Summary of Common Annotations in Spring & SpringBoot

Guide frame SpringBoot, Spring About 7105 words About 24 minutes

It's no exaggeration to say that the common Spring/SpringBoot annotations introduced in this article cover most of the common scenarios you'll encounter in your work. For each annotation, this article provides detailed usage instructions. Once you master these, developing projects with Spring Boot will be a breeze!

Why write this article?

I recently saw a widely shared article online about commonly used Spring Boot annotations. However, the content is somewhat misleading and may not be very user-friendly for developers without much practical experience. So I spent a few days summarizing this article, hoping to help everyone better understand and use Spring annotations.

Due to my limited ability and energy, if there are any errors or omissions, please feel free to point them out! Thank you very much!

Spring Boot Basic Annotations

`@SpringBootApplication` It is the core annotation of Spring Boot application, usually used to mark the main startup class.

Example:

```java
@SpringBootApplication
public class SpringSecurityJwtGuideApplication {
      public static void main(java.lang.String[] args) {

SpringApplication.run(SpringSecurityJwtGuideApplication.class,
args);
    }
}
```

We can think `@SpringBootApplication` of it as a combination of the following three annotations:

- **`@EnableAutoConfiguration`** : Enable Spring Boot's automatic configuration mechanism.
- **`@ComponentScan`** : Scan classes annotated with `@Component` , `@Service` , `@Repository` , etc. `@Controller`
- **`@Configuration`** : Allows you to register additional Spring beans or import other configuration classes.

The source code is as follows:

```java
package org.springframework.boot.autoconfigure;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
   ......
}

package org.springframework.boot;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {

}
```

Spring Bean

Dependency Injection (DI)

`@Autowired` Used to automatically inject dependencies (i.e. other Spring beans). It can be annotated on constructors, fields, setter methods, or configuration methods, and the Spring container will automatically find beans of matching types and inject them.

```java
@Service
public class UserServiceImpl implements UserService {
    // ...
}

@RestController
public class UserController {
    //
    @Autowired
    private UserService userService;
    // ...
}
```

When there are multiple beans of the same type, `@Autowired` the default injection by type may cause ambiguity. In this case, you can `@Qualifier` use it in conjunction with to accurately select the instance to be injected by specifying the bean name.

```java
@Repository("userRepositoryA")
public class UserRepositoryA implements UserRepository { /* ... */
}

@Repository("userRepositoryB")
public class UserRepositoryB implements UserRepository { /* ... */
}

@Service
public class UserService {
    @Autowired
```

```
    @Qualifier("userRepositoryA") //            "userRepositoryA"
   Bean
    private UserRepository userRepository;
    // ...
}
```

`@Primary` This also addresses the injection problem of multiple bean instances of the same type. When defining a bean (for example, using `@Bean` a Bean or class annotation) `@Primary`, add an annotation to indicate that the bean is the **preferred** injection target. When `@Autowired` injecting a Bean, if you don't `@Qualifier` specify a name using BeanName, Spring will prioritize `@Primary` the bean with BeanName.

```java
@Primary //    UserRepositoryA
@Repository("userRepositoryA")
public class UserRepositoryA implements UserRepository { /* ... */
}

@Repository("userRepositoryB")
public class UserRepositoryB implements UserRepository { /* ... */
}

@Service
public class UserService {
    @Autowired //          UserRepositoryA         @Primary
    private UserRepository userRepository;
    // ...
}
```

`@Resource(name="beanName")` This annotation, defined by the JSR-250 specification, is also used for dependency injection. It defaults to searching for beans for injection **by name** `@Autowired`, rather than **by type**. If `name` the attribute is not specified, it attempts to search by field or method name. If no such search is found, it falls back to searching by type (similar to the default `@Autowired`).

`@Resource` It can only be annotated on fields and Setter methods, and does not support constructor injection.

```java
@Service
public class UserService {
    @Resource(name = "userRepositoryA")
    private UserRepository userRepository;
    // ...
```

```
}
```

Bean Scope

`@Scope("scopeName")` Defines the scope of Spring Bean, that is, the life cycle and visibility range of the Bean instance. Common scopes include:

- **singleton** : There is only one bean instance in the IoC container. Beans in Spring are singletons by default, which is an application of the singleton design pattern.
- **prototype** : A new bean instance is created each time it is retrieved. That is, `getBean()` two consecutive Bean instances are obtained.
- **request** (only available for Web applications): Each HTTP request will generate a new bean (request bean), which is only valid within the current HTTP request.
- **session** (only available for Web applications): Each HTTP request from a new session will generate a new bean (session bean), which is only valid within the current HTTP session.
- **application/global-session** (available only for Web applications): Each Web application creates a bean (application bean) when it starts. The bean is valid only during the startup time of the current application.
- **websocket** (only available for Web applications): A new bean is created for each WebSocket session.

java

```java
@Component
//                   PrototypeBean
@Scope("prototype")
public class PrototypeBean {
    // ...
}
```

Bean Registration

The Spring container needs to know which classes should be managed as beans. Besides `@Bean` explicitly declaring them using the ___register___ method (typically within `@Configuration` a class), a more common approach is to mark classes with the Stereotype annotation and, through component scanning, let Spring automatically discover and register these classes as beans. These beans can then be `@Autowired` injected into other components using methods such as ___register___ .

The following are some common annotations for registering beans:

- `@Component`: A general annotation that can mark any class as `Spring` a component. If you don't know which layer a bean belongs to, you can use `@Component` annotations.
- `@Repository`: Corresponding to the persistence layer, namely the Dao layer, it is mainly used for database-related operations.
- `@Service`: Corresponding to the service layer, it mainly involves some complex logic and requires the use of the Dao layer.
- `@Controller`: Corresponding to the Spring MVC control layer, it is mainly used to accept user requests and call the Service layer to return data to the front-end page.
- `@RestController`: A composite annotation equivalent to `@Controller` + `@ResponseBody`. It is specifically used to build controllers for RESTful web services. `@RestController` For classes annotated with, the return values of all their handler methods are automatically serialized (usually as JSON) and written to the HTTP response body, rather than being parsed as view names.

`@Controller` vs `@RestController`

- `@Controller`: Mainly used in traditional Spring MVC applications. The method return value is usually a logical view name, which requires a view resolver to render the page. If you need to return data (such as JSON), you need to add additional `@ResponseBody` annotations to the method.
- `@RestController`: Designed for building RESTful APIs that return data. When this annotation is applied to a class, all method return values are treated as response bodies by default (essentially, each method implicitly adds them). This annotation is commonly used to return JSON or XML data. It's a popular choice `@ResponseBody` in modern applications with separate frontends and backends. `@RestController`

For a comparison of `@RestController` @RestController and @Controller `@Controller`, see this article: @RestController vs @Controller .

Configuration

Declare configuration class

`@Configuration` This is primarily used to declare a class as a Spring configuration class. While `@Component` annotations can be used instead, `@Configuration` they more clearly express the class's purpose (bean definition), provide clearer semantics, and facilitate specific Spring processing (for example, ensuring `@Bean` singleton behavior for meth via CGLIB proxies).

```java
@Configuration
public class AppConfig {

    // @Bean                          Bean
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }

    //                        @Bean
}
```

Read configuration information

In application development, we often need to manage some configuration information, such as database connection details, keys or addresses of third-party services (such as Alibaba Cloud OSS, SMS services, and WeChat authentication), etc. Usually, this information is **stored in a configuration file** (such as `application.yml` or `application.properties`) for easy management and modification.

Spring provides a variety of convenient ways to read these configuration information. Suppose we have the following `application.yml` file:

```yaml
wuhan2020: 2020


my-profile:
  name: Guide
  email: koushuangbwcx@163.com

library:
  location:
  books:
    - name:
      description:
                              —


    - name:
      description:                                “    ”
```

```
      —
  - name:
    description:
```

Here are some common ways to read configuration:

1. `@Value("${property.key}")` Inject a single property value from a configuration file (such as `application.properties` or `application.yml`). It also supports Spring Expression Language (SpEL) to implement more complex injection logic.

```java
@Value("${wuhan2020}")
String wuhan2020;
```

2. `@ConfigurationProperties` It can read configuration information and bind it with Bean, which is more commonly used.

```java
@Component
@ConfigurationProperties(prefix = "library")
class LibraryProperties {
    @NotEmpty
    private String location;
    private List<Book> books;

    @Setter
    @Getter
    @ToString
    static class Book {
        String name;
        String description;
    }
     getter/setter
  ......
}
```

You can inject it into your class just like using a normal Spring Bean.

```java
@Service
public class LibraryService {

    private final LibraryProperties libraryProperties;

    @Autowired
    public LibraryService(LibraryProperties libraryProperties) {
        this.libraryProperties = libraryProperties;
    }

    public void printLibraryInfo() {
        System.out.println(libraryProperties);
    }
}
```

Load the specified configuration file

`@PropertySource` Annotations allow you to load custom configuration files. This is useful when you need to store some configuration information independently.

```java
@Component
@PropertySource("classpath:website.properties")

class WebSite {
    @Value("${url}")
    private String url;

     getter/setter
  ......
}
```

Note : When using `@PropertySource` , make sure the external file path is correct and that the file is in the classpath.

For more information, please check out my article: How to read the configuration file elegantly in SpringBoot in 10 minutes ?.

MVC

HTTP Request

5 common request types:

- **GET** : Request to get a specific resource from the server. For example: `GET /users` (get all students)
- **POST** : Create a new resource on the server. For example: `POST /users` (create student)
- **PUT** : Updates a resource on the server (the client provides the entire updated resource). For example: `PUT /users/12` (updates student number 12)
- **DELETE** : Deletes a specific resource from the server. For example: `DELETE /users/12` (delete student number 12)
- **PATCH** : Updates resources on the server (the client provides the changed properties, which can be considered a partial update). It is rarely used, so I won't give an example here.

GET request

`@GetMapping("users")` Equivalent to `@RequestMapping(value="/users",method=RequestMethod.GET)` .

```java
@GetMapping("/users")
public ResponseEntity<List<User>> getAllUsers() {
 return userRepository.findAll();
}
```

POST request

`@PostMapping("users")` Equivalent to `@RequestMapping(value="/users",method=RequestMethod.POST)` .

`@PostMapping` Usually `@RequestBody` used with to receive JSON data and map it to objects.

```java
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody
UserCreateRequest userCreateRequest) {
 return userRepository.save(userCreateRequest);
}
```

PUT request

`@PutMapping("/users/{userId}")` Equivalent to `@RequestMapping(value="/users/{userId}",method=RequestMethod.PUT)` .

```java
@PutMapping("/users/{userId}")
public ResponseEntity<User> updateUser(@PathVariable(value =
"userId") Long userId,
  @Valid @RequestBody UserUpdateRequest userUpdateRequest) {
  ......
}
```

DELETE request

`@DeleteMapping("/users/{userId}")` Equivalent to `@RequestMapping(value="/users/{userId}",method=RequestMethod.DELETE)`

```java
@DeleteMapping("/users/{userId}")
public ResponseEntity deleteUser(@PathVariable(value = "userId")
Long userId){
  ......
}
```

PATCH request

Generally, in actual projects, we use PATCH requests to update data only after PUT is not enough.

```java
  @PatchMapping("/profile")
  public ResponseEntity updateStudent(@RequestBody
StudentUpdateRequest studentUpdateRequest) {
        studentRepository.updateDetail(studentUpdateRequest);
        return ResponseEntity.ok().build();
```

```
    }
```

Parameter Binding

When processing HTTP requests, Spring MVC provides a variety of annotations for binding request parameters to method parameters. The following are common parameter binding methods:

Extracting parameters from URL path

`@PathVariable` Used to extract parameters from the URL path. For example:

```java
@GetMapping("/klasses/{klassId}/teachers")
public List<Teacher> getTeachersByClass(@PathVariable("klassId")
Long klassId) {
    return teacherService.findTeachersByClass(klassId);
}
```

If the request URL is `/klasses/123/teachers` , then `klassId = 123` .

Binding query parameters

`@RequestParam` Used to bind query parameters. For example:

```java
@GetMapping("/klasses/{klassId}/teachers")
public List<Teacher> getTeachersByClass(@PathVariable Long klassId,
                                        @RequestParam(value =
"type", required = false) String type) {
    return teacherService.findTeachersByClassAndType(klassId,
type);
}
```

If the request URL is `/klasses/123/teachers?type=web` , then `klassId = 123` . `type = web`

Binding JSON data in the request body

`@RequestBody` This function reads the body of a request (possibly a POST, PUT, DELETE, or GET request) with **a Content-Type of application/json** . Upon receiving the data, it automatically binds it to a Java object. The system uses `HttpMessageConverter` or customizes `HttpMessageConverter` the conversion of the JSON string in the request body into a Java object.

I will use a simple example to demonstrate the basic usage!

We have a registered interface:

```java
@PostMapping("/sign-up")
public ResponseEntity signUp(@RequestBody @Valid
UserRegisterRequest userRegisterRequest) {
  userService.save(userRegisterRequest);
  return ResponseEntity.ok().build();
}
```

`UserRegisterRequest` Target:

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserRegisterRequest {
    @NotBlank
    private String userName;
    @NotBlank
    private String password;
    @NotBlank
    private String fullName;
}
```

We send a post request to this endpoint, and the body carries JSON data:

```json
{ "userName": "coder", "fullName": "shuangkou", "password":
"123456" }
```

In this way, our backend can directly map the json format data to our `UserRegisterRequest` class.

Notice :

- A method can have only one `@RequestBody` parameter, but can have multiple `@PathVariable` and `@RequestParam` .
- If you need to receive multiple complex objects, it is recommended to merge them into a single object.

Data Verification

Data validation is crucial for ensuring system stability and security. Even if data validation has been implemented in the user interface (frontend), **backend services must still revalidate received data** . This is because frontend validation can be easily bypassed (for example, by modifying requests through developer tools or directly calling the API using HTTP tools like Postman or curl), potentially allowing malicious or erroneous data to be sent directly to the backend. Therefore, backend validation is the final and most important line of defense to prevent invalid data, maintain data consistency, and ensure the correct execution of business logic.

Bean Validation is a set of specifications (JSR 303, 349, 380) that define JavaBean parameter validation standards. It provides a series of annotations that can be directly applied to JavaBean properties to achieve convenient parameter validation.

- **JSR 303 (Bean Validation 1.0):** laid the foundation, introduced core validation annotations (such as `@NotNull` , `@Size` , `@Min` , `@Max` etc.), defined how to validate JavaBean properties through annotations, and supported nested object validation and custom validators.
- **JSR 349 (Bean Validation 1.1):** Expands upon 1.0, introducing support for method parameter and return value validation and enhancing the handling of group valida
- **JSR 380 (Bean Validation 2.0):** Embraces the new features of Java 8 and make some improvements, such as supporting `java.time` date and time types in the package

and introducing some new validation annotations (such as `@NotEmpty`, `@NotBlank` etc.).

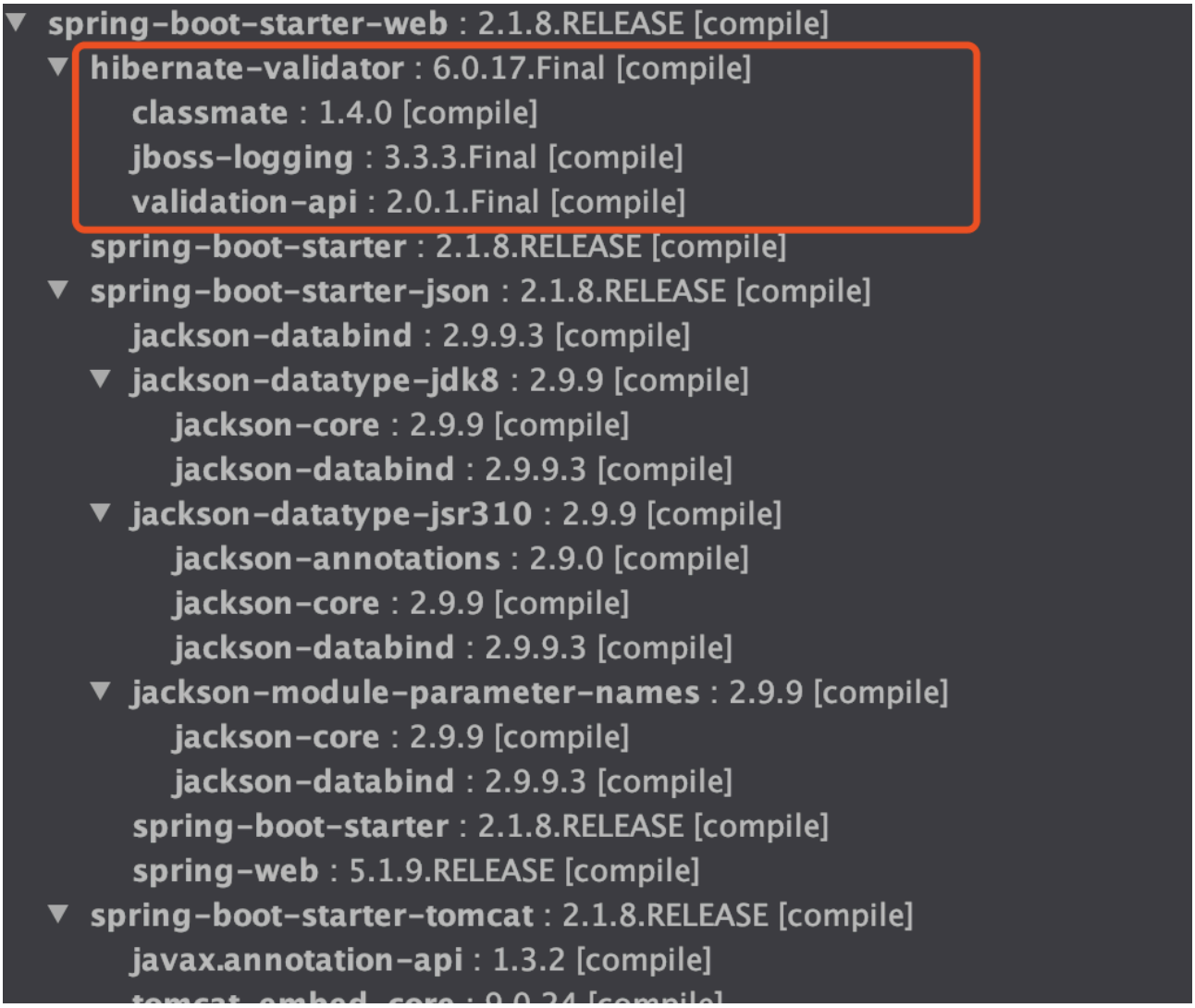
Bean Validation itself is just a set of **specifications (interfaces and annotations) . We need a concrete framework** that implements these specifications to execute the validation logic. Currently, **Hibernate Validator** is the most authoritative and widely used reference implementation of the Bean Validation specification.

- Hibernate Validator 4.x implements Bean Validation 1.0 (JSR 303).
- Hibernate Validator 5.x implements Bean Validation 1.1 (JSR 349).
- Hibernate Validator 6.x and higher implements Bean Validation 2.0 (JSR 380).

Using Bean Validation in a Spring Boot project is very convenient, thanks to Spring Boot's automatic configuration capabilities. Regarding dependency introduction, please note:

- In earlier versions of Spring Boot (usually before 2.3.x), `spring-boot-starter-web` hibernate-validator was included by default. Therefore, once you have included Web Starter, you do not need to add validation-related dependencies.
- Starting with Spring Boot 2.3.x, validation-related dependencies have been moved out of spring-boot-starter-web for more refined dependency management. If your project uses these or newer versions and requires Bean Validation functionality, you need to explicitly add `spring-boot-starter-validation` the dependency:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```



Non-SpringBoot projects need to introduce related dependency packages by themselves. I won't explain them here. For details, please refer to my article: How to do parameter validation in Spring/Spring Boot? Everything you need to know is here !

👉 Note: For all annotations, it is recommended to use JSR annotations, that is `javax.validation.constraints` , instead of `org.hibernate.validator.constraints`

Some commonly used field validation annotations

The Bean Validation specification and its implementations (such as Hibernate Validator) provide a rich set of annotations for declaratively defining validation rules. The following are some commonly used annotations and their descriptions:

- `@NotNull` : Checks that the annotated element (of any type) cannot be `null` .
- `@NotEmpty` : Check that the annotated element (such as `CharSequence` , `Collection` , `Map` , `Array`) cannot be `null` and its size/length cannot be 0. Note: For strings, `@NotEmpty` strings containing whitespace characters are allowed, such as `" "` .

- `@NotBlank` : Check that the annotated string `CharSequence` (such as `String`) cannot be `null` , and its length after removing the leading and trailing spaces must be greater than 0. (That is, it cannot be a blank string).
- `@Null` : Check that the annotated element must be `null` .
- `@AssertTrue` / `@AssertFalse` : Checks that the annotated `boolean` or `Boolean` type element must be `true` / `false` .
- `@Min(value)` / `@Max(value)` : Checks that the value of the annotated numeric type (or its string representation) must be greater than or equal to / less than or equal to the specified value `value` . Applicable to integer types (`byte` , `short` , `int` , `long` , `BigInteger` etc.).
- `@DecimalMin(value)` / `@DecimalMax(value)` : Functions similarly to `@Min` / `@Max` , but applies to numeric types that contain decimals (`BigDecimal` , `BigInteger` , `CharSequence` , `byte` , `short` , `int` , `long` and their wrappers). `value` Must be a string representation of a number.
- `@Size(min=, max=)` : Checks that the size/length of the annotated elements (such as `CharSequence` , `Collection` , `Map` , `Array`) must be within the specified `min` and `max` range (inclusive).
- `@Digits(integer=, fraction=)` : Check the value of the annotated numeric type (or its string representation). The number of digits in the integer part must be ≤ `integer` , and the number of digits in the decimal part must be ≤ `fraction` .
- `@Pattern(regexp=, flags=)` : Checks if the annotated string `CharSequence` (e.g. `String`) matches the specified regular expression (`regexp`). `flags` You can specify a matching pattern (e.g., case-insensitive).
- `@Email` : Check if the annotated `CharSequence` (eg `String`) conforms to the Email format (a relatively loose regular expression is built in).
- `@Past` / `@Future` : Checks whether the annotated date or time type (`java.util.Date` , , types under `java.util.Calendar` the JSR 310 package) is before/after the current time. `java.time`
- `@PastOrPresent` / `@FutureOrPresent` : Similar to `@Past` / `@Future` , but allows to be equal to the current time.
- ……

Verifying the Request Body

When a Controller method uses `@RequestBody` annotations to receive a request body and bind it to an object, you can add the annotation before the parameter `@Valid` to trigger validation of the object. If validation fails, it will throw an error `MethodArgumentNotValidException` .

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Person {
    @NotNull(message = "classId        ")
    private String classId;

    @Size(max = 33)
    @NotNull(message = "name        ")
    private String name;

    @Pattern(regexp = "((^Man$|^Woman$|^UGM$))", message = "sex
        ")
    @NotNull(message = "sex        ")
    private String sex;

    @Email(message = "email         ")
    @NotNull(message = "email        ")
    private String email;
}


@RestController
@RequestMapping("/api")
public class PersonController {
    @PostMapping("/person")
    public ResponseEntity<Person> getPerson(@RequestBody @Valid
Person person) {
        return ResponseEntity.ok().body(person);
    }
}
```

Verify request parameters (Path Variables and Request Parameters)

For simple types of data that map directly to method parameters (such as path variables `@PathVariable` or request parameters `@RequestParam`), the validation is slightly different:

1. **`@Validated` Add annotations to the Controller class** : This annotation is provided by Spring (not a JSR standard), which enables Spring to process method-level

parameter validation annotations. **This is a required step.**
2. **Place validation annotations directly on method parameters** : Apply validation annotations such as `@Min` , `@Max` , `@Size` , `@Pattern` etc. directly to the corresponding `@PathVariable` or `@RequestParam` parameters.

`@Validated` Don't forget to add the annotation to the class . This parameter can tell Spring to validate the method parameters.

```java
@RestController
@RequestMapping("/api")
@Validated //         1:              @Validated
public class PersonController {

    @GetMapping("/person/{id}")
    public ResponseEntity<Integer> getPersonByID(
            @PathVariable("id")
            @Max(value = 5, message = "ID         5") //         2:

            Integer id
    ) {
        //           id > 5  Spring
ConstraintViolationException
        //
        return ResponseEntity.ok().body(id);
    }

    @GetMapping("/person")
    public ResponseEntity<String> findPersonByName(
            @RequestParam("name")
            @NotBlank(message = "          ") //
@RequestParam
            @Size(max = 10, message = "              10")
            String name
    ) {
        return ResponseEntity.ok().body("Found person: " + name);
    }
}
```

Global exception handling

Let me introduce the global handling of Controller layer exceptions that are essential for our Spring project.

Related Notes:

1. `@ControllerAdvice` : Annotation definition of global exception handling class
2. `@ExceptionHandler` : Annotation declaration exception handling method

How to use it? Let's take the parameter validation in Section 5 as an example. If the method parameters are incorrect `MethodArgumentNotValidException` , an exception will be thrown. Let's handle this exception.

```java
@ControllerAdvice
@ResponseBody
public class GlobalExceptionHandler {

    /**
     *
     */
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?>
handleMethodArgumentNotValidException(MethodArgumentNotValidExceptior
ex, HttpServletRequest request) {
        ......
    }
}
```

For more information about Spring Boot exception handling, see my two articles:

1. Several common postures for SpringBoot to handle exceptions
2. Use enumeration to simply encapsulate an elegant Spring Boot global exception handling!

Affairs

`@Transactional` Just use annotations on the method where you want to start the transaction !

```java
@Transactional(rollbackFor = Exception.class)
public void save() {
  ......
}
```

We know that Exception is divided into runtime exception RuntimeException and non-runtime exception. `@Transactional` If no `rollbackFor` attribute is configured in the annotation, the transaction will only `RuntimeException` be rolled back when it encounters it. In addition `rollbackFor=Exception.class` , the transaction can be rolled back when a non-runtime exception is encountered.

`@Transactional` Annotations can generally be applied to or on.

- **Applied to classes** : When `@Transactional` the annotation is placed on a class, it indicates that all public methods of the class are configured with the same transaction attribute information.
- **Acting on methods** : When the class is configured `@Transactional` and the method is also configured `@Transactional` , the method's transaction will overwrite the class's transaction configuration information.

For more information about Spring transactions, please check out my article: Probably the most beautiful Spring transaction management explanation .

JPA

Spring Data JPA provides a set of annotations and features to help developers easily implement ORM (object-relational mapping).

Create a table

`@Entity` Used to declare a class as a JPA entity class, mapping it to a table in the database. `@Table` Specify the table name corresponding to the entity.

```java
@Entity
@Table(name = "role")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String description;

    //     getter/setter
}
```

Primary key generation strategy

`@Id` Declare a field as the primary key and `@GeneratedValue` specify the strategy for generating the primary key.

JPA provides four primary key generation strategies:

- **`GenerationType.TABLE`** : Generate a primary key through the database table.
- **`GenerationType.SEQUENCE`** : Generate primary keys through database sequences (applicable to databases such as Oracle).
- **`GenerationType.IDENTITY`** : Primary key auto-increment (applicable to databases such as MySQL).
- **`GenerationType.AUTO`** : JPA automatically selects the appropriate generation strategy (default strategy).

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

By `@GenericGenerator` declaring a custom primary key generation strategy:

```java
@Id
@GeneratedValue(generator = "IdentityIdGenerator")
@GenericGenerator(name = "IdentityIdGenerator", strategy =
"identity")
private Long id;
```

is equivalent to:

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

JPA provides the following primary key generation strategies:

```java
public class DefaultIdentifierGeneratorFactory
    implements MutableIdentifierGeneratorFactory, Serializable,
ServiceRegistryAwareService {

  @SuppressWarnings("deprecation")
  public DefaultIdentifierGeneratorFactory() {
    register( "uuid2", UUIDGenerator.class );
    register( "guid", GUIDGenerator.class );      // can be done
with UUIDGenerator + strategy
    register( "uuid", UUIDHexGenerator.class );      //
"deprecated" for new use
    register( "uuid.hex", UUIDHexGenerator.class );   // uuid.hex
is deprecated
    register( "assigned", Assigned.class );
    register( "identity", IdentityGenerator.class );
    register( "select", SelectGenerator.class );
    register( "sequence", SequenceStyleGenerator.class );
    register( "seqhilo", SequenceHiLoGenerator.class );
    register( "increment", IncrementGenerator.class );
    register( "foreign", ForeignGenerator.class );
    register( "sequence-identity", SequenceIdentityGenerator.class
);
    register( "enhanced-sequence", SequenceStyleGenerator.class );
    register( "enhanced-table", TableGenerator.class );
  }

  public void register(String strategy, Class generatorClass) {
    LOG.debugf( "Registering IdentifierGenerator strategy [%s] ->
[%s]", strategy, generatorClass.getName() );
    final Class previous = generatorStrategyToClassNameMap.put(
strategy, generatorClass );
    if ( previous != null ) {
```

```
        LOG.debugf( "    - overriding [%s]", previous.getName() );
      }
    }

}
```

Field Mapping

`@Column` Used to specify the mapping relationship between entity fields and database columns.

- **`name`** : Specify the database column name.
- **`nullable`** : Specifies whether to allow `null`.
- **`length`** : Set the length of the field (only for `String` type).
- **`columnDefinition`** : Specifies the database type and default value of the field.

```java
@Column(name = "user_name", nullable = false, length = 32)
private String userName;

@Column(columnDefinition = "tinyint(1) default 1")
private Boolean enabled;
```

Ignore Fields

`@Transient` Used to declare fields that do not need to be persisted.

```java
@Entity
public class User {

    @Transient
    private String temporaryField; //
}
```

Other fields that are not persisted:

- **`static`** : Static fields will not be persisted.
- **`final`** : Final fields will not be persisted.
- **`transient`** : Fields declared using Java's `transient` keyword will not be serializ persisted.

Large field storage

`@Lob` Used to declare large fields (such as `CLOB` or `BLOB`).

```java
@Lob
@Column(name = "content", columnDefinition = "LONGTEXT NOT NULL")
private String content;
```

Enumeration type mapping

`@Enumerated` Used to map enumeration types to database fields.

- **`EnumType.ORDINAL`** : Stores the ordinal number of the enumeration (default).
- **`EnumType.STRING`** : Stores the name of the enumeration (recommended).

```java
public enum Gender {
    MALE,
    FEMALE
}

@Entity
public class User {

    @Enumerated(EnumType.STRING)
    private Gender gender;
}
```

The values stored in the database `MALE` are or `FEMALE` .

Audit function

Through the audit function of JPA, information such as creation time, update time, creator and updater can be automatically recorded in the entity.

Audit base class:

```java
@Data
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class AbstractAuditBase {

    @CreatedDate
    @Column(updatable = false)
    private Instant createdAt;

    @LastModifiedDate
    private Instant updatedAt;

    @CreatedBy
    @Column(updatable = false)
    private String createdBy;

    @LastModifiedBy
    private String updatedBy;
}
```

Configure the audit function:

```java
@Configuration
@EnableJpaAuditing
public class AuditConfig {

    @Bean
    public AuditorAware<String> auditorProvider() {
        return () ->
Optional.ofNullable(SecurityContextHolder.getContext())
                .map(SecurityContext::getAuthentication)
                .filter(Authentication::isAuthenticated)
                .map(Authentication::getName);
    }
}
```

Briefly introduce some of the annotations involved above:

1. `@CreatedDate` : Indicates that this field is a creation time field. When this entity is inserted, the value will be set.
2. `@CreatedBy` : Indicates that this field is the creator. When this entity is inserted, the value will be set `@LastModifiedDate` . `@LastModifiedBy` Similarly.

3. `@EnableJpaAuditing`：Enable JPA auditing.

Modify and delete operations

`@Modifying` Annotations are used to identify modification or deletion operations and must `@Transactional` be used together with .

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Modifying
    @Transactional
    void deleteByUserName(String userName);
}
```

Association

JPA provides four types of relationship annotations:

- **`@OneToOne`**：One-to-one relationship.
- **`@OneToMany`**：One-to-many relationship.
- **`@ManyToOne`**：Many-to-one relationship.
- **`@ManyToMany`**：Many-to-many relationship.

```java
@Entity
public class User {

    @OneToOne
    private Profile profile;

    @OneToMany(mappedBy = "user")
    private List<Order> orders;
}
```

JSON data processing

In web development, converting Java objects to and from JSON is a common task. Sp often integrates the Jackson library to accomplish this. Here are some common Jackson annotations that can help customize JSON serialization (Java object to JSON) and

deserialization (JSON to Java object) processes.

Filtering JSON fields

Sometimes we don't want certain fields of a Java object to be included in the final generated JSON, or we don't want certain JSON properties to be processed when converting JSON to a Java object.

`@JsonIgnoreProperties` It is used on the class to filter out specific fields and not return or parse them.

```java
//       JSON       userRoles
//                   JSON                              ignoreUnknown
= true
@JsonIgnoreProperties({"userRoles"})
public class User {
    private String userName;
    private String fullName;
    private String password;
    private List<UserRole> userRoles = new ArrayList<>();
    // getters and setters...
}
```

`@JsonIgnore` Acts on the field or `getter/setter` method level and is used to specify that a particular property should be ignored during serialization or deserialization.

```java
public class User {
    private String userName;
    private String fullName;
    private String password;

    //       JSON       userRoles
    @JsonIgnore
    private List<UserRole> userRoles = new ArrayList<>();
    // getters and setters...
}
```

`@JsonIgnoreProperties` It is more suitable for explicitly excluding multiple fields in class definition, or excluding fields in inheritance scenarios; `@JsonIgnore` it is more direc used to mark a single specific field.

Formatting JSON data

`@JsonFormat` Used to specify the format of the attribute during serialization and deserialization. Commonly used for formatting date and time types.

for example:

```java
//     Date            ISO 8601                        GMT
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", timezone = "GMT")
private Date date;
```

Flattening JSON objects

`@JsonUnwrapped` The annotation acts on the field and is used to "promote" the properties of its nested objects to the level of the current object during serialization, and perform the opposite operation during deserialization. This can make the JSON structure flatter.

Suppose there is `Account` a class containing two nested objects `Location` and `PersonInfo`.

```java
@Getter
@Setter
@ToString
public class Account {
    private Location location;
    private PersonInfo personInfo;

  @Getter
  @Setter
  @ToString
  public static class Location {
     private String provinceName;
     private String countyName;
  }
  @Getter
  @Setter
  @ToString
```

```
  public static class PersonInfo {
    private String userName;
    private String fullName;
  }
}
```

JSON structure before flattening:

```json
{
  "location": {
    "provinceName": "    ",
    "countyName": "    "
  },
  "personInfo": {
    "userName": "coder1234",
    "fullName": "shaungkou"
  }
}
```

Use `@JsonUnwrapped` Flatten Object:

```
@Getter
@Setter
@ToString
public class Account {
    @JsonUnwrapped
    private Location location;
    @JsonUnwrapped
    private PersonInfo personInfo;
    ......
}
```

The flattened JSON structure:

```json
{
  "provinceName": "    ",
  "countyName": "    ",
  "userName": "coder1234",
  "fullName": "shaungkou"
}
```

test

`@ActiveProfiles` Generally acts on the test class and is used to declare the effective Spring configuration file.

```java
//       RANDOM_PORT                          "test" profile
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles("test")
@Slf4j
public abstract class TestBase {
    // Common test setup or abstract methods...
}
```

`@Test` It is an annotation provided by the JUnit framework (usually JUnit 5 Jupiter) to mark a method as a test method. Although it is not a Spring annotation, it is the basis for performing unit tests and integration tests.

`@Transactional` The data of the declared test method will be rolled back to avoid polluting the test data.

`@WithMockUser` This annotation is provided by the Spring Security Test module and is used to simulate an authenticated user during testing. It allows you to easily specify username, password, roles (authorities), and other information to test secured endpoints or methods.

```java
public class MyServiceTest extends TestBase { // Assuming TestBase
provides Spring context

    @Test
    @Transactional //
    @WithMockUser(username = "test-user", authorities = {
"ROLE_TEACHER", "read" }) //            "test-user"      TEACHER
     read
    void should_perform_action_requiring_teacher_role() throws
```

```
Exception {
        // ...          ...
        //                 "ROLE_TEACHER"
    }
}
```


Recently Updated2025/4/25 06:00

Contributors: shuang.kou , AethLi , guide , ipofss , TommyMerlin , anaer , East Third Ring Road Niu Ge , yellowgg , yzqdev , Guide , Mr.Hope , aucub