USING JAVA ANNOTATION PROCESSING TO ACCELERATE WEB API DEVELOPMENT - SPRING REST PROCESSOR

A THESIS SUBMITTED TO
THE FACULTY OF ARCHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

INDRIT BRETI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR BACHELOR DEGREE
IN SOFTWARE ENGINEERING

JUNE, 2023

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: ……..

Signature:

ABSTRACT

USING JAVA ANNOTATION PROCESSING TO ACCELERATE WEB API DEVELOPMENT - SPRING REST PROCESSOR

Breti, Indrit

B.Sc., Department of Computer Engineering

Supervisor: M.Sc. Igli Draçi

Annotations are a form of metadata for the code. Their popularity has mainly increased in the past few years despite being available since Java SE 5 (September 2004). They are extensively used by popular frameworks such as Hibernate and SpringBoot to offer commodities for developers. The ability to process annotations both in runtime and in compile time allows us to make use of the information they provide to generate anything we need, including Java code, documentation, configuration files, and more before compiling our actual code.

In this paper, we will see how to make use of annotations to accelerate REST API development by creating our custom annotation processors to generate fully sortable/filterable APIs based on Spring and Hibernate. "SPRING REST PROCESSOR" will be able to generate the Java code needed to define REST controllers and JPA repositories in Spring by simply analyzing the defined Entities. This annotation processor can be used to greatly speed up API development in SpringBoot while still allowing full customization capabilities to the developer.

Keywords: annotation, annotation processor, Java, generate, Spring, API;

Dedicated to my family
and everyone I met on this journey…

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Annotations are a standard metadata mechanism used to associate additional information to different elements like fields, methods, or types (classes) in Java. The ability to add metadata directly from the source code, without using an external configuration file makes annotations easy to use while still maintaining (somewhat) code readability. Annotations can be preserved in the binary representation of the Java code, allowing components and libraries to utilize the information contained by annotations at runtime.

Being introduced in J2SE 5.0 released in 2004 annotations are now gaining more and more popularity. It is worth noting that there are many uses for annotations, starting from providing information to the compiler (i.e.: using the predefined @Override annotation), adding extra logging and testing capabilities by utilizing the reflection API, generating documentation (i.e.: using @Documented for Javadoc), generating additional files such as XML, JSON data files, and most importantly generating source code.

Different tools, frameworks, and libraries such as the Hibernate object-relational mapping API, the Spring Framework, JUnit, and more use annotations to offer commodities for Java developers. The use of annotations by these frameworks reduces drastically the need to write repetitive code that can be auto-generated with the help of the metadata provided by annotations. To clarify the role of annotations in reducing the amount of code that developers need to write we need to understand what code can be generated with annotations.

Most of us are familiar with different techniques to reduce duplication of code, i.e.: making use of functions, inheritance, polymorphism, and different design patterns. However, those techniques can only help us with specific procedures that we need to run multiple times or specific object specifications that we can reuse. A function is simply the same set of instructions being executed with different parameters. What if we need to write code that is specifically tied to the fields of a class?

A great example is writing a builder for a type. The builder follows the same ideology for all classes that it is created for, no matter the implementation its duty is to expose chainable methods that serve as setters. However, the names of each method of the builder need to match the names of the fields that the class contains. We cannot create one universal builder, instead, we are forced to write a builder for each class. There is no way to achieve this with standard coding methodologies. The solution to this problem is code generation using annotations.

Let us take for example this class:

```
public class Article {
  private Long id;
  private String title;
  private List<String> tags;
}
```

The Article class contains the field id, title, and tags. To create a builder for this class we need to manually define the methods. However, using annotation processing we can automatically generate the code for the builder by simply analyzing the fields of this class. All we need to do is annotate the class with *@Builder* from Project Lombok. We will get into more detail on how this works later on.

While annotations possess numerous practical applications we should consider the downsides of this promising feature. The abstraction they provide combined with excessive use, can lead to what is known as annotation hell. As previously seen, we can simply annotate an element and let the library generate the code and handle all the complex details for us. This is great, until someone that is not familiar with the annotation we are using tries to understand the code. Annotations can be hard to embrace at first, however, even developers that are experienced with them can face difficulties understanding, maintaining, and navigating annotated source code. This also brings up another disadvantage of annotation-generated source code. The generated sources cannot be modified, leading to difficulties in maintaining the codebase and most importantly limitations on the capabilities of the software itself which becomes closely tied (coupled) to the annotations it is using. Since we are now well aware of not only the benefits, but also the limitations of annotations, let us have a more detailed look into their structure, declaration, and usage.

1.1. Annotations

Annotations are a special kind of Java construct used to decorate a class, method, field, parameter, variable, constructor, or package. Before annotations (J2SE 1.4 and earlier), we can see some other techniques for providing metadata, i.e.: using the transient keyword, the Serializable marker interface, or the old @deprecated comment for Javadoc. Annotations became a generalized approach to adding metadata. From an implementation perspective, annotations can be viewed as a distinct type of interface. To differentiate annotations from interfaces we use the @interface keyword. Annotations can make use of access modifiers just like interfaces. A sample code fragment that declares an annotation called RESTField: *public @interface RESTField {}*

1.1.1. Annotation Parameters

Parameters are elements within annotations used to hold different details about the field that is being annotated. They give us the ability to associate characteristics to the annotation which can then be retrieved when we resolve the annotation during compilation or runtime. You might be familiar with the "*@Order*" annotation from jUnit, which takes as a parameter the order index, i.e.: *@Order(1)*. It is important to note that parameters must be primitive types, String, Class, enum, annotation, or an array of these types. Their values may never be null. They are written as simple methods (no arguments, no throws clauses, etc) that are later used as getters to retrieve the values. Each parameter can have a default value which is declared using the default keyword. The code fragment below declares an annotation with the parameter apiName.

```
public @interface RESTField {
  String apiName() default "";
}
```

Annotation parameters can be passed only as named parameters i.e.: @Annotation(param1="test", param2=3.14) However, the specially reserved parameter name "value" can be omitted if it is the only parameter being passed. Sample of passing a parameter to an annotation:

```
public class AuditData {
  @RESTField(apiName = "updated_at_custom")
  Long updatedAt;
}
```

1.1.2. Marker Annotations

Are a special form of "empty" annotations defined with no parameters. Those annotations are used as flags, to identify elements that have a specific characteristic represented by this annotation. The only metadata those annotations provide is their presence, or absence based on which, different libraries and frameworks can enable

specific processing behaviors. The best example for this case is the @Deprecated annotation, which simply marks the element as deprecated without providing details.

1.1.3. Annotation Retention Policy

As discussed, annotations can be used both in compile time and runtime. However, there are cases where preserving this extra information in runtime or bytecode is redundant. Java allows us to control this by setting retention policies for each annotation that we declare. An annotation's retention policy can be set to one of the 3 available types: Source (available only during compile time), Class (not available in runtime, preserved in the Java bytecode), and Runtime (available in runtime, present in the bytecode, and processed during compilation). The retention policy of an annotation defaults to Class.

To set the retention policy of an annotation we can annotate it with *"@Retention(RetentionPolicy)"* as shown in the example below:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface IgnoreRESTField {}
```

1.1.4. Annotation Target

By default any declared annotation can be used on any supported Java element type. However, this might cause uncertainty and ambiguity for the developers. We might want to declare an annotation that can be used only on methods and constructors, to do so, simply annotate the annotation definition with *"@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})"*. This way we can restrict the use of the annotation only on elements of type *ElementType*.

1.2. Runtime Reflection API

We can make use of annotations at runtime by utilizing the Java Reflection API. Reflection allows us to retrieve details and metadata for any element at runtime. Additionally, it allows us to modify its behavior and perform operations that would otherwise be impossible. Java introduced basic reflection in J2SE 1.2, and support for annotations was added in J2SE 5.0. The main method used to analyze annotations in runtime is the "*getAnnotations*()" method which returns all the annotations attached to the given element. While Reflection is powerful it must be used carefully since it bypasses many security checks, and can lead to performance degradation. Since this paper focuses mainly on compile-time annotation processing we will not expand more on this topic.

1.3. Compile-Time Annotation Processing

Source-level annotation processing was introduced in Java 5. Annotation processing allows the generation of additional source files during compilation, this includes documentation, data persistence files, java source code, and any other type of file based on annotation usages in the source code. The processing API can only be used to generate files, modifying existing sources is not possible. While runtime annotation handling can cause performance degradation, compile time processing might only slow down compilation, the runtime performance will not be affected by the processor. We will get into full detail on the usage and limitations of annotation processing in Chapter 3 as we discuss its use to generate powerful REST APIs.

CHAPTER 2

LITERATURE REVIEW

Annotation processing is used by many existing projects and frameworks as a way to increase productivity, reduce the amount of code that developers need to write, and generally speed up software development. Some of the most important and popular frameworks are Hibernate, Spring Boot, and Jackson.

2.1. Hibernate

Hibernate is a popular object-relational mapping (ORM) framework for Java. The Hibernate Annotation Processor is responsible for processing the annotations defined in Java classes and generating necessary metadata at compile time. The annotation processor scans the annotated classes, extracts information from the annotations, and generates XML mapping files and Java source code used by Hibernate itself.

The annotations provide enough information to the Hibernate processor for it to be able to define mappings between Java classes and database tables. What would be configured by the use of a separate XML configuration file is now defined through annotations making it easier to write, understand and maintain.

2.2. Spring Boot

In Spring Boot, annotation processing plays a crucial role in enabling various features and functionalities. Spring Boot leverages annotation processors to automatically configure and initialize beans, handle request mappings, manage transactions, and perform other tasks.

Spring Boot uses the Spring Framework, which includes its own set of annotation processors. These processors analyze the annotated classes and generate the necessary configurations or perform specific actions during the application's startup or runtime. Spring Boot uses multiple annotation processors, two of the main ones are **ComponentScan**, which registers beans from Spring components using @Component, @Service, @Repository, and **RequestMappingHandlerMapping,** responsible to generate code that maps HTTP requests to the corresponding methods of the controller marked with @Controller or @RestController.

2.3. FasterXML - Jackson

FasterXML Jackson is a high-performance JSON processing library for Java. It provides functionalities for reading and writing JSON data, converting JSON to Java objects (deserialization), and converting Java objects to JSON (serialization).

"Jackson Annotation Processor" (also known as "Jackson-module-jsonSchema") is a great example of using annotation processing to generate JSON schema definitions from annotated Java classes. It scans the classes annotated with specific Jackson annotations, such as *@JsonSchema, @JsonProperty, @JsonFormat*, etc., and generates JSON schema definitions based on the annotated properties and their configurations.

The generated JSON schema provides a structured representation of the data model defined by the annotated Java classes. It describes the expected structure, types, and constraints of the JSON data that can be serialized or deserialized using Jackson.

The Jackson Annotation Processor is a useful tool for documenting and validating the JSON data exchanged in applications. It helps ensure that the JSON data conforms to the expected structure and provides additional metadata for serialization and deserialization.

The use of the processor shows great improvement in both development and performance speed. Defining JSON schemas is really easy through annotations, and since the process occurs during compilation it does not affect the performance.

2.4. Project Lombok

Is one of the most popular libraries that offers helpful annotations that can generate builders, getters, setters, constructors, and more by simply analyzing the fields of the class and the annotations that the developer sets. It is important to note that Project Lombok is a unique way of using annotations since it uses the internal Javac API to modify existing source code by casting Elemtens to AST nodes. Normal annotation processors will simply create a new Java class since modifying existing sources is not directly supported by the Java compiler.

To create a builder for a class we can simply annotate it with *@Builder*. In this case, the annotation is used as a simple marker telling the Lombok processor that we need to write the code for a builder for this class.

2.5. Review Conclusions

While working with Spring Boot I noticed that there is more room for improvement on API definitions. Spring makes use of Hibernate as an ORM solution, but there does not exist a solution that automates the mapping of objects to API endpoints that allow full sort and filter capabilities for the entity. Let me clarify the need for such a solution. In order to retrieve all records of an entity in Spring, the developer needs to define the controller, the parameters needed, and the repository method/JPA query. However, by looking into how the up-mentioned frameworks/libraries leverage annotation processing I got inspired to build an annotation processor that can analyze the fields of an entity and build the respective REST controller that allows filtering, sorting, and paginating the results based on the fields that the entity contains, including joined entities without having to write extensive code.

CHAPTER 3

MATERIALS AND METHODS

3.1. Materials

Java Annotation Processing and Creating a Builder (Baeldung, https://www.baeldung.com/java-annotation-processing-builder lastly visited on 22 June 2023)

https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html (lastly visited on 22 June 2023)

https://www.baeldung.com/hibernate-criteria-queries (lastly visited on 22 June 2023)

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ (lastly visited on 22 June 2023)

3.2. Methods

The first step in creating this framework was building a dynamic query builder based on custom filters and sorting definitions. The next step was generating REST controllers that automatically bind filters and sorting options to the fields of the entity that it exposes. This can be achieved with the use of an annotation processor that analyzes the entities and generates the source code.

Working with annotation processors is quite difficult as you are working with uncompiled Java source code that you cannot use the same way you would with the Reflection API. As the project got more complex I wrote a few utility methods that assist in working with the Java Mirror API mentioned in section 3.2.11.

3.2.1. Designing JPA repositories to support dynamic queries

This framework intends to simplify the process of building powerful REST controllers that allow filtering and sorting by all the supported fields. To achieve this we will rely on JPA and Hibernate. The framework should build the JPA query based on the API parameters that it exposes, and the parameters that it exposes are based on the fields the Entity contains.

Each entity in spring boot is linked to a repository. The repository is simply an interface on which we declare methods that get translated to JPA queries. The developer can use methods of this repository to execute queries on the database, without having to actually write the query. Our query builder will expose (for now) 4 main methods into an interface called *DynamicQueryRepository*.

```
public interface DynamicQueryRepository<T> {
   Stream<T> findAllByCriteriaAsStream(CriteriaParameters cp);
  Stream<T> findAllByCriteriaAsStream(int page, int size, MultiColumnSort sortBy,
                                      List<Filter<?>> filters, Hashtable<String,
                                     FunctionArg[]>  sortByFunctionArgs);
   Page<T> findAllByCriteria(CriteriaParameters cp);
   Page<T> findAllByCriteria(int page, int size, MultiColumnSort sortBy,
                             List<Filter<?>> filters, Hashtable<String,
                            FunctionArg[]> sortByFunctionArgs);
}
```

The methods are similar to each other, however, *findAllByCriteria()* returns a page of results with the requested limit and offset while *findAllByCriteriaAsStream()* returns a stream allowing further processing to be done by utilizing the Java Stream API. The methods are overloaded, the simplified version takes only *CriteriaParameters* as a parameter, making it easier to transfer all the filters, sort, page size, and offset details. The need for *CriteriaParameters* is explained in section 3.2.14

To use those methods, each repository will have to extend from an interface that uses this naming convention *<entity name>DynamicQueryRepository,* the interface itself must extend a concrete implementation *DynamicQueryRepository*. For example, the demo entity used in the demo of the framework is called “Product”. We must declare the ProductRepository as follows:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long>,
ProductDynamicQueryRepository {}
```

The implementation of *ProductDynamicQueryRepository* follows the same naming convention, with Impl appended, *ProductDynamicQueryRepositoryImpl*. The duty of each implementation is to simply call the same methods of the interface in the shared, already implemented methods of *DynamicQueryRepositoryUtils,* with the addition of providing the class of the entity, the EntityManager, and field details for the entity (a hash table of all the fields that the entity contains, those are resolved by our annotation processor, see section 3.2.11). An example implementation for *ProductDynamicQueryRepositoryImpl* is

```
@Override
public Page<Product> findAllByCriteria(CriteriaParameters cp) {
  return DynamicQueryRepositoryUtils.findAllByCriteria(Product.class, em,
  org.indritbreti.restprocessor.FieldDetailsRegistry.instance().lookup(Product.class), cp);
}
```

As you can see, we are forced to write the code for *<entity name>DynamicQueryRepository* and *<entity name>DynamicQueryRepositoryImp* for all the repositories that we intend to use the dynamic query builder on. Our annotation processor will be able to do this for us, we will simply extend from *<entity name>DynamicQueryRepository* without having to worry about the implementation (see section 3.2.13 for more details on how the code for each dynamic repository is generated).

3.2.2. Building queries in JPA

As seen in the previous section, all implementations of *DynamicQueryRepository* will use *DynamicQueryRepositoryUtils* to build and execute the query. If you recall, each repository implementation will pass its *jakarta.persistence.EntityManager* and its entity class to any method of *DynamicQueryRepositoryUtils* that it calls.

The entity manager and the specified entity class will be used to create the CriteriaBuilder and CriteriaQuery as shown below:

```
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery criteriaQuery = criteriaBuilder.createQuery(entityClass);
```

From this point, we can get the JPA Root by using:

```
Root<ENTITY_TYPE> root = criteriaQuery.from(entityClass);
```

All that is left to do is build the WHERE and ORDER BY clauses. The duty of building JPA expressions is delegated to other methods explained in sections 3.2.3 and 3.2.4.

```
criteriaQuery.where(PredicateBuilder.predicatesFromFilters(filters, criteriaBuilder,
                    root));
criteriaQuery.orderBy(DynamicQueryBuilderUtils.getOrdersFromSortDetails(
                        criteriaBuilder, root, sortBy.getSortDetails(),
                        sortableFieldDetails, sortByFunctionArgs));
```

There is a special case that must be handled carefully. When building paged responses we need to be aware of the total elements count. To do so we can use the same where clause, remove the order by since it is irrelevant to COUNT(), and create the query with return type Long *criteriaBuilder.createQuery(Long.class);*. To allow the reuse of the same *where clause* without writing duplicate code, the method *DynamicQueryRepositoryUtils#buildCriteriaQuery()* uses a parameter *boolean isCountQuery* to control if we need to get the records or only the count.

3.2.3. Predicates - Filters

The criteriaQuery.where() method takes as parameter a list of Predicates. The problem with jakarta.persistence.criteria.Predicate is that it is built using the CriteriaBuilder, which is resolved by the entity manager. Furthermore, the criteria builder makes use of the JPA Root which is resolved by CriteriaQuery. In other words, building predicates directly from request parameters would be a difficult, hard-to-maintain task since we need to have access to the repository. Instead, each request filterable parameter will first be mapped to a Filter<T>, which will then be used to build the predicate by using *PredicateBuilder.predicatesFromFilters(filters, criteriaBuilder, root)*.

In order to be able to explain how the Filter class works, I will have to first explain how we will resolve filter values from request parameters. Currently, the generated controllers can resolve filters only from request parameters. Filters can have different operators, i.e.: equals, not equals, in, not in, greater than, etc. Those are defined in the enum *CriteriaOperator* (Figure 1).

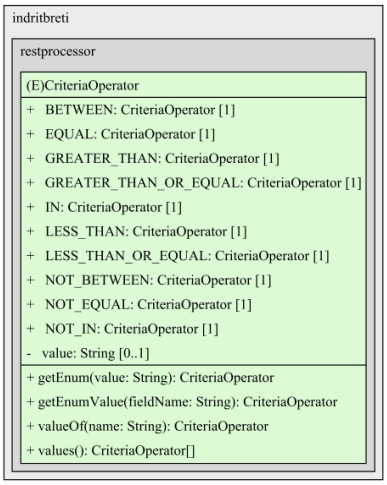

Figure 1 Class Diagram of *CriteriaOperator* (Enum)

To be able to parse them from request parameters I created the class *RHSColonExpression*. This class takes as input a string and parses it by following the

RHS Colon (Right Hand Side Colon) REST standard. This standard parses the values by following this format: <parameter>=<operator>:<value1>;<value2>. Some examples to clarify the way the values can be defined: id=gt:100, id=in:1;3;4, id=btn:10;30. Note that something like id=eq:10 can be simplified to id=10 by removing the redundant 'eq' operator. The operator and the values are separated by a colon which can be escaped with a backslash i.e.: name=string_that_contains\:_colon. Multiple values are separated by a semicolon since the comma is used as a URL parameter delimiter. Similarly to the colon, the semicolon can be escaped by a backslash i.e.: name=strign_that_contains_\;semicolon.

Since filters may contain 1, 2, or many values we need to keep track of the number of values that are provided. The enum *CriteriaOperatorValuesType* defines the values type as SingleValue, Range, and MultiValue (Figure 2). This allows us to check that the operator supports the number of values provided. For example, we can **not** use a 'btn' operator with only 1 value since it requires a range. This will throw the exception *UnsupportedCriteriaOperatorException*

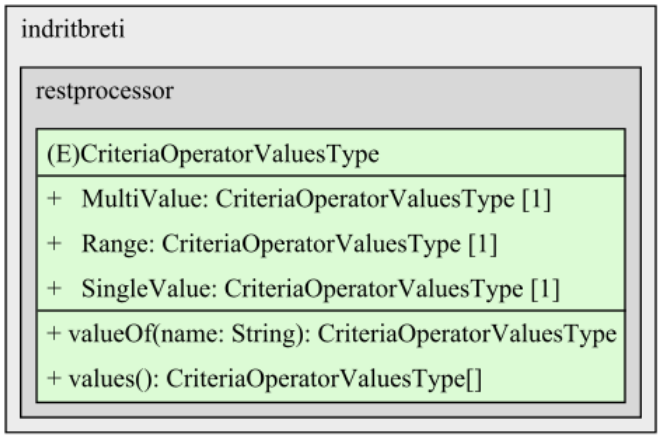

Figure 2 Class Diagram of *CriteriaOperatorValuesType* (Enum)

After parsing the string, *RHSColonExpression* gives us access to the *CriteriaOperator*, the *CriteriaOperatorValuesType,* and a list of raw values. *RHSColonExpression* is used in request parameters as follows:

@RequestParam(name = "used", required = false) List<RHSColonExpression> usedFilters

Note that it is defined as a *List<>* since Spring gives us the ability to parse multiple request parameters which can be translated to multiple filters, i.e.: /products?id=gte:2&id=neq:3. Which will retrieve all elements with id>2 and id=3.

Since we are now aware of how filter values are resolved we can get more deep into explaining how *Filter<R>* is designed. The filter class has 4 main fields. First is the *CriteriaOperator* which holds the criteria operator resolved from *RHSColonExpression*. Additionally, the filter holds the *leftExpressionPaths* which represent the JPA paths (i.e.: product.id, product.categories.id) or any left-hand side of an expression i.e. the name of an SQL function. To complete the predicate, we hold the right expression as a value of type <R> based on the filter's generic. In addition to those core properties, the filters use a HashSet named supportedOperators which holds a set of *CriteriaOperator* to mark the supported operators by the filter.

Different filter classes can be created to fulfill our needs by extending the base class Filter<R> (Figure 3). For example, the RangeFilter<R extends Comparable> is used to store the details for a range operator, it makes sure that the type is comparable since we need to make use of greater than, less than operators.

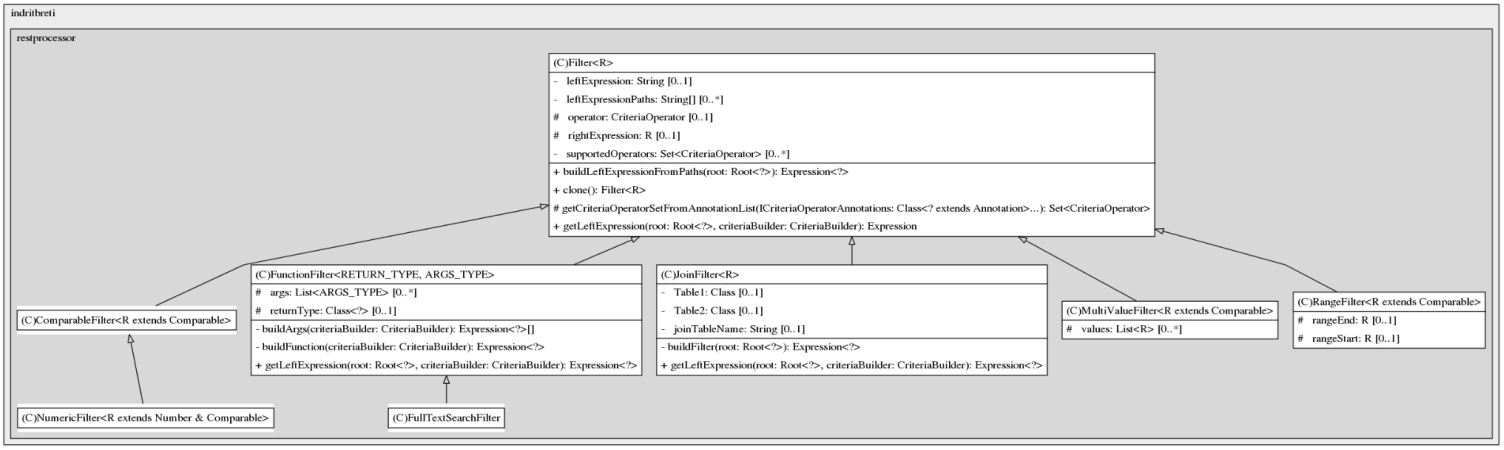

Figure 3 Class Diagram of *Filter* and classes extending from it

Let us now look at an example of using the filters. The *ProductController#demoFilters()* method shows an example of binding a filter on the id field and hard-coding a filter to exclude all products with id=2. First we add the parameter to the controllers method: *@RequestParam(name = "id", required = false) List<RHSColonExpression> idFilters)*

Now we can build the filter from the RHSColonExpression using the FilterFactory:

```
filters.addAll(FilterFactory.getNumericFiltersFromRHSColonExpression(
                Long.class, "id", idFilters));
```

To add a filter that excludes all products with id=2 we initialize the filter like this:

```
filters.add(new NumericFilter<Long>("id", CriteriaOperator.NOT_EQUAL, 2L));
```

Finally, we use the dynamic repository to get the paged results.

```
Page<Product> resultsPage = productRepository.findAllByCriteria(0, 30, null,
                                filters, null);
```

As you can see, this is not really convenient. While it is better than writing methods or queries manually we are still forced to write request parameters and build filters

manually for each field that we want to filter by. Additionally, we might end up making mistakes while building filters since the left expression is based on the name of the field. Changing the name of the field of an entity would lead to an error if we do not change the name we are using to build the left expression on the filter. Here we can clearly see the need of using an annotation processor that automatically builds the REST controller for us by defining all the request parameters and building the filters without any input from us. It can directly analyze the entity declare @RequestParam for each field, build the filter based on the type and name of the field and call the dynamic repository. This is covered in section 3.2.14.

3.2.4. Order/Sorting

In addition to filtering, the endpoints should support sorting. To parse sorting details from the request parameter the framework uses only 1 parameter which by default is named “sortBy”. This parameter is of type *MultiColumnSort*, which is a simple class that takes a string or a list of strings as parameters. The strings represent sorting expressions such as columns or function names i.e.: sortBy=price. Multiple sort expressions can be provided by delimiting them with a semicolon. For example, sortBy=price;id. will use id to sort the results in case the price is equal. The semicolon can be escaped using a backslash. In addition, we can provide multiple values by specifying sortBy multiple times as a request param, i.e.: */api/products?sortBy=id&sortBy=price*. By default the values are sorted in ascending order, however, different fields can have different default sort order. The order can be specified by providing a ‘+’ (ascending) or ‘-’ (descending) symbol before the expression i.e.: sortBy=-price;+id

In theory, providing a sorting expression should be pretty simple. The use provides it as sortBy=expression. However, this has 2 major problems. Firstly the user cannot know the list of available expressions, let it be fields, function names, etc. Secondly, the user

can exploit this by providing field names for fields that we might want to disable sorting for, i.e.: sortBy=password. To resolve this we should provide a list of *IRestFieldDetails* to the methods of *DynamicQueryRepositoryUtils*.

IRestFieldDetails is a simple interface that exposes the API name for the field, the default sort order, if the field is sortable, and if the field is filterable (Figure 4). Normally this information will have to be built manually for each field. Here we face the same issue as with filters, where we have to provide already existing information manually. To resolve this, we can use annotations that provide this metadata for us. The metadata can then be used to build an implementation of *IRestFieldDetails*. Implementations of IRestFieldDetails hold the necessary information needed to build List<Order> from *DynamicQueryBuilderUtils#getOrdersFromSortDetails()*

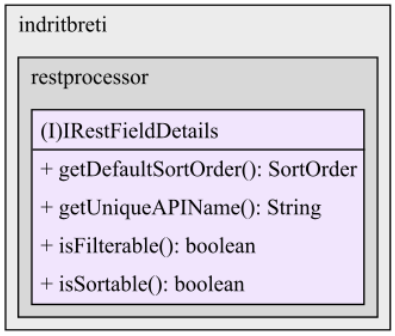

Figure 4 Class Diagram of *IRestFieldDetails* (Interface)

In addition to sorting by fields, we might need to sort by a function, i.e.: sort by a ranking function or sort by the length of the 'description' column. To do so, the developer can bind a custom field '*SortByFunction*' (Figure 5) to the *FieldDetailsRegistry* of an entity (more details on the *FieldDetailsRegistry* can be found in sections 3.2.11 and 3.2.15).

Sample code:

```
FieldDetailsRegistry.instance().bindField(Product.class,
new SortByFunction<Float>("custom_ts_rank", Float.class, "searchBestMatch", 1,
SortOrder.DESC));
```

FieldDetailsRegistry.instance().bindField(Product.class,
new SortByFunction<Float>("length", Long.class, "descriptionLength", 1,
SortOrder.ASC, new PathFunctionArg(0, "description")));

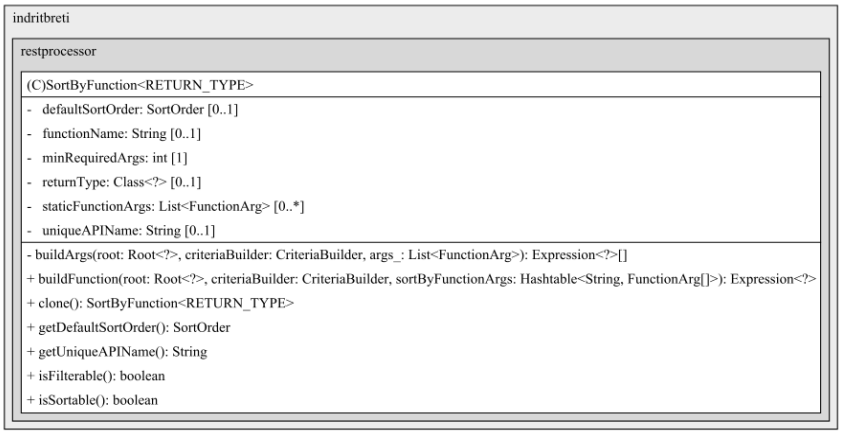

Figure 5 Class Diagram of *SortByFunction*

3.2.5. Setting up the project to use annotation processing

Annotation processors are used during the compilation phase of the source code that we are trying to compile. That means that the source code of the annotation processor should already be compiled if we need to use it in the compilation process for the main source code. After we finish writing the code for the annotation processor we can simply package the jar and include it in the project, however, this is not practical while developing the processor itself since we need to test and debug things. In that case, we can set up a multi-module maven project which contains the processor module, and the main module which has a dependency on the processor module. This way when running the project, the processor module is compiled before the main module, which then will be able to use the processor during compilation.

To be able to reference the generated source files we can use the *'maven-compiler-plugin*' to add the generated sources directory. Additionally, we will utilize the '*build-helper-maven-plugin*' to set a different output directory for our generated sources. SpringRestProcessor will use this path by default *${project.build.directory}/restprocessor-generated-sources/*. This will also be covered in Chapter 4.

3.2.6. Defining an annotation processor

The processor is a simple class that extends and implements the methods of *javax.annotation.processing.AbstractProcessor*. In this project, I named it *RestProcessor*. The method *getSupportedAnnotationTypes()* should return a set of strings that represent the full reference of each annotation that the annotation processor is expected to process. This set is used by the Java compiler when it calls the processor to make sure that it only resolves and passes a set of these annotations to the processor. The method *getSupportedSourceVersion()* should return the latest source version supported by this annotation processor. The method *process()* takes as parameters a set of TypeElement which is a subset of *getSupportedAnnotationTypes()* that contains the annotations that the compiler was able to resolve in the source files, and the RoundEnvironment which contains the metadata and all annotations for all classes. The processor should return true if it is done processing the annotations that it was given, and false otherwise, i.e.: if something went wrong.

It is worth noting that the processor that we are defining must be triggered even if none of our main annotations are used. However, the Java compiler will only trigger the processor if it is able to find in the source code any of the annotations returned from *getSupportedAnnotationTypes()*. To overcome this we can create a marker annotation

@EnableRestProcessor that we can add in our main class (or anywhere else) to simply enable the processor.

3.2.7. Registering the processor

An annotation processor can be specified in the javac command by using the -processor argument followed by the full reference of the processor, in this case, *org.indritbreti.restprocessor.RestProcessor*. However, this is not practical, especially when compiling within an IDE or while using Maven. A better solution would be using the maven-compiler-plugin and specifying the custom annotation processor, however, this still requires extra configurations and might cause issues when integrating with other processors. The best way to register an annotation processor is by building the JAR and adding a reference to the processor's fully qualified class name into the file *META-INF/services/javax.annotation.processing.Processor*. However the file needs to be added after the jar is created since otherwise we will be requesting the use of our annotation processor while compiling itself, which is not possible. A simple way to achieve this is utilizing Google's auto-service library which can automatically register our processors as a service by simply annotating the processor with *@AutoService({Processor.class, AbstractProcessor.class})*

3.2.8. Processing rounds

Annotation processing is done in multiple rounds. This behavior cannot be controlled per processor since this is how the Java compiler works. One round is enough to process all the annotations of the source code, however, if our processor generates new source code that contains annotations that need to be processed by the same annotation processor we will need to run another round. The rounds are completed the moment that no more source files are generated by the processor. Our annotation processor must run

only once to avoid generating duplicate persistence data (mentioned in section 3.2.15). Since it does not produce source files that contain annotations that we need to process we skip all the other rounds by adding this condition at the beginning of our processor *if (annotations.size() == 0) return true;*

3.2.9. Debugging the processor

While developing the processor we might find the need to debug the code of the processor itself. There are many ways to achieve this based on the way you are compiling the project. If you are using IntelliJ you can press Ctrl+Shift+A, search for debug build process and enable it, additionally, you will need to add this custom VM option -Dcompiler.process.debug.port=8000. At this point, you will need to simply create a new Remote JVM Debug configuration that listens to the specified port. The same thing can be done in Maven by using the command mvndebug instead of mvn.

3.2.10. Annotations needed for SpringRestProcessor

At this point we have a well-configured annotation processor and debug environment. It is time to get to the main part of annotation processing which is resolving the metadata needed to generate the sources that we intend to use. We can rely on existing annotations used by Spring Boot to resolve most of the details, however, we will need a few custom annotations in order to enable specific functionalities and allow the developer to customize behaviors.

As a recall for our goal, we need to build fully filterable, sortable, and pageable APIs based on the fields an Entity contains. To achieve this, we need to resolve all entities declared in a Spring Boot project, resolve their fields, look for methods annotated with

@DynamicRestMapping build the corresponding REST controller that exposes the fields as API parameters and use them to build a JPA CriteriaQuery that allows sorting and filtering based on those fields, get the results and return them to the original method annotated with @DynamicRestMapping.

By default, the processor will expose all the fields of an entity as sortable/filterable fields based on their name and parent's name. However, in addition to simply resolving the fields the developer might want to use a custom name to expose the field through the API, mark it as a required field, set a different default sort order, or even exclude some of them from sorting and/or filtering capabilities. To achieve this I created the @RESTField annotation. The definition of this annotation is given in Figure 6.

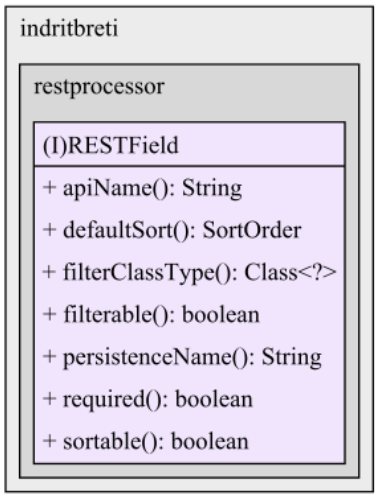

Figure 6 Class Diagram of *@RESTField* annotation

In addition to @RESTField the marker annotation @IgnoreRESTField can be used to mark the field as not sortable and not filterable. This annotation takes priority and totally skips the field details resolving process.

To declare the dynamic endpoints I created the @DynamicRestMapping annotation, which can be used only on methods. This annotation will allow us to generate a

@RequestMapping annotated method that takes as API parameters the sorting/filtering details and returns back to the original method the requested parameters and an instance of *CriteriaParameters* which contains all the filters and sort details. This annotation takes as parameters the path/s that will be mapped, the request method, and the entity that it is going to expose. More details on Figure 7.

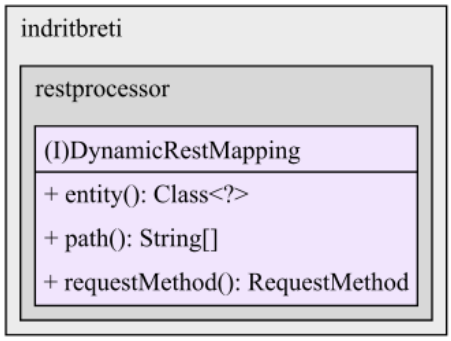

Figure 7 Class Diagram of *@DynamicRestMapping* annotation

Finally, as mentioned in section 3.2.6. we have already declared another marker annotation named @EnableRestProcessor to simply enable our annotation even in cases when none of the annotations above is used. This will make sure that the processor runs and all dynamic query repositories are generated so we can implement them in the original repositories before even declaring a @DynamicRestMapping.

3.2.11. Annotation processing - Resolving field details

Using the annotation processor we configured and the annotations that we defined earlier we are able to generate the needed source code as mentioned in sections 3.2.2, 3.2.3, and 3.2.4. However, as discussed, we need to be aware of the fields of an entity in order to be able to define filters and sort expressions. To resolve the fields of each entity

we will need to iterate through all the elements that are annotated with *@jakarta.persistence.Entity*, note this is a standard JPA annotation. To do so we need to get all the annotated elements using this method of *RoundEnvironment*:

roundEnv.getElementsAnnotatedWith(Entity.class)

This method will return a Set of <*Element*> however we need to cast them to *TypeElement* (a subclass of *Element*) since *@Entity* is used to annotate classes (Types). To do so we can use the utility method defined by me in *TypeMirrorUtils.* Most methods of *TypeMirrorUtils* are wrappers for *processingEnvironment.getTypeUtils()* that additionally handle edge cases such as casting primitive types to boxed classes.

TypeElement entityTypeElement = TypeMirrorUtils.getTypeElement(entityClass, processingEnv);

Once we resolve the entityTypeElement we can use the following method to resolve the fields for the entity: *FieldResolverUtils.getFields(entityClass, processingEnv, roundEnv).* The duty of this method is to build *FieldDetails* (Figure 8) for each field that the entity class contains. *FieldDetails* implements *IRestFieldDetails* (Figure 4) and is used to keep track of all the details that we will need to build filters and sorting options for each field, such as the default sort order, the API name, the JPA path, booleans if the field is sortable and/or filterable, the type of the field (Integer, Boolean) and more.

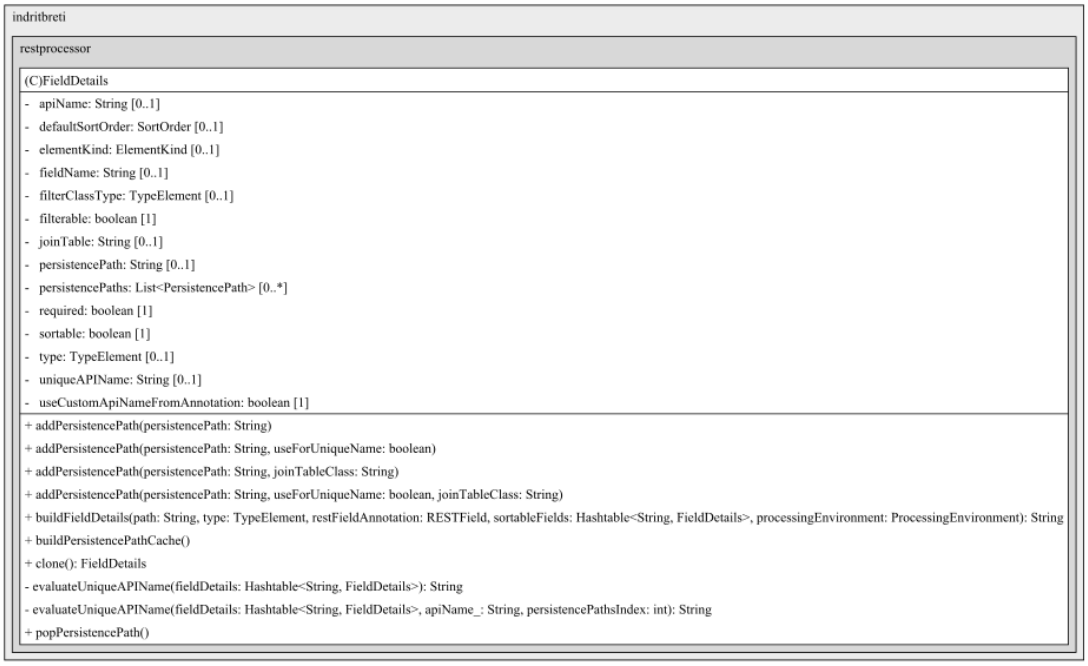

Figure 8 Class Diagram of *FieldDetails*

To build *FieldDetails* for each field, *FieldResolverUtils* follows this logic: It iterates through all the elements of the class, if the element is not of kind field (i.e.: a method), is static or final, or is marked with *@IgnoreRESTField*, the field is skipped. Otherwise, if the field is marked with *@RESTField* we use the parameters of this annotation to build the field details such as the API name, the default sort order, and more (refer to section 3.2.10 Figure 6 for more). However even if the field is not annotated with *@RESTField* we can build the details if it is a serializable field that can be mapped to a JDBC type i.e.: String, Number, Character, Enum.

If the field does not fulfill any of these conditions, the field might be a complex type that cannot be serialized. In that case, it is added to *queuedNestedRestFields*. Those fields will be iterated at the end. If they are marked with *@jakarta.persistence.Embedded* they are considered embedded fields, those types of fields are expanded as normal columns

by JPA. We can resolve the details for embedded fields by calling *getFields(nestedClass, fields, restFieldDetailsPersist, processingEnvironment, roundEnvironment);*

At the end of iterations, we check if the original TypeElement (class) extends from a parent class. If the superclass name does not match “Object.class” we need to recursively check the fields of the parent class. This covers things like *class Product extends ProductBase* where we need to resolve the fields of the parent class.

After resolving the fields for the given entity we need to store them somewhere to be used to generate the filters and check if sorting expressions are valid. To store the fields I created *ProcessorFieldDetailsRegistry,* a simple singleton class that acts as a registry on which we can bind a Class to its FieldDetails. The FieldDetails for each Class can be looked up using the class's full name. Figure 9 shows the structure of this class.

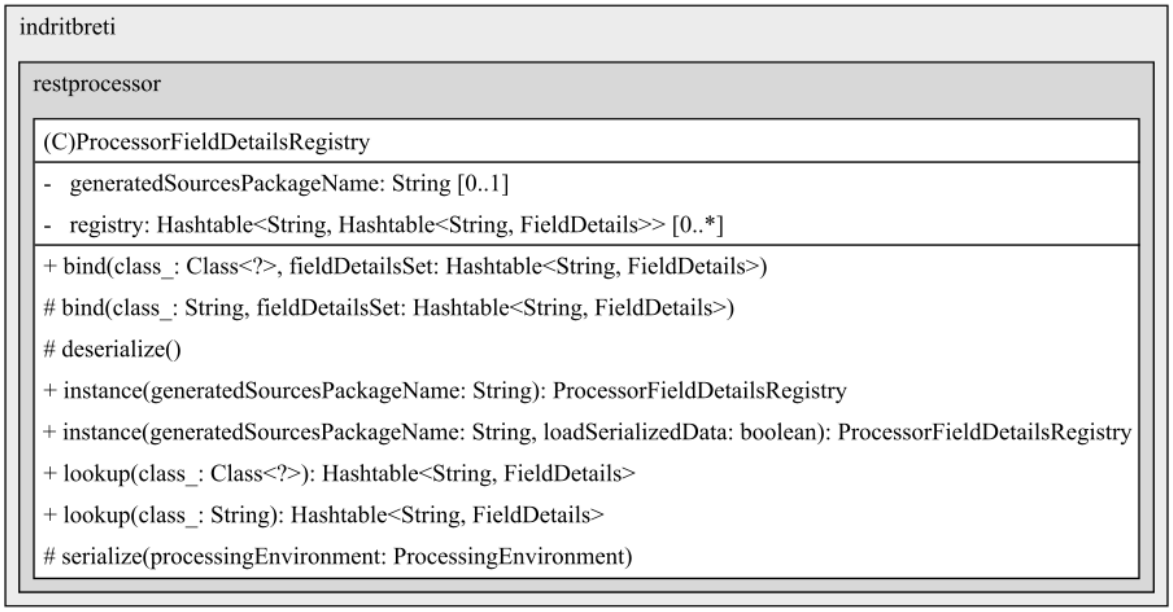

Figure 9 Class Diagram of *ProcessorFieldDetailsRegistry*

In addition to lookup and bind methods, this class contains a serialize method. We resolved and stored FieldDetails for each entity in compile time, but we cannot access them in compile time. To solve this issue we can serialize the data, more details in section 3.2.15.

3.2.12. Annotation processing - Generating source code

The *ClassBuilder* (Figure 10) is a simple class that I created to make writing Java classes from an annotation processor easier. A more powerful open-source version of it is JavaPoet. However, JavaPoet is an overkill for our needs. ClassBuilder on the other hand contains only a few methods that allow us to: define the filename, package name, a list of imports, and the main body code. The source file writer is created using the processing environment filer:

processingEnvironment.getFiler().createSourceFile(generatedSourcesPackageName+"."+fileName);

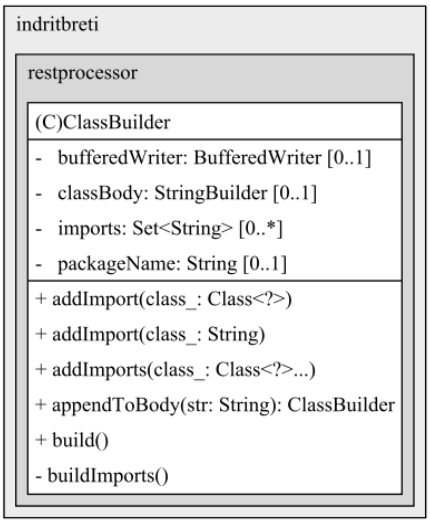

Figure 10 Class Diagram of *ClassBuilder*

3.2.13. Annotation processing - Generating Repositories

Section 3.2.1 states clearly how each *DynamicQueryRepository* contains code that is specific to each repository since they use different entities. This forces us to write a *DynamicQueryRepository* and *DynamicQueryRepositoryImpl* for each entity. However, this code can easily be auto-generated by knowing the class of the entity that we need to generate the repository for.

To generate the dynamic repositories for all entities we need to iterate through all the elements that are annotated with *@jakarta.persistence.Entity*, the same way as we did in section 3.2.11 when we resolved *FieldDetails*. On each iteration we call *buildDynamicQueryRepository(entityTypeElement);* This method is pretty simple, it uses *ClassBuilder* (section 3.2.12) to create a Java source file and write the code needed for the repository. The code that we need to write for each *DynamicQueryRepositoryImpl* is explained in section 3.2.1. At this point, we simply write it as a string, and replace parts of the code with elements that are specific to the entity we are generating the repository for, i.e.: the name of the repository, the class reference of the entity, etc.

3.2.14. Annotation processing - Generating dynamic REST controllers

Up to this point we have the *FieldDetails* and the dynamic repositories for each entity. We can now generate the source code needed to define the dynamic endpoints. To define a dynamic mapping the user must define **a method that takes as the first parameter a field of type *CriteriaParameters*** (mentioned in section 3.2.1). This method will be called by the dynamic controller which will put all the resolved filters and sort details into *CriteriaParameters*. The user can then modify *CriteriaParameters* by adding new

filters, removing filters, changing the sort, the page size, and more by utilizing the methods that *CriteriaParameters* exposes (Figure 11)

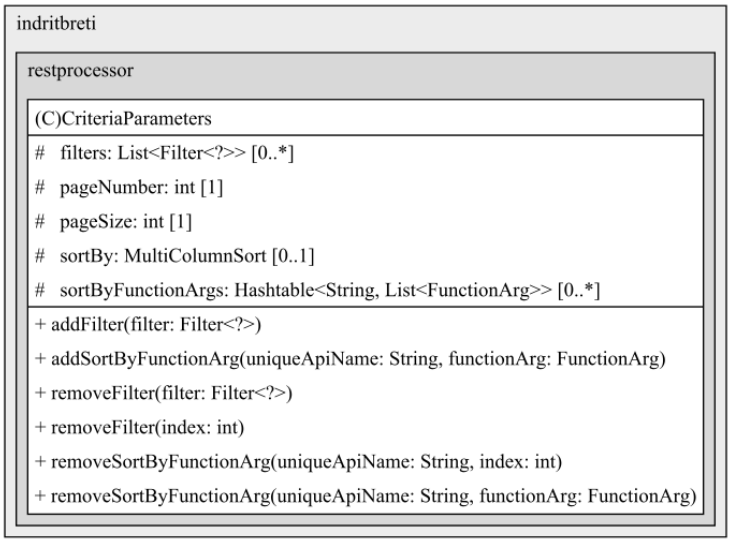

Figure 11 Class Diagram of *CriteriaParameters*

Additionally, the user needs to mark the method with *@DynamicRestMapping.* When doing so, all the parameters of *DynamicRestMapping* must be specified, including the API endpoint that will be mapped, the HTTP request method that this endpoint supports, and finally the entity that we want to expose through this endpoint.

A sample definition of a DynamicRestMapping:

```
@DynamicRestMapping(path = "/api/products", requestMethod = RequestMethod.GET,
entity = Product.class)
public ResponseEntity<PageResponse<GetModerateProductDTO>>
      getAllProducts(CriteriaParameters cp) {
      Page<Product> resultsPage = productService.getAllByCriteria(cp);
      return ResponseFactory.buildPageResponse(resultsPage,
                                                GetModerateProductDTO::new);
}
```

While the user can specify other parameters to this method, CriteriaParameters must remain first. The dynamic controller will be able to generate the code needed to retrieve all the other requested parameters i.e.:

```
getAllProducts(CriteriaParameters cp, @RequestHeader(value =
HttpHeaders.AUTHORIZATION, required = false) String authorizationHeader,
@RequestParam(name = "searchQuery", required = false) String searchQuery)
```

To generate the REST controllers, the annotation processor iterates through all methods that are annotated with *@DynamicRestMapping* and puts them in a hash map that will group all mappings that are part of the same controller together. After doing so, we can iterate all controllers and use *ClassBuilder* to generate the needed source code. We start by resolving the package name of the class on which *@DynamicRestMapping* is used to annotate the method, and append ".generated" to it. This way we can define a new controller "ProductController" without clashing with the definition of the original controller. After that, we add all the needed imports and start defining the controller.

When defining the new controller we need to make sure that we persist the annotations that the original controller was using, except for *"@Component"*. For example, if the original controller had an annotation *@RequestMapping(path = "api/products")* we need to add the same annotation to our generated controller. After doing so we set a random name for the bean that this controller represents by using *@Component*. This is done to avoid conflicts with beans defined by the user that use the same name.

The original annotations are resolved using this line of code:

```
classBuilder.appendToBody(dynamicRestControllerTypeElement.getAnnotationMirrors()
.stream().filter(a ->
!TypeMirrorUtils.matchesClassName(TypeMirrorUtils.getTypeElement(a.getAnnotation
Type(), processingEnv),
Component.class)).map(Object::toString).collect(Collectors.joining("\n")));
```

After defining the new generated controller, we need to define a method for each DynamicRestMapping. The first step is to add the appropriate request mapping annotation based on *DynamicRestMapping.requestMethod()*. After that, same as with controllers, we need to persist the annotations that the original method contains. For example, if the original method is:

```
@DynamicRestMapping(path = "", requestMethod = RequestMethod.GET, entity = Product.class)
@SecurityRequirements(@SecurityRequirement(name = "bearerAuth"))
public ResponseEntity<PageResponse<GetModerateProductDTO>> getAllProducts(...
```

The generated method must contain the *@SecurityRequirements* annotation. The generated code should look like:

```
@GetMapping({""})
@SecurityRequirements(@SecurityRequirement(name = "bearerAuth"))
public ResponseEntity<PageResponse<GetModerateProductDTO>> getAllProducts(...
```

At this point, we need to define the return type of the method, the parameters and finally call the original method and return whatever the original method returns. Since *@DynamicRestMapping* is used on methods, we can retrieve the return type by simply calling *ExecutableElement#getReturnType*. To resolve existing method parameters we use *ExecutableElement#getParameters*, in addition to existing parameters, we need to generate the request parameters needed to build filters and sort details as explained in sections 3.2.3 and 3.2.4. In the end, we need to call the original method and pass all the parameters that it requested, in addition to *CriteriaParameters.* To call the method we simply autowire the original Controller which the original method is part of and call the method. This will result in something like

```
@Autowired
com.indritbreti.restprocessor.API.DemoEntity.ProductController controller_;
…
return this.controller_.getAllProducts(cp, authorizationHeader, searchQuery);
```

3.2.15. Persisting data to runtime

As mentioned in section 3.2.11 we did resolve FieldDetails for each entity during compilation using the annotation processor. However, we cannot access the values from *ProcessorFieldDetailsRegistry* in runtime since the instance of this object was part of the annotation processor. To overcome this we can persist the data into runtime by serializing it during annotation processing and deserializing it once in runtime. After the annotation processor is done it calls the following method to serialize the Hashtable containing FieldDetails *processorFieldDetailsRegistry.serialize(processingEnv);*

The data is written through an ObjectOutputStream using:

processingEnvironment.getFiler().createResource(StandardLocation.CLASS_OUTPUT, generatedSourcesPackageName+".persist", "field_details_registry.data");

Instead of using *ProcessorFieldDetailsRegistry* the runtime can use *FieldDetailsRegistry,* a generated source that contains a *deserialize()* method that reads *field_details_registry.data* into the Hashtable. The reason why we need to use *FieldDetailsRegistry* instead of *ProcessorFieldDetailsRegistry* is that we cannot reference *field_details_registry.data* from *ProcessorFieldDetailsRegistry* since this one is part of our package. Being a generated source, *FieldDetailsRegistry* lies in the same package as *field_details_registry.data* from there it can use the relative path to reference the data file.

It is worth mentioning that we should be careful when serializing the data. As pointed out in section 3.2.8 the annotation processor will run on multiple rounds. We must assure that we serialize all the needed data. In our case, the processor should be done within one round since the generated sources will not use the supported annotations.

Additionally, note to always use *StandardLocation.CLASS_OUTPUT* when generating resources that are note Java source files. If we were to persists the .data file using *getFiler().createSourceFile()* same as we do when generating Java files, the data file would not be copied into the JAR after we packaged the project.

3.2.16. Data Flow Diagram

To have a better understanding of how the data flows through the classes designed for this project we can refer to these two DFD diagrams. The first diagram represents the way the annotation processors uses and persits *FieldDetails.* The second one shows how data flows through the generated classes, from the REST controller to the database and back.

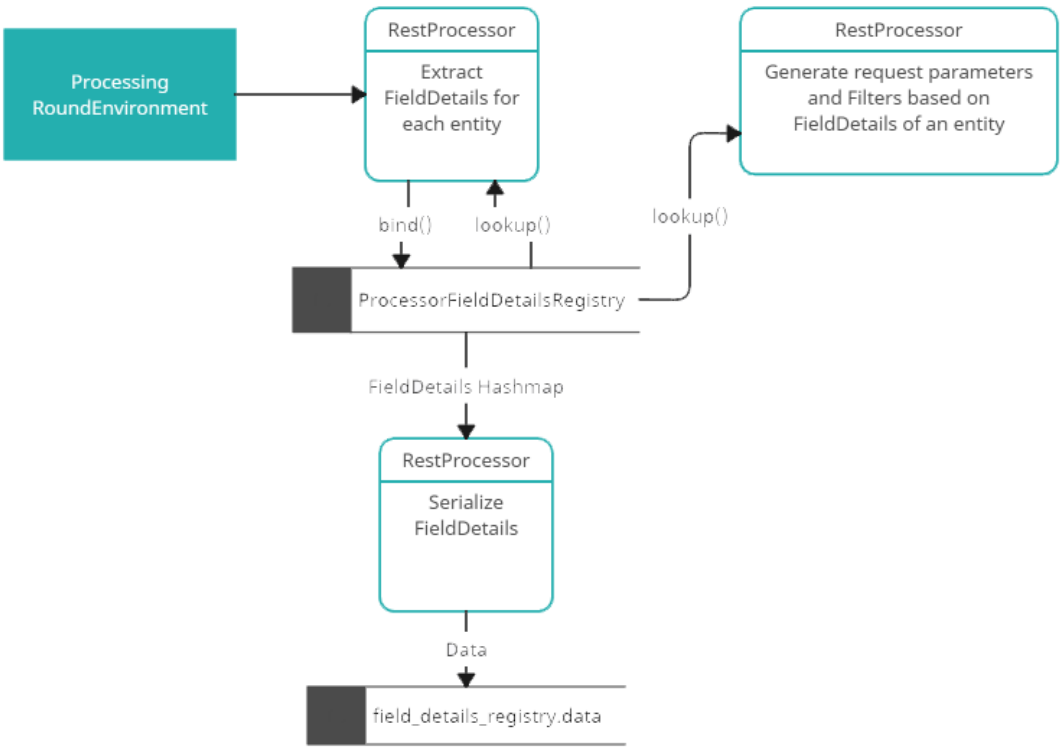

Figure 12 Data Flow Diagram of *FieldDetails* for *RestProcessor*

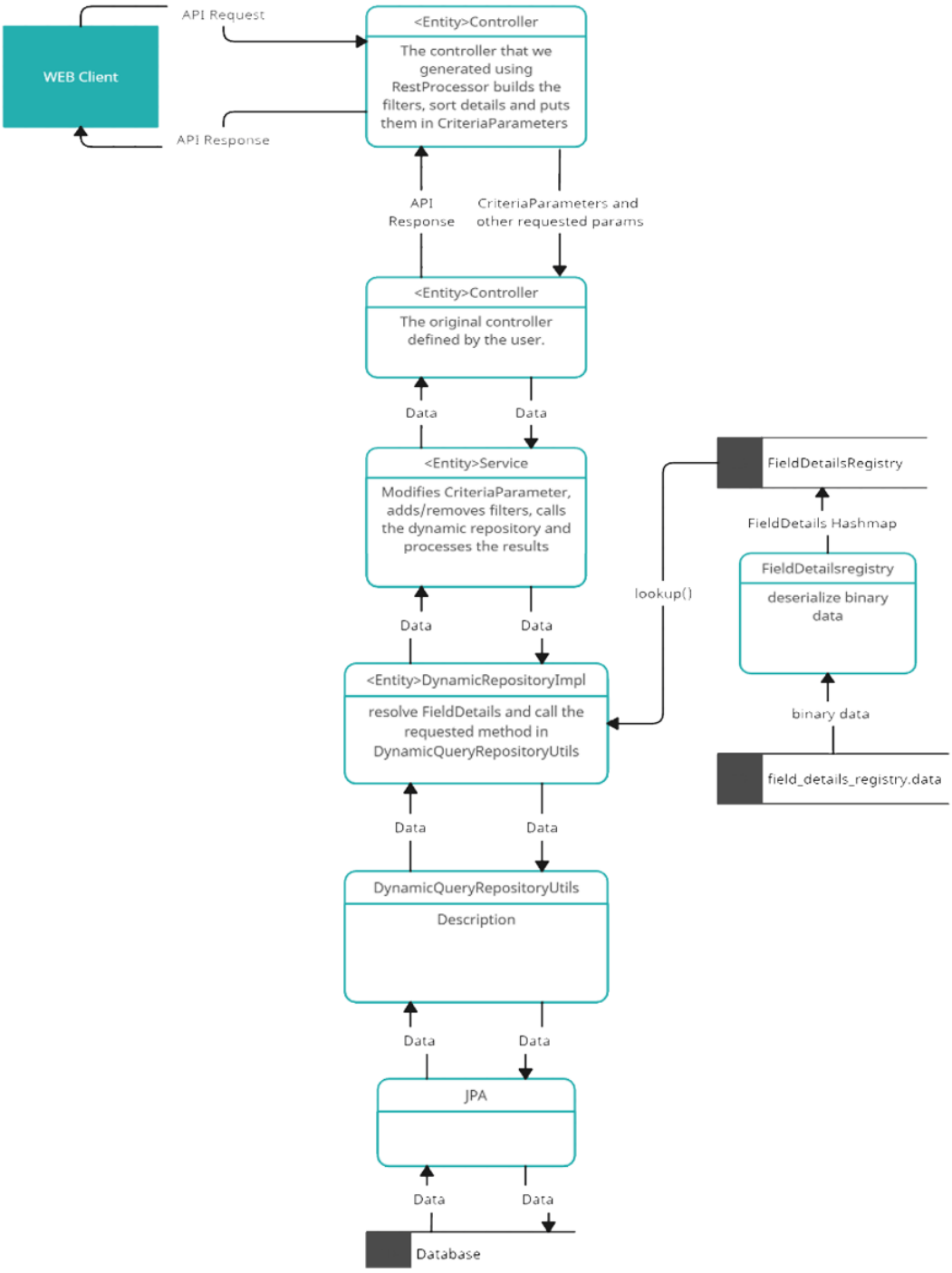

Figure 13 Data Flow Diagram of *RestProcessor*

3.2.17. Simplified Class Diagram

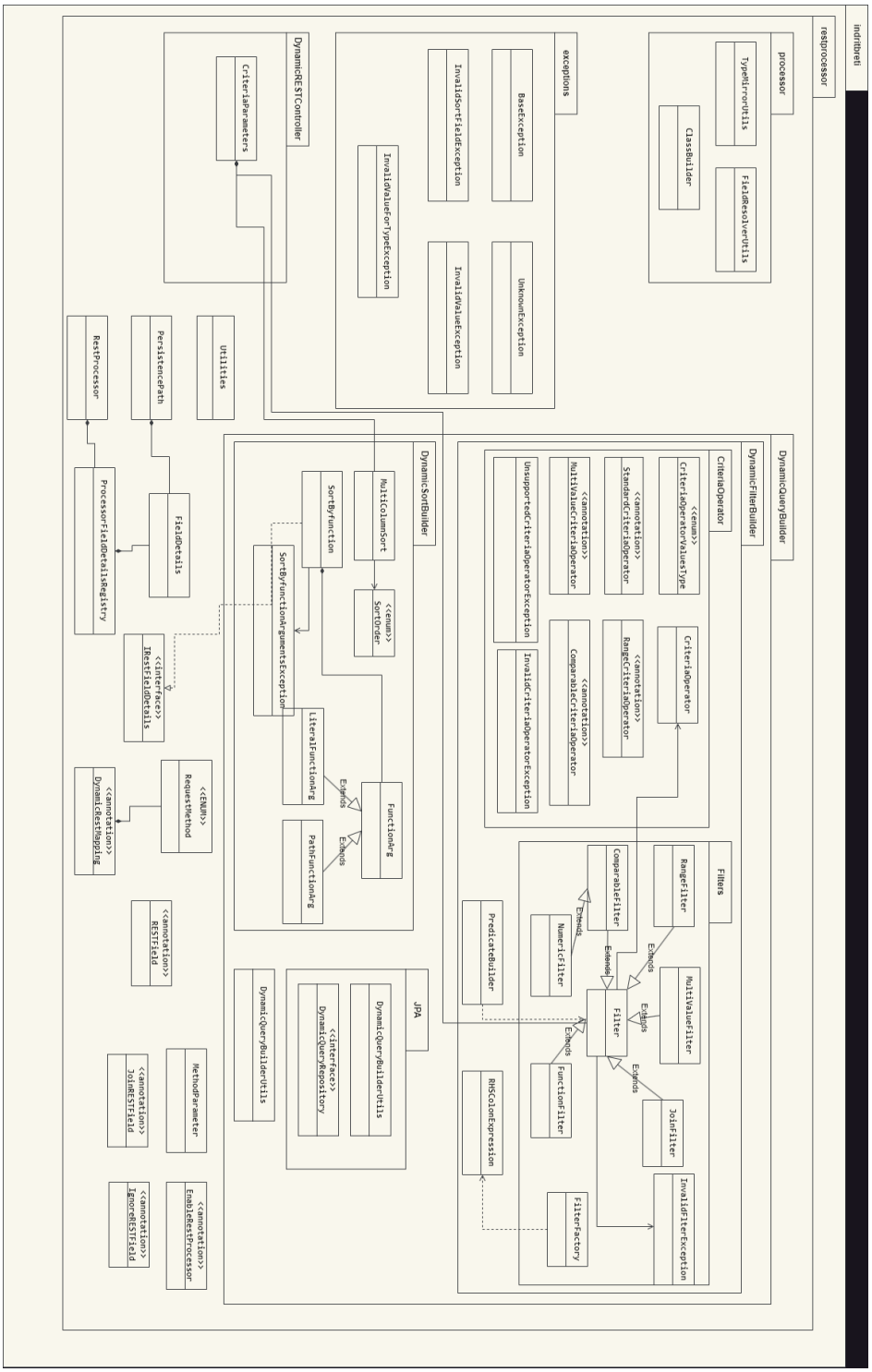

Figure 14 Simplified class diagram for SpringRestProcessor

3.2.18. Building the JAR

Building the jar for a maven project is pretty simple. We use the command "*mvn clean package"* and it packages the JAR for us. However, this will not include all the dependencies that we are using, leading to issues or extra configurations needed from the user. In addition to the normal JAR, we can release a "fat" JAR that includes all the referenced libraries. To do so we can simply rely on the *'maven-jar-plugin'* and the *'maven-assembly-plugin'*.

3.2.19. Integrating the framework on a project

To use the framework, install the JAR file with all the dependencies to your local maven repository using this command:

```
mvn install:install-file
-Dfile='C:/Users/indri/SpringBoot-RestProcessor/springrestprocessor/restprocessor/target/restprocessor-0.1.0-SNAPSHOT-jar-with-dependencies.jar'
-DgroupId='grad-project.indritbreti'
-DartifactId='restprocessor'
-Dversion='0.1.0-SNAPSHOT'
-Dpackaging='jar'
-DgeneratePom='true'
```

After doing so, add the dependency into the pom file:

```
<dependency>
  <groupId>grad-project.indritbreti</groupId>
  <artifactId>restprocessor</artifactId>
   <version>0.1.0-SNAPSHOT</version>
   <scope>compile</scope>
 <dependency>
```

To recognize the generated source files as Java sources we need to add the following plugins:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.6.1</version>
  <configuration>
    <source>17</source>
    <target>17</target>
     <encoding>UTF-8</encoding>
     <verbose>true</verbose>
    <generatedSourcesDirectory>${project.build.directory}/restprocessor-generated-sou
    rces</generatedSourcesDirectory>
  </configuration>
<plugin>

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>1.4</version>
    <executions>
      <execution>
      <id>test</id>
      <phase>generate-sources</phase>
        <goals>
          <goal>add-source</goal>
        </goals>
       <configuration>
         <sources>
           <source>${project.build.directory}/restprocessor-generated-sources/</source>
         </sources>
       </configuration>
     </execution>
  </executions>
</plugin>
```

At this point the annotation processor should be detected automatically. If not, create the file *META-INF/services/javax.annotation.processing.Processor* under the resources directory and add this line in the file *"org.indritbreti.restprocessor.RestProcessor"*

(explained in section 3.2.7). You can now start by annotating the main Applicaion class with "*@EnableRestProcessor*". The source code is generated when the project gets built. Manually build the project and look into target/restprocessor-generated-sources to check if everything is working .

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Testing the functionality

While working on this project I was also building an API for an electric scooter e-commerce website as part of my Software Project Management course. I used this as an opportunity to put the framework to the test. After setting up the project and installing Spring-RestProcessor as described in section 3.2.18 I decided to create an endpoint that allows me to retrieve all *Products* by providing filters and sort orders on any of the fields that the Product entity contains. To do so, I extended the *ProductRepository* from the autogenerated repository *ProductDynamicQueryRepository*

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long>,
ProductDynamicQueryRepository
```

After that, I created a new method *getAllProducts(CriteriaParameters cp)* inside *ProductController*. To map this method to a dynamic endpoint I annotated it with *@DynamicRestMapping(path = "", requestMethod = RequestMethod.GET, entity = Product.class)* I can now call a method on my *ProductService* which then calls *ProductRepository#findAllByCriteria(CriteriaParameters)* as explained in section 3.2.1. The code of the controller looks as below:

```
@DynamicRestMapping(path = "", requestMethod = RequestMethod.GET, entity =
Product.class)
public ResponseEntity<PageResponse<GetModerateProductDTO>>
 getAllProducts(CriteriaParameters cp){
      Page<Product> resultsPage = productService.getAllByCriteria(cp);
      return ResponseFactory.buildPageResponse(resultsPage, product -> new
                              GetModerateProductDTO(product, productService));
}
```

What if I want to use a custom parameter? For example, I would like to filter out all products that have visible=false **if the authenticated user is not an admin user**. To do so I can simply place *@RequestHeader(value = HttpHeaders.AUTHORIZATION, required = false) String authorizationHeader* as a parameter on the *getAllProducts()* method, and the annotation processor will add the same parameter to the RequestMapping that it generates, it will then return the value to us. Now I can check if the user is an admin, and pass a boolean to my service method:

```
productService.getAllByCriteria(!AuthorizationFacade.isAdminAuthorization(authorizat
ionHeader, jwtUtils, appUserService), cp);
```

The definition of the service method we are calling:

```
ProductService#getAllByCriteria(boolean isVisibleRequired, CriteriaParameters cp)
```

Now we can add a new filter to CriteriaParameters from the service:

```
    if (isVisibleRequired)
        cp.addFilter(new Filter<>("visible", CriteriaOperator.EQUAL, true));
```

The user can specify any other filters in the request. Even if the user specifies *visible=true* i.e.: */api/products?id=gt:20&price=lte:300&visible=true* and he is not an admin, the value will be overwritten by cp.addFilter() being called after the controller has constructed cp itself.

The power of SpringRestProcess can be seen as we are able to easily write requests that can filter data based on joined entities, i.e.: */api/products?categories.id=1* will retrieve all products with category id=1. We can do the same for sorting i.e.: */api/products?sortBy=categories.updatedAt;-price*

4.2 Discussion

In terms of performance, the framework has a really small effect on API request response time, that being parsing RHSColonExpressions, building filters, and translating them to predicates. Those operations are relatively cheap and would have to be done in one way or another. On the other hand, the annotation processor does not have any effect on performance other than possibly adding some unnoticeable amount of time to build time and some extra unnoticeble memory usage. However the whole JAR with dependencies is quite big at 38MB, this can be improved in the future.

It is worth mentioning that SpringRestProcessor was designed with extensibility in mind. Fields are resolved by the annotation processor, but users can manually register or remove fields using FieldDetailsRegistry. Filters and API parameters are auto-generated but the user can define a custom parameter at any moment and is able to add and remove filters from CriteriaParameters. All these commodities are offered in order to avoid one of the main downsides of autogenerated code, which is the lack of ability to edit the generated sources mentioned in the introduction of this paper.

While the library has proven to be pretty powerful, it still has much more room for improvements. The project is quite ambiguous and I have left a lot of notes around the code and in the readme stating possible enhancements that we can make. Some notable advancements that need to be made include resolving join fields without the need for @JoinRestField, supporting complex non-JDBC types for serialization, adding support

to send the request parameters in the body to avoid URL length limitations, adding support for full CRUD operations and more.

In addition, I plan to make this an open-source framework but the platform is not ready. There is a huge lack of documentation that needs to be resolved on both the code design and usage samples. Another great enhancement for the framework would be documenting API parameters in swagger. While we do expose them as available parameters we do not provide any information on possible values and the RHSColon format.

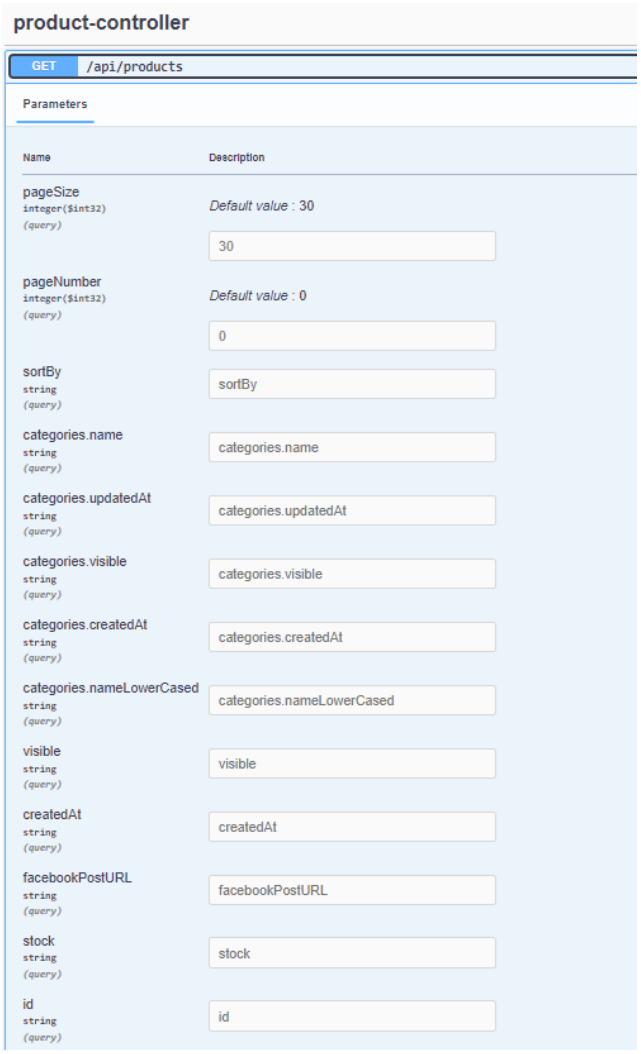

Figure 15 Swagger Documentation

CHAPTER 5

CONCLUSION

While being a promising feature annotation processors are somewhat difficult to work with and quite costly to maintain. We must always do a good analysis of the problem before deciding to use annotation processors to solve a problem to avoid using this feature unnecessarily. However, this project showed that if used correctly, annotation processing can have great benefits. The framework I built demonstrates how annotation processors can be used to speed up WEB API development in Spring, but not only. The ability to generate code by using simple metadata information can be used to enhance programming experience on any Java based software. SpringRestProcessor can drastically reduce the amount of code that developers need to write in order to build powerful APIs. In addition to speed and efficiency, code generation assures us that changes in the main code will be automatically reflected to anything that is generated based of it. This reduces the amount of code that needs to be maintained, making the software less prone to mistakes. The features of this library will allow developers to have full control over the generated code, showing that we can overcome difficulties in uneditable generated sources if we design the software properly.

REFERENCES

Rocha Henrique, Valente Marco, How Annotations are Used in Java: An Empirical Study. SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (2011/01/01)

Peter Pigula, Milan Nosal, Unified compile-time and runtime java annotation processing. Federated Conference on Computer Science and Information Systems (FedCSIS, Technical University of Kosice, Kosice, Slovakia, 2015)

Laplante Phillip, Darwin Ian, AnnaBot: A Static Verifier for Java Annotation Usage, Hindawi Publishing Corporation, 1687-8655 (2009/12/20)

https://docs.oracle.com/javase/tutorial/java/annotations/index.html (lastly visited on 22 June 2023).

https://www.oracle.com/technical-resources/articles/java/ma14-architect-annotations.html (lastly visited on 22 June 2023)

https://www.oracle.com/technical-resources/articles/hunter-meta.html (lastly visited on 22 June 2023)

https://www.oracle.com/technical-resources/articles/hunter-meta1.htm (lastly visited on 22 June 2023)

https://www.oracle.com/technical-resources/articles/hunter-meta2.html (lastly visited on 22 June 2023)

https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/RetentionPolicy.html (lastly visited on 22 June 2023)

https://github.com/FasterXML/jackson (lastly visited on 22 June 2023)

https://www.baeldung.com/java-annotation-processing-builder (lastly visited on 22 June 2023)

https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html (lastly visited on 22 June 2023)

https://www.baeldung.com/hibernate-criteria-queries (lastly visited on 22 June 2023)

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ (lastly visited on 22 June 2023)