Validation in Spring Boot

Last modified: January 30, 2021

by Alejandro Ugarte

Spring Boot

1. Overview

When it comes to validating user input, Spring Boot provides strong support for this common, yet critical, task straight out of the box.

Although Spring Boot supports seamless integration with custom validators, **the de-facto standard for performing validation is Hibernate Validator**, the Bean Validation framework's reference implementation.

In this tutorial, **we'll look at how to validate domain objects in Spring Boot**.

2. The Maven Dependencies

In this case, we'll learn how to validate domain objects in Spring Boot **by building a basic REST controller.**

The controller will first take a domain object, then it will validate it with Hibernate Validator, and finally it will persist it into an in-memory H2 database.

The project's dependencies are fairly standard:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.197</version>
    <scope>runtime</scope>
</dependency>
```

As shown above, we included *spring-boot-starter-web* in our *pom.xml* file because we'll need it for creating the REST controller. Additionally, let's make sure to check the latest versions of *spring-boot-starter-jpa* and the H2 database on Maven Central.

Starting with Boot 2.3, we also need to explicitly add the *spring-boot-starter-validation*dependency:

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

3. A Simple Domain Class

With our project's dependencies already in place, next we need to define an example JPA entity class, whose role will solely be modelling users.

Let's have a look at this class:

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @NotBlank(message = "Name is mandatory")
    private String name;

    @NotBlank(message = "Email is mandatory")
    private String email;

    // standard constructors / setters / getters / toString

}
```

The implementation of our *User* entity class is pretty anemic indeed, but it shows in a nutshell how to use Bean Validation's constraints to constrain the *name* and *email* fields.

For simplicity's sake, we constrained the target fields using only the *@NotBlank* constraint. Also, we specified the error messages with the *message* attribute.

Therefore, when Spring Boot validates the class instance, the constrained fields **must be not null and their trimmed length must be greater than zero**.

Additionally, Bean Validation provides many other handy constraints besides *@NotBlank*. This allows us to apply and combine different validation rules to the constrained classes. For further information, please read the official bean validation docs.

Since we'll use Spring Data JPA for saving users to the in-memory H2 database, we also need to define a simple repository interface for having basic CRUD functionality on *User* objects:

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {}
```

4. Implementing a REST Controller

Of course, we need to implement a layer that allows us to get the values assigned to our *User* object's constrained fields.

Therefore, we can validate them and perform a few further tasks, depending on the validation results.

Spring Boot makes **this seemingly complex process really simple** through the implementation of a REST controller.

Let's look at the REST controller implementation:

```
@RestController
public class UserController {

    @PostMapping("/users")
    ResponseEntity<String> addUser(@Valid @RequestBody User user) {
        // persisting the user
        return ResponseEntity.ok("User is valid");
    }

    // standard constructors / other methods

}
```

In a Spring REST context, the implementation of the *addUser()* method is fairly standard.

Of course, the most relevant part is the use of the *@Valid* annotation.

When Spring Boot finds an argument annotated with *@Valid*, it automatically bootstraps the default JSR 380 implementation — Hibernate Validator — and validates the argument.

When the target argument fails to pass the validation, Spring Boot throws a *MethodArgumentNotValidException*exception.

5. The *@ExceptionHandler* Annotation

While it's really handy to have Spring Boot validating the *User* object passed on to the *addUser()* method automatically, the missing facet of this process is how we process the validation results.

The *@ExceptionHandler*annotation **allows us to handle specified types of exceptions through one single method.**

Therefore, we can use it for processing the validation errors:

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MethodArgumentNotValidException.class)
public Map<String, String> handleValidationExceptions(
  MethodArgumentNotValidException ex) {
    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getAllErrors().forEach((error) -> {
        String fieldName = ((FieldError) error).getField();
        String errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    });
    return errors;
}
```

We specified the *MethodArgumentNotValidException* exception as the exception to be handled. Consequently, Spring Boot will call this method **when the specified *User* object is invalid**.

The method stores the name and post-validation error message of each invalid field in a *Map*. Next it sends the *Map* back to the client as a JSON representation for further processing.

Simply put, the REST controller allows us to easily process requests to different endpoints, validate *User* objects, and send the responses in JSON format.

The design is flexible enough to handle controller responses through several web tiers, ranging from template engines such as Thymeleaf, to a full-featured JavaScript framework such as Angular.

6. Testing the REST Controller

We can easily test the functionality of our REST controller with an integration test.

Let's start mocking/autowiring the *UserRepository* interface implementation, along with the *UserController* instance, and a *MockMvc* object:

```
@RunWith(SpringRunner.class)
@WebMvcTest
@AutoConfigureMockMvc
public class UserControllerIntegrationTest {

    @MockBean
    private UserRepository userRepository;

    @Autowired
    UserController userController;

    @Autowired
    private MockMvc mockMvc;

    //...

}
```

Since we're only testing the web layer, we use the *@WebMvcTest* annotation. It allows us to easily test requests and responses using the set of static methods implemented by the *MockMvcRequestBuilders* and *MockMvcResultMatchers* classes.

Now let's test the *addUser()* method with a valid and an invalid *User* object passed in the request body:

```
@Test
public void whenPostRequestToUsersAndValidUser_thenCorrectResponse() throws
Exception {
    MediaType textPlainUtf8 = new MediaType(MediaType.TEXT_PLAIN,
Charset.forName("UTF-8"));
    String user = "{\"name\": \"bob\", \"email\" : \"bob@domain.com\"}";
    mockMvc.perform(MockMvcRequestBuilders.post("/users")
      .content(user)
      .contentType(MediaType.APPLICATION_JSON_UTF8))
      .andExpect(MockMvcResultMatchers.status().isOk())
      .andExpect(MockMvcResultMatchers.content()
        .contentType(textPlainUtf8));
}

@Test
public void whenPostRequestToUsersAndInValidUser_thenCorrectResponse() throws
Exception {
    String user = "{\"name\": \"\", \"email\" : \"bob@domain.com\"}";
    mockMvc.perform(MockMvcRequestBuilders.post("/users")
      .content(user)
      .contentType(MediaType.APPLICATION_JSON_UTF8))
      .andExpect(MockMvcResultMatchers.status().isBadRequest())
      .andExpect(MockMvcResultMatchers.jsonPath("$.name", Is.is("Name is
mandatory")))
      .andExpect(MockMvcResultMatchers.content()
        .contentType(MediaType.APPLICATION_JSON_UTF8));
    }
}
```

In addition, we can test the REST controller API **using a free API life cycle testing application**, such as Postman.

7. Running the Sample Application

Finally, we can run our example project with a standard *main()* method:

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public CommandLineRunner run(UserRepository userRepository) throws Exception
{
        return (String[] args) -> {
            User user1 = new User("Bob", "bob@domain.com");
            User user2 = new User("Jenny", "jenny@domain.com");
            userRepository.save(user1);
            userRepository.save(user2);
            userRepository.findAll().forEach(System.out::println);
        };
    }
}
```

As expected, we should see a couple of *User* objects printed out in the console.

A POST request to the http://localhost:8080/usersendpoint with a valid *User* object will return the *String* “User is valid”.

Likewise, a POST request with a *User* object without *name* and *email* values will return the following response:

```
{
  "name":"Name is mandatory",
  "email":"Email is mandatory"
}
```

8. Conclusion

In this article, **we learned the basics of performing validation in Spring Boot**.

As usual, all the examples shown in this article are available over on GitHub.