Java Persistence API Course

Java Persistence API

Introduction

- Who is your trainer today?
- Who are you?
- What do you expect to learn?

Set up

JPA Basics

- Entity classes
- EntityManager
- Querying/modifying entities

JPA Advanced

- Associations
- Inheritance
- JPA Lifecycle Events

- JSR 380 Bean validations

Spring Data JPA

- Spring Data JPA Configuration
- Automatic schema generation
- Spring Data JPA Repositories
- Solving concurrency issues
- Spring Data JPA Auditing
- Database migrations
- Pitfalls
- Extra content

What is a database?

Relational Database

Table 1. Example database table for a Person entity

ID	**NAME**	**SURNAME**	**DOB**
1	Jon	Snow	01-01-1980
2	Donald	Duck	03-08-1954
…	…	…	…

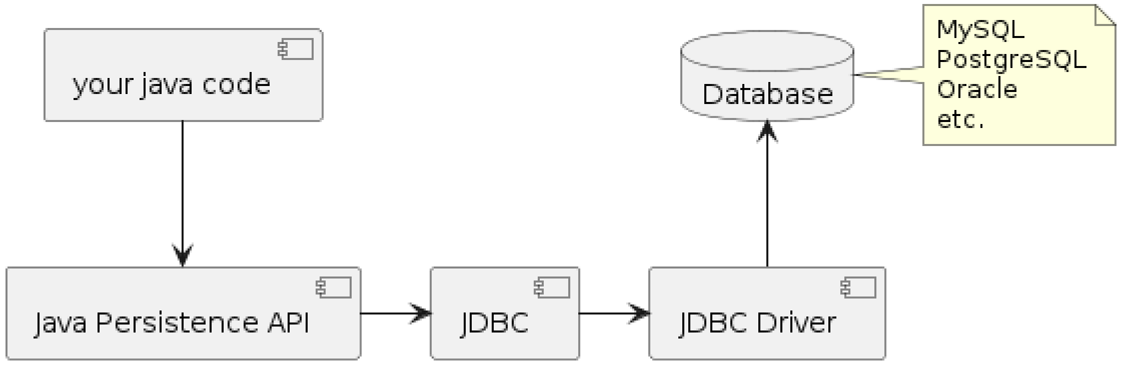
ID	NAME	SURNAME	DOB
999	Peter	Parker	10-08-2001

Assignment 01

JPA Basics

What is JPA?

- JPA is a Java specification for object-relational mapping (ORM) in Java.
- JPA provide a way to map Java objects (entities) to database tables.
- Most popular implementation of JPA is Hibernate



How does JPA actually do it?

Defining your entity classes

- JPA works with standard POJOs (Plain old Java Objects).
- Entity class describes a row in the database and is filled with data by JPA
- Configuring entity classes:
    - should have at least a public empty constructor
    - should have getters and setters
    - should be properly annotated with `@Entity` and `@Id`

Defining your entity classes

```
@Entity
@Table(name = "persons") // optionally specify custom table name
public class Person {
    @Id
    @GeneratedValue
    private final Long id;
    private final String name;
    @Column(name = "dob") // optionally specify custom column name
    private final Date dateOfBirth;

    public Person() {}
}
```

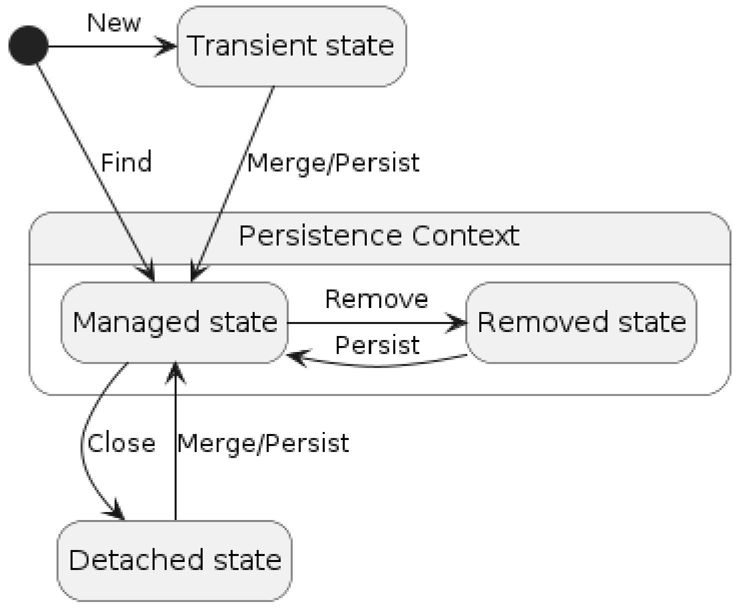
JPA Entity Annotations Demo

Assignment 02

Querying the database

- EntityManager is JPA-provided interface to perform CRUD operations on database entities and manage their lifecycle
- What is meant by lifecycle?

Lifecycle



EntityManager operations

- `entityManager.find(class, primaryKey)` - find entity by key

- `entityManager.persist(object)` - make an entity managed and persisted
- `entityManager.merge(object)` - merge entity into a current persistence context
- `entityManager.remove(object)` - remove entity

persist(Object obj) vs <T> merge(<T> obj)

void persist(Object entity): Make an instance managed and persistent.

Here, `entity` is an object of a class that is mapped to a table in database. It should be a new, unmanaged entity instance. After the persist operation executed successfully:

- a new row would be inserted into the corresponding table in database.
- the entity object becomes a managed instance or persistent object.
- the entity manager track changes of persistent objects.
- Any changes made to the mapped fields of the entity object will be synchronized with the database, provided that the changes happen within the transaction boundary.

```
@Transactional
   public void testPersistNewObject() {

       Contact newContact = new Contact();
       newContact.setName("John Doe");
       newContact.setEmail("john.doe@gmail.com");
       newContact.setAddress("Fremont, CA");
       newContact.setPhone("123456-2111");

       boolean entityManaged = entityManager.contains(newContact);
```

```
        System.out.println("Before persist, entity managed: " + entityManaged);

        entityManager.persist(newContact);

        entityManaged = entityManager.contains(newContact);
        System.out.println("After persist, entity managed: " + entityManaged);
    }
```

```
Before persist, entity managed: false
Hibernate: insert into...
After persist, entity managed: true
```

<T> T merge(T entity): Merge the state of the given entity into the current persistence context

Here, `entity` can be an unmanaged or persistent entity instance. This method returns the managed instance that the state was merged to.

After the merge operation executed successfully:

- a new row would be inserted, or an existing one would be updated.
- the returned object is different than the passed object.
- if the passed object is unmanaged, it says unmanaged.
- the entity manager tracks changes of the returned, managed object within transaction boundary.

Now, let’s see some code examples. Given the following code that is used to update an existing contact:

```
@Transactional
public void testUpdateExistingObject() {
    Contact existContact = new Contact();
    existContact.setId(13);
    existContact.setName("Nam Ha Minh");
    existContact.setEmail("namhm@codejava.net");
    existContact.setAddress("Tokyo, Japan");
    existContact.setPhone("123456-2111");

    Contact updatedContact = entityManager.merge(existContact);

    boolean passedObjectManaged =  entityManager.contains(existContact);
    System.out.println("Passed object managed: " + passedObjectManaged);

    boolean returnedObjectManaged = entityManager.contains(updatedContact);
    System.out.println("Returned object managed: " + returnedObjectManaged);
}
```

```
Hibernate: select ... from contact contact0_ where...
Passed object managed: false
Returned object managed: true
Hibernate: update contact set address=?,...
```

You see, this means the merge operation firstly selects a row from the database according to ID field of the entity object, then it merges the changes from the code with the one in database, which results in the SQL update statement.

NOTE | you can also use the `merge()` method to persist a new, unmanaged entity instance like the `persist()` method.

EntityManager Demo

JPA Persistence context configuration

- JPA Persistence context is configured via `META-INF/persistence.xml`
- Persistence context contains the following configuration:
  - Entity classes mapping
  - Data source config (via JDBC)
  - JPA Library-specific configuration (e.g. Hibernate)

JPA Persistence context configuration Demo

JPA Queries

JPA EntityManager allows you to perform database queries in several ways:

- Query, written in Java Persistence Query Language (JPQL) syntax
- NativeQuery, written in plain SQL syntax
- Criteria API Query, constructed programmatically via different methods

JPA Queries: Query

JPA Query is a special query using JPQL syntax, which is similar to plain SQL, but has Java specifics.

```
public class CustomerService {
    public List<Customer> findAll() {
        final Query query = entityManager
```

```
                .createQuery("SELECT c FROM Customer c", Customer.class);

        return query.getResultList();
    }
}
```

JPA Queries: NativeQuery

JPA NativeQuery allows to use plain SQL

```
public class CustomerService {
    public List<Customer> findAll() {
        final Query query = entityManager
                .createNativeQuery("SELECT * FROM customer", Customer.class);

        return query.getResultList();
    }
}
```

JPA Queries: using query parameters

JPA supports named or positional parameters in queries

```
public class CustomerService {
    public Customer getById(final Integer id) {
// final Query query = entityManager.createQuery("SELECT c FROM Customer c WHERE c.id = " + id,
Customer.class); // not recommended, avoid
        final Query query = entityManager
                .createQuery("SELECT c FROM Customer c WHERE c.id = :id", Customer.class);
        return query
```

```
                .setParameter("id", id)
                .getSingleResult();
    }

    public Customer getById(final Integer id) {
        final Query query = entityManager
                .createQuery("SELECT c FROM Customer c WHERE c.id = ?1", Customer.class);
        return query
                .setParameter(1, id)
                .getSingleResult();
    }
}
```

WARNING | Injecting parameters directly into query string is not recommended, since it might enable SQL injection possibility.

Assignment 03

Database transactions

- What is a transaction?
  - Unit of work wherein changes can be committed and persisted
- ACID: Atomicity Consistency Isolation Durability
  - Atomicity: all or nothing (if transaction is not working: rollback)
  - Consistency: all data always validates the constraints - also after rollback
  - Isolation: changes in one transaction, not visible for others
  - Durability: once changes are committed they are persisted

JPA Transactions

JPA offers API to control database transactions:

- Through entityManager
- Through using `@Transactional` annotation on classes or methods

Using entityManager to handle transactions

```
public class CustomerService {
    public void updateCustomer(final Customer customer) {
        entityManager.getTransaction().begin();
        entityManager.persist(customer); // your persistence operation
        entityManager.getTransaction().commit();
    }
}
```

JPA Mutating queries

JPA also offers use of mutating queries (update/remove):

```
public class CustomerService {
    void updateCustomerName(final Long id, final String newName) {
        entityManager.getTransaction().begin();
        final var result = entityManager.createQuery("UPDATE Customer " +
                                "SET name = :name" +
                                "WHERE id = :id")
                .setParameter("name", newName)
                .setParameter("id", id)
                .executeUpdate();
        entityManager.getTransaction().commit();
    }

    void deleteCustomerById(final Long id) {
        entityManager.getTransaction().begin();
        entityManager.createQuery("DELETE FROM Customer where id = :id")
                .setParameter("id", id)
                .executeUpdate();
        entityManager.getTransaction().commit();
    }
}
```

JPA Mutating queries Demo

Assignment 04

Assignment 05

JPA Extra annotations

JPA `@Transient` annotation

Entity fields marked with `@Transient` are going to be ignore by JPA.

```
@Entity
public class Customer {
    @Id
    private Long id;
    private String name;
    @Transient
    private String customField;
}
```

Hibernate Dynamic mapping

Hibernate also offers some extra annotations to dynamically map values onto entity fields.

- `@Formula`
- `@Where`
- `@Filter`

WARNING | These annotations only have effect when used through Hibernate’s session

Hibernate `@Formula`

Properties annotated with `@Formula` are calculated based on provided expressions

- Property value is calculated once
- SQL syntax is used, so it is possible to call database functions, stored procedures, etc.

Hibernate `@Formula`

```
@Entity
public class Beer {
    // ...
    private Long fullPrice;
    private Long tax;
    // Java way
    public Long getFinalPrice() {
        return fullPrice - (fullPrice / 100 * tax);
    }
    // Hibernate way
    @Formula("fullPrice - (fullPrice / 100 * tax)")
    private Long finalPrice;
}
```

Hibernate `@Where`

`@Where` annotation on an entity will be added to any query or subquery for this entity.

```
@Where(clause = "deleted = false")
public class Customer {
    // ...
    private boolean deleted;

    // can also be used on collections
    @OneToMany
    @JoinColumn(name = "address_id")
    @Where(clause = "deleted = false")
    private List<Address> addresses = new ArrayList<>();
}
```

Hibernate `@Filter`

- `@Filter` allows to define specific parameterized filters for a given entity that can be enabled by demand.

- `@FilterDef` defines filter: name and parameter set

Hibernate `@Filter`

```
@FilterDef(
        name = "priceFilter",
        parameters = @ParamDef(name = "priceLimit", type = "Long")
)
@Filter(
        name = "priceFilter",
        condition = "price <= :priceLimit"
)
public class Beer { }

public class BeerService {
    public List<Beer> findAllCheapBeers() {
        // filters are configured on a Hibernate session
        final Session session = entityManager.unwrap(org.hibernate.Session.class);
        // enable and configure filter
        session.enableFilter("priceFilter").setParameter("priceLimit", 150);
        // do a query in a normal way
        return session.createQuery("SELECT b FROM Beer b").getResultList();
    }
}
```

Extra annotations demo

- `@Transient`
- `@Formula`

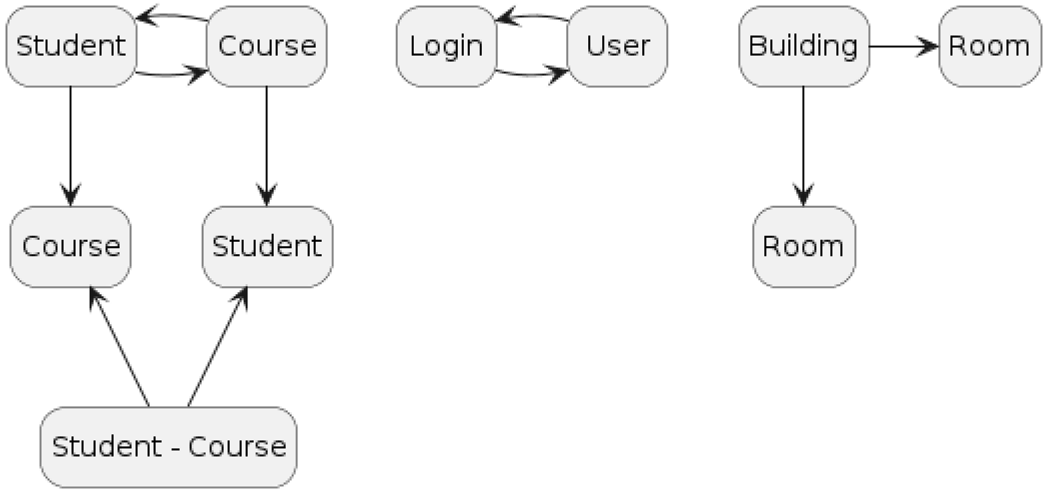
- `@Where`

JPA Advanced

JPA Associations

- One-to-One
- One-to-Many
- Many-to-Many

Associations



One to Many

Table 2. Buildings

id	name
1	White House
2	Palace of Westminster

Table 3. Rooms

id	name	building_id
1	Oval Office	1
2	House of Lords Chamber	2

Many to Many

Table 4. Student

id	name
1	Jon
2	Sam

```
Select id from students where name = 'Paul'
```

Table 5. students_courses

id	student_id	course_id

1	1	1
2	1	2

```
Select course_id from students_courses where student_id = 1
```

Table 6. Course

id	name
1	Ancient History
2	Advanced Physics

```
Select * from courses Where id in (…)
```

JPA Associations: `@OneToOne`

- `@OneToOne` Annotation can be used on one or on both sides of relationship
- One side of relationship must be owning:
    - Owning side is the side that has the data that refers to the other side of relationship
    - Owning side is configured with `@JoinColumn` annotation

JPA Associations: `@OneToOne`

```
@Entity
public class User {
    // ...
    @OneToOne
    @JoinColumn(name = "login_id", referenceColumnName = "id") // owning side
    private Login login;
}
@Entity
public class Login {
    // ...
    @OneToOne(mappedBy = "login") // non-owning side
    private User user;
}
```

JPA Associations: `@OneToMany` and `@ManyToOne`

- `@OneToMany` and `@ManyToOne` are used to describe Many-To-One relationships
- One side of relationship must be owning

JPA Associations: `@OneToMany` and `@ManyToOne`

```
@Entity
public class Student {
    // ...
    @ManyToOne
    @JoinColumn(name = "school_id", nullable = false) // owning side
    private School school;
}
@Entity
public class School {
    // ...
    @OneToMany(mappedBy = "school")
    private List<Student> studentList = new ArrayList<>();
}
```

IMPORTANT It is recommended for the `@ManyToOne` side to be owning

JPA Associations: `@ManyToMany`

- `@ManyToMany` can be used on one or on both sides of relationship
- One side of relationship must be owning:
    - Owning side is the side that has the data that refers to the other side of relationship
    - Owning side is configured with `@JoinTable` annotation

JPA Associations: `@ManyToMany`

```
@Entity
public class Student {
    // ...
    @ManyToMany
    private List<Course> courses = new ArrayList<>();
}
@Entity
public class Course {
    // ...
    @ManyToMany
  @JoinTable(name = "course_students",
            joinColumns = @JoinColumn(name = "course_id", referencedColumnName = "id"),
            inverseJoinColumns = @JoinColumn(name = "student_id", referencedColumnName = "id"))
    private List<Student> students = new ArrayList<>();
}
```

Demo

Assignment 06

JPA Eager/Lazy Loading

- Eager Loading is a design pattern in which data initialization occurs on the spot.
- Lazy Loading is a design pattern that we use to defer initialization of an object as long as it’s possible.

Lazy Loading

Lazy loading is the default mode.

- Advantages:
  - Much smaller initial load time than in the other approach
  - Less memory consumption than in the other approach
- Disadvantages:
  - Delayed initialization might impact performance during unwanted moments.
  - In some cases we need to handle lazily initialized objects with special care, or we might end up with an exception.

Lazy Loading

```
@Entity
public class Student {
    // ...
    @ManyToOne
    @JoinColumn(name = "school_id", nullable = false)
    private School school;
}
@Entity
public class School {
    // ...
    @OneToMany(mappedBy = "school", fetch = FetchType.LAZY) // only loaded when field is accessed
    private List<Student> studentList = new ArrayList<>();
}
```

Eager Loading

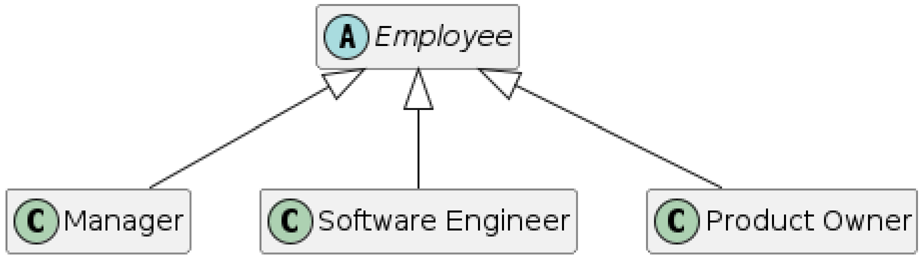
- Advantages:
  - No delayed initialization-related performance impacts
- Disadvantages:
  - Long initial loading time
  - Loading too much unnecessary data might impact performance

Eager Loading

```
@Entity
public class Student {
    // ...
    @ManyToMany(mappedBy = "studentList", fetch = FetchType.EAGER) // always loaded
    private List<School> schoolList = new ArrayList<>();
}
@Entity
public class School {
    // ...
    @ManyToMany(mappedBy = "schoolList")
    @JoinColumn
    private List<Student> studentList = new ArrayList<>();
}
```

Demo

JPA Inheritance



JPA offers several inheritance strategies:

- MappedSuperclass
- Single table (default)
- Joined table
- Table per class

Inheritance Strategies: MappedSuperClass

MappedSuperclass strategy, inheritance is only evident in the entity classes but not the database model.

Table 7. Software Engineers

id	name
1	Jon Snow

Table 8. Managers

id	name
1	Bruce Wayne

Table 9. Product Owners

id	name
1	Scrooge McDuck

Inheritance Strategies: MappedSuperClass

```
@Entity
@MappedSuperClass
public class Employee {
    @Id
    private Long id;
    private String name;
}
@Entity
public class SoftwareEngineer extends Employee {
    // ...
}
@Entity
public class ProductOwner extends Employee {
    // ...
}
```

WARNING Ancestors of entity annotated `@MappedSuperClass` cannot contain associations with other entities.

Inheritance Strategies: Single table

The Single Table strategy creates one table for each class hierarchy.

Table 10. Employees

id	name	role (discriminator column)
1	Jon Snow	Software Engineer
2	Scrooge McDuck	Product Owner
3	Bruce Wayne	Manager

Inheritance Strategies: Single table JPA Configuration

Hibernate differentiates entity type via discriminator column. Available discriminator types are `STRING`, `INTEGER`, `CHAR`.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="role", discriminatorType = DiscriminatorType.STRING)
public class Employee {
    @Id
    private Long id;
    private String name;
}
```

```
@Entity
@DiscriminatorValue("SoftwareEngineer")
public class SoftwareEngineer extends Employee {
    // ...
}
@Entity
@DiscriminatorValue("ProductOwner")
public class ProductOwner extends Employee {
    // ...
}
```

Inheritance Strategies: Joined table

The Joined table strategy maps each class in the hierarchy is to its own table. With Joined Table strategy the only column repeated in all tables is the identifier.

Table 11. Employees

id	name
1	Jon Snow
2	Scrooge McDuck
3	Bruce Wayne

Table 12. Product Owners

id	employee_id	some-other-column
1	2	0987654321

Inheritance Strategies: Joined table JPA Configuration

Joined Table strategy has a disadvantage: retrieving entities requires joins between tables, which affects performance.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Employee {
    @Id
    private Long id;
    private String name;
}
@Entity
public class SoftwareEngineer extends Employee {
    // ...
}
```

Inheritance Strategies: Table per class

The Table per Class strategy maps each entity to its table, which contains all the properties of the entity, including the ones inherited.

Table 13. Software Engineers

id	name
1	Jon Snow

Table 14. Managers

id	name
1	Bruce Wayne

Table 15. Product Owners

id	name
1	Scrooge McDuck

Inheritance Strategies: Table per class JPA Configuration

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Employee {
    @Id
    private Long id;
    private String name;
}
@Entity
public class SoftwareEngineer extends Employee {
    // ...
}
```

WARNING When querying the base class, all subclasses will be returned using `UNION`, which is performance-heavy

Demo

Assignment 07

Named queries

- Named query is a statically defined query
- Improves code organization by separating the query strings from the Java code
- Enforces the use of query parameters which makes the queries more efficient and sometimes safe.

Named queries

```
@Entity
@NamedQuery(name = "Customer.findByName", query = "FROM Customer WHERE name = ?1")
public class Customer { }

public class CustomerService {
    public Customer findCustomerByName(final String name) {
        final Query q = entityManager.createNamedQuery("Customer.findByName");
        q.setParameter(1, name);
        return q.getSingleResult;
    }
}
```

Demo

Assignment 08

JPA Lifecycle Events

JPA allows to define callbacks for lifecycle events

- `@PrePersist`: before persist is called for a new entity

- `@PostPersist`: after persist is called for a new entity
- `@PreRemove`: before an entity is removed
- `@PostRemove`: after an entity has been deleted
- `@PreUpdate`: before the update operation
- `@PostUpdate`: after an entity is updated
- `@PostLoad`: after an entity has been loaded

JPA Lifecycle Events configuration

JPA Lifecycle event callbacks can be defined via:

- Methods in the entity
- EntityListener class

IMPORTANT | Callback methods are required to have a `void` return type.

IMPORTANT | If exception is thrown inside a callback, current transaction will be called back

JPA Lifecycle Events configuration: Entity

```
@Entity
public class Customer {
    // ...
    @PrePersist
    public void prePersist() {
        System.out.println("Saving " + this.name);
    }
}
```

JPA Lifecycle Events configuration: EntityListener

```
@Entity
@EntityListeners(CustomerEntityListener.class)
public class Customer { }

public class CustomerEntityListener {

    @PrePersist
    public void prePersist(final Customer customer) {
        System.out.println("Saving " + customer.name);
    }
}
```

Demo

Assignment 09

Bean Validations

- JSR 380 is a specification of the Java API for bean validation
- Bean Validations allow to validate bean/entity properties via special annotations

```
public class MyUser {
    @NotNull(message = "Id cannot be null")
    private Long id;
    @Size(min = 2, max = 75, message = "Name should be between 2 and 75 characters")
    private String name;
}
```

Bean Validations: Maven dependencies

- Bean validations require additional dependencies:
    - Hibernate validator library
    - Jakarta EL (Expression Language) Library (Required by Hibernate for evaluating expressions inside error message templates)

```
<dependencies>
    <dependency>
        <groupId>org.hibernate.validator</groupId>
        <artifactId>hibernate-validator</artifactId>
        <version>6.2.3.Final</version>
    </dependency>
    <dependency>
        <groupId>org.glassfish</groupId>
        <artifactId>jakarta.el</artifactId>
        <version>3.0.4</version>
    </dependency>
</dependencies>
```

TIP | Make sure versions of Jakarta EL and Hibernate are compatible

Bean Validations: Standard annotations

- `@NotNull` - annotated property value must not be `null`
- `@AssertTrue` - annotated property value must be `true`
- `@Size` - annotated property has a size between the attributes `min` and `max`. Can be applied to `String`, `Collection`, `Map` and array type of properties
- `@Min`, `@Max` - annotated property value must be larger/smaller than the `value` attribute

Bean Validations: Standard annotations

- `@Email` - annotated property must represent a valid email address
- `@NotEmpty` - annotated property must not be `null` or empty. Can be applied to `String`, `Collection`, `Map` and array type of properties.
- `@NotBlank` - annotated property must not be `null` or only contain whitespace. Can be applied to `String` properties

Bean Validations: Standard annotations

- `@Positive`/`@Negative`/`@PositiveOrZero`/`@NegativeOrZero` - annotated property must be strictly positive or negative, or positive/negative including zero
- `@Past`/`@Future`/`@PastOrPresent`/`@FutureOrPresent` - annotated property must be in the past/future or in the past/future including the present. Can be applied to date types.

Bean Validations usage

```
public class MyUser {
    @NotNull(message = "Id cannot be null")
    private Long id;
    @Size(min = 2, max = 75, message = "Name should be between 2 and 75 characters")
    private String name;
    private List<@NotBlank String> notes; // validations can also be applied on elements of a collection
}

public class MyUserService {
    final ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    final Validator validator = factory.getValidator();
    public void validateUser(final MyUser user) {
        final Set<ConstraintViolation<MyUser>> violations = validator.validate(user);
        for (final ConstraintViolation<User> violation : violations) {
            System.out.println("Found violation: " + violation.getMessage());
        }
    }
}
```

Demo

Assignment 10

Spring Data JPA

Spring Data JPA

Spring Data JPA is a Spring-compatible API library providing a number of helpful features for working with the database:

- Simple way of configuring JDBC (datasources)

- Spring Data Repositories (no code DAO/repositories)
- Pagination support
- Auditing support
- Various other features that reduce the amount of code required to work with the database

Spring Data JPA

Spring Data JPA is represented by a single dependency

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

NOTE Do not forget to add a database driver for your database (PostgreSQL/MySQL/etc.)

Spring Data Configuration

- Can be configured programmatically or through properties/yaml file
- Preferred way is through properties/yaml file
- Programmatic way can be used in complex environments, since it gives more control over when to enable the configuration

Spring Data JPA Programmatic configuration

```
@Configuration
```

```
public class DatabaseConfig {
    @Bean
    public DataSource dataSource() {
        final DriverManagerDataSource dataSource = new DriverManagerDataSource();

        dataSource.setDriverClassName("org.postgresql.Driver");
        dataSource.setUsername("postgres");
        dataSource.setPassword("postgres");
        dataSource.setUrl( "jdbc:postgresql://localhost:5432/postgres");

        return dataSource;
    }
}
```

Spring Data JPA Properties configuration

```
spring.datasource:
    driver-class-name: org.postgresql.Driver
    username: postgres
    password: postgres
    url: jdbc:postgresql://localhost:5432/postgres
```

Debugging queries with Spring Data JPA

Spring Data allows to configure several useful options for debugging queries

```
spring.jpa.show-sql: true # dumps running queries to standard out
spring.jpa.properties.hibernate.format_sql: true # prettifies queries
spring.h2.console.enabled: true # If using H2 Database, enable
spring.h2.console.path: /h2 # customize H2 database console path
```

IMPORTANT | H2 Console only works if spring-web dependency is included

Spring Data JPA Debugging Demo

Spring Data JPA Automatic schema generation

Spring Data JPA automatically configures Hibernate to generate schema from entity classes, e.g. hibernate will create database tables for you

```
spring.jpa.hibernate.ddl-auto: create-drop
```

- `none`
- `validate` - validate the schema, makes no changes to the database
- `update` - update the schema
- `create` - creates the schema, destroying previous data
- `create-drop` - drop the schema when the application is stopped*

Configuring schema generation

- For automated schema generation you can add custom constraints:
    - Nullability
    - Uniqueness
    - Indexes

Database constraints: Nullability

- The `nullable` attribute specifies whether the column can have a nullable value

```
@Entity
public class Customer {
    // ...
    @Column(nullable=false)
    private String name;
    // ...
}
```

@NotNull and nullable difference

There’s a big difference between `@NotNull` and `nullable=false`:

- `@NotNull` validates incoming data via JSR 380 (Bean Validation) *only programmatically*
- `nullable=false` ensures nullability is configured at schema level, so inserting null values won’t be allowed by the *database*

Database constraints: Unique

- The `unique` attribute specifies whether the column is a unique key

```
@Entity
public class Customer {
    // ...
    @Column(unique=true)
    private String name;
    // ...
}
```

Database constraints: Table constraints

```
@Entity
@Table(uniqueConstraints = { @UniqueConstraint(name="unique_name_and_status", columnNames = { "name",
"active" }) })
public class Customer {
    // ...
    private String name;
    private boolean active;
    // ...
}
```

NOTE `@Column(unique=true)` is a shortcut for `@UniqueConstraint`

Database Indexes

```
@Entity
@Table(indexes = {
        @Index(name = "customer_name_idx", columnList = "name")
})
public class Customer {
    // ...
    private String name;
    private boolean active;
    // ...
}
```

Demo

Assignment 11

Spring Data Repository

- Data Repositories are offering ready-made CRUD operations for your entities
- Just extend one of the `Repository` interfaces, and it’s done!
- No more `EntityManager` or manually building DAO’s

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {}

@Service
public class CustomerService {
    private CustomerRepository customerRepository;
    public List<Customer> findAllCustomers() {
        final List<Customer> allCustomers = customerRepository.findAll();
        return allCustomers;
    }
}
```

Spring Data Repository types

- `CrudRepository`

  Just CRUD functions: save (create/update), find (findAll/findById), delete (deleteAll/deleteById)

- `PagingAndSortingRepository`

  provides methods to do pagination and sort records (findAll)

- `JpaRepository`

  provides JPA related methods such as flushing the persistence context and delete records in a batch

Using Spring Data Repositories

```
public interface CustomerRepository extends JpaRepository<Customer, Long> { }
public class CustomerService {
    public void exampleMethod() {
        // querying
        final List<Customer> customers = customerRepository.findAll();
        // querying by id
        final Customer customer = customerRepository.getById(1L);
        // saving/updating
        final Customer savedCustomer = customerRepository.save(new Customer(1L, "name"));
        // deleting
        customerRepository.deleteById(1L);
    }
}
```

Using Spring Data Repositories with automatic custom queries

- It is possible to generate queries through defining methods with specific names
- Just use special notation for method names and Spring Data will handle respective code for you

Using Spring Data Repositories with automatic custom queries

```
@Entity
public class Customer {
    @Id
    private Long id;
    private String name;
    private String address;
}

public interface CustomerRepository extends JpaRepository<Customer, Long> {
    Customer findCustomerByName(final String name);

    Customer findCustomerByAddress(final String address);
}
```

Spring Data documentation contains a list of all supported keywords

Using Spring Data Repositories with manual custom queries

- It is also possible to define custom queries via `@Query` annotation

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    // Spring @Query with positional arguments (default)
    @Query("SELECT c FROM Customer c WHERE c.name = ?1")
    Customer findCustomerByName2(final String name);

    // Spring @Query with named arguments
    @Query("SELECT c FROM Customer c WHERE c.name = :name")
    Customer findCustomerByName3(@Param("name") final String name);
}
```

IMPORTANT We're using `@Query` from Spring Data, not from JPA

Demo

Using Spring Data Repositories with Pagination and Sorting

- `@PagingAndSortingRepository` (or `@JpaRepository`, since it extends the same interface) offers possibility to use pagination or sorting:

Using Spring Data Repositories with Pagination and Sorting

```
public class CustomerService {
    public void exampleMethod() {
        //special pagerequest object to define requested page number and amount of items
        final Pageable firstTenElements = PageRequest.of(0, 10);

        // calling library method, note the return type
        final Page<Customer> customersPage = customerRepository.findAll(firstTenElements);
        // to get actual items from the Page object, call getContent()
        final List<Customer> customersList = customersPage.getContent();

        // calling custom method
        final List<Customer> customersByName = customerRepository.findAllByName("Jon Snow",
firstTenElements);
    }
}

public interface CustomerRepository extends JpaRepository<Customer, Long> {
    List<Customer> findAllByName(final String name, final Pageable pageable);
}
```

Demo

Assignment 12

Using Spring Data Repositories with Modifying Queries

- @Modifying annotation allows modifying behavior for the query
- Only works on methods annotated with `@Query`

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    @Modifying
    @Query("UPDATE Customer c SET c.name = ?1 WHERE c.id = ?2")
    int setFirstNameForId(final String name, final Long id);
}
```

NOTE | Note the return type, modifying methods return the number of rows updated

Spring Data REST

- Spring Data REST exposes your repositories directly as REST resources without the need to write controllers by hand
- Is implemented by a separate library `spring-boot-starter-data-rest`

```
@RepositoryRestResource(collectionResourceRel = "bookAuthors", path = "bookAuthors")
```

```
public interface AuthorRepository extends JpaRepository<Author, Long> {
    List<Author> findByName(@Param("name") String name);
}
```

NOTE | Annotation `@RepositoryRestResource` is optional

Demo

Spring Data database initialization using SQL scripts

Spring Boot can automatically create the schema (DDL scripts) of your DataSource and initialize it. It loads SQL from the standard root classpath locations: `schema.sql` and `data.sql`. Database initialization using SQL scripts should be enabled in the configuration:

```
spring.sql.init.mode: always
spring.jpa.defer-datasource-initialization: true # if schema generation is managed by hibernate
```

IMPORTANT | Do not use `schema.sql` with Hibernate initialization enabled

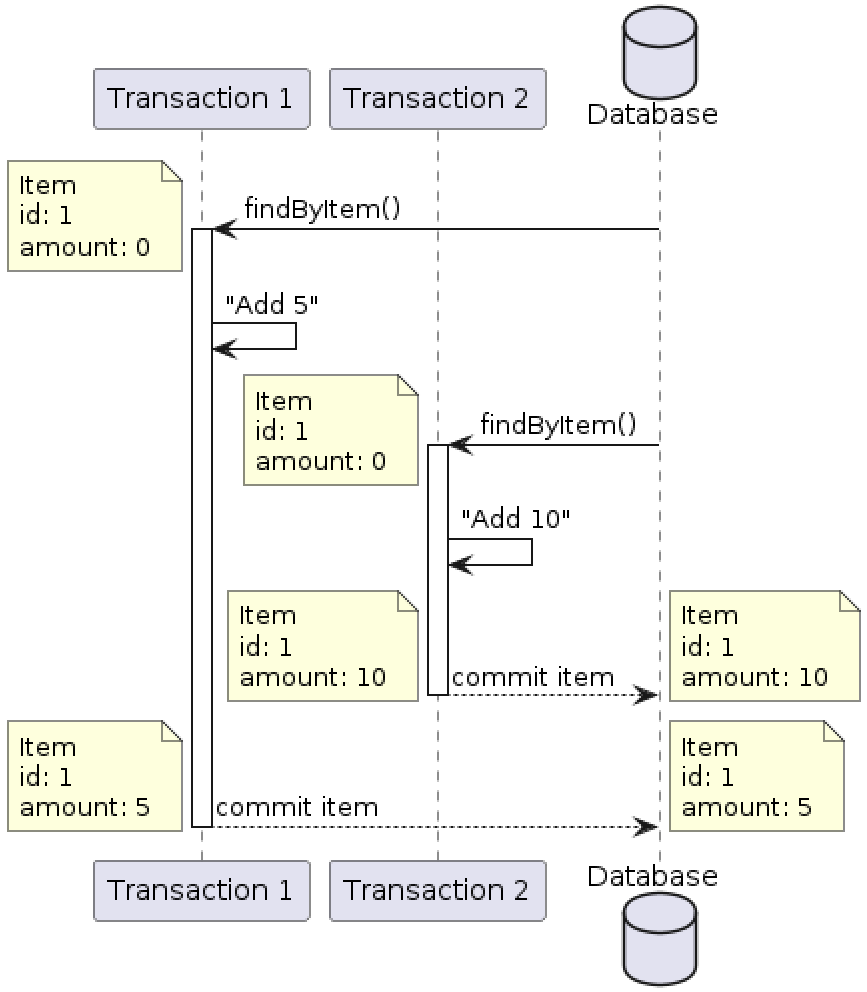
Spring Transactions

- Spring Data offers a few convenient ways to manage transactions
- `@Transactional` can be on a class or method

```
@Service
@Transactional
public class CustomerService {
    //...
}
```

- Useful for performing complex operations, e.g. updating multiple entities in one operation

Database concurrency



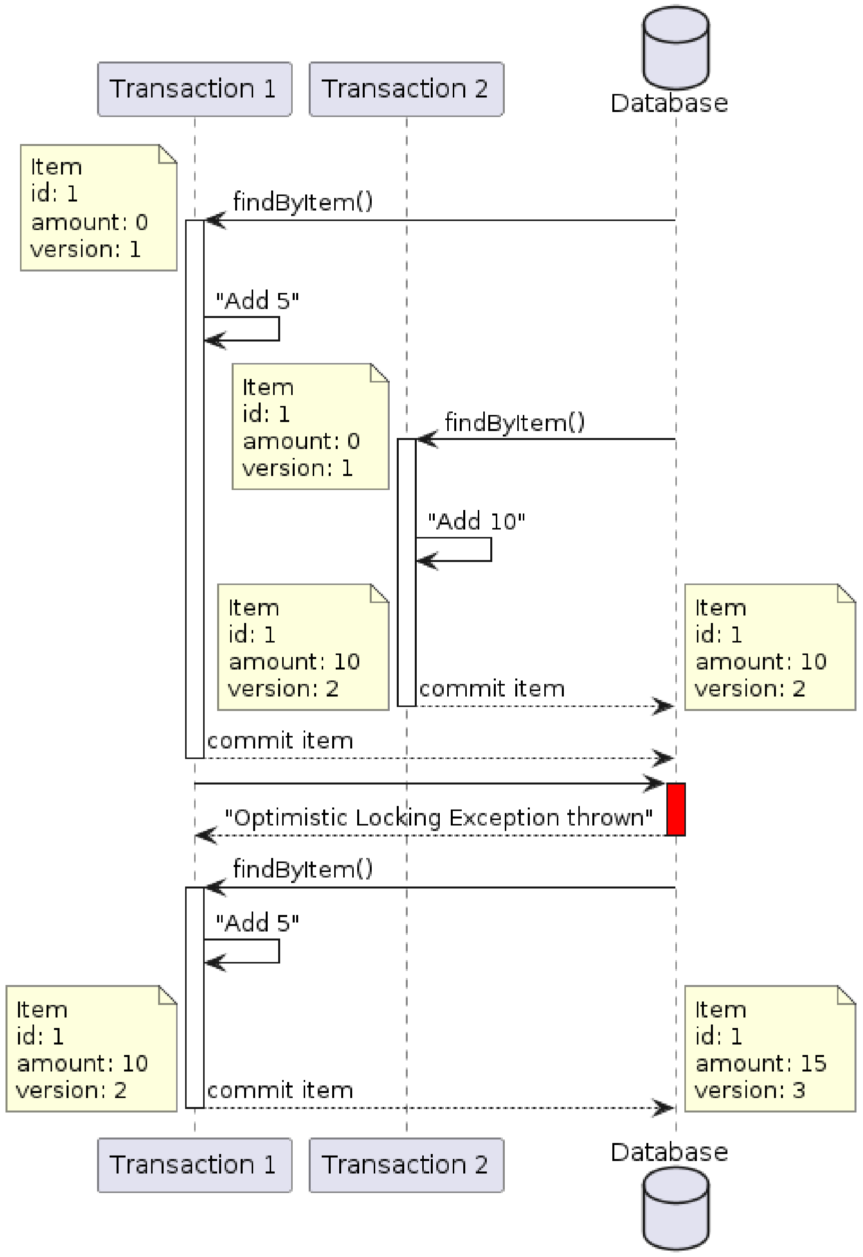
Data loss due to concurrent transactions: the amount should be 15, but instead it’s 5

Solutions to concurrency issues

- Optimistic locking
  - Preferred solution
  - Stores version in the database and compares versions during persisting
  - Gives an OptimisticLockException
- Pessimistic locking
  - Locks immediately
  - When lock cannot be obtained within the specified timeout, gives an exception

Optimistic locking

Commonly uses versioning for comparing entities


Transaction 1
Transaction 2
Database
Item id: 1 amount: 0 version: 1
findByItem()
"Add 5"
Item id: 1 amount: 0 version: 1
findByItem()
"Add 10"
Item id: 1 amount: 10 version: 2
commit item
Item id: 1 amount: 10 version: 2
commit item
"Optimistic Locking Exception thrown"
findByItem()
"Add 5"
Item id: 1 amount: 10 version: 2
commit item
Item id: 1 amount: 15 version: 3
Transaction 1
Transaction 2
Database

Optimistic locking in Spring Data JPA

Entity should have a version field annotated with `@Version`:

```
public class Customer {
    // ...
    @Version
    private Long version;
    // ...
}
```

Optimistic locking in Spring Data JPA

Add `@Lock` annotation on the repository method and specify lock type

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    @Lock(LockModeType.PESSIMISTIC_WRITE)
    Customer save(final Customer customer);
}
```

`@Lock` annotation can be used with a query method or if you derive the query from the method name.

NOTE Optimistic locking is automatically enabled if entity contains fields annotated with `@Version`

Assignment 13

Implement optimistic locking in your app:

- Add version field to one of the entities
- Correctly annotate the repository methods you use to work with this entity
- Experiment with saving and updating your entity:
    - What happens when the versions don’t match?

Spring Data JPA Auditing

Auditing works via JPA Lifecycle events processing.

Event listeners can be defined directly on the entity itself:

```
@Entity
public class Customer {

    @PrePersist
    public void onPrePersist() { }
}
```

Available annotations: `@PrePersist`, `@PreUpdate`, `@PreRemove`

Spring Data JPA Auditing

Events can be also defined in a dedicated event listener class:

```
@Entity
@EntityListeners(AuditListener.class)
public class Customer { }
public class AuditListener {
    @PrePersist
    @PreUpdate
    @PreRemove
    private void beforeAnyOperation(final Object object) { }

}
```

Available annotations: `@PrePersist`, `@PreUpdate`, `@PreRemove`

Spring Data JPA Auditing built-in functions

- Spring Data JPA offers annotations for easier auditing of persistence operations:
    - `@CreatedDate`
    - `@LastModifiedDate`
    - `@CreatedBy`
    - `@LastModifiedBy`
- Annotated fields will be automatically updated by Spring Data on each respective persistence operation

Spring Data JPA Auditing configuration

Implement auditing for one (or more) of your entities:

- Add `@EnableJpaAuditing` on a configuration class (can be an application class)
- Add `@EntityListeners(AuditingEntityListener.class)` on an entity class

- Add one or more date fields with `@CreatedDate` or `@LastModifiedDate` annotations
- Add required annotations on the entity fields

Spring Data JPA Auditing configuration

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class Customer {
    // ...
    @CreatedDate
    private long createdDate;
    // ...
}
```

Demo

Assignment 14

Database migrations with Flyway

- Often your database schema evolves: fields added/removed, new tables/indexes created
- How to manage these changes?
  - Spring integrates with database migration libraries (Flyway, Liquibase)
- Migration library functionality:
  - applying migrations from provided SQL files / programmatically*

    - keeping track of applied migrations via a special table in the same database

Enabling Database migrations with Flyway

- Add flyway dependency to pom file
- Disable automatic schema generation `spring.jpa.hibernate.ddl-auto: none`
- Define your migrations
    - Migrations are stored in `main/resources/db/migration` directory

WARNING | Flyway won't work on non-empty schema

Enabling Database migrations with Flyway: naming syntax

- `<Prefix><Version>__<Description>.sql`
    - Prefix is usually just `V`
    - Version is a version number, eg `1.0`, `2022.05.01`, etc
    - Description is a text, like `adding_time_column`

    Example name: `V1_2__adding_audit_index.sql`
- By default, Spring applies flyway migrations on application startup
- It is possible to apply/verify flyway migrations from command line or via maven plugin

Demo

Assignment 15

Pitfalls when working with JPA

Demo: `@GeneratedValue` using the same sequence

Spring Data: more stuff

Spring Data project offers support for more types of databases:

- Key-value databases
  - Spring Data Redis
- Document databases
  - Spring Data MongoDB
- …

Q&A