Refaktoryzacja + nowe funkcjonalności

Co Zamiast dziedziczenia?

Trudności w wypadku dziedziczenia(w naszym projekcie):

Administrator/pracownik będzie chciał dodawać nowe pojazdy z nowymi właściwościami, więc w naszym systemie nie mamy pewności jakie pojazdy pojawią się w przyszłości.

Możemy użyć albo json, albo kompozycji oraz wzorca EAV (Entity-Attribute-Value)

Trudności przy użyciu dziedziczenia

Sztywność struktury:
Nowy rodzaj pojazdu = nowy typ = zmiany w kodzie

Złożność struktury bazodanowej:
Dużo tabel/kolumn

Reprezentacja nowych typów pojazdów będzie w przyszłości wymagała zmian w stukturze bazy danych – nowe tabele lub nowe kolumny.

Zmiany niemożliwe do wprowadzenia bez programisty!
(może to jednak zaleta?)

Jakie są rozwiązania?

Przykład reprezentacji tabeli z dziedziczeniem:

id	type	brand	model	year	plate	seats	licence_category	color
1	Bus	Volkswagen	T2	1985	LU123	20	NULL	NULL
2	Motorcycle	Honda	CBR600	2016	LU456	NULL	A	NULL

Jakie są rozwiązania?

Atrybuty w dodatkowym polu tekstowym jako json

id	type	brand	model	year	plate	attributes
1	Bus	Volkswagen	T2	1985	LU123	{"seats":20,"atrybut_busa":"wa...
2	Motorcycle	Honda	CBR600	2016	LU456	{"licence_category":"A","naped...

Jakie są rozwiązania?

Encja-Atrybut-Wartość(Entity-Attribute-Value)

id	vehicle_id	attr_key	attr_value
1	1	seats	20
2	2	licence_category	A

Zalety Entity-Attribute-Value:

Elastyczna struktura:
Dodawanie nowych typów pojazdów nie wymaga zmian w kodzie.

Prostota struktury bazodanowej:
Dla pojazdów wystarczą 2 tabele:
vehicle,vehicle_attribute

Brak zmian w strukturze bazodanowej.

Wady Entity-Attribute-Value:

Mniejsza czytelność atrybutów dla typów pojazdów.

Potrzeba walidacji typów tych atrybutów – atrybut to zawsze para nazwa:String,wartosc:String.

Większa złożność zapytań SQL(duża ilość joinów) i problemy z wydajnością przy dużych zbiorach danych!

Podejście JSON

Większa czytelność atrybutów dla typów pojazdów.

W większości przypadków działa szybciej niż EAV – poza zapytaniami zwiazanymi z pojedyńczymi atrybutami.

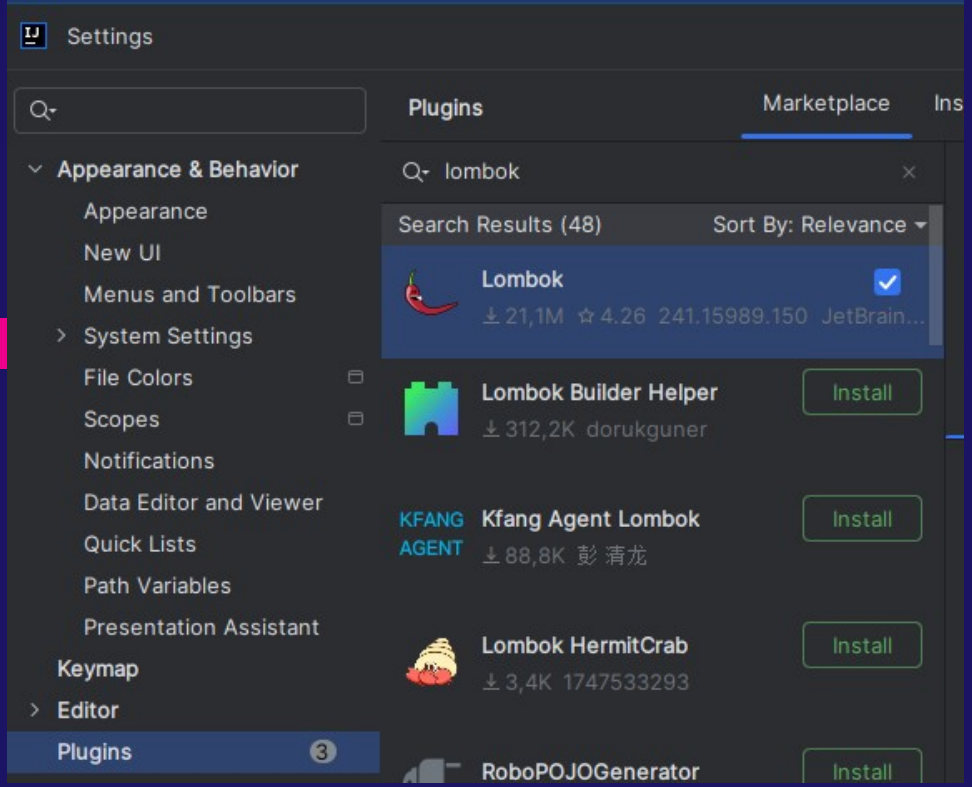
Łatwiejsze zapytania niż w przypadku EAV.

Przykładowe porównanie:
https://docs.evolveum.com/midpoint/projects/midscale/design/repo/repository-json-vs-eav/

Przed implementacją:

Aby pozbyć się boilerplate’u wykorzystamy w aplikacji Project Lombok.

Lombok pozwala na wygenerowanie wiele powtarzalnych składowych klasy, stosując odpowiednie adnotacje.




```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.36</version>
    <scope>provided</scope>
</dependency>
```

Implementacja:

```
import lombok.*;

import java.util.Map;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Vehicle {

    private String id;
    private String category;
    private String brand;
    private String model;
    private int year;
    private String plate;
    @Builder.Default
    private Map<String, Object> attributes = new
HashMap<>(); //Map.of() jeżeli ma być niezmienne po
builderze;


    public Object getAttribute(String key) {
        return attributes.get(key);
    }
//Jak możemy zmienić mapę to jeszcze add i remove:
    public void addAttribute(String key, Object
value) {
        attributes.put(key, value);
    }

    public void removeAttribute(String key) {
        attributes.remove(key);
    }
}
```

Implementacja:

@Data – generuje automatycznie kod:

- @Getter – gettery dla wszystkich pól
- @Setter – settery dla wszystkich pól
- @ToString() – zwraca Stringa z obiektu
- @EqualsAndHashCode() porównania
- @RequiredArgsConstructor()- z polami final

@NoArgsConstructor – konstruktor bez argumentów
@AllArgsConstructor – konstruktor z wszystkimi argumentami
@Builder – kod z wzorcem buidlera.

Przykład użycia buildera, przy tworzeniu nowego pojazdu(np. w funkcji main):

```
Map<String, Object> attributes = new HashMap<>();
attributes.put(key, value);
Vehicle vehicle = Vehicle.builder()
        .brand(brand)
        .model(model)
        .year(year)
        .plate(plate)
        .category(category)
        .attributes(attributes)
        .build();
```

Interfejs:

Repozytorium, które będzie wykorzystywało klasę Vehicle, powinno implementować nastepujące metody:

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

public interface VehicleRepository {
    List<Vehicle> findAll();
    Optional<Vehicle> findById(String id);
    Vehicle save(Vehicle vehicle);
    void deleteById(String id);
}
```

Optional opakowuje odpowiedź. Używamy, gdy możemy otrzymać null. Np, gdy chcemy wypożyczyć pojazd o nieistniejącym id:

```
if (vehicleRepository.findById(vehicleId).isEmpty())
   //...albo: .isPresent()
```

Implementacja:

Można zamiast do csv, zserializować obiekty używane w projekcie do json, używając zewnętrznych bibliotek:

```
<!--
https://mvnrepository.com/artifact/com.google.code.gson/
gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.12.1</version>
</dependency>
```

Repozytorium:

Można zamiast do csv, zserializować obiekty używane w projekcie do json, używając zewnętrznych bibliotek:

```
<!-- Prosty generyczny odczyt i zapis przez gson w formie
klasy javy zostanie udostepniony na ćiczeniach -->
//zapis:
String json = gson.toJson(data);
Files.writeString(path, json, StandardOpenOption.CREATE,
StandardOpenOption.TRUNCATE_EXISTING);
//odczyt:
String json = Files.readString(path);
List<T> list = gson.fromJson(json, type);
```

JSON:

Atrybuty pojazdów w klasach javy są reprezentowane przez mapę.

Przykład pliku vehicles.json:

```
[
  {
    "id": "1",
    "category": "Bus",
    "brand": "Volkswagen",
    "model": "T2",
    "year": 1985,
    "plate": "LU123",
    "attributes": {
      "seats": 20
    }
  },
  {
    "id": "2",
    "category":"Motorcycle",
    "brand": "Honda",
    "model": "CBR600",
    "year": 2016,
    "plate": "LU456",
    "attributes": {
      "licence_category": "A",
      "drive": "chain"
    }
  }
]
```

Obiekty w javie:

Atrybuty pojazdów w klasach javy są reprezentowane przez mapę.

Vehicle(id=1, category=Bus, brand=Volkswagen, model=T2, year=1985, plate=LU123, attributes={seats=20.0})
Vehicle(id=2, category=Motorcycle, brand=Honda, model=CBR600, year=2016, plate=LU456, attributes={licence_category=A, drive=chain})
Vehicle(id=3, category=Car, brand=Toyota, model=Corolla, year=2024, plate=LU789, attributes={})

Repozytorium VehicleJsonRepository:

```
public class VehicleJsonRepository
implements VehicleRepository {
```

TypeToken umożliwia przekazanie do Gsona informacji o typie generycznym (List<Vehicle>), który normalnie znika w czasie działania programu (type erasure).

```
    private final JsonFileStorage<Vehicle> storage =
            new JsonFileStorage<>("vehicles.json", new
    TypeToken<List<Vehicle>>(){}.getType());

    private final List<Vehicle> vehicles;
```

Repozytorium VehicleJsonRepository:

W konstruktorze ładujemy dane z pliku json:

```
public VehicleJsonRepository() {
    this.vehicles = new
ArrayList<>(storage.load());
}
```

Implementacja metody zwracającej wszystkie pojazdy:

```
@Override
public List<Vehicle> findAll() {
    return new ArrayList<>(vehicles);
}
```

Repozytorium VehicleJsonRepository:

Do odnalezienia pojazdu po id, można wykorzystać stumienie z metodą filtrującą:

```
@Override
public Optional<Vehicle> findById(String id) {
    return vehicles.stream().filter(v ->
v.getId().equals(id)).findFirst();
}
```

Repozytorium VehicleJsonRepository:

SaveOrUpdate:
Używamy UUID aby prosto wygenerować nowe id

```
@Override
public Vehicle save(Vehicle vehicle) {
    if (vehicle.getId() == null ||
vehicle.getId().isBlank()) {
        vehicle.setId(UUID.randomUUID().toString());
    } else {
        deleteById(vehicle.getId());
    }
    vehicles.add(vehicle);
    storage.save(vehicles);
    return vehicle;
}
```

Repozytorium VehicleJsonRepository:

Usunięcie pojazdu:

```
@Override
public void deleteById(String id) {
    vehicles.removeIf(v -> v.getId().equals(id));
    storage.save(vehicles);
}
```

Repozytorium:

W projekcie analogicznie trzeba zaimplementować repozytorium Usera, dodając findByLogin:

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

public interface UserRepository {
    List<User> findAll();
    Optional<User> findById(String id);
    Optional<User> findByLogin(String login);
    User save(User user);
    void deleteById(String id);
}
```

Wypożyczenie auta:

Nowa funkcjonalność:

-admin, powinien mieć wgląd kto i kiedy wypożyczył auto

Rozwiązanie:

-nowa klasa reprezenująca wypożyczenie.
-nowe repozytorium
-nowy plik json.

Wypożyczenie auta:

Nowa funkcjonalność:

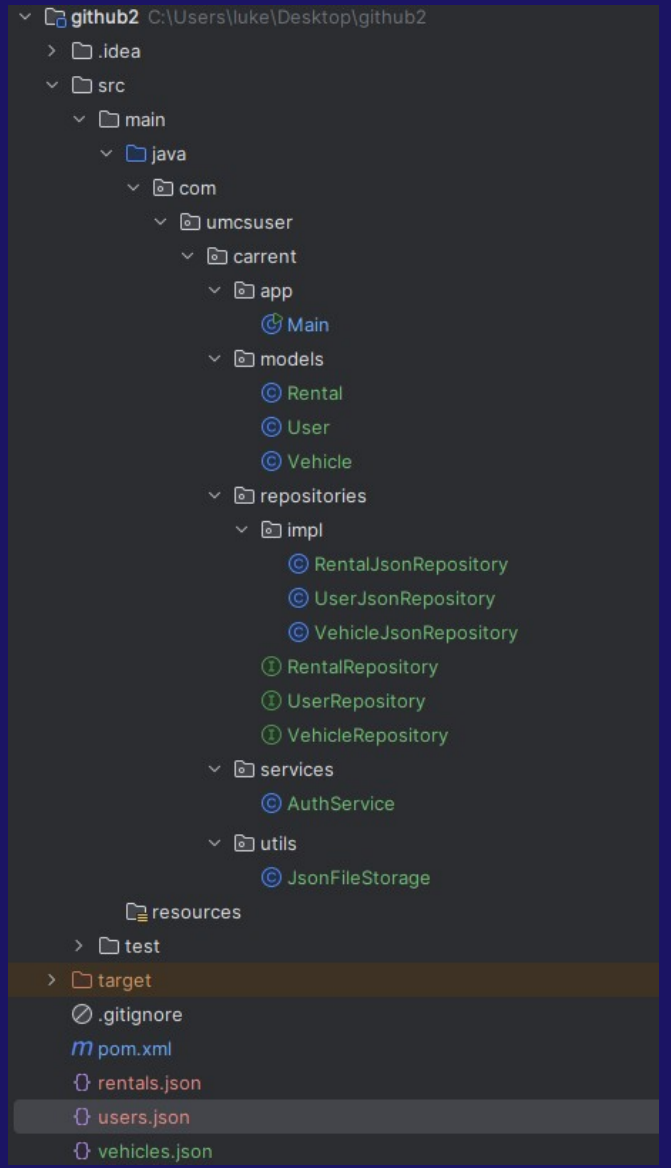
-admin, powinien mieć wgląd kto i kiedy wypożyczył auto

Rozwiązanie:

-nowa klasa reprezenująca wypożyczenie.
-nowe repozytorium
-nowy plik json.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Rental {
    private String id;
    private String vehicleId;
    private String userId;
    private String rentDateTime;
    private String returnDateTime;

}
```


github2 C:\Users\luke\Desktop\github2
.idea
src
main
java
com
umcsuser
carrent
app
Main
models
Rental
User
Vehicle
repositories
impl
RentalJsonRepository
UserJsonRepository
VehicleJsonRepository
RentalRepository
UserRepository
VehicleRepository
services
AuthService
utils
JsonFileStorage
resources
test
target
.gitignore
pom.xml
rentals.json
users.json
vehicles.json

Zadanie

Przerobienie klas: Vehicle,User, dodanie klasy Rental – wypożyczenie.
Przerobienie/utworzenie klas Repozytoriów:
VehicleRepository,UserRepository,RentalRepository

Umieszczenie logiki odpowiadajacej za login i rejestrację w klasie AuthService.
Użycie Bcrypt(jbcrypt) do hashowania i sprawdzania hasła.
Dwie metody: login i register, co powinny zwracać?

Utworzenie logiki do wypożyczania i zwracania pojazdów wykorzystując nowe repozytoria.
Admin może dodawać pojazdy i usuwać, oraz przeglądać wszystkie, również wypożyczone. User może wypożyczyć i zwrócić pojazd. Widzi tylko dostępne pojazdy.

Opcjonalnie - Przeniesienie logiki związanej z dostępem do danych – zamiast bezpośrednio używać repozytorium w main/app to użycie klas Service.
Zaproponowanie metod takich serwisów.