Creating Spring JPA Entity And Repository Dynamically

2021-10-14 15:00:00 +0530

# Introduction

At the outset, be warned that this is not for the fainthearted. Be prepared for a long and arduous read. Having said that, lets begin.

Many a times, I have wondered, if I can ever create a JPA Entity and a JPA Repository class dynamically, on the fly? Well, its certainly possible, though a bit complicated.

# Library REST Service

We will be building a mock library service with Spring Boot and a in-memory H2 Database. There are broadly 2 REST endpoints: 1. Author 1. Book

Both of these allow us to fetch all items, and also search by partial name. These are GET calls.

On the Book, we have an additional PUT call that allows us the change the Author of a Book.

# Start Simple

We will build a very normal JPA based Spring Boot Application. We have called this the *static-jpa-repo-simple*. It consists of 2 Entities and JPA Repos for Book and Author respectively.

This is the Book entity:

@Entity  
@Table(name = "book")  
@Data  
public class Book {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private int id;  
  
 @Column  
 private String name;  
  
 @OneToOne(fetch = FetchType.EAGER)  
 @JoinColumn(name = "author\_id")  
 private Author author;  
  
}

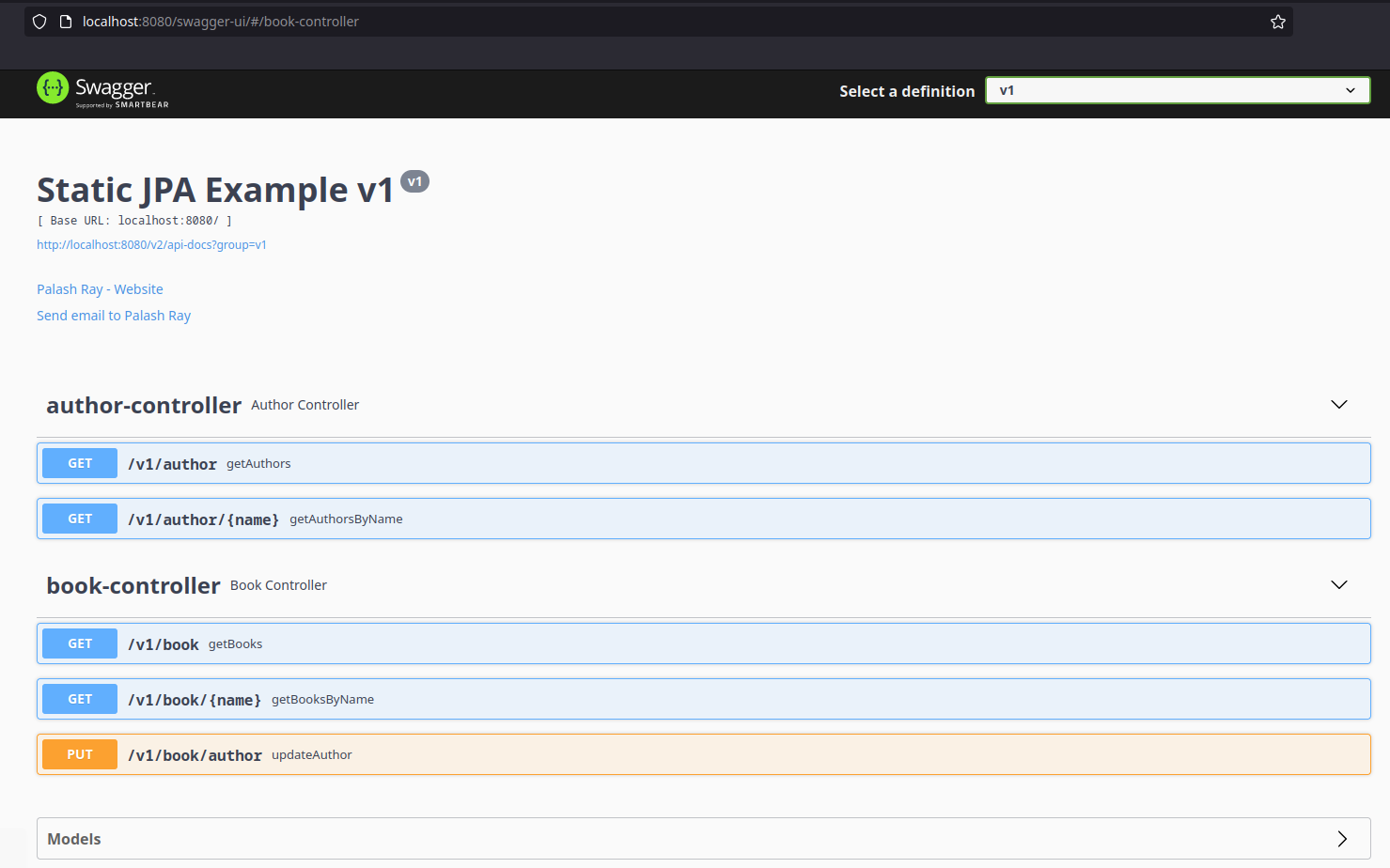
And this is the corresponding JPA Repository:

public interface BookDao extends CrudRepository<Book, Integer> {  
  
 List<Book> findByNameContainingIgnoreCase(String name);  
  
 @Transactional  
 @Modifying  
 @Query("update Book set author.id = :authorId where id = :bookId")  
 int updateAuthor(int bookId, int authorId);  
  
}

The first method *findByNameContainingIgnoreCase()*, as the name indicates, would do a case-insensitive search on the *name* of a *Book*.

The second method *updateAuthor()* would update an *Author* in a given *Book*.

Pretty mundane, I would say. If you browse to the Swagger Page: <http://localhost:8080/swagger-ui/>, this is how it looks like:



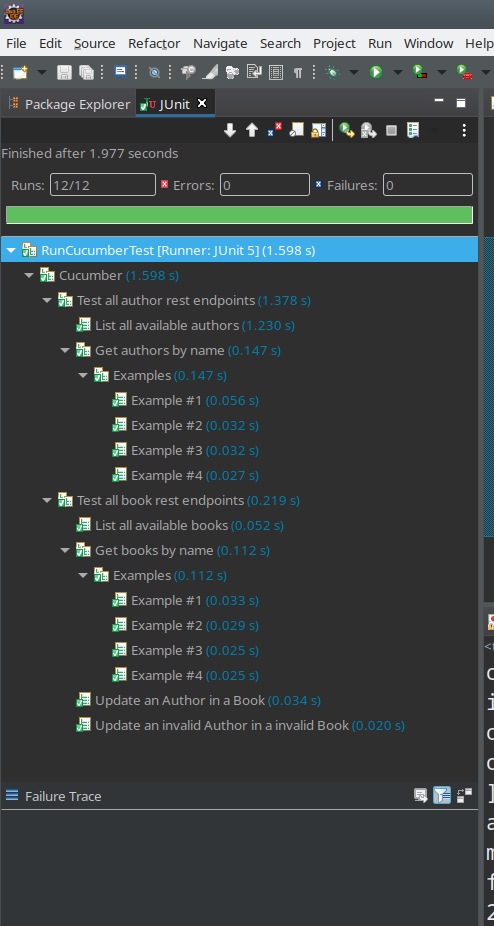
Swagger Page for Dynamic JPA

## Sources

These sources can be found here: <https://github.com/paawak/spring-boot-demo/tree/master/dynamic-jpa/static-jpa-repo-simple>

# Test Harness

At this point, before we delve any further, we will write a BDD/Cucumber based test harness to test all our REST endpoints. Run the main class such that the Application starts at *http://localhost:8080*. Then, run the *RunCucumberTest*. This is very similar to our previous entry: <https://palashray.com/example-of-creating-cucumber-based-bdd-tests-using-junit5-and-spring-dependency-injection/>. As shown below, ensure all the tests are green.



Run Cucumber Tests from Eclipse

# Making JPA Dynamic

OK, so what exactly are we trying to do? We are trying to generate the JPA Entity *Book* and the JPA Repository *BookDao* dynamically using *ByteBuddy*. Again, very similar to our previous blog post: <https://palashray.com/creating-class-dynamically-using-bytebuddy-and-springboot/>.

## Abstraction

To make this happen, we have to delete both of these classes as the first step. Since many other classes like the Rest Controller depends on these 2 classes, we have to first create an abstraction for these 2.

We will start by copying the *static-jpa-repo-simple*, and renaming it to *static-jpa-repo-abstraction*. We will refactor the code to suit our needs.

### Abstracting the Entity

For the Entity, we will first create an interface, closely resembling the *Book*:

public interface BookTemplate {  
  
 int getId();  
  
 void setId(int id);  
  
 String getName();  
  
 void setName(String name);  
  
 Author getAuthor();  
  
 void setAuthor(Author author);  
  
}

We will use this interface in place of *Book*, like in our Rest layer, the *BookController*. In the next step, to further simplify our Entity, we will create a *MappedSuperClass* with all the JPA annotated fields.

@MappedSuperclass  
@Data  
public class BookTemplateImpl implements BookTemplate {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private int id;  
  
 @Column  
 private String name;  
  
 @OneToOne(fetch = FetchType.EAGER)  
 @JoinColumn(name = "author\_id")  
 private Author author;  
  
}

Now, our Entity has become very simple, it just needs to extend the *BookTemplateImpl* and have a few JPA annotations of its own.

@Entity  
@Table(name = "book")  
public class Book extends BookTemplateImpl {  
  
}

### Abstracting the Dao

We will look at the *BookDao* and create a simple interface that does not extend the *CrudRepository*, with only the methods that would be used by us.

public interface BookDaoTemplate {  
  
 List<BookTemplate> findByNameContainingIgnoreCase(String name);  
  
 int updateAuthor(int bookId, int authorId);  
  
 Iterable<? extends BookTemplate> findAll();  
  
}

We would be using this in our entire project instead of the *BookDao*. As a side effect, our *BookDao* has become simpler:

public interface BookDao extends BookDaoTemplate, CrudRepository<Book, Integer> {  
  
 @Override  
 @Transactional  
 @Modifying  
 @Query("update Book set author.id = :authorId where id = :bookId")  
 int updateAuthor(int bookId, int authorId);  
  
}

### Registering the JPA Repository Dynamically

We have pretty much figured out how to generate the Entity and the Repository dynamically. However, once we have generated these classes, how to dynamically register the Repository as a Spring Bean?

The answer lies in configuring a *JpaRepositoryFactoryBean*. When using Spring Boot, if we do not specify a *JpaRepositoryFactoryBean* explicitly, the *JpaRepositoriesAutoConfiguration* kicks in and starts looking for JPA Repositories in the classpath. However, we can use the *JpaRepositoryFactoryBean* manually and register our JPA Repositories ourselves, as shown below:

@Configuration  
public class JpaRepoConfig {  
  
 @Bean  
 public JpaRepositoryFactoryBean<AuthorDao, Author, Integer> authorRepository() {  
 return new JpaRepositoryFactoryBean<>(AuthorDao.class);  
 }  
  
 @Bean  
 public JpaRepositoryFactoryBean<BookDao, Book, Integer> bookRepository() {  
 return new JpaRepositoryFactoryBean<>(BookDao.class);  
 }  
  
}

### Sources

The sources for the *static-jpa-repo-abstraction* can be found here: <https://github.com/paawak/spring-boot-demo/tree/master/dynamic-jpa/static-jpa-repo-abstraction>.

## Take the plunge

We will copy the code from the *static-jpa-repo-abstraction* and rename it to *dynamic-jpa-repo*.

We are finally ready! Lets delete the below classes as a start: 1. Book 1. BookDao 1. JpaRepoConfig

You should not get any compilation error, as these classes are not used by anyone directly.

We will create a helper class *DynamicClassGenerator* to create the Entity and the Dao class dynamically with *ByteBuddy*.

### Generating the Entity

The below code does the trick:

Unloaded<?> generatedClass = new ByteBuddy().subclass(BookTemplateImpl.class)  
 .annotateType(AnnotationDescription.Builder.ofType(Entity.class).build(),  
 AnnotationDescription.Builder.ofType(Table.class).define("name", "book").build())  
 .name(entityClassName).make();

As is evident, we start by extending the *BookTemplateImpl*. We then go on to add 2 class level annotation: 1. @Entity 1. @Table(name = “book”)

### Generating the Repository

This is how the Repository can be generated:

Generic crudRepo = Generic.Builder.parameterizedType(CrudRepository.class, entityClass, Integer.class).build();  
  
Unloaded<?> generatedClass = new ByteBuddy().makeInterface(crudRepo).implement(BookDaoTemplate.class)  
 .method(ElementMatchers.named("updateAuthor")).withoutCode()  
 .annotateMethod(AnnotationDescription.Builder.ofType(Transactional.class).build())  
 .annotateMethod(AnnotationDescription.Builder.ofType(Modifying.class).build())  
 .annotateMethod(AnnotationDescription.Builder.ofType(Query.class)  
 .define("value",  
 "update " + entityClass.getSimpleName()  
 + " set author.id = :authorId where id = :bookId")  
 .build())  
 .name(repositoryClassName).make();

Relax, don’t get intimidated by the above. It is actually quite simple. Let me list down the steps: 1. Create the fragment *CrudRepository<Book, Integer>* with the variable *CrudRepository<Book, Integer>* 1. Start with that, and extend from the *BookDaoTemplate* 1. Define the method *updateAuthor* 1. Define the 3 annotations on this method along with their parameters

### Saving the generated classes on disk

Now, this is really the tricky thing. Since Spring JPA needs these classes to be present as .class files on the classpath, it is absolutely necessary to save this on the disk.

Loaded<?> loadedClass = unloadedClass.load(getClass().getClassLoader(), ClassLoadingStrategy.Default.INJECTION);  
  
try {  
 loadedClass.saveIn(new File("target/classes"));  
} catch (IOException e) {  
 throw new RuntimeException(e);  
}

I am saving it in a directory, but you can also save it in a jar file.

### Tying it together

We define a class that implements the *BeanFactoryPostProcessor*.

@Configuration  
public class DynamicJpaBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
  
 @Override  
 public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {  
 }  
  
}

This is invoked after Spring Beans are registered. We would generate the classes in the *postProcessBeanFactory()*. Also, after our classes are created, we would then, as a last step, register our freshly minted JPA Repository as a Spring Bean using the *JpaRepositoryFactoryBean*.

private void registerJpaRepositoryFactoryBean(Class<?> jpaRepositoryClass,  
 DefaultListableBeanFactory defaultListableBeanFactory) {  
 String beanName = jpaRepositoryClass.getSimpleName();  
 BeanDefinitionBuilder beanDefinitionBuilder = BeanDefinitionBuilder  
 .rootBeanDefinition(JpaRepositoryFactoryBean.class).addConstructorArgValue(jpaRepositoryClass);  
 defaultListableBeanFactory.registerBeanDefinition(beanName, beanDefinitionBuilder.getBeanDefinition());  
}

This above code is basically just mimicking the below code dynamically:

@Bean  
public JpaRepositoryFactoryBean<BookDao, Book, Integer> bookRepository() {  
 return new JpaRepositoryFactoryBean<>(BookDao.class);  
}

Since the generic arguments are non mandatory, this works just fine with only the constructor argument.

### Verification

Run our Test Harness *RunCucumberTest* and ensure all tests pass.

### Sources

The sources for the *dynamic-jpa-repo* can be found here: <https://github.com/paawak/spring-boot-demo/tree/master/dynamic-jpa/dynamic-jpa-repo>.

# Conclusion

As we saw, it is certainly possible to generate an Entity and JPA Repository dynamically, it comes with its own set of challenges. The code is hard to test and maintain. So, use with care!