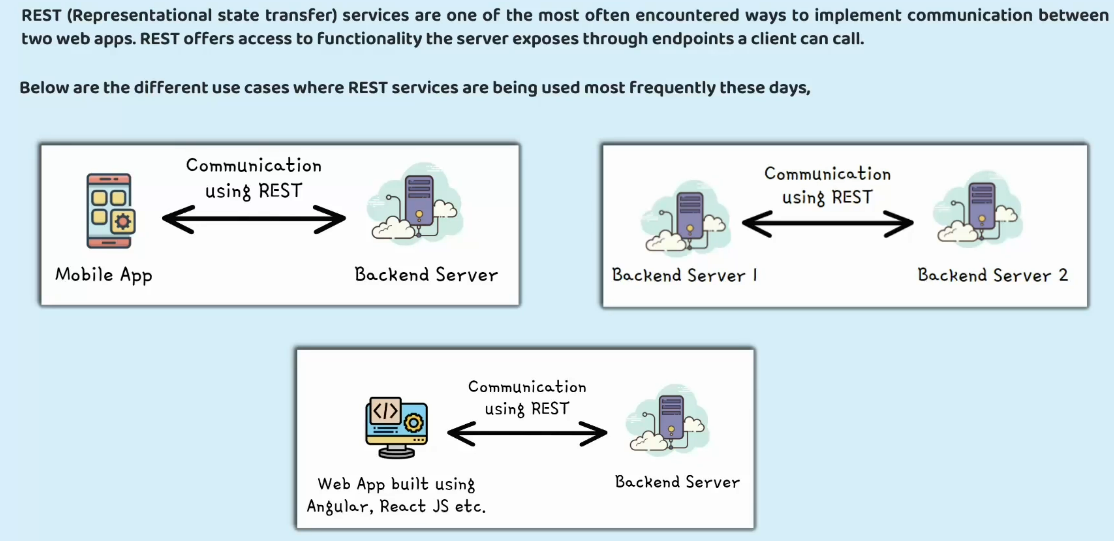
What is microservice?

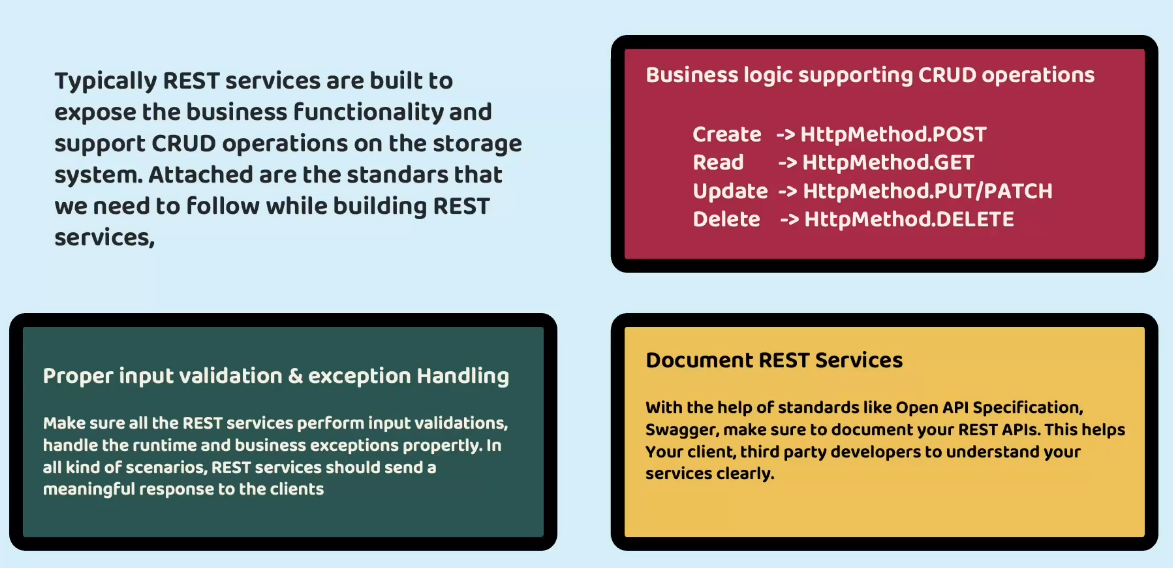
It is a service that exposes its business logic to other API or to external UI application with the help of REST API.

Using REST services, we can establish synchronous communication between multiple API’s or multiple services.

Note: Synchronous communication means when a request comes from an external application ro a microservice, the external application is going to wait for the response so that it can proceed to the next request or function call.



Standard Best practices while Implementing REST Services



PUT: When you are updating a major chunk of data use PUT  
PATCH: When you are updating a very small set of data use PATCH  
Note: This thing is relative

In a Spring application using Hibernate as the JPA (Java Persistence API) provider, the configuration **spring.jpa.hibernate.ddl-auto** specifies how Hibernate should handle database schema generation.   
This property typically accepts different values to control whether Hibernate should create, update, validate, or none (meaning it should not touch the schema) when the application starts up.

Here's what each value typically means:

create: Hibernate creates the database schema when the application starts up, destroying any previous data.

update: Hibernate updates the database schema according to the entity mappings when the application starts up, but it doesn't drop existing tables, nor does it recreate them. It only adds new columns, indexes, etc.

validate: Hibernate validates the existing database schema against the entity mappings when the application starts up, but it doesn't make any changes to the schema.

none: Hibernate does not do anything with the database schema. It assumes that the schema is already created and validated.

The choice of this configuration depends on the development stage and the deployment environment. In development, create or update might be convenient to automatically synchronize the database schema with entity changes. However, in production, it's often safer to use validate or none to prevent accidental modifications to the database schema.

In a Spring Boot application, the **spring.jpa.database-platform** configuration property specifies the database dialect to be used by the Hibernate JPA provider.   
The database dialect determines how Hibernate generates SQL statements specific to the underlying database.

For example,   
if you're using PostgreSQL, the value for spring.jpa.database-platform would typically be org.hibernate.dialect.PostgreSQLDialect.   
Similarly, for MySQL, it would be org.hibernate.dialect.MySQLDialect.

Here's a breakdown:

This class encapsulates the differences between various SQL databases and generates appropriate SQL statements.

By setting this property, you ensure that Hibernate generates SQL statements compatible with your chosen database, which helps in ensuring that your application functions correctly and efficiently with the database system you are using.

In a Spring Boot application, the **spring.datasource.driver-class-name** configuration property specifies the fully qualified name of the JDBC driver class for the database being used. This property tells Spring Boot which Java class to load to establish a connection to the specified database.

For example,

if you are using MySQL, you would set this property to the JDBC driver class provided by MySQL. Similarly, if you are using PostgreSQL, you would set it to the JDBC driver class for PostgreSQL.

This configuration is essential for Spring Boot to establish a connection to the database and execute SQL queries.

In a Spring application, **spring.datasource.url** is a configuration property used to define the URL of the database that your application will connect to.   
This URL typically includes information such as the protocol (like jdbc), the database server's address, port, and the name of the specific database.

For example, a MySQL database URL might look like this:  
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

Here:

jdbc:mysql:// indicates the protocol and driver being used (JDBC for MySQL).

localhost:3306 is the address and port of the MySQL server.

mydatabase is the name of the database.

By setting this property in your Spring application's configuration file, you're instructing Spring to use this URL when establishing a connection to the database.

In a Spring Boot application, **data.sql** and **schema.sql** are special files that Spring Boot uses during the application startup to initialize the database.

Here's what they do:

**schema.sql**: This file typically contains SQL statements to create the database schema, including tables, indexes, constraints, etc. When Spring Boot starts up, it checks for the presence of schema.sql in the src/main/resources directory. If found, it executes the SQL statements in this file to create the necessary database structure.

For example, schema.sql might contain SQL statements like:

CREATE TABLE users (

id BIGINT AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(50) NOT NULL,

password VARCHAR(100) NOT NULL

);

**data.sql**: This file contains SQL statements to insert data into the database tables. Similar to schema.sql, Spring Boot checks for the presence of data.sql in the src/main/resources directory during startup. If found, it executes the SQL statements in this file after the database schema has been created.

For example, data.sql might contain SQL statements like:

INSERT INTO users (username, password) VALUES ('user1', 'password1');

INSERT INTO users (username, password) VALUES ('user2', 'password2');

Using schema.sql and data.sql allows developers to initialize the database with the required schema structure and seed data without needing to write additional code.

This is particularly useful for development and testing purposes where you want to bootstrap your application's database with predefined schema and data.   
However, in production scenarios, you might prefer more sophisticated approaches like database migrations.

**spring-boot-starter-data-jpa** is a starter module provided by Spring Boot to simplify the setup and configuration of Spring Data JPA in your Spring Boot applications.

Spring Data JPA is a part of the larger Spring Data project, which aims to provide a consistent and easy-to-use approach for working with various data access technologies in a Spring-based application.

Here's what spring-boot-starter-data-jpa does:

**Dependency Management**: It manages dependencies required for using Spring Data JPA and related libraries, such as Hibernate (as the default JPA provider), Spring Data Commons, and other necessary dependencies.

**Auto-Configuration**: It automatically configures beans and components required for setting up Spring Data JPA, including entity managers, transaction managers, data source configuration, and more. This greatly simplifies the configuration process, reducing the amount of boilerplate code you need to write.

**Convention over Configuration**: It follows the principle of convention over configuration, meaning that it provides sensible default configurations and behaviour based on commonly used conventions. However, you can still override these defaults and customize the configuration according to your specific requirements.

**Integration with Spring Boot Features**: It seamlessly integrates with other Spring Boot features, such as externalized configuration (through application.properties or application.yml), property binding, and logging configuration.

By including spring-boot-starter-data-jpa as a dependency in your Spring Boot project, you gain access to powerful data access capabilities provided by Spring Data JPA while benefiting from the simplicity and convenience of Spring Boot's auto-configuration and dependency management features. This makes it easier to develop data-driven applications with Spring Boot.

@MappedSuperclass

@Setter @Getter @ToString

public class BaseEntity {

**@Column(updatable = false)**

    private LocalDateTime createdAt;

    @Column(updatable = false)

    private String createdBy;

**@Column(insertable = false)**

    private LocalDateTime updatedAt;

    @Column(insertable = false)

    private String updatedBy;

}

When we update an existing record, we do not want the spring data JPA to consider this property and update or re-populate the value.  
When we insert a new record, we do not want the spring data JPA to consider this property and update or populate the value, keep it null.

import **jakarta.persistence.MappedSuperclass** is a Java annotation that is part of the Jakarta Persistence API. This annotation is used to designate a superclass as a mapped superclass, meaning that it is not mapped to a specific table in the database but can define

Here's what it does:

**Inheritance**: When you annotate a class with @MappedSuperclass, you're essentially saying that this class is a superclass for other entity classes, but it won't be mapped to its own table. Instead, its attributes and mappings will be inherited by its subclasses, which will be mapped to their respective tables.

**Common Attributes and Methods**: You can define common attributes and methods in the superclass, which will be inherited by its subclasses. This helps in avoiding code duplication and promoting code reusability.

**No Table Mapping**: Since @MappedSuperclass is not mapped to a specific table, you cannot query or persist instances of the superclass directly. You can only query or persist instances of its subclasses, which are mapped to their own tables.

Here's an example:

@MappedSuperclass

@Setter @Getter @ToString

public class BaseEntity {

    @Column(updatable = false)

    private LocalDateTime createdAt;

    @Column(updatable = false)

    private LocalDateTime updatedAt;

}

@Entity

@Getter @Setter @ToString

public class Customer extends BaseEntity{

    @Column(name = "customer\_id")

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private int customerId;

    @Column(name = "name")

    private String name;

    @Column(name = "email")

    private String email;

    @Column(name = "mobile\_number")

    private String mobileNumber;

}

In this example, BaseEntity is a mapped superclass annotated with @MappedSuperclass. It defines common attributes like createdAt and updatedAt which will be inherited by its subclasses. However, instances of BaseEntity will not be persisted to a separate table in the database. Instead, its subclasses will be mapped to their own tables, incorporating the attributes defined in BaseEntity.

The @CreatedBy annotation is part of the Spring Data framework in Java, particularly in Spring Data JPA. This annotation is used to automatically populate the entity field representing the user who created the entity.

Here's how it typically works:

You annotate a field in your entity class with @CreatedBy.

When an entity is saved to the database, Spring Data JPA automatically populates this field with the information about who created the entity. This information can be obtained from the security context or any other source configured in your application.

For example, if you have an Author entity and you want to track who created each author, you can annotate a field like createdBy with @CreatedBy:

import org.springframework.data.annotation.CreatedBy;

@Entity

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@CreatedBy

private String createdBy;

// Constructors, getters, setters, etc.

}

In this example, when you save an instance of the Author entity using Spring Data JPA's repository, the createdBy field will be populated automatically with the information of the user who created the author entity.

This annotation is particularly useful in applications where auditing and tracking the history of entity modifications are required. It allows you to easily keep track of who created an entity without explicitly setting this information in your code every time you save an entity.

The **javax.persistence.Entity** annotation is used in Java Persistence API (JPA) to designate a class as an entity.

In Java EE (Enterprise Edition) and Jakarta EE, which are platforms for developing enterprise-level Java applications, JPA is a standard API for managing relational data in Java applications. It provides a framework for mapping Java objects to database tables and vice versa, and also includes features for querying and manipulating data.

When you annotate a Java class with @Entity, you're essentially telling the JPA provider (such as Hibernate, EclipseLink, etc.) that instances of this class should be mapped to corresponding records in the database. Each instance of the annotated class represents a row in the database table.

Here's a basic example:

import javax.persistence.Entity;

@Entity

public class Product {

@Id

private Long id;

private String name;

private double price;

// Getters and setters

}

In this example, the Product class is annotated with @Entity, indicating that instances of Product should be persisted in the database.

The @Id annotation designates the primary key field.

When you use javax.persistence.Entity, you're working with the Java Persistence API. However, as of Jakarta EE 9, the package for JPA has been moved from javax.persistence to jakarta.persistence. So, instead of javax.persistence.Entity, you should now use jakarta.persistence.Entity in newer versions of Jakarta EE for the same purpose.

The **ResponseEntity** class is part of the Spring Framework, specifically in the org.springframework.http package. It represents an HTTP response, including the HTTP status code, headers, and body.

Here's an overview of its main components:

HTTP Status Code: The ResponseEntity allows you to set an HTTP status code for the response. This code indicates the success, failure, or other status of the HTTP request. For example, you can set it to 200 for a successful request or 404 for a resource not found.

Headers: It allows you to include any HTTP headers you want to send back in the response. Headers can contain metadata about the response, such as content type, content length, caching directives, etc.

Body: The ResponseEntity also allows you to set the body of the response. This can be any object representing the data you want to send back to the client. It can be a plain object, a collection, or even an HTML page.

Here's an example of how you might use ResponseEntity in a Spring MVC controller method:

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class MyController {

@GetMapping("/example")

public ResponseEntity<String> exampleEndpoint() {

String responseBody = "Hello, world!";

return ResponseEntity

.status(HttpStatus.OK)

.header("Custom-Header", "SomeValue")

.body(responseBody);

}

}

In this example:

The exampleEndpoint() method returns a ResponseEntity<String>.

🡺 It sets the HTTP status code to 200 (OK) using HttpStatus.OK.

🡺 It adds a custom header named Custom-Header with a value of SomeValue.

🡺 It sets the body of the response to "Hello, world!".

Using ResponseEntity gives you fine-grained control over the HTTP response that your Spring MVC controller method generates. You can set the status code, headers, and body according to your application's requirements.

**@AllArgsConstructor @NoArgsConstructor**

In JPA entities, it's common to have both annotations present, as you often need constructors with and without parameters for different scenarios, such as entity instantiation for data retrieval from the database or object creation for new data insertion.

In JPA (Java Persistence API), **javax.persistence.GenerationType** is an enum used to specify the primary key generation strategy for entities. This enum is typically used in conjunction with the @GeneratedValue annotation to indicate how the primary key for an entity should be generated by the underlying database.

The possible values of javax.persistence.GenerationType are:

AUTO: This is the default generation strategy. The JPA provider (e.g., Hibernate) selects an appropriate strategy based on the underlying database. It may use identity columns, sequences, or other mechanisms depending on the database.

IDENTITY: This strategy relies on an auto-incremented database column to generate primary key values. This is commonly used with databases like MySQL, PostgreSQL, and SQL Server, which support auto-incrementing columns.

SEQUENCE: This strategy uses a database sequence to generate primary key values. Sequences are database objects that generate unique values, and they are often used in databases like Oracle and PostgreSQL.

TABLE: This strategy involves creating a separate table to hold the next available primary key value for each entity. This table is managed by the JPA provider. It's a portable strategy but might not be as efficient as others, as it requires additional database operations.

Here's an example of how you might use javax.persistence.GenerationType with the @GeneratedValue annotation:

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY) // Using IDENTITY strategy

private Long id;

private String name;

private double price;

// Getters and setters

}  
In this example, the @GeneratedValue annotation specifies that the primary key for the Product entity should be generated using the IDENTITY strategy, which typically corresponds to an auto-incrementing column in the database

CREATE TABLE customer(

   customer\_id INT AUTO\_INCREMENT PRIMARY KEY,

)

public class Customer extends BaseEntity{

~~@Column(name = "customer\_id")~~

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int customerId;

}

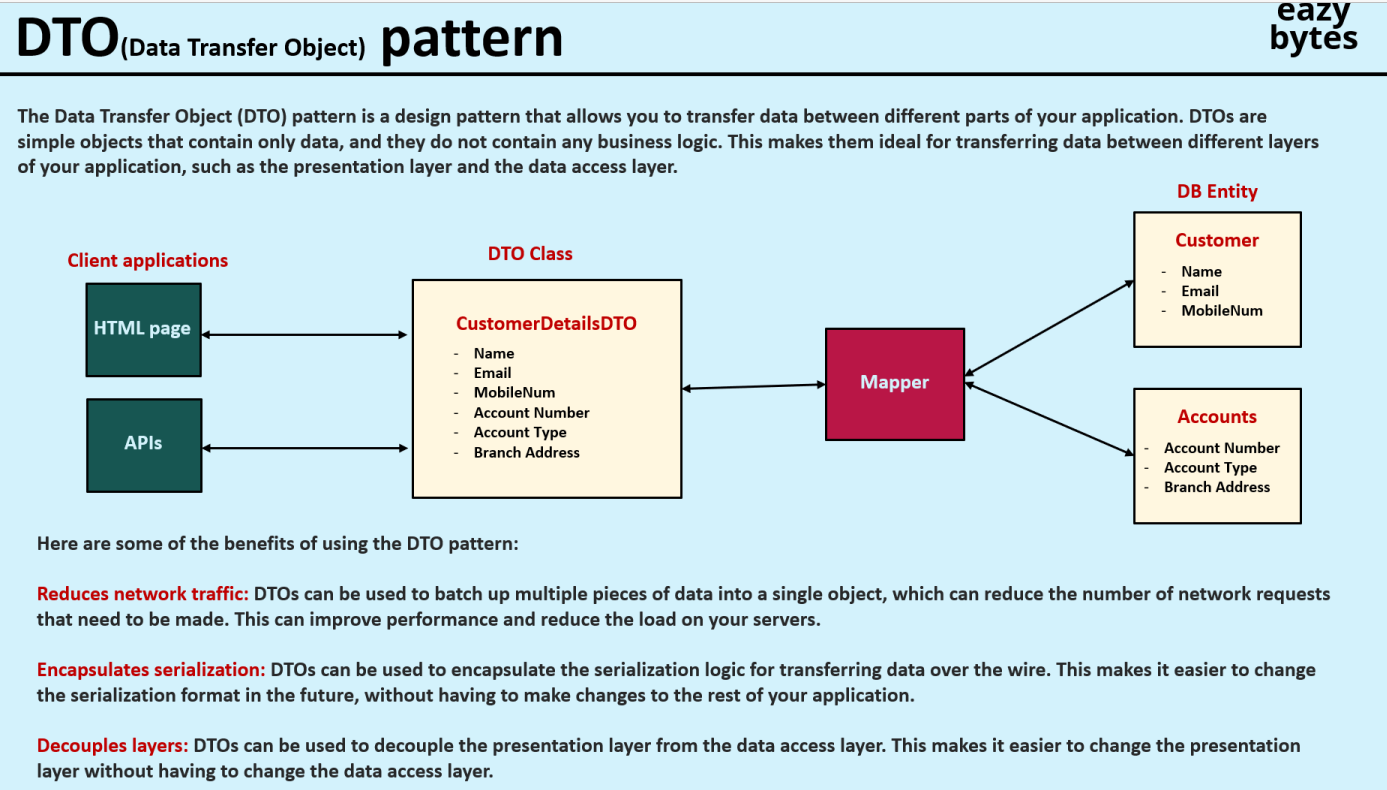
The column name and field name are matching, so we can skip the @Column annotation.   
Good practice is to have one.

public interface CustomerRepository extends **JpaRepository<Customer, Integer>** {

////

}

JpaRepository<Entity class handled by this repository class, Data type of Primary Key>



Never send the entity class object in response as entity. Always map/wrap the entity classes data to DTO classes data and send those in the response.

package com.eazybytes.accounts.constants;

public final class AccountsConstants {

    private AccountsConstants() {

    }

    public static final String  SAVINGS = "Savings";

    public static final String  ADDRESS = "123 Main Street, New York";

    }

Always create private constructor in a constant class, so no one can create an object of that class and constants are not polluted.

In Springboot the controller layer is responsible only to accept the request and do validations. To invoke the business logic, we should always create a service layer.

@Service

public class AccountsServiceImpl implements IAccountsService {

    @Autowired

    private AccountsRepository accountsRepository;

    @Autowired

    private CustomerRepository customerRepository;

….

}

**You must do @Autowired if there is no @AllArgConstructor annotation.**

@Service

@AllArgConstructor

public class AccountsServiceImpl implements IAccountsService {

    private AccountsRepository accountsRepository;

    private CustomerRepository customerRepository;

….

}

**You not do@Autowired if there is @AllArgConstructor annotation.**

@Entity

@Getter @Setter @ToString @AllArgsConstructor @NoArgsConstructor

public class Customer extends BaseEntity{

    @Column(name = "customer\_id")

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int customerId;

    @Column(name = "name")

    private String name;

    @Column(name = "email")

    private String email;

    @Column(name = "mobile\_number")

    private String mobileNumber;

}

@Repository

public interface CustomerRepository extends JpaRepository<Customer, Integer> {

}  
We have defined the primary key as customer\_id

Based on the above entity class and repository class , the repository class would have only findXXX methods based on the primary key value which is customer\_id.   
So, it won’t have find method like findByMobileNumber or findByEmail

So, we will have to write as different method in CustomerRepository like   
@Repository

public interface CustomerRepository extends JpaRepository<Customer, Integer> {

    Optional<Customer> findByMobileNumber(String mobileNumber);

}

Springdata JPA will take care of writing the internal SQL logic to return the result.

findByMobileNumber  
findBy 🡺 SELECT query  
MobileNumber 🡺 Column  
  
If you need multiple columns

findByMobileNumberAndEmail

findBy 🡺 SELECT query  
MobileNumber 🡺 Column

Email 🡺 Column

**@ControllerAdvice**

public class GlobalExceptionHandler {

**@ExceptionHandler(CustomerAlreadyExistsException.class)**

**public ResponseEntity<ErrorResponseDto> handleCustomerAlreadyExistsException(CustomerAlreadyExistsException  
 exception, WebRequest webRequest){**

        ErrorResponseDto errorResponseDTO = new ErrorResponseDto(

                webRequest.getDescription(false), 🡺 false will get api information

                HttpStatus.BAD\_REQUEST,

                exception.getMessage(),

                LocalDateTime.now());

return new ResponseEntity<>(errorResponseDTO, HttpStatus.BAD\_REQUEST);

}

}

By using this annotation, you are telling the framework whenever an exception happens in any of my controller, please invoke the method that I am going to define inside this class.

Certainly! Let's walk through an example to illustrate the usage of @jakarta.transaction.Transactional annotation in a Jakarta EE application.

Suppose we have a simple application with a service class (UserService) responsible for managing user data in a database. We want to ensure that all methods in this service are executed within a transactional context.

Here's how we can use the @jakarta.transaction.Transactional annotation:

import jakarta.transaction.Transactional;

import javax.inject.Inject;

import javax.persistence.EntityManager;

public class UserService {

@Inject

private EntityManager entityManager; // Injecting EntityManager for database operations

@Transactional

public void addUser(User user) {

entityManager.persist(user); // Persisting user entity

}

@Transactional

public void updateUser(User user) {

entityManager.merge(user); // Updating user entity

}

@Transactional

public void deleteUser(long userId) {

User user = entityManager.find(User.class, userId); // Finding user by ID

if (user != null) {

entityManager.remove(user); // Removing user entity

}

}

// Other methods for user management...

}

In this example:

We have a UserService class with methods for adding, updating, and deleting users.

Each method is annotated with @jakarta.transaction.Transactional, indicating that they should be executed within a transactional context.

Inside each method, database operations are performed using EntityManager, which is injected into the class.

Now, when any method of UserService is invoked, the Jakarta EE container will automatically manage the transactions. If an exception occurs during the execution of any method, the transaction will be rolled back, ensuring data consistency.

Here's how you could use this UserService class in a Jakarta EE application:

import jakarta.enterprise.context.RequestScoped;

import javax.inject.Inject;

import javax.ws.rs.\*;

import javax.ws.rs.core.MediaType;

@Path("/users")

@RequestScoped

public class UserResource {

@Inject

private UserService userService;

@POST

@Consumes(MediaType.APPLICATION\_JSON)

public void addUser(User user) {

userService.addUser(user);

}

@PUT

@Path("/{userId}")

@Consumes(MediaType.APPLICATION\_JSON)

public void updateUser(@PathParam("userId") long userId, User user) {

user.setId(userId); // Ensure the user ID is set

userService.updateUser(user);

}

@DELETE

@Path("/{userId}")

public void deleteUser(@PathParam("userId") long userId) {

userService.deleteUser(userId);

}

}  
  
In this example, the UserResource class acts as a RESTful web service endpoint, and it injects and uses the UserService to perform user management operations. The @Transactional annotation ensures that these operations are transactional, providing data consistency and reliability.

The @Modifying annotation in Spring Data JPA is used to signal that the method annotated with it will modify the database. This annotation is typically used in conjunction with the @Query annotation to execute custom JPQL (Java Persistence Query Language) or native SQL queries that perform INSERT, UPDATE, DELETE, or DDL (Data Definition Language) operations.

Here's an example to illustrate the usage of @Modifying annotation:

Suppose we have an entity class User representing users in a system:

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String username;

private String email;

// Constructors, getters, and setters

}  
In this example:

We define a method updateUsername in the UserRepository interface. This method is intended to update the username of a user with a specified ID.

The @Query annotation is used to specify a JPQL query that performs the update operation. The query updates the username attribute of the User entity where the ID matches the provided userId.

The @Modifying annotation is used alongside @Query to indicate that this method will modify the database. This is necessary because JPQL queries by default are assumed to be read-only unless @Modifying is specified.

Now, when you invoke the updateUsername method on the UserRepository, it will execute the JPQL update query to update the username of the specified user in the database.

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

@Transactional

public void changeUsername(Long userId, String newUsername) {

userRepository.updateUsername(userId, newUsername);

}

}  
  
In this example, UserService is a service class that uses the UserRepository to perform database operations. The changeUsername method in UserService calls the updateUsername method defined in the repository, which in turn executes the JPQL update query to modify the username of the specified user in the database.

The spring-boot-starter-validation is a starter module in Spring Boot that includes support for validation in your application. It's based on the Java Bean Validation API (JSR 380), which provides a framework for declaring and enforcing validation constraints on Java objects.

Here's a breakdown of what this dependency provides:

1. **Bean Validation API**: The core of the spring-boot-starter-validation dependency is the Java Bean Validation API. This API defines a set of standard annotations (e.g., @NotNull, @Size, @Email) that you can use to specify validation constraints on your Java beans.
2. **Constraint Validation**: The dependency includes implementations for various constraint validators. These validators are responsible for checking whether the constraints defined on your Java beans are satisfied or not. For example, the @Email constraint validator checks if a given string is a valid email address.
3. **Integration with Spring Boot**: Spring Boot provides integration with the Bean Validation API, allowing you to easily apply validation constraints to your Spring beans, REST endpoints, and MVC controllers. When you use the @Valid annotation on a method parameter or field in a Spring component, Spring Boot automatically triggers validation of the object against its declared constraints.
4. **Error Handling**: When validation constraints are not satisfied, Spring Boot automatically generates validation errors and returns them to the client. These errors typically include information about which fields failed validation and the error messages associated with each failed constraint.
5. **Custom Validation**: You can create your own custom validation constraints by implementing custom constraint annotations and their corresponding validators. Spring Boot supports the integration of custom validation constraints into the validation process seamlessly.

By including spring-boot-starter-validation as a dependency in your Spring Boot project, you gain access to all these features, making it easy to enforce validation rules and ensure data integrity throughout your application. This is particularly useful in web applications where you need to validate user input from forms or API requests.

The @Validated annotation in Spring Framework is used to apply validation constraints on method arguments in Spring-managed beans. It is particularly useful when you need to validate method parameters in service layer classes, where input data needs to be validated before processing.