Customizing Spring Data JPA Repository

by Boris Lam MVB · Sep. 27, 12 · Java Zone

Spring Data is a very convenient library. However, as the project as quite new, it is not well featured. By default, Spring Data JPA will provide implementation of the DAO based on SimpleJpaRepository. In recent project, I have developed a customize repository base class so that I could add more features on it. You could add vendor specific features to this repository base class as you like.

Configuration

You have to add the following configuration to you spring beans configuration file. You have to specified a new repository factory class. We will develop the class later.

```
<jpa:repositories base-package="example.borislam.dao"
factory-class="example.borislam.data.springData.DefaultRepositoryFactoryBean/>
```

Just develop an interface extending JpaRepository. You should remember to annotate it with @NoRepositoryBean.

```
@NoRepositoryBean
public interface GenericRepository <T, ID extends Serializable>
 extends JpaRepository<T, ID> {
}
```

Define Custom repository base implementation class

Next step is to develop the customized base repository class. You can see that I just one property (i.e. springDataRepositoryInterface) inside this customized base repository. I just want to get more control on the behaviour of the customized behaviour of the repository interface. I will show how to add more features of this base repository class in the next post.

```
@SuppressWarnings("unchecked")
@NoRepositoryBean
public class GenericRepositoryImpl<T, ID extends Serializable>
 extends SimpleJpaRepository<T, ID>  implements GenericRepository<T, ID> , Serializable{

 private static final long serialVersionUID = 1L;

 static Logger logger = Logger.getLogger(GenericRepositoryImpl.class);

    private final JpaEntityInformation<T, ?> entityInformation;
    private final EntityManager em;
    private final DefaultPersistenceProvider provider;

    private  Class<?> springDataRepositoryInterface;
 public Class<?> getSpringDataRepositoryInterface() {
  return springDataRepositoryInterface;
 }

 public void setSpringDataRepositoryInterface(
   Class<?> springDataRepositoryInterface) {
  this.springDataRepositoryInterface = springDataRepositoryInterface;
 }

 /**
     * Creates a new {@link SimpleJpaRepository} to manage objects of the given
     * {@link JpaEntityInformation}.
     *
     * @param entityInformation
     * @param entityManager
     */
    public GenericRepositoryImpl (JpaEntityInformation<T, ?> entityInformation, EntityManager entityMa
     super(entityInformation, entityManager);
     this.entityInformation = entityInformation;
     this.em = entityManager;
     this.provider = DefaultPersistenceProvider.fromEntityManager(entityManager);
     this.springDataRepositoryInterface = springDataRepositoryInterface;
     }

    /**
     * Creates a new {@link SimpleJpaRepository} to manage objects of the given
     * domain type.
     *
     * @param domainClass
     * @param em
     */
    public GenericRepositoryImpl(Class<T> domainClass, EntityManager em) {
        this(JpaEntityInformationSupport.getMetadata(domainClass, em), em, null);
    }

    public <S extends T> S save(S entity)
    {
        if (this.entityInformation.isNew(entity)) {
            this.em.persist(entity);
            flush();
            return entity;
```

```
          }
  entity = this.em.merge(entity);
  flush();
        return entity;
    }


    public T saveWithoutFlush(T entity)
    {
      return
       super.save(entity);
    }

    public List<T> saveWithoutFlush(Iterable<? extends T> entities)
    {
     List<T> result = new ArrayList<T>();
  if (entities == null) {
   return result;
  }

  for (T entity : entities) {
   result.add(saveWithoutFlush(entity));
  }
  return result;
    }
}
```

As a simple example here, I just override the default save method of the SimpleJPARepository. The default behaviour of the save method will not flush after persist. I modified to make it flush after persist. On the other hand, I add another method called saveWithoutFlush() to allow developer to call save the entity without flush.

Define Custom repository factory bean

The last step is to create a factory bean class and factory class to produce repository based on your customized base repository class.

```
public class DefaultRepositoryFactoryBean <T extends JpaRepository<S, ID>, S, ID extends Serializable>
  extends JpaRepositoryFactoryBean<T, S, ID> {
    /**
     * Returns a {@link RepositoryFactorySupport}.
     *
     * @param entityManager
     * @return
     */
    protected RepositoryFactorySupport createRepositoryFactory(
            EntityManager entityManager) {

        return new DefaultRepositoryFactory(entityManager);
    }
}


/**
 *
 * The purpose of this class is to override the default behaviour of the spring JpaRepositoryFactory c
```

```
 * It will produce a GenericRepositoryImpl object instead of SimpleJpaRepository.
 *
 */
public  class DefaultRepositoryFactory extends JpaRepositoryFactory{

 private final EntityManager entityManager;
    private final QueryExtractor extractor;


    public DefaultRepositoryFactory(EntityManager entityManager) {
     super(entityManager);
        Assert.notNull(entityManager);
        this.entityManager = entityManager;
        this.extractor = DefaultPersistenceProvider.fromEntityManager(entityManager);
    }

    @SuppressWarnings({ "unchecked", "rawtypes" })
    protected <T, ID extends Serializable> JpaRepository<?, ?> getTargetRepository(
            RepositoryMetadata metadata, EntityManager entityManager) {

        Class<?> repositoryInterface = metadata.getRepositoryInterface();

        JpaEntityInformation<?, Serializable> entityInformation =
                getEntityInformation(metadata.getDomainType());

        if (isQueryDslExecutor(repositoryInterface)) {
            return new QueryDslJpaRepository(entityInformation, entityManager);
        } else {
            return new GenericRepositoryImpl(entityInformation, entityManager, repositoryInterface); /
        }
    }

    @Override
    protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {

        if (isQueryDslExecutor(metadata.getRepositoryInterface())) {
            return QueryDslJpaRepository.class;
        } else {
            return GenericRepositoryImpl.class;
        }
    }

    /**
     * Returns whether the given repository interface requires a QueryDsl
     * specific implementation to be chosen.
     *
     * @param repositoryInterface
     * @return
     */
    private boolean isQueryDslExecutor(Class<?> repositoryInterface) {

        return QUERY_DSL_PRESENT
                && QueryDslPredicateExecutor.class
                        .isAssignableFrom(repositoryInterface);
    }
}
```

Conclusion

You could now add more features to base repository class. In your program, you could now create your own repository interface extending GenericRepository instead of JpaRepository.

```
public interface MyRepository <T, ID extends Serializable>
  extends GenericRepository <T, ID> {
   void someCustomMethod(ID id);
}
```

In next post, I will show you how to add hibernate filter features to this GenericRepository.

Like This Article? Read More From DZone

Spring Tips: Reactive WebSockets With Spring Framework 5 [Video]

Abstraction Matters: Containers and App Delivery

5 Things That are Changing the Trends of Office Work Culture

Free DZone Refcard

Getting Started With Play Framework

Topics:

Okta

Get Started with Spring Boot, OAuth 2.0, and Okta

Okta