SPRING BOOT AND TOOLS NOTES

# What is REST?

* REST (Representational State Transfer): Architectural style for distributed systems.
* REST is an architectural style for designing networked applications.
* It uses a stateless communication protocol, typically HTTP, to enable communication between client and server.

Resources:

* + Everything in REST is a resource, identified by a URI (Uniform Resource Identifier).
  + Example: /users/1 represents the user with ID 1.

Stateless Communication:

* + Each request from the client to the server must contain all the information needed to understand and process the request.
  + The server does not store client state between requests.

HTTP Methods:

* + REST uses standard HTTP methods for operations on resources:
    - **GET**: Fetch resource(s).
    - **POST**: Create a resource.
    - **PUT**: Update a resource.
    - **DELETE**: Remove a resource.
    - **PATCH:** Partially update a resource.

Representation

* + *Resources can be represented in various formats like JSON, XML, or HTML.*
  + REST enforces a standard way of interacting with resources, making APIs predictable and easy to use.

Advantages of REST

* + Simplicity: Uses HTTP, a widely adopted protocol, making it easy to understand and implement.
  + Scalability: Statelessness ensures scalability since the server doesn't need to store client-specific data.
  + Interoperability: REST APIs can communicate with a wide variety of clients (e.g., browsers, mobile apps, IoT devices).
  + Flexibility: REST supports multiple formats like JSON and XML, allowing clients to choose what they prefer.
  + Caching: REST can leverage HTTP caching mechanisms to reduce server load and improve performance.

What Are the Alternatives to REST?

* + SOAP (Simple Object Access Protocol):
    - A protocol for exchanging structured information using XML.
    - Advantages over REST: Strong security (WS-Security), transactional support.
    - Disadvantages: More complex, less lightweight compared to REST.
  + GraphQL:
    - A query language for APIs, allowing clients to fetch exactly the data they need.
    - Advantages over REST: Reduces over-fetching/under-fetching of data, single endpoint.
    - Disadvantages: More complex setup, potential for poorly designed queries impacting performance.
  + gRPC (Google Remote Procedure Call):
    - A high-performance framework for connecting services in a distributed system using HTTP/2.
    - Advantages over REST: Faster communication, supports bi-directional streaming.
    - Disadvantages: Not human-readable, harder to debug.
  + WebSockets:
    - Enables two-way communication between client and server in real time.

Advantages over REST: Real-time updates, suitable for applications like chat and live feeds.

* + - Disadvantages: Not ideal for CRUD-based operations.

Fundamental Principles of REST

* Statelessness:
  + The server does not store any information about the client's state between requests.
  + Each request from the client contains all the information necessary to process it.
  + Example:
    - A request to retrieve a user (GET /users/1) must include the authentication token (if required) and should not rely on prior interactions.
  + Benefits:
    - Scalability: Servers can handle multiple requests without tracking sessions.
    - Simplicity: No need for session management.
* Client-Server Architecture:
  + The client and server are separate entities that communicate over HTTP.
  + The server handles business logic and data storage, while the client focuses on the user interface.
  + Benefits:
    - Loose coupling between client and server allows independent development and scaling.
* Uniform Interface:
  + REST enforces a standard way of interacting with resources, making APIs predictable and easier to use.
  + Uniformity is achieved through:
    - Resource identification via URIs.
    - Standard HTTP methods (GET, POST, PUT, DELETE, PATCH).
    - Self-descriptive messages: Responses should include enough information for the client to process them.
    - HATEOAS (Hypermedia as the Engine of Application State): Links provided in responses allow clients to navigate resources dynamically.
* Cacheability:
  + Responses must define whether they are cacheable or not using HTTP headers like Cache-Control.
  + Cached data reduces server load and speeds up responses.
* Layered System:
  + A REST API can have multiple layers (e.g., authentication, caching, business logic), each operating independently.

Setting Up a Spring Boot Project

* @SpringBootApplication
  + Role: Marks the main class of the Spring Boot application.
  + Importance: This annotation is a combination of three other annotations:
    - @Configuration: Indicates that the class contains bean definitions.
    - @EnableAutoConfiguration: Enables Spring Boot's auto-configuration mechanism.
    - @ComponentScan: Automatically scans for components, configurations, and services in the package.
* Files and Best Practices:
  + Application.java (Main class with @SpringBootApplication):
  + Role: The main entry point for the Spring Boot application, where the main() method runs to start the application.
  + Best Practice: This class should be placed in the root package to allow Spring Boot to scan for components, configurations, and services in sub-packages.

*@SpringBootApplication*

*public class MyApplication {*

*public static void main(String[] args) {*

*SpringApplication.run(MyApplication.class, args);*

*}*

*}*

* @RestController:
  + Role: It is used to mark a class as a RESTful web service controller, and it automatically applies @ResponseBody to all methods inside the controller. This means that the return values of methods will be directly written to the HTTP response body as JSON or XML (depending on the client's request).
  + Importance: This annotation simplifies the process of building REST APIs by eliminating the need to manually add @ResponseBody to each method.
  + Best Practice: Use @RestController when you’re building REST APIs to ensure that responses are automatically serialized into JSON.
* @RequestMapping:
  + Role: Maps HTTP requests to handler methods of REST controllers. It can handle all HTTP methods (GET, POST, PUT, DELETE), but it’s more common to use specialized annotations like @GetMapping or @PostMapping.
  + Importance: Defines the base URL for all methods in the controller.
* Files and Best Practices:
  + UserController.java (Controller file):
  + Role: This file defines the API endpoints (routes) for interacting with resources (e.g., User). Each method handles a different HTTP request (GET, POST, PUT, DELETE).
  + Best Practice: Keep controller classes focused on routing and delegating business logic to services.

@RestController

@RequestMapping("/api/users")

public class UserController {

}

* @GetMapping, @PostMapping, @PutMapping, @DeleteMapping:
  + Role: These are specialized forms of @RequestMapping, used for handling specific HTTP methods:
    - @GetMapping: Handles HTTP GET requests (retrieving data).
    - @PostMapping: Handles HTTP POST requests (creating data).
    - @PutMapping: Handles HTTP PUT requests (updating data).
    - @DeleteMapping: Handles HTTP DELETE requests (removing data).
  + Importance: They make the code more readable and concise by providing specific mappings for each HTTP verb.
  + Best Practice: Use the specific mapping annotations to make the intent of the method clearer and improve readability.
* @RequestBody:
  + Role: Binds the incoming HTTP request body to a method parameter. It's typically used with POST and PUT requests to handle the data sent in the body of the request.
  + Importance: Allows Spring to automatically deserialize JSON or XML from the request body into Java objects.
  + Best Practice: Always use @RequestBody for POST/PUT requests when you need to bind incoming data to a Java object.
* @ResponseBody:
  + Role: Indicates that the return value of a method should be written directly to the HTTP response body (as JSON or XML).
  + Importance: It is implied in @RestController, but if you're using @Controller, you need to explicitly add this annotation to ensure that data is returned in the response body.
* Files and Best Practices:
  + UserController.java (Controller file):
  + Role: This file contains methods to handle HTTP requests (GET, POST, PUT, DELETE).
  + Best Practice: Ensure that the controller methods are clean, concise, and delegate business logic to services.

@GetMapping("/{id}")

public User getUserById(@PathVariable Long id)

{

return userService.getUserById(id);

}

@GetMapping

public List<User> getUsersByRole(@RequestParam(required = false) String role)

{

return userService.getUsersByRole(role);

}

Entity in Spring Boot (JPA)

* An Entity is a class that represents a table in the database. The fields in the entity class represent columns in the table, and the methods (getters and setters) represent the data in those columns.
* Entities are typically used with JPA (Java Persistence API), which is a specification for object-relational mapping (ORM) in Java. Spring Data JPA provides a high-level abstraction over JPA to simplify data access and management.
* @Entity
  + Role: Marks a class as an entity that should be persisted in the database.
  + Importance: Without this annotation, JPA won't recognize the class as a persistent entity.
  + Best Practice: Every entity class must have this annotation, and it should be placed at the class level.
* @Id
  + Role: Denotes the primary key of the entity.
  + Importance: Every entity must have a primary key to uniquely identify each row in the corresponding table.
  + Best Practice: Always use @Id on the field that represents the primary key. This is a required annotation for any entity.
* @GeneratedValue
  + Role: Specifies how the primary key value is generated (e.g., auto-increment, UUID).
  + Importance: This is essential when the primary key is automatically generated by the database, such as for an auto-increment field.
  + Best Practice: Use this annotation with @Id if the database is generating the primary key value.
    - Generation Types:
      * GenerationType.AUTO: The persistence provider will choose the generation strategy (default).
      * GenerationType.IDENTITY: Relies on the database to generate the primary key (e.g., auto-increment).
      * GenerationType.SEQUENCE: Uses a database sequence to generate primary keys (useful for databases like Oracle).
      * GenerationType.TABLE: Uses a special database table to generate keys.
* @Column
  + Role: Defines a column in the database table that corresponds to a field in the entity class.
  + Importance: It allows you to customize column properties such as name, length, nullable, etc.
  + Best Practice: Use this annotation to map fields to specific columns with additional configuration.
* @Table
  + Role: Specifies the name of the table in the database that the entity maps to.
  + Importance: If not provided, JPA will use the class name as the table name by default.
  + Best Practice: Explicitly define the table name if it differs from the entity class name or if you want to apply additional constraints like unique constraints.

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "user\_name")

private String name;

}

* JPA supports different types of relationships between entities. These annotations define how entities are related to each other.
  + @OneToMany: Defines a one-to-many relationship (e.g., one user can have many posts).
  + @ManyToOne: Defines a many-to-one relationship (e.g., many posts can belong to one user).
  + @ManyToMany: Defines a many-to-many relationship (e.g., many users can have many roles).
  + @OneToOne: Defines a one-to-one relationship (e.g., one user can have one address).

#### @OneToMany Example:

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@OneToMany(mappedBy = "user")

private List<Post> posts;

}

#### @ManyToOne Example:

@Entity

public class Post {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

@JoinColumn(name = "user\_id") // Foreign key column

private User user;

}

#### @ManyToMany Example:

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToMany

@JoinTable(

name = "user\_role",

joinColumns = @JoinColumn(name = "user\_id"),

inverseJoinColumns = @JoinColumn(name = "role\_id")

)

private List<Role> roles;

}

#### @OneToOne Example:

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@OneToOne

@JoinColumn(name = "address\_id")

private Address address;

}

* @JoinColumn
  + Role: Defines the foreign key column used in relationships (used in @ManyToOne, @OneToOne).
  + Importance: Specifies the column that holds the foreign key in the table for a relationship.
  + Best Practice: Always use @JoinColumn when creating foreign key relationships to specify the column name.
* @Transient
  + Role: Marks a field that should not be persisted in the database.
  + Importance: Useful for fields that are needed in the entity but don't need to be stored in the database (e.g., calculated fields).
  + Best Practice: Use @Transient for non-persistent fields.

@Transient

private String temporaryField; // This will not be persisted in the database

* @Enumerated
  + **Role:** Specifies how an enum type should be persisted in the database.
  + **Importance:** Enums can be stored either as strings (EnumType.STRING) or integers (EnumType.ORDINAL).
  + **Best Practice:** Use EnumType.STRING to store enums as their string representation, which is more readable.

@Enumerated(EnumType.STRING)

private Role role; // Role is an enum

Java Bean Validation

* Java Bean Validation is a specification for defining and validating constraints on Java objects. In Spring Boot, **Hibernate Validator** is the reference implementation of the Bean Validation specification.
* Spring Boot makes it easy to integrate validation using annotations and supports **automatic validation** through annotations like @Valid or @Validated.
* @NotNull
  + **Role:** Ensures that the field is not null.
  + **Use Case:** You can use @NotNull to make sure that a field has been provided and is not missing.
* @Size
  + **Role:** Ensures that the size (length) of the field is within a specified range. This can be used for String, Collection, Map, and Array.
  + **Use Case:** Use it to limit the length of strings or collections.
* @Min and @Max
  + **Role**: These annotations specify the minimum and maximum values allowed for a numeric field.
  + **Use** **Case**: These are commonly used for validating numeric fields, such as age or quantity.
* @Email
  + **Role**: Validates that the field contains a valid email address.
  + **Use Case**: This is used when you expect a field to be an email address.
* @Pattern
  + **Role:** Validates that the field matches a given regular expression.
  + **Use Case:** This is useful when you need to validate specific formats like phone numbers, zip codes, or custom identifiers.
* @NotBlank
  + **Role:** Ensures that the field is not null and that it contains at least one non-whitespace character.
  + **Use Case:** Use this when you need to ensure that a string field is not just whitespace.
* @NotEmpty
  + **Role:** Ensures that the field is not null and not empty (e.g., for collections, strings, and arrays).
  + **Use Case:** This is often used for collections or arrays where an empty collection would be considered invalid.
* @AssertTrue and @AssertFalse
  + **Role:** Ensures that a boolean field is either true or false.
  + **Use Case:** You can use these for flags that need to be validated as true or false.
* @Valid and @Validated
  + **Role:** These annotations are used to trigger the validation of nested objects or lists.
  + **Use Case:** If an entity contains other entities or lists of entities that need to be validated, you use @Valid or @Validated on the parent entity.

public class User {

@NotNull

private Long id;

@NotNull

private String name;

@Size(min = 2, max = 50) private String name;

@Min(1)

@Max(100)

private Integer quantity;

@Email

private String email;

@Pattern(regexp = "^\\+?[0-9]{10,12}$", message = "Invalid phone number")

private String phoneNumber;

@NotEmpty

private List<String> roles;

@NotBlank

private String name;

@AssertTrue

private Boolean isActive;

@Valid // Triggers validation on OrderItem

private List<OrderItem> items;

}

Custom Validation Annotations

* Steps to Create a Custom Validation Annotation
  + **Define the Annotation**:
    - Create an annotation to define the custom validation logic.
  + **Implement the Validator**:
    - Implement the ConstraintValidator interface to define the validation logic.
  + **Use the Annotation**:
    - Use your custom annotation on fields that need validation.

1) Create the Custom Annotation:

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.ANNOTATION\_TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Constraint(validatedBy = EmailDomainValidator.class) // Validator class

@Documented

public @interface ValidEmailDomain {

String message() default "Invalid email domain. Must be 'example.com'.";

Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};

}

2) Implement the Validator:

public class EmailDomainValidator implements

ConstraintValidator<ValidEmailDomain, String>

{

@Override

public void initialize(ValidEmailDomain constraintAnnotation)

{ }

@Override public boolean isValid(

String email,

ConstraintValidatorContext context

)

{

if (email == null) {

return true; // Null check is done separately with @NotNull

}

return email.endsWith("@example.com");

}

}

3) Use the Custom Annotation:

public class User

{

@ValidEmailDomain

private String email;

}

Validating the Request in Controllers

* To trigger validation in the controller, you need to use the @Valid or @Validated annotations.
  + @Valid: This is used for validating simple cases, and it is typically used for validating single objects.
  + @Validated: This is used when you want to validate groups or multiple levels of validation.

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping

public ResponseEntity<String> createUser(@Valid @RequestBody User user) {

// If validation fails, it will automatically throw

// MethodArgumentNotValidException

return ResponseEntity.ok("User created successfully");

}

}

BindingResult for Custom Error Handling

* You can also capture validation errors using BindingResult and return a customized error response.

*@RestController*

*@RequestMapping("/users")*

*public class UserController {*

*@PostMapping*

*public ResponseEntity<String> createUser(@Valid @RequestBody User user, BindingResult result) {*

*if (result.hasErrors()) {*

*String errorMessage = result.getAllErrors().stream()*

*.map(ObjectError::getDefaultMessage)*

*.collect(Collectors.joining(", "));*

*return ResponseEntity.badRequest().body(errorMessage);*

*}*

*return ResponseEntity.ok("User created successfully");*

*}*

*}*

To centralize error handling for validation failures, you can use @ControllerAdvice to handle validation exceptions globally.

*@ControllerAdvice*

*public class GlobalExceptionHandler {*

*@ExceptionHandler(MethodArgumentNotValidException.class)*

*public ResponseEntity<String> handleValidationExceptions(MethodArgumentNotValidException ex) {*

*String errorMessage = ex.getBindingResult().getAllErrors().stream()*

*.map(ObjectError::getDefaultMessage)*

*.collect(Collectors.joining(", "));*

*return ResponseEntity.badRequest().body("Validation failed: " + errorMessage);*

*}*

*}*

Error Handling and Global Exception Handling

Basic Error Handling in Spring Boot

* Spring Boot automatically handles some errors for you (e.g., 404 when an endpoint is not found, 400 for bad requests). However, for more advanced scenarios, you will need to implement custom error handling to give the client meaningful responses.
* When an exception is thrown, Spring Boot returns an appropriate HTTP status code and an error message by default. For example:
  + @ResponseStatus annotation can be used to specify the HTTP status code.
  + @ExceptionHandler is used for handling specific exceptions in controllers.

Custom Error Responses

* In a REST API, you generally want to return more structured and informative error responses, including an error code, message, and optionally additional details like a stack trace or specific error fields.
* Creating a Custom Error Response
* You can define a class that will represent the structure of your error response.

*public class ErrorResponse {*

*private String message;*

*private int status;*

*private long timestamp;*

*// Constructor*

*public ErrorResponse(String message, int status) {*

*this.message = message;*

*this.status = status;*

*this.timestamp = System.currentTimeMillis();*

*}*

*// Getters and Setters*

*public String getMessage() {*

*return message;*

*}*

*public void setMessage(String message) {*

*this.message = message;*

*}*

*public int getStatus() {*

*return status;*

*}*

*public void setStatus(int status) {*

*this.status = status;*

*}*

*public long getTimestamp() {*

*return timestamp;*

*}*

*public void setTimestamp(long timestamp) {*

*this.timestamp = timestamp;*

*}*

*}*

Global Exception Handling with @ControllerAdvice

* Instead of handling exceptions manually in every controller, Spring Boot allows you to centralize exception handling in one place using @ControllerAdvice. This is a powerful way to handle exceptions globally across all controllers in your application.
* @ControllerAdvice for Global Exception Handling
* @ControllerAdvice is used to define a global exception handler that can catch exceptions thrown by any controller in your application. It can be used for logging, custom responses, and handling multiple exceptions.

*@ControllerAdvice*

*public class GlobalExceptionHandler {*

*// Handle specific exceptions*

*@ExceptionHandler(ResourceNotFoundException.class)*

*public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex) {*

*ErrorResponse error = new ErrorResponse(ex.getMessage(), HttpStatus.NOT\_FOUND.value());*

*return new ResponseEntity<>(error, HttpStatus.NOT\_FOUND);*

*}*

*// Handle general exceptions*

*@ExceptionHandler(Exception.class)*

*public ResponseEntity<ErrorResponse> handleGeneralException(Exception ex) {*

*ErrorResponse error = new ErrorResponse("An unexpected error occurred", HttpStatus.INTERNAL\_SERVER\_ERROR.value());*

*return new ResponseEntity<>(error, HttpStatus.INTERNAL\_SERVER\_ERROR);*

*}*

*// Handle validation exceptions*

*@ExceptionHandler(MethodArgumentNotValidException.class)*

*public ResponseEntity<ErrorResponse> handleValidationException(MethodArgumentNotValidException ex) {*

*StringBuilder errorMessage = new StringBuilder();*

*ex.getBindingResult().getAllErrors().forEach(error -> {*

*errorMessage.append(error.getDefaultMessage()).append(", ");*

*});*

*ErrorResponse error = new ErrorResponse(errorMessage.toString(), HttpStatus.BAD\_REQUEST.value());*

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

*}*

*}*

* @ExceptionHandler(ResourceNotFoundException.class): This method handles a specific exception (ResourceNotFoundException) and returns a custom error response.
* @ExceptionHandler(Exception.class): This method handles any generic exception that may occur and provides a fallback error response.
* @ExceptionHandler(MethodArgumentNotValidException.class): This method handles validation

errors, typically when the input data doesn't meet the constraints defined in the entity or DTO.

* When you use **Spring's Bean Validation** annotations (@NotNull, @Size, etc.) and validation fails, Spring throws a MethodArgumentNotValidException. You can handle this exception in the global exception handler to return a custom error response.
* MethodArgumentNotValidException:
  + This exception is thrown when validation fails for an object annotated with @Valid or @Validated in a controller method.
  + The error message is built from the validation errors in the exception's binding result.

Custom Exception Classes

* In a real-world application, it's common to define custom exceptions for specific error scenarios. These custom exceptions can be handled centrally by @ControllerAdvice.

*public class ResourceNotFoundException extends RuntimeException {*

*public ResourceNotFoundException(String message) {*

*super(message);*

*}*

*}*

A screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generatedA screen shot of a computer program

Description automatically generated

MongoDB in Spring Boot

* MongoDB is a NoSQL database that stores data in a flexible, JSON-like format called BSON (Binary JSON). It's ideal for applications that require:
  + High scalability
  + Schema-less or flexible data models
  + Quick iteration cycles
* Spring Boot provides excellent support for MongoDB via the Spring Data MongoDB module.

Entity Modeling in MongoDB

* In MongoDB, entities are mapped to collections. Spring Data MongoDB uses the @Document annotation to indicate that a class is a MongoDB document.

*@Data*

*@Document(collection = "users") // Maps to the "users" collection*

*public class User {*

*@Id*

*private String id; // MongoDB's default ID field is "\_id"*

*private String name;*

*private String email;*

*}*

MongoDB-Specific Annotations

* @Document
  + Indicates that the class is a MongoDB document.
  + Optional attribute: collection specifies the collection name. If not provided, the class name (lowercased) is used as the collection name.
* @Id
  + Marks the field as the primary key for the document.
  + Maps to the MongoDB \_id field.
* @Field
  + Maps a field in the class to a field in the MongoDB document.
  + Useful if the field name in MongoDB is different from the property name in the Java class.
* @Indexed
  + Creates an index on the field to improve query performance.
* @Transient
  + Indicates that a field is not to be persisted in the database.
* @CreatedDate and @LastModifiedDate
  + Automatically populate the created and last modified timestamps.
* @DBRef
  + Creates a reference to another document.

*@Field("full\_name") // Maps to the "full\_name" field in MongoDB*

*private String name;*

*@Indexed(unique = true) // Unique index*

*private String email;*

*@Transient*

*private int tempData; // This field won't be stored in MongoDB*

*@CreatedDate*

*private Date createdDate;*

*@LastModifiedDate*

*private Date lastModifiedDate;*

*@DBRef*

*private Address address;*

Repository Layer

* Spring Data MongoDB provides the **MongoRepository** interface for CRUD operations.

*import org.springframework.data.mongodb.repository.MongoRepository;*

*public interface UserRepository extends MongoRepository<User, String> {*

*// Custom query methods*

*User findByEmail(String email);*

*}*

Query Methods

* Spring Data MongoDB allows you to define custom queries using:
  + Method Naming Convention: Derived queries based on method names.
  + Custom Query Annotation: Using @Query.

*User findByName(String name);*

*List<User> findByEmailContaining(String domain);*

*@Query("{ 'email' : ?0 }") // MongoDB JSON-style query*

*User findByEmailCustom(String email);*

Validation works the same way as in JPA entities. You can use annotations from the **Java Bean Validation API**.

Pagination in MongoDB

* Spring Data MongoDB supports pagination out of the box using the Pageable interface.
* Repository Setup
  + Your repository should extend MongoRepository or PagingAndSortingRepository to support pagination*.*

*public interface UserRepository extends MongoRepository<User, String> {*

*Page<User> findByNameContaining(String name, Pageable pageable);*

*}*

*@RestController*

*@RequestMapping("/users")*

*public class UserController {*

*@Autowired*

*private UserRepository userRepository;*

*@GetMapping*

*public Page<User> getUsers(@RequestParam(defaultValue = "0") int page,*

*@RequestParam(defaultValue = "10") int size) {*

*Pageable pageable = PageRequest.of(page, size);*

*return userRepository.findAll(pageable);*

*}*

*}*

Sorting in MongoDB

* Sorting allows you to retrieve data in a specific order based on one or more fields.
* Repository Setup
  + Spring Data MongoDB provides a Sort class for specifying sort criteria.

*@RestController*

*@RequestMapping("/users")*

*public class UserController {*

*@Autowired*

*private UserRepository userRepository;*

*@GetMapping("/sorted")*

*public List<User> getUsersSorted(@RequestParam(defaultValue = "name") String sortBy,*

*@RequestParam(defaultValue = "asc") String direction) {*

*Sort sort = Sort.by(direction.equalsIgnoreCase("asc") ? Sort.Direction.ASC : Sort.Direction.DESC, sortBy);*

*return userRepository.findAll(sort);*

*}*

*}*

Filtering in MongoDB

* Filtering allows you to retrieve specific data based on dynamic criteria.

*public interface UserRepository extends MongoRepository<User, String> {*

*List<User> findByEmailContaining(String emailFragment);*

*List<User> findByNameAndEmail(String name, String email);*

*}*

*@GetMapping("/filter")*

*public List<User> filterUsers(@RequestParam String emailFragment) {*

*return userRepository.findByEmailContaining(emailFragment);*

*}*

Custom Queries with @Query

* Spring Data MongoDB allows you to write custom MongoDB queries using the @Query annotation.

*@Query("{ 'email' : ?0 }")*

*User findByEmail(String email);*

*@Query("{ 'name' : ?0, 'email' : ?1 }")*

*List<User> findByNameAndEmail(String name, String email);*

*@Query(value = "{ 'email' : ?0 }", fields = "{ 'name' : 1, 'email' : 1 }")*

*User findByEmailWithProjection(String email);*

*@GetMapping("/paginated-sorted")*

*public Page<User> getUsersPaginatedAndSorted(@RequestParam(defaultValue = "0") int page,*

*@RequestParam(defaultValue = "10") int size,*

*@RequestParam(defaultValue = "name") String sortBy,*

*@RequestParam(defaultValue = "asc") String direction) {*

*Sort sort = Sort.by(direction.equalsIgnoreCase("asc") ? Sort.Direction.ASC : Sort.Direction.DESC, sortBy);*

*Pageable pageable = PageRequest.of(page, size, sort);*

*return userRepository.findAll(pageable);*

*}*

Filtering with Dynamic Criteria

* For more complex filtering scenarios, you can use the MongoTemplate class.

*@Autowired*

*private MongoTemplate mongoTemplate;*

*public List<User> filterUsers(String name, String email) {*

*Query query = new Query();*

*if (name != null) {*

*query.addCriteria(Criteria.where("name").is(name));*

*}*

*if (email != null) {*

*query.addCriteria(Criteria.where("email").is(email));*

*}*

*return mongoTemplate.find(query, User.class);*

*}*

*@GetMapping("/dynamic-filter")*

*public List<User> filterUsers(@RequestParam(required = false) String name,*

*@RequestParam(required = false) String email) {*

*return userService.filterUsers(name, email);*

*}*

Aggregation for Advanced Queries in MongoDB

* MongoDB's aggregation framework is a powerful tool for data processing and analysis. It allows you to perform operations such as filtering, grouping, sorting, transforming, and aggregating data across multiple stages
* Common Aggregation Stages
  + $match: Filters documents based on criteria (like a WHERE clause in SQL).
  + $group: Groups documents by a specified field and performs aggregations (e.g., sum, average).
  + $sort: Sorts the documents in ascending or descending order.
  + $project: Reshapes documents by including, excluding, or transforming fields.
  + $limit: Limits the number of documents in the output.
  + $skip: Skips a specified number of documents.
  + $lookup: Performs a join with another collection.
  + $unwind: Deconstructs an array field into multiple documents.

Aggregation Example: Basic Pipeline

* Scenario: Find the total sales for each product.
* MongoDB Query

*db.sales.aggregate([*

*{ $group: { \_id: "$productId", totalSales: { $sum: "$amount" } } }*

*]);*

Spring Data MongoDB Example

*@Autowired*

*private MongoTemplate mongoTemplate;*

*public List<Document> getTotalSalesByProduct() {*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.group("productId").sum("amount").as("totalSales")*

*);*

*return mongoTemplate.aggregate(aggregation, "sales", Document.class).getMappedResults();*

*}*

*Example Raw Data ->*

*[*

*{ "\_id": 1, "productId": "P001", "amount": 100 },*

*{ "\_id": 2, "productId": "P002", "amount": 200 },*

*{ "\_id": 3, "productId": "P001", "amount": 150 },*

*{ "\_id": 4, "productId": "P003", "amount": 300 },*

*{ "\_id": 5, "productId": "P002", "amount": 100 },*

*{ "\_id": 6, "productId": "P003", "amount": 200 },*

*{ "\_id": 7, "productId": "P001", "amount": 250 }*

*]*

*Example Output ->*

*[*

*Document{{\_id=P001, totalSales=500}},*

*Document{{\_id=P002, totalSales=300}},*

*Document{{\_id=P003, totalSales=500}}*

*]*

Advanced Aggregation with Spring Data MongoDB

* $match Stage
  + Filters documents based on criteria.

*Example Raw Data ->*

*[*

*{ "\_id": 1, "status": "active", "name": "John Doe" },*

*{ "\_id": 2, "status": "inactive", "name": "Jane Smith" },*

*{ "\_id": 3, "status": "active", "name": "Alice Brown" },*

*{ "\_id": 4, "status": "pending", "name": "Bob White" }*

*]*

*Example Output ->*

*Document{{\_id=1, status=active, name=John Doe}}*

*Document{{\_id=3, status=active, name=Alice Brown}}*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.match(Criteria.where("status").is("active"))*

*);*

*List<Document> results = mongoTemplate.aggregate(aggregation, "collection", Document.class).getMappedResults();*

*results.forEach(System.out::println);*

* $group Stage
  + Groups documents by a field and applies aggregation functions.

*Example Raw Data ->*

*[*

*{ "\_id": 1, "category": "Electronics", "name": "Smartphone" },*

*{ "\_id": 2, "category": "Electronics", "name": "Laptop" },*

*{ "\_id": 3, "category": "Clothing", "name": "T-shirt" },*

*{ "\_id": 4, "category": "Clothing", "name": "Jeans" },*

*{ "\_id": 5, "category": "Electronics", "name": "Tablet" },*

*{ "\_id": 6, "category": "Furniture", "name": "Chair" }*

*]*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.group("category").count().as("totalProducts")*

*);*

*List<Document> results = mongoTemplate.aggregate(aggregation, "products", Document.class).getMappedResults();*

*results.forEach(System.out::println);*

*Example Output ->*

*Document{{\_id=Electronics, totalProducts=3}}*

*Document{{\_id=Clothing, totalProducts=2}}*

*Document{{\_id=Furniture, totalProducts=1}}*

* $sort Stage
  + Sorts documents in ascending or descending order.

*Example Raw Data ->*

*[*

*{ "\_id": 1, "category": "Electronics", "name": "Smartphone" },*

*{ "\_id": 2, "category": "Electronics", "name": "Laptop" },*

*{ "\_id": 3, "category": "Clothing", "name": "T-shirt" },*

*{ "\_id": 4, "category": "Clothing", "name": "Jeans" },*

*{ "\_id": 5, "category": "Electronics", "name": "Tablet" },*

*{ "\_id": 6, "category": "Furniture", "name": "Chair" }*

*]*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.group("category").count().as("totalProducts"),*

*Aggregation.sort(Sort.Direction.DESC, "totalProducts")*

*);*

*List<Document> results = mongoTemplate.aggregate(aggregation, "products", Document.class).getMappedResults();*

*results.forEach(System.out::println);*

*Example Output ->*

*Document{{\_id=Electronics, totalProducts=3}}*

*Document{{\_id=Clothing, totalProducts=2}}*

*Document{{\_id=Furniture, totalProducts=1}}*

* Combining Stages
  + You can combine multiple stages to create complex pipelines.

*Example Raw Data ->*

*[*

*{ "\_id": 1, "category": "Electronics", "status": "active", "name": "Smartphone" },*

*{ "\_id": 2, "category": "Electronics", "status": "inactive", "name": "Laptop" },*

*{ "\_id": 3, "category": "Clothing", "status": "active", "name": "T-shirt" },*

*{ "\_id": 4, "category": "Clothing", "status": "active", "name": "Jeans" },*

*{ "\_id": 5, "category": "Electronics", "status": "active", "name": "Tablet" },*

*{ "\_id": 6, "category": "Furniture", "status": "active", "name": "Chair" },*

*{ "\_id": 7, "category": "Furniture", "status": "inactive", "name": "Desk" }*

*]*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.match(Criteria.where("status").is("active")),*

*Aggregation.group("category").count().as("totalProducts"),*

*Aggregation.sort(Sort.Direction.DESC, "totalProducts")*

*);*

*List<Document> results = mongoTemplate.aggregate(aggregation, "products", Document.class).getMappedResults();*

*results.forEach(System.out::println);*

*Example Output ->*

*Document{{\_id=Electronics, totalProducts=3}}*

*Document{{\_id=Clothing, totalProducts=2}}*

*Document{{\_id=Furniture, totalProducts=1}}*

Joining Collections with $lookup

* Scenario: Join orders with customers to get customer details for each order.
* MongoDB Query

*db.orders.aggregate([*

*{*

*$lookup: {*

*from: "customers",*

*localField: "customerId",*

*foreignField: "\_id",*

*as: "customerDetails"*

*}*

*}*

*]);*

*Example Raw Data ->*

*orders Collection:*

*[*

*{ "\_id": 1, "orderId": "O001", "customerId": 101, "totalAmount": 150 },*

*{ "\_id": 2, "orderId": "O002", "customerId": 102, "totalAmount": 200 },*

*{ "\_id": 3, "orderId": "O003", "customerId": 101, "totalAmount": 250 }*

*]*

*customers Collection:*

*[*

*{ "\_id": 101, "name": "John Doe", "email": "john.doe@example.com" },*

*{ "\_id": 102, "name": "Jane Smith", "email": "jane.smith@example.com" }*

*]*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.lookup("customers", "customerId", "\_id", "customerDetails")*

*);*

*List<Document> results = mongoTemplate.aggregate(aggregation, "orders", Document.class).getMappedResults();*

*results.forEach(System.out::println);*

*Example Output ->*

*Document{{\_id=1, orderId=O001, customerId=101, totalAmount=150, customerDetails=[{\_id=101, name=John Doe, email=john.doe@example.com}]}}*

*Document{{\_id=2, orderId=O002, customerId=102, totalAmount=200, customerDetails=[{\_id=102, name=Jane Smith, email=jane.smith@example.com}]}}*

*Document{{\_id=3, orderId=O003, customerId=101, totalAmount=250, customerDetails=[{\_id=101, name=John Doe,* [*email=john.doe@example.com}]}*](mailto:email=john.doe@example.com%7d]%7d)*}*

Working with Arrays Using $unwind

* Scenario: Flatten an array field to create one document per array element.
* MongoDB Query

db.orders.aggregate([

{ $unwind: "$items" }

]);

*Example Raw Data ->*

*[*

*{*

*"\_id": 1,*

*"orderId": "O001",*

*"customerId": 101,*

*"items": [*

*{ "productId": "P001", "quantity": 2, "price": 50 },*

*{ "productId": "P002", "quantity": 1, "price": 100 }*

*]*

*},*

*{*

*"\_id": 2,*

*"orderId": "O002",*

*"customerId": 102,*

*"items": [*

*{ "productId": "P003", "quantity": 1, "price": 200 }*

*]*

*}*

*]*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.unwind("items")*

*);*

*List<Document> results = mongoTemplate.aggregate(aggregation, "orders", Document.class).getMappedResults();*

*results.forEach(System.out::println);*

*Example Output ->*

*Document{{\_id=1, orderId=O001, customerId=101, items={productId=P001, quantity=2, price=50}}}*

*Document{{\_id=1, orderId=O001, customerId=101, items={productId=P002, quantity=1, price=100}}}*

*Document{{\_id=2, orderId=O002, customerId=102, items={productId=P003, quantity=1, price=200}}}*

Projection with $project

* Scenario: Reshape documents to include only specific fields and add computed fields.
* MongoDB Query

db.orders.aggregate([

{

$project: {

\_id: 0,

orderId: 1,

totalAmount: { $multiply: ["$quantity", "$price"] }

}

}

]);

*Example Raw Data ->*

*[*

*{ "\_id": 1, "orderId": "O001", "quantity": 2, "price": 50 },*

*{ "\_id": 2, "orderId": "O002", "quantity": 1, "price": 100 },*

*{ "\_id": 3, "orderId": "O003", "quantity": 3, "price": 30 }*

*]*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.project("orderId")*

*.andExpression("quantity \* price").as("totalAmount")*

*);*

*List<Document> results = mongoTemplate.aggregate(aggregation, "orders", Document.class).getMappedResults();*

*results.forEach(System.out::println);*

*Example Output ->*

*Document{{\_id=1, orderId=O001, totalAmount=100}}*

*Document{{\_id=2, orderId=O002, totalAmount=100}}*

*Document{{\_id=3, orderId=O003, totalAmount=90}}*

Pagination with Aggregation

* You can use $skip and $limit stages for pagination.

*Example Raw Data ->*

*[*

*{ "\_id": 1, "productId": "P001", "totalSales": 500 },*

*{ "\_id": 2, "productId": "P002", "totalSales": 200 },*

*{ "\_id": 3, "productId": "P003", "totalSales": 300 },*

*{ "\_id": 4, "productId": "P004", "totalSales": 700 },*

*{ "\_id": 5, "productId": "P005", "totalSales": 100 },*

*{ "\_id": 6, "productId": "P006", "totalSales": 400 },*

*{ "\_id": 7, "productId": "P007", "totalSales": 600 },*

*{ "\_id": 8, "productId": "P008", "totalSales": 800 },*

*{ "\_id": 9, "productId": "P009", "totalSales": 900 },*

*{ "\_id": 10, "productId": "P010", "totalSales": 350 },*

*{ "\_id": 11, "productId": "P011", "totalSales": 450 },*

*{ "\_id": 12, "productId": "P012", "totalSales": 550 }*

*]*

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.sort(Sort.Direction.DESC, "totalSales"),*

*Aggregation.skip(10),*

*Aggregation.limit(10)*

*);*

*List<Document> results = mongoTemplate.aggregate(aggregation, "sales", Document.class).getMappedResults();*

*results.forEach(System.out::println);*

*Example Output ->*

*Document{{\_id=11, productId=P011, totalSales=450}}*

*Document{{\_id=6, productId=P006, totalSales=400}}*

*Document{{\_id=10, productId=P010, totalSales=350}}*

*Document{{\_id=3, productId=P003, totalSales=300}}*

*Document{{\_id=12, productId=P012, totalSales=550}}*

*Document{{\_id=7, productId=P007, totalSales=600}}*

*Document{{\_id=1, productId=P001, totalSales=500}}*

*Document{{\_id=4, productId=P004, totalSales=700}}*

*Document{{\_id=9, productId=P009, totalSales=900}}*

*Document{{\_id=8, productId=P008, totalSales=800}}*

Aggregation Results Mapping

* Spring Data MongoDB can map aggregation results directly to Java objects.

*Example Raw Data ->*

[

{ "\_id": 1, "productId": "P001", "amount": 100 },

{ "\_id": 2, "productId": "P002", "amount": 200 },

{ "\_id": 3, "productId": "P001", "amount": 150 },

{ "\_id": 4, "productId": "P003", "amount": 50 },

{ "\_id": 5, "productId": "P002", "amount": 300 }

]

*Aggregation aggregation = Aggregation.newAggregation(*

*Aggregation.group("productId").sum("amount").as("totalSales")*

*);*

*List<SalesSummary> results = mongoTemplate.aggregate(aggregation, "sales", SalesSummary.class).getMappedResults();*

*results.forEach(System.out::println);*

*@Data*

*public class SalesSummary {*

*private String productId;*

*private double totalSales;*

*}*

*Example Output ->*

SalesSummary{productId='P001', totalSales=250.0}

SalesSummary{productId='P002', totalSales=500.0}

SalesSummary{productId='P003', totalSales=50.0}

Transactions in MongoDB

* MongoDB supports multi-document ACID transactions for replica sets and sharded clusters starting from version 4.0. Transactions allow you to perform a sequence of operations across multiple documents and collections atomically.

Understanding ACID Properties in MongoDB

* **Atomicity**:
  + All operations in a transaction are treated as a single unit. Either all operations succeed, or none of them are applied.
* **Consistency**:
  + The database moves from one valid state to another, maintaining the integrity of the data.
* **Isolation**:
  + Transactions are isolated from each other. Changes made by a transaction are not visible to other transactions until it is committed.
* **Durability**:
  + Once a transaction is committed, the changes are permanent, even in the event of a system crash.

Transactions in Spring Data MongoDB

* Spring Data MongoDB supports transactions using the @Transactional annotation. To use transactions, MongoDB must be running on a replica set (even for local development, you can set up a single-node replica set).
* Setup for Transactions

Enable Transactions in MongoDB:

* Start MongoDB as a replica set:
  + mongod --replSet rs0
* Initialize the replica set in the MongoDB shell:
  + rs.initiate()
* Enable Transactions in Spring Boot:
  + No special configuration is required. The MongoTransactionManager is automatically configured when the application connects to a replica set.

Using Transactions in Spring Data MongoDB

*@Service*

*public class UserService {*

*@Autowired*

*private UserRepository userRepository;*

*@Autowired*

*private OrderRepository orderRepository;*

*@Transactional*

*public void performTransactionalOperation() {*

*User user = new User("Alice", "alice@example.com");*

*userRepository.save(user);*

*Order order = new Order(user.getId(), "Laptop", 1);*

*orderRepository.save(order);*

*// Simulate an error to trigger rollback*

*if (true) {*

*throw new RuntimeException("Simulated exception");*

*}*

*}*

*}*

* The @Transactional annotation ensures that all operations within the method are treated as a single transaction.
* If an exception is thrown, the transaction is rolled back, and none of the changes are applied

# Spring Data JPA

* **Spring Data JPA** is a part of the larger **Spring Data** family that aims to simplify the development of JPA-based data access layers. It integrates **JPA** (Java Persistence API) with Spring, offering repositories that handle common database operations (CRUD) and making it easier to work with relational databases.
* Key Features of Spring Data JPA:
  + **Automatic Query Generation**: It allows you to define repository methods by simply declaring them, and Spring will generate the corresponding SQL queries.
  + **Pagination and Sorting**: Built-in support for paginated and sorted data.
  + **Custom Queries**: You can define custom queries using the @Query annotation.
  + **Auditing**: Automatic tracking of entity creation and modification times.
  + **Transaction Management**: Built-in support for transaction management using @Transactional.

Repository Layer with Spring Data JPA

* Spring Data JPA provides a powerful abstraction for data access. You can create repositories by extending one of the following interfaces:
  + JpaRepository: Provides CRUD operations and pagination support.
  + CrudRepository: A simpler version, just for basic CRUD operations.
  + PagingAndSortingRepository: Adds support for pagination and sorting.
* JpaRepository provides built-in methods like save(), findById(), delete(), findAll(), and more.
* Custom Queries: You can define your custom queries using method names or @Query.

*import org.springframework.data.jpa.repository.JpaRepository;*

*public interface UserRepository extends JpaRepository<User, Long> {*

*// Custom queries can be added here*

*User findByUsername(String username);*

*}*

Custom Queries with @Query

* Spring Data JPA allows you to define custom queries using the @Query annotation.
  + @Query: Defines a JPQL (Java Persistence Query Language) query.
  + @Param: Maps the method parameters to the query parameters.

*import org.springframework.data.jpa.repository.Query;*

*import org.springframework.data.repository.query.Param;*

*public interface UserRepository extends JpaRepository<User, Long> {*

*@Query("SELECT u FROM User u WHERE u.email = :email")*

*User findByEmail(@Param("email") String email);*

*@Query("SELECT u FROM User u WHERE u.username = :username AND u.password = :password")*

*User findByUsernameAndPassword(@Param("username") String username, @Param("password") String password);*

*}*

Pagination and Sorting

* Spring Data JPA provides built-in support for pagination and sorting using Pageable and Sort.
  + PageRequest.of(page, size): Creates a pageable object that defines the page number and page size.
  + findAll(pageable): Returns a Page object that contains paginated results.

import org.springframework.data.domain.Page;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Pageable;

public Page<User> getUsersPaginated(int page, int size) {

Pageable pageable = PageRequest.of(page, size);

return userRepository.findAll(pageable);

}

* + Sort.by(Sort.Order.asc(sortBy)): Creates a sorting object based on the field name (sortBy).

import org.springframework.data.domain.Sort;

public List<User> getUsersSorted(String sortBy) {

Sort sort = Sort.by(Sort.Order.asc(sortBy));

return userRepository.findAll(sort);

}

Transactions with @Transactional

* Spring Data JPA supports declarative transaction management with the @Transactional annotation. This ensures that all database operations in a method are executed within a transaction.
  + @Transactional: Marks the method as transactional. If an exception occurs, all operations in the method will be rolled back.

import org.springframework.transaction.annotation.Transactional;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

@Transactional

public void createUser(User user) {

userRepository.save(user);

// Other operations can go here

}

}

Built-in Methods in Spring Data JPA

* Spring Data JPA provides a rich set of **built-in methods** for performing CRUD (Create, Read, Update, Delete) operations and advanced querying. These methods are derived from JpaRepository, CrudRepository, and PagingAndSortingRepository interfaces.

|  |  |
| --- | --- |
| Method | Description |
| save(S entity) | Saves a given entity. If the entity already exists, it updates it. |
| saveAll(Iterable<S> entities) | Saves multiple entities in a batch. |
| findById(ID id) | Retrieves an entity by its ID. Returns Optional<T>. |
| existsById(ID id) | Checks if an entity exists by its ID. |
| findAll() | Retrieves all entities from the database.` |
| findAllById(Iterable<ID> ids) | Retrieves all entities by their IDs. |
| count() | Returns the total number of entities in the database. |
| deleteById(ID id) | Deletes the entity with the given ID. |
| delete(T entity) | Deletes a specific entity. |
| deleteAllById(Iterable<? extends ID> ids) | Deletes entities by their IDs. |
| deleteAll(Iterable<? extends T> entities) | Deletes multiple entities in a batch. |
| deleteAll() | Deletes all entities from the database. |
| findAll(Pageable pageable) | Retrieves entities in a paginated format. |
| findAll(Sort sort) | Retrieves all entities sorted by the given criteria. |
| findAll(Specification<T> spec, Pageable pageable) | Retrieves entities based on a specification with pagination. |

* Example: Sorting
  + List<User> users = userRepository.findAll(Sort.by("lastName").ascending());
* Example: Pagination
  + Page<User> page = userRepository.findAll(PageRequest.of(0, 10));

Query Methods Derived from Method Names

* Spring Data JPA allows you to define custom queries by deriving method names based on entity properties.
* Structure of Derived Queries
  + Prefix: findBy, readBy, queryBy, getBy, etc.
  + Conditions: Combine entity properties using logical operators (And, Or, etc.).
  + Sorting: Add OrderBy<PropertyName> to the method name.

|  |  |
| --- | --- |
| findByFirstName(String firstName) | Finds all entities where firstName matches the given value. |
| findByLastNameAndAge(String lastName, int age) | Finds entities matching both lastName and age. |
| findByAgeGreaterThan(int age) | Finds entities where age is greater than the given value. |
| findByLastNameLike(String lastName) | Finds entities where lastName matches the given pattern (e.g., %Smith). |
| findTop3ByOrderByAgeDesc() | Retrieves the top 3 entities ordered by age in descending order. |

Advanced Query Methods

|  |  |
| --- | --- |
| countBy<PropertyName>(...) | Counts entities based on a condition. |
| existsBy<PropertyName>(...) | Checks if an entity exists based on a condition. |
| |  | | --- | | deleteBy<PropertyName>(...) |  |  | | --- | |  | | Deletes entities based on a condition. |
| removeBy<PropertyName>(...) | Similar to deleteBy, removes entities based on a condition. |

* Count:
  + long count = userRepository.countByStatus("ACTIVE");
* Exists:
  + boolean exists = userRepository.existsByEmail("test@example.com");
* Delete:
  + userRepository.deleteByStatus("INACTIVE");

Custom Queries Using @Query

* For more complex queries that cannot be derived from method names, you can use the @Query annotation.
* JPQL Example
  + - @Query("SELECT u FROM User u WHERE u.firstName = ?1 AND u.lastName = ?2")

List<User> findByFirstAndLastName(String firstName, String lastName);

* Native SQL Example
  + - @Query(value = "SELECT \* FROM users WHERE email = :email", nativeQuery = true)

User findByEmailNative(@Param("email") String email);

* Updating Data
  + Use @Modifying for update and delete operations:
    - @Modifying

@Query("UPDATE User u SET u.status = :status WHERE u.id = :id")

int updateUserStatus(@Param("id") Long id, @Param("status") String status);

Auditing

* Spring Data JPA supports auditing to automatically track entities' creation and modification times.
* Enable Auditing:
  + Add @EnableJpaAuditing in your configuration class.

*@Configuration*

*@EnableJpaAuditing*

*public class JpaConfig {*

*}*

* Entity with Auditing:

*import org.springframework.data.annotation.CreatedDate;*

*import org.springframework.data.annotation.LastModifiedDate;*

*import javax.persistence.Entity;*

*import java.util.Date;*

*@Entity*

*public class User {*

*@CreatedDate*

*private Date createdAt;*

*@LastModifiedDate*

*private Date updatedAt;*

*}*

ADVANCE AUDITING

*Auditable Class*

*@Setter*

*@Getter*

*@MappedSuperclass*

*@EntityListeners(AuditingEntityListener.class)*

*public abstract class Auditable {*

*@CreatedDate*

*@Column(nullable = false, updatable = false)*

*private LocalDateTime createdAt;*

*@LastModifiedDate*

*private LocalDateTime updatedAt;*

*@CreatedBy*

*@Column(updatable = false)*

*private String createdBy;*

*@LastModifiedBy*

*private String updatedBy;*

*}*

* This class is used as a base class for other entities in your application, which automatically adds auditing fields like created and modified timestamps and the user who created/modified the entity.
* Annotations:
  + @Setter and @Getter:
    - These annotations come from Lombok, and automatically generate the setter and getter methods for the fields in the class.
    - @Setter generates setter methods for all fields.
    - @Getter generates getter methods for all fields.
  + @MappedSuperclass:
    - This annotation marks the class as a superclass that will not be directly mapped to a table in the database, but its fields will be inherited by the subclasses that are annotated with @Entity.
    - This is useful for defining common fields like createdAt, updatedAt, etc., in one place and reusing them across multiple entities.
  + @EntityListeners(AuditingEntityListener.class):
    - This annotation tells Spring JPA to listen for auditing events, like the creation and modification of an entity.
    - The AuditingEntityListener listens for changes to entities and automatically populates auditing fields like createdAt, updatedAt, createdBy, and updatedBy.
* Fields:
  + createdAt:
    - This field is annotated with @CreatedDate, which tells Spring Data JPA to automatically populate it with the timestamp when the entity is first persisted (created).
    - @Column(nullable = false, updatable = false) ensures that the column is not null and cannot be updated after creation.
  + updatedAt:
    - This field is annotated with @LastModifiedDate, which tells Spring Data JPA to automatically populate it with the timestamp whenever the entity is updated.
  + createdBy:
    - This field is annotated with @CreatedBy, which automatically stores the identifier (e.g., username or user ID) of the user who created the entity.
    - @Column(updatable = false) ensures that the createdBy field is not updated after creation.
  + updatedBy:
    - This field is annotated with @LastModifiedBy, which automatically stores the identifier of the user who last modified the entity.

*AuditorAwareImpl Class*

*@Component*

*public class AuditorAwareImpl implements AuditorAware<String> {*

*private static final ThreadLocal<String> currentAuditor = new ThreadLocal<>();*

*@Override*

*public Optional<String> getCurrentAuditor() {*

*return Optional.ofNullable(currentAuditor.get());*

*}*

*public static void setCurrentAuditor(String auditor) {*

*currentAuditor.set(auditor);*

*}*

*public static void clear() {*

*currentAuditor.remove();*

*}*

*}*

* This class implements AuditorAware and provides the current auditor's information (e.g., username or user ID) that will be used to populate the createdBy and updatedBy fields in the Auditable class.
* AuditorAware<String>:
  + AuditorAware is a Spring Data JPA interface that provides a mechanism to retrieve the current auditor (the user who is performing the operation). The getCurrentAuditor() method should return the current auditor's information.
  + In this case, the auditor is a String, which could represent a username or user ID.
* ThreadLocal<String>:
  + ThreadLocal is used to store the current auditor for the current thread. It ensures that each thread has its own independent value for the auditor, which is useful when processing multiple requests concurrently.
  + currentAuditor.get() retrieves the auditor for the current thread.
* getCurrentAuditor():
  + This method retrieves the current auditor from the ThreadLocal storage. It returns an Optional<String>, which can be either the auditor's ID (if set) or an empty Optional if no auditor is set.
* setCurrentAuditor(String auditor):
  + This static method allows you to set the current auditor for the current thread. This method is typically called at the start of a request, usually in a filter or interceptor, where you capture the current logged-in user (e.g., the username) and set it for auditing.
* clear():
  + This method clears the current auditor from the ThreadLocal once the request is processed. It's important to clear the auditor to avoid leaking state between requests.

Using Auditable Class in Entities

* Once you've set up auditing, you can now use the Auditable class as a base class for other entities that require auditing information.

*@Entity*

*public class Product extends Auditable {*

*@Id*

*@GeneratedValue(strategy = GenerationType.IDENTITY)*

*private Long id;*

*private String name;*

*private Double price;*

*// Getters and Setters*

*}*

* The Product entity inherits all the auditing fields (createdAt, updatedAt, createdBy, updatedBy) from the Auditable class.
* You don't need to manually add the auditing fields to Product; they will be automatically populated by Spring Data JPA when the entity is saved or updated.

*Setting the Current Auditor*

*@WebFilter("/api/\*")*

*public class AuditorFilter implements Filter {*

*@Override*

*public void doFilter(HttpServletRequest request, javax.servlet.ServletResponse response, FilterChain chain)*

*throws IOException, ServletException {*

*String username = request.getUserPrincipal().getName(); // Get logged-in user*

*AuditorAwareImpl.setCurrentAuditor(username); // Set current auditor*

*chain.doFilter(request, response); // Continue processing the request*

*}*

*@Override*

*public void init(FilterConfig filterConfig) throws ServletException {}*

*@Override*

*public void destroy() {}*

*}*

* In this example, the AuditorFilter intercepts incoming HTTP requests, retrieves the logged-in user's username (or user ID), and sets it as the current auditor using AuditorAwareImpl.setCurrentAuditor(username).

Query Optimization

* Lazy Loading:
  + With lazy loading, related entities are not loaded immediately. Instead, they are loaded on-demand when they are accessed for the first time.
  + Performance Impact: Lazy loading helps to reduce the number of database queries in some cases, especially when not all relationships are needed. However, it can lead to N+1 query problems when the related entities are accessed in a loop.

*@Entity*

*public class Author {*

*@OneToMany(fetch = FetchType.LAZY)*

*private List<Book> books;*

*}*

* Eager Loading:
  + With eager loading, related entities are fetched immediately along with the main entity, meaning all related data is loaded at once.
  + Performance Impact: While it reduces the number of queries, eager loading can increase memory usage and database load when you retrieve unnecessary data.

*@Entity*

*public class Author {*

*@OneToMany(fetch = FetchType.EAGER)*

*private List<Book> books;*

*}*

N+1 Query Problem:

* **N+1 Query Problem** occurs when you have a parent-child relationship, and for each parent entity, a separate query is issued to fetch the child entities. This can lead to excessive queries and poor performance.

*@Entity*

*public class Author {*

*@OneToMany(fetch = FetchType.LAZY)*

*private List<Book> books;*

*}*

When you load authors, it triggers one query for all authors (N), and then an additional query for each author’s books (1 query per author).

* How to Avoid:
  + Use JOIN FETCH in JPQL or HQL to fetch related entities in one query.

*@Query("SELECT a FROM Author a JOIN FETCH a.books")*

*List<Author> findAllAuthorsWithBooks();*

Alternatively, use @EntityGraph to fetch related entities eagerly without changing the fetch type globally.

*@EntityGraph(attributePaths = "books")*

*List<Author> findAll();*

Criteria API (Dynamic and Type-Safe Queries)

* The Criteria API allows you to create dynamic, type-safe queries in Java without writing JPQL or SQL strings.
* Core Components:
  + CriteriaBuilder: Used to construct query components like conditions, orderings, etc.
  + CriteriaQuery: Represents the query structure.
  + Root: Represents the root entity of the query.

*public List<Author> findAuthorsByBookTitle(String title) {*

*CriteriaBuilder cb = entityManager.getCriteriaBuilder();*

*CriteriaQuery<Author> cq = cb.createQuery(Author.class);*

*Root<Author> authorRoot = cq.from(Author.class);*

*cq.select(authorRoot).where(cb.equal(authorRoot.get("books").get("title"), title));*

*return entityManager.createQuery(cq).getResultList();*

*}*

This is a dynamic query where you can change the conditions programmatically.

Optimistic and Pessimistic Locking

* Optimistic Locking:
  + Optimistic Locking is used when multiple transactions might update the same entity. It assumes that conflicts will be rare and checks for conflicts only when a transaction commits.
  + Implemented using @Version to track the version of an entity.

*@Entity*

*public class Product {*

*@Id*

*private Long id;*

*@Version*

*private Long version;*

*private String name;*

*}*

* How it works: When a transaction updates an entity, the version is checked. If another transaction has already updated the entity, a OptimisticLockException is thrown.
* Pessimistic Locking:
  + Pessimistic Locking locks the entity as soon as it is retrieved from the database to prevent other transactions from modifying it until the lock is released.

Custom Repository Implementations

* Sometimes, you need to implement complex queries or behavior that can’t be easily handled by JpaRepository or CrudRepository.
* Step 1: Create a custom interface:

*public interface CustomAuthorRepository {*

*List<Author> findAuthorsWithBooks();*

*}*

* Step 2: Implement the custom interface:

*@Repository*

*public class CustomAuthorRepositoryImpl implements CustomAuthorRepository {*

*@PersistenceContext*

*private EntityManager entityManager;*

*@Override*

*public List<Author> findAuthorsWithBooks() {*

*String query = "SELECT a FROM Author a JOIN FETCH a.books";*

*return entityManager.createQuery(query, Author.class).getResultList();*

*}*

*}*

* Step 3: Use the custom method in your main repository:

*public interface AuthorRepository extends JpaRepository<Author, Long>, CustomAuthorRepository {*

*}*

JPA Lifecycle Callbacks

* JPA provides lifecycle annotations that allow you to run methods at certain points in the entity lifecycle, such as when an entity is persisted, updated, or deleted.
* Common Lifecycle Annotations:
  + @PrePersist: Before an entity is persisted.
  + @PostPersist: After an entity is persisted.
  + @PreUpdate: Before an entity is updated.
  + @PostUpdate: After an entity is updated.

Multi-Tenant JPA

* Schema-based Multi-Tenancy: Different schemas for different tenants.
* Discriminator-based Multi-Tenancy: A discriminator column in a shared schema to separate tenant data.

JPA Performance Tuning

* Second-Level Cache: Hibernate provides a second-level cache to cache entities between sessions. It reduces database hits for frequently accessed data.
* Configure Cache: You can enable and configure caching in your application.properties.
* Example:
  + spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=true
  + spring.jpa.properties.hibernate.cache.region.factory\_class=org.hibernate.cache.ehcache.EhCacheRegionFactory
* Query Cache: You can also enable query caching to cache the results of a query.
* Example:
  + spring.jpa.properties.hibernate.cache.use\_query\_cache=true

HQL and JPQL Advanced Concepts

* Subqueries:
  + You can use subqueries in JPQL to fetch data based on conditions from another query.
* Example:

*@Query("SELECT a FROM Author a WHERE a.id IN*

*(SELECT b.author.id FROM Book b WHERE b.title = ?1)")*

*List<Author> findAuthorsByBookTitle(String title);*

* Named Queries:
  + You can define reusable queries with @Query annotations or in the orm.xml file.
* Example:

*@Entity*

*@NamedQuery(name = "Author.findByName", query = "SELECT a*

*FROM Author a WHERE a.name = :name")*

*public class Author {}*

* Projection:
  + Instead of returning full entities, you can return DTOs or specific columns.
* Example:

*public interface AuthorProjection {*

*String getName();*

*String getEmail();*

*}*

*public List<AuthorProjection> findAllAuthors();*

Specifications in Spring Data JPA

* Specifications in Spring Data JPA provide a way to build dynamic queries using a type-safe and flexible approach. Specifications are a great alternative to using JPQL or Criteria API when you need to create complex queries with conditions that may change dynamically.
* Specifications are based on the JPA Criteria API but provide a more simplified and reusable approach, especially when working with Spring Data JPA repositories.

What is Specification?

* A **Specification** is a predicate or condition that can be used to query entities in a flexible, reusable way. It allows you to create dynamic queries based on user input, filters, or any other conditions.
* Specifications are often used in combination with JpaRepository to build more complex queries. They are a great alternative to @Query and Criteria API when the query logic can be abstracted into smaller, reusable components.

Why Use Specifications?

* Dynamic Queries: Build queries dynamically based on the parameters.
* Type-safe: Unlike JPQL or native SQL, Specifications are type-safe, meaning you avoid the risk of errors like mismatched field names or types.
* Reusability: Specifications can be reused across multiple queries or parts of your application.
* Composability: You can combine multiple Specifications to create complex queries.

How to Create and Use Specifications in Spring Data JPA

* To use Specifications, you need to:
  + Define a Specification interface.
  + Create the Specification logic.
  + Combine Specifications when necessary.

Step 1: Add the Specification Interface

* Spring Data JPA has a Specification interface that you can implement. The Specification interface has a single method toPredicate, which returns a Predicate that can be used to build the query.

*import org.springframework.data.jpa.domain.Specification;*

*import javax.persistence.criteria.CriteriaBuilder;*

*import javax.persistence.criteria.CriteriaQuery;*

*import javax.persistence.criteria.Predicate;*

*import javax.persistence.criteria.Root;*

*public class AuthorSpecification implements Specification<Author> {*

*private String name;*

*public AuthorSpecification(String name) {*

*this.name = name;*

*}*

*@Override*

*public Predicate toPredicate(Root<Author> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {*

*if (name != null) {*

*return criteriaBuilder.like(root.get("name"), "%" + name + "%");*

*}*

*return criteriaBuilder.conjunction(); // No filter applied*

*}*

*}*

Step 2: Use the Specification in the Repository

* You can use Specifications with Spring Data JPA repositories. JpaRepository provides a method findAll(Specification<T> spec) to use specifications.

*import org.springframework.data.jpa.repository.JpaRepository;*

*import org.springframework.data.jpa.repository.JpaSpecificationExecutor;*

*public interface AuthorRepository extends JpaRepository<Author, Long>, JpaSpecificationExecutor<Author> {*

*// Custom queries can be added here, or use the Specification API for dynamic queries.*

*}*

Step 3: Create the Specification and Use It

* You can now create specifications dynamically in your service or controller layer. Combine multiple specifications using and, or, or not operations.

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.stereotype.Service;*

*import java.util.List;*

*@Service*

*public class AuthorService {*

*@Autowired*

*private AuthorRepository authorRepository;*

*public List<Author> getAuthorsByName(String name) {*

*AuthorSpecification spec = new AuthorSpecification(name);*

*return authorRepository.findAll(spec);*

*}*

*}*

Combining Multiple Specifications

* You can combine multiple specifications using the Specification.where() method, and(), or(), and not() methods.

*public Specification<Author> hasBooks() {*

*return (root, query, builder) -> builder.isNotEmpty(root.get("books"));*

*}*

*public Specification<Author> hasName(String name) {*

*return (root, query, builder) -> builder.like(root.get("name"), "%" + name + "%");*

*}*

*// Combining specifications*

*Specification<Author> spec = Specification.where(hasBooks()).and(hasName("John"));*

*List<Author> authors = authorRepository.findAll(spec);*

Examples of Common Specifications

* Here are some common use cases for Specifications.
* Search by Name:

*public Specification<Author> hasName(String name) {*

*return (root, query, builder) -> builder.like(root.get("name"),*

*"%" + name + "%");*

*}*

* Search by Age:

*public Specification<Author> hasAgeGreaterThan(int age) {*

*return (root, query, builder) -> builder.greaterThan(*

*root.get("age"), age);*

*}*

* Search by Date Range:

*public Specification<Author> createdAfter(Date startDate) {*

*return (root, query, builder) ->*

*builder.greaterThanOrEqualTo(root.get("createdAt"), startDate);*

*}*

*public Specification<Author> createdBefore(Date endDate) {*

*return (root, query, builder) ->*

*builder.lessThanOrEqualTo(root.get("createdAt"), endDate);*

*}*

* Search by Multiple Criteria:

*public Specification<Author> hasNameAndAge(String name, int age) {*

*return Specification.where(hasName(name)).and(hasAgeGreaterThan(age));*

*}*

Using Specifications for Advanced Queries

You can also use join conditions, order by, and more advanced features in your specifications.

* Example of Join Condition:

*public Specification<Author> hasBooksWithTitle(String title) {*

*return (root, query, builder) -> {*

*Join<Author, Book> books = root.join("books", JoinType.INNER);*

*return builder.like(books.get("title"), "%" + title + "%");*

*};*

*}*

* Example of Sorting with Specifications:
  + You can sort the results by adding an Order to your CriteriaQuery.

*public Specification<Author> sortByName() {*

*return (root, query, builder) -> {*

*query.orderBy(builder.asc(root.get("name")));*

*return builder.conjunction();*

*};*

*}*

Batch Operations in Spring Data JPA

* Batch operations are useful when you need to perform large-scale insert, update, or delete operations in your application. Performing batch operations can help optimize performance, reduce the number of database calls, and handle large datasets efficiently. Spring Data JPA supports batch operations with certain configurations.
* Batch operations allow you to process multiple entities in a single database transaction. In JPA, batch operations are usually achieved by using the @Modifying annotation with custom queries or by leveraging @BatchSize to control how many records are processed in a single batch.

How to Enable Batch Operations in Spring Data JPA

* To enable batch processing in Spring Data JPA, you need to configure your persistence provider (e.g., Hibernate) to allow batching.
* Hibernate Configuration for Batch Processing
  + To enable batch operations in Hibernate, configure the following properties in your application.properties or application.yml:

*# Enable batching*

*spring.jpa.properties.hibernate.jdbc.batch\_size=50*

*# Enable the JDBC batching feature in Hibernate*

*spring.jpa.properties.hibernate.order\_inserts=true*

*spring.jpa.properties.hibernate.order\_updates=true*

*# Optional: Disable automatic flush on commit to optimize performance*

*spring.jpa.properties.hibernate.flushMode=COMMIT*

* + hibernate.jdbc.batch\_size: Specifies the number of records to be processed in each batch. In this case, it’s set to 50, meaning 50 records will be inserted, updated, or deleted in a single batch.
  + hibernate.order\_inserts and hibernate.order\_updates: These properties optimize batch processing by reordering insert and update statements to minimize database round-trips.

Batch Operations with @Modifying and @Query

* For batch operations like bulk updates or bulk deletes, you can use the @Modifying annotation with a @Query in your Spring Data JPA repository.
* Example: Batch Update
  + Let's assume you have an entity Product and you want to update the prices of all products that belong to a specific category.

*import org.springframework.data.jpa.repository.Modifying;*

*import org.springframework.data.jpa.repository.Query;*

*import org.springframework.transaction.annotation.Transactional;*

*public interface ProductRepository extends JpaRepository<Product, Long> {*

*@Modifying*

*@Transactional*

*@Query("UPDATE Product p SET p.price = :newPrice WHERE p.category = :category")*

*int updateProductPricesByCategory(@Param("newPrice") double newPrice, @Param("category") String category);*

*}*

* + The @Modifying annotation indicates that this is a modifying query (an update, delete, or insert).
  + The @Transactional annotation ensures that the query is executed within a transaction.
* This query will execute in batches based on the batch size defined in the Hibernate configuration.

*import org.springframework.data.jpa.repository.Modifying;*

*import org.springframework.data.jpa.repository.Query;*

*import org.springframework.transaction.annotation.Transactional;*

*public interface ProductRepository extends JpaRepository<Product, Long> {*

*@Modifying*

*@Transactional*

*@Query("DELETE FROM Product p WHERE p.category = :category")*

*int deleteProductsByCategory(@Param("category") String category);*

*}*

Using @BatchSize for Batch Fetching

* While @Modifying is used for batch updates, @BatchSize is used to control the batch size when fetching related entities. It is useful when you want to load multiple child entities in batches to optimize performance and avoid the N+1 query problem.
* Example: Batch Fetching with @BatchSize
  + Let's assume you have a Customer entity with a one-to-many relationship to Order. You want to fetch orders in batches for multiple customers.

*import org.hibernate.annotations.BatchSize;*

*@Entity*

*public class Customer {*

*@Id*

*private Long id;*

*private String name;*

*@OneToMany(mappedBy = "customer")*

*@BatchSize(size = 20) // Fetch orders in batches of 20*

*private List<Order> orders;*

*// getters and setters*

*}*

In this example, Hibernate will fetch Order entities in batches of 20 for each Customer.

Using @EntityGraph for Optimized Fetching

* Another technique to optimize fetching large datasets in Spring Data JPA is to use @EntityGraph. It allows you to specify which related entities to load eagerly or lazily, providing more control over the fetching behaviour.
* Example: Using @EntityGraph

*@Entity*

*public class Customer {*

*@Id*

*private Long id;*

*private String name;*

*@OneToMany(mappedBy = "customer")*

*private List<Order> orders;*

*// getters and setters*

*}*

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

*@EntityGraph(attributePaths = {"orders"})*

*List<Customer> findAllWithOrders();*

*}*

In this example, when calling findAllWithOrders(), Spring Data JPA will fetch all customers with their associated orders in a single query. This helps avoid multiple database queries and can improve performance for batch operations.

# RestTemplate in Spring

* RestTemplate is a synchronous client provided by Spring Framework for making HTTP requests to RESTful web services. It simplifies communication with HTTP servers and integrates seamlessly with Spring applications.
* Key Features
  + Supports all HTTP methods: GET, POST, PUT, DELETE, PATCH, HEAD, and OPTIONS.
  + Handles serialization and deserialization of request and response bodies automatically.
  + Allows customizations like setting headers, query parameters, and request body.
  + Offers error handling and exchange patterns.
* Basic Usage
  + To use RestTemplate, you can either create a bean or instantiate it directly.
  + RestTemplate Bean Configuration

@Bean

public RestTemplate restTemplate(RestTemplateBuilder builder) {

return builder.build();

}

* Direct Instantiation

RestTemplate restTemplate = new RestTemplate();

Common RestTemplate Methods

|  |  |
| --- | --- |
| Method | Description |
| getForObject | Fetches data from a URL and maps the response to a specified object. |
| getForEntity | Similar to getForObject but returns ResponseEntity for additional metadata (e.g., headers). |
| postForObject | Sends data to a URL and maps the response to a specified object. |
| postForEntity | Similar to postForObject but returns ResponseEntity. |
| put | Updates a resource at a specified URL. |
| delete | Deletes a resource at a specified URL. |
| exchange | Executes a request using any HTTP method and provides full control over headers, body, and type. |
| headForHeaders | Retrieves headers for a resource. |

Examples

* GET Request

*RestTemplate restTemplate = new RestTemplate();*

*String url = "https://jsonplaceholder.typicode.com/posts/1";*

*// Fetch as String*

*String response = restTemplate.getForObject(url, String.class);*

*System.out.println(response);*

*// Fetch as a custom object*

*Post post = restTemplate.getForObject(url, Post.class);*

*System.out.println(post);*

* POST Request

*Post newPost = new Post("Spring RestTemplate", "Understanding REST client");*

*Post createdPost = restTemplate.postForObject("https://jsonplaceholder.typicode.com/posts", newPost, Post.class);*

*System.out.println(createdPost);*

* PUT Request

*Post updatedPost = new Post("Updated Title", "Updated Content");*

*restTemplate.put("https://jsonplaceholder.typicode.com/posts/1", updatedPost);*

* DELETE Request

*restTemplate.delete("https://jsonplaceholder.typicode.com/posts/1");*

* Using exchange for Custom Requests

*HttpHeaders headers = new HttpHeaders();*

*headers.set("Authorization", "Bearer my-token");*

*HttpEntity<String> entity = new HttpEntity<>(headers);*

*ResponseEntity<String> response = restTemplate.exchange(*

*"https://jsonplaceholder.typicode.com/posts",*

*HttpMethod.GET,*

*entity,*

*String.class*

*);*

*System.out.println(response.getBody());*

Error Handling

* By default, RestTemplate throws RestClientException for client and server errors. To handle these gracefully, you can use ResponseErrorHandler.

*public class CustomErrorHandler implements ResponseErrorHandler {*

*@Override*

*public boolean hasError(ClientHttpResponse response) throws IOException {*

*return response.getStatusCode().isError();*

*}*

*@Override*

*public void handleError(ClientHttpResponse response) throws IOException {*

*System.out.println("Error occurred: " + response.getStatusCode());*

*}*

*}*

*// Usage*

*RestTemplate restTemplate = new RestTemplate();*

*restTemplate.setErrorHandler(new CustomErrorHandler());*

Best Practices

* Reuse RestTemplate Bean: Always use a single bean instance to reduce overhead.
* Use Interceptors: For adding common headers (e.g., authentication tokens).
* Error Handling: Implement custom error handling for better control.
* Timeout Configuration: Set timeouts for better control over long-running requests.

SimpleClientHttpRequestFactory factory = new SimpleClientHttpRequestFactory();

factory.setConnectTimeout(5000);

factory.setReadTimeout(5000);

RestTemplate restTemplate = new RestTemplate(factory);

Sending Data with POST Request Using RestTemplate

* To send data with a POST request in RestTemplate, you can use methods like postForObject, postForEntity, or exchange. Here’s how you can send data in various scenarios:
* Using postForObject
  + The postForObject method sends the request body to the specified URL and maps the response to a specified type.

*RestTemplate restTemplate = new RestTemplate();*

*String url = "https://jsonplaceholder.typicode.com/posts";*

*Post post = new Post();*

*post.setTitle("Spring RestTemplate");*

*post.setBody("Sending data with POST request");*

*post.setUserId(1);*

*// Send the POST request and map the response to a Post object*

*Post response = restTemplate.postForObject(url, post, Post.class);*

*System.out.println("Response: " + response);*

* Using postForEntity
  + The postForEntity method sends the request body and returns a ResponseEntity that includes status code, headers, and the response body.

*HttpHeaders headers = new HttpHeaders();*

*headers.setContentType(MediaType.APPLICATION\_JSON);*

*Post post = new Post();*

*post.setTitle("Spring RestTemplate");*

*post.setBody("Sending data with POST request");*

*post.setUserId(1);*

*HttpEntity<Post> request = new HttpEntity<>(post, headers);*

*String url = "https://jsonplaceholder.typicode.com/posts";*

*ResponseEntity<Post> response = restTemplate.postForEntity(url, request, Post.class);*

*System.out.println("Status Code: " + response.getStatusCode());*

*System.out.println("Response Body: " + response.getBody());*

* Using exchange for Advanced Control
  + The exchange method gives full control over the HTTP request, allowing you to specify headers, body, and HTTP method explicitly.

*HttpHeaders headers = new HttpHeaders();*

*headers.setContentType(MediaType.APPLICATION\_JSON);*

*headers.set("Authorization", "Bearer my-token");*

*Post post = new Post();*

*post.setTitle("Spring RestTemplate");*

*post.setBody("Sending data with POST request");*

*post.setUserId(1);*

*HttpEntity<Post> request = new HttpEntity<>(post, headers);*

*String url = "https://jsonplaceholder.typicode.com/posts";*

*ResponseEntity<Post> response = restTemplate.exchange(*

*url,*

*HttpMethod.POST,*

*request,*

*Post.class*

*);*

*System.out.println("Status Code: " + response.getStatusCode());*

*System.out.println("Response Body: " + response.getBody());*

* Sending Form Data
  + If you need to send data as application/x-www-form-urlencoded, use MultiValueMap to construct the form data.

*HttpHeaders headers = new HttpHeaders();*

*headers.setContentType(MediaType.APPLICATION\_FORM\_URLENCODED);*

*MultiValueMap<String, String> map = new LinkedMultiValueMap<>();*

*map.add("username", "user123");*

*map.add("password", "pass123");*

*HttpEntity<MultiValueMap<String, String>> request = new HttpEntity<>(map, headers);*

*String url = "https://example.com/login";*

*ResponseEntity<String> response = restTemplate.postForEntity(url, request, String.class);*

*System.out.println("Status Code: " + response.getStatusCode());*

*System.out.println("Response Body: " + response.getBody());*

* Sending Raw Data
  + For cases where you need to send raw JSON, XML, or text payloads, construct the request body as a String.

*HttpHeaders headers = new HttpHeaders();*

*headers.setContentType(MediaType.APPLICATION\_JSON);*

*String rawJson = """*

*{*

*"title": "Spring RestTemplate",*

*"body": "Sending raw JSON data with POST request",*

*"userId": 1*

*}*

*""";*

*HttpEntity<String> request = new HttpEntity<>(rawJson, headers);*

*String url = "https://jsonplaceholder.typicode.com/posts";*

*ResponseEntity<String> response = restTemplate.postForEntity(url, request, String.class);*

*System.out.println("Status Code: " + response.getStatusCode());*

*System.out.println("Response Body: " + response.getBody());*

# Feign Client

* Feign is a declarative web service client, part of Spring Cloud. With Feign, you can define HTTP requests by simply creating an interface with annotations. Feign internally uses HttpURLConnection or HttpClient and provides a higher-level abstraction.
  + Declarative: Feign uses interfaces with annotations to define HTTP requests.
  + Simplifies Communication: It eliminates the need to write boilerplate code for HTTP communication (e.g., creating and managing HttpRequest objects).
  + Integration with Ribbon: Feign integrates seamlessly with Ribbon for client-side load balancing.
  + Integration with Eureka: Feign integrates with Eureka for service discovery.

Enable Feign in Spring Boot

* To enable Feign in your Spring Boot application, add the @EnableFeignClients annotation in your main application class.

*@SpringBootApplication*

*@EnableFeignClients*

*public class Application {*

*public static void main(String[] args) {*

*SpringApplication.run(Application.class, args);*

*}*

*}*

This annotation scans for interfaces annotated with @FeignClient and generates the required implementation.

Defining a Feign Client

* A Feign Client is an interface where you define the HTTP methods using annotations like @RequestMapping, @GetMapping, @PostMapping, etc.

*@FeignClient(name = "user-service", url = "http://localhost:8081")*

*public interface UserClient {*

*@GetMapping("/users/{id}")*

*User getUserById(@PathVariable("id") Long id);*

*@PostMapping("/users")*

*User createUser(@RequestBody User user);*

*}*

* name: Name of the service to call (can also be used with service discovery like Eureka).
* url: The base URL of the external service (if you are not using service discovery).

Feign Client with Service Discovery

* If you're using Spring Cloud Eureka for service discovery, you don't need to specify the URL. Feign will resolve the service name from the service registry.

*@FeignClient(name = "user-service")*

*public interface UserClient {*

*@GetMapping("/users/{id}")*

*User getUserById(@PathVariable("id") Long id);*

*}*

* Here, user-service is the name of the service registered in Eureka.
* Feign will automatically discover the correct URL for user-service.

Feign Client Error Handling

* Feign provides an easy way to handle errors via @FeignClient annotations.

*@FeignClient(name = "user-service", fallback = UserClientFallback.class)*

*public interface UserClient {*

*@GetMapping("/users/{id}")*

*User getUserById(@PathVariable("id") Long id);*

*}*

*@Component*

*class UserClientFallback implements UserClient {*

*@Override*

*public User getUserById(Long id) {*

*// Fallback logic (e.g., return a default user or error message)*

*return new User("Fallback User", "Fallback email");*

*}*

*}*

* Fallback: The fallback attribute defines a fallback method to be used in case of errors (e.g., service unavailability).
* The UserClientFallback class contains the logic to return a default or fallback response when the service fails.

Feign Client Configuration

* Feign allows you to configure certain aspects of HTTP communication, such as timeouts, logging, and custom interceptors.

*@Configuration*

*public class FeignConfig {*

*@Bean*

*public Request.Options requestOptions() {*

*return new Request.Options(5000, 5000); // Connect timeout and read timeout*

*}*

*}*

* Then, you can specify this configuration in the Feign Client:

*@FeignClient(name = "user-service", configuration = FeignConfig.class)*

*public interface UserClient {*

*// Your methods here*

*}*

Feign Client Logging

* Feign also supports logging, which can be configured to log request/response details for debugging.

*@Configuration*

*public class FeignConfig {*

*@Bean*

*Logger.Level feignLoggerLevel() {*

*return Logger.Level.FULL; // Logs all details*

*}*

*}*

* In your Feign Client, you can specify the configuration to use this logging level:

*@FeignClient(name = "user-service", configuration = FeignConfig.class)*

*public interface UserClient {*

*// Your methods here*

*}*

* To receive data via a GET request, you define a method in your Feign Client interface that uses the @GetMapping annotation. The Feign client will map the response from the API to a Java object.
* To send data via a POST request, you define a method in your Feign Client interface that uses the @PostMapping annotation. You can send data in the request body using @RequestBody.
* Feign Client Interface

*@FeignClient(name = "user-service", url = "http://localhost:8081")*

*public interface UserClient {*

*@GetMapping("/users/{id}")*

*User getUserById(@PathVariable("id") Long id);*

*@PostMapping("/users")*

*User createUser(@RequestBody User user);*

*}*

User Model

*public class User {*

*private Long id;*

*private String name;*

*private String email;*

*}*

UserService

*@Service*

*public class UserService {*

*private final UserClient userClient;*

*@Autowired*

*public UserService(UserClient userClient) {*

*this.userClient = userClient;*

*}*

*public User getUser(Long id) {*

*return userClient.getUserById(id);*

*}*

*public User createUser(User user) {*

*return userClient.createUser(user);*

*}*

*}*

UserController

*@RestController*

*@RequestMapping("/api/users")*

*public class UserController {*

*private final UserService userService;*

*@Autowired*

*public UserController(UserService userService) {*

*this.userService = userService;*

*}*

*@GetMapping("/{id}")*

*public ResponseEntity<User> getUser(@PathVariable Long id) {*

*User user = userService.getUser(id);*

*return ResponseEntity.ok(user);*

*}*

*@PostMapping*

*public ResponseEntity<User> createUser(@RequestBody User user) {*

*User createdUser = userService.createUser(user);*

*return ResponseEntity.status(HttpStatus.CREATED).body(createdUser);*

*}*

*}*

# Multi-Module Project Architecture

* A multi-module project architecture is commonly used in large-scale enterprise applications to organize and manage different aspects of the project. It helps in maintaining a modular, scalable, and maintainable structure. This approach is especially useful when your application is complex, and you want to separate concerns, improve code reuse, and avoid having a single, monolithic codebase.
* In Spring Boot, a multi-module project typically involves creating separate modules (also known as sub-projects) for different functionalities, which are then integrated into a single parent project.
* Key Concepts:
  + Parent Module: The root module of the project that contains shared configurations, dependencies, and the structure of the submodules.
  + Child Modules: The individual modules that provide specific functionality and can be reused independently.
  + Dependencies: The child modules can have dependencies on each other or on external libraries.
  + Build Management: The parent module manages the build lifecycle, such as compilation, packaging, and dependency management, using a build tool like Maven or Gradle.
* Why Use Multi-Module Architecture?
  + Separation of Concerns: Different modules handle different parts of the application (e.g., service layer, persistence layer, API layer).
  + Code Reusability: You can reuse modules in other projects.
  + Better Dependency Management: Dependencies can be shared and versioned across modules.
  + Faster Builds: Smaller modules lead to faster builds as only changed modules need to be rebuilt.
  + Scalability: You can scale each module independently.
* Benefits of Multi-Module Architecture
  + Separation of Concerns: Highlight how separating the application into different modules makes the codebase more maintainable and easier to understand. Each module has a specific responsibility (e.g., persistence, API, services).
  + Code Reusability: Mention how modules can be reused across different projects. For instance, a module for user authentication can be reused in multiple applications.
  + Modularization: Emphasize that modules can evolve independently, meaning one module can be updated without affecting the others.
  + Parallel Development: Different teams can work on different modules in parallel, improving productivity and reducing conflicts.
* How Dependency Management Works
  + Centralized Versioning: In multi-module projects, versioning is managed centrally by the parent POM (pom.xml). Dependencies can be defined at the parent level, reducing version mismatches across child modules.
  + Module Dependencies: Discuss how one module can depend on another using <dependency> in the child module's POM. For example, if your module-2 (API layer) depends on module-1 (core services), you will define that dependency in module-2's POM.
  + Transitive Dependencies: If module-1 has dependencies, they are automatically available in module-2 if module-2 depends on module-1.
* Maven vs Gradle in Multi-Module Projects
  + Maven: Maven is the most commonly used build tool in multi-module Spring Boot projects. In Maven, you can define parent-child relationships using <modules> and manage dependencies centrally using <dependencyManagement>.
  + Gradle: Gradle is also an option and has some advantages in terms of speed and flexibility. Gradle uses a slightly different approach to managing multi-modules but offers powerful features like incremental builds and custom tasks.
* Build Lifecycle and Effective POM
  + Effective POM: The effective POM is a combination of the parent POM and child POMs. It's important to understand how the child modules inherit configuration and dependencies from the parent POM and how the effective POM is constructed during the build process.
  + Build Lifecycle: In a multi-module project, when you run a Maven command (e.g., mvn clean install), it will build all the modules in the correct order based on their dependencies.
* Modularization Best Practices
  + Single Responsibility Principle: Each module should have a single responsibility. For example, don't combine your service layer and persistence layer in one module. Keep the API layer separate from the business logic.
  + Avoid Circular Dependencies: Ensure that there are no circular dependencies between modules. For example, module-1 should not depend on module-2, and module-2 should not depend on module-1. Circular dependencies can lead to compilation errors and logical issues.
  + Layered Architecture: In most cases, a typical multi-module Spring Boot project will follow a layered architecture where modules are organized by their role: persistence, service, API, etc.

* Testing in Multi-Module Projects
  + Unit Tests: Ensure that each module is independently testable. For instance, unit tests for the core service layer should not depend on the persistence layer or API layer.
  + Integration Tests: You should also test the integration between modules. This can be done by creating a test configuration that integrates the required modules and runs the application context.
  + Mocking Dependencies: If one module depends on another, consider mocking the dependencies for unit tests to isolate the module under test.
  + Test Coverage: You can discuss tools like JUnit and Mockito for testing and how you ensure high test coverage across modules.
* Managing Shared Resources
  + Shared Configuration: If multiple modules need the same configuration (e.g., database connection settings, API configurations), centralize this in the parent module or a common configuration module.
  + Shared Utility Modules: If there are utility functions (e.g., string manipulation, date handling, etc.) that multiple modules require, create a separate module for shared utilities.

* Handling Versioning in Multi-Module Projects
  + Versioning of Modules: Explain how versioning is handled in multi-module projects. For example, if module-1 is updated, the parent POM version might change, which would then require module-2 and module-3 to use the updated version.
  + Releases: Discuss the release strategy for modules. Should all modules be released together, or can some modules be versioned and released independently?

* Handling Common Problems
  + Circular Dependency Issue: Circular dependencies occur when modules depend on each other directly or indirectly. This leads to a build failure. Discuss how to avoid or resolve circular dependencies by rethinking the module design or restructuring dependencies.
  + Module Initialization Order: When building modules, Maven ensures that modules are built in the correct order. If you face any dependency order issues, you can solve them by defining the proper dependencies in the parent module's pom.xml.
  + Consistency: With multiple modules, it’s easy for the project to lose consistency. Always ensure that coding standards, configurations, and dependency versions are consistent across modules.
* Performance Considerations
  + Module Granularity: While modularizing the project, it’s important not to overdo it. Too many modules can make the project more difficult to manage and can hurt performance. The goal is to find a balance between maintainability and performance.
  + Build Performance: In multi-module Maven builds, sometimes incremental builds can be slow if dependencies are not set up properly. Discuss how to handle large-scale projects efficiently.
* Use of Profiles in Multi-Module Projects
  + Profiles: Spring allows you to define different profiles for different environments (e.g., dev, test, prod). You can configure specific settings for each module depending on the environment, and Maven profiles can help you activate specific modules or configurations during builds.

Parent POM

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.3.1</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<packaging>pom</packaging>

<groupId>com.rusty</groupId>

<artifactId>employee-management</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>employee-management</name>

<description>Employee Management Parent Project</description>

<properties>

<java.version>17</java.version>

</properties>

<modules>

<module>controller</module>

<module>service</module>

<module>repository</module>

</modules>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

</dependencies>

</project>

Child POM

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.rusty</groupId>

<artifactId>employee-management</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>service</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>service</name>

<description>Service Module</description>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>com.rusty</groupId>

<artifactId>repository</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

# KAFKA

* Apache Kafka is a distributed event streaming platform designed to handle real-time data feeds with high throughput, scalability, and fault tolerance. Let's dive into Kafka, covering its core concepts, use cases, setup, and coding examples in Java.
  + Distributed: It runs on a cluster of servers.
  + Publish-Subscribe Messaging System: Producers send messages to topics, and consumers read messages from those topics.
  + Event Streaming Platform: Processes and stores streams of records in real-time.
  + Fault-Tolerant: Replicates data across multiple nodes to ensure reliability.
* Core Concepts
  + Topics: Logical channels where messages are sent and received.
  + Topics are partitioned for scalability.
  + Producers: Applications that publish messages to topics.
  + Consumers: Applications that subscribe to topics to read messages.
  + Brokers: Kafka servers that store and distribute data.
  + Partitions: Topics are split into partitions for parallelism and scalability.
  + Offset: Unique identifier for each message in a partition.
  + Consumer Groups: Group of consumers that share the workload of consuming messages from a topic.
  + ZooKeeper: Used for managing Kafka's metadata and cluster coordination (now being replaced by Kafka Raft in newer versions).

Why Use Kafka?

* Handles High Throughput: Suitable for large-scale data pipelines and real-time processing.
* Scales Horizontally: Add more brokers to handle increased load.
* Provides Durability: Messages are written to disk and replicated.
* Low Latency: Enables near real-time data streaming.
* Integrates Easily: Works well with tools like Spark, Hadoop, and Elasticsearch.

Use Case Scenarios

* Log Aggregation: Centralize logs from various systems for monitoring and analysis.
* Real-Time Analytics: Process real-time data streams for analytics (e.g., fraud detection).
* Event Sourcing: Capture changes to application state as a sequence of events.
* Message Queuing: Acts as a message broker for asynchronous communication.
* Data Integration: Acts as a central hub to integrate data from different systems.
* Streaming ETL: Extract, transform, and load data in real-time.

Kafka Setup

* Normal Setup from Command Line
  + Download Kafka:

Get Kafka from Kafka website (<https://kafka.apache.org/downloads>).

* + Extract the tarball:

tar -xzf kafka\_2.13-<version>.tgz

cd kafka\_<version>

* + Start Zookeeper: Kafka requires ZooKeeper to manage cluster state (for older versions):

bin/zookeeper-server-start.sh config/zookeeper.properties

bin/windows/zookeeper-server-start.bat ..\..\config\zookeeper.properties

( start zookeeper and provide properties )

* + Start Kafka Broker:

bin/kafka-server-start.sh config/server.properties

bin/windows/kafka-server-start.bat ..\..\config\server.properties

( start kafka and broker and give server properties )

* + Create a Topic:

bin/kafka-topics.sh --create --topic test-topic --bootstrap-server

localhost:9092 --partitions 3 --replication-factor 1

kafka-topics.bat --create --topic my-topic --bootstrap-server

localhost:9092 --replication-factor 1 --partitions 3

( create topic (bootstrap-server => broker) )

* + Produce Messages:

bin/kafka-console-producer.sh --topic test-topic

--bootstrap-server localhost:9092

kafka-console-producer.bat --broker-list localhost:9092 --topic my-topic

( starts the producer )

* + Consume Messages:

bin/kafka-console-consumer.sh --topic test-topic

--bootstrap-server localhost:9092 --from-beginning

kafka-console-consumer.bat --bootstrap-server localhost:9092

--topic my-topic --from-beginning

( starts the consumer )

* + If key is not send then the values are saved in round robin fashion
  + If key is send then partitioner checks the key and generates hash value
  + Produce Messages with Keys:

kafka-console-producer.bat --broker-list localhost:9092 --topic fruits

--property "key.separator=-" --property "parse.key=true"

( starts the producer and tells that the key are coming which are

separated by the – and also parse should be true )

Eg->

>hello-apple

>hello-banana

>hello-kiwi

>bye-papaya

>bye-graper

* + Consume Messages with Keys:

kafka-console-consumer.bat --bootstrap-server localhost:9092

--topic fruits --from-beginning --property "key.separator=-"

--property "print.key=false"

( Starts the consumer and tells that the key is separated by the – and

the printing of the key should be either true or false )

* + Lists all the topics:

kafka-topics.bat --list --bootstrap-server localhost:9092

* + Lists the Consumer Groups:

kafka-consumer-groups.bat --list --bootstrap-server localhost:9092

* + Gives a Particular Group to a Consumer:

kafka-console-consumer.bat --bootstrap-server localhost:9092

--topic my-topic --group console-consumer-50913

* + Describes a Topic:

.\kafka-topics.bat --describe --topic my-topic --bootstrap-server

localhost:9092

Kafka Setup in JAVA

Example Producer Config and usage

*@Configuration*

*public class KafkaProducerConfig {*

*@Bean*

*public ProducerFactory<String, String> producerFactory() {*

*Map<String, Object> configProps = new HashMap<>();*

*configProps.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");*

*configProps.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG,*

*StringSerializer.class);*

*configProps.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG,*

*StringSerializer.class);*

*return new DefaultKafkaProducerFactory<>(configProps);*

*}*

*@Bean*

*public KafkaTemplate<String, String> kafkaTemplate() {*

*return new KafkaTemplate<>(producerFactory());*

*}*

*}*

* Key Components:
  + @Configuration:
    - Marks the class as a configuration class for Spring.
    - Allows Spring to process it and generate beans as per the methods annotated with @Bean.
  + Producer Factory:
    - Creates instances of Kafka producers.
    - Uses configurations such as:
      * BOOTSTRAP\_SERVERS\_CONFIG:
        + Specifies the Kafka broker(s) to connect to.
      * KEY\_SERIALIZER\_CLASS\_CONFIG:
        + Defines how keys are serialized (converted to bytes).
      * VALUE\_SERIALIZER\_CLASS\_CONFIG:
        + Defines how values are serialized.
* KafkaTemplate:
  + A Spring utility built on the producer factory.
  + Simplifies sending messages to Kafka.
* Methods:
  + producerFactory():
    - Returns a ProducerFactory object configured with the necessary properties.
    - The ProducerFactory is responsible for creating producer instances.
  + kafkaTemplate():
    - Returns a KafkaTemplate object that uses the ProducerFactory.
    - Used in EmployeeEventProducer for sending messages.
* Configurations in producerFactory():
  + ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG:
    - Specifies the Kafka cluster (broker addresses) the producer connects to.
    - Example: "localhost:9092".
  + ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG:
    - Defines the serializer for keys.
    - Example: StringSerializer.class.
  + ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG:
    - Defines the serializer for values.
    - Example: StringSerializer.class.

*@Component*

*public class EmployeeEventProducer {*

*private static final String TOPIC = "employee-events";*

*@Autowired*

*private KafkaTemplate<String, String> kafkaTemplate;*

*public void sendEvent(String key, String message) {*

*kafkaTemplate.send(TOPIC, key, message);*

*}*

*}*

* Key Components:
  + @Component:
    - Marks the class as a Spring-managed component.
    - Enables Spring to detect and manage this class as a bean.
  + KafkaTemplate<String, String>:
    - A Spring-provided utility for sending messages to Kafka topics.
    - Uses the KafkaProducer internally to send messages.
    - Supports sending messages with or without keys.
* Methods:
  + sendEvent(String key, String message):
    - Sends a message to the employee-events topic.
    - Key Parameter:
      * Allows messages to be routed to specific partitions.
      * Ensures that messages with the same key are sent to the same partition.

Example Consumer Config and usage

*@Component*

*public class EmployeeEventConsumer {*

*@KafkaListener(topics = "employee-events", groupId = "employee\_group")*

*public void listen(String message) {*

*System.out.println("Received message: " + message);*

*}*

*}*

* Key Components:
  + @Component:
    - Marks the class as a Spring-managed component.
    - Enables Spring to detect and manage this class as a bean.
  + @KafkaListener:
    - A Spring annotation used to define a method as a Kafka message listener.
    - Listens to specified Kafka topics and invokes the method when messages are received.
* Configuration:
  + Topics:
    - The @KafkaListener annotation specifies the Kafka topic to listen to (e.g., employee-events).
  + Group ID:
    - Defines the consumer group ID (employee\_group).
    - Consumers in the same group share the load of processing messages from partitions of a topic.
* Method:
  + listen(String message):
    - Processes the incoming message from the Kafka topic.
    - The message parameter contains the deserialized message value.