Projekt Bazy Danych

Temat: System rezerwacji kortów tenisowych

Technologia: JavaScript(Express)

System zarządzania bazą danych: MongoDB

Imiona i nazwiska autorów: Maciej Brzeżawski Szymon Ciosek

Schemat bazy danych

users

Field	Type
id	Int
Surname	Varchar
Name	Varchar
Credentials	Array(3) [login, password, email]
Reserved	Array(16) [reservationId, hour, fieldid]

dayreservations

Field	Type
id	Int
Date	Data
Reservations	Array(10) [user, hour, fieldid]

tenisfields

Field	Type
id	Int
Type	Varchar
Price	Varchar

Users

Tabela **Users** przechowuje informacje o użytkownikach systemu. Zawiera następujące pola:

- `id`: Unikalny identyfikator użytkownika (typ: Int).
- `Surname`: Nazwisko użytkownika (typ: Varchar).
- `Name`: Imię użytkownika (typ: Varchar).
- `Credentials`: Tablica zawierająca login, hasło i email użytkownika (typ: Array(3)).
- `Reserved`: Tablica zawierająca informacje o rezerwacjach dokonanych przez użytkownika (maksymalnie 16), w tym identyfikator rezerwacji, godzinę oraz identyfikator boiska (typ: Array(16)).

DayReservations

Tabela **DayReservations** przechowuje informacje o rezerwacjach na dany dzień. Zawiera następujące pola:

- `id`: Unikalny identyfikator rezerwacji dnia (typ: Int).
- `Date`: Data rezerwacji (typ: Data).
- `Reservations`: Tablica zawierająca informacje o rezerwacjach, w tym użytkownika, godzinę oraz identyfikator boiska (typ: Array(10)).

Tenisfields

Tabela **Tenisfields** przechowuje informacje o dostępnych kortach tenisowych. Zawiera następujące pola:

- `id`: Unikalny identyfikator kortu (typ: Int).
- `Type`: Typ kortu (np. ziemny, trawiasty, itp.) (typ: Varchar).
- `Price`: Cena za wynajęcie kortu (typ: Varchar).

Implementacja bazy danych w mooongose

Users

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

const userSchema = new mongoose.Schema({
  id: Number,
  surname: String,
  name: String,
  credentials: [
    {
```

```
      login: String,
      password: String,
      email: String,
    },
  ],
  reserved: [
    {
      reservationId: Number,
      hour: String,
      date: String,
      fieldId: Number,
    },
  ],
});

module.exports = mongoose.model("User", userSchema, "users");
```

DayReservations

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

const reservationsSchema = new mongoose.Schema({
  id: Number,
  date: String,
  reservations: [
    {
      userId: Number,
      hour: String,
      fieldid: Number,
    },
  ],
});

module.exports = mongoose.model(
  "DayReservation",
  reservationsSchema,
  "dayreservations"
);
```

Tenisfields

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

const courtSchema = new mongoose.Schema({
  id: Number,
  Type: String,
  Price: String,
});

module.exports = mongoose.model("Court", courtSchema, "tenisfield");
```

Aplikacja serwera

- Importujemy wymagane moduły: Express (framework do tworzenia serwerów), Mongoose (do pracy z MongoDB) i CORS (do obsługi żądań między różnymi domenami).

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

- Tworzymy instancję aplikacji Express, konfigurujemy połączenie z bazą danych MongoDB i używamy middleware do obsługi żądań JSON oraz CORS.

```
const app = express();
const mongoURI = "mongodb://localhost:27017/Tenis";
app.use(express.json());
app.use(cors());
```

- Łączymy się z bazą danych MongoDB, a w przypadku sukcesu wyświetlamy komunikat, jeśli wystąpi błąd, logujemy go.

```
mongoose
  .connect(mongoURI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("Połączono z MongoDB"))
  .catch((err) => console.error("Błąd połączenia z MongoDB:", err));
```

- Definiujemy domyślny endpoint, który zwraca wiadomość powitalną.

```
app.get("/", (req, res) => {
  res.send("Witaj w aplikacji Express z MongoDB!");
});
```

- Ustawiamy port, na którym będzie działać serwer, i uruchamiamy go.

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Serwer działa na porcie ${PORT}`);
});
```

- Importujemy modele Mongoose, które będą używane do interakcji z kolekcjami w MongoDB.

```
const User = require("./models/user");
const Court = require("./models/tenisfield");
const DayReservation = require("./models/dayreservations");
```

Endpoint do pobierania użytkowników Definiujemy endpoint do pobierania wszystkich użytkowników z bazy danych i zwracania ich w formie tabeli HTML.

```
app.get("/Users", async (req, res) => {
  try {
    const users = await User.find();
    let htmlResponse = `<table border="1"><tr><th>ID</th><th>Surname</th><th>Name</th><th>Login</th><th>Email</th><th>Reservations</th></tr>`;
    console.log(users);
    users.forEach((user) => {
      console.log(user);
      htmlResponse += `<tr><td>${user.id}</td><td>${user.surname}</td><td>${
        user.name
      }</td><td>${user.credentials
        .map((l) => l.login)
        .join(", ")}</td><td>${user.credentials
        .map((l) => l.email)
        .join(", ")}</td><td>${user.reserved
        .map((r) => `${r.date} at ${r.hour} on field ${r.fieldId}`)
        .join(", ")}</td></tr>`;
    });
    htmlResponse += `</table>`;
    res.send(htmlResponse);
  } catch (error) {
    console.error("Błąd przy pobieraniu danych:", error);
    res.status(500).send("Nie można pobrać danych");
  }
});
```

Endpoint do pobierania kortów tenisowych Definiujemy endpoint do pobierania wszystkich kortów tenisowych z bazy danych i zwracania ich w formie tabeli HTML.

```
app.get("/tennisfields", async (req, res) => {
  try {
    const courts = await Court.find();
    let htmlResponse = `<h2>Tennis Courts</h2><table border="1"><tr><th>ID</th><th>Type</th><th>Price</th></tr>`;
    courts.forEach((court) => {
      htmlResponse += `<tr><td>${court.id}</td><td>${court.Type}</td><td>${court.Price}</td></tr>`;
    });
    htmlResponse += `</table>`;
    res.send(htmlResponse);
  } catch (error) {
    console.error("Error retrieving tennis field data:", error);
    res.status(500).send("Unable to retrieve tennis field data");
  }
});
```

Endpoint do pobierania rezerwacji dziennych Definiujemy endpoint do pobierania wszystkich rezerwacji dziennych z bazy danych i zwracania ich w formie tabeli HTML.

```
app.get("/dayreservations", async (req, res) => {
  try {
    const reservations = await DayReservation.find();
    let htmlResponse = `<h2>Day Reservations</h2><table border="1"><tr><th>ID</th><th>Date</th><th>Reservations</th></tr>`;
    reservations.forEach((reservation) => {
      const reservationDetails = reservation.reservations
        .map((r) => `${r.hour} for field ${r.fieldid} by user ${r.userId}`)
        .join(", ");
      htmlResponse += `<tr><td>${reservation.id}</td><td>${reservation.date}</td><td>${reservationDetails}</td></tr>`;
    });
    htmlResponse += `</table>`;
    res.send(htmlResponse);
  } catch (error) {
    console.error("Error retrieving day reservation data:", error);
    res.status(500).send("Unable to retrieve day reservation data");
```

```
  }
});
```

Endpoint do dodawania użytkowników Definiujemy endpoint do dodawania nowego użytkownika do bazy danych, sprawdzając najpierw, czy użytkownik już istnieje.

```
app.post("/add", async (req, res) => {
  console.log(req.body);
  const userWithHighestId = await User.findOne()
    .sort({ id: -1 })
    .select("id -_id")
    .limit(1);
  console.log(userWithHighestId);
  console.log(req.body.login);
  try {
    let email = req.body.email;
    let login = req.body.login;
    const existingUser = await User.findOne({ $or: [{ email }, { login }] });

    if (existingUser) {
      return res.status(409).json({ message: "Użytkownik już istnieje" });
    }

    const newUser = new User({
      id: userWithHighestId.id + 1,
      surname: req.body.surname,
      name: req.body.name,
      credentials: [
        {
          login: req.body.login,
          password: req.body.password,
          email: req.body.email,
        },
      ],
      reserved: [],
    });
    await newUser.save();

    res
      .status(201)
      .json({ message: "Użytkownik dodany pomyślnie", user: newUser });
  } catch (error) {
    console.error("Error during user search or creation:", error);
    res.status(500).send({ message: "Internal Server Error" });
  }
});
```

Endpoint do logowania użytkowników Definiujemy endpoint do logowania użytkownika, weryfikując dane logowania.

```
app.post("/log", async (req, res) => {
  const { login, password } = req.body;

  try {
    const user = await User.findOne({
      "credentials.login": login,
      "credentials.password": password,
    });

    if (user) {
      res.status(200).json({ message: "Użytkownik zweryfikowany pomyślnie." });
    } else {
      res.status(401).json({ message: "Nieprawidłowy login lub hasło." });
    }
  } catch (error) {
    console.error("Error during authentication:", error);
    res.status(500).send("Internal Server Error");
  }
});
```

Endpoint do pobierania rezerwacji użytkownika Definiujemy endpoint do pobierania rezerwacji konkretnego użytkownika na podstawie jego loginu.

```
app.post("/reserved", async (req, res) => {
  const { login } = req.body;
  console.log(login);
  try {
    const user = await User.findOne({ "credentials.login": login }, "reserved");

    if (user) {
      res.status(200).json({
        message: "Rezerwacje użytkownika znalezione pomyślnie.",
        reserved: user.reserved,
      });
    } else {
      res
```

```
        .status(404)
        .json({ message: "Nie znaleziono użytkownika o podanym loginie." });
    }
  } catch (error) {
    console.error("Error retrieving user reservations:", error);
    res.status(500).send("Internal Server Error");
  }
});
```

Endpoint do rezerwacji kortu Definiujemy endpoint do rezerwacji kortu, sprawdzając dostępność kortu i aktualizując odpowiednie kolekcje w bazie danych.

```
app.post("/reserve-court", async (req, res) => {
  const { userId, reservationDetails } = req.body;
  try {
    const date = await DayReservation.findOne({
      date: reservationDetails.date,
    });

    const dayReservationsWithHighestId = await DayReservation.findOne()
      .sort({ id: -1 })
      .select("id -_id")
      .limit(1);
    currentId = dayReservationsWithHighestId.id + 1;

    if (!date) {
      const newDayReservation = new DayReservation({
        id: dayReservationsWithHighestId.id + 1,
        date: reservationDetails.date,
        reservations: [
          {
            userId: userId,
            hour: reservationDetails.hour,
            fieldid: reservationDetails.fieldId,
          },
        ],
      });
      await newDayReservation.save();
    } else if (date) {
      if (date.reservations.some((e) => e.hour === reservationDetails.hour)) {
        return res
          .status(404)
          .json({ message: "Istnieje już taka rezerwacja" });
      }
      console.log(reservationDetails);
      date.reservations.push({
        userId: userId,
        hour: reservationDetails.hour,
        fieldid: reservationDetails.fieldId,
      });
      await date.save();
      currentId = date.id;
    }
    const user = await User.findOne({ id: userId });
    if (!user) {
      return res.status(404).json({ message: "Nie znaleziono użytkownika." });
    }

    user.reserved.push({
      reservationId: currentId,
      hour: reservationDetails.hour,
      date: reservationDetails.date,
      fieldId: reservationDetails.fieldId,
    });
    await user.save();

    res.status(200).json({
      message: "Rezerwacja została dodana.",
      reservation: reservationDetails,
    });
  } catch (error) {
    console.error("Error during reservation:", error);
    res.status(500).send("Internal Server Error");
  }
});
```

Endpoint do sprawdzania dostępności kortów Definiujemy endpoint do sprawdzania dostępności kortów na podstawie daty i godziny.

```
app.post("/available-courts", async (req, res) => {
  const { date, hour } = req.body;

  try {
    const reservations = await DayReservation.find({
      date: date,
      "reservations.hour": hour,
    });
```

```
    console.log(reservations);
    const occupiedFields = reservations.flatMap((reservation) =>
      reservation.reservations.map((r) => r.fieldid)
    );
    console.log(occupiedFields);
    const allCourts = await Court.find({});
    console.log(allCourts);

    const availableCourts = allCourts.filter(
      (court) => !occupiedFields.includes(court.id)
    );
    console.log(availableCourts);
    res.status(200).json({
      message: "Dostępne korty na wybraną godzinę i datę:",
      availableCourts,
    });
  } catch (error) {
    console.error("Error fetching available courts:", error);
    res.status(500).send("Internal Server Error");
  }
});
```

Funkcje testujące

Rezerwacja Kortu Funkcja bookCourt wysyła żądanie POST do endpointu /reserve-court, aby zarezerwować kort dla użytkownika na określoną datę i godzinę.

```
async function bookCourt() {
  const url = "http://localhost:3000/reserve-court";

  const data = {
    userId: 2,
    reservationDetails: {
      date: "2024-06-05",
      hour: "10:00",
      fieldId: 2,
    },
  };

  try {
    const response = await fetch(url, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data),
    });

    if (response.ok) {
      const result = await response.json();
      console.log("Sukces:", result);
    } else {
      console.error("Błąd:", response.statusText);
    }
  } catch (error) {
    console.error("Błąd sieci:", error);
  }
}

bookCourt();
```

Sprawdzanie Dostępności Kortów Funkcja checkAvailableCourts wysyła żądanie POST do endpointu /available-courts, aby sprawdzić dostępność kortów na określoną datę i godzinę.

```
async function checkAvailableCourts() {
  const url = "http://localhost:3000/available-courts";
  const data = {
    date: "2024-06-05",
    hour: "10:00",
  };

  try {
    const response = await fetch(url, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data),
    });

    if (response.ok) {
      const result = await response.json();
      console.log("Dostępne korty:", result.availableCourts);
    } else {
      console.error("Błąd podczas odbierania danych:", await response.text());
    }
  } catch (error) {
```

```
    console.error("Błąd sieci:", error);
  }
}

checkAvailableCourts();
```

Pobieranie Rezerwacji Użytkownika Funkcja checkCourtsByLogin wysyła żądanie POST do endpointu /reserved, aby pobrać rezerwacje użytkownika na podstawie loginu.

```
async function checkCourtsByLogin() {
  const url = "http://localhost:3000/reserved";

  const data = {
    login: "jan_kowalski",
  };

  try {
    const response = await fetch(url, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data),
    });

    if (response.ok) {
      const result = await response.json();
      console.log("Sukces:", result);
    } else {
      console.error("Błąd:", response.statusText);
    }
  } catch (error) {
    console.error("Błąd sieci:", error);
  }
}

checkCourtsByLogin();
```

Logowanie Użytkownika Funkcja logUser wysyła żądanie POST do endpointu /log, aby zalogować użytkownika na podstawie podanych danych logowania.

```
async function logUser() {
  const url = "http://localhost:3000/log";

  const data = {
    login: "alamakota",
    password: "wcaleniema",
  };

  try {
    const response = await fetch(url, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data),
    });

    if (response.ok) {
      const result = await response.json();
      console.log("Sukces:", result);
    } else {
      console.error("Błąd:", response.statusText);
    }
  } catch (error) {
    console.error("Błąd sieci:", error);
  }
}

logUser();
```

Dodawanie Nowego Użytkownika Funkcja addUser wysyła żądanie POST do endpointu /add, aby dodać nowego użytkownika do systemu.

```
async function addUser() {
  const url = "http://localhost:3000/add";

  const data = {
    name: "John",
    surname: "Doe",
    email: "john.doe@example.com",
    login: "alamakota",
    password: "wcaleniema",
  };

  try {
```

```
    const response = await fetch(url, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data),
    });

    if (response.ok) {
      const result = await response.json();
      console.log("Sukces:", result);
    } else {
      console.error("Błąd:", response.statusText);
    }
  } catch (error) {
    console.error("Błąd sieci:", error);
  }
}

addUser();
```

Wnioski

1. **Integracja technologii**:
   - **Node.js i Express**: Framework Express okazał się być prostym i wydajnym narzędziem do budowy serwera aplikacji.
   - **MongoDB i Mongoose**: MongoDB, jako baza danych NoSQL, dostarczyła elastyczności w zarządzaniu danymi. Użycie Mongoose jako ODM umożliwiło proste definiowanie schematów danych oraz operacji CRUD, co znacząco usprawniło proces programowania.
2. **Skalowalność i elastyczność**:
   - Dzięki elastycznej strukturze bazy danych MongoDB, aplikacja jest łatwo skalowalna. Dodawanie nowych funkcji, takich jak dodatkowe typy rezerwacji czy zaawansowane zarządzanie użytkownikami, jest możliwe bez znaczących zmian w istniejącej architekturze.
   - Modularność kodu serwera, dzięki zastosowaniu Express, umożliwia łatwe dodawanie nowych endpointów, co ułatwia przyszły rozwój i utrzymanie aplikacji.
3. **Bezpieczeństwo i walidacja danych**:
   - Projekt pokazał jakie znaczenie ma walidacja danych wejściowych oraz zarządzania błędami. Odpowiednie sprawdzanie danych użytkowników podczas logowania i rejestracji, a także obsługa błędów sieciowych, są kluczowe dla zapewnienia bezpieczeństwa i stabilności aplikacji.
4. **Testowanie i debugowanie**:
   - Implementacja funkcji testujących pozwoliła na wczesne wykrycie i naprawienie błędów, co zwiększyło niezawodność aplikacji.