Theory: Bean Validation

19 minutes 0 / 5 problems solved

§1. Data validation

As you have learned, in web-based applications, a client can communicate with the server. It means that a client can request data from a server as well as send data to it. The question is: what if a user sends data that violates the business logic of the application? For example, the user can specify a negative age or empty name while filling a registration form. As it could lead to unexpected errors, we need to prevent a situation where a user sends data that violates **specified constraints**. In cases like that, we can use annotations from the `javax.validation` package together with Spring Boot annotations.

Here's an example. Imagine you are creating a web application for special agents registration, so now users can send data to the server using the `POST` HTTP request method with a request body that contains data about special agents:

```
{
    "name": "James",
    "surname": "Bond",
    "code": "007",
    "status": "special agent",
    "age": 51
}
```

Data in the request body represents a POJO class `SpecialAgent`:

```
public class SpecialAgent {

    private String name;
    private String surname;
    private String code;
    private String status;
    private int age;

    // getters and setters

}
```

Now we can set constraints for the fields of the `SpecialAgent` class using annotations from the `javax.validation` package.

§2. @NotNull, @NotEmpty, @NotBlank constraints

`@NotNull` annotation is one of the most popular constraint annotations, and it declares that a *field cannot be null*.

> All constraint annotations should be placed on top of the data class fields.

Let's annotate a special agent's name with `@NotNull`:

```
@NotNull
private String name;
```

Now special agent's data that we will send to the server should contain a field `name` and should not be `null`. Besides `@NotNull` annotation, we can use `@NotEmpty` and `@NotBlank` annotations. `@NotEmpty` annotation declares that an annotated field *should not be null*, and the length of the field value should be *greater than 0*. `@NotBlank` declares that annotated field value should *not be empty after trimming.* Let's have a look at some examples:

```
@NotEmpty
private String motto;

@NotBlank
private String status;
```

The values of the `motto` and `status` fields cannot be `null` and cannot be empty. However, `motto` value can be equal to `" "` unlike `status`, which trimmed value cannot be empty.

§3. @Size constraint

The next useful constraints are provided by `@Size` annotation. This annotation has `"min = "` and `"max = "` parameters that specify the *boundaries of the field* (constrain the length).

```
@Size(min = 1, max = 3)
private String code;
```

Now, a special agent's `code` can contain from one to three symbols. `@Size` annotation can be used not only for `String` fields but also for `Collection` fields. It specifies the *minimum and the maximum number of elements in the collection*. For example, let's constrain the number of special agent's cars to be from 0 to 4:

```
@Size(min = 0, max = 4)
private List<String> cars;
```

§4. @Min, @Max constraints

If we want to set boundaries for the *numeric value*, we can use `@Min` and `@Max` annotations with the `"value = "` parameter. We can omit the `"value = "` parameter name and specify the integer number only:

```
@Min(value = 18)
private int age;

@Max(5)
private int numberOfCurrentMissions;
```

Now a special agents' minimum age is 18, and they cannot have more than 5 current missions.

§5. @Pattern and @Email

Another useful annotation is `@Pattern` that constrains the value of the annotated field *to match the regular expression* defined in the `"regexp = "` parameter. For example, we would like to specify that a special agent's `code` can contain from 1 to 3 digits only:

```
@Pattern(regexp = "[0-9]{1,3}")
private String code;
```

Let's add an `email` field to the `SpecialAgent` class. We know that email address is made up of a local-part, an @ symbol, then a case-insensitive domain. We can write a regular expression for the email address validation or use a special case of the `@Pattern` , `@Email` annotation, which approves that the annotated property is a *valid email address*.

```
@NotNull
@Email
private String email;
```

§6. @Valid annotation

Now we have a `SpecialAgent` class with fields that are constrained by annotations. To allow a client to communicate with a server, we can create a REST Controller with annotated `@PostMapping` methods and `@RequestBody` parameters. However, we need to add `@Valid` annotation from `javax.validation` package to the **request body** parameter to "tell" Spring Boot that request body must be **validated** according to the specified annotations. Without this annotation, `SpecialAgent` class properties will not be validated.

```
@RestController
public class SpecialAgentController {

    @PostMapping("/agent")

 public ResponseEntity<String> validate(@Valid @RequestBody SpecialAgent agent)
{
          return ResponseEntity.ok("Agent info is valid.");
    }
}
```

Great, we've annotated the `agent` parameter of the POST method by `@Valid` annotation, so now the agent's data will be validated. What happens if we send data that violates the constraints? The server will return an HTTP response with a `400 Bad Request` status and the body that contains a field `"defaultMessage"` with a description of the violated constraint. However, we can customize this message by specifying the `"message = "` parameter of the constraint annotation.

> Any validation annotation has a `"message = "` parameter that can be used to display validation failures messages.

For example, let's add a `"message = "` to the `@Min` annotation of the field `age`:

```
@Min(value = 18, message = "Age must be greater than or equal to 18")
private int age;
```

If we try to send special agent's data and specify the value of the field `age` lower than 18, our application will return a response with a `400 Bad Request` status and the body that contains a field `"defaultMessage": "Age must be greater than or equal to 18"`.

§7. @Validated annotation

We can validate not only request body but also **path variables** and **request parameters**. To do so, we should annotate the REST Controller class with `@Validated` annotation. Now we can use the same annotations that were already described together with the `@PathVariable` or `@RequestParam` annotations. Here's an example:

```
@RestController
@Validated
public class SpecialAgentController {

    @GetMapping("/agents/{id}")
    ResponseEntity<String> validateAgentPathVariable(
            @PathVariable("id") @Min(1) int id) {
        return ResponseEntity.ok("Agent id is valid.");
    }



    @GetMapping("/agents")

    ResponseEntity<String> validateAgentRequestParam(

            @RequestParam("code") @Pattern(regexp = "[0-9]
{1,3}") String code) {

        return ResponseEntity.ok("Agent code is valid.");

    }

}
```

We have constrained a path variable `id` with a `@Min(1)` annotation. It means that `id` cannot be lower than 1. Besides a path variable, we have constrained a request parameter `code` with a `@Pattern(regexp = "[0-9]{1,3}")` annotation. It means that `code` can consist of 1 to 3 digits only.

§8. Conclusion

Now you know how to validate data that a client sends to the server. To constrain the class field, we need to annotate it with annotations from the `javax.validation` package. To validate sent data following specified constraints, we need to add the `@Valid` annotation to the request body parameter of the corresponding `POST` method in the controller class. We can validate not only request body but also path variables and request parameters. To do so, we can add the `@Validated` annotation to the REST Controller class and use any constraint annotation from the `javax.validation` package together with the path variable or request parameter of the method.

Report a typo

55 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (5) Hints (1) Useful links (0) Show discussion