Spring 6 Data JPA - Data Access with JPA

Written by: Hong Le Nguyen, last update: 02.2025

Code to example on Github : https://github.com/hong1234/spring6-dataJpa

1 - Infrastructure beans configuration

We use Spring Data Jpa to implement database access layer.

Let's assume that the data source is a MySQL database, the Jpa vendor is Hibernate and the configuration properties are stored in the database.properties file, represented by DatabaseProperties class.

DataSource enable

dataSource bean of type javax.sql.DataSource, represents database MySQL as data source

```
@Configuration
public class DataSourceConfig {
   private final DatabaseProperties dbProperties;
   @Bean
   public DataSource dataSource() { … }
```

Jpa Layer configuration

The bean entityManagerFactory of type LocalContainerEntityManagerFactoryBean needs to know the following things

data source : represented by dataSource bean
jpa vendor : is an object of type HibernateJpaVendorAdapter
jpa properties : are Hibernate jpa properties stored in the database.properties file
and metadata : supplied by the entities, repositories classes, we locate the entities and repositories as follows

```
@EnableJpaRepositories(basePackages = {DataJpaConfig.JPA_REPO_LOCATION})
factory.setPackagesToScan(JPA_ENTITY_LOCATION);
```

For database initialization via metadata we set property "hibernate.hbm2ddl.auto" to "update"

```
properties.put("hibernate.hbm2ddl.auto", "update");
```

Transaction enable

bean transactionManager of type PlatformTransactionManager, represents Jpa Transaction Management

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = {DataJpaConfig.JPA_REPO_LOCATION})
public class DataJpaConfig {

   public static final String JPA_REPO_LOCATION = "com.hong.spring.repo";
   public static final String JPA_ENTITY_LOCATION = "com.hong.spring.entity";
   private final DataSource dataSource;
   private final DatabaseProperties dbProperties;
```

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
   HibernateJpaVendorAdapter jpaVendor = new HibernateJpaVendorAdapter();
   Properties jpaProperties = dbProperties.hibernateProperties();
   LocalContainerEntityManagerFactoryBean factory =
                                 new LocalContainerEntityManagerFactoryBean();
   factory.setDataSource(dataSource);
   factory.setJpaVendorAdapter(jpaVendor);
   factory.setJpaProperties(jpaProperties);
   factory.setPackagesToScan(JPA_ENTITY_LOCATION);
   return factory;
 }

// transactions enable
@Bean
public PlatformTransactionManager transactionManager() {
   JpaTransactionManager transactionManager = new JpaTransactionManager();
   transactionManager.setEntityManagerFactory(entityManagerFactory().getObject());
   return transactionManager;
}
```

JPA configuration in Spring Boot 3

To use Spring Data JPA in your Spring Boot application, we add the spring-boot-starter-data-jpa dependency. Spring Boot *auto-configures* the dataSource, entityManager, transactionManager beans.

To override the defaults, you need to provide your own bean declaration, either per Java config or in the application.properties file. For example fine-tuning Jpa properties using application.properties file.

```
…

# JPA

hibernate.show_sql=true

hibernate.hbm2ddl.auto=update

# hibernate.dialect=
```

2 - Repository Implementation with Spring Data JPA

Using the CrudRepository<T, ID> interface

You specify the repository, Spring Data JPA generates the repository implementation for you.

For example, the CustomerRepository interface, the basic CRUD operations provided by CrudRepository.

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

Customizing repositories

In addition to the basic CRUD operations, you also need the other operations.

Implicit query per method naming convention --

To fetch all the customers to a given last name, you can add the following method declaration to CustomerRepository:

```
List<Customer> findByLastName(String lastName);
```

When generating the repository implementation, Spring Data examines each method in the repository interface, parses the method name, and attempts to understand the method's purpose in the context of the persisted object (a Customer, in this case).

Explicitly specify the query with @Query annotation --

for more complex queries to be performed when the method is called. This example shows:

```
public interface BookRepository extends CrudRepository<Book, Integer> {

  @Query("from Book b where b.title like %?1%")                  // a custom JPQL statement
  List<Book> searchByTitle(String title);

  @Query(value="select * from books where title = ?1", nativeQuery=true)  // a pure SQL
  List<Book> findBookWithPureSql(String bookTitle);
```

Using entityManager

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.PersistenceContextType;

@Repository
public class CustomRepositoryImpl // implements CustomRepository {

  @PersistenceContext // (type = PersistenceContextType.EXTENDED)
  private EntityManager entityManager;

  // @Override
  public List<Customer> getAll() {
    String jpql = "SELECT c FROM Customer c";   // a custom JPQL statement
    TypedQuery<Customer> query = entityManager.createQuery(jpql, Customer.class);
    return query.getResultList();
  }

  public void insertUser(User user) {
    entityManager.persist(user);
  }
```

3 - Transaction

Once you have configured a transaction manager, you can use the @Transactional annotation to mark your methods and classes as transactional.

```
import jakarta.transaction.Transactional;
@Service
@Transactional
@RequiredArgsConstructor
public class BookService {
  private final BookRepository bookRepository;
  private final ReviewRepository reviewRepository;
```