Complete Guide to Validation With Spring Boot

Bean Validation is the de-facto standard for implementing validation logic in the Java ecosystem. It's well integrated with Spring and Spring Boot.

However, there are some pitfalls. This tutorial goes over all major validation use cases and sports code examples for each.

Code Example

This article is accompanied by a working code example on GitHub.

Setting Up Validation

Spring Boot's Bean Validation support comes with the validation starter, which we can include into our project (Gradle notation):

```
implementation('org.springframework.boot:spring-boot-starter-validation')
```

It's not necessary to add the version number since the Spring Dependency Management Gradle plugin does that for us. If you're not using the plugin, you can find the most recent version here.

However, if we have also included the web starter, the validation starter comes for free:

```
implementation('org.springframework.boot:spring-boot-starter-web')
```

Note that the validation starter does no more than adding a dependency to a compatible version ofhibernate validator, which is the most widely used implementation of the Bean Validation specification.

Bean Validation Basics

Very basically, Bean Validation works by defining constraints to the fields of a class by annotating them with certain annotations.

Then, you pass an object of that class into a Validator which checks if the constraints are satisfied.

We'll see more details in the examples below.

Validating Input to a Spring MVC Controller

Let's say we have implemented a Spring REST controller and want to validate the input that' passed in by a client. There are three things we can validate for any incoming HTTP request:

- the request body,
- variables within the path (e.g. `id` in `/foos/{id}`) and,
- query parameters.

Let's look at each of those in more detail.

Validating a Request Body

In POST and PUT requests, it's common to pass a JSON payload within the request body. Spring automatically maps the incoming JSON to a Java object. Now, we want to check if the incoming Java object meets our requirements.

This is our incoming payload class:

```
class Input {

  @Min(1)
  @Max(10)
  private int numberBetweenOneAndTen;

  @Pattern(regexp = "^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}$")
  private String ipAddress;

  // ...
}
```

We have an `int` field that must have a value between 1 and 10, inclusively, and a `String` field that must contain an IP address (the regex actually still allows invalid IP addresses with octets greater than 255, but we're fixing that later in the tutorial).

Here's the REST controller that takes an `Input` object in the request body and validates it:

```
@RestController
class ValidateRequestBodyController {

 @PostMapping("/validateBody")
 ResponseEntity<String> validateBody(@Valid @RequestBody Input input) {
  return ResponseEntity.ok("valid");
 }

}
```

We simply have added the `@Valid` annotation to the `Input` parameter, which is also annotated with `@RequestBody` to mark that it should be read from the request body. By doing this, we're telling Spring to pass the object to a `Validator` before doing anything else.

Use `@Valid` on Complex Types

If the `Input` class contains a field with another complex type that should be validated, this field, too, needs to be annotated with `@Valid`.

If the validation fails, it will trigger a `MethodArgumentNotValidException`. By default, Spring will translate this exception to a HTTP status 400 (Bad Request).

We can verify this behavior with an integration test:

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(controllers = ValidateRequestBodyController.class)
class ValidateRequestBodyControllerTest {

 @Autowired
 private MockMvc mvc;

 @Autowired
 private ObjectMapper objectMapper;

 @Test
 void whenInputIsInvalid_thenReturnsStatus400() throws Exception {
  Input input = invalidInput();
  String body = objectMapper.writeValueAsString(input);

  mvc.perform(post("/validateBody")
        .contentType("application/json")
        .content(body))
        .andExpect(status().isBadRequest());
 }
}
```

You can find more details about testing Spring MVC controllers in my article about the `@WebMvcTest` annotation.

Validating Path Variables and Request Parameters

Validating path variables and request parameters works a little differently.

We're not validating complex Java objects in this case, since path variables and request parameters are primitive types like `int` or their counterpart objects like `Integer` or `String`.

Instead of annotating a class field like above, we're adding a constraint annotation (in this case `@Min`) directly to the method parameter in the Spring controller:

```
@RestController
@Validated
class ValidateParametersController {

 @GetMapping("/validatePathVariable/{id}")
 ResponseEntity<String> validatePathVariable(
    @PathVariable("id") @Min(5) int id) {
  return ResponseEntity.ok("valid");
 }

 @GetMapping("/validateRequestParameter")
 ResponseEntity<String> validateRequestParameter(
    @RequestParam("param") @Min(5) int param) {
  return ResponseEntity.ok("valid");
 }
}
```

Note that we have to add Spring's `@Validated` annotation to the controller at class level to tell Spring to evaluate the constraint annotations on method parameters.

The `@Validated` annotation is only evaluated on class level in this case, even though it's allowed to be used on methods (we'll learn why it's allowed on method level when discussing validation groups later).

In contrast to request body validation a failed validation will trigger a `ConstraintViolationException` instead of a `MethodArgumentNotValidException`. Spring does not register a default exception handler for this exception, so it will by default cause a response with HTTP status 500 (Internal Server Error).

If we want to return a HTTP status 400 instead (which makes sense, since the client provided an invalid parameter, making it a bad request), we can add a custom exception handler to our contoller:

```
@RestController
@Validated
class ValidateParametersController {

 // request mapping method omitted

 @ExceptionHandler(ConstraintViolationException.class)
 @ResponseStatus(HttpStatus.BAD_REQUEST)
 ResponseEntity<String> handleConstraintViolationException(ConstraintViolationException e) {
  return new ResponseEntity<>("not valid due to validation error: " + e.getMessage(), HttpStatus.BAD_REQUEST);
 }

}
```

Later in this tutorial we will look at how to return a structured error response that contains details on all failed validations for the client to inspect.

We can verify the validation behavior with an integration test:

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(controllers = ValidateParametersController.class)
class ValidateParametersControllerTest {

 @Autowired
 private MockMvc mvc;

 @Test
 void whenPathVariableIsInvalid_thenReturnsStatus400() throws Exception {
  mvc.perform(get("/validatePathVariable/3"))
        .andExpect(status().isBadRequest());
 }

 @Test
 void whenRequestParameterIsInvalid_thenReturnsStatus400() throws Exception {
  mvc.perform(get("/validateRequestParameter")
        .param("param", "3"))
        .andExpect(status().isBadRequest());
 }

}
```

Validating Input to a Spring Service Method

Instead of (or additionally to) validating input on the controller level, we can also validate the input to any Spring components. In order to to this, we use a combination of the `@Validated` and `@Valid` annotations:

```
@Service
@Validated
class ValidatingService{

  void validateInput(@Valid Input input){
    // do something
  }

}
```

Again, the `@Validated` annotation is only evaluated on class level, so don't put it on a method in this use case.

Here's a test verifying the validation behavior:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class ValidatingServiceTest {

 @Autowired
 private ValidatingService service;

 @Test
 void whenInputIsInvalid_thenThrowsException(){
  Input input = invalidInput();

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateInput(input);
  });
 }

}
```

Validating JPA Entities

The last line of defense for validation is the persistence layer. By default, Spring Data uses Hibernate underneath, which supports Bean Validation out of the box.

Is the Persistence Layer the right Place for Validation?

We usually don't want to do validation as late as in the persistence layer because it means that the business code above has worked with potentially invalid objects which may lead to unforeseen errors. More on this topic in my article about Bean Validation anti-patterns.

Let's say want to store objects of our `Input` class to the database. First, we add the necessary JPA annotation `@Entity` and add an ID field:

```
@Entity
public class Input {

  @Id
  @GeneratedValue
  private Long id;

  @Min(1)
  @Max(10)
  private int numberBetweenOneAndTen;

  @Pattern(regexp = "^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}$")
  private String ipAddress;

  // ...

}
```

Then, we create a Spring Data repository that provides us with methods to persist and query for `Input` objects:

```
public interface ValidatingRepository extends CrudRepository<Input, Long> {}
```

By default, any time we use the repository to store an `Input` object whose constraint annotations are violated, we'll get a `ConstraintViolationException` as this integration test demonstrates:

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
class ValidatingRepositoryTest {

  @Autowired
  private ValidatingRepository repository;

  @Autowired
  private EntityManager entityManager;

  @Test
  void whenInputIsInvalid_thenThrowsException() {
    Input input = invalidInput();

    assertThrows(ConstraintViolationException.class, () -> {
      repository.save(input);
      entityManager.flush();
    });
  }

}
```

You can find more details about testing Spring Data repositories in my article about the `@DataJpaTest` annotation.

Note that Bean Validation is only triggered by Hibernate once the `EntityManager` is flushed. Hibernate flushes thes `EntityManager` automatically under certain circumstances, but in the case of our integration test we have to do this by hand.

If for any reason we want to disable Bean Validation in our Spring Data repositories, we can set the Spring Boot property `spring.jpa.properties.javax.persistence.validation.mode` to `none`.

Implementing A Custom Validator

If the available constraint annotations do not suffice for our use cases, we might want to create one ourselves.

In the `Input` class from above, we used a regular expression to validate that a String is a valid IP address. However, the regular expression is not complete: it allows octets with values greater than 255 (i.e. “111.111.111.333” would be considered valid).

Let’s fix this by implementing a validator that implements this check in Java instead of with a regular expression (yes, I know that we could just use a more complex regular expression to achieve the same result, but we like to implement validations in Java, don’t we?).

First, we create the custom constraint annotation `IpAddress`:

```
@Target({ FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = IpAddressValidator.class)
@Documented
public @interface IpAddress {

  String message() default "{IpAddress.invalid}";

  Class<?>[] groups() default { };

  Class<? extends Payload>[] payload() default { };

}
```

A custom constraint annotation needs all of the following:

- the parameter `message`, pointing to a property key in `ValidationMessages.properties`, which is used to resolve a message in case of violation,
- the parameter `groups`, allowing to define under which circumstances this validation is to be triggered (we’re going to talk about validation groups later),
- the parameter `payload`, allowing to define a payload to be passed with this validation (since this is a rarely used feature, we’ll not cover it in this tutorial), and
- a `@Constraint` annotation pointing to an implementation of the `ConstraintValidator` interface.

The validator implementation looks like this:

```
class IpAddressValidator implements ConstraintValidator<IpAddress, String> {

  @Override
  public boolean isValid(String value, ConstraintValidatorContext context) {
    Pattern pattern =
      Pattern.compile("^([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})$");
    Matcher matcher = pattern.matcher(value);
    try {
      if (!matcher.matches()) {
        return false;
      } else {
        for (int i = 1; i <= 4; i++) {
          int octet = Integer.valueOf(matcher.group(i));
          if (octet > 255) {
            return false;
          }
        }
        return true;
      }
    } catch (Exception e) {
      return false;
    }
  }
}
```

We can now use the `@IpAddress` annotation just like any other constraint annotation:

```
class InputWithCustomValidator {

  @IpAddress
  private String ipAddress;

  // ...

}
```

Validating Programmatically

There may be cases when we want to invoke validation programmatically instead of relying on Spring's built-in Bean Validation support.

In this case, **we can just create a `Validator` by hand** and invoke it to trigger a validation:

```
class ProgrammaticallyValidatingService {

 void validateInput(Input input) {
  ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
  Validator validator = factory.getValidator();
  Set<ConstraintViolation<Input>> violations = validator.validate(input);
  if (!violations.isEmpty()) {
   throw new ConstraintViolationException(violations);
  }
 }

}
```

This requires no Spring support whatsoever.

However, Spring Boot provides us with a pre-configured `Validator` instance. We can inject this instance into our service and use this instance instead of creating one by hand:

```
@Service
class ProgrammaticallyValidatingService {

 private Validator validator;

 ProgrammaticallyValidatingService(Validator validator) {
  this.validator = validator;
 }

 void validateInputWithInjectedValidator(Input input) {
  Set<ConstraintViolation<Input>> violations = validator.validate(input);
  if (!violations.isEmpty()) {
   throw new ConstraintViolationException(violations);
  }
 }
}
```

When this service is instantiated by Spring, it will automatically have a `Validator` instance injected into the constructor.

The following unit test proves that both methods above work as expected:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class ProgrammaticallyValidatingServiceTest {

 @Autowired
 private ProgrammaticallyValidatingService service;

 @Test
 void whenInputIsInvalid_thenThrowsException(){
  Input input = invalidInput();

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateInput(input);
  });
 }

 @Test
 void givenInjectedValidator_whenInputIsInvalid_thenThrowsException(){
  Input input = invalidInput();

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateInputWithInjectedValidator(input);
  });
 }

}
```

Using Validation Groups to Validate Objects Differently for Different Use Cases

Often, certain objects are shared between different use cases.

Let's take the typical CRUD operations, for example: the "Create" use case and the "Update" use case will most probably both take the same object type as input. However, there may be validations that should be triggered under different circumstances:

- only in the "Create" use case,
- only in the "Update" use case, or
- in both use cases.

The Bean Validation feature that allows us to implement validation rules like this is called "Validation Groups".

We have already seen that all constraint annotations must have a `groups` field. This can be used to pass any classes that each define a certain validation group that should be triggered.

For our CRUD example, we simply define two marker interfaces `OnCreate` and `OnUpdate`:

```
interface OnCreate {}

interface OnUpdate {}
```

We can then use these marker interfaces with any constraint annotation like this:

```
class InputWithGroups {

  @Null(groups = OnCreate.class)
  @NotNull(groups = OnUpdate.class)
  private Long id;

  // ...

}
```

This will make sure that the ID is empty in our “Create” use case and that it’s not empty in our “Update” use case.

Spring supports validation groups with the `@Validated` annotation:

```
@Service
@Validated
class ValidatingServiceWithGroups {

   @Validated(OnCreate.class)
   void validateForCreate(@Valid InputWithGroups input){
     // do something
   }

   @Validated(OnUpdate.class)
   void validateForUpdate(@Valid InputWithGroups input){
     // do something
   }

}
```

Note that the `@Validated` annotation must again be applied to the whole class. To define which validation group should be active, it must also be applied at method level.

To make certain that the above works as expected, we can implement a unit test:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class ValidatingServiceWithGroupsTest {

 @Autowired
 private ValidatingServiceWithGroups service;

 @Test
 void whenInputIsInvalidForCreate_thenThrowsException() {
  InputWithGroups input = validInput();
  input.setId(42L);

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateForCreate(input);
  });
 }

 @Test
 void whenInputIsInvalidForUpdate_thenThrowsException() {
  InputWithGroups input = validInput();
  input.setId(null);

  assertThrows(ConstraintViolationException.class, () -> {
   service.validateForUpdate(input);
  });
 }

}
```

Careful with Validation Groups

Using validation groups can easily become an anti-pattern since we're mixing concerns. With validation groups the validated entity has to know the validation rules for all the use cases (groups) it is used in. More on this topic in my article about **Bean Validation anti-patterns**.

Returning Structured Error Responses

When a validation fails, we want to return a meaningful error message to the client. In order to enable the client to display a helpful error message, **we should return a data structure that contains an error message for each validation that failed**.

First, we need to define that data structure. We'll call it `ValidationErrorResponse` and it contains a list of `Violation` objects:

```
public class ValidationErrorResponse {

 private List<Violation> violations = new ArrayList<>();

 // ...
}

public class Violation {

 private final String fieldName;

 private final String message;

 // ...
}
```

Then, we create a global `ControllerAdvice` that handles all `ConstraintViolationExceptions` that bubble up to the controller level. In order to catch validation errors for request bodies as well, we will also handle `MethodArgumentNotValidExceptions`:

```
@ControllerAdvice
class ErrorHandlingControllerAdvice {

 @ExceptionHandler(ConstraintViolationException.class)
 @ResponseStatus(HttpStatus.BAD_REQUEST)
 @ResponseBody
 ValidationErrorResponse onConstraintValidationException(
    ConstraintViolationException e) {
  ValidationErrorResponse error = new ValidationErrorResponse();
  for (ConstraintViolation violation : e.getConstraintViolations()) {
   error.getViolations().add(
     new Violation(violation.getPropertyPath().toString(), violation.getMessage()));
  }
  return error;
 }

 @ExceptionHandler(MethodArgumentNotValidException.class)
 @ResponseStatus(HttpStatus.BAD_REQUEST)
 @ResponseBody
 ValidationErrorResponse onMethodArgumentNotValidException(
    MethodArgumentNotValidException e) {
  ValidationErrorResponse error = new ValidationErrorResponse();
  for (FieldError fieldError : e.getBindingResult().getFieldErrors()) {
   error.getViolations().add(
     new Violation(fieldError.getField(), fieldError.getDefaultMessage()));
  }
  return error;
 }

}
```

What we're doing here is simply reading information about the violations out of the exceptions and translating them into our `ValidationErrorResponse` data structure.

Note the `@ControllerAdvice` annotation which makes the exception handler methods available globally to all controllers within the application context.

Conclusion

In this tutorial, we've gone through all major validation features we might need when building an application with Spring Boot.

If you want to get your hands dirty on the example code, have a look at the github repository.

Update History

- **10-25-2018:** added a word of caution on using bean validation in the persistence layer (see this thread on Twitter).

Follow me on Twitter for more tips on how to become a better software developer.

Tom Hombergs

As a professional software engineer, consultant, architect, and general problem solver, I've been practicing the software craft for more than ten years and I'm still learning something new every day. I love sharing the things I learned, so you (and future me) can get a head start.