Validação em Spring Boot

Última modificação: 30 de janeiro de 2021

por Alejandro Ugarte

Spring Boot

1. Visão Geral

Quando se trata de validar a entrada do usuário, Spring Bootfornece forte suporte para esta tarefa comum, mas crítica, direto da caixa.

Embora Spring Boot ofereça suporte à integração contínua com validadores personalizados, **o padrão de fato para realizar a validação é o Hibernate Validator** , a implementação de referência da estrutura de validação do Bean .

Neste tutorial, **veremos como validar objetos de domínio no Spring Boot** .

2. As dependências do Maven

Neste caso, aprenderemos como validar objetos de domínio no Spring Boot **construindo um controlador REST básico.**

O controlador primeiro pegará um objeto de domínio, então o validará com o Hibernate Validator e, finalmente, o persistirá em um banco de dados H2 na memória.

As dependências do projeto são razoavelmente padronizadas:

```
<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>
```

Conforme mostrado acima, incluímos *spring-boot-starter-web* em nosso arquivo *pom.xml* porque precisaremos dele para criar o controlador REST. Além disso, vamos verificar as versões mais recentes de *spring-boot-starter-jpa* e o banco de dados H2no Maven Central.

A partir do Boot 2.3, também precisamos adicionar explicitamente a dependência *spring-boot-starter-validation* :

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

3. Uma classe de domínio simples

Com as dependências do nosso projeto já implementadas, em seguida, precisamos definir uma classe de entidade JPA de exemplo, cuja função será apenas modelar usuários.

Vamos dar uma olhada nesta aula:

```
@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

}
```

A implementação de nossa classe de entidade *User* é bastante anêmica, mas mostra em poucas palavras como usar as restrições do Bean Validation para restringir os campos de *nome* e e- *mail* .

Para simplificar, restringimos os campos de destino usando apenas a restrição *@NotBlank* . Além disso, especificamos as mensagens de erro com o atributo *message* .

Portanto, quando Spring Boot valida a instância da classe, os campos restritos **não devem ser nulos e seu comprimento aparado deve ser maior que zero** .

Além disso, a Validação do Bean fornece muitas outras restrições úteis além de *@NotBlank*. Isso nos permite aplicar e combinar diferentes regras de validação para as classes restritas. Para obter mais informações, leia os documentos oficiais de validação de bean .

Como usaremos Spring Data JPApara salvar usuários no banco de dados H2 na memória, também precisamos definir uma interface de repositório simples para ter a funcionalidade CRUD básica em objetos de *usuário* :

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

4. Implementando um controlador REST

Obviamente, precisamos implementar uma camada que nos permita obter os valores atribuídos aos campos restritos do nosso objeto *Usuário* .

Portanto, podemos validá-los e realizar algumas tarefas adicionais, dependendo dos resultados da validação.

O Spring Boot torna **esse processo aparentemente complexo realmente simples**por meio da implementação de um controlador REST.

Vejamos a implementação do controlador REST:

```
@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

}
```

Em um contexto Spring REST , a implementação do método *addUser ()* é bastante padrão.

Claro, a parte mais relevante é o uso da anotação *@Valid* .

Quando o Spring Boot encontra um argumento anotado com *@Valid* , ele inicializa automaticamente a implementação JSR 380 padrão - Hibernate Validator - e valida o argumento.

Quando o argumento de destino falha em passar na validação, Spring Boot lança uma exceção *MethodArgumentNotValidException* .

5. A anotação *@ExceptionHandler*

Embora seja realmente útil ter Spring Boot validando o objeto *User* passado para o método *addUser ()* automaticamente, a faceta que falta neste processo é como processamos os resultados da validação.

A anotação *@ExceptionHandler* **nos permite lidar com tipos específicos de exceções por meio de um único método.**

Portanto, podemos usá-lo para processar os erros de validação:

```
@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;
}
```

Especificamos a exceção *MethodArgumentNotValidException* como a exceção a ser tratada . Conseqüentemente, o Spring Boot chamará esse método **quando o objeto *User* especificado for inválido** .

O método armazena o nome e a mensagem de erro pós-validação de cada campo inválido em um *Mapa.* Em seguida, ele envia o *mapa de* volta ao cliente como uma representação JSON para processamento posterior.

Simplificando, o controlador REST nos permite processar facilmente solicitações para diferentes terminais, validar objetos de *usuário* e enviar as respostas no formato JSON.

O design é flexível o suficiente para lidar com as respostas do controlador por meio de várias camadas da web, variando de mecanismos de modelo, como Thymeleaf , a uma estrutura JavaScript completa, como Angular .

6. Testando o controlador REST

Podemos testar facilmente a funcionalidade de nosso controlador REST com um teste de integração .

Vamos começar a simular / autowiring a implementação da interface *UserRepository* , junto com a instância *UserController* e um objeto *MockMvc* :

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

    @MockBean
    private UserRepository userRepository;

    @Autowired
    UserController userController;

    @Autowired
    private MockMvc mockMvc;

    //...

}
```

Como estamos apenas testando a camada da web, usamos a anotação *@WebMvcTest* . Ele nos permite solicitações facilmente testar e respostas usando o conjunto de métodos estáticos implementadas pelos *MockMvcRequestBuilders* e *MockMvcResultMatchers* classes.

Agora vamos testar o método *addUser ()* com um objeto *User* válido e um inválido passado no corpo da solicitação:

```
@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));
    }
}
```

Além disso, podemos testar a API do controlador REST **usando um aplicativo gratuito de teste de ciclo de vida da API** , como o Postman .

7. Executando o aplicativo de amostra

Finalmente, podemos executar nosso projeto de exemplo com um método *main ()*padrão :

```
@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);
        };
    }
}
```

Como esperado, devemos ver alguns objetos *User* impressos no console.

Uma solicitação POST para o endpoint http: // localhost: 8080 / users com um objeto *User* válido retornará a *String* “User is valid”.

Da mesma forma, uma solicitação POST com um objeto *User* sem *nome* e valores de e- *mail* retornará a seguinte resposta:

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

8. Conclusão

Neste artigo, **aprendemos o básico para realizar a validação no Spring Boot** .

Como de costume, todos os exemplos mostrados neste artigo estão disponíveis no GitHub .