WPROWADZENIE DO SPRING

BARTOSZ ANDREATTO

Programista w Banku Pekao S.A z bogatym doświadczeniem w zakresie technologii back-end'owych wykorzystujących wirtualną maszynę Javy. Pasjonat rozwiązań opartych na ekosystemie Spring oraz rozwiązań Oracle. Certyfikowany programista Java. Uwielbia nauczać oraz dzielić się wiedzą.

LinkedIn: https://www.linkedin.com/in/bartosz-andreatto-02b03413a/

e-mail: bandreatto@gmail.com

ZAKRES SZKOLENIA DZIEŃ 2

- Konfigurowanie połączenia do bazy danych
- Menadżer transakcji – zasada działania, dostępne implementacje
- Parametry transakcji – propagacja, reguły wycofywania, poziom izolacji, czas ważności
- Konfigurowanie mechanizmu transakcyjnego
- Integracja z JPA i frameworkiem Hibernate
- Tworzenie warstwy utrwalania z użyciem Spring Data

SPRING DATA

Aplikacje webowe najczęściej przechowują dane w pewnych bazach danych (**SQL** lub **NoSQL**). Wybierając relacyjną bazę danych i tworząc warstwę bazodanową, najczęściej wykorzystujemy **framework Hibernate**.
W takim przypadku, tworząc aplikację opartą o Spring Boota najczęściej wykorzystujemy starter:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

AUTOKONFIGURACJA

spring-boot-starter-data-jpa, tak jak inne startery, upraszczają pracę programistów, zwalniając ich z konieczności konfigurowania pewnych rzeczy. Starter ten zwalnia nas z konieczności konfiguracji **Hibernata** oraz w prosty sposób pozwala skonfigurować **DataSource** (np. w pliku **application.properties**).

TYPY BAZ DANYCH

Relacyjne bazy danych możemy podzielić na dwie główne grupy, tzn. takie, które:

- **przechowują dane w pamięci** (np. H2, HSQLDB)
- **przechowują dane w plikach** (MySQL, MariaDB)

spring-boot-starter-data-jpa, w zależności od tego, jakie **sterowniki baz danych znajdują się na classpathie** inaczej skonfiguruje DataSource:

- baza in-memory - połączenie do niej zostanie skonfigurowanie automatycznie
- MySQL i H2 - nastąpi próba połączenie do bazy „nie in memory”

BAZA H2

Baza danych H2 jest bardzo wygodną bazą danych, która warto wykorzystać podczas **procesu wytwarzania oprogramowania** (np. w **profilu deweloperskim**).
Możemy ją wybrać podczas bootstrapowania projektu.
Domyślnym użytkownikiem jest **sa**, jego **hasło jest puste**, wykorzystywany dialekt to:
org.hibernate.dialect.H2Dialect, a driver to **org.h2.Driver**. Domyślnie nazwa bazy jest generowana, a adres wypisywany w logach podczas startu.
Baza H2 umożliwia przeglądać jej zawartość za pomocą przeglądarki. Domyślnie konsola ta jest jednak wyłączona.

```
spring.h2.console.path=/h2
spring.h2.console.enabled=true
```

```
2020-08-28 21:48:21.508  INFO 17728 --- [main] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2'.
Database available at 'jdbc:h2:mem:679bb4b4-7115-4e55-8eae-f423e8f97900'
```

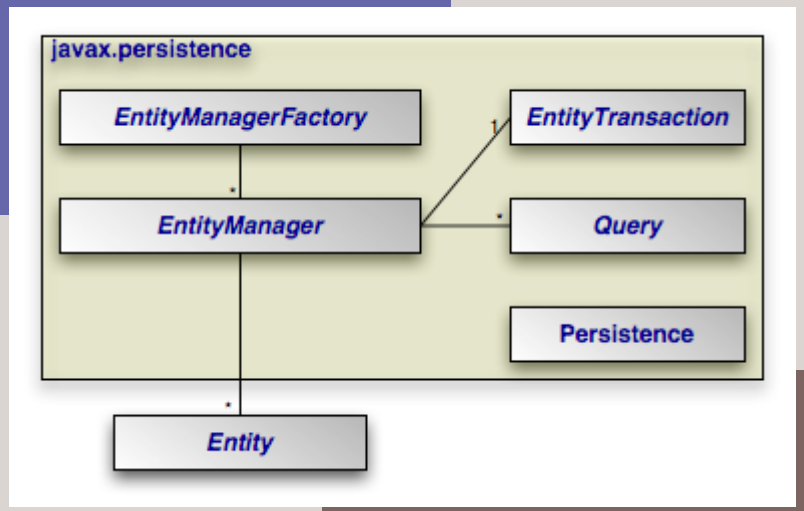
KONFIGURACJA DATASOURCE

Automatycznie dobrane wartości reprezentujące **DataSource** możemy zmienić za pomocą następujących właściwości konfiguracyjnych:

```
spring.datasource.username
spring.datasource.url
spring.datasource.password
spring.datasource.driver-class-name
```

ENTITYMANAGER

Dzięki wykorzystaniu **spring-boot-starter-data-jpa**, jesteśmy w stanie do dowolnego beana wstrzyknąć obiekt **EntityManager**. Pozwala on na wykonywanie prostych operacji (CRUD) i wykonywania zapytań z pomocą języka **HQL**. Pamiętajmy, że komponenty reprezentujące warstwę bazodanową lepiej, **zamiast adnotacją @Component, oznaczyć adnotacją @Repository**. Przykład pokazuje jego wykorzystanie.



ENTITYMANAGER

Zdefiniowana klasa DbInitializer, podczas wywołania linii kodu oznaczonej (x) **wyrzuci wyjątek** o następującej treści: `o EntityManager with actual transaction available for current thread - cannot reliably process 'persist' call`

```java
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "fruits")
public class Fruit {
  @Id
  @GeneratedValue
  private Long id;

  private String name;

  private Double weight;

  public Fruit(final String name, final Double weight) {
    this.name = name;
    this.weight = weight;
  }
}
```

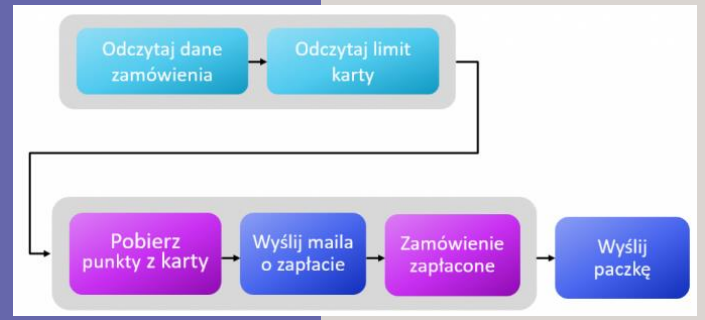
```java
import org.springframework.stereotype.Repository;

import javax.persistence.EntityManager;
import java.util.List;
import java.util.Optional;

@Repository
public class FruitRepository {

  private final EntityManager entityManager;

  public FruitRepository(final EntityManager entityManager) {
    this.entityManager = entityManager;
  }

  public Optional<Fruit> getById(final Long id) {
    return Optional.ofNullable(entityManager.find(Fruit.class, id));
  }

  public List<Fruit> findAll() {
    return entityManager.createQuery("SELECT f FROM fruits f", Fruit.class).getResultList();
  }

  public void delete(final Fruit fruit) {
    entityManager.remove(fruit);
  }

  public void deleteById(final Long id) {
    entityManager.createQuery("DELETE FROM fruits f WHERE f.id = :id")
        .setParameter("id", id)
        .executeUpdate();
  }

  public Fruit createFruit(final Fruit fruit) {
    entityManager.persist(fruit);
    return fruit;
  }
```

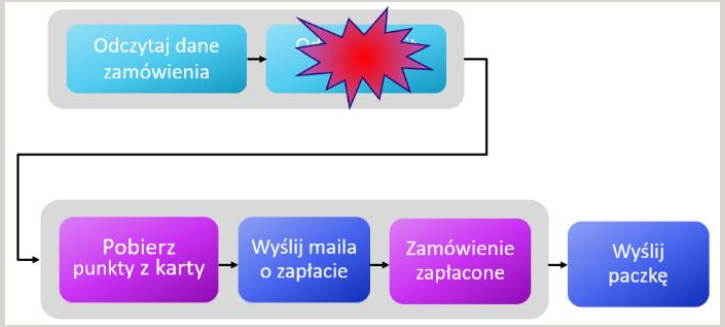
```java
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class DbInitializer implements CommandLineRunner {

  private final FruitRepository fruitRepository;

  public DbInitializer(final FruitRepository fruitRepository) {
    this.fruitRepository = fruitRepository;
  }

  @Override
  public void run(final String... args) throws Exception {
    final Fruit fruitA = new Fruit("Apple", 20.0);
    final Fruit fruitB = new Fruit("Banana", 18.0);
    fruitRepository.createFruit(fruitA); // (x)
    fruitRepository.createFruit(fruitB);
  }
}
```

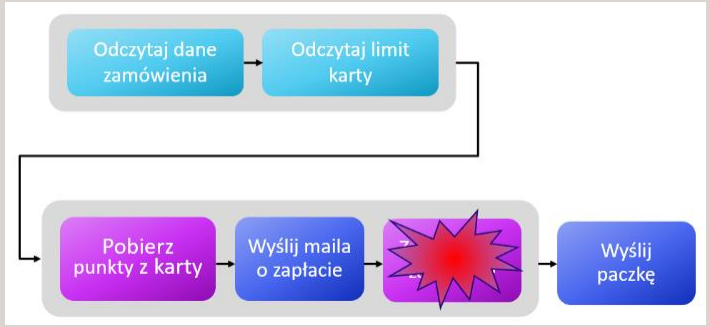
TRANSAKCJE – LOGIKA APLIKACJI

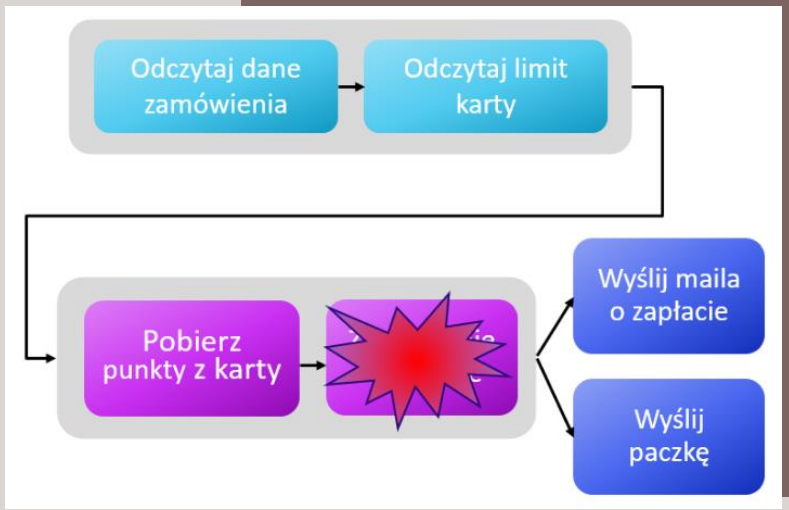
Logika aplikacji, która modyfikuje dane, zazwyczaj zmienia dane wielu rodzajów obiektów. Chcemy, aby:

- Operacje modyfikacji **wykonały się w ściśle określonej kolejności**, gdyż bieżące operacje mogą bazować na wynikach poprzedników.
- Zmiany zostały zaaplikowane **albo wszystkie, albo wcale.**









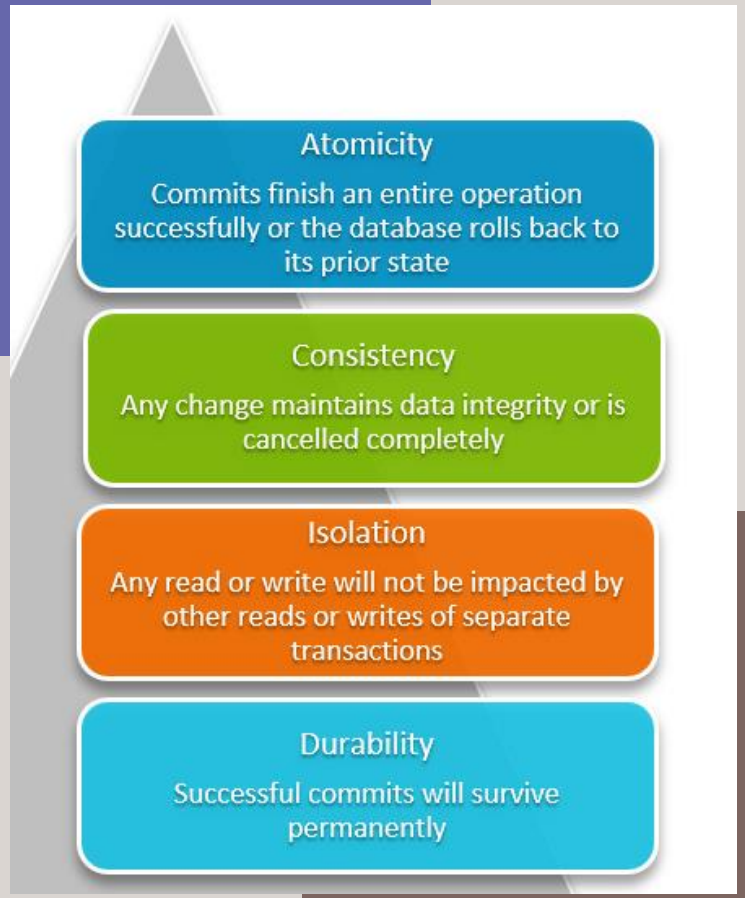
KONCEPCJA TRANSAKCJI

Patrząc na przedstawioną sytuację, chcielibyśmy, aby wszystkie modyfikacje danych na bazie danych **zostały zaaplikowane jednocześnie**. Taki mechanizm to właśnie transakcje w bazie danych. Relacyjne bazy danych, które są transakcyjne (**nie wszystkie są transakcyjne**) wspierają takie operacje. Po stronie bazy danych proces wygląda w następujący sposób:

- **Rozpoczęcie transakcji**
- **Modyfikacje danych**
- **Zakończenie transakcji (zatwierdzenie / wycofanie)**

ACID

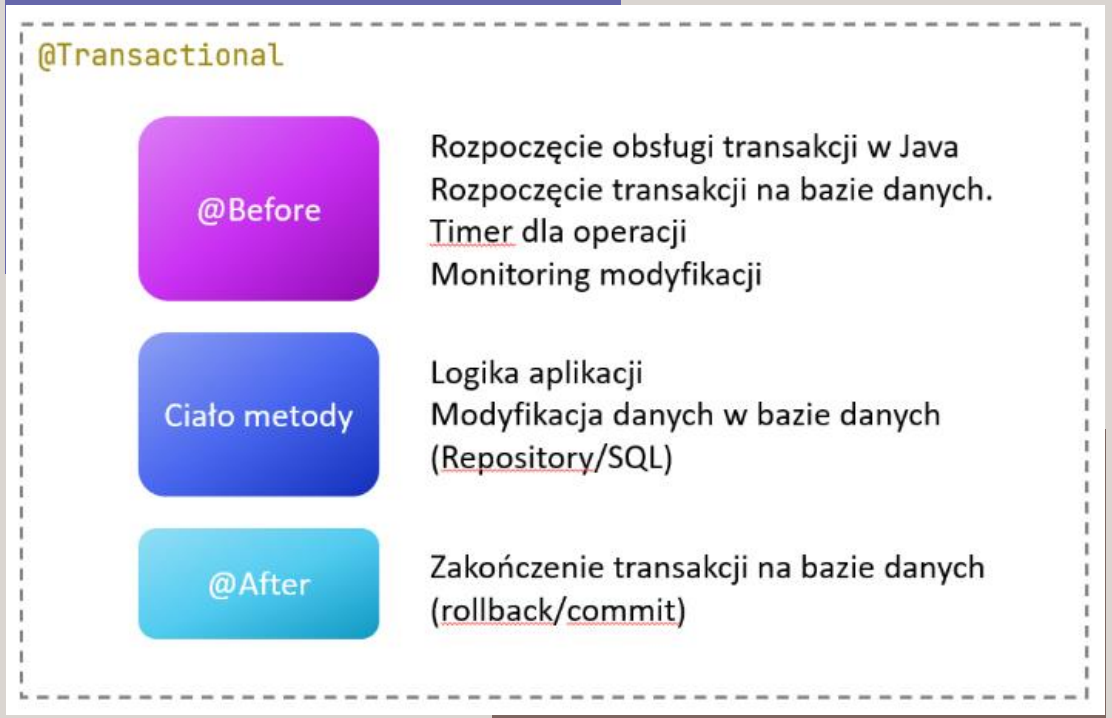
- **A (ang. *atomicity*)** – atomowość – transakcja może zostać wykonana w całości albo anulowana
- **C (ang. *consistency*)** – spójność – mówiąc potocznie oznacza, że baza jest w prawidłowym stanie
- **I (ang. *isolation*)** – izolacja – odseparowanie operacji wykonywanych jednocześnie
- **D (ang. durability)** – trwałość – gwarancja, że zatwierdzona transakcja nie zniknie niespodziewanie z systemu



ADNOTACJA TRANSACTIONAL

W Spring dostępny jest **mechanizm wspierający obsługę transakcji**. Mechanizm ten jest ściśle powiązany z anotacją **@Transactional**.

- Włącza obsługę transakcji
- Dostarcza danych oraz API dla frameworka ORM
- Tworzy proxy metody komponentu Spring:
  - Rozpoczyna transakcję
  - Umożliwia dwukierunkową integrację Spring z ORM
  - Uruchamia mechanizmy monitorujące czas wykonywania operacji czy też rzucane wyjątki
  - Zatwierdza lub wycofuje zmiany na bazie po wyjściu z metody



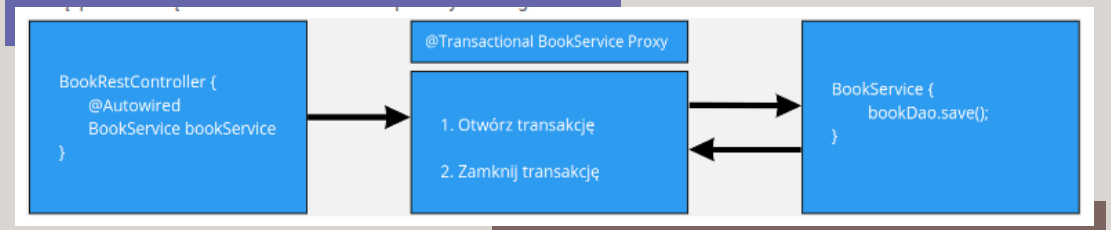
ADNOTACJA TRANSACTIONAL

Ręczne tworzenie i zarządzanie transakcjami potrafi być uciążliwe, a kod może zrobić się nieczytelny. Spring, dzięki wykorzystaniu **programowania aspektowego** (które jest poza zakresem tego szkolenia) eliminuje wszystkie te bolączki. Oferuje adnotację **@Transactional**, którą możemy umieszczać nad **metodami publicznymi lub klasą** (co jest równoznaczne z umieszczeniem jej **na wszystkich metodach publicznych w klasie**). Adnotacja ta gwarantuje, że kod, który ją wykorzystuje, zostanie uruchomiony w transakcji. Domyślnie, jeżeli zostanie wywołana metoda oznaczona tą adnotacją, a transakcja została już otwarta, **nowa transakcja nie zostanie rozpoczęta.**

ADNOTACJA TRANSACTIONAL

UWAGA: Importując adnotację **@Transactional**, możemy wybrać pomiędzy tą umieszczoną w pakiecie **javax.transaction** lub **org.springframework.transaction.annotation**. Framework stworzy transakcję jeżeli wybierzesz dowolną z nich, jednakże ta z pakietu org.springframework.transaction.annotation oferuje **więcej funkcjonalności**.

UWAGA: Błąd w poprzednim przykładzie można naprawić dodając adnotację **@Transactional nad klasę FruitRepository.**



TRANSAKCJE W SPRING – ZASADY DZIAŁANIA

- Obsługę transakcji w Spring włącza się poprzez adnotację **@EnableTransactionManagement** umieszczoną w klasie adnotowanej **@Configuration** lub z poziomu **konfiguracji Spring w XML**.

```
@Configuration
@EnableTransactionManagement
public class OrderPaymentsConfig { ... }
```

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
<tx:annotation-driven transaction-manager="transactionManager"/>
```

- Adnotacja **@Transactional musi być umieszczona w klasie**, a nie w interfejsie.

```
@Service
@Transactional
public class ShippingProcessor {

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    void markOrderShipped(String orderNumber) {
        ......
    }
}
```

```
@Transactional
public interface CardApi {

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    void addMoneyToCard(String cardUUid, BigDecimal amountToAdd);
}
```

TRANSAKCJE W SPRING – ZASADY DZIAŁANIA

- Realizujemy **wywołania z wykorzystaniem proxy** – jeden komponent używa metody transakcyjnej innego komponentu.

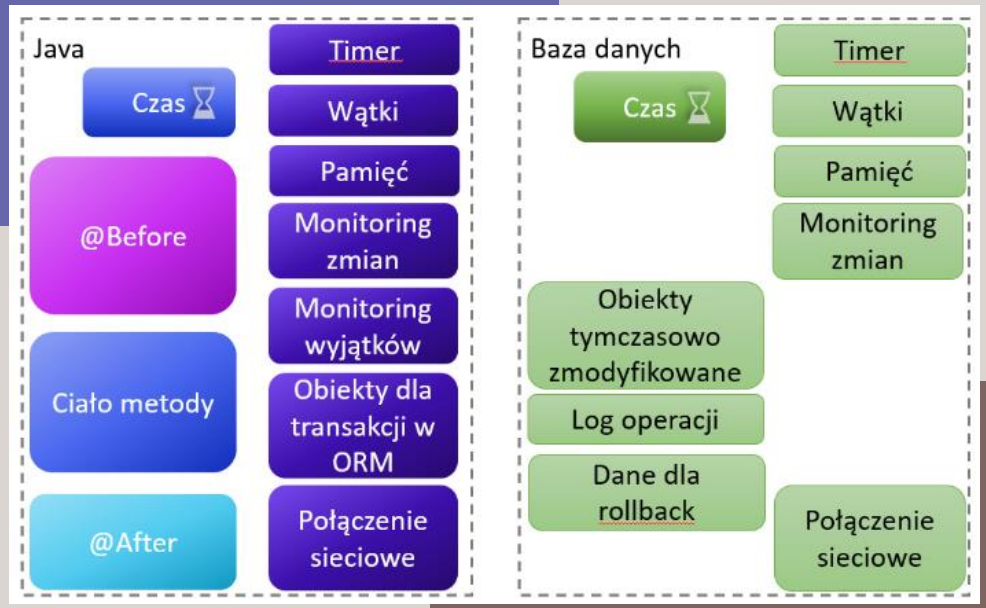
```
1  @Service
2  public class CustomerOrderManager {
3      ..........................
4      CustomerRepository customerRepository;
5
6      void processPaymentForOrder(String orderNumber, String chargedCustom
7          var customer = customerRepository.findById(chargedCustomerId);
8          orderPayments.chargeCardForOrder(customer.getCardId(), orderNumb
9          mailNotifier.sendOrderPaidMessage(customer.getEmailAddress(),orc
10     }
11 }
12
13 @Service
14 @RequiredArgsConstructor
15 public class OrderPayments {
16
17     @Transactional
18     void chargeCardForOrder(String cardUuid, String orderNumber) {
19     ......................
20     }
21 }
```

```
1  @Service
2  public class CardManager {
3
4      @Transactional
5      void lockCard(String cardUuid) {
6              ................
7      }
8
9      void lockAllUserCards(String userId) {
10         findUsersCards(userId)
11                 .forEach(this::lockCard);
12     }
13
14     private List<String> findUserCards(String userId) {
15             ............
16     }
17 }
```

KOSZT TRANSAKCJI W SPRING

Mechanizm transakcji **jest dość kosztowny**. Wymaga on dodatkowych zasobów zarówno po stronie aplikacji Java (**Spring**, **framework ORM**), jak i **serwera bazy danych**. Dodatkowo potrzebny jest czas na wykonanie takich operacji jak:

- **uruchomienie dodatkowych zasobów**
- **zwolnienie dodatkowych zasobów**
- **zatwierdzenie** lub **wycofanie zmian**
- **monitoring** (zmiany, czas trwania, wyjątki)

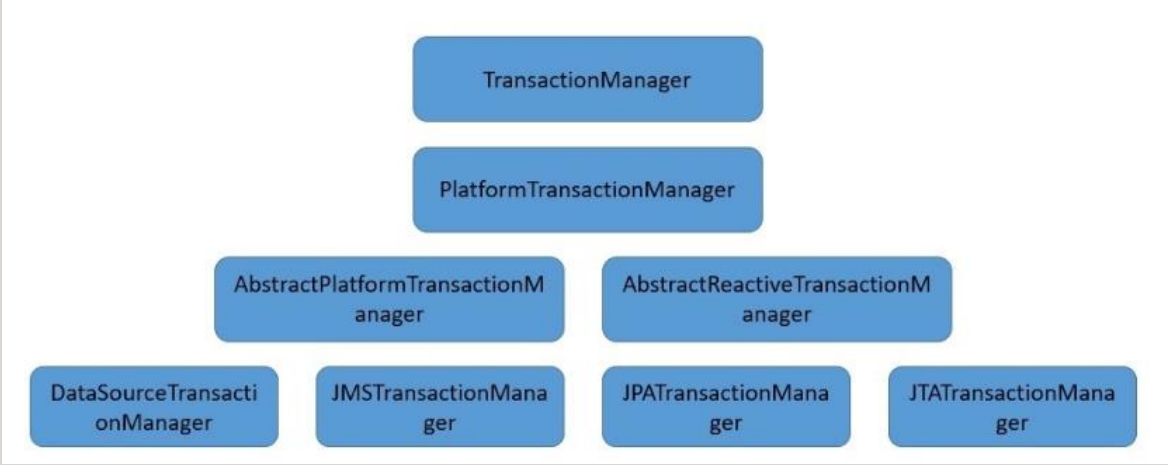


KONFIGURACJA @TRANSACTIONAL

Adnotacja **@Transactional** posiada kilka atrybutów, służących do konfiguracji transakcji:

- **Manager transakcji** – w Spring możliwe jest zdefiniowanie więcej niż jednego managera transakcji.
- **Rodzaj propagacji transakcji** – steruje sposobem wykorzystania transakcji. Najważniejsze to:
  - **REQUIRED**
  - **REQUIRES_NEW**
  - **NOT_SUPPORTED**

MANAGER TRANSAKCJI




```java
public interface PlatformTransactionManager extends TransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

```java
public interface ReactiveTransactionManager extends TransactionManager {

    Mono<ReactiveTransaction> getReactiveTransaction(TransactionDefinition definition) throws TransactionException;

    Mono<Void> commit(ReactiveTransaction status) throws TransactionException;

    Mono<Void> rollback(ReactiveTransaction status) throws TransactionException;
}
```

POZIOMY PROPAGACJI

- **REQUIRED** - Spring sprawdza czy istnieje aktywna transakcja, i jeżeli nie, tworzy ją. W przeciwnym razie logika biznesowa zostaje wykonana w ramach istniejącej transakcji.
- **SUPPORTS** – Spring najpierw sprawdza czy istnieje aktywna transakcja. Jeżeli tak, wtedy jest ona wykorzystywana do wykonania logiki biznesowej, w przeciwnym razie logika wykonana jest poza transakcją.
- **MANDATORY** – Podobnie, jak w przypadku SUPPORTS, Spring najpierw poszukuje aktywnej transakcji, i jeżeli ją znajdzie, to ją wykorzystuje. W przeciwnym razie rzucany jest wyjątek.

```
@Transactional(propagation = Propagation.REQUIRED)
public void requiredExample(String user) {
    // ...
}
```

```
@Transactional
public void requiredExample(String user) {
    // ...
}
```

```
@Transactional(propagation = Propagation.SUPPORTS)
public void supportsExample(String user) {
    // ...
}
```

```
@Transactional(propagation = Propagation.MANDATORY)
public void mandatoryExample(String user) {
    // ...
}
```

POZIOMY PROPAGACJI

- **NEVER** - Spring rzuca wyjątek w przypadku wykrycia aktywnej transakcji.
- **NOT_SUPPORTED** - Jeżeli istnieje aktywna transakcja, Spring przerywa ją, a następnie logika biznesowa wykonywana jest poza transakcją.
- **REQUIRES_NEW** - Podobnie, jak w przypadku NOT_SUPPORTED, Spring przerywa aktywną transakcję, lecz tym razem tworzy nową na potrzebę wykonania logiki biznesowej w odrębnej transakcji.
- **NESTED** – Spring sprawdza czy istnieje aktywna transakcja. Jeżeli tak, to utworzony zostaje punkt zapisu.

```
@Transactional(propagation = Propagation.NEVER)
public void neverExample(String user) {
    // ...
}
```

```
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public void notSupportedExample(String user) {
    // ...
}
```

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void requiresNewExample(String user) {
    // ...
}
```

```
@Transactional(propagation = Propagation.NESTED)
public void nestedExample(String user) {
    // ...
}
```

POZIOMY PROPAGACJI

Propagation Levels

- REQUIRED (default)
  - tx → join tx
  - no → create and run tx
- REQUIRES_NEW
  - tx1 → suspend tx1; create and run tx2; resume tx1
  - no → create and run tx
- SUPPORTS
  - tx → join tx
  - no → nothing
- NOT_SUPPORTED
  - tx → suspend tx; run outside tx; resume tx
  - no → nothing
- MANDATORY
  - tx → join tx
  - no → throw a TransactionRequiredException
- NEVER
  - tx → throw an exception
  - no → nothing

KONFIGURACJA @TRANSACTIONAL

- **Poziom izolacji transakcji** - parametr używany do bardziej zaawansowanego użycia (**musi być wspierany także po stronie bazy danych**). Zmieniając wartość, wskazujemy sposób dostępu do danych z innych transakcji (w tym także niezatwierdzonych). Przykładowo **READ_UNCOMMITTED** pozwala odczytać niezatwierdzone zmiany z innych transakcji.
- Dopuszczalny **maksymalny czas życia transakcji**, po którego przekroczeniu wystąpi wyjątek i wycofanie zmian.

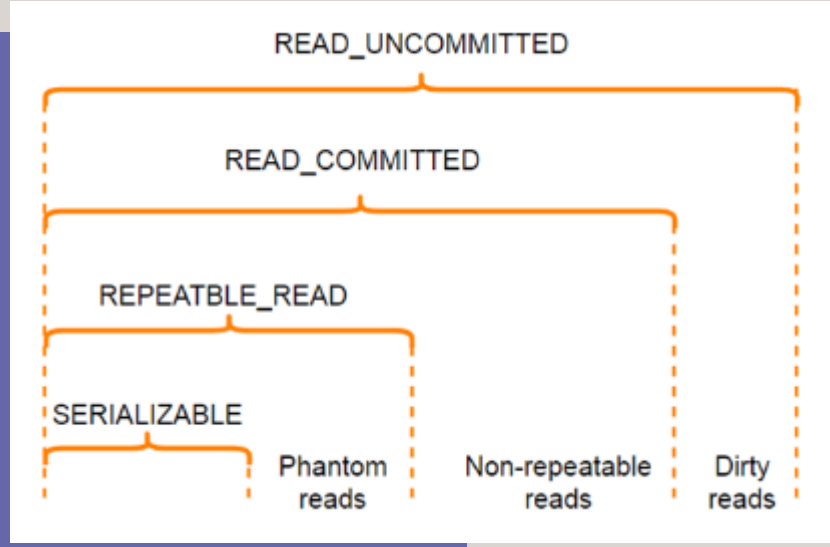
POZIOMY IZOLACJI TRANSAKCJI

- **Brudny odczyt** - polega na dostępie do zaktualizowanej wartości, która nie została jeszcze zatwierdzona
- **Niepowtarzalny odczyt** – ma miejsce wtedy, gdy transakcja A pobiera wiersz, transakcja B go aktualizuje, a następnie transakcja A pobiera ten sam wiersz ponownie W takim przypadku transakcja A widzi różne dane (w dwóch odczytach)
- **Odczyt fantomowy** – sytuacja, gdy transakcja A pobiera zbiór wierszy w danym stanie. Transakcja dodaje nowy wiersz. Transakcja A ponownie odczytuje zbiór i widzi nowy wiersz określany jako fantom.

Poziom izolacji	Brudne odczyty	Niepowtarzalne odczyty	Fantomowe odczyty
READ_UNCOMMITTED	Dozwolone	Dozwolone	Dozwolone
READ_COMMITTED	Uniemożliwione	Dozwolone	Dozwolone
REPEATABLE_READ	Uniemożliwione	Uniemożliwione	Dozwolone
SERIALIZABLE	Uniemożliwione	Uniemożliwione	Uniemożliwione

POZIOMY IZOLACJI TRANSAKCJI

Należy pamiętać, że **zastosowanie wyższego poziomu izolacji wiąże się ze spadkiem wydajności aplikacji**. Poziomy izolacji pozwalają zatem na zachowanie **równowagi między wydajnością, a wymaganą izolacją**. Projektując aplikację trzeba pamiętać także o tym, że różne **SZBD dostarczają różną liczbę poziomów izolacji**. Dla przykładu w MySQL mamy 4 poziomy izolacji, podczas gdy w bazie Oracle poziomy izolacji są tylko 3.




```java
@Transactional(isolation = Isolation.READ_UNCOMMITTED)
public void log(String message) {
    // ...
}
```

```java
@Transactional(isolation = Isolation.READ_COMMITTED)
public void log(String message){
    // ...
}
```

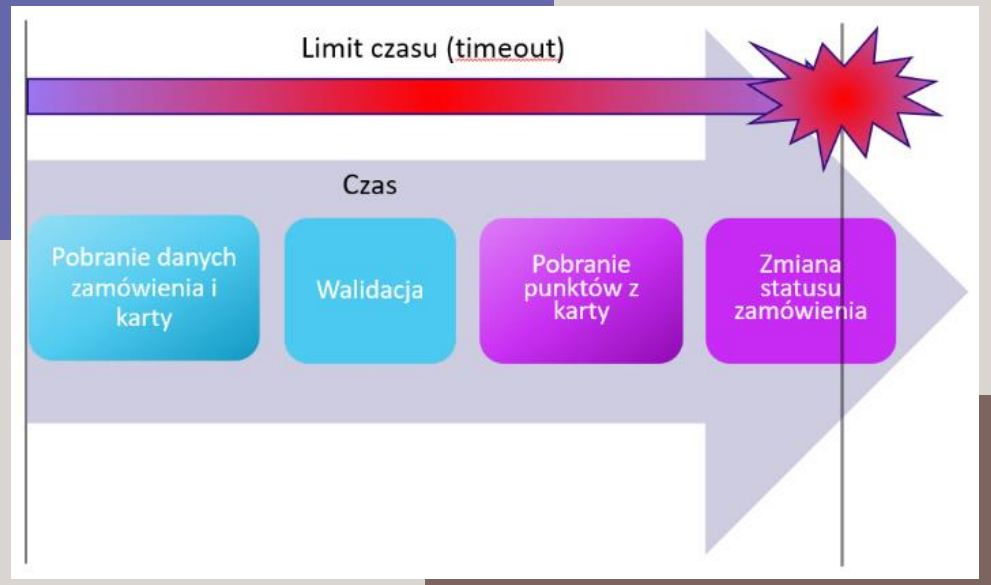
```java
@Transactional(isolation = Isolation.REPEATABLE_READ)
public void log(String message){
    // ...
}
```

```java
@Transactional(isolation = Isolation.SERIALIZABLE)
public void log(String message){
    // ...
}
```

MAKSYMALNY CZAS TRWANIA OPERACJI

Jeżeli chodzi o maksymalny dopuszczalny czas trwania operacji, to możemy wyróżnić następujące ograniczenia:

- Na poziomie adnotacji **@Transactional** – **własność timeout**.
- Dla **JDBC** (dla pojedynczej operacji SQL) – **Statement.setQueryLimit**.
- W **konfiguracji** samej **bazy danych**.



KONFIGURACJA @TRANSACTIONAL

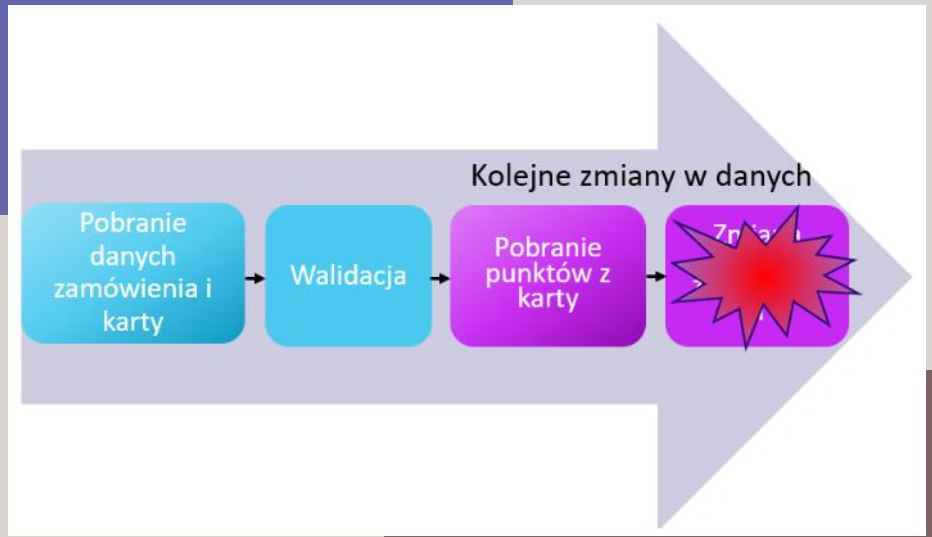
- **Flaga tylko do odczytu**, włączenie skutkuje tym iż w przypadku próby zmiany danych na bazie zostanie rzucony wyjątek,
- **Sterowanie wyjątkami dla wycofywania transakcji** – zbiór atrybutów wskazujący, które klasy wyjątków będą powodowały wycofanie wyjątków, a które nie.

READ ONLY

Adnotacja **@Transactional** posiada parametr **readOnly**. Przy odpowiednim ustawieniu **@Transactional(readOnly = true)** transakcja powinna być tylko do odczytu. Jednak nie jest to prawda. **Dzięki czemu możemy w ten sposób zaoszczędzić trochę pamięci i czasu procesora**.

WYJĄTKI A TRANSACTIONAL

Zakończenie metody transakcyjnej wyjątkiem **może spowodować wycofanie zmian**. Słowo **może** jest tutaj bardzo ważne, gdyż **zależy to od konfiguracji wycofywania transakcji**. Domyślnie Spring wycofuje transakcję, jeśli zostanie rzucony wyjątek **RuntimeException**. Jeśli natomiast rzucilibyśmy wyjątek FileNotFoundException, to zmiany dokonane przed wystąpieniem wyjątku zostaną zatwierdzone.



WYJĄTKI A TRANSACTIONAL

Domyślnie transakcje są **rollbackowane tylko dla UncheckedException**, można to zachowanie zmienić poprzez użycie parametrów **noRollbackFor** i **rollbackFor**. **noRollbackFor** pozwala podać wyjątek (**unchecked**), który **nie z rollbackuje transakcji**. Natomiast **rollbackFor** pozwala podać, który wyjątek (**checked**) spowoduje **rollback transakcji**.

WYJĄTKI A TRANSACTIONAL

Na podstawie dokumentacji:

„While the Spring default behavior for declarative transaction management follows EJB convention(roll back is automatic only on unchecked exceptions), it is often useful to customize this behavior."

`rollbackFor`	Array of `Class` objects, which must be derived from `Throwable.`	Optional array of exception types that must cause rollback.
`rollbackForClassName`	Array of exception name patterns.	Optional array of exception name patterns that must cause rollback.
`noRollbackFor`	Array of `Class` objects, which must be derived from `Throwable.`	Optional array of exception types that must not cause rollback.
`noRollbackForClassName`	Array of exception name patterns.	Optional array of exception name patterns that must not cause rollback.

DEKLARATYWNIE CZY PROGRAMOWO?

Programowe zarządzanie transakcjami:

- pozwala na „zakodowanie” logiki transakcji między logiką biznesową
- jest elastyczne, ale trudne w utrzymaniu z dużą ilością logiki biznesowej
- preferowane, gdy stosowana jest względnie mała logika obsługi transakcji

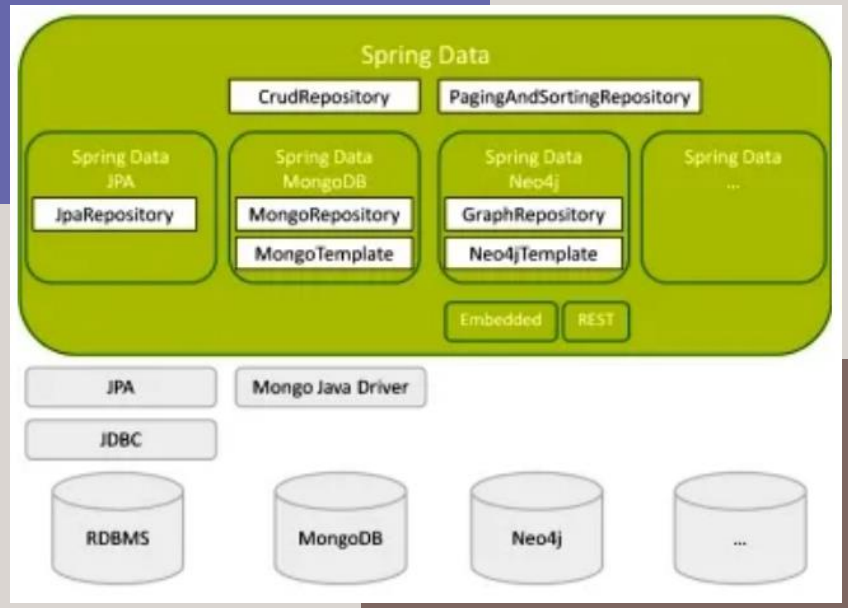
DEKLARATYWNIE CZY PROGRAMOWO?

Deklaratywne zarządzanie transakcjami:

- pozwala na zarządzanie transakcjami poprzez konfigurację
- oznacza oddzielenie logiki transakcji od logiki biznesowej
- używa się adnotacji (lub XML) do zarządzania transakcjami
- łatwe w utrzymaniu – „boilerplate” trzymany jest z dala od logiki biznesowej
- preferowane podczas pracy z dużą ilością logiki obsługi transakcji

REPOZYTORIA JPA

Warstwa bazodanowa, tzn. klasy reprezentujące repozytoria, zazwyczaj posiadają **metody reprezentujące podstawowe operacje CRUD**, których implementacja najczęściej wygląda identycznie. W związku z tym twórcy startera spring-boot-starter-data-jpa dodali funkcjonalność, która **pozwala implementować repozytoria za pomocą definiowania interfejsu**. Implementacja takiego interfejsu zostanie wygenerowana, a dostęp do niej zapewnia **mechanizm proxy**.



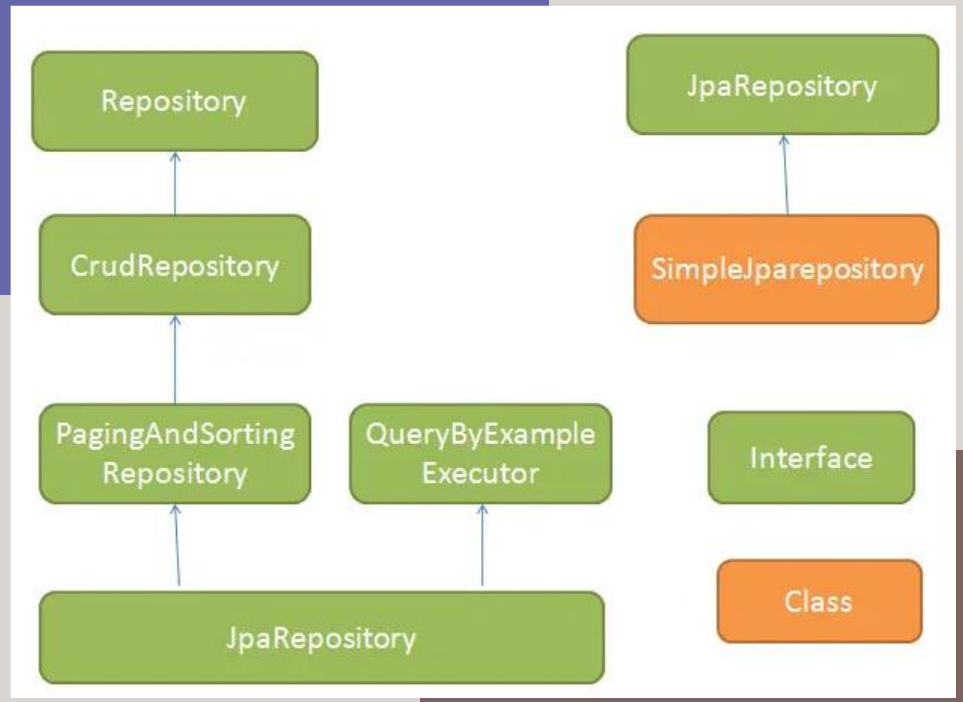
REPOZYTORIA JPA

Mamy możliwość rozszerzenia jednego z trzech interfejsów:

- **CrudRepository**
- **PagingAndSortingRepository**
- **JpaRepository**

Podstawowym interfejsem jest **CrudRepository**, który zapewnia dostęp do metod:

save, saveAll, findById, existsById, findAll, findAllById, count, deleteById, deleteAll



REPOZYTORIA JPA

Warto zwrócić uwagę na metodę **save**, która jest odpowiedzialna za zarówno tworzenie, jak i aktualizację rekordu w bazie. Zwalnia nas więc z konieczności używania metod **merge** (aktualizacja) i **persist** (zapis) dostępnych na obiekcie **EntityManager**.

Pozostałe dwa interfejsy dają dostęp do dodatkowych metod, np.: **findAll(Sort)**, **findAll(Pageable)**, **saveAndFlush**, **getOne**, **deleteInBatch**, **deleteAllInBatch**.

REPOZYTORIA JPA

Rozszerzenie jednego z interfejsów wymaga on nas podania dwóch typów generycznych:

- **typu identyfikatora encji**
- **typu encji**

```java
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "fruits")
public class Fruit {
  @Id
  @GeneratedValue
  private Long id;

  private String name;

  private Double weight;

  public Fruit(final String name, final Double weight) {
    this.name = name;
    this.weight = weight;
  }
}
```

```java
import org.springframework.data.repository.CrudRepository;

public interface FruitRepository extends CrudRepository<Fruit, Long> {
}
```

```java
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class DbInitializer implements CommandLineRunner {

  private final FruitRepository fruitRepository;

  public DbInitializer(final FruitRepository fruitRepository) {
    this.fruitRepository = fruitRepository;
  }

  @Override
  public void run(final String... args) throws Exception {
    Fruit apple = fruitRepository.save(new Fruit("apple", 23.1));
    Fruit banana = fruitRepository.save(new Fruit("banana", 20.1));

    // Fruit(id=1, name=apple, weight=23.1) Fruit(id=2, name=banana, weight=20.1)
    fruitRepository.findAll().forEach(System.out::println);
    System.out.println(fruitRepository.count()); // 2
    fruitRepository.deleteAll(List.of(apple, banana));
    System.out.println(fruitRepository.count()); // 0
  }
}
```

REPOZYTORIA JPA

UWAGA: Repozytoria JPA **nie muszą być oznaczane żadną adnotacją**. Ich implementacja znajdzie się w kontekście, którą możemy wstrzykiwać za pomocą interfejsu.

UWAGA: Nad interfejsami reprezentującymi repozytorium **nie musimy dodawać adnotacji @Transactional**.

TWORZENIE ZAPYTAŃ

Repozytoria JPA posiadają unikalną funkcjonalność. Pozwalają na tworzenie zapytań **za pomocą nazwy metody zdefiniowanej w interfejsie**. Jej implementacja zostanie wygenerowana. Tworząc takie zapytania, musimy pamiętać o pewnych regułach:

- w przypadku pobierania danych nazwa metody musi zaczynać się od słowa **find**, **read**, **query** lub **get** (w przykładach wykorzystywane będzie find). Po tym słowie kluczowym wstawiamy **opcjonalne słowo kluczowe, a następnie słowo By**

TWORZENIE ZAPYTAŃ

- po słowie By podajemy (opcjonalnie) **szczegóły zapytania**, które możemy utożsamiać z częścią zapytania SQL, która **następuje po słowie kluczowym WHERE**.
  - części te powinny wskazywać na **nazwy pól encji**, o której wartość pytamy (nazwy pól zaczynamy wielką literą)
  - gdy chcemy zapytać o wartość kilku pól, **nazwy pól łączymy**, podobnie jak w zwykłym zapytaniu SQL, słowem: **OR** lub **AND**
  - w zależności od tego o wartości ilu pól pytamy, metoda powinna mieć **argument odpowiedniego typu dla każdego z nich** (z zachowaniem kolejności)

```java
import org.springframework.data.repository.CrudRepository;

import java.util.Optional;

public interface FruitRepository extends CrudRepository<Fruit, Long> {
  // SELECT TOP 1 * FROM fruits WHERE weight = ?
  Optional<Fruit> findByWeight(Double weight);

  // SELECT TOP 1 * FROM fruits WHERE name = ?
  Optional<Fruit> findByName(String name);

  // SELECT TOP 1 * fruits WHERE weight = ? AND name = ?
  Optional<Fruit> findByWeightAndName(Double weight, String name);

  // SELECT TOP 1 * fruits WHERE name = ? OR weight = ?
  Optional<Fruit> findByNameOrWeight(String name, Double weight);
}
```

ZWRACANE TYPY

Tworząc zapytania **za pomocą nazw metod**, pomiędzy słowo **find** a **By** możemy wstawić kilka opcjonalnych słów kluczowych, które mają również swoje odpowiedniki w języku SQL, np.:

- **Distinct**
- **All**
- **TopX**, gdzie X jest liczbą naturalną

ZWRACANE TYPY

W zależności od tego, czy zapytanie może zwrócić co najwyżej jeden rekord, lub wiele, powinniśmy odpowiednio dobrać zwracany typ danych. Niektóre z możliwości to:

- w przypadku chęci pobrania **co najwyżej jednego obiektu**:
  - **Optional**
  - **Obiekt encji** (null w przypadku braku rekordu)
- W przypadku pobierania **wielu rekordów**:
  - **List<TypEncji>**
  - **Set<TypEncji>**

```java
import org.springframework.data.repository.CrudRepository;

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

public interface FruitRepository extends CrudRepository<Fruit, Long> {
  Optional<Fruit> findByWeight(Double weight);
  Fruit findByName(String name);
  List<Fruit> findAllDistinctByName(String name);
  List<Fruit> findTop3ByName(String name);
}
```

ZAPYTANIA ZAAWANSOWANE

Oprócz funkcjonalności opisanych wyżej, repozytoria JPA oferują również wiele słów kluczowych, które można użyć, aby jeszcze skuteczniej filtrować pobrane dane (lub wykorzystywać inne słowa kluczowe języka SQL).
Część z nich to: **LessThan, GreaterThan, After, Before, StartingWith, EndingWith, Containing, In, NotIn, Like, NotLike, IgnoreCase, OrderBy**

Słowa te umieszczamy albo **za nazwami pól lub na końcu zapytania** (jak na przykład OrderBy). Reguły są bardzo podobne do tych z języka SQL.

```java
import org.springframework.data.repository.CrudRepository;

import java.util.List;
import java.util.Set;

public interface FruitRepository extends CrudRepository<Fruit, Long> {
  List<Fruit> findAllByNameStartingWithAndWeightBetweenOrderByNameDesc(
      String nameStartingWith, Double minWeight, Double maxWeight);
  List<Fruit> findAllByWeightLessThan(Double upperBound);
  List<Fruit> findFirstByNameContaining(String nameContaining);
  List<Fruit> findAllByNameIn(Set<String> names);
  List<Fruit> findAllByNameLike(String nameLike);
  List<Fruit> countDistinctByNameEndingWith(String nameEnding);
}
```

ZAPYTANIA NIESTANDARDOWE

Czasami istnieje potrzeba napisania zapytania, które jest **bardzo długie** lub, którego **nie da stworzyć się za pomocą nazwy metody** i jesteśmy zmuszeni do wykorzystania składni **HQL**. W tym celu możemy zdefiniować metodę, która ma dowolną nazwę, ale nad którym umieszczamy adnotację **@Query**. Wewnątrz Query umieszczamy treść zapytania HQL.

- **Parametry zapytania** umieszczamy jako argumenty metody (nazwy wewnątrz adnotacji @Query)
- Zapytanie modyfikujące stan bazy danych - **@Modifying**
- Natywne zapytanie: **nativeQuery = true**

```
1  import org.springframework.data.jpa.repository.Query;
2  import org.springframework.data.repository.CrudRepository;
3  import org.springframework.data.repository.query.Param;
4
5  import java.util.List;
6
7  public interface FruitRepository extends CrudRepository<Fruit, Long> {
8
9    @Query(value = "SELECT f FROM fruits f WHERE f.name = :name", nativeQuery = false)
10   List<Fruit> selectAllFruitsByProvidedName(@Param("name") String name);
11 }
12
```

DZIĘKUJĘ ZA UWAGĘ