienia
- ocena (ocena i liczba)
- link do widoku z lista wycieczek
- formularz opini/oceny z wycieczki + wyświetlenie już dodanych opinii

Formularz opini powinien zawierać:

- Użyty NICK
- nazwa wycieczki której dotyczy (wymagane)

- obszar tekstu opini (wymagany, recenzja powinna być dłuższa niż 50 znaków, ale krótsza niż 500 znaków)

- data zakupu/skorzystania z wycieczki (typ wejścia="data") (opcjonalnie)

Po przesłaniu formularza zweryfikuj wszystkie pola. Wszystkie pola poza datą zakupu są wymagane. Gdyby coś nie jest ok np. opinia jest za krótka lub wymagany atrybut jest pusty, dodaj nowy błąd do tablicy błędów i wyświetlaj je. Po każdym przesłaniu formularza wyczyść tablicę błędów. Jeśli formularz jest poprawny, dodaj opinię do tablicy opinii (tylko lokalnie, nie wysyłaj żadnych zapytań) i wyczyść formularz pola. Powinniśmy być w stanie dodać tyle opini, ile chcemy.

Zadanie 4. Lista zakupionych wycieczek (rozszerzenie) (2 pkt)

Wybrane wycieczki znajdują się w koszyku. Z jego widoku możliwy jest zakup wycieczki. Przy każdej wycieczce powinien być możliwy zakup wycieczki, który zmiana stan wycieczki na kupiona. Wycieczka znika z koszyka i zostaje dodana do widoku historia zakupów. Prezentuje się tam lista zakupionych wycieczek wraz z info, kiedy kupiona oraz danymi dotyczącymi wycieczki (cena, data startu i zakończenia, lokalizacja, ilości biletów). Każda wycieczka ma status – przed (oczekiwania na rozpoczęcie), w trakcie (aktywna), zakończona (archiwalna). System sam przypomina wyświetlając w sekcji menu – info (odpowiednia ikona) o zbliżającym się starcie wycieczki.

Widok pozwala na filtrowanie wycieczek po statusie.

Realizacja Backend (max 10 pkt)

Zadanie 5. W katalogu Studenci_Firebase znajdziesz szkielet aplikacji służącej do zarzadzania danymi studenta. Uzupełnij kod oraz utwórz konto na platformie Firebase tak aby aplikacja została zasilona danymi pochodzącymi z Firebase oraz możliwe było tworzenie, usuwanie i modyfikacje wpisów w bazie danych. Posłuż się bazą typu FireDatabase lub FireStore (2 pkt)

Wybierz którą wersje backendu zrealizujesz:

Łatwiejszą:

Zadanie 6. BiuroTutystyczne – Integracja z FireBase (4 pkt)

Wracamy do naszej główne aplikacji **Wycieczki - BiuroTurystyczne**. Na podstawie doświadczeń jakie uzyskałeś realizując zadanie 5 oraz materiałów znajdujących się na początku lab zintegruj swoja aplikację z platformą Firebase. Niech dane o wycieczkach pochodzą z Firebase. Pozwoli to nam na rzeczywistą persystencje danych. Wybierz dowolny typ bazy danych znajdujący się w Firebase i zaimplementuj usługę pozwalająca na odczyt, dodawanie, modyfikacje lub usuwanie nowej pozycji z bazy.

Lub

Trudniejszą:

Zadanie 7. BiuroTurystyczne – Własny serwer REST API (8 pkt)

W wersji trudniejszej wymagane jest samodzielne napisanie serwera RESTAPI z wykorzystaniem NodeJS/ExpressJS. Do przechowywania danych danych możesz użyć bazy w FireBase lub bazy MongoDB znajdującej się w chmurze pod adresem https://www.mongodb.com/atlas/database.

Materiały potrzebne do realizacji znajdziesz w moich materiałach wykładowych. Po zaimplementowaniu serwera przetestuj jego funkcjonalność używając narzędzi typu np. curl lub Postman ewentualnie zainstaluj w środowisku Visual Studio Code kolejny plugin – Thunder Client.

W aplikacji klienckiej zaimplementuj usługę do komunikacji z serwerem aplikacyjnym. Wykorzystaj do tego celu moduł HttpClient. Następnie wywołaj usług w tych komponentach, które pozwolą Ci na zasilania danymi całej aplikacji oraz obsłużą metody do usuwania, dodawania i modyfikacji danych w aplikacji. Zastosuj poprawną adresacje REST API.