Spring Boot

Banuprakash C

banu@lucidatechnologies.com

banuprakashc@yahoo.co.in

Enable Caching

- EnableCaching annotation would register the same ConcurrentMapCacheManager.
- There is no need for a separate bean declaration for CacheManager.

```
@SpringBootApplication
@EnableCaching
public class CacheExampleApplication implements CommandLineRunner {
```

Enable Caching

- @Cacheable
  - enable caching behavior with the name of the cache where the results would be stored
- @CacheEvict
  - removal one or more/all values – so that fresh values can be loaded into the cache again

```
@Service
public class DeliveryService {
   @Autowired
   DeliveryRepository deliveryRepository;

   @Cacheable("deliveries")
   public List<DeliveryDto> findAll(){
      ...
   }

   @Cacheable("delivery")
   public DeliveryDto findById(Long id){
     ....
   }

   @Caching(evict = {
         @CacheEvict(value="delivery", allEntries=true),
         @CacheEvict(value="deliveries", allEntries=true)})
   public Delivery saveOrUpdate(DeliveryDto deliveryDto){
      ...
}
}
```

Enable Caching

- @CachePut
  - update the content of the cache without interfering the method execution. That is, the method would always be executed and the result cached

```
@CachePut(value="addresses")
public String getAddress(Customer customer) {...}

@CachePut(value="addresses", unless="#result.length()<64")
public String getAddress(Customer customer) {...}
```

Spring Data JPA

- Spring Data JPA @Modifying Annotation
  - The @Modifying annotation is used to enhance the @Query annotation to execute not only SELECT queries but also INSERT, UPDATE, DELETE, and even DDL queries.

  - @Modifying@Query("update User u set u.active = false where u.lastLoginDate < :date")void void deactivateUsersNotLoggedInSince(@Param("date") LocalDate date);

  - @Modifying(clearAutomatically = true)
    - persistence context is cleared after our query execution
  - @Modifying(flushAutomatically = true)
    - the EntityManager is flushed before our query is executed.

Spring Data JPA Query Projections

- Scalar Projections
  - Allows you to select entity attributes instead of fetching the entire entity

  @Query(“SELECT name, price from Product”)

  List<Object[]> getNameAndPrice()

- DTO Projections
  - Create a DTO class and use constructor expression in query

  @Query(“SELECT new com.banu.prj.ReportDTO(c.email, o.orderDate, o.total) from Order o left outer join o.customer c”)

  List<ReportDTO> getReportDTO()

Spring Data JPA Query Projections

- Derived Query without constructor expression
  - Write DTO class with only one constructor and parameters matching names of entity class attributes, spring generates a query with required constructor expression
  - List<ProductDTO> findByName(String name);
- DTOs as interface
  - Instead of defining a class, we can use interface with only getters as DTO projection

```
public interface ProductDTO {
    String getName();
    String getPrice();
}
```

Spring Data JPA Query Projections

- Nested Associations
  - Spring Data maps List of orders to OrderDTO
  - This uses n+ 1 select

```
public interface CustomerDTO {
        String getName();
        List<OrderDTO> getOrders();
        interface OrderDTO {
                Date getOrderDate();
                double getTotal();
        }
}
```

Spring Data JPA Query Projections

- Dynamic Projections
  - Depending on the class you provide when you call the repository method, Spring Data JPA uses one of the different mechanism to define the projection and map it

```
public interface CustomerDAO  extends CrudRepository<Customer, String> {
   <T> T findByName(String name, Class<T> type);
}
```

```
CustomerDTO customerDTO = customerDao.findByName("Banu", CustomerDTO.class);

Customer customer = customerDao.findByName("Banu", Customer.class);
```

Spring Data JPA Query Projections

- **Using Spring's Expression Language**

Pagination

- Paginating the Query Results of Our Database Queries
  - We can paginate the query results of our database queries by following these steps:
  - Obtain the Pageable object that specifies the information of the requested page.
    - We can create it manually.
    - We can use Spring Data web support.
  - Pass the Pageable object forward to the correct repository method as a method parameter.

Pagination

- Slice and Page
  - Page is just a sub-interface of Slice with a couple of additional methods.
  - A Page contains information about the total number of elements and pages available in the database. It is because the Spring Data JPA triggers a count query to calculate the number of elements.
  - To avoid this costly count query, you should instead return a Slice. Unlike a Page, a Slice only knows about whether the next slice is available or not. This information is sufficient to walk through a larger result set.
  - You should use Slice if you don't need the total number of items and pages. A good example of such a scenario is when you only need Next Page and Previous Page buttons.

Query by Example

- The Query by Example API consists of three parts:
  - Probe: The actual example of a domain object with populated fields.
  - ExampleMatcher:
    - The ExampleMatcher carries details on how to match particular fields. It can be reused across multiple Examples.
  - Example:
    - An Example consists of the probe and the ExampleMatcher. It is used to create the query.

- Query by Example also has several limitations:
  - No support for nested or grouped property constraints, such as firstname = ?0 or (firstname = ?1 and lastname = ?2).
  - Only supports starts/contains/ends/regex matching for strings and exact matching for other property types.

- Query by Example is well suited for several use cases:
  - Querying your data store with a set of static or dynamic constraints.
  - Frequent refactoring of the domain objects without worrying about breaking existing queries.
  - Working independently from the underlying data store API.

Query by Example

- QueryByExampleExecutor interface to support query by example:

```
public interface QueryByExampleExecutor<T> {
    <S extends T> Optional<S> findOne(Example<S> var1);
    <S extends T> Iterable<S> findAll(Example<S> var1);
    <S extends T> Iterable<S> findAll(Example<S> var1, Sort var2);
    <S extends T> Page<S> findAll(Example<S> var1, Pageable var2);
    <S extends T> long count(Example<S> var1);
    <S extends T> boolean exists(Example<S> var1);
}
```

```
public interface Example<T> {

   static <T> Example<T> of(T probe) {
      return new TypedExample(probe, ExampleMatcher.matching());
   }

   static <T> Example<T> of(T probe, ExampleMatcher matcher) {
      return new TypedExample(probe, matcher);
   }

   T getProbe();

   ExampleMatcher getMatcher();
}
```

Query by Example

- `import org.springframework.data.domain.Example;`
- `import org.springframework.data.domain.ExampleMatcher;`
- `import static org.springframework.data.domain.ExampleMatcher.GenericPropertyMatchers.exact;`

```
            Person p = new Person();
            p.setLastName("R");

            ExampleMatcher matcher = ExampleMatcher.matching()
                    .withMatcher("lastName", match -> match.startsWith())
//                  .withMatcher("lastName", exact())
                    .withIgnorePaths("age");

            Example<Person> example = Example.of(p, matcher);
            personDao.findAll(example).forEach(System.out::println);;
```

```
                    .withMatcher("lastName", ExampleMatcher.GenericPropertyMatchers.endsWith())
```

SpEL support in Spring Data JPA

- #{#entityName}
  - To avoid stating the actual entity name in the query string of a @Query

```
@Query("select u from #{#entityName} u where u.lastname = ?1")
List<User> findByLastname(String lastname);
```

```
@Entity
public class User {

@Entity(name = "MyUser")
public class User {
```

SpEL support in Spring Data JPA

- SpEL for LIKE expressions

```
@Query("select e from Employee e where e.name like %:#{[0]}% and e.name like %:n%")
List<Employee> searchByName(@Param("n") String name);
```

- When using like-conditions with values that are coming from a not secure source the values should be sanitized so they can't contain any wildcards and thereby allow attackers to select more data than they should be able to.
- For this purpose the escape(String) method is made available in the SpEL context.
- It prefixes all instances of _ and % in the first argument with the single character from the second argument.
- `List<Employee> emps = employeeDao.searchByName("Raj_ev");`

Spring Boot JPA entity graphs

Applying the Entity graphs fetches the mapped entities eagerly while querying the database.

We can use Entity graphs to eliminate the N+1 select query problem.

- Entity graph fetch types
  - FetchGraph: Applying this fetch type results in fetching all the entity graph attribute nodes eagerly. Other mapped entities will be lazily fetched irrespective of the specified fetch type.
  - LoadGraph: Applying this fetch type results in fetching all the entity graph attribute nodes eagerly. Other mapped entities are treated based on the default/specified fetch type.

Spring Boot JPA entity graphs

```
@NamedEntityGraphs({
        @NamedEntityGraph(name = "companyWithDepartmentsGraph",
                attributeNodes = {@NamedAttributeNode("departments")}),

        @NamedEntityGraph(name = "companyWithDepartmentsAndEmployeesGraph",
                attributeNodes = {@NamedAttributeNode(value = "departments", subgraph = "departmentsWithEmployees")},
                subgraphs = @NamedSubgraph(
                        name = "departmentsWithEmployees",
                        attributeNodes = @NamedAttributeNode("employees"))),

        @NamedEntityGraph(name = "companyWithDepartmentsAndEmployeesAndOfficesGraph",
                attributeNodes = {@NamedAttributeNode(value = "departments", subgraph = "departmentsWithEmployeesAndOffices")},
                subgraphs = @NamedSubgraph(
                        name = "departmentsWithEmployeesAndOffices",
                        attributeNodes = {@NamedAttributeNode("employees"), @NamedAttributeNode("offices")}))
})
public class Company {
    @Id
        @Column(name = "id", updatable = false, nullable = false)
        private Long id = null;

        private String name;

        @OneToMany(mappedBy = "company", fetch = FetchType.LAZY)
        @JsonManagedReference
        private Set<Department> departments = new HashSet<>();

        @OneToMany(mappedBy = "company", fetch = FetchType.LAZY)
        @JsonManagedReference
        private Set<Car> cars = new HashSet<>();
```

```
@Entity
public class Department {
    @Id
        @Column(name = "id", updatable = false, nullable = false)
        private Long id = null;

        private String name;

        @OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
        @JsonManagedReference
        private Set<Employee> employees = new HashSet<>();

        @OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
        @JsonManagedReference
        private Set<Office> offices = new HashSet<>();
```

Spring Data JPA Specifications

- Criteria API
  - Criteria API offers a programmatic way to create typed queries, which helps us avoid syntax errors
  - Use case: send a voucher to all long term customers on their birthday's

```
LocalDate today = new LocalDate();

CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> query = builder.createQuery(Customer.class);
Root<Customer> root = query.from(Customer.class);

Predicate hasBirthday = builder.equal(root.get(Customer_.birthday), today);
Predicate isLongTermCustomer = builder.lessThan(root.get(Customer_.createdAt), today.minusYears(2);
query.where(builder.and(hasBirthday, isLongTermCustomer));

em.createQuery(query.select(root)).getResultList();
```

Spring Data JPA Specifications

- Specifications
  - To reuse Predicate

```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery query, CriteriaBuilder cb);
}
```

Spring Data JPA Specifications

- Example

```
public enum SearchOperation {
    GREATER_THAN,
    LESS_THAN,
    GREATER_THAN_EQUAL,
    LESS_THAN_EQUAL,
    NOT_EQUAL,
    EQUAL,
    MATCH,
    MATCH_START,
    MATCH_END,
    IN,
    NOT_IN
}
```

```
public class SearchCriteria {
    private String key;
    private Object value;
    private SearchOperation operation;

    public SearchCriteria() {

    public SearchCriteria(String key, Object value, SearchOperation operation) {

    public String getKey() {

    public void setKey(String key) {

    public Object getValue() {

    public void setValue(Object value) {

    public SearchOperation getOperation() {

    public void setOperation(SearchOperation operation) {

    public String toString() {
}
```

Spring Data JPA Specifications: Specific implementation

```java
public class MovieSpecification implements Specification<Movie> {

    private static final long serialVersionUID = 1L;

    private List<SearchCriteria> list;

    public MovieSpecification() {
        this.list = new ArrayList<>();
    }

    public void add(SearchCriteria criteria) {
        list.add(criteria);
    }
```

```java
    @Override
    public Predicate toPredicate(Root<Movie> root, CriteriaQuery<?> query, CriteriaBuilder builder) {

        //create a new predicate list
        List<Predicate> predicates = new ArrayList<>();

        //add add criteria to predicates
        for (SearchCriteria criteria : list) {
            if (criteria.getOperation().equals(SearchOperation.GREATER_THAN)) {
                predicates.add(builder.greaterThan(
                        root.get(criteria.getKey()), criteria.getValue().toString()));
            } else if (criteria.getOperation().equals(SearchOperation.LESS_THAN)) {
                predicates.add(builder.lessThan(
                        root.get(criteria.getKey()), criteria.getValue().toString()));
            } else if (criteria.getOperation().equals(SearchOperation.GREATER_THAN_EQUAL)) {
                predicates.add(builder.greaterThanOrEqualTo(
                        root.get(criteria.getKey()), criteria.getValue().toString()));
            }  else if (criteria.getOperation().equals(SearchOperation.NOT_EQUAL)) {
                predicates.add(builder.notEqual(
                        root.get(criteria.getKey()), criteria.getValue()));
            } else if (criteria.getOperation().equals(SearchOperation.EQUAL)) {
                predicates.add(builder.equal(
                        root.get(criteria.getKey()), criteria.getValue()));
            } else if (criteria.getOperation().equals(SearchOperation.MATCH)) {
                predicates.add(builder.like(
                        builder.lower(root.get(criteria.getKey())),
                        "%" + criteria.getValue().toString().toLowerCase() + "%"));
            } else if (criteria.getOperation().equals(SearchOperation.IN)) {
                predicates.add(builder.in(root.get(criteria.getKey())).value(criteria.getValue()));
            }
        }

        return builder.and(predicates.toArray(new Predicate[0]));
    }
}
```

Spring Data JPA Specifications

- To execute specification we need to extend dao repository interface with JpaSpecificationExecutor

```
public interface MovieDao extends CrudRepository<Movie, Long>, JpaSpecificationExecutor<Movie> {

}
```

```
@Autowired
MovieDao movieDao;
```

```
// search movies by `title` and `rating` > 7
System.out.println("search movies by `title` and `rating` > 7");
MovieSpecification msTitleRating = new MovieSpecification();
msTitleRating.add(new SearchCriteria("title", "black", SearchOperation.MATCH));
msTitleRating.add(new SearchCriteria("rating", 7, SearchOperation.GREATER_THAN));
List<Movie> msTitleRatingList = movieDao.findAll(msTitleRating);
msTitleRatingList.forEach(System.out::println);
System.out.println();
```

Spring Data — Envers

- **Maintain the data versioning info**
- Hibernate Envers project aimed to track data changes at the entity level with easy configurations in properties level and entity class level using annotations
- The spring-data-envers project builds on top of Hibernate Envers and comes up as an extension of the Spring Data JPA project

Spring Data — Envers

Add the following dependency:

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

Spring Data — Envers

- Add the @Audited annotation either on an @Entity (to audit the whole entity) or on specific @Columns (if you need to audit specific properties only)

```
@Entity
@Audited
public class Post {

    @OneToMany()
    @Audited
    private Set<Comment> comments;

    @ManytoOne()
    @NotAudited
    private Set<Comment> comments;

}
```

Spring Data — Envers

- audit tables:
  - <EntityName>_AUD (if you've set EntityName as @Audited) table should be generated automatically.
  - The audit tables copy all audited fields from the entity's table with two fields, REVTYPE (values are: “0” for adding, “1” for updating, “2” for removing an entity) and REV.
  - Besides these, an extra table named REVINFO will be generated by default, it includes two important fields
    - REV and REVTSTMP and records the timestamp of every revision.

Spring Data — Envers

- We can customized audit table prefix, suffix, and a few other naming conventions by changing the following properties.
  - Spring.jpa.properties.org.hibernate.envers.audit_table_prefix
  - Spring.jpa.properties.org.hibernate.envers.audit_table_suffix (Default value _AUD)
  - Spring.jpa.properties.org.hibernate.envers.revision_field_name (Default value — REV)
  - Spring.jpa.properties.org.hibernate.envers.revision_type_field_name — (Default value — REVTYPE)

Spring Data — Envers

- Repository interface
- RevisionRepository<T,ID,N extends Number & Comparable<N>>
  - By implementing RevisionRepository allows the following 4 different methods to get versioning information:

`Optional<Revision<N,T>>`	**`findLastChangeRevision`**`(`**`ID`**` id)` Returns the revision of the entity it was last changed in.
`Optional<Revision<N,T>>`	**`findRevision`**`(`**`ID`**` id, `**`N`**` revisionNumber)` Returns the entity with the given ID in the given revision number.
`Revisions<N,T>`	**`findRevisions`**`(`**`ID`**` id)` Returns all **`Revisions`** of an entity with the given id.
`Page<Revision<N,T>>`	**`findRevisions`**`(`**`ID`**` id, `**`Pageable`**` pageable)` Returns a **`Page`** of revisions for the entity with the given id.

Spring Data — Envers

- EnversRevisionRepositoryFactoryBean
  - FactoryBean creating RevisionRepository instances.

  - Integrating spring-data-envers project to your Spring Boot project.

```
@SpringBootApplication
@EnableJpaRepositories(repositoryFactoryBeanClass = EnversRevisionRepositoryFactoryBean.class)
```

Spring Data envers

- Tables on inserting products

SELECT * FROM PRODUCTS;

ID	NAME	PRICE
1	iPhone 12	98000.0

SELECT * FROM PRODUCTS_AUD;

ID	REV	REVTYPE	NAME	PRICE
1	1	0	iPhone 12	98000.0

SELECT * FROM REVINFO;

REV	REVTSTMP
1	1604206708704

SELECT * FROM PRODUCTS;

ID	NAME	PRICE
1	iPhone 12	98000.0
2	HP Laptop	135000.0

SELECT * FROM PRODUCTS_AUD;

ID	REV	REVTYPE	NAME	PRICE
1	1	0	iPhone 12	98000.0
2	2	0	HP Laptop	135000.0

SELECT * FROM REVINFO;

REV	REVTSTMP
1	1604206708704
2	1604206819340

Spring Data envers

- Updating a product

SELECT * FROM PRODUCTS_AUD;

ID	REV	REVTYPE	NAME	PRICE
1	1	0	iPhone 12	98000.0
2	2	0	HP Laptop	135000.0
2	3	1	HP Laptop	76000.0

SELECT * FROM REVINFO;

REV	REVTSTMP
1	1604207363816
2	1604207369235
3	1604207430339

- Deleting a Product

SELECT * FROM PRODUCTS_AUD;

ID	REV	REVTYPE	NAME	PRICE
1	1	0	iPhone 12	98000.0
2	2	0	HP Laptop	135000.0
2	3	1	HP Laptop	76000.0
2	4	2	*null*	*null*

SELECT * FROM REVINFO;

REV	REVTSTMP
1	1604207363816
2	1604207369235
3	1604207430339
4	1604207580576

Protocol buffers, or Protobuf

- Binary format created by Google to serialize data between different services.
- It provides support, out of the box, to the most common languages, like JavaScript, Java, C#, Ruby and others
- **Protobuf performs up to 6 times faster than JSON**
- **Two Java backends communicating results**

The following results were extracted from executing 40 requests per protocol on the test sample.

	gzipped time (avg)	non-gzipped time (avg)
protobuf	234ms	146ms
json	701ms	590ms

Protobuf

- Protobuf is more than a message format, it is also a set of rules and tools to define and exchange messages.
- Protobuf has more data types than JSON, like enumerates and methods, and is also heavily used on RPCs (Remote Procedure Calls).

How Do We Use Protobuf?

- Protobuf has three main components that we have to deal with:
  - Message descriptors: we have to define our messages structures in .proto files.
  - Message implementation: We have to generate classes/objects to deal with data in the chosen programming language.
  - Parsing and Serialization. After defining and creating Protobuf messages, we need to be able to exchange these messages.