



Academia Tehnică Militară "Ferdinand I"
Facultatea de Sisteme Informaticе și Securitate Cibernetică

PARADIGME DE PROGRAMARE (ÎN JAVA)

CURS 12 - QUARKUS FRAMEWORK

COL(R) TRAIAN NICULA

TEMATICĂ DE CURS

- Introducere în paradigme de programare (1)
- Programare orientată pe obiecte (OOP) (3)
- Programare funcțională și asincronă (3)
- Programare reactivă (1)
- Servicii web REST (2)
- **Quarkus Framework** (2/4)

CUPRINS CURS

- Logging
- Bean Validation
- Java Persistence API
- Java Transaction API
- Hibernate ORM cu Panache
- Bibliografie



LOGGING

- În procesul de dezvoltare software se folosesc **debuggers** pentru a depista și corecta eventuale probleme
- În producție ne bazăm pe mecanisme de **logging** pentru a depista eventuale bug-uri
- În Java există multe API pentru **logging** care de obicei **nu se integrează bine** unul cu altul
- **Quarkus** integrează următoarele API:
 - **JDK JUL (Java Util Logging)**
 - **JBoss Logging**
 - **SLF4J**
 - **Apache Commons Logging**
- În exemple anterioare am folosit **JBoss Logging** (suportă **@Inject**)



LOGGING

- JBoss Logger pune la dispoziție următoarele **nivele de log**

Nivel	Descriere
FATAL	Problemă critică a serviciul care face imposibilă procesarea de cereri
ERROR	O problemă semnificativă într-o cerere care face imposibil răspunsul
WARN	O eroare necritică ce poate necesita o intervenție imediată
INFO	Evenimente din ciclul de viață al serviciul de obicei cu frecvență scăzută
DEBUG	Informații adiționale care ajută procesul de debugging
TRACE	Informații la nivel de cerere de mare frecvență care pot ajuta procesul de debugging

- API conține metode precum **fatal()**, **info()**, etc. care corespund unui nivel de logging FATAL, INFO etc.

logger.info(invoice);

- Configurația de log a Quarkus se găsește sub spațiu de nume **quarkus.log.*** și are o mulțime de chei (ex. **quarkus.log.level**)



BEAN VALIDATION

- **Java Persistence API (JPA), Java Transaction API (JTA)** și sursele de date asigură un mediu în care **obiectele sunt mapate la o bază de date relațională** prin intermediul tranzacțiilor
- **O aplicație/microserviciu manipulează date**, le validează și le stochează într-o bază de date
- **Validarea bean-urilor** permite **impunerea de constrângeri asupra datelor** astfel încât acestea să fie valide
- **API pentru validarea bean-urilor** se găsește în pachetul **javax.validation**
- **Validarea bean-urilor** vine cu un set de constrângeri proprii dar permite și crearea de constrângeri noi
- Pentru a folosi este necesară **adăugarea** la proiect a extensiei **quarkus-hibernate-validator**
- Prezentăm constrângările prin exemple



BEAN VALIDATION

- Constrângările predefinite pot fi aplicate pe **atribute, colecții, constructori și metode**

```
public class Order {  
    @NotNull  
    @Pattern(regexp = "[CMD][0-9]+")  
    public String orderId;  
    @NotNull  
    @Min(1)  
    public BigDecimal totalAmount;  
    @PastOrPresent  
    public Instant creationDate;  
    @Future  
    public LocalDate deliveryDate;  
    @NotNull  
    public List<OrderLine> orderLines;  
  
    public Order(@PastOrPresent Instant creationDate) {  
        this.creationDate = creationDate;  
    }  
    @NotNull  
    public Double calculateTotalAmount(@Positive Double changeRate) {  
        return 1d;  
    }  
}
```

BEAN VALIDATION

```
// Order continues
public void addOrderLine(OrderLine orderLine) {
    if (this.orderLines == null)
        this.orderLines = new ArrayList<>();
    this.orderLines.add(orderLine);
}
```

```
public class OrderLine {
    public String item;
    public Double unitPrice;
    public Integer quantity;

    public OrderLine() {
    }

    public OrderLine(String item, Double unitPrice, Integer quantity) {
        this.item = item;
        this.unitPrice = unitPrice;
        this.quantity = quantity;
    }
}
```



BEAN VALIDATION

- Cel mai comun mod de adnotare al unui bean este la nivelul atributelor clasei

```
package atm.paradigms;

import javax.validation.constraints.Digits;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Positive;
import javax.validation.constraints.Size;

public class Book {
    @NotNull
    public String title;
    @Digits(integer = 4, fraction = 2)
    public Float price;
    @Size(max = 2000)
    public String description;
    public Integer isbn;
    @Positive
    public Integer nbOfPages;
    @Email
    public String authorEmail;
}
```

Adnotări comune

- Generale @NotNull
- Integer @Positive, @Min, @Max
- Float @Digits, @DecimalMin, @DecimalMax
- String @Size, @Pattern, @Email



BEAN VALIDATION

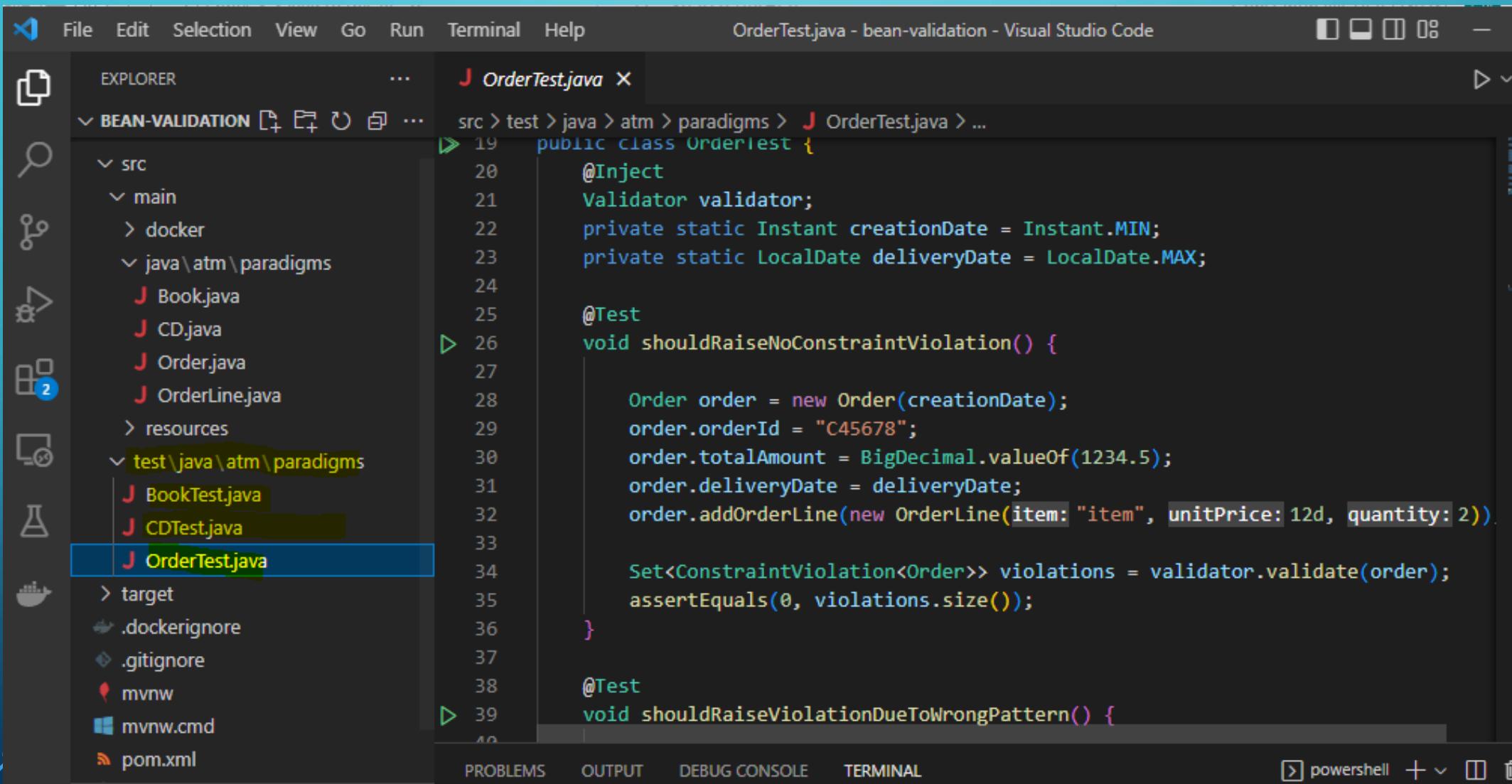
[Cuprins](#)

```
public class CD {  
    @NotNull  
    @Size(min = 4, max = 50)  
    public String title;  
    @NotNull  
    @Positive  
    public Float price;  
    @Size(min = 10, max = 5000)  
    public String description;  
    @Pattern(regexp = "[A-Z][a-z]+")  
    public String musicCompany;  
    @Max(value = 5)  
    public Integer numberOfWorks;  
    public Float totalDuration;  
  
    @NotNull  
    @DecimalMin("5.8")  
    public Float calculatePrice(@DecimalMin("1.4") Float discountRate) {  
        return price * discountRate;  
    }  
  
    @DecimalMin("9.99")  
    public Float calculateVAT() {  
        return price * 0.196f;  
    }  
    ...  
}
```

BEAN VALIDATION

- După adăugarea adnotărilor este necesară validarea acestora folosind interfața `javax.validation.Validator`
- **Validator** se obține programatic sau prin injecție
- Pentru clasele prezentate anterior **vom construi în proiect clase și metode de test** pentru validarea constrângerilor impuse
- Pentru a valida toate proprietățile din bean se creează o instanță pe care se cheamă metoda `Validator.validate()`
- Dacă instanța este validă atunci este returnat un set gol de tip `ConstraintViolation`
- Pentru verificare se folosesc metodele `assert` din `JUnit`

BEAN VALIDATION



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** OrderTest.java - bean-validation - Visual Studio Code.
- Explorer:** Shows the project structure under BEAN-VALIDATION:
 - src
 - main
 - docker
 - java\atm\paradigms
 - Book.java
 - CD.java
 - Order.java
 - OrderLine.java
 - resources
 - test\java\atm\paradigms
 - BookTest.java
 - CDTest.java
 - OrderTest.java (highlighted)
 - target
 - .dockerignore
 - .gitignore
 - mvnw
 - mvnw.cmd
 - pom.xml
 - Editor:** Displays the content of OrderTest.java:

```
OrderTest.java X
src > test > java > atm > paradigms > OrderTest.java > ...
19  public class OrderTest {
20      @Inject
21      Validator validator;
22      private static Instant creationDate = Instant.MIN;
23      private static LocalDate deliveryDate = LocalDate.MAX;
24
25      @Test
26      void shouldRaiseNoConstraintViolation() {
27
28          Order order = new Order(creationDate);
29          order.orderId = "C45678";
30          order.totalAmount = BigDecimal.valueOf(1234.5);
31          order.deliveryDate = deliveryDate;
32          order.addOrderLine(new OrderLine(item: "item", unitPrice: 12d, quantity: 2));
33
34          Set<ConstraintViolation<Order>> violations = validator.validate(order);
35          assertEquals(0, violations.size());
36      }
37
38      @Test
39      void shouldRaiseViolationDueToWrongPattern() {
```
 - Bottom Bar:** PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL.
 - Terminal:** powershell + ▾

BEAN VALIDATION

```
@QuarkusTest
public class OrderTest {
    @Inject
    Validator validator;
    private static Instant creationDate = Instant.MIN;
    private static LocalDate deliveryDate = LocalDate.MAX;

    @Test
    void shouldRaiseNoConstraintViolation() {
        Order order = new Order(creationDate);
        order.orderId = "C45678";
        order.totalAmount = BigDecimal.valueOf(1234.5);
        order.deliveryDate = deliveryDate;
        order.addOrderLine(new OrderLine("item", 12d, 2));

        Set<ConstraintViolation<Order>> violations = validator.validate(order);
        assertEquals(0, violations.size());
    }
    ...
}
```

Clasa de test este anotată cu **@QuarkusTest**. Se injectează **Validator**. Se creează o instanță validă a clasei **Order** și se testează cu metoda **validate()**. Testăm cu **assertEquals()** că nu există încălcări ale constrângerilor. Dacă nu există nici una testul este succes.

BEAN VALIDATION

```
@QuarkusTest
public class BookTest {
    @Inject
    Validator validator;

    . . .

    @Test
    void shouldRaiseViolationDueToEmail() {

        Book book = new Book();
        book.title = "title";
        book.price = 2.99F;
        book.description = "description";
        book.nbOfPages = 10;
        book.authorEmail = "dummy";

        Set<ConstraintViolation<Book>> violations = validator.validate(book);
        assertEquals(1, violations.size());
    }
}
```

O altă metodă de test în care email-ul nu este formatat corect.
În `assertEquals()` ne așteptăm să găsim o singură încălcare a constrângerilor.
Dacă este doar una testul este succes.



BEAN VALIDATION

- Se rulează testele cu comanda **./mvnw test**

[INFO] **Running atm.paradigms.BookTest**

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.024 s - in atm.paradigms.BookTest

[INFO] **Running atm.paradigms.CDTest**

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.165 s - in atm.paradigms.CDTest

[INFO] **Running atm.paradigms.OrderTest**

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.02 s - in atm.paradigms.OrderTest

2022-10-12 13:28:00,413 INFO [io.quarkus] (main) bean-validation stopped in 0.077s

[INFO]

[INFO] Results:

[INFO]

[INFO] **Tests run: 6, Failures: 0, Errors: 0, Skipped: 0**

[INFO]

JAVA PERSISTENCE API

- **Java Persistence API (JPA)** este o specificație Java care se ocupă cu **stocarea obiectelor într-o bază de date relațională**
- **Implementarea JPA folosită de Quarkus este Hibernate ORM**
- **JPA oferă programatorului o perspectivă orientată pe obiecte care permite lucrul cu entități mai curând decât cu tabele**
- **JPA API se găsește în pachetul `javax.persistence` având ca principale interfețe:**

Interfață	Descriere
EntityManager	Principala interfață JPA folosită pentru gestionarea obiectelor persistente
Query și TypedQuery	Interfețe pentru interogarea obiectelor persistente
PersistenceException	Eroare generată de furnizorul de persistentă când apare o problemă



JAVA PERSISTENCE API

- Adnotări JPA comune:

Adnotare	Descriere
@Entity	POJO devine obiect persistent când este adnotat cu @Entity
@Column	Specifică coloana mapată la o proprietate a unui obiect persistent (nume, lungime, unică, etc.)
@GeneratedValue	Definește politica de generare a valorii pentru chei primare
@Id	Specifică cheia primară a unei entități
@Table	Specifică tabelul pentru entitatea adnotată
@Transient	Marchează proprietate ca unui entități ca nepersistentă
@OneToOne, @OneToMany, @ManyToOne, @ManyToMany	Relații între entități

- Pentru a folosi JPA se adaugă la proiect extensia **quarkus-hibernate-orm**



JAVA PERSISTENCE API

- **Bazele relaționale** stochează datele în **tabele alcătuite din rânduri și coloane**
- **Datele** sunt **identificate** prin **chei primare** care identifică unic fiecare rând din tabel
- **Relațiile** dintre tabele folosesc **chei străine**, **tabele pivot (join)** și **constrângeri de integritate**
- Acest **vocabular** este **total necunoscut** pentru **modelul OOP** al Java, care **manipulează obiecte**, instanțe ale claselor
- **Obiectele moștenesc și au referințe** la alte obiecte
 - Pentru a aduce împreună cele 2 modele se utilizează **Object-Relational Mapping (ORM)**
 - **ORM** furnizează o perspectivă orientată pe obiect pentru datele relaționale și invers



JAVA PERSISTENCE API

- Un prim **exemplu de mapare a obiectelor la baza de date folosind Hibernate ORM** este prezentat în continuare (proiect *jpa-initial*).
- **Extensiile Quarkus** necesare sunt:
 - *quarkus-hibernate-orm*
 - *quarkus-jdbc-mysql*
 - *quarkus-narayana-jta*
- **Datele de conexiune** la baza de date sunt specificate în fișierul ***application.properties*** din folder-ul **/main/resources**
- Tot aici se găsește și fișierul ***import.sql*** care conține instrucțiunile SQL pentru crearea schemei din baza de date (se va rula automat)
- În modelul de persistență JPA o entitate este un POJO

JAVA PERSISTENCE API

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "JPA-INITIAL". Key files include ".mvn", ".vscode", "src" (containing "main" with "docker" and "java\atm\paradigms" subfolders, and "resources" with "application.properties" and "import.sql"), "test\java\atm\paradigms" (containing "BookTest.java"), and "target", ".dockerignore", ".gitignore", "mvnw", "mvnw.cmd", "pom.xml", and "README.md".
- Terminal:** Shows the command "application.properties - jpa-initial - Visual Studio Code".
- Code Editor:** Displays the "application.properties" file content:

```
1 # datasource configuration
2 quarkus.datasource.db-kind = mysql
3 quarkus.datasource.username = user
4 quarkus.datasource.password = qaz123QAZ!@#
5 quarkus.datasource.jdbc.url = jdbc:mysql://localhost:3306/hibernate_db
6
7 # drop and create the database at startup
8 quarkus.hibernate-orm.database.generation=drop-and-create
9
```

A yellow callout box points to the line "quarkus.hibernate-orm.database.generation=drop-and-create".

Cu această proprietate rescrierea schemei bazei de date se face la fiecare pornire a Quarkus.
Atenție! În modul dev funcționează reîncărcarea la cald iar scriptul **import.sql** va fi rulat ori de câte ori se fac modificări în cod.

JAVA PERSISTENCE API

- Scriptul pentru crearea schemei este prezentat mai jos
- Se rulează automat dacă este setată proprietatea

quarkus.hibernate-orm.database.generation=drop-and-create

```
DROP TABLE BOOK;
CREATE TABLE BOOK
(
    ID INTEGER NOT NULL PRIMARY KEY,
    DESCRIPTION VARCHAR(255),
    ILLUSTRATIONS TINYINT,
    ISBN VARCHAR(255),
    NBOFPAGES INTEGER,
    PRICE FLOAT,
    TITLE VARCHAR(255)
);
```



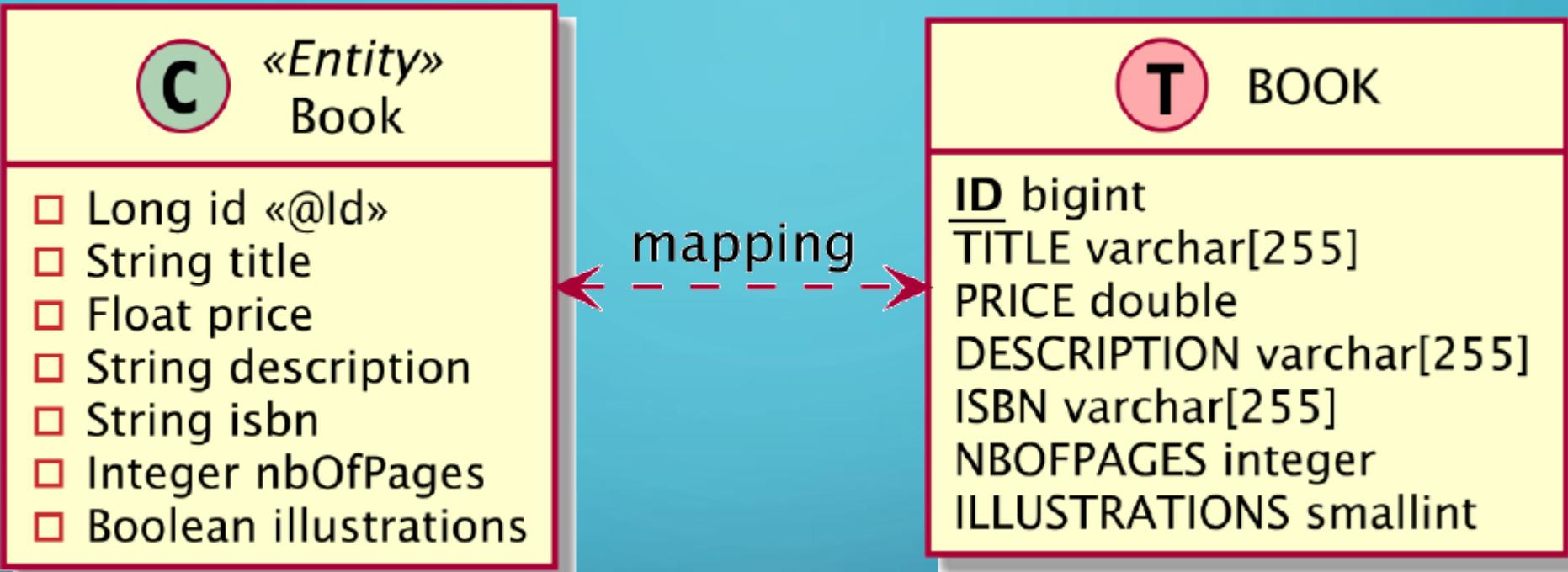
JAVA PERSISTENCE API

- Codul pentru **crearea entității Book** este de fapt o **clasă Java normală la care adăugăm adnotări**
- **Entitatea Book folosește adnotările JPA** pentru ca provider-ul de persistență **să mapeze atributele acesteia la coloanele din tabelul BOOK**

```
package atm.paradigms;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPages;
    private Boolean illustrations;
    // constructors, setters and getters
}
```

JAVA PERSISTENCE API



- Numele entități este mapat la numele tabelului din baza de date MySQL
- Se observă că **numele atributelor sunt mapate la numele coloanelor** din tabel



JAVA PERSISTENCE API

- Pentru a testa persistența folosim o clasă de test **BookTest**
- Se rulează cu comanda **.\mvnw test**

```
@QuarkusTest
public class BookTest {
    @Inject
    EntityManager entityManager;

    @Test
    @Transactional
    void shouldCreateABook() {
        Book book = new Book();
        book.setTitle("Morometzi");
        entityManager.persist(book);
        assertNotNull(book.getId(), "Id should not be null");
    }
}
```

Clasa de test injectează **EntityManager**. În metodă, executată **tranzacțional**, se creează o instanță **Book** și i se atribuie un titlu.

Se persistă entitatea în baza de date cu metoda **persist()**.

Dacă operațiunea reușește entitatea **Book** va primi o valoare pentru **id**, de la valoarea cheii primare.

Metoda **assertNotNull()** verifică dacă acest lucru s-a întâmplat.

JAVA PERSISTENCE API

- Vom prezenta un alt exemplu în care maparea este **customizată** (proiect *jpa-custom*)
- Acum numele **tabelului nu mai coincide cu numele entități și nici numele coloanelor nu mai coincid cu numele atributelor entității**
- Moi mult se impun constrângeri pe coloane care trebuie să se reflecte și la nivelul atributelor (folosim **validare de bean** eu extensia **quarkus-hibernate-validator**)

```
DROP TABLE T_BOOK;
CREATE TABLE T_BOOK
(
    ID INTEGER NOT NULL PRIMARY KEY,
    DESCRIPTION VARCHAR(255),
    ILLUSTRATIONS TINYINT,
    ISBN VARCHAR(255),
    NB_OF_PAGES INTEGER NOT NULL,
    PRICE FLOAT,
    BOOK_TITLE VARCHAR(255) NOT NULL
);
```

JAVA PERSISTENCE API

```
@Entity
@Table(name = "t_book")
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    @NotNull
    @Column(name = "book_title", nullable = false, updatable = false)
    private String title;
    @Min(1)
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    @Column(name = "nb_of_pages", nullable = false)
    private Integer nbOfPages;
    private Boolean illustrations;
    @Transient
    private Instant creationDate;
    // constructors, getters and setters
}
```



JAVA PERSISTENCE API

- Folosim o clasă de test pentru verificarea persistenței

```
@QuarkusTest
public class BookTest {
    @Inject
    EntityManager entityManager;

    @Test
    @Transactional
    void shouldCreateABook() {
        Book book = new Book();
        book.setTitle("Morometzi");
        book.setNbOfPages(345);
        entityManager.persist(book);
        assertNotNull(book.getId(), "Id should not be null");
    }
}
```

The screenshot shows a database interface with the following details:

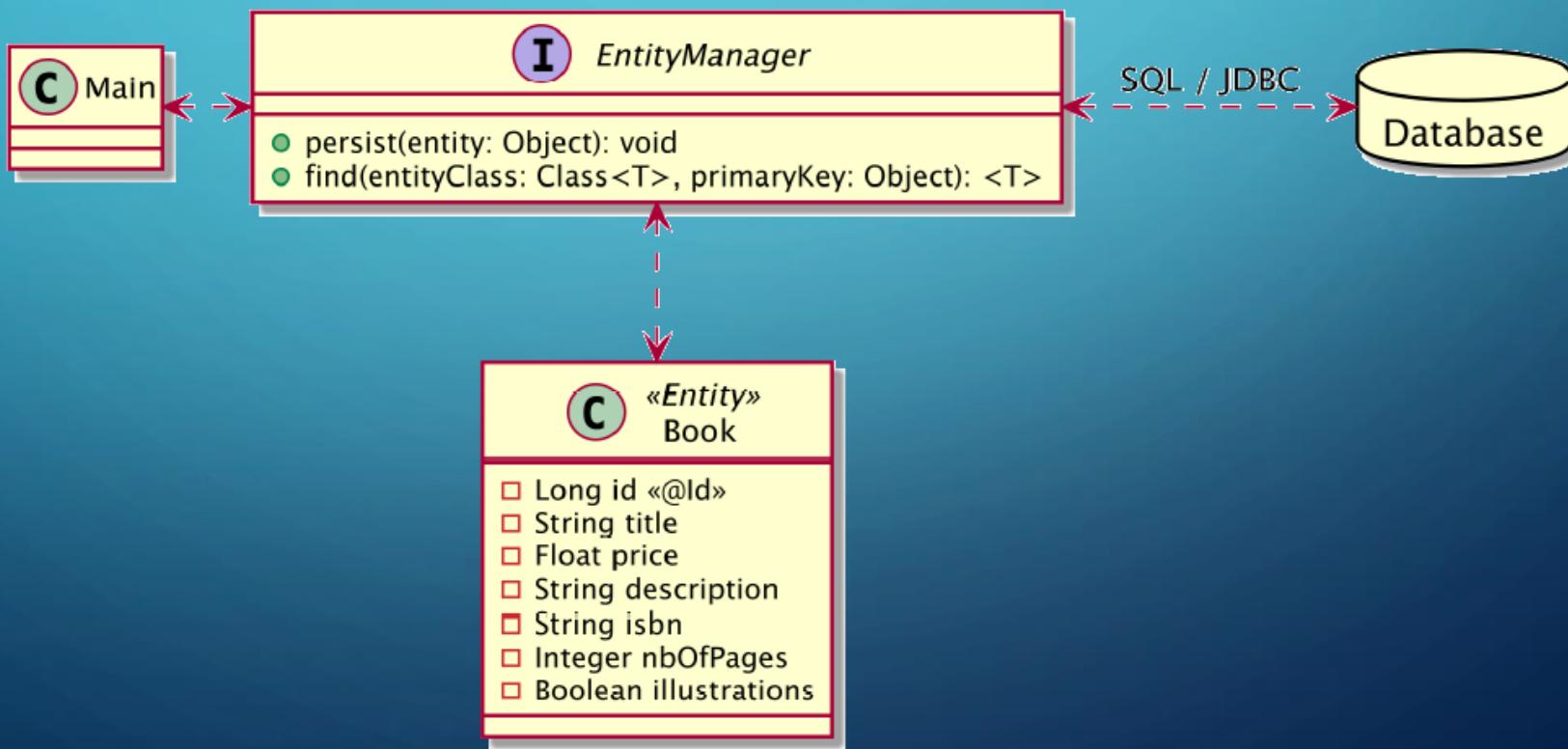
- Table Structure:** The table is named `t_book`. It has columns: ID, DESCRIPTION, ILLUSTRATIONS, ISBN, NB_OF_PAGES, PRICE, and BOOK_TITLE.
- Data:** There is one row of data:

ID	DESCRIPTION	ILLUSTRATIONS	ISBN	NB_OF_PAGES	PRICE	BOOK_TITLE
1	NULL	NULL	NULL	345	NULL	Morometzi



JAVA PERSISTENCE API

- `javax.persistence.EntityManager` are rolul de a **gestiona entitățile**, să le **scrie** și să le **citească** din baza de date, să permită operații **CRUD (create, read, update, and delete)** precum și interogări complexe folosind **JPQL (Java Persistence Query Language)**
- **EntityManager** delegă toate operațiile *low level* către JDBC





JAVA PERSISTENCE API

- Prezentăm un exemplu conținând 2 entități (**Customer** și **Address**) între care există o relație **one-to-one** (proiect **jpa-advanced**)
- Schema** din baza de date va fi **creată de Hibernate ORM pe baza entităților** și nu prin rularea scriptului **import.sql**

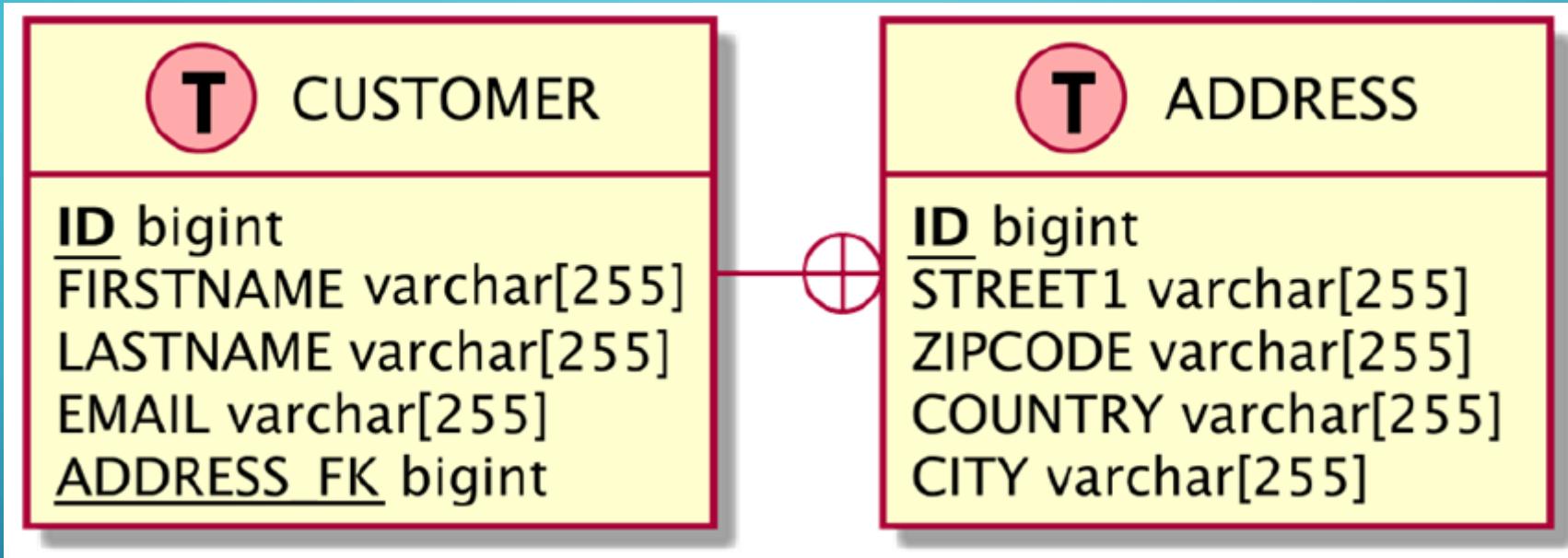
```
@Entity  
public class Address {  
    @Id  
    @GeneratedValue  
    private Long id;  
    private String street;  
    private String city;  
    private String zipcode;  
    private String country;  
    // constructors, getters  
    //and setters  
}
```

```
@Entity  
public class Customer {  
    @Id  
    @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    @OneToOne  
    @JoinColumn(name = "address_fk")  
    private Address address;  
    // constructors, getters and setters  
}
```



JAVA PERSISTENCE API

- Cele 2 entități sunt mapate la baza de date conform schemei



- Persistența entităților înseamnă inserarea în baza de date dacă nu există deja
- Va fi creată o clasă de test CustomerTest pentru testarea persistenței

JAVA PERSISTENCE API

```
@QuarkusTest
public class CustomerTest {
    @Inject
    EntityManager em;

    @Test
    @Transactional
    public void shouldPersistACustomerWithOneAddressSet() {
        Customer customer = new Customer("Anthony", "Balla", "aballa@mail.com");
        Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
        customer.setAddress(address);
        // Persists the object
        em.persist(customer);
        em.persist(address);
        assertNotNull(customer.getId());
        assertNotNull(address.getId());
    }
    ...
}
```

- Se cheamă ***persist()*** pe **ambele entități**, ordinea nu contează
- Până la comiterea tranzacției datele rămân în memorie

JAVA PERSISTENCE API

```
@Test  
@Transactional  
public void shouldFindACustomer() throws Exception {  
    Customer createdCustomer = new Customer("Anthony", "Balla", "aballa@mail.com");  
    Address createdAddress = new Address("Ritherdon Rd", "London", "8QE", "UK");  
    createdCustomer.setAddress(createdAddress);  
    // Persists the object  
    em.persist(createdCustomer);  
    em.persist(createdAddress);  
    em.flush();  
    assertNotNull(createdCustomer.getId());  
    assertNotNull(createdAddress.getId());  
    Long id = createdCustomer.getId();  
    // Clear cache  
    em.clear();  
    Customer customer = em.find(Customer.class, id);  
    assertNotNull(customer);  
}
```

Metoda **flush()** forțează scrierea entităților în baza de date.
Metoda **clear()** șterge entitățile din memorie.
Cu **find()** se caută în baza de date entitatea cu **id** specificat.
Se testează cu **assertNotNull()** dacă există entitatea respectivă.

- Pentru **a găsi o entitate în baza de date** folosim metoda **find()** care primește 2 parametrii: **clasa entității și identificatorul unic (ID)**

JAVA PERSISTENCE API

```
@Test  
public void shouldRemoveACustomer() throws Exception {  
    Customer customer = new Customer("Anthony", "Balla", "aballa@mail.com");  
    Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");  
    customer.setAddress(address);  
    // Persists the object  
    em.persist(customer);  
    em.persist(address);  
    assertNotNull(customer.getId());  
    assertNotNull(address.getId());  
    // Removes the object from the database  
    em.remove(customer);  
    em.remove(address);  
    // The entities are not in the database  
    assertNull(em.find(Customer.class, customer.getId()));  
    assertNull(em.find(Address.class, address.getId()));  
}
```

Se creează, persistă și testează dacă cele 2 entități au primit **id** din baza de date.
Se șterg entitățile cu metoda **remove()**.
Se testează dacă entitățile mai există în baza de date cu **assertNull()**.

- Pentru a **șterge** o entitate din baza de date se folosește metoda **remove()**
- **Entitatea este detașată** de *EntityManager* dar **nu este ștearsă imediat din memorie**

JAVA PERSISTENCE API

The screenshot shows a database management interface with the following components:

- Navigator:** A sidebar on the left containing a "SCHEMAS" section with a search bar labeled "Filter objects". It lists tables: address, book, and customer. The address and customer tables are highlighted in yellow.
- Query Editor:** A top panel with tabs for mybooks, mybooks, mybooks, book, t_book, customer, address, address, and customer. The customer tab is selected. It contains a toolbar with icons for file operations, a search bar for "Limit to 1000 rows", and a query editor with the SQL command: `1 • SELECT * FROM hibernate_db.customer;`
- Result Grid:** A bottom panel titled "Result Grid" showing the results of the query. The columns are id, email, firstName, lastName, and address_fk. The data is as follows:

	id	email	firstName	lastName	address_fk
▶	3	aballa@mail.com	Anthony	Balla	4
▼	5	aballa@mail.com	Anthony	Balla	6
*	NULL	NULL	NULL	NULL	NULL

Text Output: Below the interface, there is a black box containing the following text output from a test run:

```
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.859 s - in
atm.paradigms.CustomerTest
2022-10-14 10:15:21,692 INFO [io.quarkus] (main)
jpa-advanced stopped in 0.046s
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

JAVA PERSISTENCE API

- JPQL (Java Persistence Query Language) este folosit pentru a căuta entități persistente în baza de date
- JPQL are originea în SQL, dar diferența principală este că pentru SQL rezultatele sunt sub formă de rânduri și coloane, pe când pentru JPQL rezultatele sunt entități sau colecții de entități
- Interogare simplă:
SELECT b FROM Book b
- Returnează o listă de zero sau mai multe instanțe ale obiectului **Book**
- Sintaxa JPQL

SELECT <select clause>

FROM <from clause>

[WHERE <where clause>]

[ORDER BY <order by clause>]

[GROUP BY <group by clause>]

[HAVING <having clause>]

JAVA PERSISTENCE API

- JPA permite diferite tipuri de interogări care pot fi folosite în cod
- În exemplele următoare **vom prezenta interogări dinamice**
- Pentru a crea **interogări dinamice** se folosește metoda
EntityManager.createQuery()
- Metoda returnează fie tipul **Query** când rezultatul este de tip **Object**, fie tipul **TypedQuery** când **se așteaptă un anumit tip** preferat
- Pentru a obține rezultatul folosim următoarele metode pe **Query** sau **TypedQuery**:
 - ***getResultSet()*** – returnează o listă
 - ***getResultStream()*** – returnează un stream
 - ***getSingleResult()*** – returnează un singur rezultat sau eroare dacă sunt mai multe

JAVA PERSISTENCE API

- Pentru a explora interogările dinamice folosim proiectul *jpa-query*
- Au fost create 3 entități cu relație **one-to-one**: *Customer > Address > Country*

```
@Entity  
public class Customer {  
    @Id  
    @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    private Integer age;  
    @OneToOne(cascade = {PERSIST})  
    private Address address;  
...}
```

```
@Entity  
public class Address {  
    @Id  
    @GeneratedValue  
    private Long id;  
    private String street;  
    private String city;  
    private String zipcode;  
    @OneToOne(cascade = {PERSIST})  
    private Country country;
```

```
@Entity  
public class Country {  
    @Id  
    @GeneratedValue  
    private Long id;  
    private String code;  
...}
```

JAVA PERSISTENCE API

```
private void initializeData() {  
    customer01 = new Customer("Anthony", "Balla", "tballa@mail.com", 14);  
    Address address01 = new Address("Procession St", "Paris", "75015");  
    Country country01 = new Country("FR");  
    address01.setCountry(country01);  
    customer01.setAddress(address01);  
    em.persist(customer01);
```

În metodă se initializează baza de date.
Se execută o interogare JPQL care
returnează o listă de tip **Customer**, cu
getResultSet(). Se tipărește iterativ
prenumele pentru fiecare client.
Se testează cu **assertEquals()** dacă
dimensiunea listei este cea așteptată.
Se sterg entitățile din baza de date.

```
@Test  
@Transactional  
public void adhocTypedQuery() throws Exception {  
    initializeData();  
    TypedQuery<Customer> typedQuery = em.createQuery(  
        "SELECT c FROM Customer c", Customer.class);  
    List<Customer> customers = typedQuery.getResultSet();  
    for (Customer customer : customers) {  
        System.out.println(customer.getFirstName());  
    }  
    assertEquals(ALL_CUSTOMERS, customers.size());  
    removeData();  
}
```

JAVA PERSISTENCE API

```
@Test  
@Transactional  
public void adhocTypedOneQuery() throws Exception {  
    initializeData();  
    TypedQuery<Customer> typedQuery = em.createQuery(  
        "SELECT c FROM Customer c WHERE c.firstName = 'Mike'", Customer.class);  
    Customer customer = typedQuery.getSingleResult();  
    assertEquals("Mike", customer.getFirstName());  
    removeData();  
}  
  
@Test  
@Transactional  
public void adhocTypedQueryStream() throws Exception {  
    initializeData();  
    TypedQuery<Customer> typedQuery = em.createQuery(  
        "SELECT c FROM Customer c", Customer.class);  
    Stream<Customer> customers = typedQuery.getResultStream();  
    customers.map(c -> c.getFirstName()).forEach(System.out::println);  
    removeData();  
}
```

Clienții din baza de date sunt returnați ca **Stream<Customer>**.

Folosind Stream API se tipărește iterativ prenumele pentru fiecare client.

JAVA PERSISTENCE API

```
@Test  
    @Transactional  
    public void adhocQueryParam() throws Exception {  
        initializeData();  
        TypedQuery<Customer> typedQuery = em.createQuery(  
            "SELECT c FROM Customer c WHERE c.firstName = :fname", Customer.class);  
        typedQuery.setParameter("fname", "Mike");  
        List<Customer> customers = typedQuery.getResultList();  
        assertEquals(1, customers.size());  
        removeData();  
    }  
}
```

Interogare JPQL cu un parametru.
Valoarea parametrului se trece cu metoda
`setParameter()`.

[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.561 s - in

atm.paradigms.DynamicQueriesTest

2022-10-14 13:06:04,150 INFO [io.quarkus] (main) jpa-query stopped in
0.019s

[INFO]

[INFO] Results:

[INFO]

[INFO] **Tests run: 6, Failures: 0, Errors: 0, Skipped: 0**

[INFO]

JAVA TRANSACTION API

- Dacă JPA se ocupă cu maparea entităților la tabele, gestionarea și interogarea lor, **Java Transaction API (JTA)** se ocupă cu **managementul entităților în mod tranzacțional**
- **JTA API** se găsește în pachetul ***javax.transaction***
- JTA vine cu un set de annotări:
- **@Transactional** – permite **controlul tranzacției în mod declarativ**
- **@TransactionScoped** – permite definirea de **instanțe bean care există doar pe timpul tranzacției**
- JTA este implementat în extensia Quarkus **quarkus-narayana-jta**, care se adaugă la proiect



JAVA TRANSACTION API

- **Tranzacțiile** se folosesc pentru **păstrarea integrității datelor**
- Ele reprezintă un **grup de operații** care trebuie **executate ca o singură unitate**
- Aceste operații pot **însemna persistența datelor** în una sau mai multe baze de date, **trimiterea de mesaje la MOM (Message-Oriented Middleware)**, sau **invocarea de servicii web**
- Aceste **operații de business** sunt executate secvențial sau paralel într-un interval scurt de timp
- **Fiecare operație trebuie să reușească** pentru ca **tranzacția să fie comisă (commit)**
- Dacă **cel puțin una din operații eșuează**, **tranzacția eșuează** și se produce **rollback**

JAVA TRANSACTION API

- La **utilizarea declarativă** a tranzacțiilor **containerul** se ocupă de **gestionarea și stabilirea limitelor** tranzacției
- În proiectul *jta-initial* am creat o clasă serviciu tranzacțional. Adnotarea **@Transactional** aplicată clasei *BookService* face ca **invocarea metodelor să fie interceptată și tratată ca o tranzacție**
- Astfel **invocarea metodei *createBook()*** containerul **începe o nouă tranzacție**
- La **ieșirea** din metodă **comite** tranzacție sau face **rollback** dacă sunt erori
- Pentru a avea **un control mai mare** pe modul de execuție al tranzacțiilor folosim atributele adnotării **@Transactional**

JAVA TRANSACTION API

```
@ApplicationScoped  
@Transactional  
public class BookService {  
    @Inject  
    EntityManager em;  
  
    public List<Book> findBooks() {  
        return em.createQuery("SELECT b FROM Book b", Book.class).getResultList();  
    }  
  
    public Book createBook(Book book) {  
        em.persist(book);  
        return book;  
    }  
}
```

Adnotarea **@Transactional** fiind aplicată la nivelul clasei, la invocarea metodei **createBook()** containerul inițiază o tranzacție.



JAVA TRANSACTION API

- Testarea serviciului se face cu clasa de test **BookServiceTest**

```
@QuarkusTest
public class BookServiceTest {

    @Inject
    BookService bookService;

    @Test
    void shouldAddABook() {
        // persist book
        Book book = new Book();
        book.setTitle("Morometzi");
        book.setIllustrations(false);
        book.setPrice(24.5f);
        book.setNbOfPages(255);

        book = bookService.createBook(book);
        assertEquals(1, bookService.findBooks().size());
    }
}
```

În metoda de test nu este necesară inițierea unei noi tranzacții, deoarece metoda **createBook()** se execută deja într-un context tranzacțional.



JAVA TRANSACTION API

- Atributele adnotării `@Transactional`:

Atribut	Descriere
REQUIRED	Atribut implicit care spune că metoda trebuie să fie întotdeauna invocată într-o tranzacție. Containerul creează o tranzacție nouă dacă este chemată dintr-un client netranzacțional (clasa de test). Dacă clientul (clasa de test) are un context tranzacțional metoda rulează în tranzacția client
REQUIRES_NEW	Containerul întotdeauna creează o tranzacție nouă indiferent dacă clientul are un context tranzacțional
SUPPORTS	Metoda tranzacțională moștenește contextul tranzacțional al clientului. Dacă există un context tranzacțional acesta este folosit dacă nu metoda este invocată fără context tranzacțional
MANDATORY	Containerul are nevoie de un context tranzacțional înainte de a chama metoda dar nu creează unul dacă lipsește
NOT_SUPPORTED	Metoda nu poate fi chemată într-un context tranzacțional. Dacă există îl suspendă, cheamă metoda, apoi reia tranzacția
NEVER	Metoda nu trebuie chemată într-un context tranzacțional. Dacă clientul are un context tranzacțional containerul generează excepție



JAVA TRANSACTION API

[Cuprins](#)

- Prezentăm un exemplu de clasă serviciu **PublisherService** cu **context tranzacțional delimitat de atribute** (proiect *jta-initial*)

```
@Entity
public class Publisher {
    @Id
    @GeneratedValue
    private Long id;
    @Column(length = 30)
    private String name;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

JAVA TRANSACTION API

```
@ApplicationScoped  
@Transactional(SUPPORTS)  
public class PublisherService {  
    @Inject  
    EntityManager em;  
  
    @Transactional(REQUIRED)  
    public Publisher persist(Publisher publisher) {  
        em.persist(publisher);  
        return publisher;  
    }  
  
    @Transactional(REQUIRED)  
    public Publisher update(Publisher publisher) {  
        return em.merge(publisher);  
    }  
  
    @Transactional(REQUIRED)  
    public void deleteById(Long id) {  
        em.remove(em.find(Publisher.class, id));  
    }  
    ...  
}
```

Modul de utilizare contextului tranzacțional delimitat de attribute.
La nivelul clasei avem atributul **SUPPORTS** care propagă conținutul tranzacțional delimitat de **REQUIRED**.
În cazul interogărilor nu este necesar context tranzacțional.

JAVA TRANSACTION API

```
    . . .
    @Transactional(REQUIRED)
    public int deleteByName(String name) {
        int rowsDeleted = em.createQuery("DELETE FROM Publisher p WHERE p.name = :name ")
            .setParameter("name", name)
            .executeUpdate();
        return rowsDeleted;
    }

    public List<Publisher> findAll() {
        return em.createQuery("SELECT p FROM Publisher p", Publisher.class).getResultList();
    }
}
```

JAVA TRANSACTION API

```
@QuarkusTest  
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)  
public class PublisherServiceTest {  
  
    @Inject  
    PublisherService publisherService;  
  
    private static long publisherId;  
  
    @Test  
    @Order(1)  
    void shouldAddAnPublisher() {  
        Publisher publisher = new Publisher();  
        publisher.setName("First publisher");  
        publisher = publisherService.persist(publisher);  
        assertEquals(1, publisherService.findAll().size());  
        publisherId = publisher.getId();  
    }  
    ...  
}
```

Modalitate de executare în ordine a metodelor de test cu adnotările **@TestMethodOrder** și **@Order**. Se utilizează contextul tranzacțional stabilit de client, respectiv metodele clasei **PublisherService**.

JAVA TRANSACTION API

```
...
@Test
@Order(2)
void shouldUpdateAPublisher() {
    Publisher publisher = new Publisher();
    publisher.setId(publisherId);
    publisher.setName("Updated Publisher");
    // Updates the previously created publisher
    publisherService.update(publisher);
    assertEquals(1, publisherService.findAll().size());
}

@Test
@Order(3)
void shouldRemoveAPublisher() {
    // Deletes the previously created publisher
    publisherService.deleteById(publisherId);
    // Checks there is less a publisher in the database
    assertEquals(0, publisherService.findAll().size());
}
```

HIBERNATE ORM CU PANACHE

- Am văzut că **Quarkus** suportă **JPA** și **JTA**
- **Hibernate ORM** este **implementarea JPA folosită de Quarkus** care suportă mapări și interogări complexe. Cu toate acestea **nu este simplu de utilizat**
- **Hibernate ORM cu Panache** simplifică lucrul cu entitățile JPA
- Pentru a o utiliza **se adaugă la proiect extensia quarkus-hibernate-orm-panache**
- **Entitățile Panache încapsulează** atât **datele** cât și **comportamentul**



HIBERNATE ORM CU PANACHE

- Exemplu de entitate *Panache*:

```
@Entity
public class Publisher extends PanacheEntity {
    @Column(length = 30)
    public String name;

    public static Optional<Publisher> findByName(String name){
        return find("name", name).firstResultOptional();
    }
    public static long deleteByName(String name){
        return delete("name", name);
    }
}
```

Diferențe față de implementarea JPA:

- **Atributele sunt publice** fără getters și setters
- **Identifierul (id) lipsește** deoarece nu este relevant pentru model, așa că este **definit de superclasa (PanacheEntity)**



HIBERNATE ORM CU PANACHE

- *Starea și comportamentul (behaviour) sunt definite în aceeași clasă* (atributul name și metoda `findByName()`)
- *Interogări simplificate* - JPQL este un limbaj bogat și puternic dar cu **multă verbozitate**

Reguli de urmat pentru utilizarea Hibernate ORM cu Panache:

- *Entitățile trebuie să extindă PanacheEntity*. Câmpul `id` este autogenerat
- **Câmpurile clasei sunt publice**
- **Logica de procesare a entităților se implementează prin metode statice în clasa entitate**

HIBERNATE ORM CU PANACHE

A*PanacheEntityBase*

- delete(): void
- flush(): void
- isPersistent(): boolean
- persist(): void
- persistAndFlush(): void
- deleteAll(): long
- deleteById(Object id): boolean
- findById(Object id): PanacheEntityBase
- findByIdOptional(Object id): Optional<PanacheEntityBase>
- persist(Iterable<?> entities): void
- persist(Stream<?> entities): void
- ...

A*«@MappedSuperclass»
PanacheEntity*

- id: Long «@Id» «@GeneratedValue»

C*«Entity»
Publisher*

- name: String

HIBERNATE ORM CU PANACHE

- În JPA am folosit interfața *EntityManager* pentru a interacționa cu entitățile
- **Entitățile Panache**, datorită superclasei *PanacheEntityBase* conțin majoritatea operațiilor **CRUD** în clasa însăși
- Testăm cu clasa de test *PublisherTest* (proiect panache-entities)

```
package atm.paradigms.model;
import java.util.Optional;
import javax.persistence.Column;
import javax.persistence.Entity;
import io.quarkus.hibernate.orm.panache.PanacheEntity;
@Entity
public class Publisher extends PanacheEntity {
    @Column(length = 30)
    public String name;
    public static Optional<Publisher> findByName(String name){
        return find("name", name).firstResultOptional();
    }
    public static long deleteByName(String name){
        return delete("name", name);
    }
}
```

HIBERNATE ORM CU PANACHE

```
@QuarkusTest
@Transactional
public class PublisherTest {
    @Test
    void shouldManagePublishers() {
        // Creates a publisher
        Publisher publisher = new Publisher();
        publisher.name = "AGoncal Fascicle";
        publisher.persist();
        Long publisherId = publisher.id;
        assertNotNull(publisherId);
        // Gets a list of all publisher entities
        List<Publisher> allPublishers = Publisher.listAll();
        assertTrue(allPublishers.size() >= 1);
    ...
}
```

HIBERNATE ORM CU PANACHE

```
    . . .
    // Finds a specific publisher by ID
    publisher = Publisher.findById(publisherId);
    assertEquals("AGoncal Fascicle", publisher.name);
    // Counts all publishers
    long countAll = Publisher.count();
    assertTrue(countAll >= 1);
    // Checks if it's persistent
    if (publisher.isPersistent()) {
        // Deletes it
        publisher.delete();
    }
    // Deletes by id
    boolean deleted = Publisher.deleteById(publisherId);
    assertFalse(deleted);
}
```

Metodele corespunzătoare operațiilor **CRUD** se găsesc în clasă.
Nu avem nevoie de **EntityManager**.

HIBERNATE ORM CU PANACHE

- Panache dă posibilitatea de a scrie **interrogări JPQL** la fel ca pentru JPA dar și **simplificate**
- Metoda ***list()*** poate lua **fragment** de JPQL și contextualizează restul
- Exemple în clasa ***BookTest*** din proiectul ***panache-entities***

```
@Entity
public class Book extends PanacheEntity {
    @Column(length = 100)
    public String title;
    @Column(name = "unit_cost")
    public Float unitCost;
    @Column(length = 3000)
    public String description;
    @Column(length = 15)
    public String isbn;
    @Column(name = "nb_of_pages")
    public Integer nbOfPages;
    @Column(name = "publication_date")
    public Instant publicationDate;
    @Enumerated(EnumType.STRING)
    public Language language;
    @ManyToOne
    @JoinColumn(name = "publisher_pk")
    public Publisher publisher;
    ...
}
```

HIBERNATE ORM CU PANACHE

```
@QuarkusTest
@Transactional
public class BookTest {
    @Test
    void shouldFind() {
        // find() returns a PanacheQuery
        PanacheQuery<Book> bookQuery = Book.find("nbOfPages > 100 ORDER BY title");
        List<Book> books = bookQuery.list();
        Long nbBooks = bookQuery.count();
        Book firstBook = bookQuery.firstResult();
        Optional<Book> oBook = bookQuery.firstResultOptional();
        assertEquals(5, books.size());
        assertEquals(5, nbBooks);
        assertEquals("Beginning Java EE 7", firstBook.title());
        assertEquals("Beginning Java EE 7", oBook.get().title());
        // list() is a shortcut to find().list()
        books = Book.find("nbOfPages > 100 ORDER BY title").list();
        assertEquals(5, books.size());
        books = Book.list("nbOfPages > 100 ORDER BY title");
        assertEquals(5, books.size());
    }
    ...
}
```

HIBERNATE ORM CU PANACHE

```
@Test
void shouldQueryWithParameters() {
    float min = 0f;
    float max = 30f;
    List<Book> cheapBooks;
    // Hard coded parameters
    cheapBooks = Book.list("unitCost between 0 and 30");
    assertEquals(1, cheapBooks.size());
    // Positional parameters
    cheapBooks = Book.list("unitCost between ?1 and ?2", min, max);
    assertEquals(1, cheapBooks.size());
    // Named parameters
    Map<String, Object> params = Map.of("min", min, "max", max);
    cheapBooks = Book.list("unitCost between :min and :max", params);
    assertEquals(1, cheapBooks.size());
    // Using the Parameters class
    cheapBooks = Book.list("unitCost between :min and :max",
        Parameters.with("min", min).and("max", max));
    assertEquals(1, cheapBooks.size());
    // Passing an enumeration
    List<Book> englishBooks = Book.list("language", Language.ENGLISH);
    assertEquals(4, englishBooks.size());
}
```

BIBLIOGRAFIE

- **Understanding Quarkus**, Antonio Goncalves, Independently Published, 2020
- **Quarkus Cookbook**, Alex Soto Bueno and Jason Porter, O'Reilly Media, 2020
- **Beginning Quarkus Framework**, Tayo Koleoso, Apress, 2020
- [Quarkus Guides -Latest](#)