MiniProjekt - Wypożyczalnia

Julia Demitraszek, Natalia Curzytek, Paweł Czajczyk

June 2025

1 Opis projektu

Stworzyliśmy projekt o charakterze wypożyczalni, która ma zarówno **panel dla użytkownika**, gdzie może **wyszukać** i **wypożyczać** produkty oraz **zobaczyć aktualne bestsellery**, czy **napisać opinie** odnośnie produkti, jak i **panel dla administratora**, gdzie administrator może **wypożyczyć produkt** wypozyczony przez klienta, **zwrócić produkt** wypozyczony przez klienta, **zobaczyć listę dłużników**, **usunąć** zużyty lub zniszczony produkt, **zanulować** rezerwacje.

2 Wykorzystane technologie

Aby stworzyć nasz projekt skorzystaliśmy z nierelacyjnej bazy danych **MongoDB**, gdzie dane są przechowywane w **MongoDB Atlas**, aby móc połączyć sie z nimi zdalnie, oraz backendowe oprogramowanie **Express**, a w celu wytworzenia frontendu użyliśmy oprogramowania **React**.

3 Model bazy danych

3.1 Kolekcja produkt

Pole	Typ	Wymagan	Opis/Ograniczenia
title	String	Tak	Tytuł produktu, usuwane białe znaki
description	String	Tak	Opis produktu, usuwane białe znaki
category	String	Tak	Kategoria produktu, usuwane białe znaki
type	String	Tak	Typ: ’movie’, ’audiobook’, ’book’, ’music’, ’game’ (domyślnie: ’movie’)
status	String	Tak	Status: ’available’, ’rented’, ’reserved’, ’damaged’, ’maintenance’ (domyślnie: ’available’)
condition	String	Tak	Stan: ’new’, ’good’, ’fair’, ’poor’, ’damaged’ (domyślnie: ’new’)
reviews	Array	Nie	Tablica recenzji (patrz struktura poniżej)
createdAt	Date	Nie	Data utworzenia (automatyczne)
updatedAt	Date	Nie	Data aktualizacji (automatyczne)

3.2 Struktura recencji

Pole	Typ	Wymagan	Opis/Ograniczenia
clientId	ObjectId	Tak	Referencja do kolekcji ’Client’
rating	Number	Tak	Ocena od 1 do 5
comment	String	Nie	Komentarz do recenzji, usuwane białe znaki
date	Date	Nie	Data recenzji (domyślnie: bieżąca data)

3.3 Kolekcja historii wypożyczeń

Pole	Typ	Wymagan	Opis/Ograniczenia
product	ObjectId	Tak	Referencja do kolekcji 'Product'
client	ObjectId	Tak	Referencja do kolekcji 'Client'
worker	ObjectId	Tak	Referencja do kolekcji 'Worker'
rentalPeriod	Object	Tak	Okres wypożyczenia (patrz struktura poniżej)
status	String	Tak	Status: 'rented', 'returned', 'overdue', 'damaged', 'lost', 'cancelled' (domyślnie: 'rented')
notes	String	Nie	Notatki, usuwane białe znaki
conditionBefore	String	Tak	Stan przed wypożyczeniem: 'new', 'good', 'fair', 'poor'
conditionAfter	String	Nie	Stan po zwrocie: 'new', 'good', 'fair', 'poor', 'damaged'
createdAt	Date	Nie	Data utworzenia (automatyczne)
updatedAt	Date	Nie	Data aktualizacji (automatyczne)

3.4 Struktura okresu wypożyczenia

Pole	Typ	Wymagan	Opis/Ograniczenia
start	Date	Tak	Data rozpoczęcia wypożyczenia
end	Date	Tak	Planowana data zwrotu
returned	Date	Nie	Rzeczywista data zwrotu (domyślnie: null)

3.5 Kolekcja klienta

Pole	Typ	Wymagan	Opis/Ograniczenia
name	String	Tak	Imię i nazwisko klienta, usuwane białe znaki
email	String	Tak	Adres email (unikalny), usuwane białe znaki
phone	String	Tak	Numer telefonu, usuwane białe znaki
address	Object	Nie	Adres klienta (patrz struktura poniżej)
rank	String	Tak	Ranking: 'bronze', 'silver', 'gold' (domyślnie: 'bronze')
createdAt	Date	Nie	Data utworzenia (automatyczne)
updatedAt	Date	Nie	Data aktualizacji (automatyczne)

3.6 Struktura adresu

Pole	Typ	Wymagan	Opis/Ograniczenia
street	String	Nie	Ulica i numer, usuwane białe znaki
city	String	Nie	Miasto, usuwane białe znaki

3.7 Kolekcja pracownika

Pole	Typ	Wymagan	Opis/Ograniczenia
name	String	Tak	Imię i nazwisko pracownika, usuwane białe znaki
email	String	Tak	Adres email (unikalny), usuwane białe znaki
phone	String	Tak	Numer telefonu, usuwane białe znaki
position	String	Tak	Stanowisko: 'manager', 'casual' (domyślnie: 'casual')
createdAt	Date	Nie	Data utworzenia (automatyczne)
updatedAt	Date	Nie	Data aktualizacji (automatyczne)

4 Model bazy danych - kod

4.1 Kolekcja product

```
const productSchema = new mongoose.Schema({
title: {
    type: String,
    required: true,
    trim: true
},
description: {
    type: String,
    required: true,
    trim: true
},
category: {
    type: String,
    required: true,
    trim: true
},
type: {
    type: String,
    required: true,
    enum: ['movie', 'audiobook', 'book', 'music', 'game'],
    default: 'movie'
},
status: {
    type: String,
    required: true,
    enum: ['available', 'rented', 'reserved', 'damaged', 'maintenance'],
    default: 'available'
},
condition: {
    type: String,
    required: true,
    enum: ['new', 'good', 'fair', 'poor', 'damaged'],
    default: 'new'
```

```
    },
    reviews: [{
        clientId: {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Client',
            required: true
        },
        rating: {
            type: Number,
            required: true,
            min: 1,
            max: 5
        },
        comment: {
            type: String,
            trim: true
        },
        date: {
            type: Date,
            default: Date.now
        }
    }],
}, {timestamps: true})
```

4.2 Kolekcja rentalHistory (historia wypożyczeń)

```
    const rentalHistorySchema = new mongoose.Schema({
    product: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Product',
        required: true
    },
    client: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Client',
        required: true
    },
    worker: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Worker',
        required: true
    },
    rentalPeriod: {
        start: {
            type: Date,
            required: true
        },
        end: {
            type: Date,
            required: true
        },
```

```
        returned: {
            type: Date,
            default: null
        }
    },
    status: {
        type: String,
        required: true,
        enum: ['rented', 'returned', 'overdue', 'damaged', 'lost', 'cancelled'],
        default: 'rented'
    },
    notes: {
        type: String,
        trim: true
    },
    conditionBefore: {
        type: String,
        enum: ['new', 'good', 'fair', 'poor'],
        required: true
    },
    conditionAfter: {
        type: String,
        enum: ['new', 'good', 'fair', 'poor', 'damaged'],
        default: null
    }
}, {timestamps: true});
```

4.3 Kolekcja client

```
const clientSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        trim: true
    },
    email: {
        type: String,
        required: true,
        unique: true,
        trim: true
    },
    phone: {
        type: String,
        required: true,
        trim: true
    },
    address: {
        street: {
            type: String,
            trim: true
        },
```

```
        city: {
            type: String,
            trim: true
        },
    },
    rank: {
        type: String,
        required: true,
        enum: ['bronze', 'silver', 'gold'],
        default: 'bronze'
    },

}, {timestamps: true});
```

4.4 Kolekcja worker

```
const workerSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        trim: true
    },
    email: {
        type: String,
        required: true,
        unique: true,
        trim: true
    },
    phone: {
        type: String,
        required: true,
        trim: true
    },
    position: {
        type: String,
        required: true,
        enum: ['manager', 'casual'], // manager moze np dodawac produkty i przegladac zamowienia
        default: 'casual'
    }
}, {timestamps: true});
```

5 Operacje

5.1 Operacje CRUD

5.1.1 CREATE (POST)

Metoda createClient()

```
    // Tworzenie nowego klienta
const createClient = async (req, res) => {
```

```
    console.log("hello: ", req.body);
    const { name, email, phone, address, rank } = req.body;

    // Walidacja podstawowych p l
    if (!name || !phone || !email) {
        return res.status(400).json({ message: 'Imie, email i telefon  s  wymagane' });
    }

    try {
        // Sprawdzenie czy klient z tym adresem email ju  istnieje
        const existingClient = await Client.findOne({ email });
        if (existingClient) {
            return res.status(400).json({ message: 'Klient z tym adresem email juz istnieje' });
        }

        const clientData = {
            name,
            email,
            phone,
        };

        if (address) clientData.address = address;
        if (rank) clientData.rank = rank;

        const newClient = new Client(clientData);

        await newClient.save();
        res.status(201).json(newClient);
    } catch (error) {
        res.status(500).json({ message: 'Basd podczas tworzenia klienta', error: error.message });
    }
};
```

Metoda createProduct()

```
const createProduct = async (req, res) => {
    console.log(req.body)

    const { title , description , category, type,
        status = "available", condition = "new" } = req.body;

    if (!title || !description || !category || !type) {
        return res.status(400).json({ message: 'Wymagane pola: tytu , opis, kategoria i typ' });
    }

    try {
        const newProduct = new Product({
            title ,
            description ,
            category,
            type,
            status ,
            condition ,
```

```
            reviews: []
        });

        await newProduct.save();
        res.status(201).json(newProduct);
    } catch (error) {
        res.status(500).json({message: 'Error creating product', error: error.message});
    }
}
```

Metoda addProductReview()

```
    // Dodawanie opinii dla produktu
const addProductReview = async (req, res) => {
    const { id } = req.params;
    const { username, rating, comment } = req.body;


    // Sprawdzenie poprawnosci danych wesciowych
    if (!username || !rating) {
        return res.status(400).json({ message: 'Wymagane pola: klient (ID lub email) i ocena' });
    }

    // Sprawdzenie czy ocena jest prawidlowa (zakres 1-5)
    if (rating < 1 || rating > 5) {
        return res.status(400).json({ message: 'Ocena musi by  w zakresie od 1 do 5' });
    }

    // Rozpocz cie transakcji
    const session = await mongoose.startSession();
    session.startTransaction();

    try {
        // Identyfikacja klienta na podstawie ID lub adresu email
        let clientDoc;

        if (mongoose.Types.ObjectId.isValid(username)) {
            clientDoc = await Client.findById(username).session(session);
        } else {
            clientDoc = await Client.findOne({ email: username }).session(session);
        }

        if (!clientDoc) {
            await session.abortTransaction();
            session.endSession();
            return res.status(404).json({ message: 'Nie znaleziono klienta' });
        }

        const clientId = clientDoc._id;

        // Sprawdzamy, czy produkt istnieje
        const product = await Product.findById(id).session(session);
```

```
        if (!product) {
            await session.abortTransaction();
            session.endSession();
            return res.status(404).json({ message: 'Produkt nie zosta znaleziony' });
        }

        const newReview = {
            clientId: clientId,
            rating,
            comment: comment || '',
        };
        product.reviews.push(newReview);

        await product.save({ session });

        // Zatwierdzenie transakcji
        await session.commitTransaction();
        session.endSession();

        res.status(201).json({
            message: 'Opinia zosta a dodana pomy lnie',
            review: newReview
        });
    } catch (error) {
        // W przypadku b   du wycofujemy transakcj
        await session.abortTransaction();
        session.endSession();
        res.status(500).json({ message: 'B   d podczas dodawania opinii', error: error.message });
    }
}
```

Metoda rentProduct() - zwrot produktu

```
const rentProduct = async (req, res) => {
const {productId, client, worker, rentalTime = 14} = req.body;

if (!productId || !client || !worker) {
    return res.status(400).json({message: 'Wszystkie pola s  wymagane'});
}

// transakcja
const session = await mongoose.startSession();
session.startTransaction();

try {
    // Znajd  klienta po ID lub email
    let clientDoc;
    if (mongoose.Types.ObjectId.isValid(client)) {
        clientDoc = await Client.findById(client).session(session);
    } else {
        clientDoc = await Client.findOne({ email: client }).session(session);
    }
```

```
if (!clientDoc) {
    await session.abortTransaction();
    session.endSession();
    return res.status(404).json({message: 'Nie znaleziono klienta'});
}

// Znajd  pracownika po ID lub email
let workerDoc;
if (mongoose.Types.ObjectId.isValid(worker)) {
    workerDoc = await Worker.findById(worker).session(session);
} else {
    workerDoc = await Worker.findOne({ email: worker }).session(session);
}

if (!workerDoc) {
    await session.abortTransaction();
    session.endSession();
    return res.status(404).json({message: 'Nie znaleziono pracownika'});
}

// upewiamy sie ze produkt istnieje i jest dost pny i od razu go zaklepujemy
const product = await Product.findOneAndUpdate(
    {
        _id: productId,
        status: 'available'
    },
    {
        status: 'rented'
    },
    {
        new: true,
        session,
        runValidators: true
    }
);

if (!product) {
    await session.abortTransaction();
    session.endSession();
    return res.status(400).json({message: 'Produkt nie jest dost pny do wypozyczenia'});
}

const conditionBefore = product.condition;

const rentalPeriod = {
    start: new Date(),
    end: new Date(new Date().setDate(new Date().getDate() + rentalTime))
};

const rentalHistory = new RentalHistory({
    product: productId,
```

```
            client: clientDoc._id,
            worker: workerDoc._id,
            rentalPeriod,
            status: 'rented',
            conditionBefore: conditionBefore,
            notes: req.body.notes || ''
        });

        await rentalHistory.save({session});

        // Zatwierdzenie transakcji
        await session.commitTransaction();
        session.endSession();

        res.status(201).json({
            message: 'Produkt zosta pomyslnie wypozyczony',
            rental: rentalHistory
        });
    } catch (error) {
        await session.abortTransaction();
        session.endSession();
        res.status(500).json({message: 'Blad podczas wypozyczania produktu', error: error.message});
    }
};
```

Metoda returnProduct()

```
    // zwrot produktu
const returnProduct = async (req, res) => {
    const { rentalId } = req.params;
    let { conditionAfter } = req.body;

    const session = await mongoose.startSession();
    session.startTransaction();

    try {
        const rental = await RentalHistory.findOne({
            _id: rentalId,
            status: 'rented'
        }).session(session);

        if (!rental) {
            await session.abortTransaction();
            session.endSession();
            return res.status(404).json({message: 'Nie znaleziono aktywnego
                wypozyczenia o takim ID'});
        }

        const productId = rental.product;
        conditionAfter = conditionAfter || rental.conditionBefore;

        await Product.findByIdAndUpdate(
            productId,
```

```
            { status: 'available', condition: conditionAfter },
            { new: true, session }
        );

        rental.status = 'returned';
        rental.conditionAfter = conditionAfter;
        rental.rentalPeriod.returned = new Date();
        await rental.save({session});

        await session.commitTransaction();
        session.endSession();

        return res.status(200).json({
            message: 'Produkt zostal pomyslnie zwrocony',
            rental: rental
        });
    } catch (error) {
        await session.abortTransaction();
        session.endSession();
        return res.status(500).json({message: 'Blad podczas zwrotu produktu', error: error.message});
    }
};
```

Metoda cancelRental()

```
    // Anulowanie wypozyczenia
const cancelRental = async (req, res) => {
    const {rentalId} = req.params;
    const {cancelReason} = req.body;

    const session = await mongoose.startSession();
    session.startTransaction();

    try {
        const rental = await RentalHistory.findById(rentalId).session(session);

        if (!rental || rental.status !== 'rented') {
            await session.abortTransaction();
            session.endSession();
            return res.status(400).json({message: 'Wypozyczenie nie
                istnieje lub nie mozna go anulowac'});
        }

        // Aktualizuj status produktu
        await Product.findByIdAndUpdate(
            rental.product,
            {status: 'available'},
            {session}
        );

        // Aktualizuj wypozyczenie
        rental.status = 'cancelled';
        rental.notes =
```

```
            rental.notes ? `${rental.notes}\nAnulowano: ${cancelReason}`
            : `Anulowano: ${cancelReason}`;
        rental.rentalPeriod.returned = new Date();
        await rental.save({session});

        await session.commitTransaction();
        session.endSession();

        res.status(200).json({
            message: 'Wypozyczenie zostało anulowane',
            rental
        });
    } catch (error) {
        await session.abortTransaction();
        session.endSession();
        res.status(500).json({message: 'Blad podczas anulowania wypożyczenia'
            , error: error.message});
    }
};
```

Metoda reportDamagedProduct()

```
    // Zgłoszenie uszkodzonego produktu
const reportDamagedProduct = async (req, res) => {
    const {rentalId} = req.params;
    const {damageDescription} = req.body;

    const session = await mongoose.startSession();
    session.startTransaction();

    try {
        const rental = await RentalHistory.findById(rentalId).session(session);

        if (!rental || rental.status !== 'rented') {
            await session.abortTransaction();
            session.endSession();
            return res.status(400).json({message: 'Wypozyczenie nie
                istnieje lub produkt nie jest wypozyczony'});
        }

        // Aktualizuj status produktu
        await Product.findByIdAndUpdate(
            rental.product,
            {
                status: 'damaged',
                condition: 'damaged'
            },
            {session}
        );

        // Aktualizuj wypożyczenie
        rental.status = 'damaged';
        rental.conditionAfter = 'damaged';
```

```
            rental.notes =
                rental.notes ?
                `${rental.notes}\nUszkodzenie: ${damageDescription}`
                : `Uszkodzenie: ${damageDescription}`;
            rental.rentalPeriod.returned = new Date();
            await rental.save({session});

            await session.commitTransaction();
            session.endSession();

            res.status(200).json({
                message: 'Zg oszenie uszkodzenia zosta o zapisane',
                rental
            });
        } catch (error) {
            await session.abortTransaction();
            session.endSession();
            res.status(500).json({message: 'Blad podczas zglaszania
            uszkodzenia', error: error.message});
        }
};
```

5.1.2 READ (GET)

Metoda getAllClients() - pokazuje id wszystkich klientow

```
    // Pobieranie wszystkich klient w
const getAllClients = async (req, res) => {
    const { limit = 10, page = 1, search } = req.query;
    const query = {};


    // Dodanie wyszukiwania po imieniu lub nazwisku
    if (search) {
        query.$or = [
            { name: { $regex: `^${search}\\s[a-zA-Z]*`, $options: 'i' } },
            { name: { $regex: `[a-zA-Z]*\\s${search}$`, $options: 'i' } }
        ];
    }

    try {
        const clients = await Client.find(query)
            .skip(page > 0 ? (page - 1) * limit : 0)
            .limit(parseInt(limit))
            .sort({ name: 1 });

        console.log(clients);

        const total = await Client.countDocuments(query);

        res.status(200).json({
            clients,
```

```
            total,
            currentPage: parseInt(page),
            totalPages: Math.ceil(total / limit)
        });
    } catch (error) {
        res.status(500).json({ message: 'Blad podczas pobierania klient w', error: error.message })
    }
};
```

Metoda getClientById() - pokazuje id wybranego klienta

```
    // Pobieranie klienta po ID
const getClientById = async (req, res) => {
    const { id } = req.params;
    // console.log(id);
    try {
        const client = await Client.findById(id);

        if (!client) {
            return res.status(404).json({ message: 'Klient nie zostal znaleziony' });
        }
        console.log(client);
        res.status(200).json(client);
    } catch (error) {
        res.status(500).json({ message: 'Blad podczas pobierania klienta', error: error.message });
    }
};
```

Metoda getClientRentalHistory() - pokazuje historie wypożyczeń klienta

```
    // Historia wypozyczen klienta
const getClientRentalHistory = async (req, res) => {
    const { id } = req.params;

    try {
        // Sprawdzenie czy klient istnieje
        const client = await Client.findById(id);
        if (!client) {
            return res.status(404).json({ message: 'Klient nie zosta znaleziony' });
        }

        const rentalHistory = await RentalHistory.find({ client: id })
            .populate('product')
            .populate('worker')
            .sort({ 'rentalPeriod.start': -1 });

        res.status(200).json(rentalHistory);
    } catch (error) {
        res.status(500).json({ message: 'Blad podczas pobierania
        historii wypozyczen', error: error.message });
    }
};
```

Metoda getDebtors() - pokazuje wszystkich dłużników

```
    // Pobieranie listy dluznikow (klienci z przeterminowanymi wypozyczeniami)
const getDebtors = async (req, res) => {

    const { limit = 10, page = 1 } = req.query;
    const currentDate = new Date();

    console.log('getDebtors function called');
    console.log('req.params:', req.params);
    console.log('req.query:', req.query);
    console.log('req.query.limit:', req.query.limit);
    try {
        // Znajduje wszystkie przeterminowane wypozyczenia
        const overdueRentals = await RentalHistory.find({
            status: 'rented',
            'rentalPeriod.end': { $lt: currentDate }
        })
            .populate({
                path: 'client',
                select: 'firstName lastName email phone address'
            })
            .populate({
                path: 'product',
                select: 'title category type'
            })
            .skip(page > 0 ? (page - 1) * limit : 0)
            .limit(parseInt(limit))
            .sort({ 'rentalPeriod.end': 1 });

        // Grupowanie przeterminowanych wypozyczen wedlug klientow
        const debtorsMap = {};
        overdueRentals.forEach(rental => {
            const clientId = rental.client._id.toString();

            if (!debtorsMap[clientId]) {
                debtorsMap[clientId] = {
                    client: rental.client,
                    overdueItems: []
                };
            }

            debtorsMap[clientId].overdueItems.push({
                product: rental.product,
                rentalId: rental._id,
                dueDate: rental.rentalPeriod.end,
                daysOverdue:
                Math.floor((currentDate - rental.rentalPeriod.end) / (1000 * 60 * 60 * 24))
            });
        });

        // Konwersja mapy na tablic
        const debtorsList = Object.values(debtorsMap);
```

```
        // Obliczenie ca kowitej liczby d u nik w
        const totalDebtors = await RentalHistory.aggregate([
            { $match: { status: 'rented', 'rentalPeriod.end': { $lt: currentDate } } },
            { $group: { _id: '$client' } },
            { $count: 'total' }
        ]);

        const total = totalDebtors.length > 0 ? totalDebtors[0].total : 0;

        res.status(200).json({
            debtors: debtorsList,
            total,
            currentPage: parseInt(page),
            totalPages: Math.ceil(total / limit)
        });
    } catch (error) {
        res.status(500).json({ message: 'Blad podczas pobierania
        listy dluznikow', error: error.message });
    }
};
```

Metoda getAllProducts() - pokazuje wszystkie produkty

```
const getAllProducts = async (req, res) => {
const {limit = 10, page = 1, title} = req.query;
const query = {};

if (title) {
    query.title = {$regex: title , $options: 'i'};
}

try {
    const products = await Product.find(query).skip(page > 0 ? (page - 1) * limit : 0)
    .limit(parseInt(limit))
    .sort({createdAt: -1});
    res.status(200).json(products);
} catch (error) {
    res.status(500).json({message: 'Error fetching products', error: error.message});
}
}
```

Metoda getMostPopularProducts() - pokazuje wszystkie najpopularniejsze produkty

```
const getMostPopularProducts = async (req, res) => {
const {limit = 3} = req.query;

try {
    const popularProducts = await RentalHistory.aggregate([
        {$match: {status: 'rented'}},
        {$group: {_id: "$product", count: {$sum: 1}}},
        {$sort: {count: -1}},
        {$limit: parseInt(limit)},
        {
```

```
                $lookup: {
                    from: 'products',
                    localField: '_id',
                    foreignField: '_id',
                    as: 'productDetails'
                }
            },
            {$unwind: '$productDetails'},
            {$replaceRoot: {newRoot: '$productDetails'}}
        ]);

        res.status(200).json(popularProducts);
    } catch (error) {
        res.status(500).json({message: 'Error fetching popular products', error: error.message});
    }

}
```

Metoda getProductById()

```
const getProductById = async (req, res) => {
const {id} = req.params;
try {
    const product = await Product.findById(id);
    if (!product) {
        return res.status(404).json({message: 'Product not found'});
    }
    res.status(200).json(product);
} catch (error) {
    res.status(500).json({message: 'Error fetching product', error: error.message});
}
```

Metoda getProductByCategory()

```
const getProductsByCategory = async (req, res) => {
const { category } = req.params;
const { limit = 10, page = 1 } = req.query;

try {
    const products = await Product.find({ category })
        .skip(page > 0 ? (page - 1) * limit : 0)
        .limit(parseInt(limit))
        .sort({ createdAt: -1 });

    const total = await Product.countDocuments({ category });

    res.status(200).json({
        products,
        total,
        currentPage: parseInt(page),
        totalPages: Math.ceil(total / limit)
    });
} catch (error) {
    res.status(500).json({ message: 'Blad podczas
```

```
␣␣␣␣␣␣␣␣␣␣␣␣pobierania␣produktow␣wedlug␣kategorii', error: error.message });
    }
}
```

Metoda getProductByStatus()

```
const getProductsByStatus = async (req, res) => {
const { status } = req.params;
const { limit = 10, page = 1 } = req.query;

try {
    const products = await Product.find({ status })
        .skip(page > 0 ? (page - 1) * limit : 0)
        .limit(parseInt(limit))
        .sort({ createdAt: -1 });

    const total = await Product.countDocuments({ status });

    res.status(200).json({
        products,
        total,
        currentPage: parseInt(page),
        totalPages: Math.ceil(total / limit)
    });
} catch (error) {
    res.status(500).json({ message: 'Blad␣podczas
␣␣␣␣␣␣␣␣␣␣␣␣pobierania␣produktow␣wedlug␣statusu', error: error.message });
    }
}
```

Metoda getAllRents()

```
const getAllRents = async (req, res) => {
const {limit = 10, page = 1} = req.query;
try {
    const rents = await RentalHistory.find({})
        .populate({ path: 'product', select: 'title' })
        .populate({ path: 'client', select: 'name' })
        .populate({ path: 'worker', select: 'name' })
        .skip(page > 0 ? (page - 1) * limit : 0)
        .limit(parseInt(limit))
        .sort({createdAt: -1});
    res.status(200).json(rents);
} catch (error) {
    res.status(500).json({message: 'Blad␣podczas
␣␣␣␣␣␣␣␣␣␣␣␣pobierania␣wypozyczen', error: error.message});
    }
}
```

Metoda getRentalsById()

```
const getRentalById = async (req, res) => {
const {id} = req.params;

try {
```

```
        const rental = await RentalHistory.findById(id)
            .populate('product')
            .populate('client')
            .populate('worker');

        if (!rental) {
            return res.status(404).json({message: 'Wypozyczenie nie zostalo znalezione'});
        }

        res.status(200).json(rental);
    } catch (error) {
        res.status(500).json({message: 'Blad podczas pobierania
            wypozyczenia', error: error.message});
    }
};
```

Metoda getActiveRents() - zwraca aktualne wypożyczenia

```
const getActiveRents = async (req, res) => {
const {limit = 10, page = 1} = req.query;

try {
    const activeRents = await RentalHistory.find({status: 'rented'})
        .populate('product client worker')
        .skip(page > 0 ? (page - 1) * limit : 0)
        .limit(parseInt(limit))
        .sort({'rentalPeriod.end': 1});

    const total = await RentalHistory.countDocuments({status: 'rented'});

    res.status(200).json({
        rents: activeRents,
        total,
        currentPage: parseInt(page),
        totalPages: Math.ceil(total / limit)
    });
} catch (error) {
    res.status(500).json({message: 'Blad podczas
        pobierania aktywnych wypozyczen', error: error.message});
}
};
```

Metoda getOverdueRents() - zwraca te wypożyczenie które są zaległe

```
const getOverdueRents = async (req, res) => {
const {limit = 10, page = 1} = req.query;
const currentDate = new Date();

try {
    const overdueRents = await RentalHistory.find({
        status: 'rented',
        'rentalPeriod.end': {$lt: currentDate}
    })
        .populate('product client worker')
```

```
            .skip(page > 0 ? (page - 1) * limit : 0)
            .limit(parseInt(limit))
            .sort({'rentalPeriod.end': 1});

        const total = await RentalHistory.countDocuments({
            status: 'rented',
            'rentalPeriod.end': {$lt: currentDate}
        });

        res.status(200).json({
            rents: overdueRents,
            total,
            currentPage: parseInt(page),
            totalPages: Math.ceil(total / limit)
        });
    } catch (error) {
        res.status(500).json({message: 'Blad podczas pobierania
            zleglych wypozyczen', error: error.message});
    }
};
```

Metoda getClientRentals() - zwraca wypożyczenia konkretnego klienta

```
const getClientRentals = async (req, res) => {
const {clientId} = req.params;
const {limit = 10, page = 1, status} = req.query;

const query = {client: clientId};

if (status) {
    query.status = status;
}

try {
    const rentals = await RentalHistory.find(query)
        .populate('product worker')
        .skip(page > 0 ? (page - 1) * limit : 0)
        .limit(parseInt(limit))
        .sort({createdAt: -1});

    const total = await RentalHistory.countDocuments(query);

    res.status(200).json({
        rentals,
        total,
        currentPage: parseInt(page),
        totalPages: Math.ceil(total / limit)
    });
} catch (error) {
    res.status(500).json({message: 'Blad podczas pobierania
        wypozyczen', error: error.message});
}
};
```

5.1.3 UPDATE (PUT)

Metoda updateClient()

```
    const updateClient = async (req, res) => {
    const { id } = req.params;
    const { name, email, phone, address } = req.body;

    try {
        // Sprawdzenie czy klient istnieje
        const client = await Client.findById(id);
        if (!client) {
            return res.status(404).json({ message: 'Klient nie zostal znaleziony' });
        }

        // Sprawdzenie czy nowy email jest ju  uzywany przez innego klienta
        if (email && email !== client.email) {
            const existingClient = await Client.findOne({ email });
            if (existingClient) {
                return res.status(400).json({ message: 'Klient z tym adresem email juz istnieje' });
            }
        }

        const updatedClient = await Client.findByIdAndUpdate(
            id,
            { name, email, phone, address },
            { new: true, runValidators: true }
        );

        res.status(200).json(updatedClient);
    } catch (error) {
        res.status(500).json({ message: 'Blad podczas
            aktualizacji klienta', error: error.message });
    }
};
```

Metoda updateProduct()

```
    const updateProduct = async (req, res) => {
    const { id } = req.params;
    const { title, description, category, type, status, condition } = req.body;

    try {
        const updatedProduct = await Product.findByIdAndUpdate(id, {
            title,
            description,
            category,
            type,
            status,
            condition
        }, { new: true, runValidators: true });

        if (!updatedProduct) {
            return res.status(404).json({ message: 'Produkt nie zosta znaleziony' });
```

```
        }

        res.status(200).json(updatedProduct);
    } catch (error) {
        res.status(500).json({ message: 'Bladd podczas aktualizacji produktu', error: error.message
    }
}
```

5.1.4 DELETE

Metoda deleteClient()

```
    const deleteClient = async (req, res) => {
    const { id } = req.params;

    try {
        const deletedClient = await Client.findByIdAndDelete(id);

        if (!deletedClient) {
            return res.status(404).json({ message: 'Klient nie zosta znaleziony' });
        }

        res.status(200).json({ message: 'Klient zostal pomyslnie usuniety', id });
    } catch (error) {
        res.status(500).json({ message: 'Blad podczas usuwania klienta', error: error.message });
    }
};
```

Metoda deleteProduct()

```
const deleteProduct = async (req, res) => {
    const { id } = req.params;

    try {
        const deletedProduct = await Product.findByIdAndDelete(id);

        if (!deletedProduct) {
            return res.status(404).json({ message: 'Produkt nie zostal znaleziony' });
        }

        res.status(200).json({ message: 'Produkt zosta pomyslnie usuniety', id });
    } catch (error) {
        res.status(500).json({ message: 'Blad podczas usuwania produktu', error: error.message });
    }
}
```

5.2 Operacje wymagające zużycia transakcji

1. addProductReview() - dodaje opinie do produktu
2. returnProduct() - zwrot produktu
3. rentProduct() - wypozyczenie produktu

4. reportDamagedProduct() - zgloszenie uszkodzonego produktu
5. cancelRental() - zanulowanie wypożyczenia

5.3 Operacje o charakterze raportującym

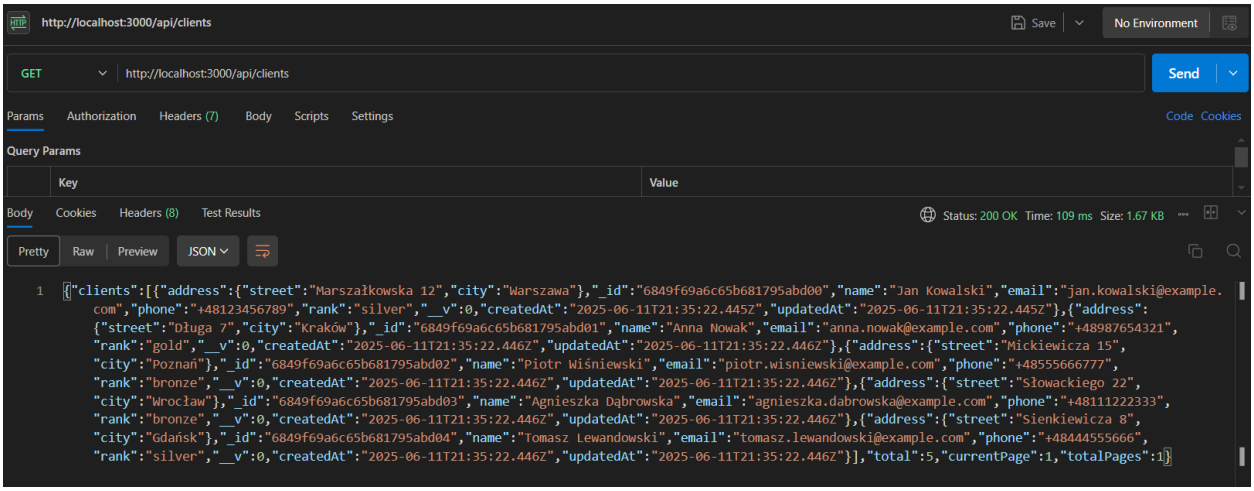
1. getClientRentalHistory() - historia wypożyczeń klienta
2. getDebtors() - lista aktualnych dłużników
3. getMostPopularProducts() - pokazuje bestsellery

6 Testy

6.1 Metody klientów:

6.1.1 Metoda GET getAllClients()

Wszyscy klienci:



Dane w **Mongodb Atlas**

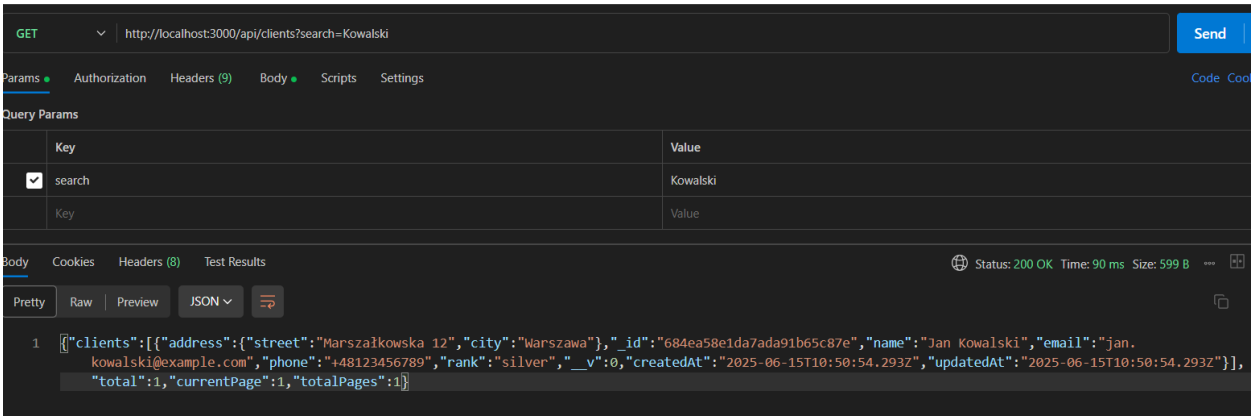
```
_id: ObjectId('6849f69a6c65b681795abd03')
name : "Agnieszka Dąbrowska"
email : "agnieszka.dabrowska@example.com"
phone : "+48111222333"
address : Object
rank : "bronze"
__v : 0
createdAt : 2025-06-11T21:35:22.446+00:00
updatedAt : 2025-06-11T21:35:22.446+00:00
```

```
_id: ObjectId('6849f69a6c65b681795abd04')
name : "Tomasz Lewandowski"
email : "tomasz.lewandowski@example.com"
phone : "+48444555666"
address : Object
rank : "silver"
__v : 0
createdAt : 2025-06-11T21:35:22.446+00:00
updatedAt : 2025-06-11T21:35:22.446+00:00
```

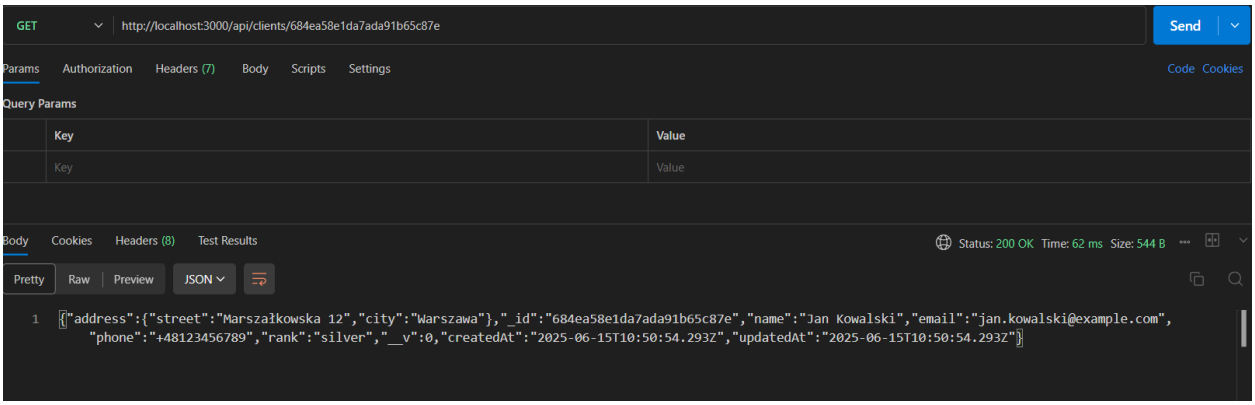
```
_id: ObjectId('6849f69a6c65b681795abd02')
name : "Piotr Wiśniewski"
email : "piotr.wisniewski@example.com"
phone : "+48555666777"
address : Object
rank : "bronze"
__v : 0
createdAt : 2025-06-11T21:35:22.446+00:00
updatedAt : 2025-06-11T21:35:22.446+00:00
```


Klient po imieniu:

Klient po nazwisku:



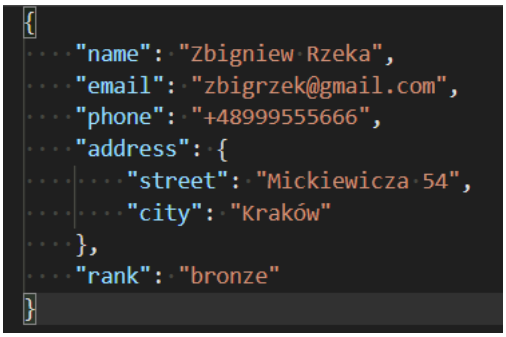
6.1.2 Metoda GET getClientById()

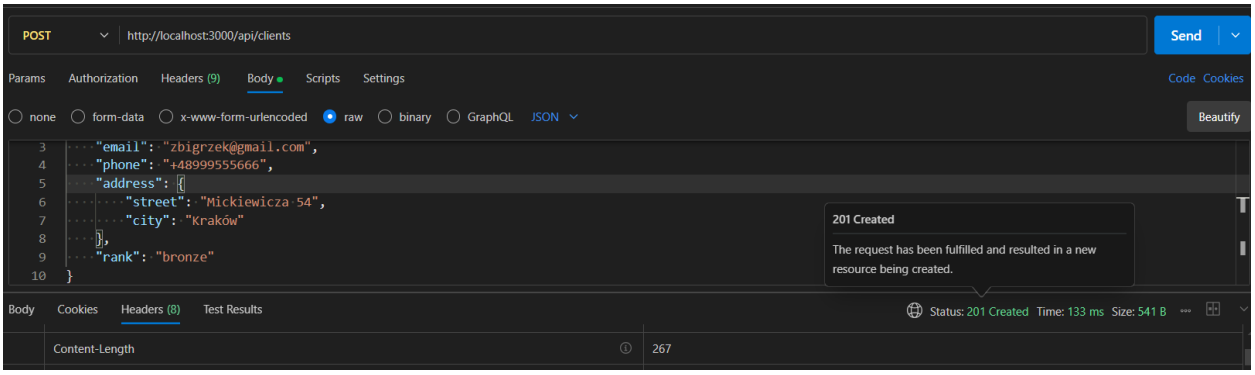


Rysunek 3: Metoda getClientById() zwraca prawidlowy wynik

6.1.3 Metoda POST createClient()

Dane które próbujemy wpisać:



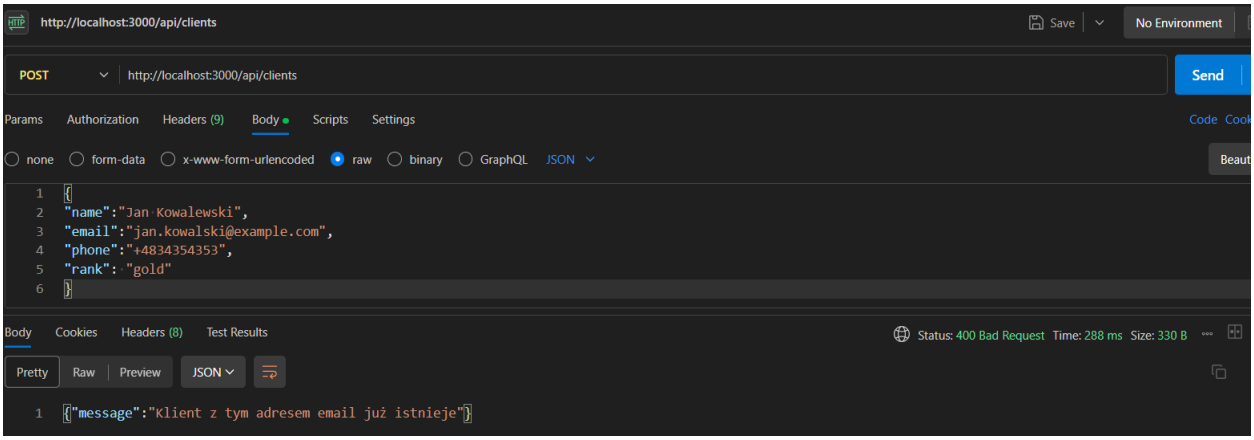


Rysunek 4: Metoda createClient() prawidłowo tworzy klienta

```
_id: ObjectId('684d7de6e5a2bdc7d56152a6')
name : "Zbigniew Rzeka"
email : "zbigrzek@gmail.com"
phone : "+48999555666"
address : Object
rank : "bronze"
createdAt : 2025-06-14T13:49:26.788+00:00
updatedAt : 2025-06-14T13:49:26.788+00:00
__v : 0
```

Rysunek 5: Nowy rekord w **MongoDB Atlas**

Jeżeli klient już istnieje:



6.1.4 Metoda PUT updateClient()

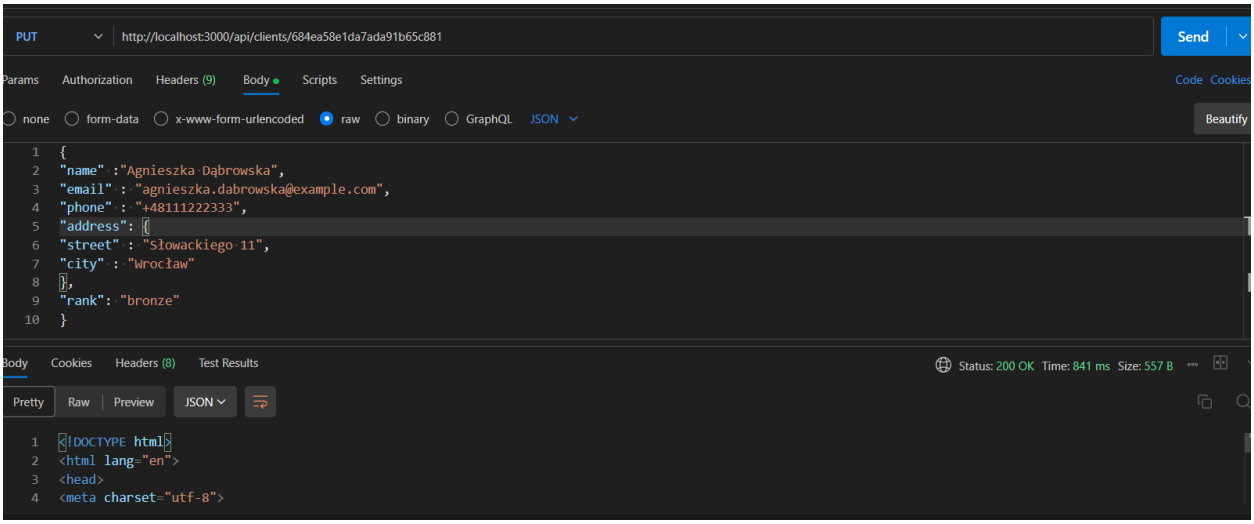
```
_id: ObjectId('684ea58e1da7ada91b65c881')
name : "Agnieszka Dąbrowska"
email : "agnieszka.dabrowska@example.com"
phone : "+48111222333"
address : Object
  street : "Słowackiego 22"
  city : "Wrocław"
rank : "bronze"
__v : 0
createdAt : 2025-06-15T10:50:54.293+00:00
updatedAt : 2025-06-15T10:50:54.293+00:00
```

Rysunek 6: Klientka której dane chcemy zaktualizować

Nowe dane:

```
{
"name" :"Agnieszka Dabrowska",
"email" : "agnieszka.dabrowska@example.com",
"phone" : "+48111222333",
```

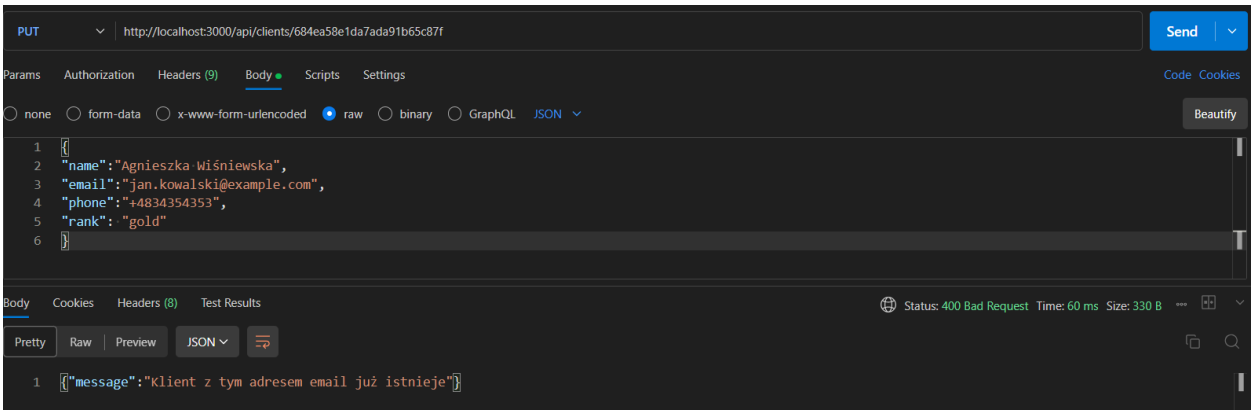
```
"address": {
"street" : "Slowackiego 11",
"city" : "Wroclaw"
},
"rank": "bronze"
}
```



Rysunek 7: Metoda updateClient() prawidłowo zmienia dane

Rysunek 8: Prawidłowo zmienione dane w **MongoDB Atlas**

Kiedy klient o takim emailu już istnieje:



Kiedy klient do aktualizacji nie istnieje:

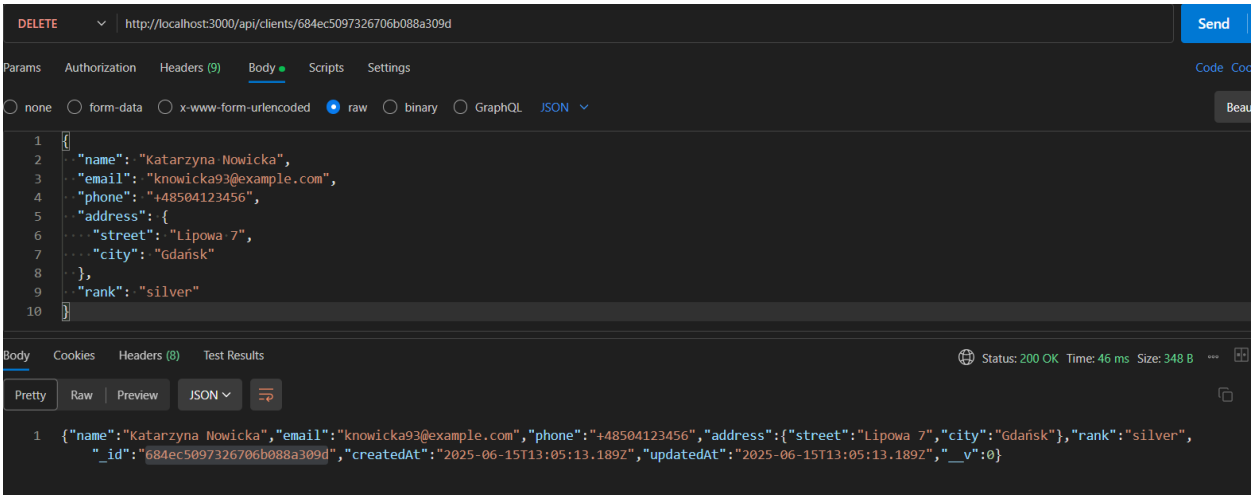
6.1.5 Metoda DELETE deleteClient()

Na potrzeby tej metody zostanie dodany nowy klient metodą **createClient()**

```
{
  "name": "Katarzyna Nowicka",
  "email": "knowicka93@example.com",
  "phone": "+48504123456",
  "address": {
    "street": "Lipowa 7",
    "city": "Gda sk"
  },
  "rank": "silver"
}
```

Został dodany nowy klient:

```
    {
  "name": "Katarzyna Nowicka",
  "email": "knowicka93@example.com",
  "phone": "+48504123456",
  "address": {
    "street": "Lipowa 7",
    "city": "Gda sk"
  },
  "rank": "silver"
}
```



Rysunek 9: Metoda deleteClient() prawidłowo usuwa podanego klienta

6.1.6 Metoda GET getClientRentalHistory()

Bierzemy tu pod uwagę klienta o id: **684ea58e1da7ada91b65c87e**

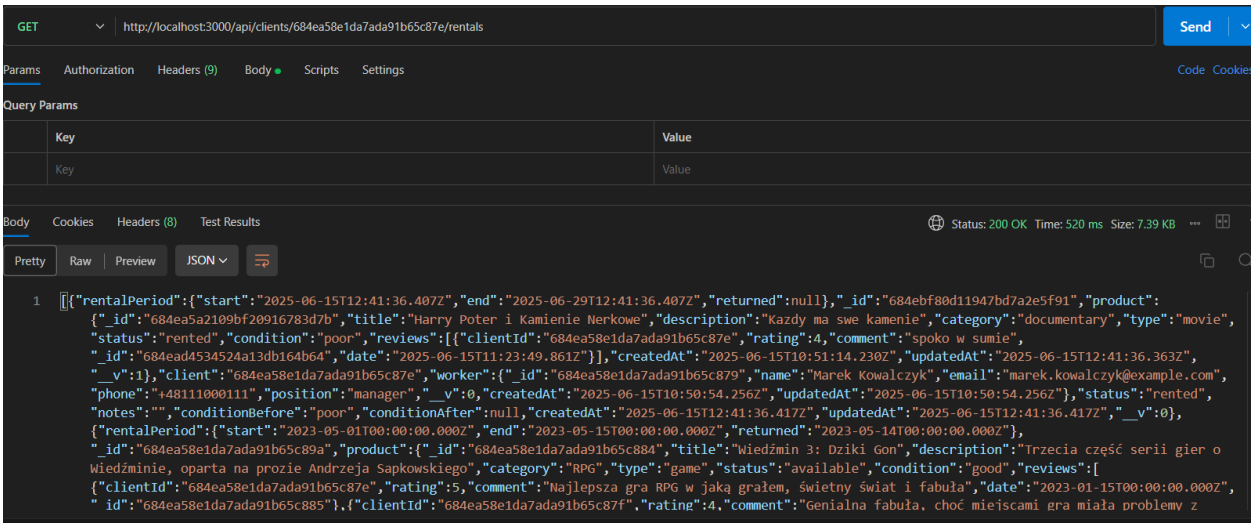
Klient wypożyczył:

```
_id: ObjectId('684ea58e1da7ada91b65c8aa')
product : ObjectId('684ea58e1da7ada91b65c88c')
client : ObjectId('684ea58e1da7ada91b65c87e')
worker : ObjectId('684ea58e1da7ada91b65c87a')
▸ rentalPeriod : Object
status : "overdue"
notes : "Niewielkie opóźnienie"
conditionBefore : "fair"
conditionAfter : "fair"
__v : 0
createdAt : 2025-06-15T10:50:54.381+00:00
updatedAt : 2025-06-15T10:50:54.381+00:00
```

```
_id: ObjectId('684ea58e1da7ada91b65c89a')
product : ObjectId('684ea58e1da7ada91b65c884')
client : ObjectId('684ea58e1da7ada91b65c87e')
worker : ObjectId('684ea58e1da7ada91b65c87a')
▸ rentalPeriod : Object
status : "returned"
notes : "Zwrócono w terminie, bez uszkodzeń"
conditionBefore : "good"
conditionAfter : "good"
__v : 0
createdAt : 2025-06-15T10:50:54.379+00:00
```

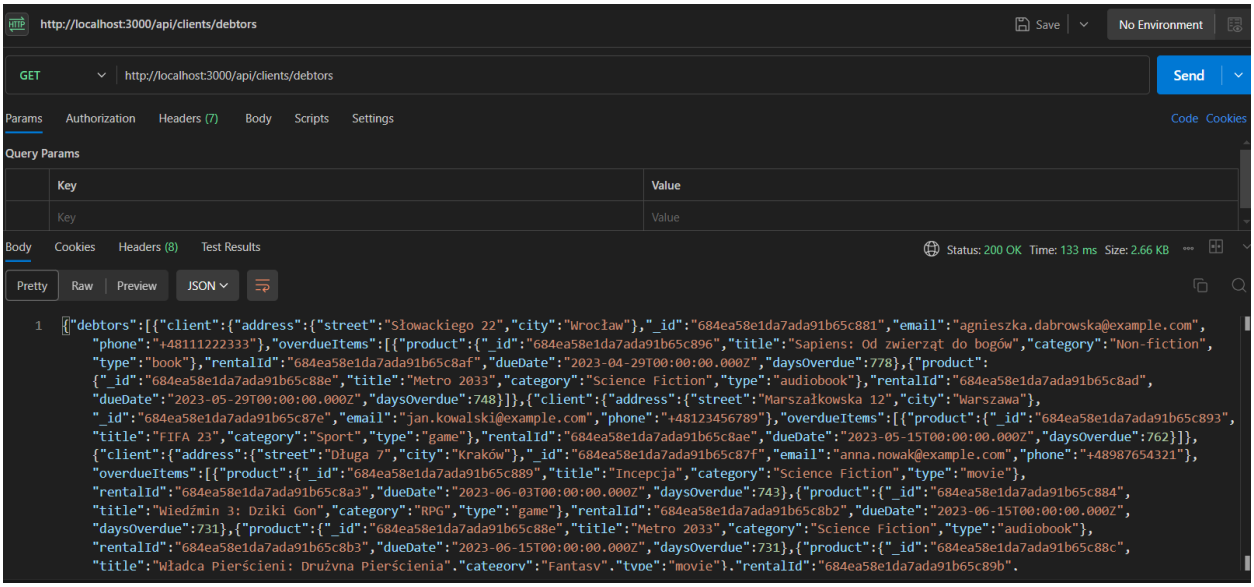
```
product : ObjectId('684ea58e1da7ada91b65c889')
client : ObjectId('684ea58e1da7ada91b65c87e')
worker : ObjectId('684ea58e1da7ada91b65c879')
▸ rentalPeriod : Object
status : "overdue"
notes : "Klient zapomniał o terminie zwrotu"
conditionBefore : "good"
conditionAfter : "good"
__v : 0
createdAt : 2025-06-15T10:50:54.380+00:00
updatedAt : 2025-06-15T10:50:54.380+00:00
```

```
_id: ObjectId('684ea58e1da7ada91b65c8a4')
product : ObjectId('684ea58e1da7ada91b65c887')
client : ObjectId('684ea58e1da7ada91b65c87e')
worker : ObjectId('684ea58e1da7ada91b65c87c')
▸ rentalPeriod : Object
status : "returned"
notes : "Wypożyczenie dla dzieci"
conditionBefore : "new"
conditionAfter : "good"
__v : 0
createdAt : 2025-06-15T10:50:54.380+00:00
```



Rysunek 12: Metoda getClientRentalHistory() zwraca prawidłowo wartość

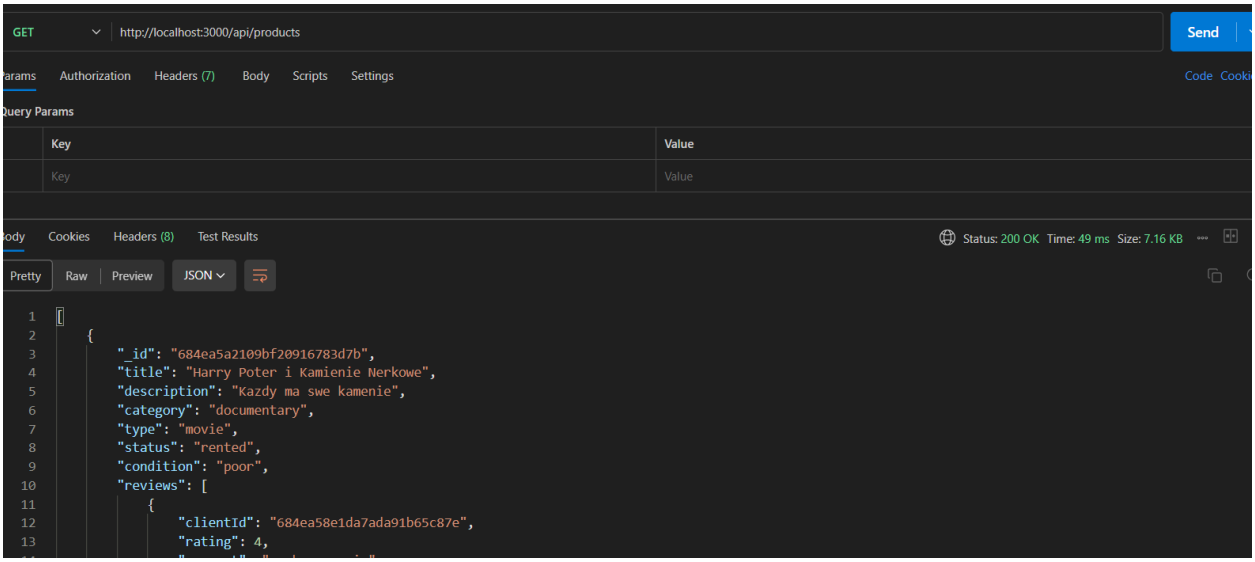
6.1.7 Metoda GET getDebtors()



Rysunek 13: Metoda getDebtors() prawidłowo zwraca wszystkich dłużnych klientów

6.2 Metody produktów

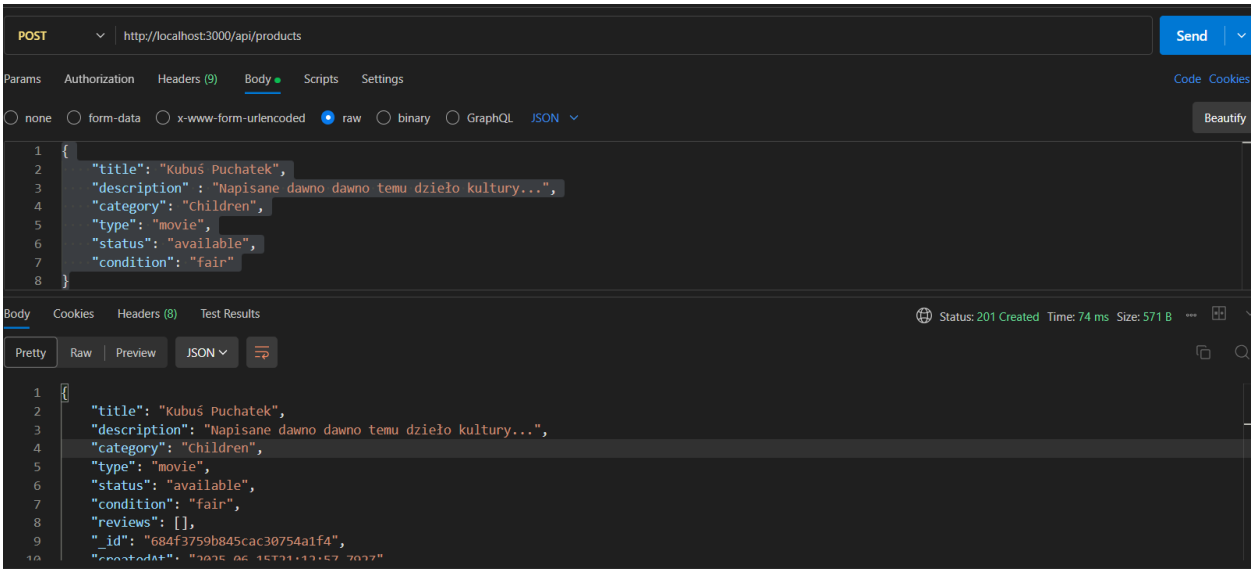
6.2.1 Metoda GET getAllProducts()



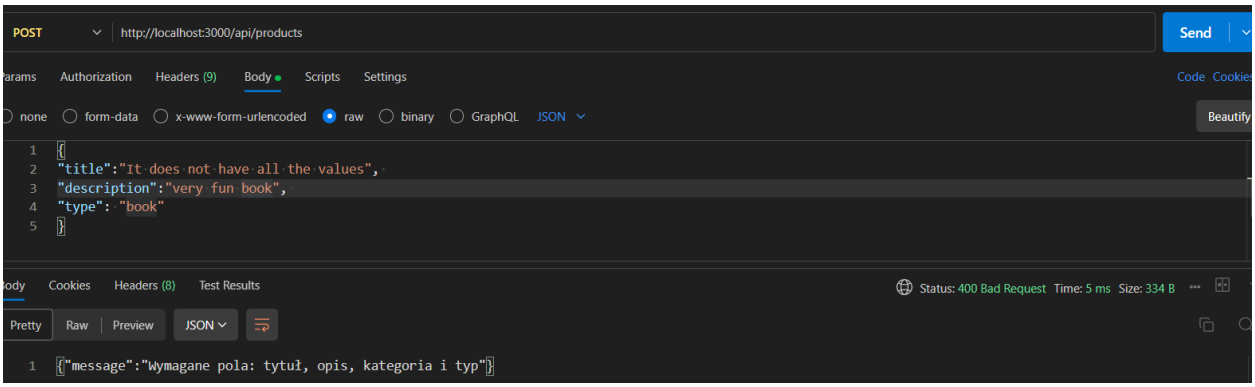
Rysunek 14: Metoda getAllProducts() prawidłowo zwraca wszystkie produkty

6.2.2 Metoda POST createProduct()

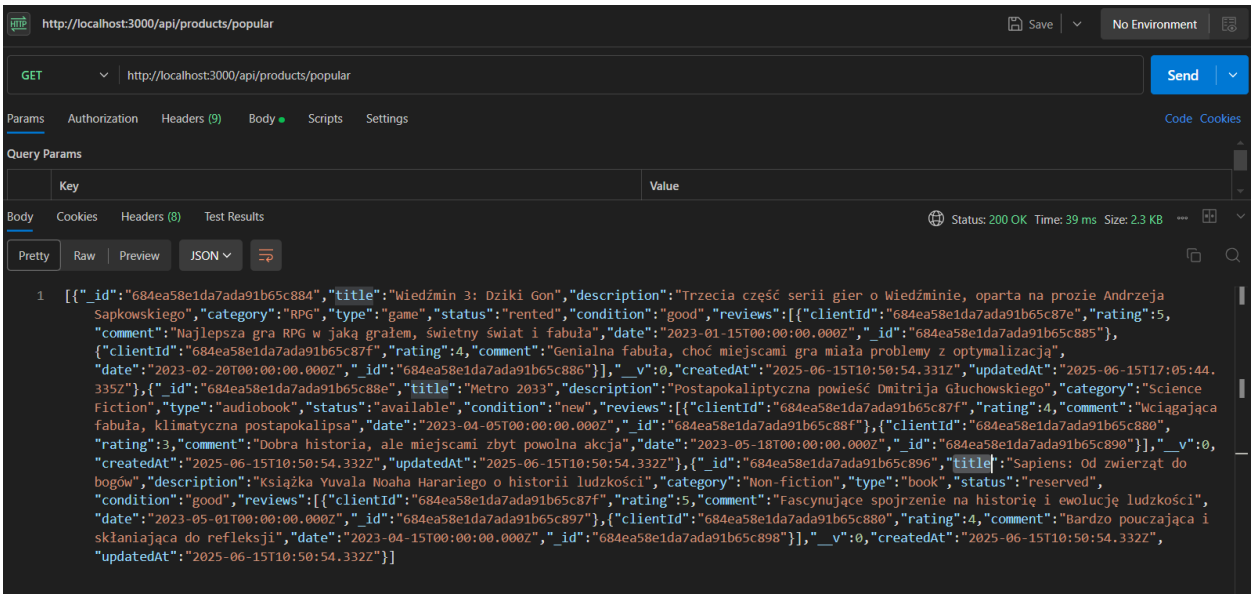
```
{
"title": "Kubu  Puchatek",
"description" : "Napisane dawno dawno temu dzie o kultury...",
"category": "Children",
"type": "movie",
"status": "available",
"condition": "fair"
}
```



Rysunek 15: Metoda createProduct() prawidłowo tworzy nowy obiekt



6.2.3 Metoda GET getMostPopularProducts()

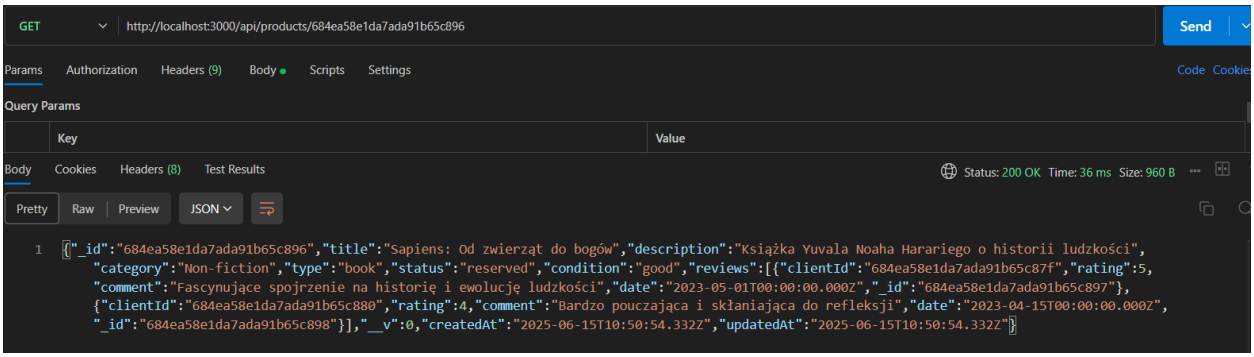


Rysunek 16: Metoda getMostPopularProducts() prawidłowo zwraca 3 najbardziej popularne produkty

6.2.4 Metoda GET getProductById()

Pobierzemy dla przykładu ten produkt:

```
_id: ObjectId('684ea58e1da7ada91b65c896')
title : "Sapiens: Od zwierząt do bogów"
description : "Książka Yuvala Noaha Hararriego o historii ludzkości"
category : "Non-fiction"
type : "book"
status : "reserved"
condition : "good"
reviews : Array (2)
__v : 0
createdAt : 2025-06-15T10:50:54.332+00:00
updatedAt : 2025-06-15T10:50:54.332+00:00
```

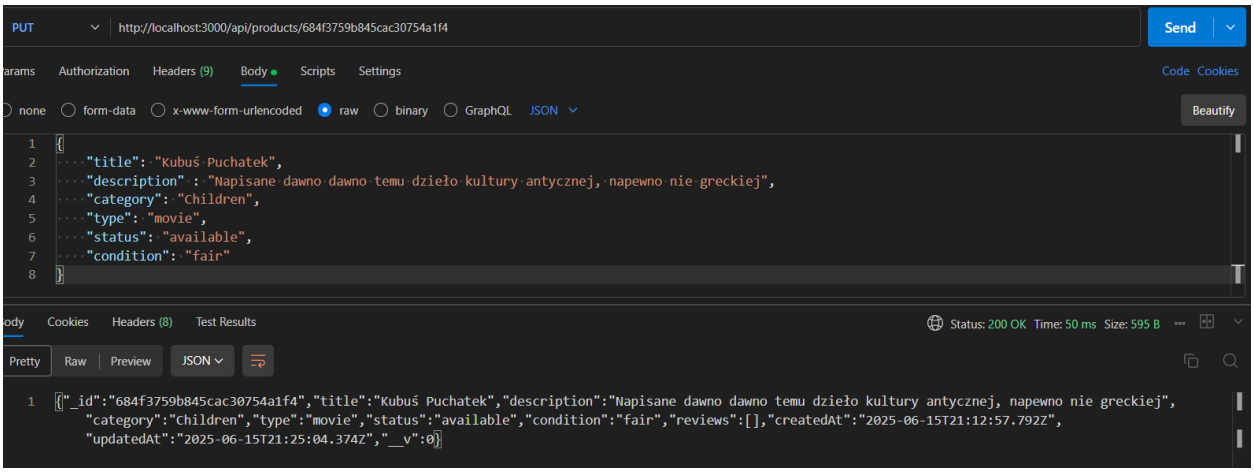


Rysunek 17: Metoda getMostPopularProducts() prawidłowo zwraca produkt o wybranym id

6.2.5 Metoda PUT updateProduct()

Bierzemy pod uwage ten produkt:

```
_id: ObjectId('684f3759b845cac30754a1f4')
title : "Kubuś Puchatek"
description : "Napisane dawno dawno temu dzieło kultury..."
category : "Children"
type : "movie"
status : "available"
condition : "fair"
▸ reviews : Array (empty)
createdAt : 2025-06-15T21:12:57.792+00:00
updatedAt : 2025-06-15T21:12:57.792+00:00
__v : 0
```

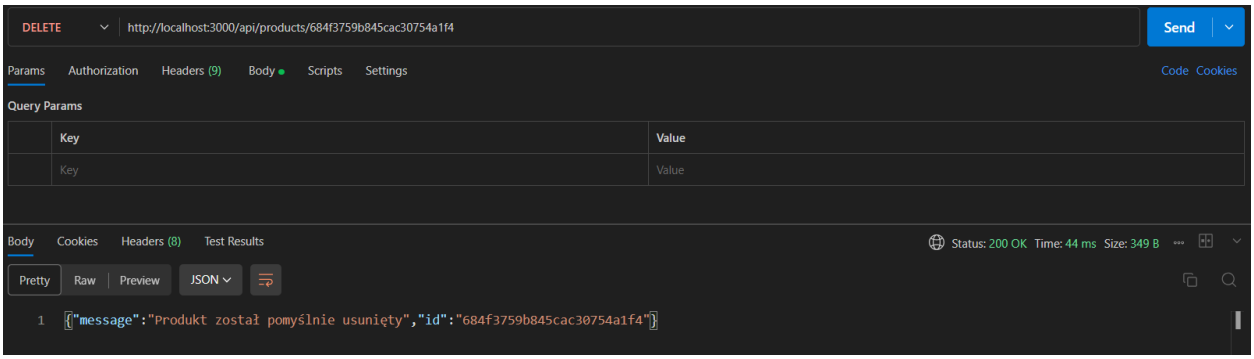


Rysunek 18: Metoda updateProduct() prawidłowo podmienia dane

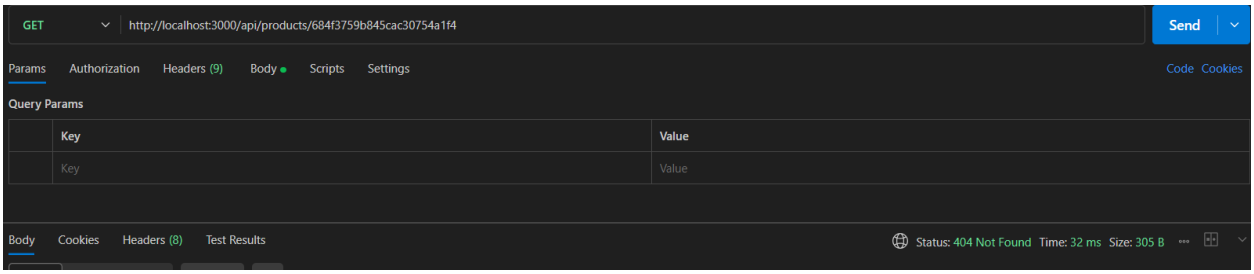
Po aktualizacji w **MongoDB Atlas**

```
_id: ObjectId('684f3759b845cac30754a1f4')
title : "Kubuś Puchatek"
description : "Napisane dawno dawno temu dzieło kultury antycznej, napewno nie grecki…"
category : "Children"
type : "movie"
status : "available"
condition : "fair"
▸ reviews : Array (empty)
createdAt : 2025-06-15T21:12:57.792+00:00
updatedAt : 2025-06-15T21:25:04.374+00:00
__v : 0
```

6.2.6 Metoda DELETE deleteProduct()

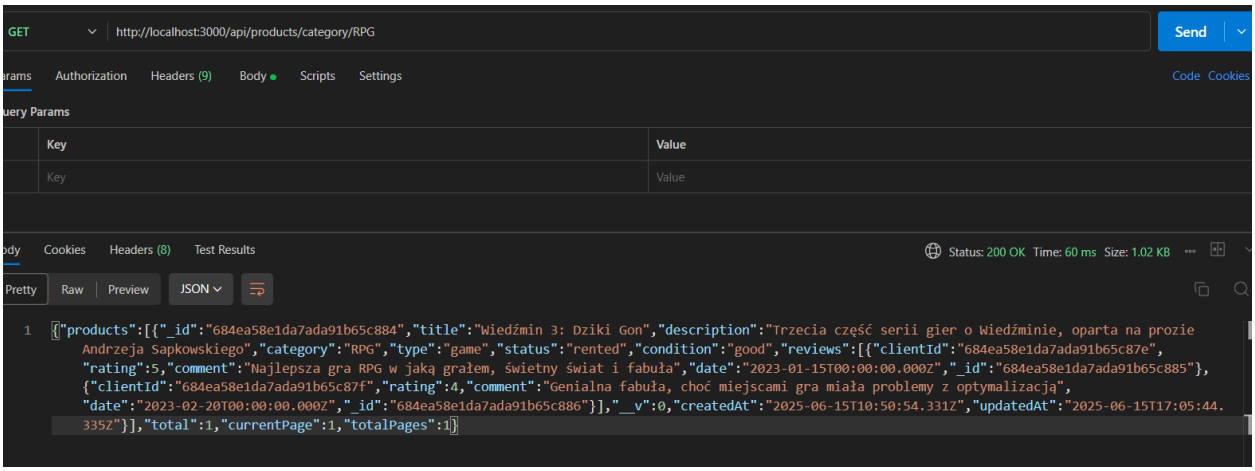


Rysunek 19: Metoda deleteProduct() prawidłowo usuwa produkt z bazy danych



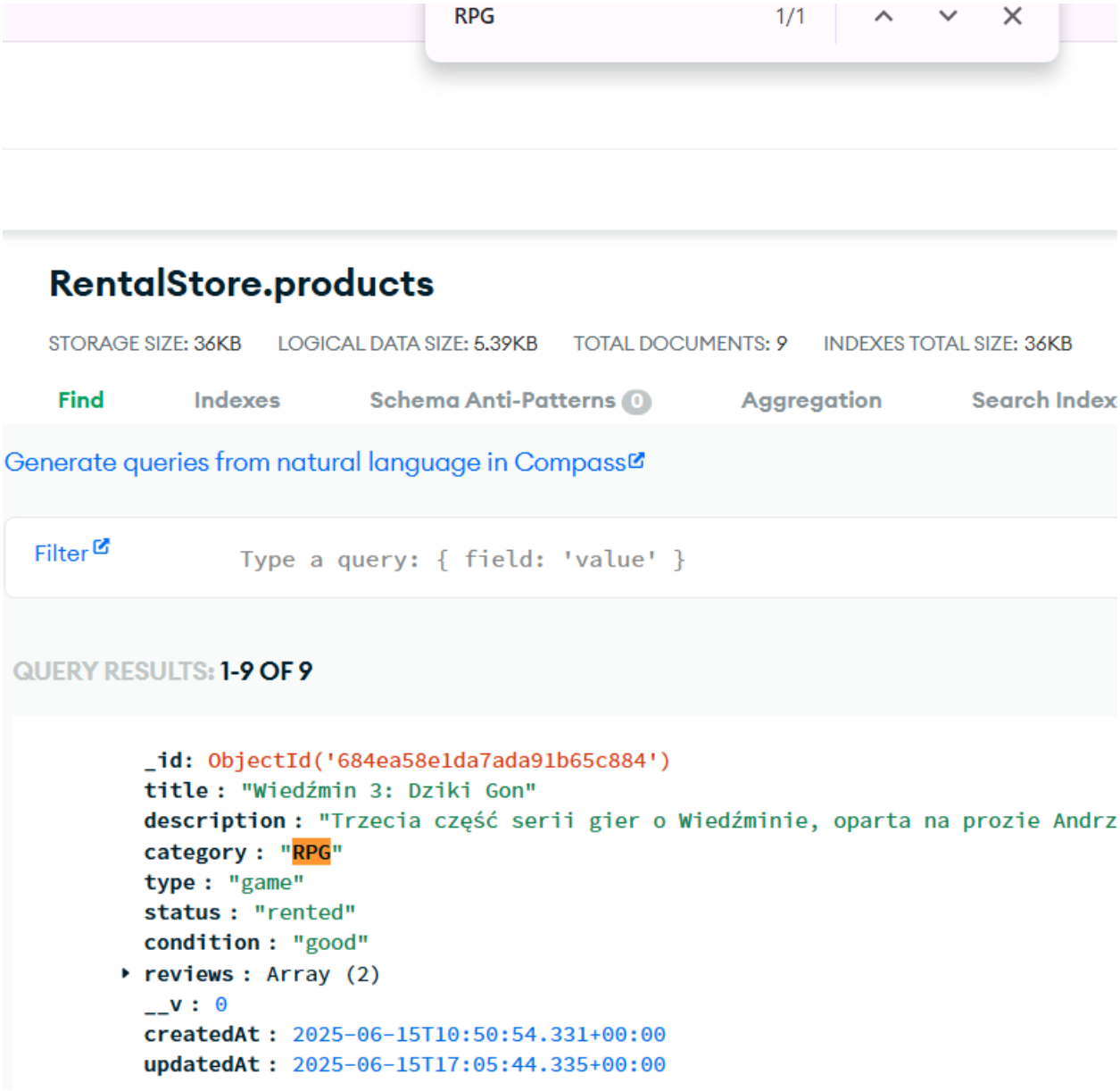
Rysunek 20: Metoda GET używająca funkcji getProductById(), aby sprawdzić czy deleteProduct napewno jest prawidłowo zaimplementowane

6.2.7 Metoda GET getProductByCategory()

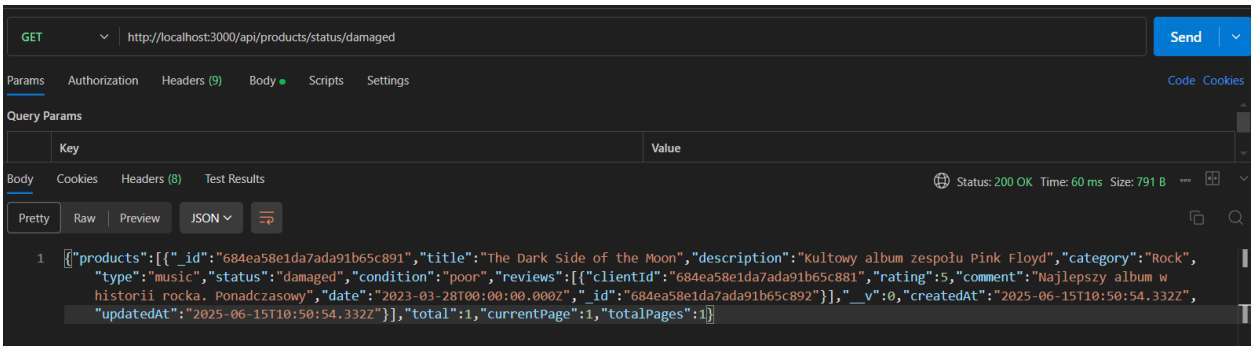


Rysunek 21: Metoda getProductByCategory() zwraca prawwidłowo wszystkie produkty w danej kategorii

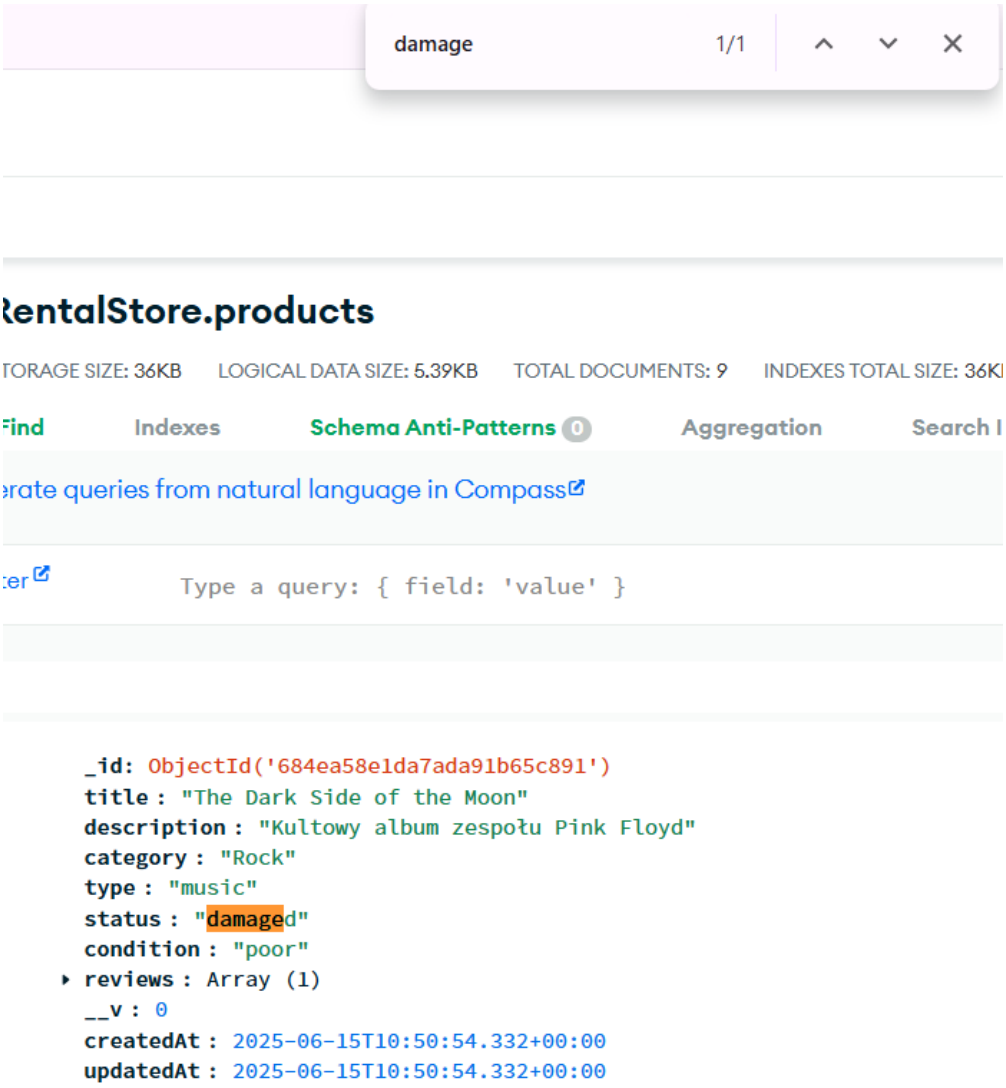
Potwierdzenie prawidłowego wywołania funkcji:


RPG
1/1
RentalStore.products
STORAGE SIZE: 36KB
LOGICAL DATA SIZE: 5.39KB
TOTAL DOCUMENTS: 9
INDEXES TOTAL SIZE: 36KB
Find
Indexes
Schema Anti-Patterns 0
Aggregation
Search Index
Generate queries from natural language in Compass
Filter
Type a query: { field: 'value' }
QUERY RESULTS: 1-9 OF 9
_id: ObjectId('684ea58e1da7ada91b65c884')
title : "Wiedźmin 3: Dziki Gon"
description : "Trzecia część serii gier o Wiedźminie, oparta na prozie Andrz
category : "RPG"
type : "game"
status : "rented"
condition : "good"
reviews : Array (2)
__v : 0
createdAt : 2025-06-15T10:50:54.331+00:00
updatedAt : 2025-06-15T17:05:44.335+00:00

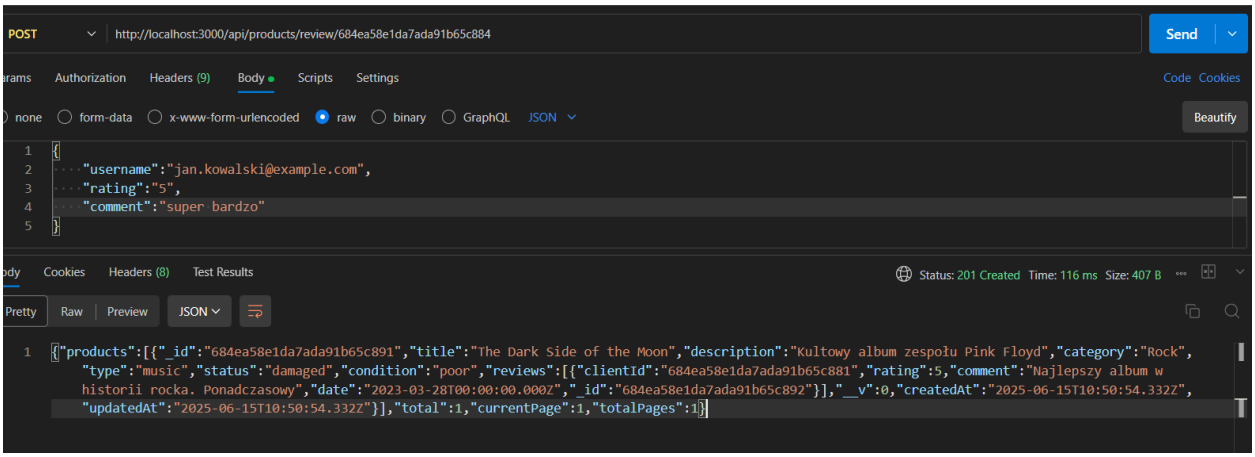
6.2.8 Metoda GET getProductByStatus()



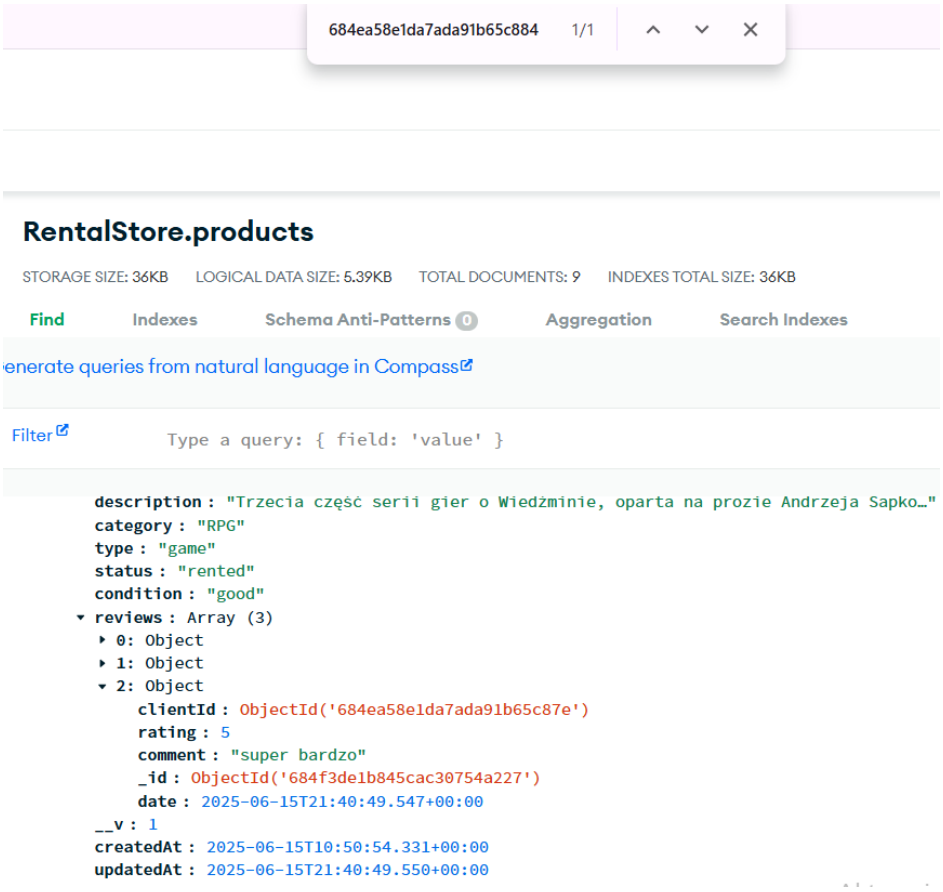
Rysunek 22: Metoda getProductByStatus() prawidłowo zwraca wszystkie produkty o danym statusie



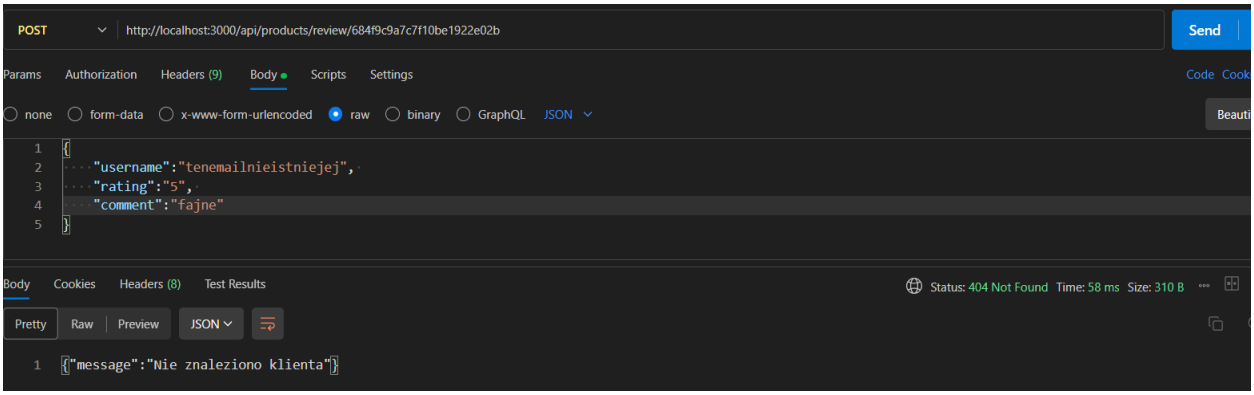
6.2.9 Metoda POST addProductReview()



Rysunek 23: Metoda addProductReview() prawidłowo dodaje opinie do produktu

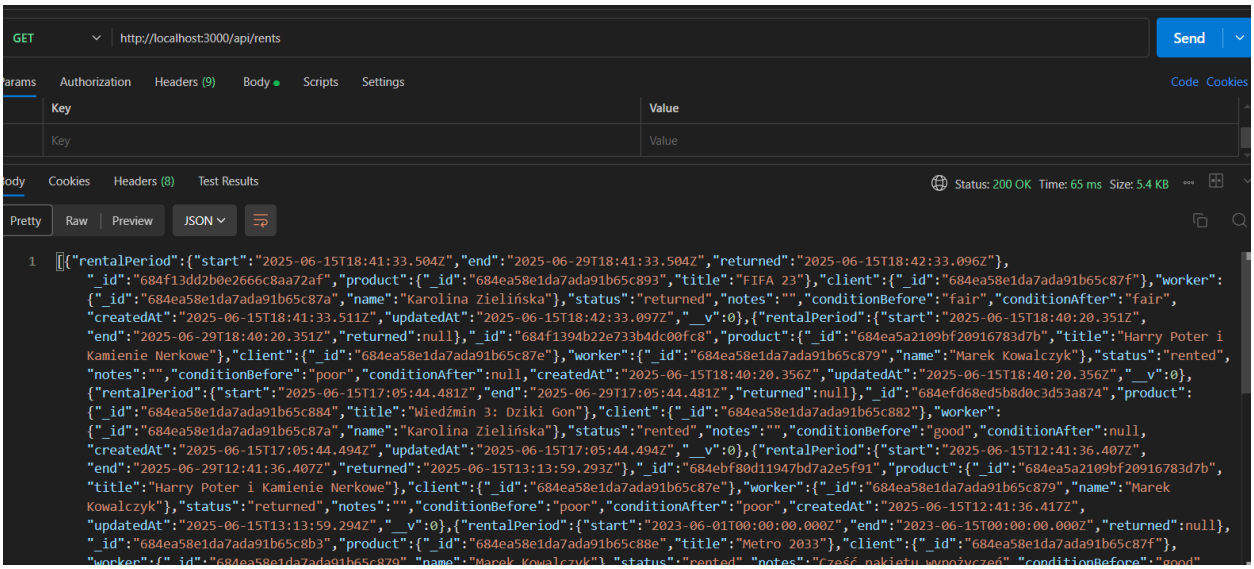


Jeżeli otrzymamy złe dane (niepełne/ zły email):



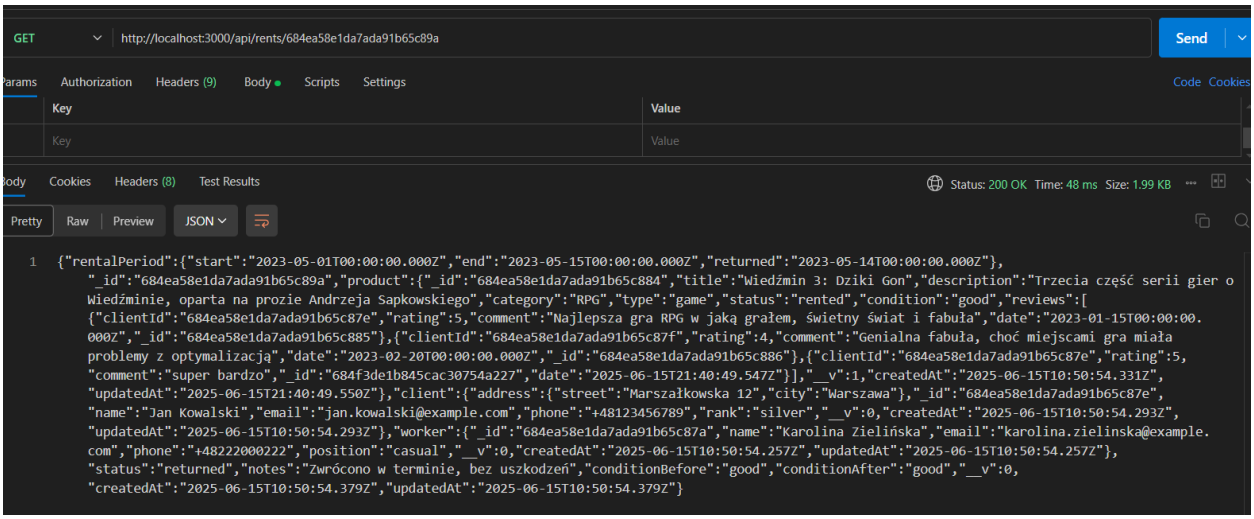
6.3 Metody wypożyczeń

6.3.1 Metoda GET getAllRents()



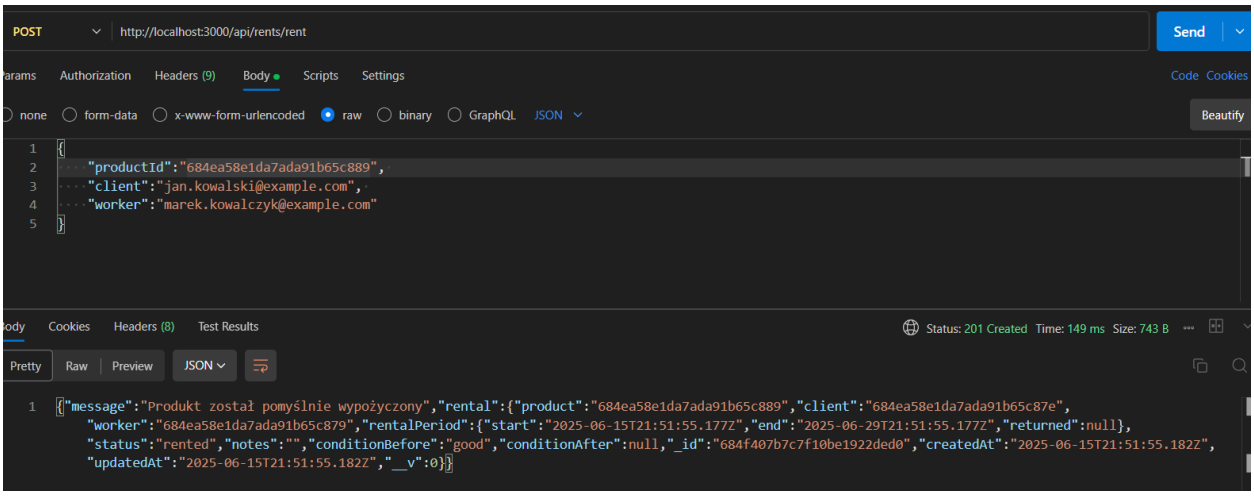
Rysunek 24: Metoda getAllRents() prawidłowo zwraca wszystkie wypożyczenia

6.3.2 Metoda GET getRentalById()



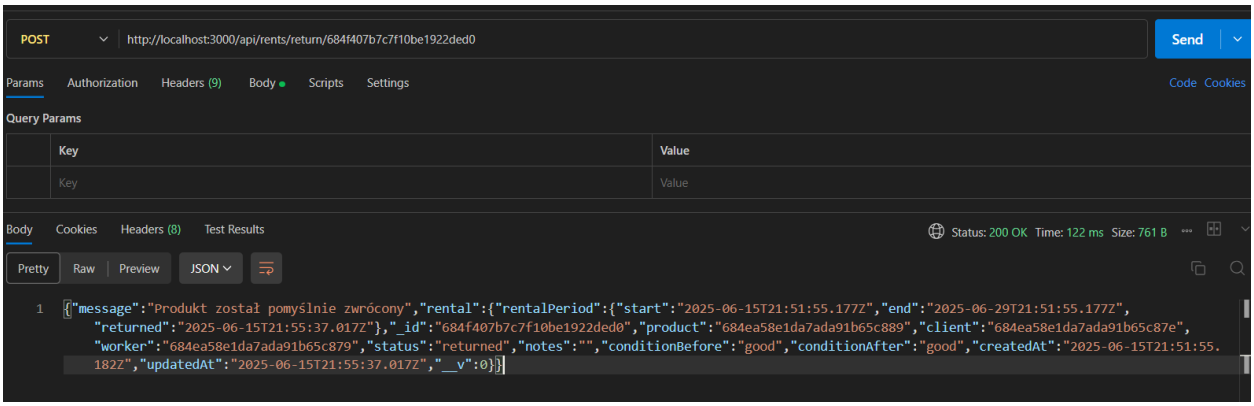
Rysunek 25: Metoda getRentalById() prawidłowo zwraca wypożyczenie po podanym id

6.3.3 Metoda POST rentProduct()



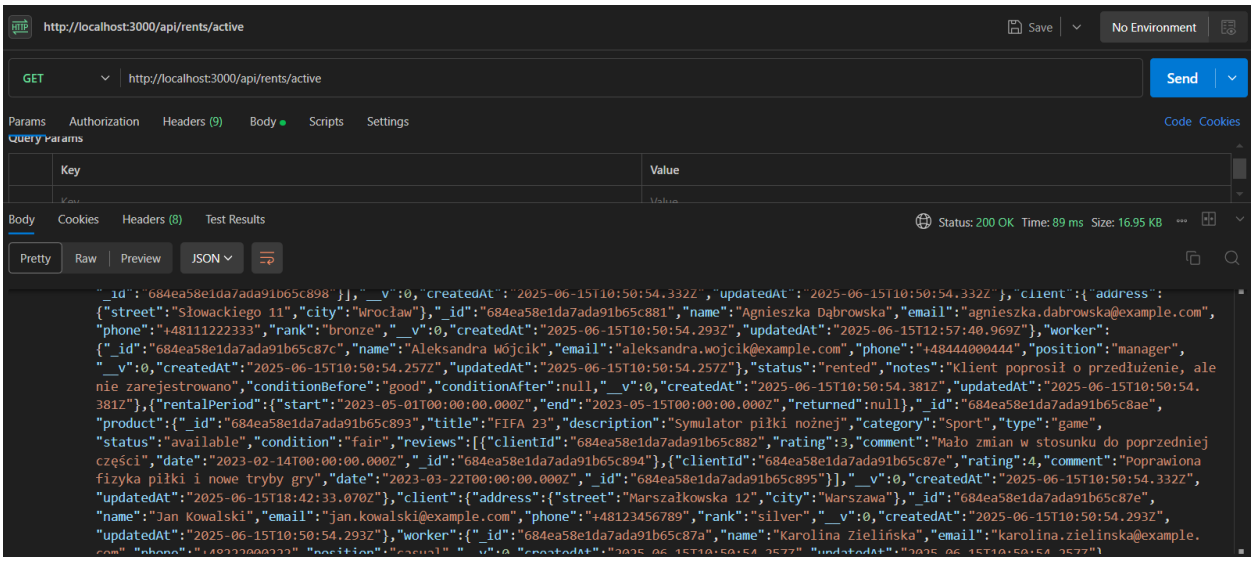
Rysunek 26: Metoda rentProduct prawidłowo tworzy wypożycznenie produktu w bazie danych

6.3.4 Metoda POST returnProduct()



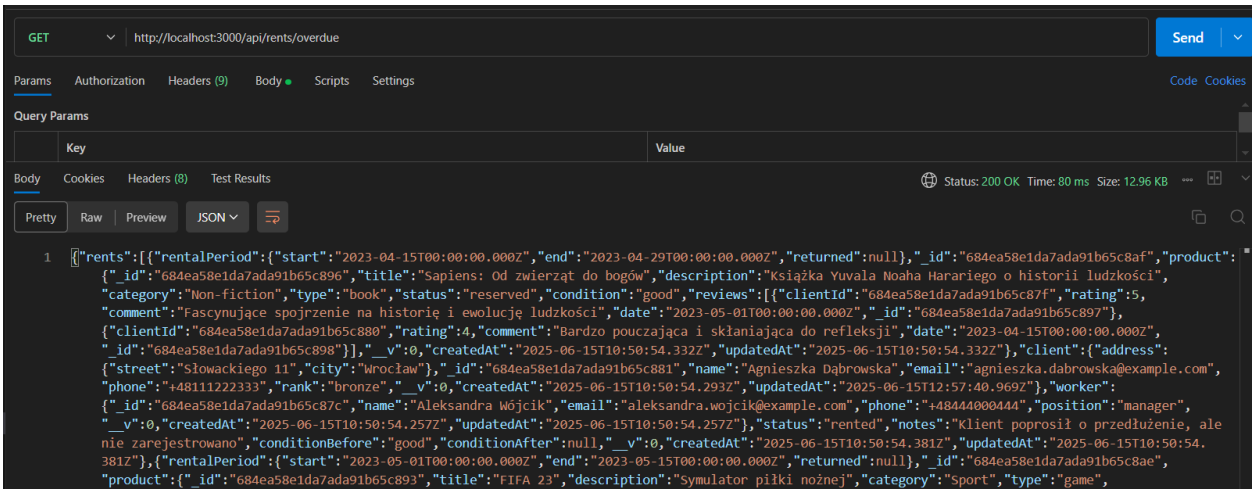
Rysunek 27: Metoda returnProduct() prawidłowo zwraca produkt w bazie danych (wykorzystany przykład z poprzednim ID)

6.3.5 Metoda GET getActiveRents()



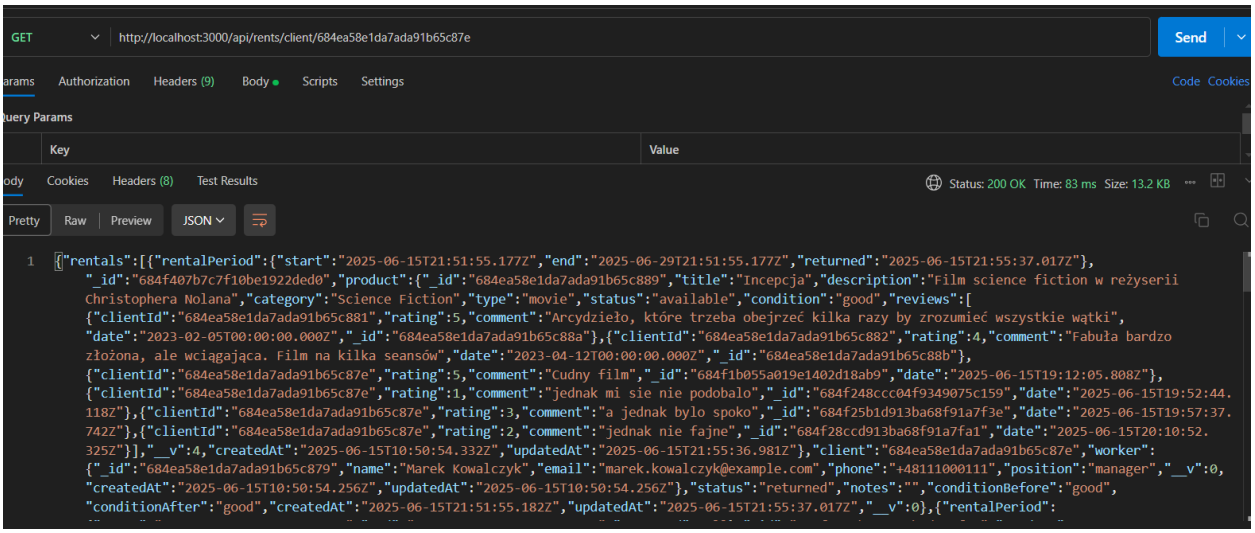
Rysunek 28: Metoda getActiveRents() prawidłowo zwraca wszystkie wypożyczenia w bazie danych które jeszcze maja miejsce

6.3.6 Metoda GET getOverdueRents()



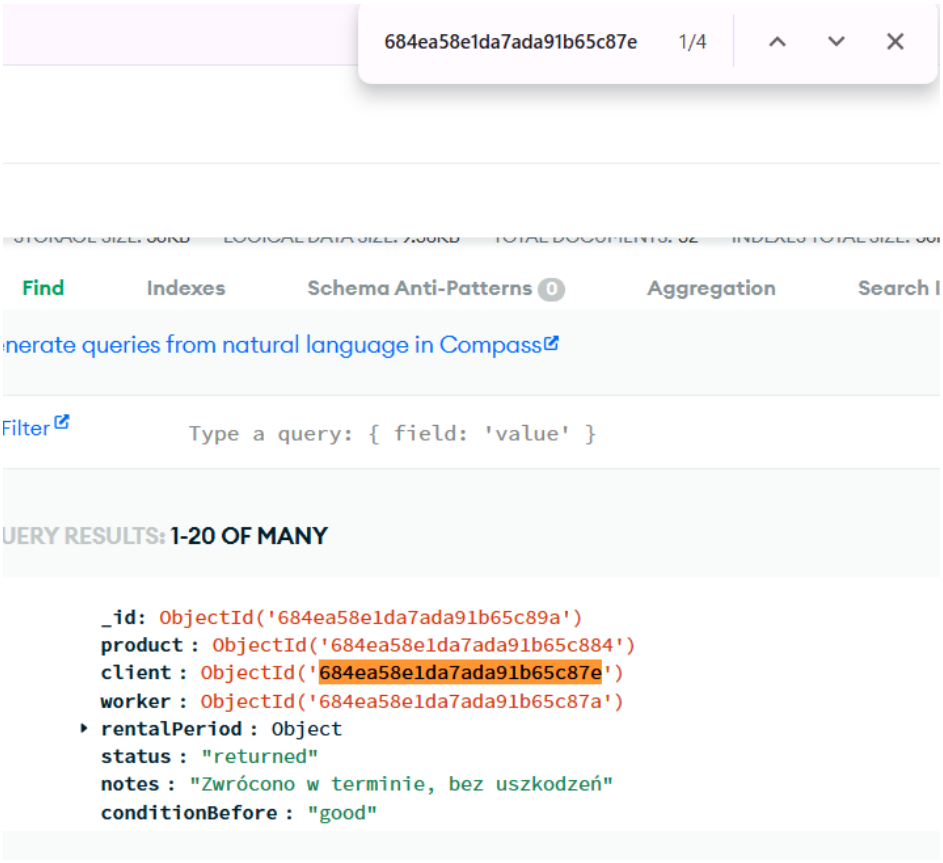
Rysunek 29: Metoda getOverdueRents() prawidłowo wszystkie zaległe wypożyczenia

6.3.7 Metoda GET getClientRentals()



Rysunek 30: Metoda getClientRentals() zwraca wszystkie wypożyczenia dokonane przez danego użytkownika

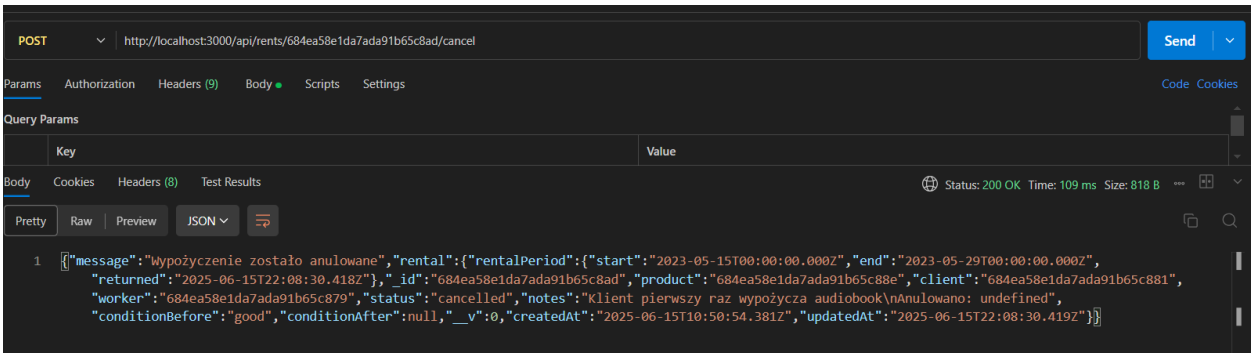
Potwierdzenie poprawności metody w **MongoDB Atlas**



6.3.8 Metoda POST cancelRental()

Wykorzystamy do testów to wypożyczenie:

```
_id: ObjectId('684ea58e1da7ada91b65c8ad')
product : 684ea58e1da7ada91b65c88e
client : 684ea58e1da7ada91b65c881
worker : 684ea58e1da7ada91b65c879
rentalPeriod : Object
status : "rented"
notes : "Klient pierwszy raz wypożycza audiobook"
conditionBefore : "good"
conditionAfter : null
__v : 0
```

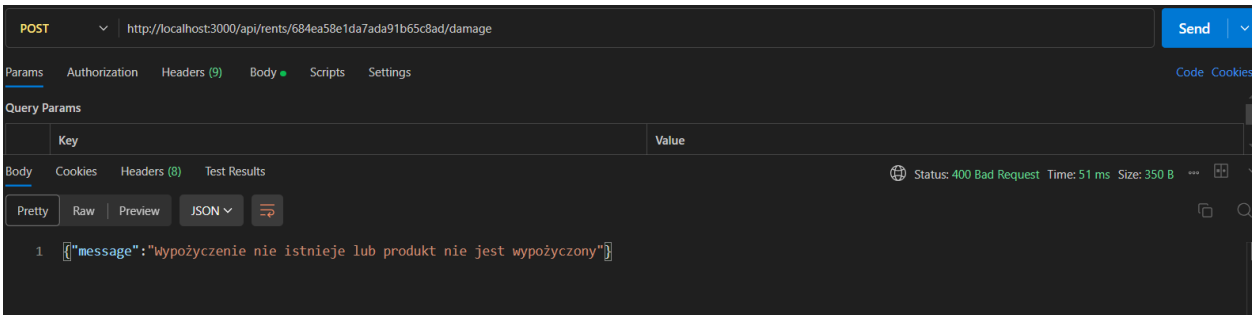


Rysunek 31: Metoda cancelRental() została wykonana pomyślnie

```
_id: ObjectId('684ea58e1da7ada91b65c8ad')
product : ObjectId('684ea58e1da7ada91b65c88e')
client : ObjectId('684ea58e1da7ada91b65c881')
worker : ObjectId('684ea58e1da7ada91b65c879')
▸ rentalPeriod : Object
status : "cancelled"
notes : "Klient pierwszy raz wypożycza audiobook
        Anulowano: undefined"
conditionBefore : "good"
conditionAfter : null
```

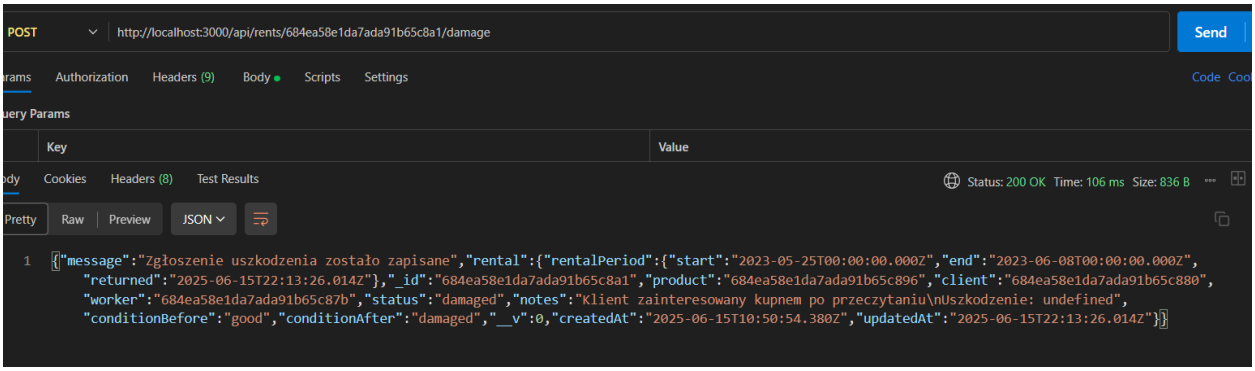
6.3.9 Metoda POST reportDamagedProduct()

Tu został wstawiony produkt, który wcześniej miał status cancelled, funkcja powstrzymała nas od zmiany jego stanu



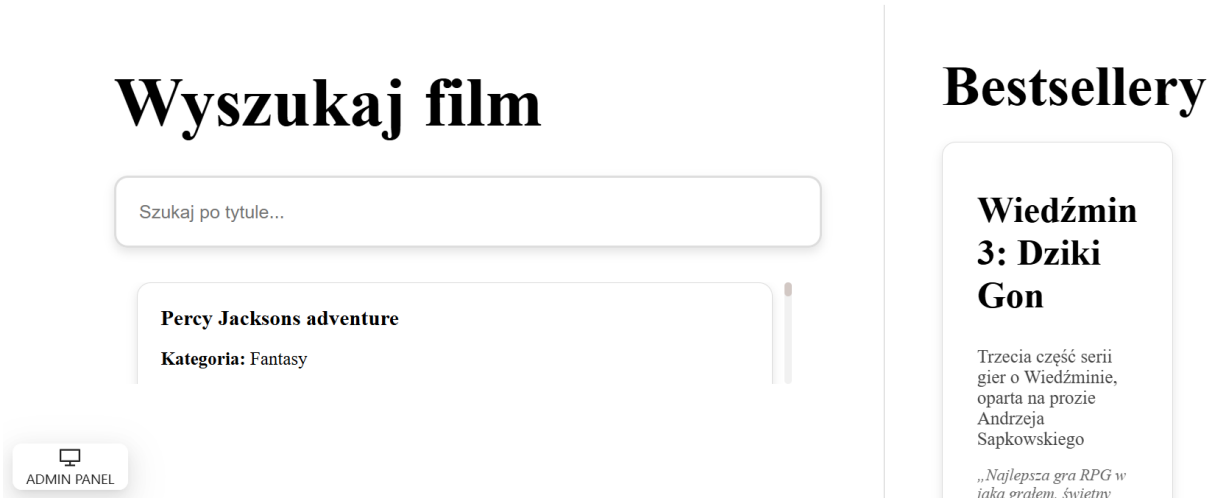
Działanie dla produktu który jest aktualnie wypożyczony:

```
_id: ObjectId('684ea58e1da7ada91b65c8a2')
product : ObjectId('684ea58e1da7ada91b65c884')
client : ObjectId('684ea58e1da7ada91b65c882')
worker : ObjectId('684ea58e1da7ada91b65c87a')
▸ rentalPeriod : Object
status : "returned"
notes : "Klient zachwycony grą"
conditionBefore : "good"
```

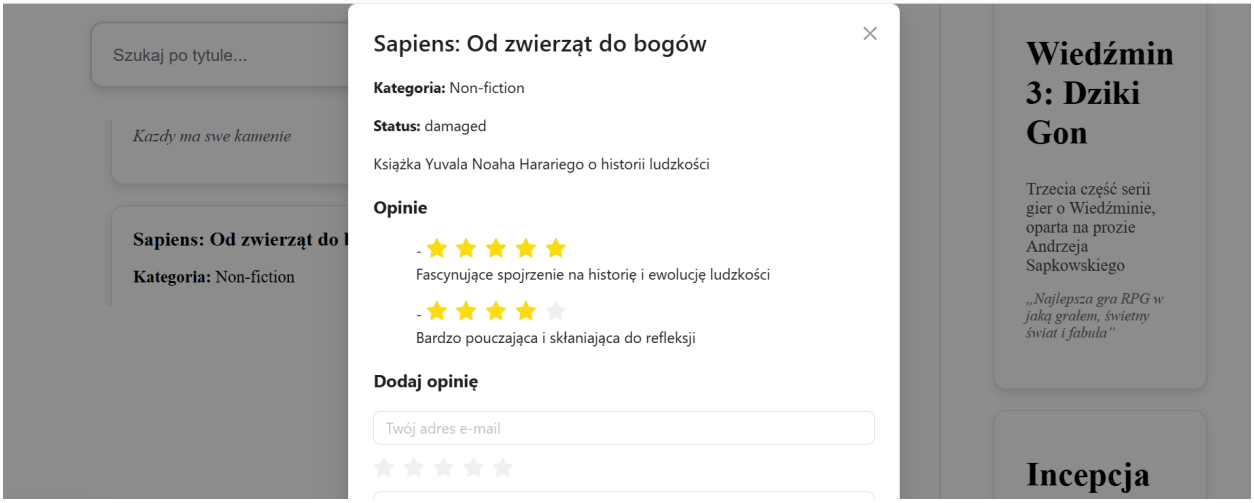


7 Frontend

7.1 Panel klienta



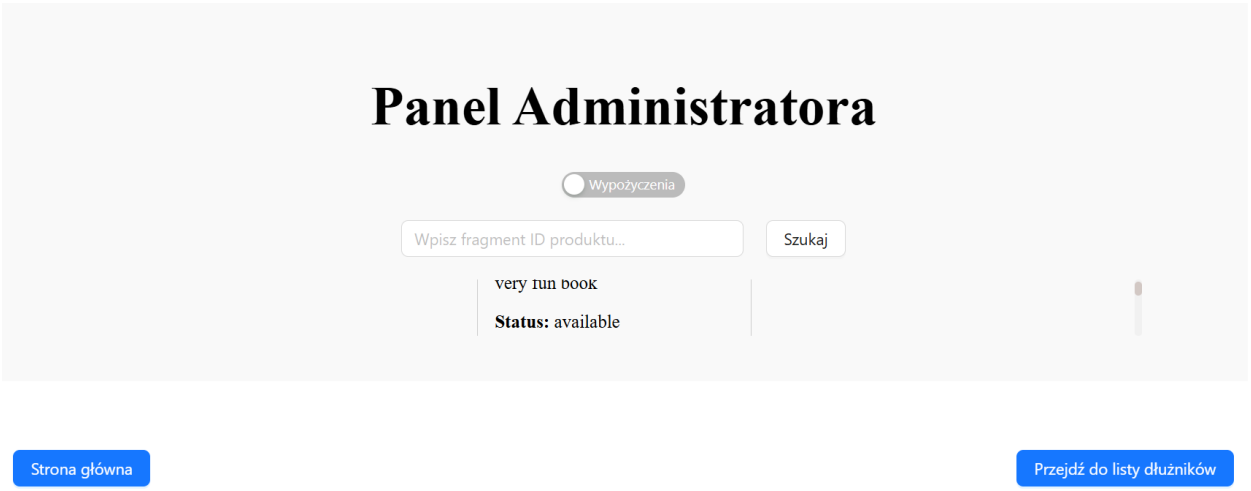
Rysunek 32: Strona główna - panel kliencki - można bo bestsellery, a także wyszukiwać filmy po tytule



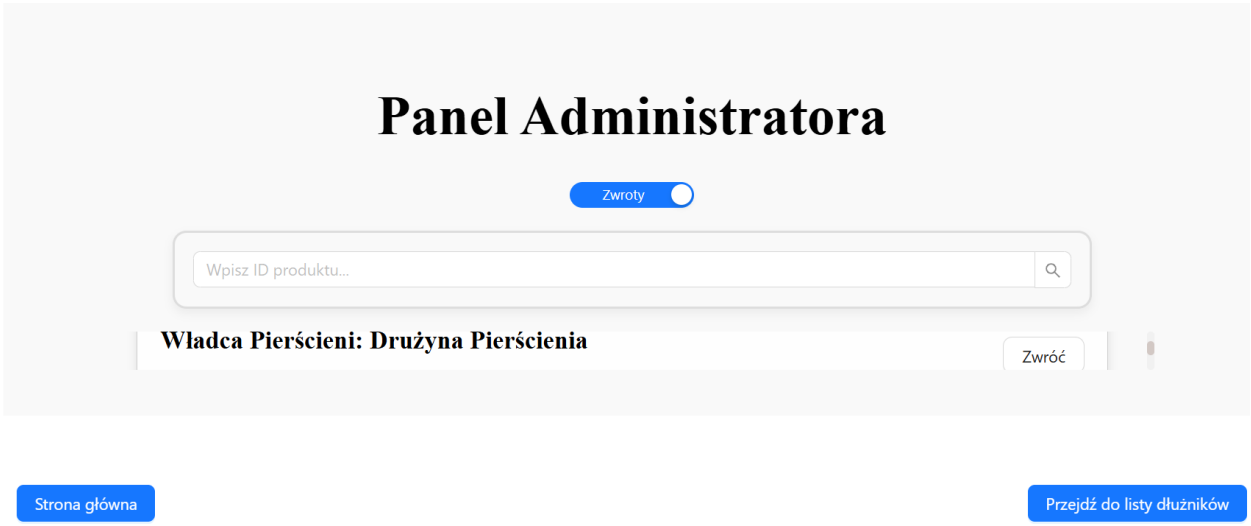
Rysunek 33: Strona główna - można dodawać opinie do produktów

Rysunek 34: Rejestracja do panelu administratora

7.2 Panel administratora



Rysunek 35: Zajmowanie się wypożyczeniami



Rysunek 36: Zajmowanie się zwrotami

8 Wnioski

MongoDB jest bardzo przyjemną i intuicyjną bazą dany do pracy, a wraz z **MongoDB Atlas** mogliśmy wszyscy mieć zdalnie dostęp do tej samej bazy danych, co znacznie ułatwiało prace. Jednym z też kluczowych i najcięższych elementów było połączenie i umożliwienie sprawnej komunikacji między backendem i frontendem i wymagało to momentami poprawienia niektórych metod.