Bazy Danych 2023/2022
Biuro podróży

Marek Żuwała

Norbert Żmija

Gracjan Filipek

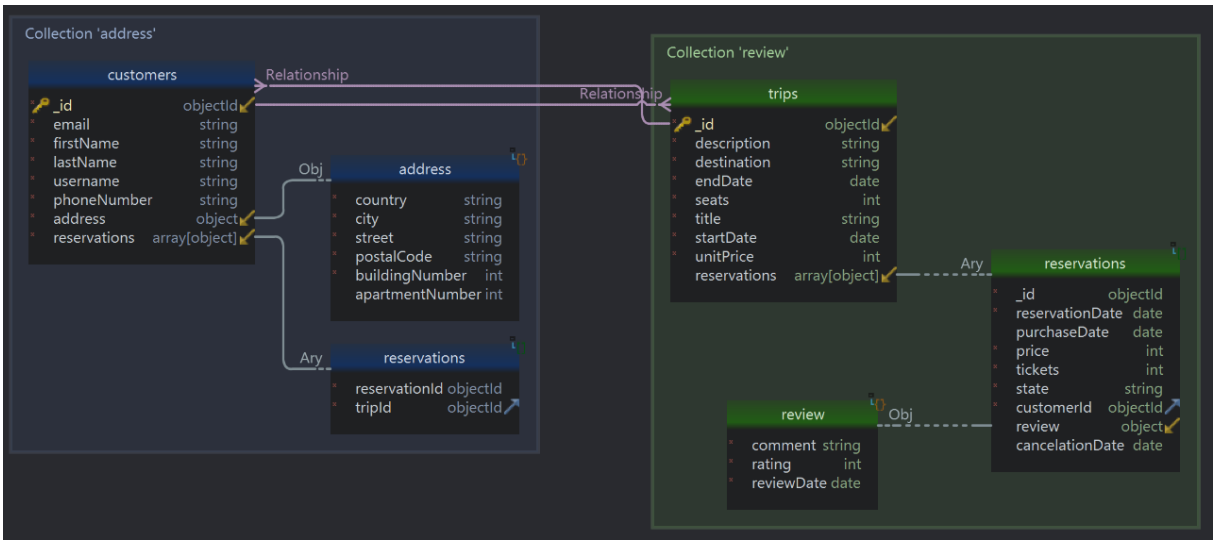
1 SPIS TREŚCI

2 OPIS PROJEKTU

Projekt polega na stworzeniu bazy danych dla biura podróży w oparciu obazę danych MongoDB, backend z użyciem Node.js i frontend w Angularze. W bazie danych będziemy trzymać informacje na temat wycieczek, użytkowników oraz ich zakupów i rezerwacji. Umożliwimy też ocenę i skomentowanie wycieczek.

3 BAZA DANYCH- MONGODB

Poniższy schemat został sporządzony przy użyciu programu DBSchema



Wyjaśnienie:

- Żółta strzałka po prawej stronie pola - zagnieżdżenie obiektu/ów
- Niebieska strzałka po prawej stronie pola - klucz obcy
- Czerwony krzyżyk po lewej stronie pola – pole jest wymagane (musi istnieć)

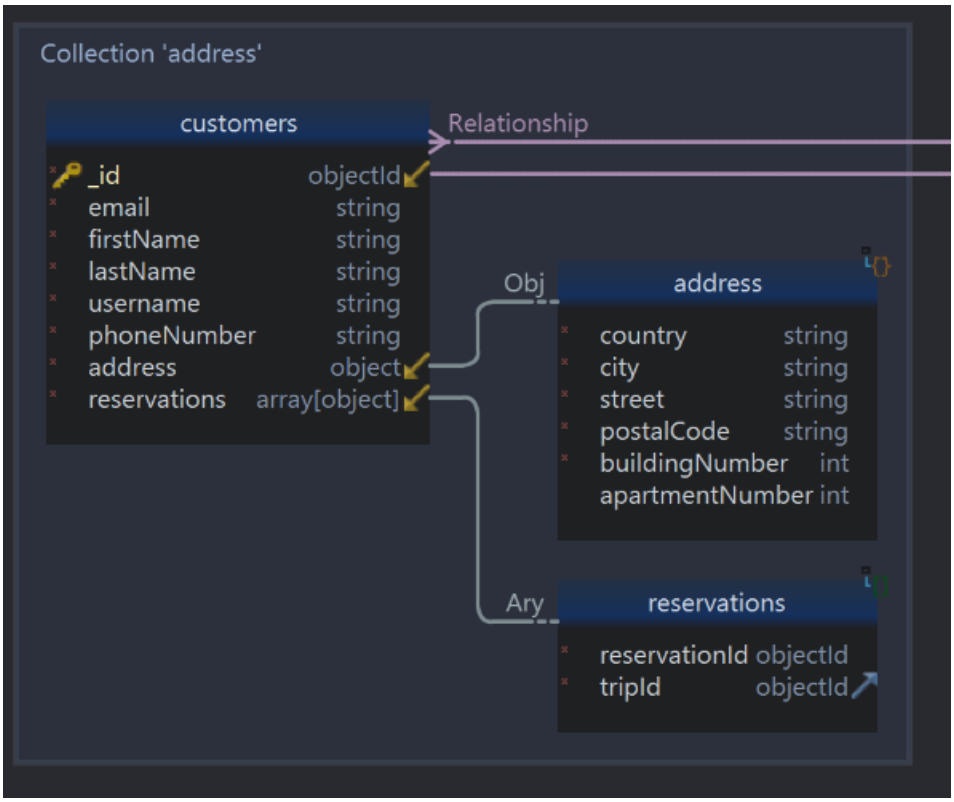
W naszej bazie przyjęliśmy zasadę nienullowania, tzn. żadne pola nie może przyjąć wartości null. Jeśli pole miało zawierać pustą tablicę elementów, to pole nie powinno istnieć. Jeśli pole byłoby pojedyncze, a nie chcemy / nie możemy umieścić tam żadnej wartości, to pole takie także nie powinno istnieć.

Wyjaśnimy teraz parę kwestii, dotyczących budowy naszego modelu danych i naszych wyborów projektowych.

- Zdecydowaliśmy się na zagnieżdżenie rezerwacji w dokumencie wycieczki, ponieważ chcemy mieć łatwy dostęp do rzeczonych rezerwacji z poziomu wycieczki. Będzie to przydatne np. podczas wyświetlania opinii na stronie internetowej pod daną wycieczką, gdyż opinie są zagnieżdżone właśnie w rezerwacjach. Oszczędzi to wielu zapytań do bazy danych, gdyż spodziewamy się, że oferta biura wycieczki (przede wszystkim strona internetowa) będzie przeglądana znacznie częściej, niż historia rezerwacji pojedynczego użytkownika. Oczywiście przeciwne podejście też jest możliwe, jednak wiąże się z innymi zaletami i wadami.
- Każda rezerwacja ma własne pole *price,* przeznaczone na cenę biletu, którą charakteryzowała się wycieczka w czasie zakupu. Umożliwia to biuru podróży zmianę ceny wycieczki w trakcie sprzedaży biletów. Uznaliśmy to za przydatną opcję, szczególnie w okresie wysokiej inflacji, kiedy ceny produktów / usług sprzedawanych z dużym wyprzedzeniem często mogą zmieniać ceny, aby dostosować się do obecnie panujących warunków.
- Każda rezerwacja może być oceniona przed użytkownika osobno nawet, jeśli dotyczą one tej samej wycieczki. Nie chcieliśmy zabraniać użytkownikowi możliwości wielokrotnej oceny wycieczki tylko dlatego, bo już kiedyś na niej był. Okoliczności i jej jakość mogły się z czasem zmienić, dlatego obiektem „nadrzędnym” jest rezerwacja, a nie bezpośrednio wycieczka.

- Dokument *customer* posiada tablicę obiektów typu *reservations* (uwaga, nie są to obiekty *reservations* z kolekcji *trips*), które zawierają id odpowiedniej wycieczki i rezerwacji. Pozwala to na szybsze dostanie się do odpowiedniej rezerwacji, gdyż nie trzeba przeszukiwać wszystkich wycieczek (mamy id tej właściwej), a id rezerwacji jednoznacznie identyfikuje rezerwację z kolekcji *trips* (więc rozwiązuje to problem niejednoznaczności, gdyby jeden klient zrobił dwie rezerwacje na tę samą wycieczkę)

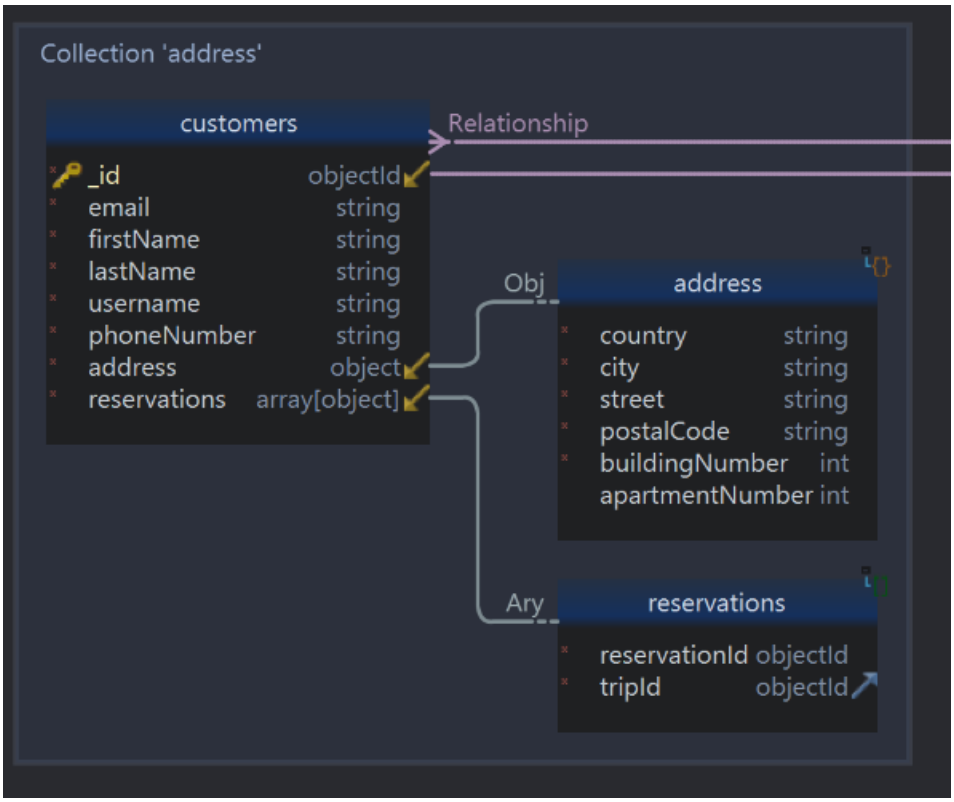
3.1 CUSTOMERS - KOLEKCJA KLIENTÓW



Dokument przedstawiający przykładowego klienta:

```
{
    "_id": ObjectId("6447a9ead297195ac0dd2408"),
    "email": "malwina@gmail.com",
    "firstName": "Malwina",
    "lastName": "Pimpuś",
    "username": "malwina13pl",
    "phoneNumber": "235-463-456",
    "address": {…},
    "reservations": [{…}, …]
}
```

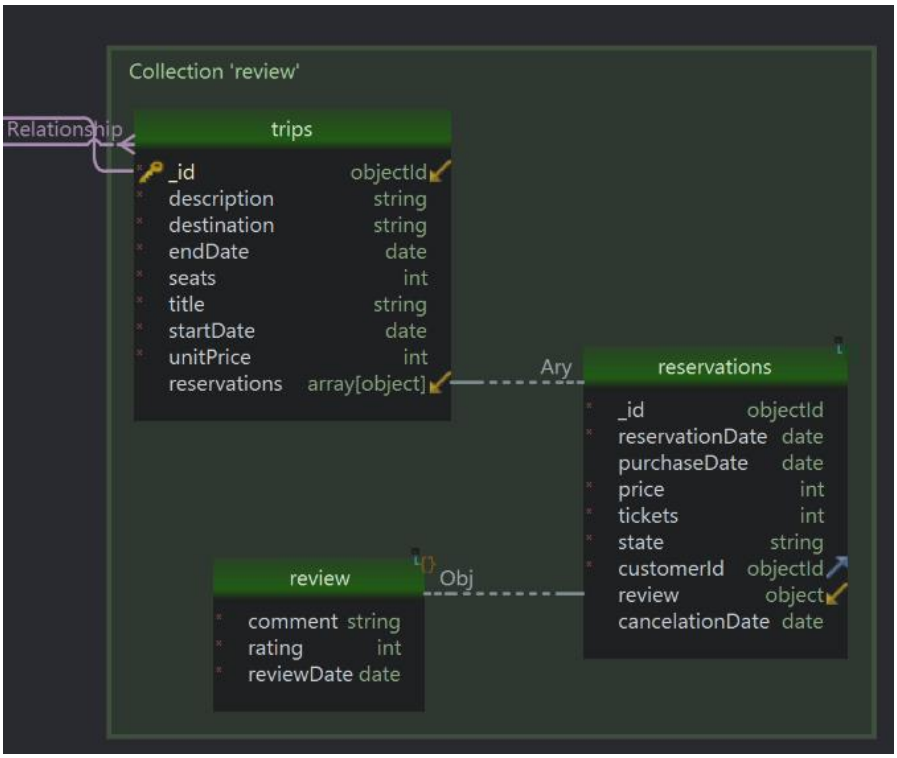
3.2 ADDRESS



Uwaga na pole *apartmentNumber* – jest nieobowiązkowe (co widać na schemacie powyżej).
Przykładowy obiekt:

```
"address": {
  "country": "Poland",
  "city": "Krakow",
  "street": "Kolorowa",
  "postalCode": "30-234",
  "buildingNumber": 999
}
```

3.3 TRIPS – KOLEKCJA WYCIECZEK



createRcreateReservationPole *seats* informuje o maksymalnej dostępnych miejsc na wycieczce.

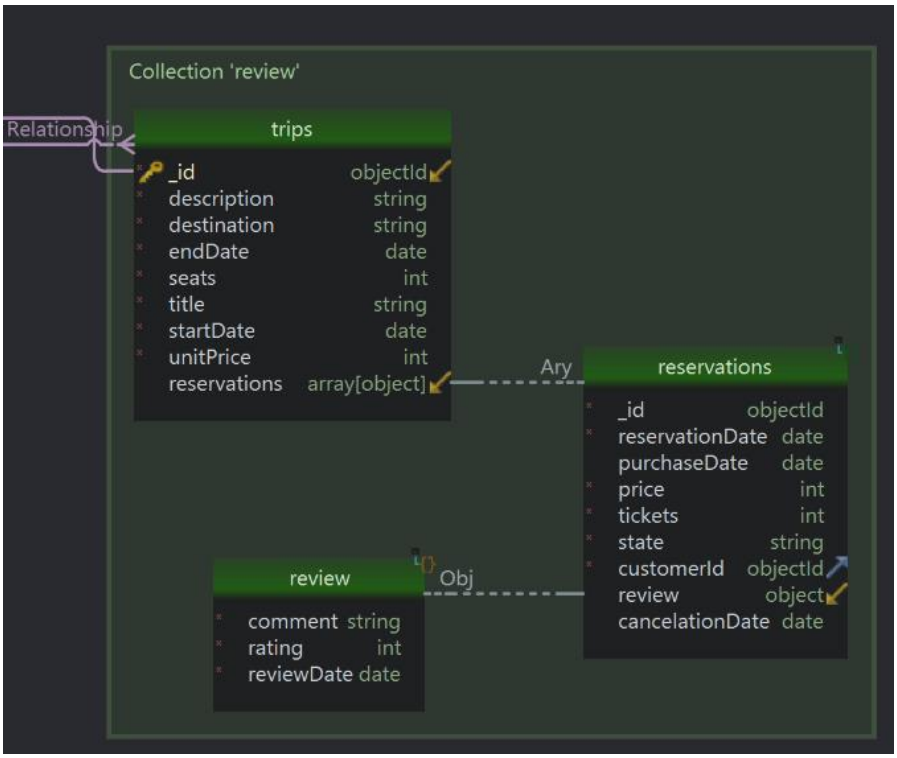
Data w polu *startDate* powinna być, naturalnie, wcześniejsza niż data w polu *endDate*.

Przewidzieliśmy, że cena wycieczki w polu *unitPrice* może się zmieniać w czasie, dlatego wprowadziliśmy osobne pole *price* w obiekcie typu *reservation,* w którym pamiętamy, po jakiej cenie zostały kupione bilety.

Przykładowy dokument:

```
{
        "_id": ObjectId("6447af70d297195ac0dd242c"),
        "description": "Słoneczny Cypr to wycieczka, która pozwoli Ci odpocząć na pięknych
plażach i zwiedzić malownicze miasta tej urokliwej wyspy. Podczas wycieczki zwiedzisz
między innymi stolicę Nikozję, wpisaną na listę UNESCO, a także zwiedzisz starożytną
twierdzę w Famagusta. Możesz również spędzić czas na relaksie na piaszczystych plażach albo
zażyć kąpieli w turkusowym morzu. Nasz wykwalifikowany przewodnik opowie Ci o historii i
kulturze tego wyjątkowego miejsca. Nie przegap okazji, aby poznać urokliwy Cypr z MarcoPolo
Travels!",
        "destination": "Cypr",
        "endDate": ISODate("2022-06-13T00:00:00.000Z"),
        "seats": 20,
        "title": "Słoneczny Cypr",
        "startDate": ISODate("2022-06-03T00:00:00.000Z"),
        "unitPrice": 1369,
        "reservations": [{…}, …]
    }
```

3.4 RESERVATIONS



Pole *purchaseDate* występuje tylko w wycieczkach, których stan to „Purchased”

Przewidzieliśmy, że cena wycieczki w polu *unitPrice* może się zmieniać w czasie, dlatego wprowadziliśmy osobne pole *price* w obiekcie typu *reservation,* w którym pamiętamy, po jakiej cenie zostały kupione bilety.

Rezerwacje mają trzy dozwolone wartości typu string w polu *state*:

- „New” – nowo złożona rezerwacja.
- „Purchased” – rezerwacja, za którą zapłacono.
- „Cancelled” – anulowana rezerwacja.

Użytkownik może zostawić ocenę w polu *review* wtedy i tylko wtedy, gdy wycieczka już się odbyła (tzn. minęła data zapisana w polu *endDate*) oraz jej stan to „Purchased”.

Jeśli klient dokona rezerwacji na wycieczkę i nie kupi jej, a wycieczka się odbędzie (minie data zapisana w *endDate*), to taka rezerwacja pozostaje w niezmienionej formie, tzn. wciąż ma status „New”. Dzięki temu możemy dokładnie wydedukować losy takiej rezerwacji.

Przykładowe obiekty:

3.4.1 Nowa rezerwacja:

```
{
    "_id": ObjectId("64482a4e8c343fb3405c251d"),
    "reservationDate": ISODate("2023-06-14T10:07:23.693Z"),
    "tickets": 1,
    "price": 1199,
    "state": "New",
    "customerId": ObjectId("6447a9ead297195ac0dd2408"),
}
```

3.4.2 Kupiona, lecz nieoceniona rezerwacja:

Musi zawierać datę dokonania opłaty.

Taka rezerwacja nie powinna mieć pola *review,* zgodnie z przyjętą zasadą nienullowalności.

Wycieczkę taka można ocenić, ale pod warunkiem, że już się odbyła.

Przykładowy obiekt:

```
{
    "_id": ObjectId("64287a4e8c343fb3345c251b"),
    "reservationDate": ISODate("2021-11-23T10:58:14.923Z"),
    "purchaseDate": ISODate("2022-03-24T12:10:13.123Z"),
    "price": 1399,
    "tickets": 1,
    "state": "Purchased",
    "customerId": ObjectId("6447a9ead297195ac0dd240c"),
}
```

3.4.3 Kupiona i oceniona rezerwacja:

Musi zawierać datę dokonania opłaty.

W polu *review* zawiera ocenę użytkownika, który złożył daną rezerwację.

Przykładowy obiekt:

```
{
       "_id": ObjectId("64287a4e8c343fb3345c251a"),
       "reservationDate": ISODate("2022-06-18T10:58:14.923Z"),
       "purchaseDate": ISODate("2022-06-20T11:00:14.923Z"),
       "price": 1399,
       "tickets": 2,
       "state": "Purchased",
       "customerId": ObjectId("6489f6b9bbb9fa29afe01fe4"),
       "review": {
            "comment": "Każdemu mogę polecić tą wycieczkę! Wszystko dobrze zorganizowane i
na wysokim poziomie. Nic tylko rezerwować.",
       "rating": 5,
        "reviewDate": ISODate("2023-08-04T11:00:14.923Z"),
    }
}
```

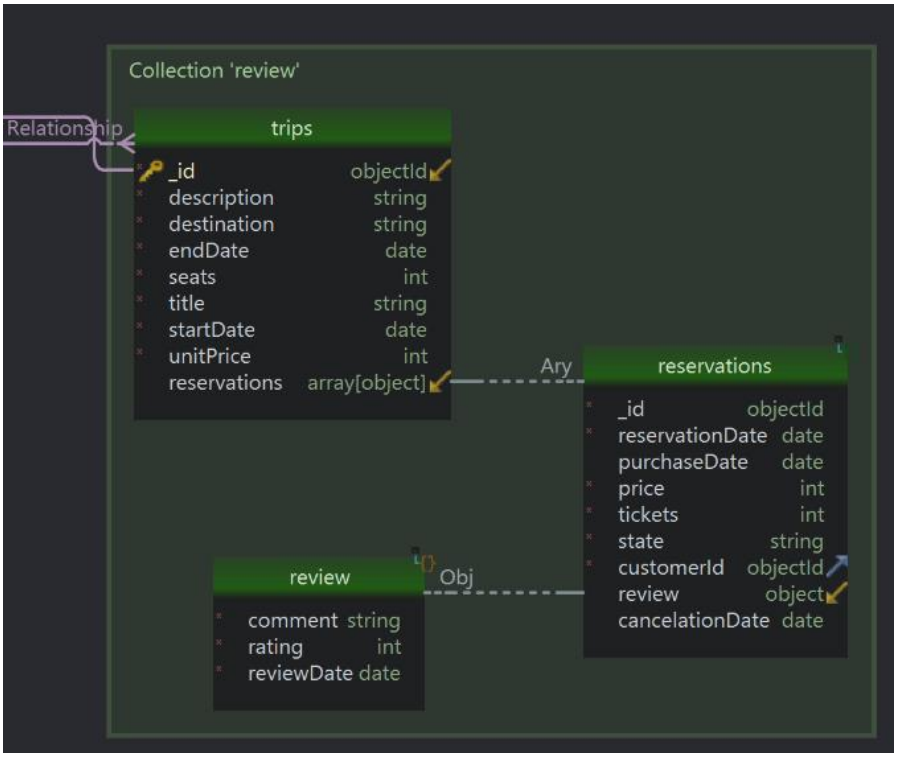
3.4.4 Anulowana rezerwacja:

Rezerwacja powinna zawierać pole *cancelationDate.* Rezerwację można anulować tylko, gdy jej stan to „New”. Rezerwacja, która została anulowana, nie powinna zawierać pola *purchaseDate.* Anulowanie wycieczki automatycznie zwraca zarezerwowane bilety do puli dostępnych biletów, bo te są wyliczane dynamicznie.

```
{
    "_id": ObjectId("64482a4e8e399eb38853342c"),
    "reservationDate": ISODate("2022-03-14T10:58:14.923Z"),
    "cancelationDate": ISODate("2022-03-15T07:03:24.421Z"),
    "price": 1899,
```

```
    "tickets": 3,
    "state": "Cancelled",
    "customerId": ObjectId("6489f6c6bbb9fa29afe01fe5"),
}
```

3.5 REVIEW



Wartości pola rating mogą przyjmować wartości całkowite od 1 do 5 obustronnie włącznie.

Komentarz jest obowiązkowy do każdej rezerwacji.

Przykładowy obiekt:

```
"review": {
    "comment": "Było luksusowo, tak jak oczekiwałam. Wreszcie na prawdę odpoczęłam od
pracy. Pogoda dopisała także na prawdę choćbym chciała, to nie mam na co narzekać. Jednym
słowem BAJKA!",
    "rating": 5,
    "reviewDate": ISODate("2022-07-30T15:22:11.923Z"),
}
```

4 BACKEND – NODE JS

Backend naszego projektu postanowiliśmy napisać w Node.js z kilku powodów:

Asynchroniczność

Node.js korzysta z asynchronicznej natury JavaScriptu, co pozwala na obsługę wielu żądań jednocześnie bez blokowania wątku głównego. To jest szczególnie korzystne dla naszej aplikacji, która może otrzymywać duże ilości żądań dotyczących wycieczek, rezerwacji i ocen.

Jednolity Język

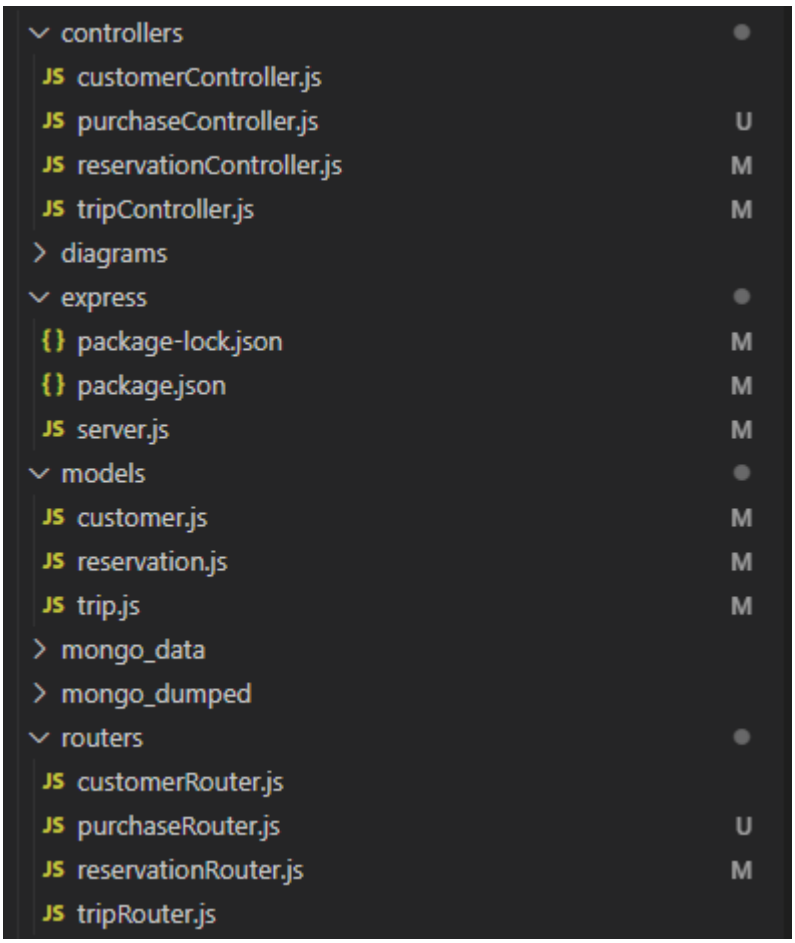
Node.js operuje w języku JavaScript, Angular w języku TypeScript, który jest superzbiorem JavaScripta. Umożliwia to nam pisaniu kodu backendu i frontendu w (prawie) wspólnym języku, dzięki czemu jest większą spójności kodu i ułatwione zarządzanie projektem.

Wszechstronność

Node.js ma wiele bibliotek, które ułatwiają interakcję z MongoDB. Jedną z nich jest Mongoose, która jest bogata w funkcje i ułatwia pracę z danymi. Mongoose oferuje funkcje takie jak walidacja danych, tworzenie i wykonywanie zapytań, definiowanie schematów z typowaniem, middleware, i wiele innych. Dzięki Mongoose, możemy łatwo definiować schematy dla naszych danych, co pozwala na lepszą kontrolę nad projektem.

4.1 STRUKTURA PLIKÓW

Struktura Node.js prezentuje się następująco:



Po krótce opiszemy każdy z folderów, a następnie przejdziemy do bardziej szczegółowych wyjaśnień:

1. **Folder "controllers"**: Ten folder zawiera pliki kontrolerów, które obsługują logikę w naszej aplikacji. Każdy kontroler odpowiada za inny aspekt aplikacji:
    - **customerController.js**: Zarządza operacjami związanymi z klientami, takimi jak tworzenie nowych klientów, aktualizowanie danych klientów, usuwanie ich itd.
    - **purchaseController.js**: Obsługuje logikę związaną z zakupami, w tym pobieranie dokonanych zakupów i wykupywanie poczynionych rezerwacji.
    - **reservationController.js**: Odpowiada za rezerwacje, w tym ich tworzenie, aktualizację i anulowanie.
    - **tripController.js**: Zarządza wszystkimi operacjami związanymi z wycieczkami, takimi jak tworzenie nowych wycieczek, aktualizowanie informacji o wycieczkach, usuwanie ich, itd.
2. **Folder "express"**: Ten folder zawiera pliki związane z konfiguracją i uruchomieniem serwera:
    - **package-lock.json** i **package.json**: Zawierają informacje o zależnościach projektu i ich wersjach.
    - **server.js**: Jest to główny plik serwera, który konfiguruje i uruchamia nasz serwer Express.js.
3. **Folder "models"**: W tym folderze znajdują się modele danych, które definiują strukturę danych w naszej bazie MongoDB:
    - **customer.js**: Definiuje strukturę danych klienta.
    - **reservation.js**: Definiuje strukturę danych rezerwacji.
    - **trip.js**: Definiuje strukturę danych wycieczki.
4. **Folder "routers"**: Tutaj znajdują się pliki, które definiują trasy dla naszej aplikacji. Każdy router odpowiada za inny aspekt:
    - **customerRouter.js**: Trasy związane z klientami.
    - **purchaseRouter.js**: Trasy związane z zakupami.
    - **reservationRouter.js**: Trasy związane z rezerwacjami.
    - **tripRouter.js**: Trasy związane z wycieczkami.

4.2 CONTROLLERS

customerController.js

Plik **customerController.js** definiuje różne metody, które służą do obsługi zapytań HTTP dotyczących klientów. Każda metoda kontrolera odpowiada na specyficzne zapytanie HTTP i wykonuje operacje na danych klientów w bazie danych.

1. **getAllCustomers**: Ta metoda jest używana do obsługi zapytań GET do pobrania wszystkich klientów. Wykorzystuje ona metodę **find()** z modelu **Customer** do pobrania wszystkich

klientów z bazy danych. W przypadku błędu zwraca status 500 (błąd serwera) wraz z wiadomością o błędzie.

2. **getCustomerById**: Służy do obsługi zapytań GET dla specyficznego klienta na podstawie jego **id**. Jeżeli klient o podanym **id** istnieje, metoda zwraca klienta. W przeciwnym razie zwraca status 404 (nie znaleziono) wraz z odpowiednią wiadomością. Ta metoda jest użyteczna np. gdy chcemy wyświetlić danemu użytkownikowi jego dane.

3. **createCustomer**: Obsługuje zapytania POST do tworzenia nowego klienta. Tworzy nową instancję modelu **Customer** z danymi przekazanymi w ciele zapytania (**req.body**). Po utworzeniu nowego klienta, metoda zapisuje klienta w bazie danych i zwraca utworzonego klienta w odpowiedzi. W przypadku błędu zwraca status 500 wraz z wiadomością o błędzie.

4. **updateCustomer**: Obsługuje zapytania PUT do aktualizacji istniejącego klienta na podstawie jego **id**. Używa ona metody **findByIdAndUpdate()** z modelu **Customer** do wyszukania i aktualizacji klienta. Opcje **new: true** i **runValidators: true** są używane, aby zwrócić zaktualizowany dokument i uruchomić walidatory schematu podczas aktualizacji. Jeżeli klient o podanym **id** nie istnieje, metoda zwraca status 404. W przeciwnym razie zwraca zaktualizowanego klienta.

5. **deleteCustomer**: Obsługuje zapytania DELETE do usuwania istniejącego klienta na podstawie jego **id**. Używa ona metody **findByIdAndDelete()** z modelu **Customer** do wyszukania i usunięcia klienta. Jeżeli klient o podanym **id** nie istnieje, metoda zwraca status 404. Jeśli klient poczynił jakieś rezerwacje, zwracany jest status 500 z odpowiednią informacją. W przeciwnym razie zwraca wiadomość o pomyślnym usunięciu klienta.

Najciekawsze funkcje

W zasadzie brak tu ciekawych funkcji, gdyż wszystkie odpowiadają za proste operacje CRUD. Wyróżnia się jedynie operacja usuwania.

```
const Customer = require("../models/customer");
  deleteCustomer: async (req, res) => {

    try {
      const customerToDelete = await
Customer.findById(req.params.id).select('reservations').populate;

      let canBeDeleted = true

      const { reservations } = await
Customer.findById(req.params.id).select('reservations').populate('reservations.tripId',
'reservations').lean()
      const result = reservations.map(r => {
        const {reservationId, tripId} = r
        const filtered = tripId.reservations.find(r => r._id.equals(reservationId) &&
r.state != 'Cancelled')
        if(filtered !== undefined) {
          canBeDeleted = true
        }
        return filtered
      })

      if(!canBeDeleted) {
        res.status(500).json({ message: "Cannot delete customer with reservations" });
        return
      }
```

```
      const deletedCustomer = await Customer.findByIdAndDelete(req.params.id);
      if (!deletedCustomer) {
        res.status(404).json({ message: "Customer not found" });
      } else {
        res.status(200).json({ message: "Customer deleted successfully" });
      }
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  },
};
```

purchaseController.js

Plik **purchaseController.js** definiuje dwie metody kontrolera do obsługi zakupów: **getCustomerPurchases** i **buyReservation**.

1. **getCustomerPurchases**: Metoda ta służy do obsługi zapytań GET do pobrania wszystkich zakupów (rezerwacji ze statusem „Purchased") dokonanych przez konkretnego klienta na podstawie jego **id**. Najpierw metoda pobiera klienta oraz jego rezerwacje z bazy danych. Następnie filtruje rezerwacje i zwraca te, których status to „Purchased" w odpowiedzi. W przypadku błędu, metoda zwraca status 500 (błąd serwera) wraz z wiadomością o błędzie.

2. **buyReservation**: Metoda ta obsługuje zapytania PUT do dokonania zakupu rezerwacji. Metoda znajduje rezerwację na podstawie **tripId** i **reservationId** przekazanych w ciele zapytania (**req.body**), a następnie aktualizuje stan rezerwacji na 'Purchased' i ustawia datę zakupu na aktualną datę. Jeżeli rezerwacja o podanym **id** nie istnieje, metoda zwraca status 404. W przeciwnym razie zwraca zaktualizowaną rezerwację.

Alternatywnie:

1. **getCustomerPurchases:** Metoda ta pozwala pobrać wszystkie zakupione rezerwacje danego klienta. Rezerwacje są zagnieżdżone w wycieczkach, dlatego niezbędne byłoby stosowanie lookupa. Okazuje się jednak, że **mongoose** ma przydatną metodę **populate**, która spełnia tę funkcję. Niestety, spowoduje to wiele zapytań do bazy, ale możemy sobie na pozwolić, gdyż spodziewamy się, że klienta rzadko będzie chciał wyświetlać swoje zakupione rezerwacje w porównaniu, a w zamian inne operacje są bardziej, dotyczące wycieczek i rezerwacji, są bardziej wydajne. Co ciekawe, wersja napisana własnoręcznie działa trochę szybciej, niż metoda populate (około 20% szybciej, tu i tu prędkość rzędu 20ms).

2. **buyReservation:** Ta metoda służy do zmiany statusu danej rezerwacji na „Purchased". Korzysta z metody findOneAndUpdate z **mongoose**, która umożliwia nam wyszukiwanie i aktualizowanie dokumentu w jednym, atomowym zapytaniu.

Używamy również metody lean() w getCustomerPurchases do konwersji Mongoose Document na zwykły obiekt JavaScript, co przyspiesza proces, gdyż nie potrzebujemy dodatkowych funkcji dostarczanych przez Mongoose Document.

Najciekawsze funkcje

Funkcja **getCustomerPurchases** musi odpytać bazę o rezerwacje danego klienta, gdyż te są przechowywane w wycieczkach. Używamy do tego funkcji **populate** z biblioteki **mongoose**.

```
const { filter } = require("rxjs");
const Customer = require("../models/customer")
const Trip = require("../models/trip")

const purchaseController = {
  getCustomerPurchases: async (req, res) => {
    try {
      const { reservations } = await
Customer.findById(req.params.id).select('reservations').populate('reservations.tripId',
'title destination startDate endDate reservations').lean()

      const result = reservations.map(r => {
        const {reservationId, tripId} = r
        const filtered = tripId.reservations.find(r => r._id.equals(reservationId) &&
r.state == 'Purchased')
        if(filtered !== undefined) {
          const { review, ...rest } = filtered
          return {
            title: tripId.title,
            destination: tripId.destination,
            startDate: tripId.startDate,
            endDate: tripId.endDate,
            tripId: tripId._id,
            review: review === undefined ? false : true,
            ...rest
          }
        }
        return filtered
      }).filter(r => r != null)
      if (!result) {
        res.status(404).json({ message: "Reservation not found" });
      } else {
        res.status(200).json(result);
      }
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  },
```

reservationController.js

Plik reservationController.js jest odpowiedzialny za zarządzanie rezerwacjami. Zawiera pięć metod:

1. **getAllReservations:** Ta metoda służy do pobierania wszystkich rezerwacji z bazy danych. Używa metody find() na modelu Reservation,. Jeżeli operacja jest pomyślna, zwraca status HTTP 200 wraz z tablicą rezerwacji w formacie JSON. W przypadku błędu, zwraca status HTTP 500 wraz z komunikatem błędu.

2. **getCustomerReservations:** Metoda do pobierania rezerwacji dla konkretnego klienta. Pobiera ona ID klienta z parametrów żądania, a następnie używa metody findById() na modelu Customer do pobrania konkretnego klienta. Następnie za pomocą metody populate(), pobierane są wszystkie szczegóły dotyczące rezerwowanych wycieczek, takie jak tytuł, cel, data rozpoczęcia i zakończenia. W przypadku powodzenia, zwraca status HTTP 200 wraz z informacjami o rezerwacjach. W przypadku błędu, zwraca status HTTP 500.

3. **createReservation:** Metoda do tworzenia nowej rezerwacji. Najpierw za pomocą metody findById() na modelu Trip szuka konkretnego wyjazdu. Następnie tworzy nową rezerwację i zapisuje ją do znalezionego obiektu trip. Potem za pomocą metody findById() na modelu Customer znajduje konkretnego klienta i zapisuje do niego informacje o nowej rezerwacji. W przypadku powodzenia, zwraca status HTTP 201 oraz nowo utworzoną rezerwację. W przypadku błędu, zwraca status HTTP 500.

4. **createReview:** Ta metoda pozwala klientom na dodanie recenzji do rezerwacji. Wykorzystuje ona metody findOneAndUpdate() na modelu Trip do znalezienia konkretnego wyjazdu i rezerwacji, a następnie aktualizuje pole review rezerwacji. W przypadku powodzenia, zwraca status HTTP 200 oraz zaktualizowaną rezerwację. W przypadku błędu, zwraca status HTTP 500.

5. **resignReservation:** Ta metoda pozwala klientom na rezygnację z rezerwacji. Podobnie jak createReview, używa metody findOneAndUpdate() na modelu Trip do znalezienia konkretnej rezerwacji, a następnie zmienia jej stan na 'Cancelled'. Z rezerwacji można zrezygnować tylko, gdy jej stan to „New”. Jeśli tak nie jest, metoda zwraca status 500 z odpowiednią informacją. W przypadku powodzenia, zwraca status HTTP 200 oraz zaktualizowaną rezerwację. W przypadku błędu, zwraca status HTTP 500.

Alternatywnie:

Metoda **resignReservation** mogłaby usuwać dokument rezerwacji, zamiast tylko zmieniać jej stan na 'Cancelled'. Niemniej, utrzymanie dokumentów rezerwacji, nawet po ich anulowaniu, pozwala na lepsze śledzenie historii klienta.

Najciekawsze funkcje

```
const Trip = require("../models/trip");
const Customer = require("../models/customer")
const Reservation = require("../models/reservation")

const reservationController = {
  createReservation: async (req, res) => {
    let newReservation = null
    try {
      const trip = await Trip.findById(req.body.tripId).select('reservations')
      newReservation = trip.reservations.create({
        reservationDate: new Date(),
        price: req.body.price,
        tickets: req.body.tickets,
        state: 'New',
        customerId: req.body.customerId
      })
      trip.reservations.push(newReservation)
```

```
      await trip.save();

      const customer = await Customer.findById(req.body.customerId).select('reservations')
      customer.reservations.push({
        tripId: req.body.tripId,
        reservationId: newReservation._id
      })
      await customer.save();
      res.status(201).json(newReservation);
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  },

  createReview: async (req, res) => {
    try {
      const updatedReservation = await Trip.findOneAndUpdate(
        { "_id": req.body.tripId, "reservations._id": req.body.reservationId },
        { "$set": {
          "reservations.$.review": {
            comment: req.body.comment,
            rating: req.body.rating,
            reviewDate: new Date()
          }
        } }
      );
      if (!updatedReservation) {
        res.status(404).json({ message: "Reservation not found" });
      } else {
        res.status(200).json(updatedReservation);
      }
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  },
};
```

tripController.js

Plik tripController.js zawiera pięć metod:

1. **getAllTrips**: Ta metoda jest używana do pobierania wszystkich zarejestrowanych wycieczek. Zaczyna od wywołania funkcji **Trip.find()**, która zwraca wszystkie wycieczki z bazy danych. Następnie przekształca każdy obiekt wycieczki, obliczając dostępne miejsca (odejmując liczbę biletów zarezerwowanych w stanie różnym od "Cancelled" od całkowitej liczby miejsc) oraz średnią ocenę (dodając wszystkie oceny i dzieląc przez ich liczbę). Rezultat jest przekazywany jako odpowiedź w formacie JSON.
2. **getTripById**: Ta metoda jest używana do pobierania pojedynczej wycieczki na podstawie przekazanego ID. Wywołuje funkcję **Trip.findById()**, aby pobrać wycieczkę o określonym ID. Oblicza dostępne miejsca i średnią ocenę w taki sam sposób jak **getAllTrips**. Ponadto, jeżeli wycieczka posiada recenzje, są one dodawane do obiektu odpowiedzi z nazwą autora.

3. **createTrip**: Ta metoda jest używana do tworzenia nowej wycieczki. Dane dla nowej wycieczki są pobierane z żądania od klienta, po czym jest tworzony nowy obiekt **Trip** z tymi danymi. Następnie nowy obiekt jest zapisywany w bazie danych. Po udanym zapisaniu, nowy obiekt wycieczki jest zwracany jako odpowiedź.

4. **updateTrip**: Ta metoda jest używana do aktualizacji istniejącej wycieczki. Na podstawie ID wycieczki przekazanego w parametrach żądania, metoda wyszukuje odpowiednią wycieczkę w bazie danych. Następnie aktualizuje jej dane za pomocą danych przekazanych w żądaniu. Po udanej aktualizacji, zaktualizowany obiekt wycieczki jest zwracany jako odpowiedź.

5. **deleteTrip**: Ta metoda służy do usuwania wycieczki, o ile nie ma na nią nieanulowanych rezerwacji. Na podstawie ID wycieczki przekazanego w parametrach żądania, metoda wyszukuje odpowiednią wycieczkę w bazie danych. Jeśli taka wycieczka istnieje, jest ona usuwana. Po udanym usunięciu, zwracany jest komunikat informujący o pomyślnym usunięciu.

Alternatywnie:

1. Dobrym pomysłem mogłoby być przechowywanie nazwy autora, **username**, w rezerwacji, co zaoszczędziłoby potrzeby wyszukania danego klienta tylko po to, aby uzyskać jego nazwę. Oczywiście trzeba by wtedy zadbać o to, żeby zmiana **username** klienta rozpropagowała się po jego recenzjach.
2. **deleteTrip** usuwa wycieczkę i wszystkie powiązane z nią rezerwacje. Alternatywnym podejściem mogłoby być zastosowanie wzorca "soft delete", w którym zamiast całkowitego usuwania wycieczki, jej status jest zmieniany na "usunięta" – podobnie jak robimy w rezerwacjach.

Najciekawsze funkcje

Metoda **getAllTrips** oblicza (nota bene, na poziomie backendu) średnią ocenę wycieczki i dostępne miejsca.

```
const Trip = require("../models/trip");

const tripController = {
  getAllTrips: async (req, res) => {
    try {
      const trips = await Trip.find().select('seats reservations title destination
startDate endDate unitPrice').lean();
      const result = trips.map(trip => {
        const {seats, reservations, ...rest} = trip
        let availableSeats = seats
        let ratingSum = 0.0
        let rates = 0
        if(reservations !== undefined)
        {
          for(const reservation of reservations) {
            if(reservation.state != "Cancelled")
              availableSeats -= reservation.tickets
            if(reservation.review) {
              ratingSum += reservation.review.rating
              rates++
            }
```

```
          }
        }
        return {
          availableSeats: availableSeats,
          avgRating: rates == 0 ? null : Math.round(ratingSum / rates),
          ...rest
        }
      })
      res.status(200).json(result);
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  },
};
```

Czego się nauczyliśmy?

1. **Podział odpowiedzialności**: Kontrolery powinny być odpowiedzialne za obsługę logiki powiązanej z żądaniami HTTP i odpowiedziami. Wszystkie zadania związane z logiką powinny być wykonywane w odpowiednich usługach lub modelach. To zapewnia separację zadań i ułatwia późniejsze odnalezienie się w projekcie.
2. **Czystość i zrozumiałość kodu**: Zrozumieliśmy jak ważne jest by kod był jak najprostszy i najbardziej zrozumiały. Długie metody utrudniają zrozumienie i utrzymanie kodu, dlatego dobrą praktyką jest dzielenie ich na mniejsze funkcje pomocnicze.
3. **Obsługa błędów**: Nauczyliśmy się, na własnych doświadczeniach, aby pamiętać o błedach, zarówno tych oczekiwanych, jak i nieoczekiwanych. Ważne jest, aby kontrolery zwracały odpowiednie kody statusów HTTP i przydatne komunikaty błędów.
4. **Optymalizacja wydajności**: Powinniśmy zwracać tylko te dane, które są rzeczywiście potrzebne klientowi. W wielu miejscach stosujemy metodę **.select(…),** która pozwala wybrać, które dokładnie pola będą przesłane do klienta. Dzięki temu oszczędzamy przepustowość i zwiększamy bezpieczeństwo systemu, gdyż nie wysyłamy wrażliwych danych.
5. **Użycie odpowiednich metod HTTP**: Dla poprawnej semantyki API, kontrolery powinny używać odpowiednich metod HTTP do reprezentowania operacji CRUD: GET do pobierania danych, POST do tworzenia nowych zasobów, PUT (lub PATCH) do aktualizacji zasobów i DELETE do usuwania zasobów.
6. **Testowanie**: Kontrolery powinny być jakkolwiek przetestowane. Nie pisaliśmy automatycznych testów, jednak zawsze staraliśmy się sprawdzać, czy kod zadziała poprawnie dla (mamy nadzieję) wszystkich scenariuszy i przypadków brzegowych.

4.3 MODELS

customer.js

```
const mongoose = require("mongoose");

const addressSchema = new mongoose.Schema({
  country: {
    type: String,
    required: true,
  },
  city: {
```

```
    type: String,
    required: true,
  },
  street: {
    type: String,
    required: true,
  },
  postalCode: {
    type: String,
    required: true,
  },
  buildingNumber: {
    type: Number,
    required: true,
    validate: {
      validator: Number.isInteger,
      message: "buildingNumber must be an integer.",
    },
    min: 1,
  },
  apartmentNumber: {
    type: Number,
    required: false,
    validate: {
      validator: Number.isInteger,
      message: "apartmentNumber must be an integer.",
    },
    min: 1,
  },
});

const customerSchema = new mongoose.Schema({
  firstName: {
    type: String,
    required: true,
  },
  lastName: {
    type: String,
    required: true,
  },
  userName: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
  },
  phoneNumber: {
    type: String,
    required: true,
  },
  address: {
    type: addressSchema,
    required: true,
  },
  reservations : {
```

```
    type: [{
      reservationId: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Reservation',
        required: true,
      },
      tripId: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Trip',
        required: true,
      }
    }],
    required: false
  }
});

const Customer = mongoose.model("Customer", customerSchema);

module.exports = Customer;
```

Plik **customer.js** zawiera definicję modelu **Customer**, który jest używany do reprezentowania klientów w naszym systemie. Ten model został utworzony, oczywiście, przy użyciu biblioteki **mongoose**. Poniżej znajduje się szczegółowy opis poszczególnych części tego modelu:

1. **addressSchema**: Jest to schemat adresu, który zawiera informacje takie jak kraj, miasto, ulica, kod pocztowy, numer budynku i numer mieszkania. Wszystkie te pola są wymagane, z wyjątkiem **apartmentNumber**, które jest opcjonalne. **buildingNumber** i **apartmentNumber** są dodatkowo walidowane, aby upewnić się, że są liczbami całkowitymi.
2. **customerSchema**: Jest to główny schemat klienta, który zawiera informacje takie jak **firstName**, **lastName**, **userName**, **email**, **phoneNumber**, oraz **address** (które jest instancją **addressSchema**).
3. **reservations**: Jest to tablica obiektów, które zawierają informacje o rezerwacjach dokonanych przez klienta. Każda rezerwacja ma **reservationId** oraz **tripId**, które są odwołaniami do odpowiednich modeli 'Reservation' i 'Trip'. Ten schemat umożliwia utrzymanie relacji między różnymi dokumentami w naszej bazie danych.
4. Na końcu, **mongoose.model("Customer", customerSchema)** tworzy model **Customer** z naszego schematu, a następnie eksportuje go do wykorzystania w innych częściach naszej aplikacji.

reservation.js

```
const mongoose = require("mongoose");

const reservationSchema = new mongoose.Schema({
  reservationDate: {
    type: Date,
    required: true,
  },
  purchaseDate: {
    type: Date,
    required: false,
  },
```

```
  tickets: {
    type: Number,
    required: true,
    validate: {
      validator: Number.isInteger,
      message: "tickets must be an integer.",
    },
    min: 1,
  },
  price: {
    type: Number,
    required: true,
    min: 1
  },
  state: {
    type: String,
    enum: ["New", "Purchased", "Cancelled"],
    required: true,
  },
  customerId: {
    type: mongoose.Types.ObjectId,
    ref: 'Customer',
    required: true,
  },
  review: {
    type: {
      comment: {
        type: String,
        required: true,
      },
      rating: {
        type: Number,
        required: true,
        validate: {
          validator: Number.isInteger,
          message: "rating must be an integer.",
        },
        min: 1,
        max: 5
      },
      reviewDate: {
        type: Date,
        required: true,
      },
    },
    required: false,
  }
});

module.exports = reservationSchema;
```

Plik **reservation.js** definiuje schemat **reservationSchema**, który jest używany do reprezentowania rezerwacji w systemie

1. **reservationDate**: Jest to data, kiedy rezerwacja została dokonana.
2. **purchaseDate**: Jest to data, kiedy rezerwacja została opłacona. Jest to pole opcjonalne.
3. **tickets**: Jest to ilość biletów zarezerwowanych. Pole to jest wymagane i musi być liczbą całkowitą większą lub równą 1.
4. **price**: Jest to cena za jeden bilet. Musi być liczbą większą lub równą 1.
5. **state**: Reprezentuje stan rezerwacji i może przyjmować jedną z wartości: "New", "Purchased", "Cancelled". Jest to pole wymagane.

6. **customerId**: Jest to identyfikator klienta, który dokonał rezerwacji. Pole to odnosi się do modelu 'Customer'.
7. **review**: Jest to pole, które przechowuje recenzję dotyczącą rezerwacji. Recenzja składa się z komentarza, oceny (liczby całkowitej od 1 do 5 obustronnie włącznie) i daty. Recenzja nie jest wymagana przy tworzeniu rezerwacji.

Alternatywnie

Alternatywnym podejściem mogłoby być przechowywanie recenzji jako osobnego dokumentu w bazie danych z odwołaniem do rezerwacji, zamiast przechowywania ich bezpośrednio w dokumencie rezerwacji, jednak byłby to słaby pomysł, gdyż recenzja jest, w naszym modelu, nierozłącznie związana z rezerwacją. Moglibyśmy natomiast zmienić koncepcję modelu i stwierdzić, że każda wycieczka może być oceniona tylko raz przez każdego klienta, jednak obecne podejście wydaje się bardziej logiczne.

trip.js

```
const reservationSchema = require('./reservation')
const mongoose = require("mongoose");

const tripSchema = new mongoose.Schema({
  description: {
    type: String,
    required: true,
  },
  destination: {
    type: String,
    required: true,
  },
  startDate: {
    type: Date,
    required: true,
  },
  endDate: {
    type: Date,
    required: true,
  },
  seats: {
    type: Number,
    required: true,
    validate: {
      validator: Number.isInteger,
      message: "seats must be an integer.",
    },
    min: 0,
  },
  title: {
    type: String,
    required: true,
  },
  unitPrice: {
    type: Number,
    required: true,
    min: 0,
  },
  reservations: {
    type: [reservationSchema],
    required: false,
  }
```

```
});

tripSchema.pre('save', function(next) {
  const reservedSeats = this.reservations.filter(x => x.state != 'Cancelled').map(x =>
x.tickets).reduce((a, b) => a + b, 0)
  if(this.seats < reservedSeats){
    next(new Error('Brak wolnych miejsc'))
  }
  else
  next()
})

const Trip = mongoose.model("Trip", tripSchema);

module.exports = Trip;
```

Plik **reservation.js** definiuje schemat **reservationSchema**, który jest używany do reprezentowania rezerwacji w systemie.

1. **reservationDate**: Jest to data, kiedy rezerwacja została dokonana.
2. **purchaseDate**: Jest to data, kiedy rezerwacja została opłacona. Jest to pole opcjonalne.
3. **tickets**: Jest to ilość biletów zarezerwowanych. Pole to jest wymagane i musi być liczbą całkowitą większą lub równą 1.
4. **price**: Jest to cena za pojedynczy bilet w momencie poczynienia rezerwacji. Daje to możliwość zmiany cen istniejących wycieczek i respektuje już dokonane rezerwacje.
5. **state**: Reprezentuje stan rezerwacji i może przyjmować jedną z wartości: "New", "Purchased", "Cancelled". Jest to pole wymagane.
6. **customerId**: Jest to identyfikator klienta, który dokonał rezerwacji. Jest to pole wymagane i odnosi się do modelu 'Customer'.
7. **review:** Jest to pole, które przechowuje recenzję dotyczącą rezerwacji. Recenzja składa się z komentarza, oceny (liczby całkowitej od 1) i daty przeglądu. Recenzja nie jest wymagana przy tworzeniu rezerwacji.

Warto zwrócić uwagę na ostatni fragment kodu, tzn. „**tripSchema.pre(…)**”. Jest to middleware, dostępny w bibliotece **mongoose**. Ten konkretny fragment pozwala upewnić się, że na pewno nie dodamy takiej rezerwacji, która spowodowałaby, że sumaryczna liczba zarezerwowanych miejsc przekroczyłaby liczbę dostępnych miejsc. Wykorzystanie middleware’u pozwala upewnić się, że każdy kod (dodający, modyfikujący itp.), który zapisuje wycieczki do bazy, będzie przestrzegać tego warunku.

Czego się nauczyliśmy?

1. **Jasna struktura**: Modele są kluczowe do zrozumienia struktury naszych danych. Dzięki definiowaniu modeli, mamy jasny obraz tego, jakie dane są przechowywane w naszej aplikacji, jak są one powiązane i jakie są wymagania dla każdego pola.

2. **Walidacja**: Modele pozwalają nam na definiowanie reguł walidacji dla naszych danych. Dzięki temu możemy upewnić się, że dane, które są zapisywane w naszej bazie danych, spełniają określone wymagania, np. mają odpowiedni typ, są w określonym zakresie, itp.
3. **Odniesienia (References)**: Modele umożliwiają definiowanie relacji między różnymi zestawami danych poprzez użycie odniesień (references). Dzięki temu możemy łatwo powiązać ze sobą różne dane, np. rezerwacje z klientami, którzy je dokonali.
4. **Łatwość utrzymania**: Definiowanie modeli obowiązujących w całej aplikacji znacznie ułatwia jej późniejsze utrzymanie. Jeśli struktura naszych danych ma ulec zmianie, łatwiej jest zmodyfikować kod, dodawać nowe funkcje itp.

4.4 ROUTERS

customerRouter.js

```
const express = require("express");
const customerController = require("../controllers/customerController");
const router = express.Router();

router.get("/", customerController.getAllCustomers);
router.get("/:id", customerController.getCustomerById);
router.post("/", customerController.createCustomer);
router.put("/:id", customerController.updateCustomer);
router.delete("/:id", customerController.deleteCustomer);

module.exports = router;
```

1. **router.get("/")**: Obsługujemy żądania GET na adres **/**. Przy użyciu tej ścieżki można pobrać listę wszystkich klientów.
2. **router.get("/:id")**: Ta ścieżka obsługuje żądania GET na adres **/:id**, gdzie **:id** to identyfikator klienta. Używając tej ścieżki, można pobrać szczegółowe informacje o konkretnym kliencie.
3. **router.post("/")**: Ta ścieżka obsługuje żądania POST na adres **/**. Za pomocą tej ścieżki można utworzyć nowego klienta.
4. **router.put("/:id")**: Ta ścieżka obsługuje żądania PUT na adres **/:id**, gdzie **:id** to identyfikator klienta. Używając tej ścieżki, można zaktualizować informacje o konkretnym kliencie.
5. **router.delete("/:id")**: Ta ścieżka obsługuje żądania DELETE na adres **/:id**, gdzie **:id** to identyfikator klienta. Używając tej ścieżki, można usunąć konkretnego klienta.

purchaseRouter.js

```
const express = require("express");
const purchaseController = require("../controllers/purchaseController");
const router = express.Router();

router.get("/:id", purchaseController.getCustomerPurchases);
router.put("/", purchaseController.buyReservation);

module.exports = router;
```

1. **router.get("/:id")**: Ta ścieżka obsługuje żądania GET na adres **/:id**, gdzie **:id** to identyfikator klienta. Przy użyciu tej ścieżki można pobrać informacje o wszystkich zakupach dokonanych przez konkretnego klienta.
2. **router.put("/")**: Ta ścieżka obsługuje żądania PUT na adres **/**. Używając tej ścieżki, klient może dokonać zakupu rezerwacji.

reservationRouter.js

```
const express = require("express");
const reservationController = require("../controllers/reservationController");
const router = express.Router();

router.get("/", reservationController.getAllReservations);
router.get("/:id", reservationController.getCustomerReservations);
router.post("/", reservationController.createReservation);
router.post("/review", reservationController.createReview);
router.put("/", reservationController.resignReservation);


module.exports = router;
```

1. **router.get("/")**: Ta ścieżka obsługuje żądania GET na adres **/** i zwraca wszystkie rezerwacje.
2. **router.get("/:id")**: obsługujemy żądania GET na adres **/:id**, gdzie **:id** to identyfikator klienta. Przy użyciu tej ścieżki można pobrać informacje o wszystkich rezerwacjach dokonanych przez konkretnego klienta.
3. **router.post("/")**: obsługujemy żądania POST na adres **/**. Używając tej ścieżki, klient może utworzyć nową rezerwację.
4. **router.post("/review")**: obsługujemy żądania POST na adres **/review**. Używając tej ścieżki, klient może dodać recenzję do swojej rezerwacji.
5. **router.put("/")**: Ta ścieżka obsługuje żądania PUT na adres **/**. Używając tej ścieżki, klient może zrezygnować z rezerwacji.

tripRouter.js

```
const express = require("express");
const tripController = require("../controllers/tripController");
const router = express.Router();

router.get("/", tripController.getAllTrips);
router.get("/:id", tripController.getTripById);
router.post("/", tripController.createTrip);
router.delete("/:id", tripController.deleteTrip);

module.exports = router;
```

1. **router.get("/")**: Ta ścieżka obsługuje żądania GET na adres **/** i zwraca wszystkie wycieczki.

2. **router.get("/:id")**: Tutaj obsługujemy żądania GET na adres **/:id**, gdzie **:id** to identyfikator wycieczki. Przy użyciu tej ścieżki można pobrać szczegółowe informacje o konkretnej wycieczce.
3. **router.post("/")**: obsługujemy żądania POST na adres **/**. Używając tej ścieżki, można utworzyć nową wycieczkę.
4. **router.delete("/:id")**: Ta ścieżka obsługuje żądania DELETE na adres **/:id**, gdzie **:id** to identyfikator wycieczki. Używając tej ścieżki, można usunąć konkretną wycieczkę.

Czego się nauczyliśmy?

1. **Modularność:** Tworzenie oddzielnych routerów dla różnych części aplikacji, takich jak użytkownicy, wycieczki, rezerwacje itp., pomaga utrzymać kod czystym i łatwym do zarządzania. Każdy router ma określone zadanie i obsługuje konkretne ścieżki, co ułatwia zrozumienie, co dany fragment kodu robi.
2. **Skalowalność:** Użycie routerów umożliwia łatwe skalowanie aplikacji. Gdy dodajemy nową funkcjonalność, możemy po prostu dodać nowy router bez konieczności modyfikowania istniejącego kodu.
3. **Separacja logiki:** Kontrolery, które są podłączone do routerów, są odpowiedzialne za logikę biznesową aplikacji. To oznacza, że kody routerów są krótkie i mają na celu tylko przekierowanie żądań do odpowiednich kontrolerów.

4.5 SERVER

server.js

```
const express = require("express");
const cors = require("cors");

const app = express();
const customerRouter = require("../routers/customerRouter");
const tripRouter = require("../routers/tripRouter");
const purchaseRouter = require("../routers/purchaseRouter");
const reservationRouter = require("../routers/reservationRouter");

var corsOptions = {
  origin: "http://localhost:4200"
};

app.use(cors(corsOptions));

app.use(express.json());

app.use(express.urlencoded({ extended: true }));

const mongoose = require("mongoose");


//baza danych
const dbURI = "mongodb://127.0.0.1:27017/travel_agency";
```

```
mongoose.connect(dbURI, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

mongoose.connection.on("connected", () => {
  console.log("Connected to MongoDB");
});

mongoose.connection.on("error", (error) => {
  console.log("Error connecting to MongoDB: ", error);
});


app.get("/", (req, res) => {
  res.json({ message: "Server works" });
});

app.use("/trips", tripRouter);
app.use("/customers", customerRouter);
app.use("/purchases", purchaseRouter);
app.use("/reservations", reservationRouter);


const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}.`);
});
```

Kod server.js:

1. Importuje niezbędne moduły i biblioteki, w tym Express.js dla tworzenia serwera, CORS dla obsługi zasobów z różnych źródeł, oraz Mongoose do obsługi bazy danych MongoDB.
2. Ustawia opcje CORS, aby zezwolić na żądania tylko z określonego źródła (tutaj localhost:4200).
3. Inicjalizuje serwer Express i konfiguruje go do obsługi danych JSON oraz URL-encoded.
4. Nawiązuje połączenie z bazą danych MongoDB, monitorując zarówno pomyślne połączenia, jak i błędy.
5. Definiuje różne ścieżki i metody HTTP, które serwer będzie obsługiwał, za pomocą zewnętrznie zdefiniowanych routerów.
6. W końcu uruchamia serwer, który zaczyna nasłuchiwać na określonym porcie.

5 FRONTEND – ANGULAR

Frontend naszego projektu postanowiliśmy napisać w Angularze z kilku powodów:

Struktura i organizacja:

Angular zapewnia dobrze zdefiniowaną strukturę projektu i ścisłe wytyczne dotyczące organizacji kodu. To pozwala tworzyć skalowalne i łatwe do zarządzania aplikacje.

Typowanie i sprawdzanie błędów

Angular jest oparty na języku TypeScript, który dostarcza statycznego typowania i sprawdzanie błędów już na etapie pisania kodu. To pomaga wychwycić błędy i zapobiega wielu potencjalnym problemom w trakcie rozwoju aplikacji.

Obszerna biblioteka:

Angular posiada bogatą bibliotekę komponentów i narzędzi, która ułatwia tworzenie interaktywnych interfejsów użytkownika. Zawiera wiele gotowych rozwiązań, takich jak obsługę HTTP i wiele innych.

5.1 SERVICES

db.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Reservation } from '../models/reservation';
import { Cancelled } from '../models/cancelled';
import { Trip } from '../models/trip';
import { Purchase } from '../models/purchase';
import { Review } from '../models/review';
import { Customer } from '../models/customer';

const tripsUrl = 'http://localhost:8080/trips';
const customersUrl = 'http://localhost:8080/customers';
const purchasesUrl = 'http://localhost:8080/purchases';
const reservationsUrl = 'http://localhost:8080/reservations';

@Injectable({
  providedIn: 'root'
})
export class DBService {

  constructor(private http: HttpClient) { }

  getTravels(): Observable<Trip[]> {
    return this.http.get<Trip[]>(tripsUrl)
  }

  getTravelByKey(key: string): Observable<Trip> {
    return this.http.get<Trip>(`${tripsUrl}/${key}`)
  }
```

```
  getCustomers(): Observable<Customer[]> {
    return this.http.get<Customer[]>(customersUrl)
  }

  getCustomerByKey(key: string): Observable<Customer> {
    return this.http.get<Customer>(`${customersUrl}/${key}`)
  }

  getReservations(customerId: string): Observable<Reservation[]> {
    return this.http.get<Reservation[]>(`${reservationsUrl}/${customerId}`)
  }

  getCancelled(customerId: string): Observable<Cancelled[]> {
    return
this.http.get<Cancelled[]>(`${reservationsUrl}/cancelled/${customerId}`)
  }

  getPurchases(customerId: string): Observable<Purchase[]> {
    return this.http.get<Purchase[]>(`${purchasesUrl}/${customerId}`)
  }

  newReservation(tripId: string, tickets: number, price: number, customerId:
string) {
    const body = new HttpParams().set("tripId", tripId)
                                 .set("tickets", tickets)
                                 .set("price", price)
                                 .set("customerId", customerId)

    return this.http.post<Reservation>(reservationsUrl, body).subscribe()
  }

  resignReservation(tripId: string, reservationId: string) {
    const body = new HttpParams().set("tripId", tripId)
                                 .set("reservationId", reservationId)

    return this.http.put<Reservation>(reservationsUrl, body).subscribe()
  }

  newPurchase(tripId: string, reservationId: string) {
    const body = new HttpParams().set("tripId", tripId)
                                 .set("reservationId", reservationId)

    return this.http.put<Reservation>(purchasesUrl, body).subscribe()
  }

  newOpinion(tripId: string, reservationId: string, comment: string, rating:
number) {
    const body = new HttpParams().set("tripId", tripId)
```

```
                                    .set("reservationId", reservationId)
                                    .set("comment", comment)
                                    .set("rating", rating)

    return this.http.post<Reservation>(`${reservationsUrl}/review`,
body).subscribe()
  }

  newTravel(title: string, destination: string, description: string,
startDate: string, endDate: string, seats: number, unitPrice: number)
{
    const body = new HttpParams().set("title", title)
                                    .set("destination", destination)
                                    .set("description", description)
                                    .set("startDate", startDate)
                                    .set("endDate", endDate)
                                    .set("seats", seats)
                                    .set("unitPrice", unitPrice)
    return this.http.post<Trip>(tripsUrl, body).subscribe()
  }

  deleteTravel(key: string) {
    return this.http.delete<Trip>(`${tripsUrl}/${key}`).subscribe()
  }

  newCustomer(c : Customer) {
    const body = new HttpParams().set("firstName", c.firstName)
                                    .set("lastName", c.lastName)
                                    .set("userName", c.username)
                                    .set("phoneNumber", c.phoneNumber)
                                    .set("email", c.email)
                                    .set("country", c.address.country)
                                    .set("city", c.address.city)
                                    .set("street", c.address.street)
                                    .set("postalCode", c.address.postalCode)
                                    .set("buildingNumber",
c.address.buildingNumber)
                                    .set("apartmentNumber",
c.address.apartmentNumber)
    return this.http.post<Customer>(customersUrl, body).subscribe()
  }

  deleteCustomer(id: string) {
    return this.http.delete<Customer>(`${customersUrl}/${id}`).subscribe()
  }
}
```

1. **getTravels():** Ta metoda jest używana do pobrania listy wszystkich podróży z serwera. Wykorzystuje ona metodę GET HTTP.
2. **getTravelByKey(key: string):** Ta metoda służy do pobierania szczegółów konkretnej podróży na podstawie jej klucza. Także używa metody GET HTTP.
3. **getCustomers():** Ta metoda służy do pobierania listy wszystkich klientów. Także korzysta z metody GET HTTP.
4. **getCustomerByKey(key: string):** Służy do pobierania szczegółów konkretnego klienta na podstawie jego klucza. Także używa metody GET HTTP.
5. **getReservations(customerId: string):** Ta metoda jest używana do pobierania rezerwacji konkretnego klienta. Wykorzystuje ona metodę GET HTTP.
6. **getCancelled(customerId: string):** Ta metoda jest używana do pobierania anulowanych rezerwacji konkretnego klienta. Wykorzystuje ona metodę GET HTTP.
7. **getPurchases(customerId: string):** Ta metoda jest używana do pobierania zakupów konkretnego klienta. Wykorzystuje ona metodę GET HTTP.
8. **newReservation(...):** Ta metoda służy do tworzenia nowej rezerwacji. Wykorzystuje ona metodę POST HTTP.
9. **resignReservation(...):** Ta metoda służy do rezygnacji z rezerwacji. Wykorzystuje ona metodę PUT HTTP.
10. **newPurchase(...):** Ta metoda jest używana do tworzenia nowego zakupu. Wykorzystuje ona metodę PUT HTTP.
11. **newOpinion(...):** Ta metoda jest używana do tworzenia nowej opinii. Wykorzystuje ona metodę POST HTTP.
12. **newTravel(...):** Ta metoda służy do tworzenia nowego wyjazdu. Wykorzystuje ona metodę POST HTTP.
13. **deleteTravel(key: string):** Ta metoda służy do usuwania podróży. Wykorzystuje ona metodę DELETE HTTP.
14. **newCustomer(c : Customer):** Ta metoda służy do tworzenia nowego klienta. Wykorzystuje ona metodę POST HTTP.
15. **deleteCustomer(id: string):** Ta metoda służy do usuwania klienta. Wykorzystuje ona metodę DELETE HTTP.

travel.service.ts

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Review } from '../models/review';
import { Trip } from '../models/trip';
import { DBService } from './db.service';
import { of } from 'rxjs';
```

```
@Injectable({
  providedIn: 'root'
})
export class TravelService {
  travels : Trip[] = []
  locations : string[] = []
  selectedLocations : string[] = []
  selectedStars : number[] = []
  selectedPrice : number
  minDate : Date
  maxDate : Date
  showArchiwe : boolean

  constructor(private dbService: DBService) {
    this.dbService.getTravels().subscribe(res => {
      this.travels = res
      this.setLocations()
      this.selectedPrice = this.maxPrice
      this.minDate = new Date()
      this.setMaxDate()
      this.showArchiwe = false
    })
  }

  refreshTravels() {
    this.dbService.getTravels().subscribe(res => {
      this.travels = res
      this.setLocations()
    })
  }

  getTravelByKey(key: string): Observable<Trip> {
    return this.dbService.getTravelByKey(key)
  }

  get maxPrice() {
    if(this.travels.length == 0) return 0
    let max = this.travels[0].unitPrice
    for(const travel of this.travels) if(travel.unitPrice > max) max =
travel.unitPrice
    return max
  }

  get minPrice() {
    if(this.travels.length == 0) return 0
    let min = this.travels[0].unitPrice
```

```
    for(const travel of this.travels) if(travel.unitPrice < min) min =
travel.unitPrice
    return min
  }

  changeArchiwe(b : boolean) {
    if(b) this.setMinDate()
    else this.minDate = new Date()
    this.showArchiwe = b
  }

  setMaxDate() {
    let date = this.travels[0].endDate
    for(const travel of this.travels) {
      let curDate = travel.endDate
      if(curDate > date) date = curDate
    }

    this.maxDate = new Date(date)
  }

  setMinDate() {
    let date = this.travels[0].startDate
    for(const travel of this.travels) {
      let curDate = travel.startDate
      if(curDate < date) date = curDate
    }

    this.minDate = new Date(date)
  }

  setLocations() {
    this.locations = this.travels.map(x => x.destination).reduce((acc:
string[], cur: string) => {
      if(!acc.includes(cur)) acc.push(cur)
      return acc
    }, []).sort()
  }

  newOpinion(tripId: string, reservationId: string, comment: string, rating:
number) {
    if(reservationId == '') return -1
    this.dbService.newOpinion(tripId, reservationId, comment, rating)
    return 1
  }

  newTravel(title: string, destination: string, description: string,
startDate: string, endDate: string, seats: number, unitPrice: number)
{
```

```
    this.dbService.newTravel(title, destination, description, startDate,
endDate, seats, unitPrice)
    this.refreshTravels()
  }

  deleteTravel(tripId : string) {
    this.dbService.deleteTravel(tripId)
    this.refreshTravels()
  }
}
```

1. **constructor(private dbService: DBService)**: Konstruktor klasy **TravelService**, który przyjmuje instancję **DBService** jako argument. Wywołuje metodę **getTravels()** z **dbService**, aby pobrać listę podróży i inicjalizuje różne zmienne, takie jak **travels**, **locations**, **selectedLocations**, **selectedStars**, **selectedPrice**, **minDate**, **maxDate** oraz **showArchive**.
2. **refreshTravels()**: Metoda służąca do odświeżenia listy podróży. Wywołuje metodę **getTravels()** z **dbService**, aby pobrać najnowszą listę podróży i zaktualizować zmienną **travels** oraz **locations**.
3. **getTravelByKey(key: string): Observable<Trip>**: Metoda służąca do pobrania szczegółów podróży na podstawie podanego klucza. Wywołuje metodę **getTravelByKey(key)** z **dbService** i zwraca obiekt typu **Observable<Trip>**, który można subskrybować.
4. **get maxPrice(): number**: Właściwość tylko do odczytu, która zwraca maksymalną cenę podróży spośród wszystkich dostępnych podróży. Przechodzi przez listę **travels** i znajduje maksymalną wartość **unitPrice**.
5. **get minPrice(): number**: Właściwość tylko do odczytu, która zwraca minimalną cenę podróży spośród wszystkich dostępnych podróży. Przechodzi przez listę **travels** i znajduje minimalną wartość **unitPrice**.
6. **setMaxDate()**: Metoda służąca do ustawienia maksymalnej daty podróży. Przechodzi przez listę **travels** i znajduje najpóźniejszą datę **endDate**. Ustawia **maxDate** na tę wartość.
7. **setMinDate()**: Metoda służąca do ustawienia minimalnej daty podróży. Przechodzi przez listę **travels** i znajduje najwcześniejszą datę **startDate**. Ustawia **minDate** na tę wartość.
8. **setLocations()**: Metoda służąca do ustawienia dostępnych lokalizacji podróży. Przetwarza listę **travels** i tworzy unikalną listę lokalizacji **destination**, która jest posortowana alfabetycznie. Ustawia wartość tej listy w zmiennej **locations**.
9. **newOpinion(tripId: string, reservationId: string, comment: string, rating: number)**: Metoda służąca do dodawania nowej opinii na temat podróży. Jeśli **reservationId** jest puste, zwraca -1. W przeciwnym razie, wywołuje metodę **newOpinion(tripId, reservationId, comment, rating)** z **dbService**, aby dodać nową opinię.
10. **newTravel(title: string, destination: string, description: string, startDate: string, endDate: string, seats: number, unitPrice: number)**: Metoda służąca do tworzenia nowej podróży. Wywołuje metodę **newTravel(title, destination, description, startDate, endDate, seats, unitPrice)** z **dbService**, aby utworzyć nową podróż, a następnie wywołuje **refreshTravels()** w celu odświeżenia listy podróży.

11. **deleteTravel(tripId: string)**: Metoda służąca do usuwania podróży. Wywołuje metodę **deleteTravel(tripId)** z **dbService**, aby usunąć podróż, a następnie wywołuje **refreshTravels()** w celu odświeżenia listy podróży.

cart.service.ts

```
import { Injectable } from '@angular/core';
import { DBService } from './db.service';
import { Reservation } from '../models/reservation';
import { Trip } from '../models/trip';
import { Customer } from '../models/customer';
import { Observable } from 'rxjs';
import { Purchase } from '../models/purchase';
import { Cancelled } from '../models/cancelled';
import { TravelService } from './travel.service';

@Injectable({
  providedIn: 'root'
})
export class CartService {
  reservations : Reservation[] = []
  purchases : Purchase[] = []
  customers: Customer[] = []
  currentCustomer : Customer
  startCustomerId = "6447a9ead297195ac0dd240c"

  ts

  fullPrice = 0
  fullReservations = 0

  constructor(private dbService: DBService, private tService : TravelService )
{
    if(localStorage.getItem('lastLogged') !== null) {
     this.startCustomerId = localStorage.getItem('lastLogged') || ""
    }

    this.ts = tService
    this.dbService.getCustomers().subscribe(res => this.customers = res)
    this.dbService.getCustomerByKey(this.startCustomerId).subscribe(res => {
      this.currentCustomer = res

      this.dbService.getReservations(this.currentCustomer._id).subscribe(res
=> {
```

```
        this.reservations = res
        this.fullReservations = this.reservations.map(r =>
r.tickets).reduce((a, b) => a + b, 0)
        this.fullPrice = this.reservations.map(r => r.tickets *
r.price).reduce((a, b) => a + b, 0)
      })
      this.dbService.getPurchases(this.currentCustomer._id).subscribe(res => {
        this.purchases = res
      })
    })
  }

  getReservationsById(id: string) {
    return this.dbService.getReservations(id).subscribe()
  }

  refreshCustomers() {
    this.dbService.getCustomers().subscribe(res => this.customers = res)
  }

  refreshCustomer() {
    this.dbService.getReservations(this.currentCustomer._id).subscribe(res =>
{
      this.reservations = res
      this.fullReservations = this.reservations.map(r => r.tickets).reduce((a,
b) => a + b, 0)
      this.fullPrice = this.reservations.map(r => r.tickets *
r.price).reduce((a, b) => a + b, 0)
    })
    this.dbService.getPurchases(this.currentCustomer._id).subscribe(res => {
      this.purchases = res
    })
  }

  changeCustomer(id: string) {
    this.dbService.getCustomerByKey(id).subscribe(res => {
      this.currentCustomer = res

      this.dbService.getReservations(this.currentCustomer._id).subscribe(res
=> {
        this.reservations = res
        this.fullReservations = this.reservations.map(r =>
r.tickets).reduce((a, b) => a + b, 0)
        this.fullPrice = this.reservations.map(r => r.tickets *
r.price).reduce((a, b) => a + b, 0)
      })
      this.dbService.getPurchases(this.currentCustomer._id).subscribe(res => {
        this.purchases = res
      })
```

```
    })
    localStorage.setItem('lastLogged', id)
  }

  purchaseForThisTrip(tripId: string) {
    for(const purchase of this.purchases) {
      if(purchase.tripId === tripId) {
        if(purchase.review)
          return ''
        else
          return purchase._id
      }
    }
    return ''
  }

  reserve(tripId: string, tickets: number, price: number) {
    this.dbService.newReservation(tripId, tickets, price,
this.currentCustomer._id)
    setInterval(this.ts.refreshTravels, 100)
    this.refreshCustomer()
  }

  resign(tripId : string, reservationId: string) {
    this.dbService.resignReservation(tripId, reservationId)
    this.ts.refreshTravels()
    this.refreshCustomer()
  }

  buy(tripId : string, reservationId: string) {
    this.dbService.newPurchase(tripId, reservationId)
    this.ts.refreshTravels()
    this.refreshCustomer()
  }

  getCustomerByKey(key: string): Observable<Customer> {
    return this.dbService.getCustomerByKey(key)
  }

  newCustomer(c : Customer) {
    this.dbService.newCustomer(c)
    this.refreshCustomers()
  }

  deleteCustomer(id : string)  {
    this.dbService.deleteCustomer(id)
    this.refreshCustomers()
  }
}
```

1. **constructor(private dbService: DBService, private tService: TravelService)**: Konstruktor klasy **CartService**, który przyjmuje instancje **DBService** i **TravelService** jako argumenty. Inicjalizuje różne zmienne, takie jak **reservations**, **purchases**, **customers**, **currentCustomer**, **startCustomerId**, **ts**, **fullPrice** oraz **fullReservations**. Wykonuje również operacje inicjalizacyjne, takie jak pobranie listy klientów, pobranie klienta o danym kluczu, pobranie rezerwacji i zakupów tego klienta.

2. **getReservationsById(id: string)**: Metoda służąca do pobrania rezerwacji na podstawie ID klienta. Wywołuje metodę **getReservations(id)** z **dbService** i zwraca obiekt **Observable**, który można subskrybować.

3. **refreshCustomers()**: Metoda służąca do odświeżenia listy klientów. Wywołuje metodę **getCustomers()** z **dbService** w celu pobrania najnowszej listy klientów i aktualizuje zmienną **customers**.

4. **refreshCustomer()**: Metoda służąca do odświeżenia danych bieżącego klienta. Wywołuje metodę **getReservations(this.currentCustomer._id)** z **dbService** w celu pobrania najnowszej listy rezerwacji dla bieżącego klienta oraz aktualizuje zmienne **reservations**, **fullReservations** i **fullPrice** na podstawie tych danych. Wywołuje również metodę **getPurchases(this.currentCustomer._id)** z **dbService** w celu pobrania najnowszej listy zakupów dla bieżącego klienta.

5. **changeCustomer(id: string)**: Metoda służąca do zmiany bieżącego klienta na podstawie podanego klucza. Wywołuje metodę **getCustomerByKey(id)** z **dbService** w celu pobrania klienta o danym kluczu i aktualizuje zmienne związane z danymi klienta oraz pobiera najnowsze rezerwacje i zakupy dla tego klienta. Zapisuje również klucz klienta w lokalnym magazynie za pomocą **localStorage.setItem('lastLogged', id)**.

6. **purchaseForThisTrip(tripId: string)**: Metoda służąca do sprawdzenia, czy dla podanego ID podróży istnieje już zakup. Przeszukuje listę zakupów i zwraca ID zakupu, jeśli istnieje, w przeciwnym razie zwraca pusty ciąg znaków.

7. **reserve(tripId: string, tickets: number, price: number)**: Metoda służąca do dokonania rezerwacji dla podróży. Wywołuje metodę **newReservation(tripId, tickets, price, this.currentCustomer._id)** z **dbService**, aby utworzyć nową rezerwację. Następnie wywołuje **refreshTravels()** z **ts** w celu odświeżenia listy podróży oraz **refreshCustomer()** w celu odświeżenia danych bieżącego klienta.

8. **resign(tripId: string, reservationId: string)**: Metoda służąca do rezygnacji z rezerwacji. Wywołuje metodę **resignReservation(tripId, reservationId)** z **dbService**, aby zrezygnować z rezerwacji. Następnie wywołuje **refreshTravels()** z **ts** w celu odświeżenia listy podróży oraz **refreshCustomer()** w celu odświeżenia danych bieżącego klienta.

9. **buy(tripId: string, reservationId: string)**: Metoda służąca do dokonania zakupu. Wywołuje metodę **newPurchase(tripId, reservationId)** z **dbService**, aby dokonać zakupu. Następnie wywołuje **refreshTravels()** z **ts** w celu odświeżenia listy podróży oraz **refreshCustomer()** w celu odświeżenia danych bieżącego klienta.

10. **getCustomerByKey(key: string): Observable<Customer>**: Metoda służąca do pobrania danych klienta na podstawie podanego klucza. Wywołuje metodę **getCustomerByKey(key)** z **dbService** i zwraca obiekt typu **Observable<Customer>**, który można subskrybować.
11. **newCustomer(c: Customer)**: Metoda służąca do tworzenia nowego klienta. Wywołuje metodę **newCustomer(c)** z **dbService**, aby utworzyć nowego klienta, a następnie wywołuje **refreshCustomers()** w celu odświeżenia listy klientów.
12. **deleteCustomer(id: string)**: Metoda służąca do usuwania klienta. Wywołuje metodę **deleteCustomer(id)** z **dbService**, aby usunąć klienta, a następnie wywołuje **refreshCustomers()** w celu odświeżenia listy klientów.