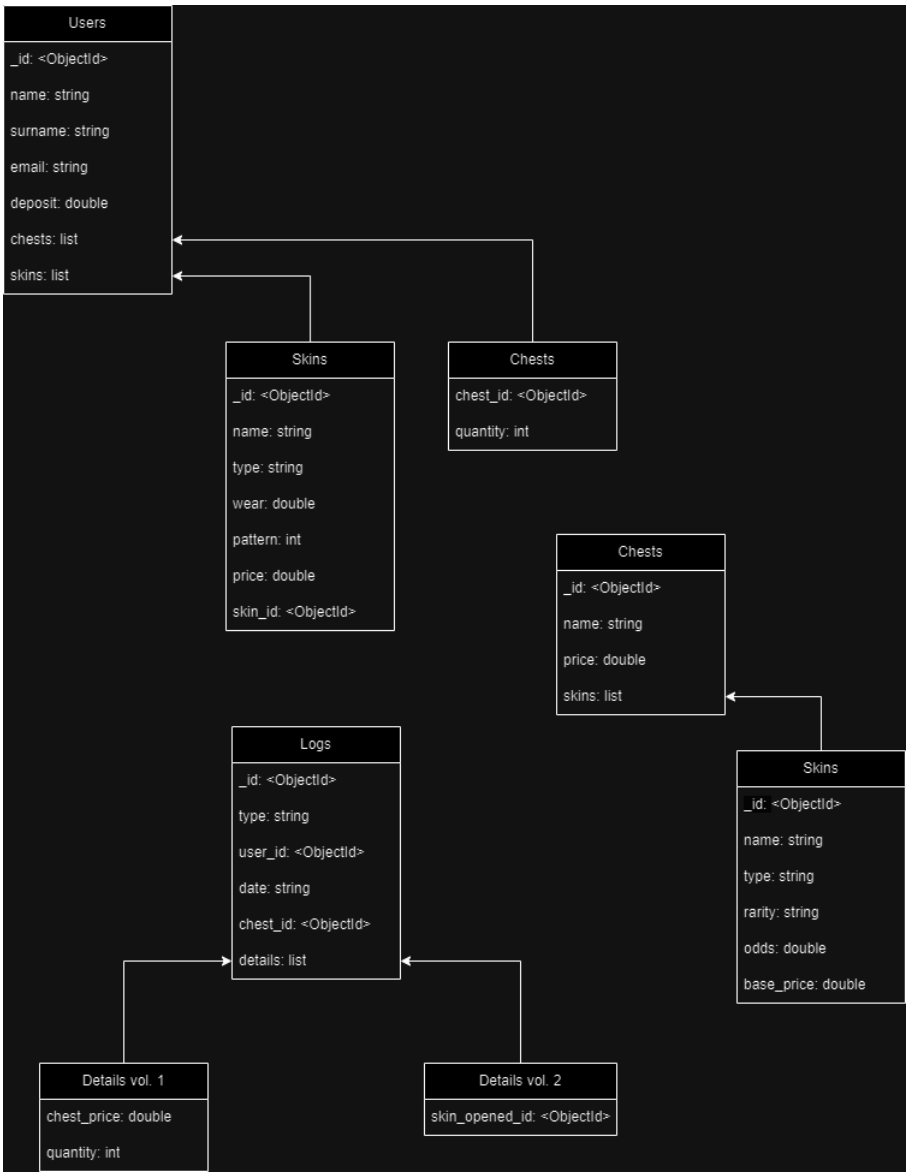
Sprawozdanie

Jakub Sadkiewicz Bartosz Knapik

Temat projektu

Nasz projekt bazy danych został stworzony z myślą o przechowywaniu skórek i ich zdobywaniu poprzez otwieranie specjalnych skrzynek w aplikacjach podobnych do popularnych gier, takich jak Counter-Strike: Global Offensive. Projekt ten ma na celu ułatwienie użytkownikom zarządzania swoimi zasobami, takimi jak skrzynie i skórki, a także śledzenie transakcji związanych z ich zakupem i otwieraniem.

Poglądowy schemat bazy danych



Ciężko przygotować faktyczny diagram bazy nierelacyjnej, dlatego posłużyliśmy się aplikacją drawio, aby przygotować przynajmniej poglądowe rozpisanie kolekcji i dokumentów.

Omówienie modelu danych

Zdecydowaliśmy na trzy kolekcje: Users, Chests oraz Logs. Stosujemy przy nich mechanizm referencji, aby móc zaprezentować relację pomiędzy kolejno:

- _id z kolekcji Users pokrywają się z user_id z kolekcji Logs
- _id z listy Skins z kolekcji Users pokrywają się z skin_opened_id z Details vol. 2
- _id z kolekcji Chests pokrywają się z chest_id z kolekcji Logs
- _id z listy Skins z kolekcji Chests pokrywają się z skin_id w liście Skins w kolekcji Users

Ogromną zaletą nierelacyjnej bazy danych jest możliwość zastosowania ciekawych obiektów w dokumentach. W naszym przypadku w kolekcji Logs trzymamy tak naprawdę dwa rodzaje dokumentów:

- CHEST_PURCHASE
- CHEST_OPEN

Jedyne różnice między nimi to pole type, które posiada odpowiednią wartość string (generowaną na podstawie Enuma) oraz pole details, które może przechowywać obiekt zawierający pola:

- chest_price i quantity

Albo:

- skin_opened_id

Dzięki temu w jednej kolekcji, możemy trzymać dwa podobnej budowy rodzaje dokumentów, bez konieczności tworzenia jakiś referencji do uzupełniających tabel.

Zyskujemy dzięki temu na wydajności, elastyczności i prostocie.

Kolejnym ciekawym rozwiązaniem jest przechowywanie listy skinów oraz listy skrzynek u użytkownika oraz jako osobnej kolekcji. To rozwiązanie pozwala na uniknięcie korzystania z mechanizmu referencji w niektórych przypadkach. Przykładowo użytkownik, często może chcieć wyświetlać listę skinów z Chests, lub swoje własne skiny z Users, a bardzo rzadko agregację tych dwóch kolekcji.

Przykładowe dokumenty z kolekcji

Przykładowy dokument z kolekcji Users


```
{
  "_id": {
    "$oid": "665cfd7c4b58ee76868aca22"
  },
  "name": "Diana",
  "surname": "Prince",
  "email": "diana.prince@example.com",
  "deposit": 6610,
  "chests": [
    {
      "chest_id": {
        "$oid": "665cff924b58ee76868aca27"
      },
      "quantity": 2
    },
    {
      "chest_id": {
        "$oid": "665d023f4b58ee76868aca2a"
      },
      "quantity": 3
    },
    {
      "chest_id": {
        "$oid": "665d034b4b58ee76868aca30"
      },
      "quantity": 5
    }
  ],
  "skins": [
    {
      "_id": {
        "$oid": "665d049e4b58ee76868aca63"
      },
      "name": "Fire Serpent",
      "type": "AK-47",
      "wear": 0.9473571186024513,
      "pattern": 927,
      "price": 198.26,
      "skin_id": {
        "$oid": "665cff924b58ee76868aca25"
      }
    },
    {
      "_id": {
        "$oid": "665d04a74b58ee76868aca71"
      },
      "name": "Mecha Industries",
      "type": "M4A1-S",
      "wear": 0.3466901528866344,
      "pattern": 810,
      "price": 470.4,
      "skin_id": {
        "$oid": "665d034b4b58ee76868aca2e"
      }
    },
    {
      "_id": {
        "$oid": "665d04af4b58ee76868aca80"
      },
      "name": "Cyrex",
      "type": "M4A1-S",
      "wear": 0.07337418194237144,
      "pattern": 992,
      "price": 1189.5,
      "skin_id": {
        "$oid": "665d023f4b58ee76868aca29"
      }
    }
  ]
}
```


Przykładowy dokument z kolekcji Chests

```
{
  "_id": {
    "$oid": "665cff924b58ee76868aca27"
  },
  "name": "Bravo Case",
  "price": 30,
  "skins": [
    {
      "_id": {
        "$oid": "665cff924b58ee76868aca24"
      },
      "name": "Howl",
      "type": "M4A4",
      "rarity": "Covert",
      "odds": 0.3,
      "base_price": 200
    },
    {
      "_id": {
        "$oid": "665cff924b58ee76868aca25"
      },
      "name": "Fire Serpent",
      "type": "AK-47",
      "rarity": "Classified",
      "odds": 0.5,
      "base_price": 100
    },
    {
      "_id": {
        "$oid": "665cff924b58ee76868aca26"
      },
      "name": "Gamma Doppler",
      "type": "Glock-18",
      "rarity": "Covert",
      "odds": 0.2,
      "base_price": 250
    }
  ]
}
```


Przykładowy dokument z kolekcji Logs (Typu CHEST_PURCHASE)


```
{
  "_id": {
    "$oid": "665d04594b58ee76868aca3f"
  },
  "type": "CHEST_PURCHASE",
  "user_id": {
    "$oid": "665cfd7c4b58ee76868aca22"
  },
  "date": "Mon Jun 03 01:46:33 CEST 2024",
  "chest_id": {
    "$oid": "665cff924b58ee76868aca27"
  },
  "details": {
    "chest_price": 30,
    "quantity": 3,
    "description": "Chest purchase",
    "_class": "com.example.mdb_spring_boot.model.DetailPurchase"
  }
}
```


Przykładowy dokument z kolekcji Logs (Typu OPEN_CHEST)


```
{
  "_id": {
    "$oid": "665d04a74b58ee76868aca72"
  },
  "type": "CHEST_OPEN",
  "user_id": {
    "$oid": "665cfd7c4b58ee76868aca22"
  },
  "date": "Mon Jun 03 01:47:51 CEST 2024",
  "chest_id": {
    "$oid": "665d034b4b58ee76868aca30"
  },
  "details": {
    "skin_opened_id": {
      "$oid": "665d04a74b58ee76868aca71"
    },
    "description": "Opened skin",
    "_class": "com.example.mdb_spring_boot.model.DetailOpen"
  }
}
```

Walidacja danych w bazie danych

W naszym projekcie postanowiliśmy skorzystać z bardzo pomocnego narzędzia jakim są Schemy. Wykorzystuje się je, aby zapewnić walidację danych w bazie danych MongoDB. Jest to bardzo użyteczne, ponieważ ogromne możliwości jakie oferują nierelacyjne bazy danych, mogą okazać się mieczem obosiecznym ze względu na zbyt dużą dowolność wprowadzanych danych.

```
const UserSchema = {
  $jsonSchema: {
    bsonType: 'object',
    required: [
      'name',
      'surname',
      'email',
      'deposit',
      'chests',
      'skins'
    ],
    properties: {
      name: {
        bsonType: 'string',
        description: 'Must be a string',
        pattern: '^.{0,20}$'
      },
      surname: {
        bsonType: 'string',
        description: 'Must be a string',
        pattern: '^.{0,20}$'
      },
      email: {
        bsonType: 'string',
        pattern: '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,}$'
      },
      deposit: {
        bsonType: 'double',
        minimum: 0
      },
      chests: {
        bsonType: 'array',
        items: {
          bsonType: 'object',
          required: [
            'chest_id',
            'quantity'
          ],
          properties: {
            chest_id: {
              bsonType: 'objectId'
            },
            quantity: {
              bsonType: 'int',
              minimum: 0
            }
          }
        }
      },
      skins: {
        bsonType: 'array',
        items: {
          bsonType: 'object',
          required: [
            '_id',
            'name',
            'type',
            'wear',
            'pattern',
            'price',
            'skin_id'
          ],
          properties: {
            _id: {
              bsonType: 'objectId'
            },
            name: {
              bsonType: 'string',
              description: 'Must be a string'
            },
            type: {
              bsonType: 'string',
              description: 'Must be a string'
            },
            wear: {
              bsonType: 'double',
              minimum: 0,
              maximum: 1
            },
            pattern: {
              bsonType: 'int',
```

```
              minimum: 0
            },
            price: {
              bsonType: 'double',
              minimum: 0
            },
            skin_id: {
              bsonType: 'objectId'
            }
          }
        }
      }
    }
  }
}

const ChestSchema = {
  $jsonSchema: {
    bsonType: 'object',
    required: [
      'name',
      'price',
      'skins'
    ],
    properties: {
      name: {
        bsonType: 'string',
        description: 'Must be a string',
        pattern: '^.{0,100}$'
      },
      price: {
        bsonType: 'double',
        minimum: 0
      },
      skins: {
        bsonType: 'array',
        items: {
          bsonType: 'object',
          required: [
            '_id',
            'name',
            'type',
            'rarity',
            'odds',
            'base_price'
          ],
          properties: {
            _id: {
              bsonType: 'objectId'
            },
            name: {
              bsonType: 'string',
              description: 'Must be a string',
              pattern: '^.{0,100}$'
            },
            type: {
              bsonType: 'string',
              description: 'Must be a string',
              pattern: '^.{0,100}$'
            },
            rarity: {
              bsonType: 'string',
              description: 'Must be a string',
              pattern: '^.{0,100}$'
            },
            odds: {
              bsonType: 'double',
              minimum: 0,
              maximum: 1
            },
            base_price: {
              bsonType: 'double',
              minimum: 0
            }
          }
        }
      }
    }
  }
}

const LogSchema = {
  $jsonSchema: {
    bsonType: 'object',
    required: [
      'type',
      'user_id',
      'date',
      'chest_id',
      'details'
```

```
      ],
      properties: {
        type: {
          bsonType: 'string',
          'enum': [
            'CHEST_PURCHASE',
            'CHEST_OPEN'
          ],
          description: 'Must be either CHEST_PURCHASE or CHEST_OPEN'
        },
        user_id: {
          bsonType: 'objectId'
        },
        date: {
          bsonType: 'string',
          description: 'Must be a string'
        },
        chest_id: {
          bsonType: 'objectId'
        },
        details: {
          bsonType: 'object',
          oneOf: [
            {
              bsonType: 'object',
              properties: {
                chest_price: {
                  bsonType: 'double',
                  minimum: 0
                },
                quantity: {
                  bsonType: 'int',
                  minimum: 1
                }
              },
              required: [
                'chest_price',
                'quantity'
              ]
            },
            {
              bsonType: 'object',
              properties: {
                skin_opened_id: {
                  bsonType: 'objectId'
                }
              },
              required: [
                'skin_opened_id'
              ]
            }
          ]
        }
      }
    }
  }
```

Zaimplemenetowane operacje transakcyjne

Niżej opisane operacje transakcyjne znajdują się w klasie UserService

Kupno skrzynki przez użytkownika

- Wymagało walidacji czy użytkownik posiada wystarczające środki na zakup skrzynki.
- Ponieważ jednocześnie wykonujemy update dokumentu danego usera oraz tworzymy nowy log skorzystaliśmy z mechanizmu transakcji poprzez dodanie adnotacji przed definicją funkcji. To rozwiązanie powoduje, że jeśli chociaż jedno zapisanie danych do repozytorium się niepowiedzie to nastąpi rollback wszystkich zmian.

Otworzenie skrzynki przez użytkownika

- Wymagało kontroli czy skrzynka istnieje oraz czy jest posiadana przez użytkownika.
- Na podstawie ogólnych danych o skórkach, generujemy konkretne skórki z ceną zależną od losowych parametrów
- Analogiczne wykorzystanie mechanizmu transakcji jak przy kupnie skrzynki.

Operacje o charakterze raportującym

Oprócz standardowego wypisywanie danych zaimplementowaliśmy:

Zapytanie: Skin Report

Zapytanie to przeprowadza łączenie kolekcji Users z kolekcją Chests, aby przygotować raport postaci: Skin ID (z kolekcji Chests), Count, Total Spent

Czyli dla każdego rodzaju skina zlicza ile razy został wylosowany oraz sumuje cenę jaką wydano na skrzynki

```
public List<JsonObject> generateSkinReport() {
    MongoCollection<Document> collection = mongoTemplate.getCollection("users");
    AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
```

```
        new Document("$unwind", "$skins"),
        new Document("$lookup", new Document("from", "chests")
            .append("let", new Document("skinId", "$skins.skin_id"))
            .append("pipeline", Arrays.asList(
                    new Document("$unwind", "$skins"),
                    new Document("$match", new Document("$expr", new Document("$eq", Arrays.asList("$skins._id", "$$skinId"))))
            ))
            .append("as", "chestSkins")),
        new Document("$unwind", "$chestSkins"),
        new Document("$group", new Document("_id", "$skins.skin_id")
            .append("count", new Document("$sum", 1))
            .append("totalSpent", new Document("$sum", "$chestSkins.price")))
    ));

    return convertDocumentsToJson(result);
}
```

Wynik zapytania:

Skin Report

- Skin ID: 665d034b4b58ee76868aca2e, Count: 1, Total Spent: 150.0
- Skin ID: 665d023f4b58ee76868aca29, Count: 1, Total Spent: 100.0
- Skin ID: 665cff924b58ee76868aca25, Count: 3, Total Spent: 90.0
- Skin ID: 665cff924b58ee76868aca24, Count: 2, Total Spent: 60.0
- Skin ID: 665d034b4b58ee76868aca2f, Count: 2, Total Spent: 300.0

Zapytanie: Users Total Skins Values.

Zapytanie to dla każdego z użytkowników sumuje wartości ich skinów oraz wypisuje je w formacie: Name, Total Value.

```
public List<JsonObject> generateUsersTotalSkinsValues() {
    AggregateIterable<Document> result = mongoTemplate.getCollection("users").aggregate(Arrays.asList(
        new Document("$unwind", "$skins"),
        new Document("$group", new Document("_id", "$_id")
                .append("name", new Document("$first", "$name"))
                .append("totalValue", new Document("$sum", "$skins.price"))),
        new Document("$sort", new Document("totalValue", -1))
    ));

    return convertDocumentsToJson(result);
}
```

Wynik zapytania:

Users Total Skins Values

- Name: Alice, Total Value: 2046.4
- Name: Diana, Total Value: 1858.1599999999999
- Name: Bob, Total Value: 1421.8600000000001
- Name: Eve, Total Value: 609.39

Zapytanie: User Total Spending:

Zapytanie to zwraca dla danego usera sumę wydanych przez niego pieniędzy na skrzynki.

```
public List<JsonObject> generateUserTotalSpending(ObjectId userId) {
    MongoCollection<Document> collection = mongoTemplate.getCollection("logs");
    AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
        new Document("$match", new Document("user_id", userId)),
        new Document("$match", new Document("type", "CHEST_PURCHASE")),
        new Document("$project", new Document("user_id", 1).append("type", 1)
            .append("totalAmount", new Document("$multiply", Arrays.asList("$details.chest_price", "$details.quantity")))),
        new Document("$group", new Document("_id", "$user_id").append("totalAmount", new Document("$sum", "$totalAmount")))
    ));

    return convertDocumentsToJson(result);
}
```

Wynik zapytania:

User Total Spending

UserId: 665cfd7c4b58ee76868aca22, Total Amount: 1390.0

Pełny kod zapytań raportujących można znaleźć w klasie ReportService

Wykorzystane technologie back-end

Podczas projektowania backendu dla naszej aplikacji do zarządzania skrzynkami i skórkami zdecydowaliśmy się na wykorzystanie technologii Spring Boot. Jest to jedno z najbardziej popularnych i wszechstronnych narzędzi w ekosystemie Java, co czyni go idealnym wyborem dla tego typu aplikacji.

W springu pracowało nam się bardzo przyjemnie. Niezwykle prosto jest tworzyć modele odpowiadające dokumentom z nierelacyjnej bazy danych. Dodatkowo mogliśmy tworzyć modele obiektów nie będących faktycznymi dokumentami, a jedynie elementami dokumentów. Pomimo początkowych obaw pisanie back-endu do nierelacyjnej bazy danych nie było takie straszne.

Oprócz tego mogliśmy skorzystać z możliwości tworzenia testów do bazy danych. Spring, a przede wszystkim java, dają tutaj ogromne możliwości.

Poniej znajdują się poszczególne klasy obsługujące naszą bazę danych w technologii Spring Boot

Moduł `model`

Skin.java

Opis: Model reprezentujący skórkę. Zawiera pola takie jak nazwa, rzadkość, oraz szansa na zdobycie skórki.

```
package com.example.mdb_spring_boot.model;

import org.bson.types.ObjectId;
import org.springframework.data.mongodb.core.mapping.Field;

import java.util.UUID;

public class Skin {
    private ObjectId id;
    private String name;
    private String type;
    private String rarity;
    private double odds;
    @Field("base_price")
    private double basePrice;

    public Skin(String name, String type, String rarity, double odds, double basePrice) {
        this.id = new ObjectId();
        this.name = name;
        this.rarity = rarity;
        this.type = type;
        this.odds = odds;
        this.basePrice = basePrice;
    }

    public ObjectId getId(){
        return id;
    }

    public String getName(){
        return name;
    }

    public String getType(){
        return type;
    }

    public String getRarity(){
        return rarity;
    }

    public double getOdds(){
        return odds;
    }

    public double getBasePrice(){
        return basePrice;
    }
}
```

User.java

Opis: Model reprezentujący użytkownika. Zawiera pola takie jak imię, nazwisko, email, depozyt oraz listy skrzyń i skór należących do użytkownika.

```java
package com.example.mdb_spring_boot.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.ArrayList;
import java.util.List;

@Document("users")
public class User {
    @Id
    private String id;

    private String name;
    private String surname;
    private String email;
    private double deposit;

    private List<UserChest> chests = new ArrayList<>();

    private List<UserSkin> skins = new ArrayList<>();

    public User(String name, String surname, String email, double deposit){
        super();
        this.name = name;
        this.surname = surname;
        this.email = email;
        this.deposit = deposit;
    }

    public void addSkin(UserSkin skin){
        skins.add(skin);
    }

    public void addChest(UserChest chest){
        for (UserChest ownedChest : this.chests) {
            if (chest.getChestId().equals(ownedChest.getChestId())) {
                ownedChest.setQuantity(ownedChest.getQuantity() + chest.getQuantity());
                return;
            }
        }
        this.chests.add(chest);
    }

    public void setDeposit(double deposit){
        this.deposit = deposit;
    }

    public void addToDeposit(double money){
        this.deposit += money;
    }

    public void removeFromDeposit(double money){
        this.deposit -= money;
    }

    public void afterOpeningChest(UserChest chest){
        if (chest.getQuantity() == 1)
            chests.remove(chest);
        else
            chest.setQuantity(chest.getQuantity() - 1);
    }

    public void setChests(List<UserChest> chests) {
        this.chests = chests;
    }

    public void setSkins(List<UserSkin> skins) {
        this.skins = skins;
    }

    public String getId(){
        return id;
    }

    public String getName(){
        return name;
    }

    public String getSurname(){
        return surname;
    }

    public String getEmail(){
        return email;
    }

    public double getDeposit(){
        return deposit;
```

```
    }

    public List<UserChest> getChests(){
        return chests;
    }

    public List<UserSkin> getSkins() {
        return skins;
    }
}
```

UserChest.java

Opis: Model reprezentujący skrzynię należącą do użytkownika. Zawiera pola chestId oraz quantity, które przechowują odpowiednio identyfikator skrzyni oraz ilość posiadanych skrzyń.

```
package com.example.mdb_spring_boot.model;

import org.bson.types.ObjectId;
import org.springframework.data.mongodb.core.mapping.Field;

public class UserChest {

    @Field("chest_id")
    private ObjectId chestId;
    private int quantity;

    public UserChest(){}

    public UserChest(ObjectId chestId){
        this(chestId, 0);
    }

    public UserChest(ObjectId chestId, int quantity){
        this.chestId = chestId;
        this.quantity = quantity;
    }

    public ObjectId getChestId(){
        return chestId;
    }

    public int getQuantity(){
        return quantity;
    }

    public void setQuantity(int quantity){
        this.quantity = quantity;
    }

    public void setChestId(ObjectId chestId){
        this.chestId = chestId;
    }
}
```

UserSkin.java

Opis: Model reprezentujący skórkę należącą do użytkownika. Zawiera pola takie jak nazwa, typ, zużycie, wzór, cena oraz UUID skórki.

```
package com.example.mdb_spring_boot.model;

import org.bson.types.ObjectId;
import org.springframework.data.mongodb.core.mapping.Field;

import java.util.UUID;

public class UserSkin {
    private ObjectId id;
    private String name;
    private String type;
    private double wear;
    private int pattern;
    private double price;
    @Field("skin_id")
    private ObjectId skinId;

    public UserSkin(String name, String type, double wear, int pattern, double price, ObjectId skinId){
        this.id = new ObjectId();
        this.name = name;
        this.type = type;
        this.wear = wear;
        this.pattern = pattern;
```

```
        this.price = price;
        this.skinId = skinId;
    }

    public ObjectId getId(){
        return id;
    }

    public String getName(){
        return name;
    }

    public String getType(){
        return type;
    }

    public double getWear(){
        return wear;
    }

    public int getPattern(){
        return pattern;
    }

    public double getPrice(){
        return price;
    }

    public ObjectId getSkinId(){
        return skinId;
    }
}
```

Chest.java

Opis: Model reprezentujący skrzynię. Zawiera pola takie jak nazwa skrzyni, cena oraz lista skórek dostępnych w skrzyni.

```
package com.example.mdb_spring_boot.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.List;

@Document("chests")
public class Chest {
    @Id
    private String id;

    private String name;
    private double price;
    private final List<Skin> skins;

    public Chest(String name, double price, List<Skin> skins){
        this.name = name;
        this.price = price;
        this.skins = skins;
    }

    public void addSkin(Skin skin){
        this.skins.add(skin);
    }
    public void removeSkin(Skin skin){
        this.skins.remove(skin);
    }

    public String getId(){
        return id;
    }

    public String getName(){
        return name;
    }

    public double getPrice(){
        return price;
    }

    public List<Skin> getSkins(){
        return skins;
    }
}
```

Log.java

Opis: Model reprezentujący log. Zawiera pola takie jak typ logu, identyfikator użytkownika, data, identyfikator skrzyni oraz szczegóły logu.

```java
package com.example.mdb_spring_boot.model;

import org.bson.types.ObjectId;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

import java.util.ArrayList;
import java.util.List;

@Document("logs")
public class Log {
    @Id
    private String id;

    private LogType type;
    @Field("user_id")
    private ObjectId userId;
    private String date;
    @Field("chest_id")
    private ObjectId chestId;
    private Detail details;

    public Log(){}

    public Log(LogType type, ObjectId userId, String date, ObjectId chestId, Detail detail){
        this.type = type;
        this.userId = userId;
        this.date = date;
        this.chestId = chestId;
        this.details = detail;
    }

    public String getId(){
        return id;
    }

    public LogType getType(){
        return type;
    }

    public ObjectId getUserId(){
        return userId;
    }

    public String getDate(){
        return date;
    }

    public ObjectId getChestId(){
        return chestId;
    }

    public Detail getDetail(){
        return details;
    }

    public void setDate(String date){
        this.date = date;
    }
}
```

LogType.java

Opis: Enum definiujący typy logów (CHEST_PURCHASE, CHEST_OPEN).

```java
package com.example.mdb_spring_boot.model;

public enum LogType {
    CHEST_PURCHASE,
    CHEST_OPEN
}
```

Detail.java

Opis: Model bazowy dla szczegółów logów. Używany jako klasa nadrzędna dla DetailOpen i DetailPurchase.

```java
package com.example.mdb_spring_boot.model;

public abstract class Detail {
    private final String description;

    public Detail(String description){
```

```
        this.description = description;
    }

    public String getDescription(){
        return description;
    }

    public abstract String getDetailType();
}
```

DetailOpen.java

Opis: Model reprezentujący szczegóły logu dotyczącego otwarcia skrzyni. Zawiera pole skinOpenedId, które przechowuje UUID otwartej skórki.

```
package com.example.mdb_spring_boot.model;

import org.bson.types.ObjectId;
import org.springframework.data.mongodb.core.mapping.Field;

import java.util.UUID;

public class DetailOpen extends Detail {
    @Field("skin_opened_id")
    private final ObjectId skinId;

    public DetailOpen(ObjectId skinId, String description){
        super(description);
        this.skinId = skinId;
    }

    public ObjectId getSkinId(){
        return skinId;
    }

    @Override
    public String getDetailType(){
        return "CHEST_OPEN";
    }

    @Override
    public String toString(){
        return "DetailOpen{" +
                "skinId=" + skinId +
                ", description='" + getDescription() + '\'' +
                '}';
    }
}
```

DetailPurchase.java

Opis: Model reprezentujący szczegóły logu dotyczącego zakupu skrzyni. Zawiera pola chestPrice i quantity, które przechowują odpowiednio cenę skrzyni i ilość zakupionych skrzyń.

```
package com.example.mdb_spring_boot.model;

import org.springframework.data.mongodb.core.mapping.Field;

public class DetailPurchase extends Detail {
    @Field("chest_price")
    private final double chestPrice;
    private final int quantity;

    public DetailPurchase(double chestPrice, int quantity, String description){
        super(description);
        this.chestPrice = chestPrice;
        this.quantity = quantity;
    }

    public double getChestPrice(){
        return chestPrice;
    }

    public int getQuantity(){
        return quantity;
    }

    @Override
    public String getDetailType(){
        return "CHEST_PURCHASE";
    }

    @Override
    public String toString(){
        return "DetailPurchase{" +
```

```
                "chestPrice=" + chestPrice +
                ", quantity=" + quantity +
                ", description='" + getDescription() + '\'' +
                '}';
    }
}
```

Moduł `repository`

UserRepository.java

Opis: Interfejs repozytorium dla operacji CRUD na kolekcji użytkowników. Dziedziczy po MongoRepository.

```
package com.example.mdb_spring_boot.repository;

import com.example.mdb_spring_boot.model.User;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends MongoRepository<User, String> {
}
```

ChestRepository.java

Opis: Interfejs repozytorium dla operacji CRUD na kolekcji skrzyń. Dziedziczy po MongoRepository.

```
package com.example.mdb_spring_boot.repository;

import com.example.mdb_spring_boot.model.Chest;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ChestRepository extends MongoRepository<Chest, String> {
}
```

LogRepository.java

Opis: Interfejs repozytorium dla operacji CRUD na kolekcji logów. Dziedziczy po MongoRepository.

```
package com.example.mdb_spring_boot.repository;

import com.example.mdb_spring_boot.model.Log;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface LogRepository extends MongoRepository<Log, String> {
}
```

Moduł `service`

UserService.java

Opis: Serwis zarządzający logiką biznesową dotyczącą użytkowników. Oferuje metody do dodawania nowych użytkowników, dodawania skrzyń i skór do użytkowników oraz pobierania szczegółów użytkowników. Wykorzystaliśmy adnotację `@Transactional`, zeby zapobiec rozspójnieniu dazy danych.

```
package com.example.mdb_spring_boot.service;

import com.example.mdb_spring_boot.exception.InsufficientFundsException;
import com.example.mdb_spring_boot.model.*;
import com.example.mdb_spring_boot.repository.ChestRepository;
import com.example.mdb_spring_boot.repository.LogRepository;
import com.example.mdb_spring_boot.repository.UserRepository;
import com.example.mdb_spring_boot.util.DrawingMachine;
import org.bson.types.ObjectId;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonObjectBuilder;
import javax.json.JsonArrayBuilder;

import java.util.Date;
```

```
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@Service
public class UserService {
    private final UserRepository userRepository;
    private final LogRepository logRepository;
    private final ChestRepository chestRepository;

    private final DrawingMachine drawingMachine = new DrawingMachine();

    @Autowired
    public UserService(UserRepository userRepository, LogRepository logRepository, ChestRepository chestRepository){
        this.userRepository = userRepository;
        this.logRepository = logRepository;
        this.chestRepository = chestRepository;
    }

    public User addUser(User user){
        return userRepository.save(user);
    }

    public void removeUser(User user){
        userRepository.delete(user);
    }

    @Transactional
    public User addChestToUser(String userId, UserChest userChest) {
        Optional<User> optionalUser = userRepository.findById(userId);
        if (optionalUser.isPresent()) {
            User user = optionalUser.get();

            Optional<Chest> optionalChest = chestRepository.findAll().stream()
                    .filter(c -> c.getId().equals(userChest.getChestId().toString()))
                    .findFirst();

            if (optionalChest.isPresent()) {
                Chest chest = optionalChest.get();
                double totalPrice = userChest.getQuantity() * chest.getPrice();
                if (user.getDeposit() >= totalPrice) {
                    user.addChest(userChest);
                    user.removeFromDeposit(totalPrice);

                    Detail detail = new DetailPurchase(chest.getPrice(), userChest.getQuantity(), "Chest purchase");
                    Log log = new Log(LogType.CHEST_PURCHASE, new ObjectId(userId), new Date().toString(), userChest.getChestId(), detail);
                    logRepository.save(log);

                    return userRepository.save(user);
                }
                else{
                    throw new InsufficientFundsException("User does not have enough money to buy chest or chests");
                }
            }
            else {
                throw new RuntimeException("Chest does not exist");
            }
        } else {
            throw new RuntimeException("User not found");
        }
    }

    public User addSkinToUser(String userId, UserSkin skin) {
        Optional<User> optionalUser = userRepository.findById(userId);
        if (optionalUser.isPresent()) {
            User user = optionalUser.get();
            user.addSkin(skin);
            return userRepository.save(user);
        } else {
            throw new RuntimeException("User not found");
        }
    }

    @Transactional
    public User openChest(String userId, String chestId) {
        Optional<User> optionalUser = userRepository.findById(userId);
        if (optionalUser.isPresent()) {
            User user = optionalUser.get();

            // Tutaj musimy znaleźć skrzynkę użytkownika
            Optional<UserChest> optionalUserChest = user.getChests().stream()
                    .filter(c -> c.getChestId().toString().equals(chestId))
                    .findFirst();

            if (optionalUserChest.isPresent()) {
                UserChest userChest = optionalUserChest.get();
                Optional<Chest> optionalChest = chestRepository.findAll().stream()
                        .filter(c -> c.getId().toString().equals(chestId))
                        .findFirst();
```

```
                if (optionalChest.isPresent()){
                    Chest chest = optionalChest.get();
                    UserSkin skin = drawingMachine.getRandomSkin(chest);
                    user.addSkin(skin);

                    // Tworzymy log otwierania skrzynki
                    Detail detail = new DetailOpen(skin.getId(), "Opened skin");
                    Log log = new Log(LogType.CHEST_OPEN, new ObjectId(userId), new Date().toString(), new ObjectId(chestId), detail);
                    logRepository.save(log);

                    user.afterOpeningChest(userChest);

                    // Zapisujemy zmiany w użytkowniku
                    return userRepository.save(user);
                }
                else{
                    throw new RuntimeException("Chest does not exist");
                }
            } else {
                throw new RuntimeException("Chest not found for user");
            }
        } else {
            throw new RuntimeException("User not found");
        }
    }

    public User getUserById(String userId) {
        Optional<User> optionalUser = userRepository.findById(userId);
        if (optionalUser.isPresent()) {
            return optionalUser.get();
        } else {
            throw new RuntimeException("User not found");
        }
    }

    public List<JsonObject> getAllUsers() {
        List<User> users = userRepository.findAll();
        return users.stream()
                .map(this::convertUserToJson)
                .collect(Collectors.toList());
    }

    private JsonObject convertUserToJson(User user) {
        JsonObjectBuilder jsonBuilder = Json.createObjectBuilder();
        jsonBuilder.add("id", Json.createObjectBuilder().add("chars", user.getId()).add("string", user.getId()).add("valueType", "STRING"));
        jsonBuilder.add("name", user.getName());
        jsonBuilder.add("surname", user.getSurname());
        jsonBuilder.add("email", user.getEmail());
        jsonBuilder.add("deposit", Json.createObjectBuilder().add("chars", String.valueOf(user.getDeposit())).add("string",
String.valueOf(user.getDeposit())).add("valueType", "NUMBER"));

        JsonArrayBuilder chestsBuilder = Json.createArrayBuilder();
        user.getChests().forEach(chest -> {
            JsonObjectBuilder chestBuilder = Json.createObjectBuilder();
            chestBuilder.add("chestId", Json.createObjectBuilder().add("chars", chest.getChestId().toHexString()).add("string",
chest.getChestId().toHexString()).add("valueType", "STRING"));
            chestBuilder.add("quantity", Json.createObjectBuilder().add("chars", String.valueOf(chest.getQuantity())).add("string",
String.valueOf(chest.getQuantity())).add("valueType", "NUMBER"));
            chestsBuilder.add(chestBuilder);
        });
        jsonBuilder.add("chests", chestsBuilder);

        JsonArrayBuilder skinsBuilder = Json.createArrayBuilder();
        user.getSkins().forEach(skin -> {
            JsonObjectBuilder skinBuilder = Json.createObjectBuilder();
            skinBuilder.add("skinId", Json.createObjectBuilder().add("chars", skin.getSkinId().toHexString()).add("string",
skin.getSkinId().toHexString()).add("valueType", "STRING"));
            skinBuilder.add("name", skin.getName());
            skinBuilder.add("type", skin.getType());
            skinBuilder.add("wear", Json.createObjectBuilder().add("chars", String.valueOf(skin.getWear())).add("string",
String.valueOf(skin.getWear())).add("valueType", "NUMBER"));
            skinBuilder.add("pattern", Json.createObjectBuilder().add("chars", String.valueOf(skin.getPattern())).add("string",
String.valueOf(skin.getPattern())).add("valueType", "NUMBER"));
            skinBuilder.add("price", Json.createObjectBuilder().add("chars", String.valueOf(skin.getPrice())).add("string",
String.valueOf(skin.getPrice())).add("valueType", "NUMBER"));
            skinsBuilder.add(skinBuilder);
        });
        jsonBuilder.add("skins", skinsBuilder);

        return jsonBuilder.build();
    }
}
```

ChestService.java

Opis: Serwis zarządzający logiką biznesową dotyczącą skrzyń. Oferuje metody do dodawania nowych skrzyń, dodawania skór do skrzyń oraz pobierania szczegółów skrzyń.

```
package com.example.mdb_spring_boot.service;

import com.example.mdb_spring_boot.model.Chest;
import com.example.mdb_spring_boot.model.Skin;
import com.example.mdb_spring_boot.repository.ChestRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.json.Json;
import javax.json.JsonArrayBuilder;
import javax.json.JsonObject;
import javax.json.JsonObjectBuilder;
import java.util.List;
import java.util.stream.Collectors;

@Service
public class ChestService {
    private final ChestRepository chestRepository;

    @Autowired
    public ChestService(ChestRepository chestRepository){
        this.chestRepository = chestRepository;
    }

    public Chest addChest(Chest chest){
        return chestRepository.save(chest);
    }

    public void removeChest(Chest chest){
        chestRepository.delete(chest);
    }

    public Chest getChestById(String chestId){
        return chestRepository.findById(chestId).orElseThrow(() -> new RuntimeException("Chest not found"));
    }

    public Chest addSkinToChest(String chestId, Skin skin){
        Chest chest = chestRepository.findById(chestId).orElseThrow(() -> new RuntimeException("Chest not found"));
        chest.addSkin(skin);
        return chestRepository.save(chest);
    }

    public List<JsonObject> getAllChests() {
        List<Chest> chests = chestRepository.findAll();
        return chests.stream()
                .map(this::convertChestToJson)
                .collect(Collectors.toList());
    }

    private JsonObject convertChestToJson(Chest chest) {
        JsonObjectBuilder jsonBuilder = Json.createObjectBuilder();
        jsonBuilder.add("id", Json.createObjectBuilder().add("chars", chest.getId()).add("string", chest.getId()).add("valueType", "STRING"));
        jsonBuilder.add("name", chest.getName());
        jsonBuilder.add("price", String.valueOf(chest.getPrice()));

        JsonArrayBuilder skinsBuilder = Json.createArrayBuilder();
        chest.getSkins().forEach(skin -> {
            JsonObjectBuilder skinBuilder = Json.createObjectBuilder();
            skinBuilder.add("name", skin.getName());
            skinBuilder.add("rarity", skin.getRarity());
            skinBuilder.add("odds", Json.createObjectBuilder().add("chars", String.valueOf(skin.getOdds())).add("string",
String.valueOf(skin.getOdds())).add("valueType", "NUMBER"));
            skinsBuilder.add(skinBuilder);
        });
        jsonBuilder.add("skins", skinsBuilder);

        return jsonBuilder.build();
    }
}
```

LogService.java

Opis: Serwis zarządzający logiką biznesową dotyczącą logów. Oferuje metody do dodawania nowych logów oraz pobierania szczegółów logów.

```
package com.example.mdb_spring_boot.service;

import com.example.mdb_spring_boot.model.Detail;
import com.example.mdb_spring_boot.model.DetailOpen;
import com.example.mdb_spring_boot.model.DetailPurchase;
import com.example.mdb_spring_boot.repository.LogRepository;
import org.bson.types.ObjectId;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.mdb_spring_boot.model.Log;

import javax.json.Json;
```

```
import javax.json.JsonObject;
import javax.json.JsonObjectBuilder;

import java.util.Date;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@Service
public class LogService {
    private final LogRepository logRepository;

    @Autowired
    public LogService(LogRepository logRepository) {
        this.logRepository = logRepository;
    }

    public Log addLog(Log log) {
        log.setDate(new Date().toString());
        return logRepository.save(log);
    }

    public void removeLog(Log log){
        logRepository.delete(log);
    }

    public List<JsonObject> getAllLogs() {
        List<Log> logs = logRepository.findAll();
        return logs.stream()
                .map(this::convertLogToJson)
                .collect(Collectors.toList());
    }

    private JsonObject convertLogToJson(Log log) {
        JsonObjectBuilder jsonBuilder = Json.createObjectBuilder();
        jsonBuilder.add("id", log.getId());
        jsonBuilder.add("type", log.getType().toString());
        jsonBuilder.add("userId", log.getUserId().toHexString());
        jsonBuilder.add("date", log.getDate());
        jsonBuilder.add("chestId", log.getChestId().toHexString());
        jsonBuilder.add("details", convertDetailToJson(log.getDetail()));
        return jsonBuilder.build();
    }

    private JsonObject convertDetailToJson(Detail details) {
        JsonObjectBuilder jsonBuilder = Json.createObjectBuilder();
        if (details instanceof DetailPurchase) {
            DetailPurchase detailPurchase = (DetailPurchase) details;
            jsonBuilder.add("chestPrice", String.valueOf(detailPurchase.getChestPrice()));
            jsonBuilder.add("quantity", String.valueOf(detailPurchase.getQuantity()));
            jsonBuilder.add("description", detailPurchase.getDescription());
        } else if (details instanceof DetailOpen) {
            DetailOpen detailOpen = (DetailOpen) details;
            jsonBuilder.add("skinOpenedId", detailOpen.getSkinId().toHexString());
            jsonBuilder.add("description", detailOpen.getDescription());
        }
        return jsonBuilder.build();
    }

    public Log getLogById(String logId) {
        Optional<Log> optionalLog = logRepository.findById(logId);
        if (optionalLog.isPresent()) {
            return optionalLog.get();
        } else {
            throw new RuntimeException("Log not found");
        }
    }
}
```

ReportService.java

Opis: Serwis odpowiedzialny za generowanie raportów na podstawie danych z systemu.

```
package com.example.mdb_spring_boot.service;

import com.mongodb.BasicDBObject;
import com.mongodb.client.AggregateIterable;
import com.mongodb.client.MongoCollection;
import org.bson.Document;
import org.bson.types.ObjectId;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Service;

import javax.json.Json;
import javax.json.JsonArrayBuilder;
import javax.json.JsonObject;
```

```
import javax.json.JsonObjectBuilder;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

@Service
public class ReportService {
    private final MongoTemplate mongoTemplate;

    @Autowired
    public ReportService(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    public List<JsonObject> generateUsersTotalSkinsValues() {
        AggregateIterable<Document> result = mongoTemplate.getCollection("users").aggregate(Arrays.asList(
                new Document("$unwind", "$skins"),
                new Document("$group", new Document("_id", "$_id")
                        .append("name", new Document("$first", "$name"))
                        .append("totalValue", new Document("$sum", "$skins.price"))),
                new Document("$sort", new Document("totalValue", -1))
        ));

        return convertDocumentsToJson(result);
    }


    public List<JsonObject> generateUserTotalSpending(ObjectId userId) {
        MongoCollection<Document> collection = mongoTemplate.getCollection("logs");
        AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
                new Document("$match", new Document("user_id", userId)),
                new Document("$match", new Document("type", "CHEST_PURCHASE")),
                new Document("$project", new Document("user_id", 1).append("type", 1)
                        .append("totalAmount", new Document("$multiply", Arrays.asList("$details.chest_price", "$details.quantity")))),
                new Document("$group", new Document("_id", "$user_id").append("totalAmount", new Document("$sum", "$totalAmount")))
        ));

        return convertDocumentsToJson(result);
    }

    public List<JsonObject> generateSkinReport() {
        MongoCollection<Document> collection = mongoTemplate.getCollection("users");
        AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
                new Document("$unwind", "$skins"),
                new Document("$lookup", new Document("from", "chests")
                        .append("let", new Document("skinId", "$skins.skin_id"))
                        .append("pipeline", Arrays.asList(
                                new Document("$unwind", "$skins"),
                                new Document("$match", new Document("$expr", new Document("$eq", Arrays.asList("$skins._id", "$$skinId"))))
                        ))
                        .append("as", "chestSkins")),
                new Document("$unwind", "$chestSkins"),
                new Document("$group", new Document("_id", "$skins.skin_id")
                        .append("count", new Document("$sum", 1))
                        .append("totalSpent", new Document("$sum", "$chestSkins.price")))
        ));

        return convertDocumentsToJson(result);
    }

    private List<JsonObject> convertDocumentsToJson(Iterable<Document> documents) {
        List<JsonObject> jsonObjects = new ArrayList<>();
        for (Document doc : documents) {
            JsonObjectBuilder builder = Json.createObjectBuilder();
            doc.forEach((key, value) -> {
                builder.add(key, String.valueOf(value));
            });
            jsonObjects.add(builder.build());
        }
        return jsonObjects;
    }
}
```

Moduł `controller`

UserController.java

Opis: Kontroler REST dla zarządzania operacjami związanymi z użytkownikami. Oferuje endpointy do dodawania nowych użytkowników, dodawania skrzyń i skór do użytkowników oraz pobierania szczegółów użytkowników.

```
package com.example.mdb_spring_boot.controller;

import com.example.mdb_spring_boot.model.User;
import com.example.mdb_spring_boot.model.UserChest;
import com.example.mdb_spring_boot.model.UserSkin;
import com.example.mdb_spring_boot.service.UserService;
```

```
import org.bson.types.ObjectId;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import javax.json.JsonObject;
import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {
    private final UserService userService;

    @PostMapping
    public User addUser(@RequestBody User user){
        return userService.addUser(user);
    }

    @PostMapping("/{userId}/chests")
    public User addChestToUser(@PathVariable String userId, @RequestBody UserChest chest) {
        chest.setChestId(new ObjectId(chest.getChestId().toString()));
        return userService.addChestToUser(userId, chest);
    }

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping("/{userId}/skins")
    public User addSkinToUser(@PathVariable String userId, @RequestBody UserSkin skin) {
        return userService.addSkinToUser(userId, skin);
    }

    @PostMapping("/open-chest")
    public User openChest(@RequestParam String userId, @RequestParam String chestId) {
        return userService.openChest(userId, chestId);
    }

    @GetMapping("/all")
    public List<JsonObject> getAllUsers() {
        return userService.getAllUsers();
    }
}
```

ChestController.java

Opis: Kontroler REST dla zarządzania operacjami związanymi ze skrzyniami. Oferuje endpointy do dodawania nowych skrzyń, dodawania skór do skrzyń oraz pobierania szczegółów skrzyń.

```
package com.example.mdb_spring_boot.controller;

import com.example.mdb_spring_boot.model.Chest;
import com.example.mdb_spring_boot.model.Skin;
import com.example.mdb_spring_boot.service.ChestService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import javax.json.JsonObject;
import java.util.List;

@RestController
@RequestMapping("/api/chests")
public class ChestController {
    private final ChestService chestService;

    @Autowired
    public ChestController(ChestService chestService) {
        this.chestService = chestService;
    }

    @PostMapping
    public Chest addChest(@RequestBody Chest chest) {
        return chestService.addChest(chest);
    }

    @PostMapping("/{chestId}/skins")
    public Chest addSkinToChest(@PathVariable String chestId, @RequestBody Skin skin) {
        return chestService.addSkinToChest(chestId, skin);
    }

    @GetMapping("/all")
    public List<JsonObject> getAllChests() {
        return chestService.getAllChests();
    }

    @GetMapping("/{chestId}")
    public Chest getChestById(@PathVariable String chestId) {
```

```
        return chestService.getChestById(chestId);
    }
}
```

LogController.java

Opis: Kontroler REST do zarządzania logami. Oferuje endpointy do dodawania nowych logów oraz pobierania szczegółów logów.

```
package com.example.mdb_spring_boot.controller;

import com.example.mdb_spring_boot.service.LogService;
import com.example.mdb_spring_boot.model.Log;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

import javax.json.JsonObject;
import java.util.List;

@RestController
@RequestMapping("/api/logs")
public class LogController {
    private final LogService logService;

    @Autowired
    public LogController(LogService logService) {
        this.logService = logService;
    }

    @PostMapping
    public Log addLog(@RequestBody Log log) {
        return logService.addLog(log);
    }

    @GetMapping("/all")
    public List<JsonObject> getAllLogs() {
        return logService.getAllLogs();
    }

    @GetMapping("/print")
    public List<JsonObject> printLogs() {
        return logService.getAllLogs();
    }

    @GetMapping("/{id}")
    public Log getLogById(@PathVariable String id) {
        return logService.getLogById(id);
    }
}
```

ReportController.java

Opis: Kontroler REST do zarządzania generowaniem raportów na podstawie danych z systemu.

```
package com.example.mdb_spring_boot.controller;

import com.example.mdb_spring_boot.service.ReportService;
import org.bson.types.ObjectId;
import org.springframework.beans.factory.annotation.Autowired;
import org.bson.Document;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.json.JsonObject;

import org.bson.types.ObjectId;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import javax.json.JsonObject;
import java.util.List;

@RestController
@RequestMapping("/api/reports")
public class ReportController {
    private final ReportService reportService;

    @Autowired
    public ReportController(ReportService reportService) {
```

```
        this.reportService = reportService;
    }

    @GetMapping("/users-total-skins-values")
    public List<JsonObject> getUsersTotalSkinsValues() {
        return reportService.generateUsersTotalSkinsValues();
    }

    @GetMapping("/user-total-spending/{userId}")
    public List<JsonObject> getUserTotalSpending(@PathVariable String userId) {
        return reportService.generateUserTotalSpending(new ObjectId(userId));
    }

    @GetMapping("/skin-report")
    public List<JsonObject> getSkinReport() {
        return reportService.generateSkinReport();
    }
}
```

Moduł config

CorsConfig.java

Opis: Klasa konfiguracyjna do ustawiania zasad CORS (Cross-Origin Resource Sharing). Umożliwia kontrolowanie, które źródła mogą uzyskiwać dostęp do zasobów API.

```
package com.example.mdb_spring_boot.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class CorsConfig {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**") // Adjust the mapping as needed
                        .allowedOrigins("http://localhost:3000") // Allow requests from this origin
                        .allowedMethods("GET", "POST", "PUT", "DELETE"); // Allow these HTTP methods
            }
        };
    }
}
```

Moduł exception

InsufficientFundsException.java

Opis: Niestandardowy wyjątek sygnalizujący brak wystarczających środków na koncie użytkownika do wykonania określonej operacji.

```
package com.example.mdb_spring_boot.exception;

public class InsufficientFundsException extends RuntimeException {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

Moduł util

DrawingMachine.java

Opis: Narzędzie do losowania skórek z otwieranych skrzynek.

```
package com.example.mdb_spring_boot.util;

import com.example.mdb_spring_boot.model.Chest;
import com.example.mdb_spring_boot.model.Skin;
import com.example.mdb_spring_boot.model.UserSkin;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.List;
```

```
import java.util.Random;

public class DrawingMachine {
    private double totalWeights;
    private final Random random;

    public DrawingMachine(){
        this.totalWeights = 0.0;
        this.random = new Random();
    }

    public UserSkin getRandomSkin(Chest chest){
        this.getTotalWeights(chest.getSkins());

        double randomNumber = random.nextDouble() * totalWeights;

        UserSkin randomSkin = null;

        double sum = 0.0;
        for (Skin skin : chest.getSkins()){
            sum += skin.getOdds();
            if (randomNumber < sum) {
                double wear = this.random.nextDouble();
                int pattern = this.random.nextInt(999) + 1;
                double price = BigDecimal.valueOf(skin.getBasePrice() / Math.max(wear, 0.05) + pattern / 10.0)
                        .setScale(2, RoundingMode.HALF_UP).doubleValue();

                randomSkin = new UserSkin(
                        skin.getName(),
                        skin.getType(),
                        wear,
                        pattern,
                        price,
                        skin.getId()
                );
                break;
            }
        }

        return randomSkin;
    }

    private void getTotalWeights(List<Skin> skins){
        this.totalWeights = 0.0;
        for (Skin skin : skins)
            totalWeights += skin.getOdds();
    }
}
```

Testy

Moduł `service`

UserServiceTest.java

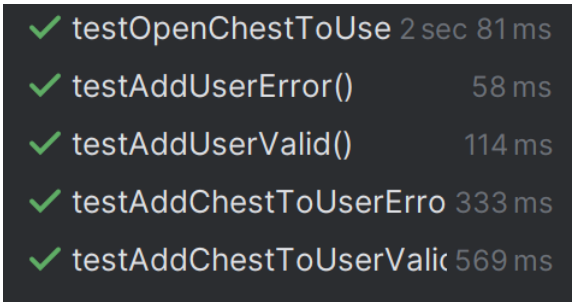
Opis: Klasa testująca operacje na bazie danych.

```
package com.example.mdb_spring_boot.service;

import com.example.mdb_spring_boot.exception.InsufficientFundsException;
import com.example.mdb_spring_boot.model.*;
import com.example.mdb_spring_boot.repository.UserRepository;
import org.bson.types.ObjectId;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.test.annotation.Rollback;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest
public class UserServiceTest {
    @Autowired
    private UserService userService;

    @Autowired
    private ChestService chestService;

    @Autowired
```

```
    private LogService logService;

    @Test
    public void testAddUserValid() {
        User user = new User("Name", "Surname", "test@example.pl", 100);

        User savedUser = userService.addUser(user);

        assertNotNull(savedUser);
        assertEquals(user.getName(), savedUser.getName());
        assertEquals(user.getSurname(), savedUser.getSurname());
        assertEquals(user.getEmail(), savedUser.getEmail());

        userService.removeUser(user);
    }

    @Test
    public void testAddUserError(){
        User user = new User("Name", "Surname", "testexample.pl", 100);

        assertThrows(DataIntegrityViolationException.class, () -> {
            userService.addUser(user);
        });
    }

    private Chest createTestChest(){
        Skin skin1 = new Skin("Name1", "Type1", "Rarity1", 0.5,100.0);
        Skin skin2 = new Skin("Name2", "Type2", "Rarity2", 0.5,100.0);
        return new Chest("ChestName", 100, List.of(skin1, skin2));
    }

    @Test
    public void testAddChestToUserValid(){
        // preparing test environment
        User user = new User("Name", "Surname", "teste@xample.pl", 100);
        User savedUser = userService.addUser(user);

        Chest chest = createTestChest();
        chestService.addChest(chest);

        UserChest userChest = new UserChest(new ObjectId(chest.getId()), 1);

        // adding chest to user
        User updatedUser = userService.addChestToUser(savedUser.getId(), userChest);

        // actual tests
        List<UserChest> chestsList = updatedUser.getChests();
        assertEquals(1, chestsList.size());
        assertEquals(chest.getId(), chestsList.get(0).getChestId().toString());
        assertEquals(userChest.getQuantity(), chestsList.get(0).getQuantity());

        List<Log> logList = logService.getLogsByUserId(user.getId());
        assertEquals(1, logList.size());

        // cleaning test environment
        userService.removeUser(user);
        chestService.removeChest(chest);
        logService.removeLog(logList.get(0));
    }

    @Test
    public void testAddChestToUserError(){
        // preparing test environment
        User user = new User("Name", "Surname", "teste@xample.pl", 50);
        User savedUser = userService.addUser(user);

        Chest chest = createTestChest();
        chestService.addChest(chest);

        UserChest userChest = new UserChest(new ObjectId(chest.getId()), 1);

        // adding chest to user
        assertThrows(InsufficientFundsException.class, () -> {
            userService.addChestToUser(savedUser.getId(), userChest);
        });
        userService.removeUser(user);
        chestService.removeChest(chest);
    }

    @Test
    public void testOpenChestToUserValid(){
        // preparing test environment
        User user = new User("Name", "Surname", "teste@xample.pl", 100);
        User savedUser = userService.addUser(user);

        Chest chest = createTestChest();
        chestService.addChest(chest);

        UserChest userChest = new UserChest(new ObjectId(chest.getId()), 1);
```

```
        // adding chest to user
        userService.addChestToUser(savedUser.getId(), userChest);

        // opening chest
        User updatedUser = userService.openChest(savedUser.getId(), chest.getId());

        // actual tests
        List<UserSkin> listUserSkin = updatedUser.getSkins();
        assertEquals(1, listUserSkin.size());

        List<Skin> skinList = chest.getSkins();

        assertTrue(listUserSkin.get(0).getSkinId().equals(skinList.get(0).getId()) ||
listUserSkin.get(0).getSkinId().equals(skinList.get(1).getId()));

        List<Log> logList = logService.getLogsByUserId(user.getId());
        assertEquals(2, logList.size());

        // cleaning test environment
        userService.removeUser(user);
        chestService.removeChest(chest);
        logService.removeLog(logList.get(0));
        logService.removeLog(logList.get(1));
    }
}
```

Wyniki testów:



Zagadnienie współdostępu do bazy danych

MongoDB zapewnia wbudowaną ochronę przed jednoczesnym dostępem do dokumentów poprzez:

1. Blokady na poziomie dokumentu: Zapewniają, że pojedynczy dokument jest modyfikowany przez jedną operację naraz, minimalizując ryzyko konfliktów.
2. Atomowe operacje na dokumencie: Wszystkie zmiany na jednym dokumencie są przeprowadzane jako jedna, niepodzielna jednostka.
3. Transakcje wielodokumentowe: Od wersji 4.0 MongoDB obsługuje transakcje, które pozwalają na grupowanie wielu operacji w jedną, atomową jednostkę, zapewniając integralność danych w różnych dokumentach i kolekcjach.

Źródło: https://www.mongodb.com/docs/manual/faq/concurrency/#faq-concurrency

FAQ: Concurrency

MongoDB allows multiple clients to read and write the same data. To ensure consistency, MongoDB uses locking and concurrency control to prevent clients from modifying the same data simultaneously. Writes to a single document occur either in full or not at all, and clients always see consistent data.

What type of locking does MongoDB use?

MongoDB uses multi-granularity locking [1] that allows operations to lock at the global, database or collection level, and allows for individual storage engines to implement their own concurrency control below the collection level (e.g., at the document-level in WiredTiger).

MongoDB uses reader-writer locks that allow concurrent readers shared access to a resource, such as a database or collection.

In addition to a shared (S) locking mode for reads and an exclusive (X) locking mode for write operations, intent shared (IS) and intent exclusive (IX) modes indicate an intent to read or write a resource using a finer granularity lock. When locking at a certain granularity, all higher levels are locked using an intent lock.

For example, when locking a collection for writing (using mode X), both the corresponding database lock and the global lock must be locked in intent exclusive (IX) mode. A single database can simultaneously be locked in IS and IX mode, but an exclusive (X) lock cannot coexist with any other modes, and a shared (S) lock can only coexist with intent shared (IS) locks.

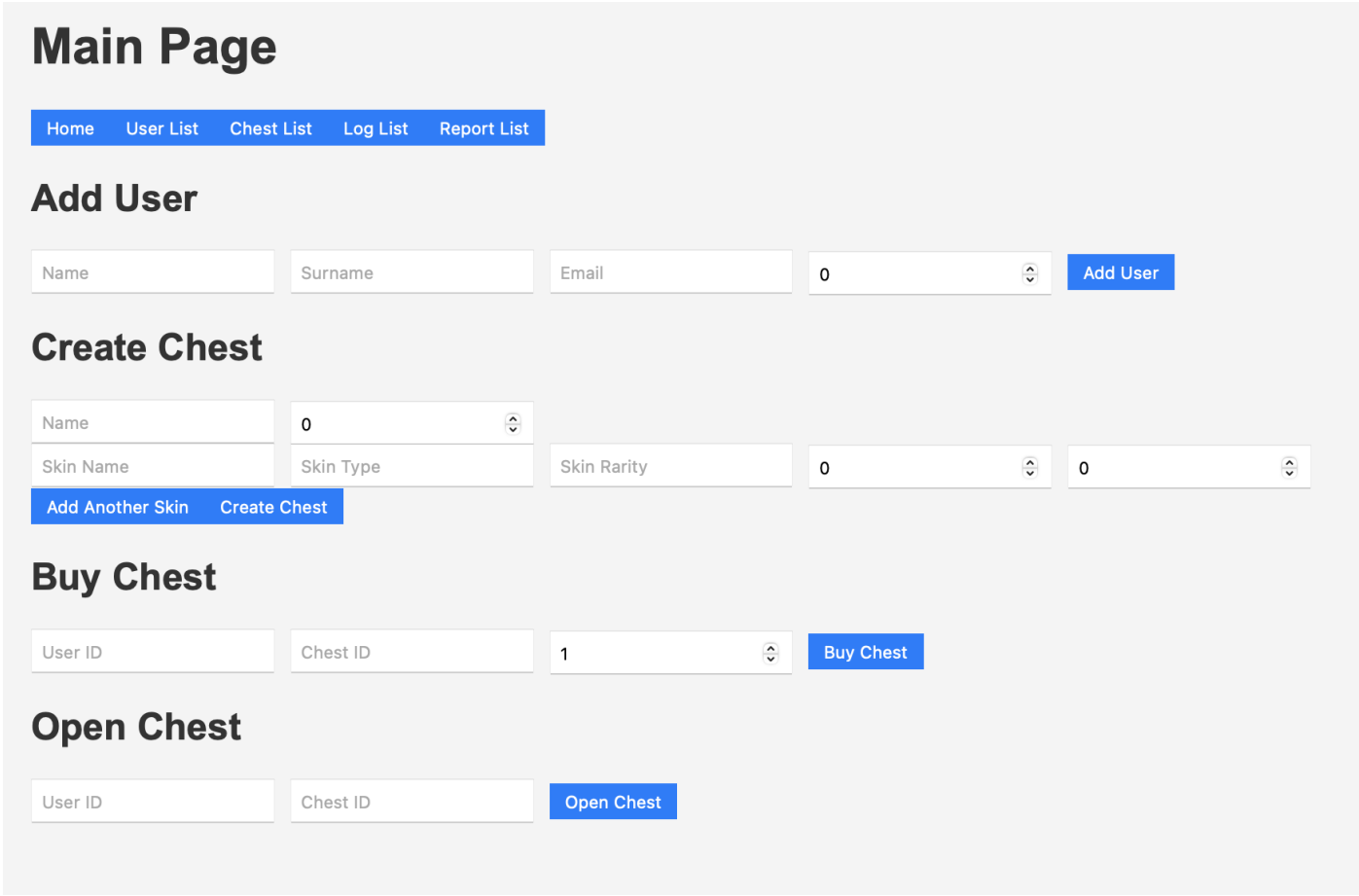
Locks are fair, with lock requests for reads and writes queued in order. However, to optimize throughput, when one lock request is granted, all other compatible lock requests are granted at the same time, potentially releasing the locks before a conflicting lock request is performed. For example, consider a situation where an X lock was just released and the conflict queue contains these locks:

IS → IS → X → X → S → IS

In strict first-in, first-out (FIFO) ordering, only the first two IS modes would be granted. Instead MongoDB will actually grant all IS and S modes, and once they all drain, it will grant X, even if new IS or S requests have been queued in the meantime. As a grant will always move all other requests ahead in the queue, no starvation of any request is possible.

Wykorzystane technologie front-end

Podczas projektowania front-end'u na potrzeby naszego projektu zdecydowaliśmy się na framework React do języka JavaScript. Wybraliśmy go głownie z uwagi na duzą swobodę działania jak i pewne nasze doświadczenie w tej technologii oraz łatwość łączenia właśnie front-end'u z back-end'em.


Main Page
Home
User List
Chest List
Log List
Report List
Add User
Name
Surname
Email
0
Add User
Create Chest
Name
0
Skin Name
Skin Type
Skin Rarity
0
0
Add Another Skin
Create Chest
Buy Chest
User ID
Chest ID
1
Buy Chest
Open Chest
User ID
Chest ID
Open Chest

User List

Alice Smith - alice.smith@example.com Deposit: 270.0

Chests:

No chests

Skins:

Name: Fire Serpent, Type: AK-47, Price: 2046.4

Bob Johnson - bob.johnson@example.com Deposit: 820.0

Chests:

Chest ID: 665cff924b58ee76868aca27 (Quantity: 3)

Skins:

Name: Howl, Type: M4A4, Price: 363.58

Name: Fire Serpent, Type: AK-47, Price: 800.09

Name: Howl, Type: M4A4, Price: 258.19

Charlie Brown - charlie.brown@example.com Deposit: 500.0

Home User List Chest List Log List Report List

Chest List

Bravo Case - Price: 30.0

Skins:

Name: Howl, Rarity: Covert, Odds: 0.3

Name: Fire Serpent, Rarity: Classified, Odds: 0.5

Name: Gamma Doppler, Rarity: Covert, Odds: 0.2

Vanguard Case - Price: 100.0

Skins:

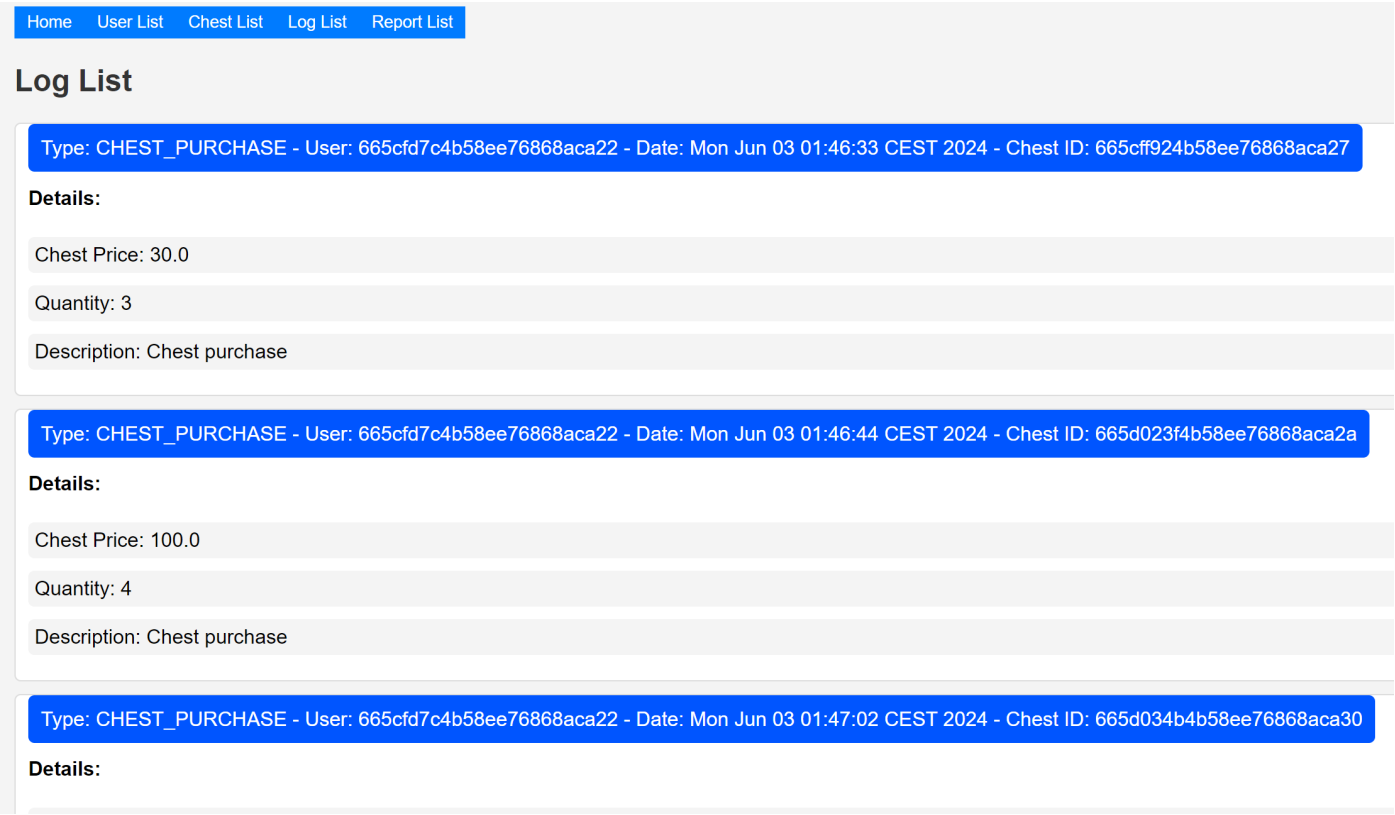
Name: Asiimov, Rarity: Covert, Odds: 0.3

Name: Cyrex, Rarity: Classified, Odds: 0.6

Spectrum Case - Price: 200.0

Skins:

Name: Neo-Noir, Rarity: Covert, Odds: 0.2



Wnioski

Chcielibyśmy zwrócić uwagę na trudności i wyzwania, które nam towarzszyły w trakcie tego projektu

1. Konwersja ObjectId na String

Problem: Dane ObjectId w MongoDB były konwertowane na złożone obiekty zawierające timestamp i inne metadane, co powodowało trudności w przesyłaniu i wyświetlaniu tych danych na froncie.

Rozwiązanie: Implementacja dodatkowych metod na backendzie do konwersji ObjectId na prostsze formaty string przed wysłaniem ich do frontendu. To zapewniło, że dane były wyświetlane poprawnie i w łatwym do odczytania formacie.

2. Błędy związane z zapytaniami MongoDB

Problem: Podczas pisania zapytań MongoDB napotkaliśmy na błędy takie jak The field 'name' must be an accumulator object. Te błędy często wynikały z nieprawidłowego użycia operatorów agregacyjnych.

Rozwiązanie: Wymagało to dogłębnej analizy dokumentacji oraz konsultacji z zespołem, aby zrozumieć, jakie są oczekiwane struktury danych dla operatorów agregacyjnych i jak je poprawnie stosować.

3. Obsługa różnorodnych typów danych w logach

Problem: Logi w systemie zawierały różnorodne typy danych, co powodowało trudności w ich jednolitym przetwarzaniu i wyświetlaniu.

Rozwiązanie: Implementacja polimorficznego podejścia do obsługi różnych typów logów (np. CHEST_PURCHASE, CHEST_OPEN) na backendzie oraz dynamiczne renderowanie tych danych na froncie. Wymagało to dokładnego zaplanowania i przetestowania różnych scenariuszy danych.

Komentarz

Przygotowanie tego miniprojektu było bardzo rozwijającym zadaniem. Nierelacyjne bazy danych, pomimo utrudnionych operacji agregacji, mają również wiele zalet i jeśli tylko dobrze je wykorzystamy, to możemy tworzyć ciekawe i wydajne projekty.