

Biuro wycieczek

Paweł Konop, Kacper Korta

Aplikacja jest narzędziem wspomagającym lokalne biuro podróży, które organizuje kilkanaście wycieczek w ciągu miesiąca. Umożliwia logowanie, rejestrację użytkowników oraz oferuje funkcje rezerwacji i zakupu wycieczek. Wykorzystuje technologię ExpressJS do budowy interfejsu API oraz Angulara do interfejsu użytkownika.

Spis treści:

Kolekcje:.....	2
a) orders.....	2
b) reviews.....	2
c) tours.....	3
d) users.....	4
 Endpointy.....	4
a) rejestrowanie użytkownika:.....	4
b) logowanie użytkownika:.....	6
c) pobieranie aktualnych wycieczek.....	8
d) składanie zamówień przez zalogowanych użytkowników:.....	10
e) dodanie recenzji przez zalogowanych użytkowników:.....	12
f) pobieranie zamówień konkretnego klienta:.....	14
g) pobieranie recenzji konkretnej wycieczki.....	15

Kolekcje:

Wyróżniliśmy 4 kolekcje:

- a) orders (server/src/schemas/orders.ts) - kolekcja, która przechowuje dokumenty reprezentujące zamówienia:
1. **userId**: Pole typu *String*, przechowujące identyfikator użytkownika złożonego zamówienia.
 2. **tourId**: Pole typu *String*, przechowujące identyfikator wycieczki (tour), która została zamówiona.
 3. **amount**: Pole typu *Number*, przechowujące wartość liczbową reprezentującą ilość zamówionych wycieczek.
 4. **date**: Pole typu *Date*, przechowujące datę złożenia zamówienia.

```
import mongoose, {Schema} from "mongoose";

const orderSchema = new mongoose.Schema( definition: {
  userId: String,
  tourId: String,
  amount: Number,
  date: Date
})

export const OrdersModel = mongoose.model( name: 'Orders', orderSchema)
```

- b) reviews (server/src/schemas/reviews.ts) - kolekcja, odpowiadająca za

przechowywanie informacji o opinach klientów dotyczących wycieczek:

1. **userId**: Pole typu *String*, przechowujące identyfikator użytkownika, który wystawił opinię.
2. **nickname**: Pole typu *String*, przechowujące pseudonim użytkownika, który wystawił opinię.
3. **tourId**: Pole typu *String*, przechowujące identyfikator wycieczki (tour), dla której została wystawiona opinia.
4. **description**: Pole typu *String*, przechowujące treść opinii.
5. **date**: Pole typu *Date*, przechowujące datę, kiedy opinia została dodana.

```
import mongoose from "mongoose";

const reviewSchema = new mongoose.Schema( definition: {
  userId: String,
  nickname: String,
  tourId: String,
  description: String,
  date: Date
})

export const ReviewsModel = mongoose.model( name: 'Reviews', reviewSchema)
```

c) tours (server/src/schemas/tours.ts) - kolekcja, która przechowuje dokumenty reprezentujące wycieczki:

1. **name**: Pole typu *String*, przechowujące nazwę wycieczki.
2. **country**: Pole typu *String*, przechowujące nazwę kraju, w którym odbywa się wycieczka.
3. **startDate**: Pole typu *String*, przechowujące datę rozpoczęcia wycieczki.
4. **endDate**: Pole typu *String*, przechowujące datę zakończenia wycieczki.
5. **price**: Pole typu *Number*, przechowujące cenę wycieczki.
6. **initialLimit**: Pole typu *Number*, przechowujące początkowy limit miejsc dostępnych dla wycieczki.
7. **limit**: Pole typu *Number*, przechowujące aktualny limit miejsc dostępnych dla wycieczki.
8. **description**: Pole typu *String*, przechowujące opis wycieczki.
9. **photoLink**: Pole typu *String*, przechowujące link do zdjęcia powiązanego z wycieczką.
10. **votes**: Pole typu *Number*, przechowujące liczbę głosów dla wycieczki.

```
import mongoose from "mongoose";

const tourSchema = new mongoose.Schema( definition: {
    name: String,
    country: String,
    startDate: String,
    endDate: String,
    price: Number,
    initialLimit: Number,
    limit: Number,
    description: String,
    photoLink: String,
    votes: Number
})

export const ToursModel = mongoose.model( name: 'Tours', tourSchema)
```

d) users (server/src/schemas/users.ts) - kolekcja zawierająca dokumenty reprezentujące informacje o użytkownikach:

1. **email**: Pole typu *String*, przechowujące adres e-mail użytkownika.
2. **password**: Pole typu *String*, przechowujące zaszyfrowane hasło użytkownika.

3. **role**: Pole typu *Number*, przechowujące rolę użytkownika. Wartość domyślna to 1.
4. **blocked**: Pole typu *Boolean*, przechowujące informację, czy konto użytkownika jest zablokowane. Wartość domyślna to *false*.

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema( definition: {
  email: String,
  password: String,
  role: {type: Number, default: 1},
  blocked: {type: Boolean, default: false}
})

export const UsersModel = mongoose.model( name: 'Users', userSchema)
```

Endpointy (server/src/server.ts):

- 1) **/register** dla żądań typu **POST** - rejestrowanie użytkownika:

- a) zrzut kodu:

```
app.post( path: "/register", handlers: async ( req : ... , res : Response<ResBody, Locals> ) => {

  const userData = req.body

  try{
    const user = await UsersModel.findOne({email: userData.email})

    if(user){
      res.status( code: 400).send( body: {message: "Email already exists"})
      return
    }

    const userInstance = new UsersModel(req.body)
    await userInstance.save()
    let payload = {subject: userInstance._id}
    let token = jwt.sign(payload, secretOrPrivateKey: 'secretKey', options: {expiresIn: "30min"})
    res.status( code: 200).send( body: {token, email: userInstance.email, role: userInstance.role,
      blocked: userInstance.blocked, _id: userInstance._id})

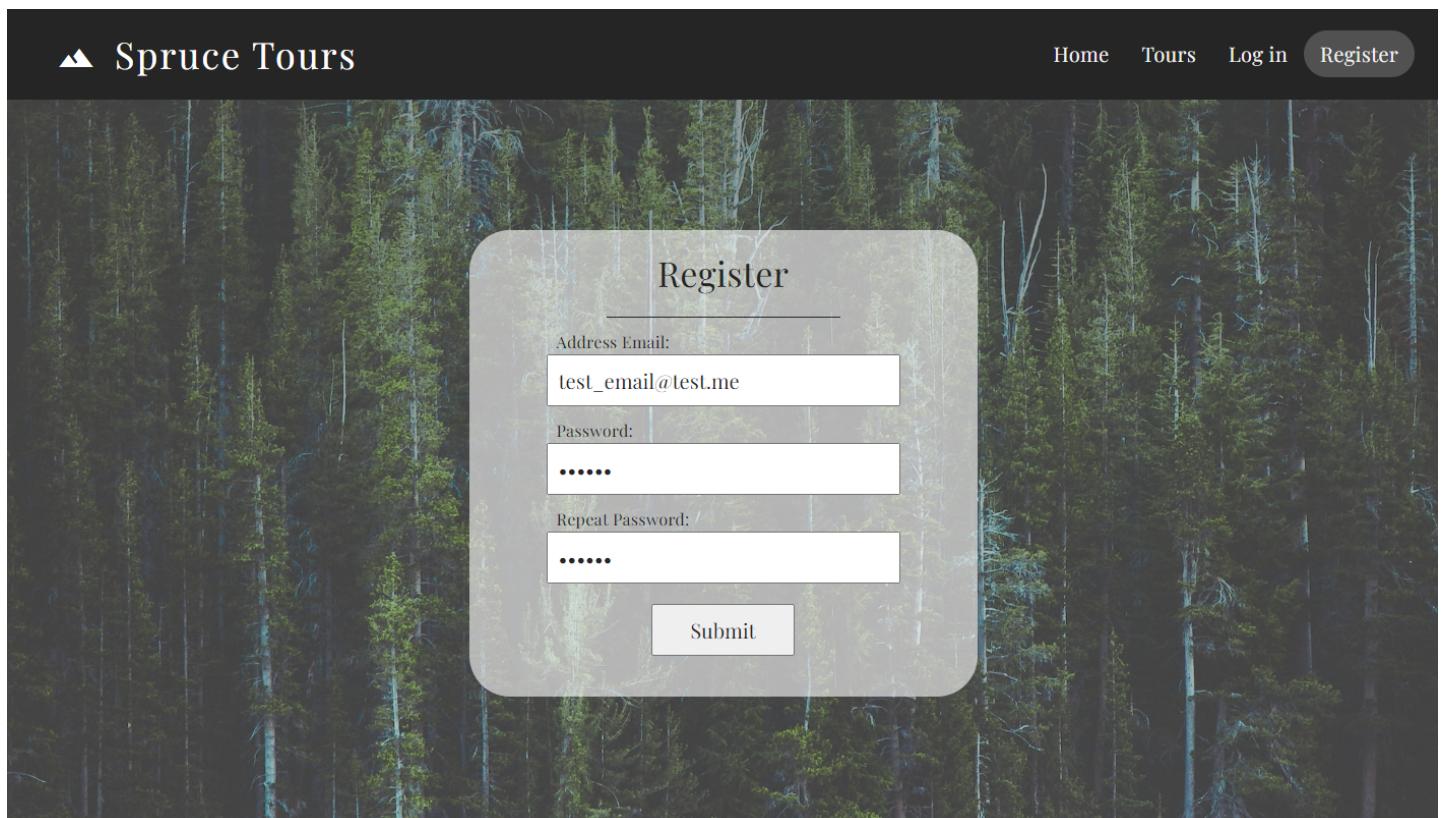
  }catch (err){
    res.status( code: 500).send( body: {message: "Internal server error"})
  }

});
```

b) Opis kodu

- I. Przyjmujemy dane żądania
- II. Sprawdzamy istnienie użytkownika
- III. Tworzymy nowego użytkownika
- IV. Generujemy token uwierzytelniający
- V. Zwracamy odpowiedź serwera

c) Działanie - rejestrujemy użytkownika:



d) Rezultat:

```
▼  tours-application
  └─ orders
  └─ reviews
  └─ tours
    └─ users   ***
        blocked: false
        __v: 0

        _id: ObjectId('648f2e309a4e11927cf3080d')
        email: "test_email@test.me"
        password: "123456"
        role: 1
        blocked: false
        __v: 0
```

2) **/login** dla żądań typu **POST** - logowanie użytkownika:

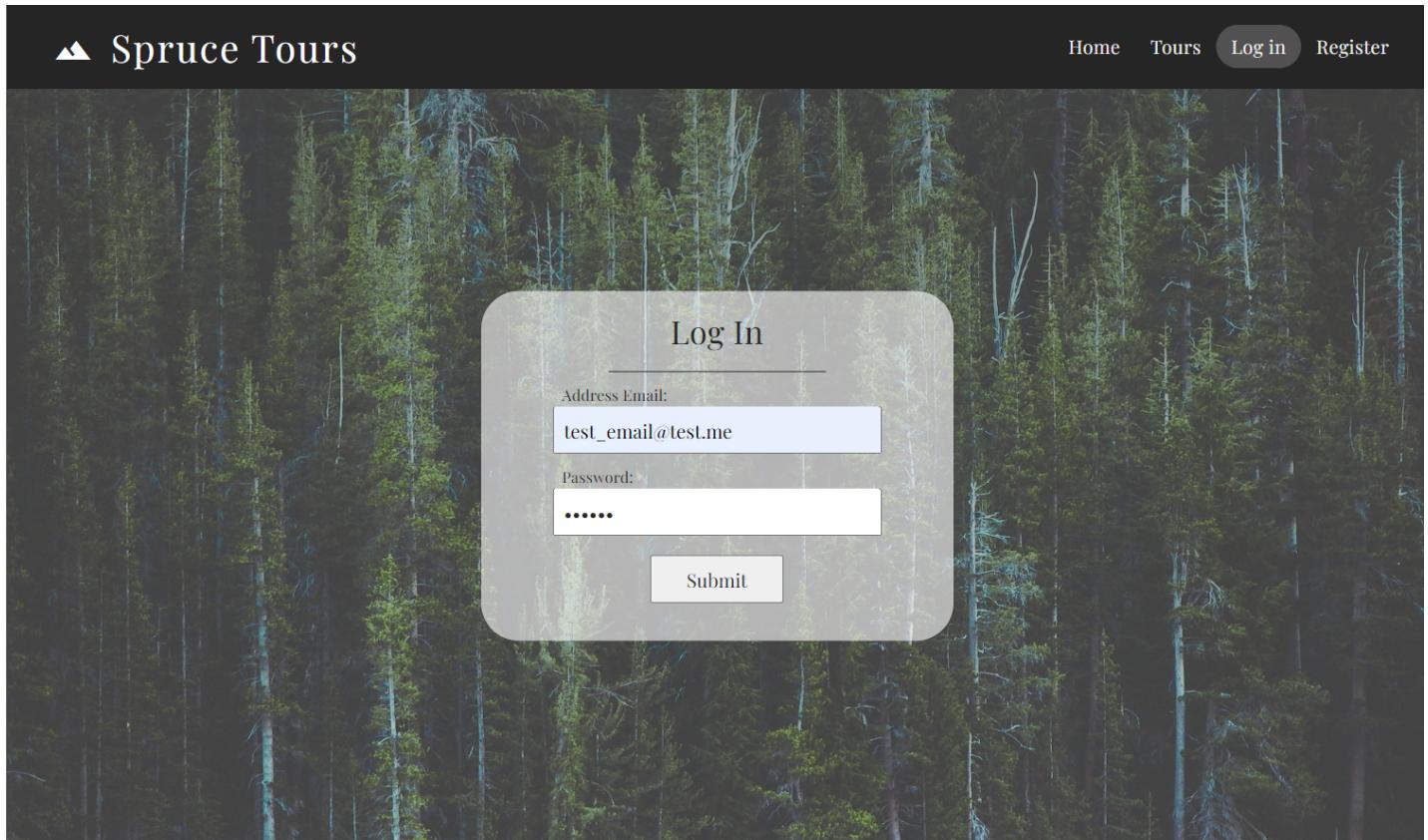
a) zrzut kodu

```
app.post( path: "/login", handlers: async ( req : Request<{}, any, any, QueryStr... , res : Response<any, Record<string, a... > ) : Promise<void> => {  
  
    const userData = req.body  
  
    try{  
        const user : Query = await UsersModel.findOne({email: userData.email})  
        if(!user){  
            res.status( code: 400).send( body: {message: "Wrong email or password"})  
            return  
        }  
        if(user.password !== userData.password){  
            res.status( code: 400).send( body: {message: "Wrong email or password"})  
            return  
        }  
        let payload :{subject: Types.ObjectId} = {subject: user._id}  
        let token :string = jwt.sign(payload, secretOrPrivateKey: 'secretKey', options: {expiresIn: "30min"})  
        res.status( code: 200).send( body: {token, email: user.email, role: user.role, blocked: user.blocked, _id: user._id})  
  
    }catch (err){  
        res.status( code: 500).send( body: {message: "Internal server error"})  
    }  
  
});
```

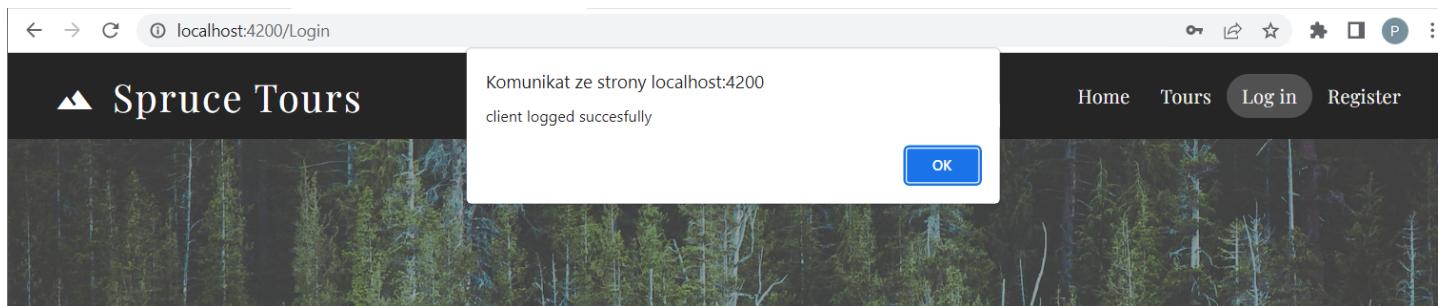
b) Opis kodu:

- I. Przyjmujemy dane żądania
- II. Za pomocą maila szukamy użytkownika w bazie
- III. Jeśli użytkownik nie zostanie znaleziony wyrzucany jest błąd
- IV. Jeśli użytkownik zostanie znaleziony ale hasło będzie błędne wyrzucany jest błąd
- V. Jeśli nie nastąpi żaden błąd kod generuje token (JWT) dla użytkownika
- VI. Zwracamy odpowiedź serwera

c) Działanie - logujemy się:



d) Rezultat po pomyślnym zalogowaniu się:



3) **/tours** dla żądań typu **GET** - pobieranie wycieczek, które są aktualne (data rozpoczęcia jest większa od obecnej daty):

a) zrzut kodu:

```
± Paweł *
app.get("/tours", async (req : Request<{}, any, any, QueryStr... , res : Response<any, Record<string, a... > ) : Promise<void> => {
  try{
    const tours : any[] = await ToursModel.aggregate([
      {
        $addFields: {
          parsedStartDate: {
            $dateFromString: {
              dateString: {
                $concat: [
                  { $substrCP: ["$startDate", 0, { $indexOfCP: ["$startDate", " "] }] },
                  { $substr: ["$startDate", { $add: [{ $indexOfCP: ["$startDate", " "] }, 1] }, 2 ] },
                  { $substr: ["$startDate", { $add: [{ $indexOfCP: ["$startDate", " "] }, 4] }, 5 ] }
                ]
              }
            }
          }
        }
      },
      {
        $match: {
          parsedStartDate: { $gt: new Date() }
        }
      }
    ]);
    res.send(tours)
  }catch (err){
    res.status( code: 500).send( body: {message: "Internal server error"})
  }
})
```

b) Opis kodu:

- I. Przyjmujemy dane żądania
- II. Korzystamy z metody agregującej w celu znalezienia wycieczek z datą rozpoczęcia późniejszą niż dzisiaj
- III. Wynik agregacji zostaje zapisany do zmiennej tours i jeśli wszystko się powiodło kod wysyła odpowiedź z pobranymi danymi

c) Działanie + rezultat - listowanie na froncie tylko aktualnych wycieczek:

The screenshot shows the homepage of Spruce Tours. At the top right, there is a navigation bar with links for Home, Tours, Log in, and Register. The main content area features three tour packages displayed in rounded rectangular boxes against a background of a forest scene.

- DESTINATION VACATION**
Country: CANADA
Start Date: Jan 29, 2024
End Date: Feb 8, 2024
Limit: 36
Price: 499\$
+ -
- HOP-ON, HOP-OFF**
Country: GREECE
Start Date: Nov 16, 2023
End Date: Nov 29, 2023
Limit: 0
Price: 899\$
+ -
- MIND THE GAP**
Country: INDONESIA
Start Date: Apr 16, 2024
End Date: Apr 24, 2024
Limit: 20
Price: 1999\$
+ -

4) **/orders** dla żądań typu **POST** - składanie zamówień przez zalogowanych użytkowników:

a) Zrzut kodu:

```
app.post( path: "/orders", authClient, async ( req: ... , res: ... ) => {  
  
    const data = req.body  
  
    try{  
        const user = await UsersModel.findOne({email: data.email})  
        if(!user){  
            res.status( code: 400).send( body: {message: "User does not exist"})  
            return  
        }  
        const tour = await ToursModel.findOne({_id: data.tourId})  
        if(!tour){  
            res.status( code: 400).send( body: {message: "Tour does not exist"})  
            return  
        }  
  
        const updatedTour = await ToursModel.findByIdAndUpdate(  
            tour._id,  
            { $inc: { limit: -data.amount } },  
            { new: true }  
        );  
  
        const orderInstance = new OrdersModel( doc: {userId: user._id, tourId: tour._id, amount: data.amount, date: data.date})  
        await orderInstance.save()  
        res.status( code: 200).send()  
  
    }catch (err){  
        res.status( code: 500).send( body: {message: "Internal server error"})  
    }  
};
```

b) Opis kodu:

- I. Przyjmujemy dane żądania
- II. Uwierzytelniamy klienta
- III. Sprawdzamy czy użytkownik istnieje a następnie czy zakupiona wycieczka istnieje
- IV. Updatujemy ilość miejsc dla zakupionej wycieczki

- V. Tworzymy obiekt orderInstance z danymi i dodajemy do bazy
VI. Zwracamy odpowiedź serwera

c) Działanie - kupujemy wycieczkę:

Entry	Total	Start Date	End Date	No.	Buy me!
1	1996 USD	Jan 29, 2024	Feb 8, 2024	4	<input type="button" value="4"/> BUY

d) Rezultat - zupdatowanie limitu miejsc wycieczki oraz dodanie zamówienia do bazy:

MongoDB Compass - ToursApplication/tours-application.orders

Connect Edit View Collection Help

ToursApplication ... Documents +

My Queries Databases Search

tours-application.orders 53 1 DOCUMENTS INDEXES

Documents Aggregations Schema Explain Plan Indexes Validation

Filter Type a query: { field: 'value' } Reset Find More Options >

+ ADD DATA EXPORT DATA 41 - 53 of 53

amount: 4
date: 2023-06-20T10:55:54.000+00:00
_id: ObjectId('64918e29a5116002c723a559')
userId: "648f2e309a4e11927cf3080d"
tourId: "638cde20196fcfa5f20e88e38"
amount: 4
date: 2023-06-20T11:31:52.000+00:00
_v: 0

> MONGOSH

5) **/reviews** dla żądań typu **POST**- dodanie recenzji przez zalogowanych użytkowników:

a) zrzut kodu:

```
app.post( path: "/reviews", authClient, async ( req: ..., res: ... ) => {
    const data = req.body

    try{
        const user = await UserModel.findOne({ _id: data.userId })
        if(!user){
            res.status( code: 400 ).send( body: { message: "User does not exist" })
            return
        }
        const tour = await ToursModel.findOne({ _id: data.tourId })
        if(!tour){
            res.status( code: 400 ).send( body: { message: "Tour does not exist" })
            return
        }

        const reviewInstance = new ReviewsModel( doc: { userId: data.userId, nickname: data.nickname,
            tourId: data.tourId, description: data.description, date: data.date } )

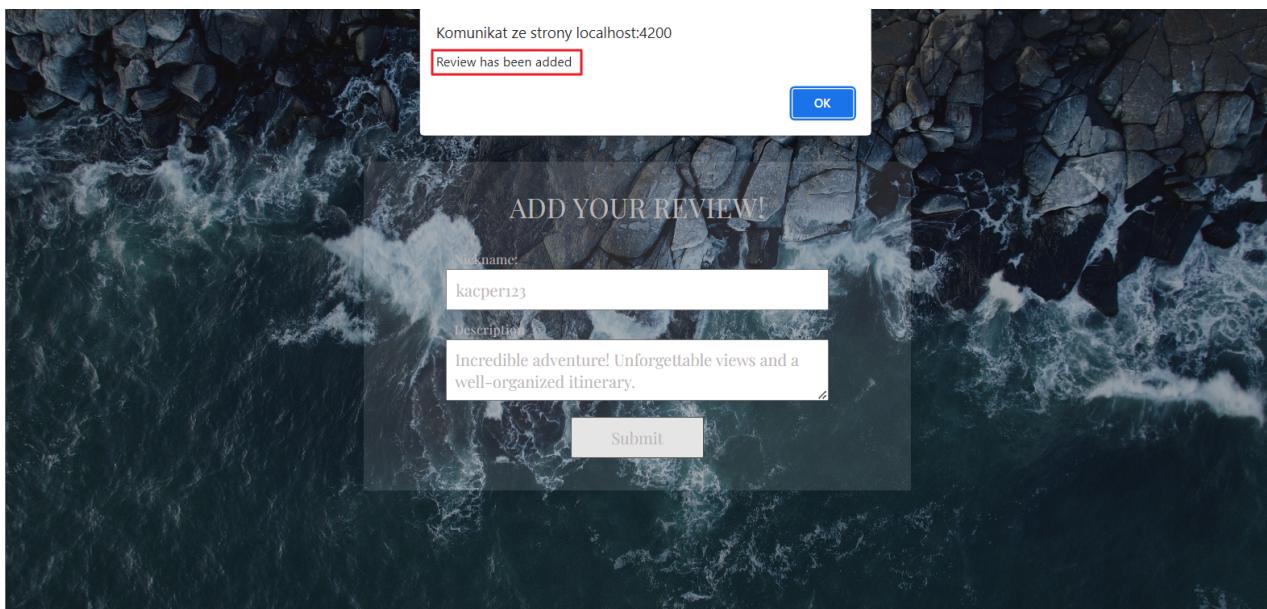
        await reviewInstance.save()
        res.status( code: 200 ).send()

    }catch (err:any){
        res.status( code: 500 ).send( body: { message: "Internal server error" })
    }
});
```

b) Opis kodu:

- I. Przyjmujemy dane żądania
- II. Uwierzytelniamy klienta
- III. Sprawdzamy czy użytkownik istnieje a następnie czy wycieczka istnieje
- IV. Tworzymy obiekt reviewInstance z danymi i dodajemy do bazy
- V. Zwracamy odpowiedź serwera

c) Działanie kodu - dodajemy recenzję dla wycieczki *Destination Vacation*:



d) Rezultat - dodanie recenzji do bazy:

A screenshot of the MongoDB Compass interface. The top navigation bar shows "MongoDB Compass - ToursApplication/tours-application.reviews". The left sidebar lists databases and collections, with "tours-application" selected and "reviews" highlighted. The main pane shows the "tours-application.reviews" collection with 17 documents and 1 index. A search bar and filter options are at the top of the list. Below the list are buttons for "ADD DATA" and "EXPORT DATA". Two documents are visible in the list, both of which have their content highlighted with a red box. The first highlighted document is a review entry with fields: nickname: "Jerry", tourId: "638cde20196fcfa5f20e88e3a", description: "The tour was just OK. The locations were nice to visit but the activit...", date: "2023-06-18T11:56:05.000+00:00", and __v: 0. The second highlighted document is another review entry with similar fields: _id: ObjectId('6491945799d998a98bad84df'), userId: "648f2e309a4e11927cf3080d", nickname: "kacper123", tourId: "638cde20196fcfa5f20e88e38", description: "Incredible adventure! Unforgettable views and a well-organized itinera...", date: "2023-06-20T11:58:15.000+00:00", and __v: 0.

6) **/orders/:userId** dla żądań typu **GET** - zamówienia konkretnego klienta:

a) zrzut kodu:

```
app.get("/orders/:userId", async (req: ..., res: Response<ResBody, Locals>) => {  
  
    try{  
        const user = await UsersModel.findOne({_id: req.params.userId})  
  
        if(!user){  
            res.status(400).send({body: {message: "No such user!"}})  
            return  
        }  
  
        const orders = await OrdersModel.find({userId: user._id})  
        res.status(200).send(orders);  
  
    }catch (err){  
        res.status(500).send({body: {message: "Internal server error"}})  
    }  
  
});
```

b) Opis kodu:

- I. Przyjmujemy dane żądania
- II. Wyszukujemy użytkownika na podstawie id
- III. Jeśli użytkownik nie istnieje wysyłany jest błąd
- IV. Wyszukujemy zamówienia danego użytkownika
- V. Zwracamy odpowiedź serwera

c) Działanie kodu + rezultat - listowanie wcześniejszych zakupionych wycieczek:

Tour Name	Country	Start Date	End Date	No.	Total	Purchase Date	Status
Destination Vacation	Canada	Jan 29, 2024	Feb 8, 2024	4	1996\$	Jun 20, 2023, 12:55:54 PM	await
Destination Vacation	Canada	Jan 29, 2024	Feb 8, 2024	4	1996\$	Jun 20, 2023, 1:31:52 PM	await

7) **/reviews/:tourId** dla żądań typu **GET** - recenzje konkretnej wycieczki

a) zrzut kodu:

```
app.get("/reviews/:tourId", async (req: ..., res: Response<ResBody, Locals>) => {  
  try{  
    const tour = await ToursModel.findOne({_id: req.params.tourId})  
  
    if(!tour){  
      res.status(400).send({body: {message: "No such tour!"}})  
      return  
    }  
  
    const reviews = await ReviewsModel.find({tourId: tour._id})  
  
    const modifiedReviews = reviews.map((review) => ({  
      description: review.description,  
      date: review.date,  
      nickname: review.nickname,  
    }));  
  
    res.status(200).send(modifiedReviews);  
  } catch (err){  
    res.status(500).send({body: {message: "Internal server error"}})  
  }  
});
```

b) Opis kodu

- I. Przyjmujemy dane żądania
- II. Wyszukujemy wycieczkę na podstawie id
- III. Jeśli wycieczka nie istnieje wysyłany jest błąd
- IV. Wyszukujemy recenzje dla danej wycieczki
- V. Tworzymy obiekt modifiedReviews który zawiera recenzje w odpowiednim dla nas formacie
- VI. Zwracamy odpowiedź do serwera

- c) Działanie kodu + rezultat - listowanie dodanych recenzji dla konkretnej wycieczki (*Destination Vacation*):

