E-commerce - Bazy danych II

Paweł Motyka & Hubert Gancarczyk

Wprowadzenie

Projekt sklepu internetowego, polegający na zasadzie iż sprzedającym jest nie tylko sklep ale i użytkownik sklepu.

Sklep oferuje możliwość wylosowania (drop) przedmiotu o losowej kwocie z wybranej przez klienta paczki. Każda taka paczka kosztuje określoną ilość pieniędzy, którą użytkownik jest w stanie wymienić na losowy przedmiot oferowany przez nasz sklep. Po wylosowaniu takiego przedmiotu jest on przypisywany do przedmiotów użytkownika, a następnie użytkownik w dowolnym momencie może zdecydować czy taki przedmiot chce otrzymać czy wystawić na sprzedaż dla innych użytkowników.

Saldo użytkownika może zostać w każdym momencie powiększone poprzez system doładowywania środków. Użytkownik jest w stanie dodawać produkty do ulubionych oraz śledzić swoje transakcje oraz ostatnie doładowania salda.

1. Opis modelu bazy danych

Opis bazy danych przedstawiamy jako modele napisane w języku JavaScript przy użyciu Mongoose. Wszystkie modele danych znajdują się w folderze /backend/models/

1.1. Opis modelu kolekcji users

Dany model znajdziemy w pliku **users.model.js**

```
const userSchema = new Schema({
  email: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  },
  name: {
    type: String
  },
  image: {
    type: String,
    default: defaultImage
  },
  role: {
    type: String,
```

```
    default: 'user'
  },
  money: {
    type: Number,
    default: 1000
  },
  favouritesProducts: {
    items: [
      {
        productId: {
          type: Schema.Types.ObjectId,
          ref: 'Product',
          required: true
        }
      }
    ]
  },
  openedPackages: [
    {
      packageId: {
        type: Schema.Types.ObjectId,
        required: true
      },
      openedAt: {
        type: Date,
        default: Date.now(),
        required: true
      }
    }
  ],
  products: [
    {
      productId: {
        type: Schema.Types.ObjectId,
        ref: 'Product',
        required: true
      },
      quantity: {
        type: Number,
        required: true
      }
    }
  ],
  orderInfo: {
```

```
    shippingAddress: {
      street: {
        type: String,
        default: null
      },
      city: {
        type: String,
        default: null
      },
      postalCode: {
        type: String,
        default: null
      },
      country: {
        type: String,
        default: null
      }
    },
    contactNumber: {
      type: String,
      default: null
    }
  }
});
```

1. **email:** Pole przechowuje adres e-mail użytkownika. Emial jest wymagany oraz musi być unikalne jako, iż służy ono w celu zalogowania użytkownika.

2. **password:** Pole przechowuje hasło użytkownika. Jest wymagane.

3. **name**: Pole przechowuje imię użytkownika.

4. **image:** Pole przechowuje ścieżkę do obrazu użytkownika. Domyślnie przyjmuje wartość defaultImage.

5. **role:** Pole przechowuje rolę użytkownika. Domyślnie przyjmuje wartość "user".

6. **money:** Pole przechowuje ilość pieniędzy użytkownika. Domyślnie przyjmuje wartość 1000 jako darmowe pieniądze do obrotu na naszej stronie.

7. **favouritesProducts:** Pole przechowuje listę ulubionych produktów użytkownika. Każdy produkt jest reprezentowany jako obiekt zawierający productId, czyli

identyfikator produktu. productId jest wymagane i odnosi się do kolekcji "Product" w bazie danych.

8. **openedPackages:** Pole przechowuje listę otwartych paczek przez użytkownika. Każda paczka jest reprezentowana jako obiekt zawierający packageId, czyli identyfikator paczki oraz openedAt, czyli data otwarcia. packageId i openedAt są wymagane.

9. **products:** Pole przechowuje listę produktów posiadanych przez użytkownika.

1.2. Opis modelu kolekcji products

Dany model znajdziemy w pliku product.model.js

```
const productSchema = new Schema(
    {
        name: {
            type: String,
            required: true
        },
        description: {
            type: String,
            required: true
        },
        price: {
            type: Number,
            required: true
        },
        category: {
            type: String,
            required: true
        },
        brand: {
            type: String,
            required: true
        },
        imageUrls: [
            {
                type: String,
                required: true
            }
        ],
        rating: {
            type: Number,
            default: 0
        },
        reviewsNumber: {
            type: Number,
            default: 0
```

```
        },
        unitsInStock: {
            type: Number,
            required: true
        },
        userId: {
            type: Schema.Types.ObjectId,
            ref: 'User'
        },
        userProductId: {
            type: Schema.Types.ObjectId,
            ref: 'User.products'
        }
    },
    { timestamps: true }
);
```

1. **name:** Pole przechowuje nazwę produktu. Pole to jest wymagane.
2. **description:** Pole przechowuje opis produktu. Pole to jest wymagane.
3. **price:** Pole przechowuje cenę produktu. Pole to jest wymagane.
4. **category:** Pole przechowuje kategorię produktu. Pole to jest wymagane.
5. **brand:** Pole przechowuje markę danego produktu. Pole to jest wymagane.
6. **imageUrls:** Pole przechowuje linki do zdjęć produktu. Pole to jest wymagane.
7. **rating:** Pole przechowuje sumaryczną wartość ocen danego produktu przez użytkowników. Pole to domyślnie przyjmuje wartość 0.
8. **reviewsNumber:** Pole przechowuje ilość ocen przez użytkowników. Pole to domyślnie przyjmuje wartość 0.
9. **userId:** Pole przechowuje referencję do id użytkownika który wystawił produkt na sprzedaż. Jeśli produkt został wystawiony przez sklep to pole to nie istnieje.
10. **userProductId:** Pole przechowuje referencję do id przedmiotu które znajduje się w polu products użytkownika. Jeśli produkt został wystawiony przez sklep to pole to nie istnieje

1.3. Opis modelu kolekcji packages

```
const packageSchema = new Schema(
  {
    name: {
      type: String,
      required: true
    },
    items: [
      {
        productId: {
          type: Schema.Types.ObjectId,
          ref: 'Product',
          required: true
        },
        probability: {
          type: Number,
          required: true
        }
      }
    ],
    cost: {
      type: Number,
      required: true
    },
    cooldown: {
      type: Number,
      required: true
    }
  },
  { timestamps: true }
);
```

1. **name**: Pole przechowuje nazwę paczki. Jest wymagane.

2. **items**: Pole przechowuje listę przedmiotów w paczce. Każdy przedmiot jest reprezentowany jako obiekt zawierający productId, czyli identyfikator produktu. productId odnosi się do kolekcji "Product" w bazie danych oraz dodatkowo, dla każdego przedmiotu jest określone probability, czyli prawdopodobieństwo jego wystąpienia w paczce. Zarówno productId jak i probability są wymagane. Suma prawdopodobieństw wylosowania przedmiotów nie musi wynosić 100, ponieważ w kodzie zastosowaliśmy skalowanie wartości na podstawie ustawionych prawdopodobieństw.

3. **cost:** Pole przechowuje koszt paczki. Pole jest typu numerycznego oraz jest ono wymagane.

4. **cooldown:** Pole przechowuje czas odnowienia paczki. Jest to pole typu numerycznego i jest wymagane.

1.4. Opis modelu kolekcji orders

```
const orderSchema = new Schema(
    {
        user: {
            type: Schema.Types.ObjectId,
            ref: 'User',
            required: true
        },
        items: [
            {
                product: {
                    type: Schema.Types.ObjectId,
                    ref: 'Product',
                    required: true
                },
                quantity: {
                    type: Number,
                    required: true
                }
            }
        ],
        shippingAddress: {
            street: {
                type: String,
                required: true
            },
            city: {
                type: String,
                required: true
            },
            postalCode: {
                type: String,
                required: true
            },
            country: {
                type: String,
                required: true
            }
        },
        contactNumber: {
            type: String,
            required: true
```

```
        },
        status: {
            type: String,
            required: true,
            default: 'pending',
            enum: ['pending', 'completed', 'cancelled']
        }
    },
    { timestamps: true }
);
```

1. **user:** Pole to przechowuję referencję do id użytkownika który złożył zamówienie. Pole to jest wymagane.

2. **items:** Pole to przechowuje przedmioty zamówione przez użytkownika oraz ich ilość. Pole to jest wymagane.

3. **shippingAddress:** Pole to jest reprezentowane przez obiekt zawierający informację na temat adresu zamawiającego takie jak ulica, miasto, kraj, kod pocztowy. Pole to jest wymagane.

4. **contactNumber:** Pole przechowuje numer kontaktowy związany z zamówieniem. Jest to pole wymagane.

5. **status:** Pole przechowuje status zamówienia. Może przyjąć jedną z trzech wartości: 'pending' (oczekujące), 'completed' (zrealizowane) lub 'cancelled' (anulowane). Domyślnie jest ustawione na 'pending'. Jest to pole wymagane i musi być jedną z wartości zdefiniowanych w enum.

1.5. Opis modelu kolekcji payments

```
const paymentSchema = new Schema({
    userId: {
        type: Schema.Types.ObjectId,
        ref: 'User',
        required: true
    },
    amount: {
        type: Number,
        required: true
    },
    date: {
        type: Date,
        default: Date.now
    },
    status: {
        type: String,
        default: 'pending',
        enum: ['completed', 'pending', 'failed']
    },
    orderId: {
        type: Schema.Types.ObjectId,
        ref: 'Order'
    }
});
```

1. **userId:** Pole przechowuje identyfikator użytkownika, dla którego dokonywana jest płatność. userId odnosi się do kolekcji "User" w bazie danych. Jest to pole wymagane.

2. **amount:** Pole przechowuje kwotę płatności. Jest to wartość numeryczna i jest wymagane.

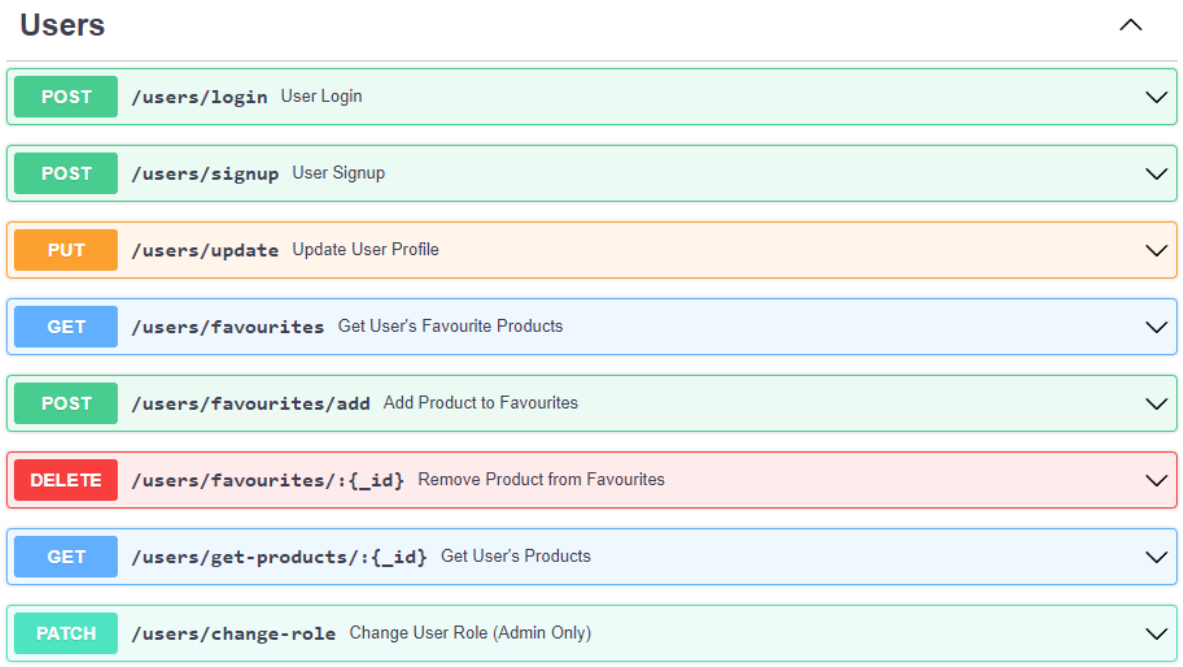
3. **date:** Pole przechowuje datę i czas płatności. Domyślnie przyjmuje aktualną datę i czas.

4. **status:** Pole przechowuje status płatności. Może przyjąć jedną z trzech wartości: 'completed' (zakończona), 'pending' (oczekująca) lub 'failed' (nieudana). Domyślnie jest ustawione na 'completed'. Jest to pole wymagane i musi być jedną z wartości zdefiniowanych w enum.

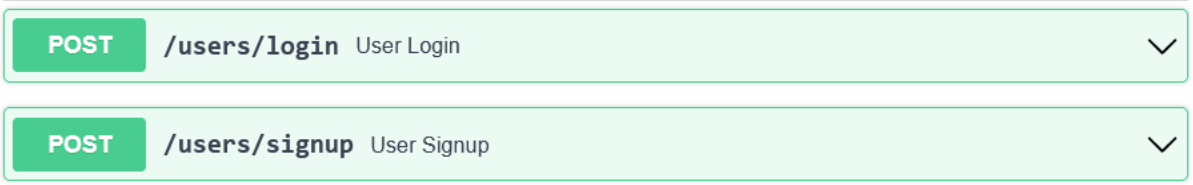
5. **orderId:** Pole przechowuje identyfikator powiązanego zamówienia. orderId odnosi się do kolekcji "Order" w bazie danych.

2. Opis najważniejszych funkcjonalności projektu

2.1. Opis funkcjonalności Users



Najważniejszymi funkcjonalnościami dla użytkowników jest możliwość logowania oraz rejestracji do sklepu.



Kod do rejestracji użytkownika:

```
const signupUser = async (req, res) => {
    const { email, password } = req.body;

    try {
        const user = await User.signup(email, password);
        const { name, image, favourites, role } = user;
        const token = createToken(user._id);
        const data = { email, token, name, favourites, role, avatar: image
```

```
};
        res.status(200).json(data);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
};

userSchema.statics.signup = async function (email, password) {
    if (!email || !password) {
        throw Error('All fields are required');
    }

    if (!validator.isEmail(email)) {
        throw Error('Email is not valid');
    }

    if (!validator.isStrongPassword(password)) {
        throw Error('Password is not strong enough');
    }

    const exists = await this.findOne({ email });

    if (exists) {
        throw Error('This email is already in use');
    }

    // The higher the value of saltRounds, the more secure the password is,
but the longer the signup process takes
    const saltRounds = 10;
    const salt = await bcrypt.genSalt(saltRounds);
    const hash = await bcrypt.hash(password, salt);
    const randomNumber = Math.floor(Math.random() * 900000) + 100000; //
Don't care about the uniqueness of the name
    const randomName = `Guest${randomNumber}`;
    const user = await this.create({ email, password: hash, name: randomName
});
    return user;
};
```

Funkcja **signupUser** jest odpowiedzialna za obsługę żądania rejestracji użytkownika. Po otrzymaniu żądania, funkcja pobiera adres e-mail i hasło z ciała żądania.

Pierwsza część funkcji sprawdza, czy adres e-mail i hasło zostały dostarczone. Jeśli którekolwiek z pól jest puste, zostanie zgłoszony błąd z komunikatem "All fields are required".

Następnie sprawdzane jest, czy podany adres e-mail jest poprawny za pomocą funkcji **validator.isEmail**. Jeśli adres e-mail jest nieprawidłowy, funkcja zgłosi błąd z komunikatem "Email is not valid".

Kolejna walidacja dotyczy siły hasła. Funkcja sprawdza, czy hasło spełnia wymagania dotyczące siły (np. minimalna długość, obecność dużych liter, małych liter, cyfr itp.) przy użyciu funkcji validator.isStrongPassword. Jeśli hasło nie jest wystarczająco silne, funkcja zgłosi błąd z komunikatem "Password is not strong enough".

Następnie funkcja sprawdza, czy istnieje już użytkownik o podanym adresie e-mail. Wykonuje to poprzez wyszukanie w bazie danych za pomocą metody **findOne** na modelu "User". Jeśli istnieje już użytkownik o tym adresie e-mail, funkcja zgłasza błąd z komunikatem "This email is already in use".

Jeśli wszystkie walidacje przebiegły pomyślnie, funkcja kontynuuje proces rejestracji. Tworzony jest tzw. salt, który jest używany do zabezpieczenia hasła przed atakami. Następnie hasło jest hashowane za pomocą funkcji bcrypt.hash wraz z wygenerowanym saltem. Wynikowy hash jest zapisywany w bazie danych wraz z adresem e-mail i wygenerowanym pseudolosowym imieniem użytkownika. W tym przypadku, pseudolosowe imię jest generowane poprzez dołączenie losowej liczby do ciągu "Guest" (np. "Guest123456").

Jeśli cały proces przebiegnie pomyślnie, zostanie utworzony token uwierzytelniający przy użyciu funkcji **createToken**. Następnie dane użytkownika zostaną wysłane w odpowiedzi jako obiekt JSON.

Kod do logowania użytkownika:

```
const loginUser = async (req, res) => {
    const { email, password } = req.body;

    try {
        const user = await User.login(email, password);
        const { name, image, favourites, role } = user;
        const token = createToken(user._id);
        const data = { email, token, name, favourites, role, avatar: image
};
        res.status(200).json(data);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
};

userSchema.statics.login = async function (email, password) {
    if (!email || !password) {
        throw Error('All fields are required');
    }
```

```
    if (!validator.isEmail(email)) {
        throw Error('Email is not valid');
    }

    const user = await this.findOne({ email });

    if (!user) {
        throw Error('Wrong email');
    }

    const match = await bcrypt.compare(password, user.password);

    if (!match) {
        throw Error('Incorrect password');
    }

    return user;
};
```

Funkcja **loginUser** jest odpowiedzialna za obsługę procesu logowania użytkownika. Po otrzymaniu żądania, funkcja pobiera adres e-mail i hasło z ciała żądania. Następnie funkcja wywołuje metodę login na modelu użytkownika w celu uwierzytelnienia użytkownika.

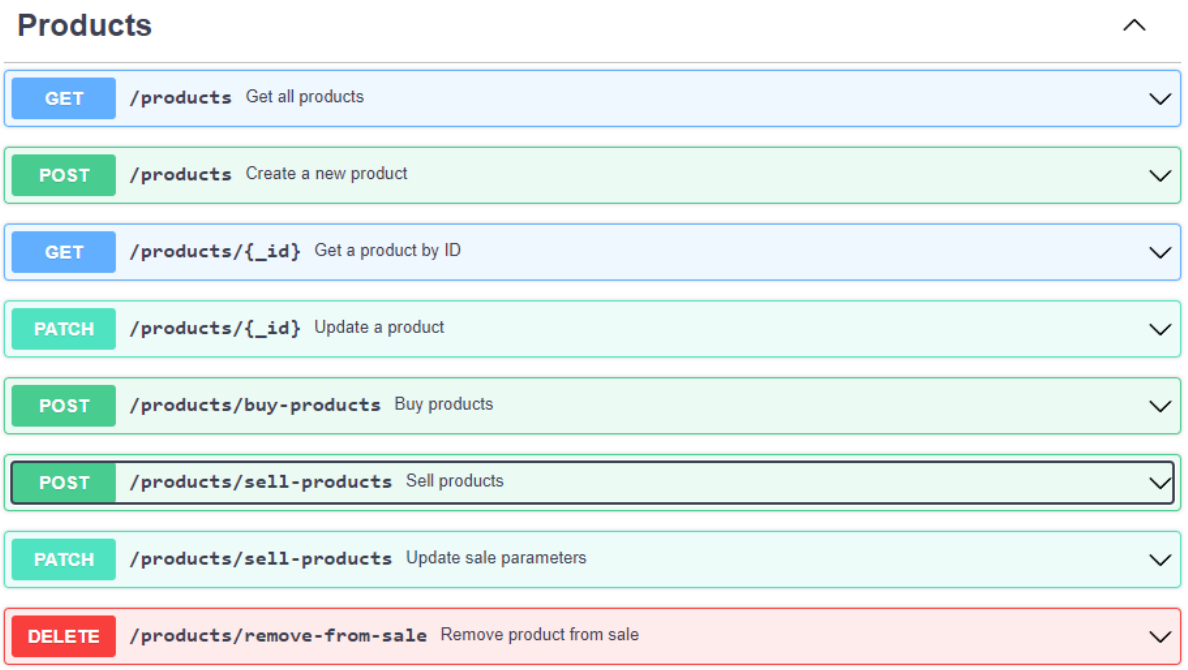
Metoda "login" na modelu użytkownika wykonuje następujące kroki:

1. Sprawdza, czy adres e-mail i hasło zostały dostarczone. Jeśli którekolwiek z pól jest puste, zostanie zgłoszony błąd z komunikatem "All fields are required".
2. Sprawdza, czy podany adres e-mail jest poprawny za pomocą funkcji **validator.isEmail**. Jeśli adres e-mail jest nieprawidłowy, zgłaszany jest błąd z komunikatem "Email is not valid".
3. Wyszukuje użytkownika w bazie danych na podstawie adresu e-mail za pomocą metody findOne na modelu użytkownika (this). Jeśli użytkownik o podanym adresie e-mail nie istnieje, zgłaszany jest błąd z komunikatem "Wrong email".
4. Porównuje podane hasło z zahaszowanym hasłem użytkownika przy użyciu funkcji **bcrypt.compare**. Jeśli hasła się nie zgadzają, zgłaszany jest błąd z komunikatem "Incorrect password".
5. Jeśli proces uwierzytelniania przebiegnie pomyślnie, zwracany jest obiekt reprezentujący uwierzytelnionego użytkownika.

Po uwierzytelnieniu użytkownika w funkcji **loginUser**, tworzony jest token uwierzytelniający przy użyciu funkcji **createToken**. Dane użytkownika są zwracane jako obiekt JSON.

Jeśli podczas procesu logowania wystąpi błąd, zostanie rzucony wyjątek, a klient otrzyma odpowiedź z kodem statusu 400 i komunikatem błędu w formacie JSON.

2.2. Opis funkcjonalności Products



Najważniejszym endpointem dla kolekcji Products, które opiszemy jest GET /product/{_id} oraz POST sell-products

```
const sellProducts = async (req, res) => {
  const { user } = req;
  const { productId, quantity, price } = req.body;

  try {
    if (!mongoose.Types.ObjectId.isValid(productId)) {
      return res.status(404).json({ error: 'There is no such product'
});
    }

    const alreadyExist =
      (await Product.findOne({ userProductId: productId })) == null
        ? false
        : true;
    if (alreadyExist) {
      return res.status(409).json({ error: 'Product already exists'
});
    }

    const index = user.products.findIndex(
      (product) => product._id.toString() === productId
```

```
    );
    if (index === -1) {
      return res
        .status(404)
        .json({ error: 'There is no such product in user inventory'
});
    }

    if (user.products[index].quantity < quantity) {
      return res.status(400).json({ error: 'There is no enough
quantity' });
    }

    const product = await
Product.findById(user.products[index].productId);
    const newProduct = await Product.create({
      ...product._doc,
      unitsInStock: quantity,
      price: price,
      userId: user._id,
      userProductId: user.products[index]._id,
      _id: new mongoose.Types.ObjectId()
    });

    res.status(200).json({ product: newProduct });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
```

Funkcja **sellProducts** jest odpowiedzialna za obsługę żądania sprzedaży produktów.

Najpierw funkcja pobiera z żądania aktualnego użytkownika (user) oraz informacje o produkcie do sprzedaży, takie jak identyfikator produktu (productId), ilość (quantity) oraz cena (price).

Następnie funkcja wykonuje szereg operacji w celu przetworzenia żądania sprzedaży. Oto opis tych operacji:

Sprawdzane jest, czy podany identyfikator produktu jest poprawnym identyfikatorem ObjectId. Jeśli nie jest, zwracany jest błąd 404 z informacją, że taki produkt nie istnieje.

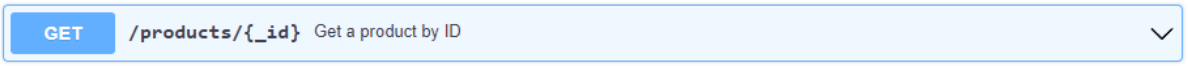
Następnie sprawdzane jest, czy produkt o podanym identyfikatorze już istnieje w bazie danych. Jeśli tak, zwracany jest błąd 409 z informacją, że produkt już istnieje i nie można go sprzedać ponownie.

Następnie funkcja wyszukuje produkt w ekwipunku użytkownika na podstawie identyfikatora produktu. Jeśli nie znajduje takiego produktu, zwracany jest błąd 404 z informacją, że produkt nie istnieje w inwentarzu użytkownika.

Jeśli produkt istnieje w ekwipunku użytkownika to, sprawdzane jest, czy ilość produktu w ekwipunku użytkownika jest większa lub równa żądanej ilości do sprzedaży. Jeśli ilość jest mniejsza, zwracany jest błąd 400 z informacją, że nie ma wystarczającej ilości produktu do sprzedaży.

Następnie pobierane są szczegóły produktu na podstawie identyfikatora produktu. Tworzony jest nowy produkt (newProduct) na podstawie tych szczegółów, który będzie reprezentował sprzedawany produkt. Nowy produkt jest zapisywany w bazie danych.

Na koniec, jeśli cały proces przebiegł pomyślnie, zwracana jest odpowiedź o statusie 200 (OK) zawierająca informacje o nowo utworzonym produkcie.




```
const getProduct = async (req, res) => {
  const { _id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(_id)) {
    return res.status(404).json({ error: 'No such product' });
  }

  const product = await Product.findById(_id);

  if (!product) {
    res.status(400).json({ error: 'No such product' });
  }

  res.status(200).json(product);
};
```

Funkcja **getProduct** służy do pobierania informacji o konkretnym produkcie na podstawie przesłanego id produktu.

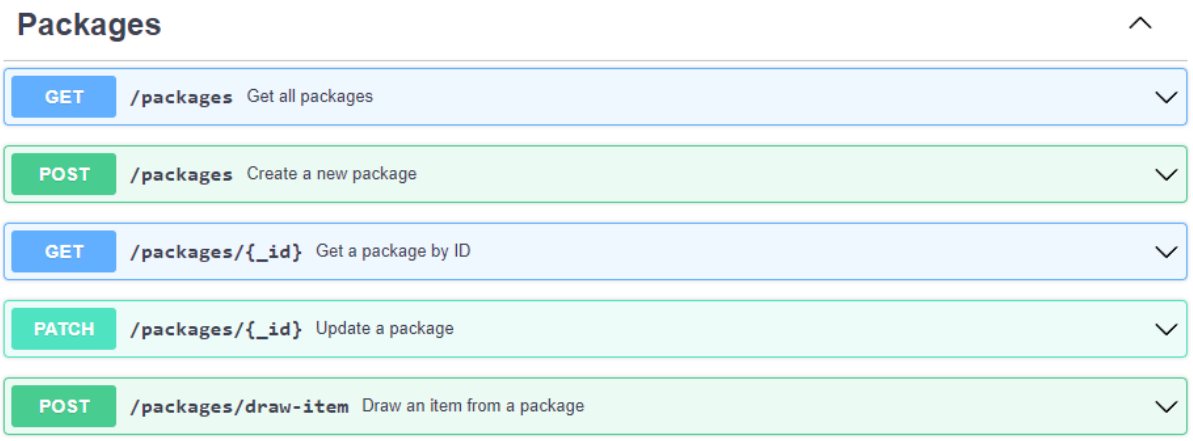
1. Najpierw pobierany jest identyfikator produktu (_id) z parametrów żądania.

2. Sprawdzane jest, czy podany identyfikator jest poprawnym identyfikatorem ObjectId. Jeśli nie jest, zwracany jest błąd 404 z informacją, że nie ma takiego produktu.

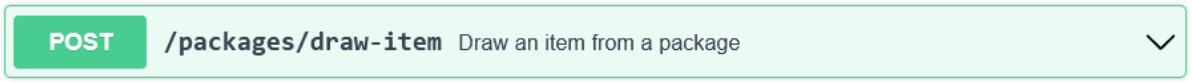
3. Następnie wykonywane jest wyszukiwanie produktu w bazie danych na podstawie identyfikatora. Jeśli nie zostanie znaleziony żaden produkt, zwracany jest błąd 400 z informacją, że nie ma takiego produktu.

4. Jeśli produkt zostanie znaleziony, zwracana jest odpowiedź o statusie 200 (OK) zawierająca informacje o produkcie.

2.3. Opis funkcjonalności Packages



Najważniejszą funkcjonalnością jest losowanie przedmiotu z wybranej paczki:




```
const getRandomItem = async (req, res) => {
    const { _id } = req.body;
    const { user } = req;

    try {
        if (!mongoose.Types.ObjectId.isValid(_id)) {
            return res.status(404).json({ error: 'No such package' });
        }

        const package = await
Package.findById(_id).populate('items.productId');
```

```
        if (!package) {
            return res.status(400).json({ error: 'No such package' });
        }

        const openedPackageIndex = user.openedPackages.findIndex(
            (pkg) => pkg.packageId.toString() === _id.toString()
        );
        if (
            openedPackageIndex !== -1 &&
            compareDates(
                new Date(),
                user.openedPackages[openedPackageIndex].openedAt
            ) < package.cooldown
        ) {
            return res.status(400).json({
                error: `You have to wait ${package.cooldown} seconds between
drawing the items`
            });
        }

        if (user.money < package.cost) {
            return res.status(400).json({ error: 'Insufficient funds' });
        }

        // Draw item
        const minProbability = package.items.reduce(
            (acc, item) => Math.min(acc, item.probability),
            Infinity
        );
        const scale = 1 / minProbability;

        const chances = package.items.reduce(
            (acc, item) => acc + item.probability * scale,
            0
        );
        const randomNumber = Math.random() * chances;

        let chancesSum = 0;
        let drawItem = null;

        for (const item of package.items) {
            chancesSum += item.probability * scale;

            if (randomNumber < chancesSum) {
                drawItem = item;
                break;
            }
        }
```

```
        // Modify information about the user drawing the item
        if (openedPackageIndex === -1) {
            user.openedPackages.push({ packageId: _id, openedAt: new Date()
});
        } else {
            user.openedPackages[openedPackageIndex].openedAt = new Date();
        }

        const drawItemIndex = user.products.findIndex(
            (product) =>
                product.productId.toString() ===
drawItem.productId._id.toString()
        );
        if (drawItemIndex === -1) {
            user.products.push({ productId: drawItem.productId, quantity: 1
});
        } else {
            user.products[drawItemIndex].quantity += 1;
        }

        user.money -= package.cost;
        await user.save();

        res.status(200).json({ _id: drawItem.productId._id });
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
};
```

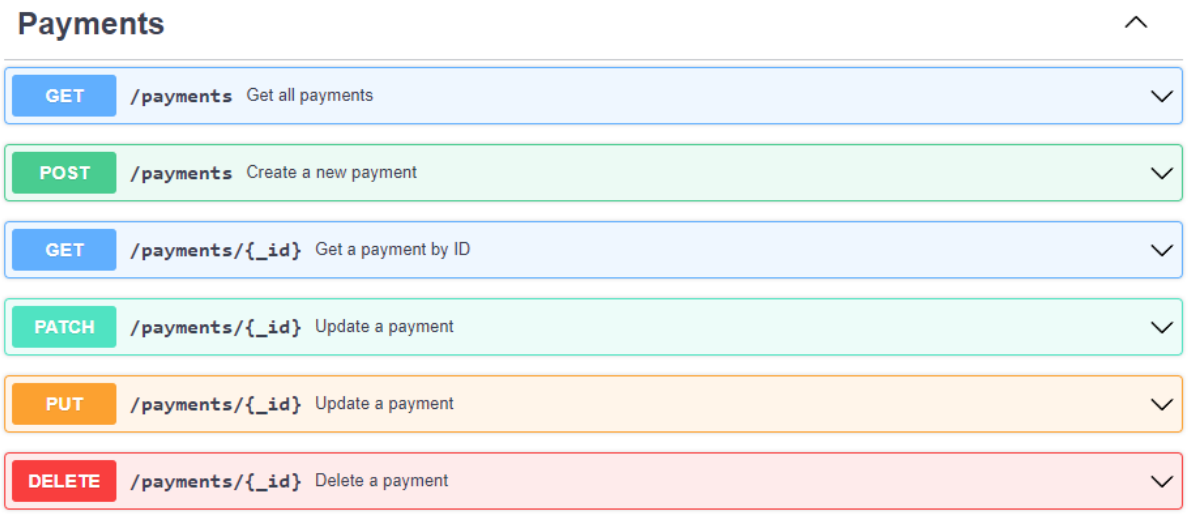
Funkcja **getRandomItem** jest odpowiedzialna za obsługę żądania losowania przedmiotu z paczki. Po otrzymaniu żądania, funkcja pobiera identyfikator paczki z ciała żądania oraz dane użytkownika z obiektu "user" w parametrze.

Funkcja wykonuje następujące kroki:

1. Sprawdza, czy podany identyfikator paczki jest poprawnym identyfikatorem ObjectId w formacie MongoDB. Jeśli identyfikator jest niepoprawny, funkcja zwraca odpowiedź z komunikatem błędu: "No such package".
2. Wyszukuje paczkę o podanym identyfikatorze przy użyciu metody **findById** na modelu Package. Dodatkowo, używa metody "populate" w celu pobrania informacji o produktach związanych z paczką. Jeśli paczka nie istnieje, funkcja zwraca odpowiedź z komunikatem błędu: "No such package".
3. Sprawdza, czy użytkownik już otworzył daną paczkę wcześniej poprzez wyszukanie indeksu otwartej paczki w tablicy openedPackages użytkownika. Jeśli paczka została już otwarta, sprawdza, czy upłynął wystarczający czas oczekiwania (określony przez wartość "cooldown" paczki) od ostatniego otwarcia. Jeśli czas oczekiwania nie minął, funkcja zwraca odpowiedź z komunikatem błędu.
4. Sprawdza, czy użytkownik ma wystarczającą ilość środków (money) do zakupu paczki. Jeśli użytkownik nie ma wystarczających środków, funkcja zwraca odpowiedź z komunikatem błędu: "Insufficient funds".
5. Losuje przedmiot z paczki na podstawie prawdopodobieństwa określonego dla każdego z produktów. Algorytm losowania wykorzystuje metodę proporcjonalnego rozkładu prawdopodobieństwa. Na początku oblicza minimalne prawdopodobieństwo spośród wszystkich produktów w paczce. Następnie skaluje te wartości, aby uzyskać sumę równą 1. Na tej podstawie generuje losową wartość i wybiera odpowiedni przedmiot z paczki zgodnie z rozkładem prawdopodobieństwa.
6. Modyfikuje informacje dotyczące użytkownika, który otwiera paczkę i otrzymuje przedmiot. Jeśli paczka nie została wcześniej otwarta przez użytkownika, dodaje wpis zawierający identyfikator paczki i bieżącą datę do tablicy openedPackages użytkownika. W przeciwnym razie aktualizuje datę otwarcia dla istniejącego wpisu.
7. Sprawdza, czy użytkownik posiada już dany produkt w swojej kolekcji. Jeśli produkt nie istnieje, dodaje nowy wpis zawierający identyfikator produktu i ilość równą 1 do tablicy products użytkownika. W przeciwnym razie zwiększa ilość istniejącego produktu o 1.
8. Odejmuje koszt paczki od środków użytkownika i zapisuje zmiany w bazie danych.
9. Zwraca odpowiedź z kodem statusu 200 i identyfikatorem wylosowanego produktu.

W przypadku wystąpienia błędu podczas procesu losowania, funkcja zwraca odpowiedź z komunikatem błędu.

2.4. Opis funkcjonalności Payments



Najważnejszymi endpoitnami, które opiszemy w tej sekcji są **PATCH /payments/{_id}** oraz **POST /payments**


```
exports.addPayment = async (req, res) => {
  try {
    const { amount } = req.body;
    const {
      user: { _id: userId }
    } = req;
    const payment = await Payment.create(userId, amount);
    res.status(201).json({ message: 'Payment added successfully',
payment });
  } catch (error) {
    res.status(500).json({ error: 'An error occurred' });
  }
};
```

Funkcja **addPayment** jest odpowiedzialna za dodawanie płatności do systemu.

1. Pobierana jest kwota płatności (amount) z ciała żądania, która znajduje się w obiekcie req.body

2. Następnie pobierany jest identyfikator użytkownika (userId) na podstawie aktualnego użytkownika (user) wyciągniętego z żądania.

3. Wykorzystując pobrane wartości, tworzona jest nowa płatność (payment). Przekazywane są identyfikator użytkownika i kwota płatności.

4. Jeśli operacja utworzenia płatności przebiegnie pomyślnie, zwracana jest odpowiedź o statusie 201 (Created) zawierająca wiadomość o sukcesie oraz informacje o nowo utworzonej płatności.

5. W przypadku wystąpienia błędu, blok catch przechwytuje wyjątek i zwraca odpowiedź o statusie 500 (Internal Server Error) z ogólnym komunikatem o błędzie.


```
exports.payForOrder = async (req, res) => {
  try {
    const { user } = req;
    const { _id: orderId } = req.params;

    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }
    const order = await Order.findById(orderId);
    if (!order) {
      return res.status(404).json({ error: 'Order not found' });
    }

    if (order.user.toString() !== user._id.toString()) {
      return res.status(403).json({ error: 'Unauthorized' });
    }

    const payment = await Payment.findOne({ orderId, status:
'pending' });

    if (!payment) {
      return res
        .status(404)
        .json({ error: 'Payment not found or already processed' });
    }

    payment.status = 'completed';
    order.status = 'completed';

    await payment.save();
    await order.save();
```

```
    res
      .status(200)
      .json({ message: 'Payment processed successfully', payment });
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: 'An error occurred' });
  }
};
```

Funkcja **payForOrder** służy do realizacji płatności za zamówienie.

1. W bloku try pobierany jest aktualny użytkownik (user) oraz identyfikator zamówienia (orderId) z żądania.

2. Sprawdzane jest, czy użytkownik istnieje. Jeśli nie, zwracany jest błąd 404 z informacją, że użytkownik nie został znaleziony.

3. Następnie wyszukiwane jest zamówienie na podstawie identyfikatora. Jeśli nie zostanie znalezione, zwracany jest błąd 404 z informacją, że zamówienie nie zostało znalezione.

4. Sprawdzane jest, czy zamówienie należy do danego użytkownika. Jeśli nie, zwracany jest błąd 403 z informacją, że dostęp jest zabroniony.

5. Następnie wyszukiwane jest płatność oczekująca (status: 'pending') na podstawie identyfikatora zamówienia. Jeśli nie zostanie znaleziona żadna płatność lub płatność została już przetworzona, zwracany jest błąd 404 z odpowiednimi informacjami.

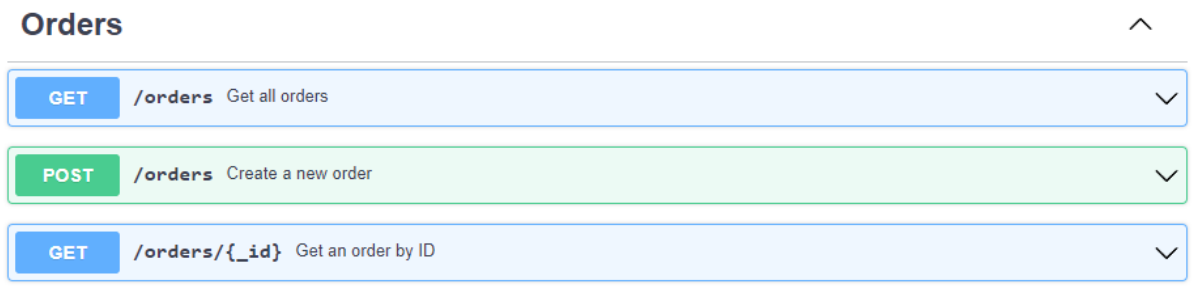
6. Status płatności oraz zamówienia są aktualizowane na "completed".

7. Zmiany są zapisywane w bazie danych.

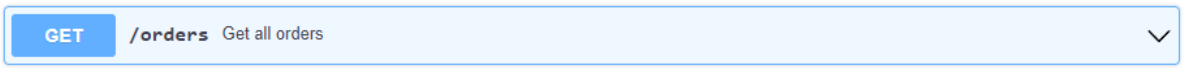
8. Jeśli proces przebiegnie pomyślnie, zwracana jest odpowiedź o statusie 200 (OK) zawierająca wiadomość o sukcesie oraz informacje o przetworzonej płatności.

9. W przypadku wystąpienia błędu, blok catch przechwytuje wyjątek i zwraca odpowiedź o statusie 500 (Internal Server Error) z ogólnym komunikatem o błędzie.

2.5. Opis funkcjonalności Orders



Opiszemy w tej sekcji endpointy **GET /orders** i **POST /orders**




```
const getOrders = async (req, res) => {
  try {
    const orders = await Order.find().populate('items.product');
    res.status(200).json(orders);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
```

Funkcja **getOrders** służy do pobierania wszystkich zamówień z bazy danych

1. W bloku try wykonuje się zapytanie do bazy danych za pomocą metody find() na modelu Order. Metoda ta zwraca wszystkie zamówienia zapisane w bazie danych.

2. Za pomocą metody populate(items.product') wykonywane jest populowanie danych dla pola items.product. Oznacza to, że dla każdego zamówienia pobierane są informacje o powiązanym produkcie.

3. Jeśli operacja pobierania danych przebiegnie pomyślnie, zwracana jest odpowiedź o statusie 200 (OK) zawierająca wszystkie zamówienia w formacie JSON.

4. W przypadku wystąpienia błędu, blok catch przechwytuje wyjątek i zwraca odpowiedź o statusie 500 (Internal Server Error) z komunikatem o błędzie.


```
const addOrder = async (req, res) => {
  const { user } = req;
  const { items } = req.body;

  const session = await mongoose.startSession();

  try {
    session.startTransaction();

    // Sprawdzenie, czy użytkownik posiada uzupełnione dane
    const { street, city, postalCode, country } =
      user.orderInfo.shippingAddress;
    const { contactNumber } = user.orderInfo;
    if (!street || !city || !postalCode || !country ||
!contactNumber) {
      return res
        .status(400)
        .json({ error: 'User shipping information is incomplete' });
    }

    // Sprawdzenie, czy wszystkie przedmioty są dostępne w
user.products
    const orderedProductIds = items.map((item) =>
item.productId.toString());
    for (let id of orderedProductIds) {
      const orderedItem = items.find(
        (item) => item.productId.toString() === id
      );
      const userProduct = user.products.find(
        (product) => product.productId.toString() === id
      );

      if (!userProduct) {
        await session.abortTransaction();
        session.endSession();
        return res
          .status(400)
          .json({ error: 'Some ordered products are not owned by the
user' });
      }
```

```
      if (orderedItem.quantity > userProduct.quantity) {
        await session.abortTransaction();
        session.endSession();
        return res.status(400).json({
          error:
            'Some ordered products are not available in the requested
quantity'
        });
      }
    }

    // Aktualizacja ilości zamówionych przedmiotów w user.products
    for (let orderedItem of items) {
      const productId = orderedItem.productId.toString();
      const quantity = orderedItem.quantity;

      const userProduct = user.products.find(
        (product) => product.productId.toString() === productId
      );

      if (quantity === userProduct.quantity) {
        user.products = user.products.filter(
          (product) => product.productId.toString() !== productId
        );
      } else {
        userProduct.quantity -= quantity;
      }
    }

    await user.save();

    const orderedItems = items.map((item) => {
      return {
        product: item.productId,
        quantity: item.quantity
      };
    });

    const order = new Order({
      user: user._id,
      items: orderedItems,
      shippingAddress: {
        street: street,
        city: city,
```

```
      postalCode: postalCode,
      country: country
    },
    contactNumber: contactNumber,
    status: 'pending'
  });

  await order.save();
  let totalPrice = 0;

  for (let item of items) {
    const product = await Product.findById(item.productId);
    totalPrice += product.price * item.quantity;
  }

  const payment = new mongoose.model('Payment')({
    userId: user._id,
    orderId: order._id,
    amount: totalPrice,
    status: 'pending'
  });

  await payment.save();

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

  // Zakładam że po 5 min jeśli nie opłacone to anuluję zamówienie
  setTimeout(async () => {
    const updatedPayment = await mongoose
      .model('Payment')
      .findById(payment._id);

    if (
      updatedPayment.status === 'pending' ||
      updatedPayment.status === 'canceled'
    ) {
      // Anulowanie zamówienia i płatności
      const session = await mongoose.startSession();
      session.startTransaction();

      await mongoose
        .model('Order')
```

```
          .findByIdAndUpdate(order._id, { status: 'canceled' });
        await mongoose
          .model('Payment')
          .findByIdAndUpdate(payment._id, { status: 'canceled' });

        // Cofnięcie zmian w user.products
        for (let orderedItem of items) {
          const productId = orderedItem.productId.toString();
          const quantity = orderedItem.quantity;

          const userProduct = user.products.find(
            (product) => product.productId.toString() === productId
          );

          if (!userProduct) {
            await session.abortTransaction();
            session.endSession();
            return res.status(400).json({
              error: 'Some ordered products are not owned by the
user'
            });
          }

          userProduct.quantity += quantity;
        }

        await user.save();

        await session.commitTransaction();
        session.endSession();
      }
    }, 5 * 60 * 1000);

    res
      .status(200)
      .json({ message: 'Order added successfully', orderId: order._id
});
  } catch (error) {
    await session.abortTransaction();
    session.endSession();
    console.log(error);
    res.status(500).json({ error: error.message });
  }
};
```

Funkcja **addOrder** służy do dodawania nowego zamówienia do bazy danych.

1. Na początku funkcji pobierane są informacje o użytkowniku z obiektu żądania (req.user) oraz informacje o zamówieniu z ciała żądania (req.body.items).

2. Używamy transakcji udostępnionej poprzez MongoDB przy wykorzystaniu metody startSession()

3. W bloku try sprawdzane są warunki dotyczące użytkownika, takie jak adres dostawy i numer kontaktowy. Jeśli którekolwiek z pól jest puste, zwracany jest błąd z odpowiednim komunikatem.

4. Następnie sprawdzane jest, czy wszystkie zamówione produkty są dostępne w ekwipuntku użytkownika. Iteruje się przez zamówione produkty i porównuje się je z produktami użytkownika. Jeśli któryś z zamówionych produktów nie jest własnością użytkownika lub nie jest dostępny w żądanej ilości, transakcja jest anulowana, a odpowiedni błąd jest zwracany w odpowiedzi.

5. Jeśli wszystkie produkty są dostępne, aktualizowane są ilości zamówionych produktów w ekwipunku użytkownika. Jeśli zamówiony produkt jest ostatnim na stanie, jest usuwany z ekwipunku użytkownika. W przeciwnym razie zmniejszana jest ilość produktu.

6. Tworzone są obiekty zamówienia i płatności na podstawie informacji przekazanych w żądaniu a następnie zapisujemy je do bazy za pomocą save().

7. Obliczana jest łączna cena zamówienia na podstawie cen produktów i ich ilości.

8. Transakcja jest zatwierdzana za pomocą metody commitTransaction()

9. Ustawiany jest timeout na 5 minut, który sprawdza status płatności po upływie czasu. Jeśli płatność nadal jest oznaczona jako **pending** lub **canceled**, zamówienie i płatność są anulowane, a zmiany w ilości zamówionych produktów są cofane.

10. Odpowiedź o statusie 200 (OK) jest zwracana wraz z identyfikatorem zamówienia jeśli wszystko przebiegło pomyślnie.

11. W przypadku wystąpienia błędu, transakcja jest anulowana, i zwracamy odpowiedź o statusie 500 (Internal Server Error).