Using Java @Annotations to Build Full Spring Boot REST API

Check out this post to learn more about the best Java @annotations for building a Spring Boot REST API.

https://dzone.com/articles/key-java-annotations-to-build-full-spring-boot-res

Jailson Evora, Sep. 25, 19

Here's more on building Java annotations for full Spring Boot REST APIs

This post aims to demonstrate important Java @annotations used to build a functional Spring Boot REST API. The use of Java annotation gives developers the capability to reduce the code verbosity by a simple annotation.

As an example, we can refer to a transaction. By the standard pro-grammatically process with a transaction template, this requires a more complex config and boilerplate code to write, while this can be achieved with a simple @Transactional declarative annotation.

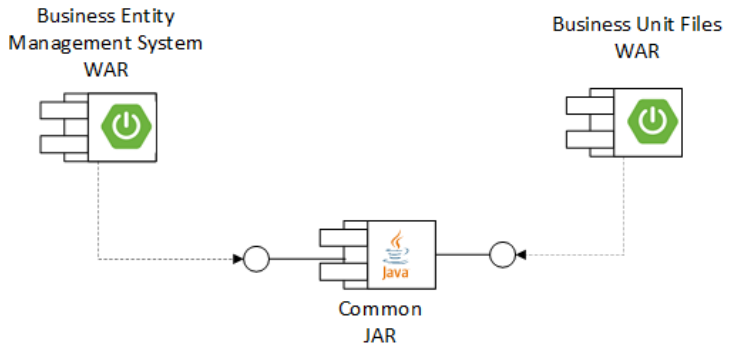
You may also like: **A Guide to Spring Framework Annotations**

In the Java programming language, an annotation is a form of syntactic metadata that can be added to Java source code. Java annotations can also be embedded in and read from Java class files generated by the Java compiler. This allows annotations to be retained by the Java Virtual Machine at run-time and read via reflection. The support for annotation starts from version 5, allowing different Java frameworks to adopt these resources [3].

Annotations can also be used in REST API. REST stands for Representational State Transfer and is an architectural style for designing distributed applications. Brought by Roy Fielding in his PhD dissertation, he proposed it as a basis six principles where it should be separated between clients and servers; the communication between client and server should be stateless; multiple hierarchical layers can exist between them; the server responses must be declared as cacheable or noncacheable; the uniformity of their interfaces must be based in all interactions between client, server, and intermediary components and finally the clients can extend their functionality by using code on demand.

Case Study

The API is a simple module to implement a CRUD operation of Business Entity from a more complex system with the intention to coordinate and harmonize economic information relating to enterprises, establishments, and groups of entities. For the sake of simplicity, the API uses the H2 in-memory database.



The project structure is constituted by three modules, but this post will focus on the module to manage entities. That module has a dependency from the Common module, where it shares things like the error handling and essential useful classes with the remaining part of the whole system. The sample code is accessible from the GitHub repository; to download it, please follow this link.

Let's get started.

Spring Boot Auto-Configuration

The great advantage of Spring Boot is that we can focus on the business rules, thus avoiding some tedious development steps, boilerplate code, and a more complex configuration improving the development and to ease the bootstrapping of new Spring applications.

To start the configuration of a new Spring Boot application, the Spring Initializr creates a simple POJO class to configure the initialization of the application. We have two ways to decorate the configuration. One is using the @SpringBootApplication annotation when we have fewer modules in our solution.

If we have a solution with a more complex structure, we need to specify the different paths or the base packages of our modules to the Spring Boot application initializer class. We can achieve it using @EnableAutoConfiguration and @ComponentScan annotations. @EnableAutoConfiguration instructs Spring Boot to start adding beans based on classpath settings, other beans, and various property settings, while @ComponentScan allows spring to look for other components, configurations, and services in the package, letting it find the controllers among other components.

These two annotations cannot be used at the same time with @SpringBootApplication. @SpringBootApplication is a convenience annotation that adds all of them. It is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes.

```
package com.BusinessEntityManagementSystem;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
@EnableAutoConfiguration
@EnableJpaAuditing
@EnableJpaRepositories(basePackages = {"com.BusinessEntityManagementSystem"})
@EntityScan(basePackages = {"com.BusinessEntityManagementSystem"})
@ComponentScan(basePackages = {"com.BusinessEntityManagementSystem"})
@Configuration
public class BusinessEntityManagementApplication {
  public static void main(String[] args) {
    SpringApplication.run(BusinessEntityManagementApplication.class, args);
  }
}
```

Another annotation present in the code snippet is the @EnableJpaAuditing. The auditing allows the system to track and logging events related to persistent entities, or entity versioning. Also related to JPA configuration, we have the @EnableJpaRepositories. This annotation enables JPA repositories. It will scan the package of the annotated configuration class for Spring Data repositories by default. Inside this annotation, we specify the base package to be scanned for annotated components.

To find the JPA entities in the project structure, we must instruct the auto-configuration to scan packages using the @EntityScan. To a specific scan, we can specify basePackageClasses(), basePackages(), or its alias value()to define specific packages to scan. If specific packages are not defined, scanning will occur from the package of the class with this annotation.

The final annotation in the class created by Spring Boot Initializr is @Configuration. @Configuration tags the class as a source of bean definitions for the application context. This can be applied in any configuration class that we need.

Java @Annotations in Swagger UI Config

Documentation is an important aspect of any project; therefore, our REST API is documented using the Swagger-UI, one of the many standards metadata to do it. Swagger is a specification and a framework for creating interactive REST API documentation. It enables documentation to be in sync with any changes made to REST services. It also provides a set of tools and SDK generators for generating API client code [2].

In the Swagger-UI class configuration, the first annotation to appear in the @Configuration. As we described above, this indicates to Spring Boot auto-configuration that a class is a configuration class that may contain bean definitions.

```
package com.BusinessEntityManagementSystem;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;
import java.util.Collections;

@Configuration
@EnableSwagger2
@EnableAutoConfiguration
public class SwaggerConfig {



  @Bean
  public Docket api() {
    return new Docket(DocumentationType.SWAGGER_2)
      .select()
      .apis(RequestHandlerSelectors.basePackage("com.BusinessEntityManagementSystem"))
      .paths(PathSelectors.ant("/api/businessEntityManagementSystem/v1/**"))
      .build()
      .apiInfo(metaData());
  }
...
```

A specific annotation for Swagger is the @EnableSwagger2. It indicates that Swagger support should be enabled and to be loaded all required beans defined in see swagger java-config class. This should be applied

to a Spring java config and should have an accompanying @Configuration annotation. The @Bean is a method-level annotation and a direct analog of the XML <bean/> element.

Domain Model

MVC is one of the most important modules of the Spring Framework. It is a common design pattern in UI design. It decouples business logic from UIs by separating the roles of model, view, and controller. The core idea of the MVC pattern is to separate business logic from UIs to allow them to change independently without affecting each other.

In this design pattern, M stands for Models. The model is responsible for encapsulating application data for views to present. It represents the shape of the data and business logic. Model objects retrieve and store model state in a database. Its models usually consist of domain objects that are processed by the service layer and persisted by the persistence layer.

TYPE Java @Annotations

In the model class, we use @Entity annotation that indicates this class is a JPA entity. JPA will be aware that the POJO class can be stored in the database. If we do not define @Table annotation, Spring config will assume that this entity is mapped to a table like the POJO class name. So, in these cases, we can specify the table name using the @Table annotation.

To handle the problem related to the serialization of the model using Jackson API when the model attributes have a lazy loading defined, we have to tell the serializer to ignore the chain or helpful garbage that Hibernate adds to classes, so it can manage lazy loading of data by declaring @JsonIgnoreProperties({"hibernateLazyInitializer", "handler"}) annotation.

```
package com.BusinessEntityManagementSystem.models;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.BusinessEntityManagementSystem.interfaces.models.IBusinessEntityModel;
import org.hibernate.annotations.OnDelete;
import org.hibernate.annotations.OnDeleteAction;
import javax.persistence.*;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
@Entity
@Table(name="BEMS_BusinessEntity")
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class BusinessEntityModel<IAddressModel extends AddressModel, IPartnerModel
                extends PartnerModel, IStoreModel extends StoreModel, IAffiliatedCompanyModel
                extends AffiliatedCompanyModel, IEconomicActivityCodeModel
                extends EconomicActivityCodeModel> extends AuditModel<String>
                implements IBusinessEntityModel, Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @NotNull
    @Column(name = "ID_BusinessEntity", updatable = false, nullable = false)
    private long id;
...
```

FIELD Java @Annotations

To a class field, there are many kinds of annotations dependent on the type and purpose of that field. For example, @Id annotation must be declared in one of the class attributes. Every entity object stored in the database has a primary key. Once assigned, the primary key cannot be modified. @GeneratedValue indicating that the framework should generate the document key value using the specified generator type like {AUTO, IDENTITY, SEQUENCE, and TABLE}.

Another interesting annotation targeting domain model field is @NotNull. It is a common Spring annotation to declare that annotated elements cannot be null. It can be used as well in a method or parameter. The @Column annotation specifies the name to the database column and also the table behavior. This behavior can be set to prevent it from being updated or to be null.

Sometimes most objects have a natural identifier, so Hibernate also allows to model this identifier as a natural identifier of an entity and provides an extra API for retrieving them from the database. This is achieved using @NaturalId annotation.

```
...
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @NotNull
   @Column(name = "ID_BusinessEntity", updatable = false, nullable = false)
   private long id;
   @NaturalId
   @NotEmpty
   @NotNull
   @Column(name = "NATURAL_ID", updatable = false, nullable = false)
   private long naturalId;
   @Column(name = "TaxId")
   @Size(min = 1, max = 50)
   private String taxId;
   @Column(name = "Status")
   private int status = 1;
...
```

If we want to prevent the element of an entity to be not null nor empty, we can also annotate it with @NotEmpty. It target a vast kind of element, since {METHOD,FIELD,ANNOTATION_TYPE, CONSTRUCTOR,PARAMETER,TYPE_USE}. The @Size annotation delimits the boundaries to the element annotated. The boundaries are specified by two attributes, min, and max.

Relationships Java @Annotations

One of the most important features of any ORM mechanism is how one specifies the mapping from relationships between Objects to their database counterparts. In the code below, there is a @OneToOne annotation to describe the relationship between the BusinessEntity class with the Address class model. The @JoinColumn annotation specifies the column that will be treated as foreign key in this relationship.

As well, as @OneToOne annotation, we can manage many to many relationships. @ManyToMany annotation describes the relationship with members of the Partner class. Like other relationship annotations, it is also possible to specify cascade rules as well as the fetch type. Dependent on the cascade chosen setting, when a BusinessEntity is deleted, the associated Partner will also be deleted.

```
...
  // region OneToOne
  @OneToOne(fetch = FetchType.EAGER, cascade =  CascadeType.ALL,
                  targetEntity= AddressModel.class)
  @JoinColumn(name = "Address", nullable = false)
  @OnDelete(action = OnDeleteAction.CASCADE)
  private IAddressModel address;

  // endregion OneToOne
  // region ManyToMany
  @ManyToMany(fetch = FetchType.LAZY,
                      cascade = {
                            CascadeType.ALL
                      },
                      targetEntity= PartnerModel.class
                      )
  @JoinTable(name = "BSUF_BusinessEntity_Partner",
                  joinColumns = { @JoinColumn(name = "ID_BusinessEntity") },
                  inverseJoinColumns = { @JoinColumn(name = "ID_Partner") })
  private Set<IPartnerModel> partner = new HashSet<>();
  // endregion ManyToMany
...
```

Together with @ManyToMany annotation, we specify the @JoinTable annotation that allows us in a many to many relations to define the bridge table between two other related table, using two essential attributes, joincolumns for the class where we declare the @ManyToMany annotation and inverseJoinColumns for another table.

```
...
 // inverser many to many
 @ManyToMany(fetch = FetchType.LAZY,
                      cascade = {
                            CascadeType.PERSIST,
                            CascadeType.MERGE
                      },
                      mappedBy = "partner")
 @JsonIgnore
 private Set<IBusinessEntityModel> businessEntity = new HashSet<>();
...
```

In the other table, it is recommended to define also the inverse relation. This declaration is slightly different from the showed in the code related to the Bussines Entity model. The inverse relation declaration differentiates by the attribute " mappedBy."

Data Transfer Object

The Data transfer object is a very popular design pattern. It is an object that defines how the data will be sent over the network. DTO is only used to pass data and does not contain any business logic.

TYPE Java @Annotations

Sometimes, we need to transfer data between entities through JSON. To serialize and deserialize the DTO object, we need to annotate these objects with Jackson annotation. The @JsonInclude(JsonInclude.Include.NON_NULL) indicates when the annotated property can be serialized. By using this annotation, we can specify simple exclusion rules based on property values. It can be used on a field, method, or constructor parameter. It can also be used in classes where in some cases the specified rules are applied to all properties of the class.

```
package com.BusinessEntityManagementSystem.dataTransferObject;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

/*@JsonInclude(JsonInclude.Include.NON_NULL)*/
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class BusinessEntity {
...
```

@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"}) allows Jackson to ignore the garbage created by Hibernate, so it can manage the lazy loading of data as referred before.

FIELD Java @Annotations

A field in DTO objects might have different kinds of annotations as well. @JsonProperty annotation is used to specify the name of the serialized properties. @JsonIgnore is annotated at a class property level to ignore it. Along with @JsonIgnore, there are also @JsonIgnoreProperties and @JsonIgnoreType. Both annotations are part of Jackson API and are used to ignore logical properties in JSON serialization and deserialization.

```
package com.BusinessEntityManagementSystem.dataTransferObject;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

/*@JsonInclude(JsonInclude.Include.NON_NULL)*/
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class BusinessEntity {
   @JsonProperty("naturalId")
   private long naturalId;
   @JsonProperty("taxId")
   private String taxId;
   @JsonProperty("status")
   private int status = 1;
...
```

Jackson API is a high-performance JSON processor for Java. It provides many useful annotations to apply on DTO objects, allowing us to serialize and deserialize objects from JSON to JSON.

Controller

Controllers stand for C on the MVC pattern. Controllers are responsible for receiving requests from users and invoking back-end services for business processing. After processing, it may return some data for views to present. Controllers collect it and prepare models for views to present. Controllers are often referred to as a dispatcher servlet. It acts as the front controller of the Spring MVC framework, and every web request must go through it so that it can manage the entire request-handling process [3]. When a web request is sent to a Spring MVC application, a controller first receives the request. Then it organizes the different

components configured in Spring's web application context or annotations present in the controller itself, all needed to handle the request.

TYPE Java @Annotations

To define a controller class in Spring Boot, a class has to be marked with the @RestController annotation. @RestController annotation is a convenience annotation that is itself annotated with @Controller and @ResponseBody. Types that carry this annotation are treated as controllers where @RequestMapping methods assume @ResponseBodysemantics by default. The value attribute may indicate a suggestion for a logical component name, to be turned into a Spring bean in case of an auto-detected component.

```
package com.BusinessEntityManagementSystem.v1.controllers;
import com.BusinessEntityManagementSystem.dataAccessObject.IBusinessEntityRepository;
import com.BusinessEntityManagementSystem.interfaces.resource.IGenericCRUD;
import com.BusinessEntityManagementSystem.models.BusinessEntityModel;
import com.Common.Util.Status;
import com.Common.exception.ResourceNotFoundException;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.Transactional;
import javax.validation.Valid;
import java.util.Optional;
@RestController("BusinessEntityV1")
@RequestMapping("/api/businessEntityManagementSystem/v1")
@Api(value = "businessEntity")
public class BusinessEntityController implements IGenericCRUD<BusinessEntityModel> {
...
```

CONSTRUCTOR and METHOD Java @Annotations

When a @RestController-annotated class receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller map each request to a handler method by one or more handler mappings. To do so, a controller class's methods are decorated with the @RequestMapping annotation, making them handler methods.

For Swagger documentation purposes, the @ApiOperation annotation is used to declare a single operation within an API resource. An operation is considered a unique combination of a path and an HTTP method. Only methods that are annotated with @ApiOperation will be scanned and added to the API declaration. Some handlers or operations need to ensure data integrity and consistency using the transaction.

Transaction management is an essential technique in enterprise applications to ensure data integrity and consistency. Spring supports both programmatic and declarative (@Transactional) transaction management.

```
...
  @Autowired
   public BusinessEntityController(IBusinessEntityRepository businessEntityRepository) {
      this.businessEntityRepository = businessEntityRepository;
   }

   @RequestMapping(value = "/businessEntity/{id}", method = RequestMethod.GET,
                              produces = "application/json")
   @ApiOperation(value = "Retrieves given entity", response=BusinessEntityModel.class)
   public ResponseEntity<?> get(@Valid @PathVariable Long id){
          checkIfExist(id);
          return new ResponseEntity<> (businessEntityRepository.findByIdAndStatus(
                                                    id, Status.PUBLISHED.ordinal()), HttpStatus.OK);
   }

   @Transactional
   @RequestMapping(value = "/businessEntity", method = RequestMethod.POST,
                              produces = "application/json")
   @ApiOperation(value = "Creates a new entity",
                           notes="The newly created entity Id will be sent in the location response header")
   public ResponseEntity<Void> create(@Valid @RequestBody BusinessEntityModel newEntity){
           return new ResponseEntity <>(null,
                              getHttpHeaders(businessEntityRepository.save(newEntity)), HttpStatus.CREATED);
   }
...
```

Managing transactions pro-grammatically, we have to include transaction management code in each transactional operation (boilerplate code). As a result, the boilerplate transaction code is repeated in each of these operations. In most cases, declarative transaction management is preferable instead of programmatic transactions. It's achieved by separating transaction management code from our business methods via declarations. This can help us to enable transactions for our applications more easily and define a consistent transaction policy, although declarative transaction management is less flexible than programmatic transaction management. Programmatic transaction management allows us to control transactions through our code [3].

Another useful annotation used in a well-designed system is @Autowired.@Autowired can be used at the constructor method to resolve and inject collaborating beans into a bean, lead us to better application design. Applications developed using the principle of separating interface from implementation and the dependency injection pattern are easy to test, both for unit testing and integration testing because that principle and pattern can reduce coupling between different units of our application.

PARAMETER Java @Annotations

Besides authentication and authorization, one area of importance in building secure web services is to ensure that inputs are always validated [2]. Java Bean annotations provide mechanisms to implement input validation. We can achieve it by using @Valid annotation in methods parameters.

Our class should validate incoming identifier requests before processes the soft delete. By simply adding the @Valid annotation to the method, Spring will ensure that the incoming identifier requests are run through our defined validation rules first.

```
...
   @Transactional
   @RequestMapping(value = "/businessEntity/{id}", method = RequestMethod.DELETE,
                           produces = "application/json")
   @ApiOperation(value = "Deletes given entity")
   public ResponseEntity<Void> delete(@Valid @PathVariable Long id,
                                      @Valid @RequestBody String lastModifiedBy){
        Optional<BusinessEntityModel> softDelete = businessEntityRepository.findByIdAndStatus(
                                                                        id, Status.PUBLISHED.ordinal());
        if (softDelete.isPresent()) {
              BusinessEntityModel afterIsPresent = softDelete.get();
              afterIsPresent.setStatus(Status.UNPUBLISHED.ordinal());
              afterIsPresent.setLastModifiedBy(lastModifiedBy);
              businessEntityRepository.save(afterIsPresent);
              return new ResponseEntity<>(HttpStatus.OK);
        }
        else
              throw new ResourceNotFoundException("Entity with id " + id + " not found");
   }
...
```

@PathVariable , as well as @RequestParam, are used to extract values from the HTTP request, there is a subtle difference between them. @RequestParam is used to get the request parameters from URL (https://www.jeevora.com/...?id=1), also known as query parameters, while @PathVariable extracts values from URI (https://www.jeevora.com/id/1) as demonstrated in our case study.

@RequestBody annotation indicates that a method parameter should be bound to the body of the web request while @ResponseBody indicates a method return value should be bound to the web response body.

Data Access Object

A typical design mistake is to mix different types of logic (e.g. presentation logic, business logic, and data access logic) in a single large module. This reduces the module's reusability and maintainability because of the tight coupling it introduces. The general purpose of the Data Access Object (DAO) pattern is to avoid these problems by separating data access logic from business logic and presentation logic. This pattern recommends that data access logic be encapsulated in independent modules called data access objects [3].

Repositories, or Data Access Objects (DAO), provide an abstraction for interacting with datastores. Repositories traditionally include an interface that provides a set of finder methods such as findById, findAll for retrieving data, and methods to persist and delete data. Repositories also include a class that implements this interface using datastore-specific technologies. It is also customary to have one repository per domain object. Although this has been a popular approach, there is a lot of boilerplate code that gets duplicated in each repository implementation [1].

```
package com.BusinessEntityManagementSystem.dataAccessObject;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.NoRepositoryBean;
import org.springframework.data.repository.PagingAndSortingRepository;
import java.util.Optional;

@NoRepositoryBean
public interface IGenericRepository<T> extends PagingAndSortingRepository<T, Long> {
        Optional<T> findByIdAndStatus(long id, int status);
        Page<T> findAllByStatus(int status, Pageable pageable);
}
```

With @NoRepositoryBean annotation, we can use it to exclude repository interfaces from being picked up and thus, in consequence, getting an instance being created. This will typically be used when providing an extended base interface for all repositories in combination with a custom repository base class to implement methods declared in that intermediate interface. In this case, we typically derive our concrete repository interfaces from the intermediate one, but we don't want to create a Spring bean for the intermediate interface.

Conclusion

By the end of this demonstration, we have used the following Java @annotations in our API represented in the table below. It is divided by the target and purpose of each annotation. Sometimes we have cases where some annotations have more than one target, although not all are indicated in the table.

	Target				
KEY @ANNOTATIONS **CONFIG**	**TYPE**	**METHOD**	**FIELD**	**PARAMETER**	
@EnableAutoConfiguration	x	x			
@EnableJpaAuditing	x	x			
@EnableJpaRepositories	x	x			
@EntityScan	x	x			
@ComponentScan	x	x			
@Configuration	x	x			
@EnableSwagger2	x	x			
@Entity		x			
@Table		x			
@RestController		x			
@RequestMapping		x			
@Api		x			
@NoRepositoryBean		x			
@Transactional		x	x		
@Autowired			x		
@ApiOperation			x		
@RequestMapping			x		
@Id				x	
@GeneratedValue				x	
@Column				x	
@NaturalId				x	
@NotEmpty				x	
@Size				x	
@OneToOne			x	x	

		Target				
KEY @ANNOTATIONS	**CONFIG**	**TYPE**	**METHOD**	**FIELD**	**PARAMETER**	
@JoinColumn				x	x	
@OnDelete					x	
@ManyToMany				x	x	
@ManyToOne				x	x	
@OneToMany				x	x	
@JoinTable				x	x	
@Valid						x
@PathVariable						x
@RequestBody						x
@ResponseBody						x

References

[1] Balaji Varanasi, Sudha Belida, Spring REST - Rest and Web Services development using Spring, 2015;

[2] Ludovic Dewailly, Building a RESTful Web Service with Spring - A hands-on guide to building an enterprise-grade, scalable RESTful web service using the Spring Framework, 2015;

[3] Marten Deinum, Daniel Rubio, Josh Long, Spring 5 Recipes: A Problem-Solution Approach, 2017.

Further Reading

How to Create a REST API With Spring Boot

Creating a REST Web Service With Java and Spring

A Guide to Spring Framework Annotations