Quarkus - Validation with Hibernate Validator

This guide covers how to use Hibernate Validator/Bean Validation for:

- validating the input/output of your REST services;
- validating the parameters and return values of the methods of your business services.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3

Architecture

The application built in this guide is quite simple. The user fills a form on a web page. The web page sends the form content to the `BookResource` as JSON (using Ajax). The `BookResource` validates the user input and returns the *result* as JSON.
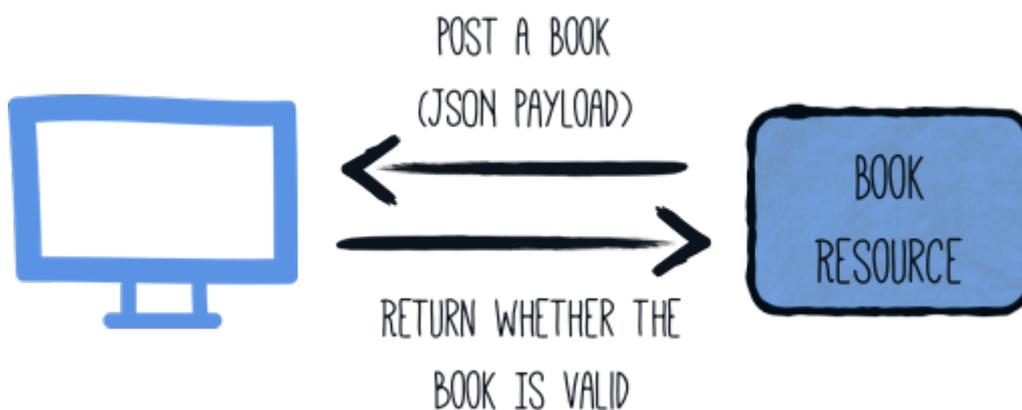
Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `validation-quickstart` directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=validation-quickstart \
    -DclassName="org.acme.validation.BookResource" \
    -Dpath="/books" \
    -Dextensions="resteasy-jsonb, hibernate-validator"
cd validation-quickstart
```

This command generates a Maven structure importing the RESTEasy/JAX-RS, JSON-B and Hibernate Validator/Bean Validation extensions.

If you already have your Quarkus project configured, you can add the `hibernate-validator` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="hibernate-validator"
```

This will add the following to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-validator</artifactId>
</dependency>
```

Accessing the Validator

Edit the `org.acme.validation.BookResource` class, and inject the `Validator` object as follows:

```
@Inject
Validator validator;
```

The `Validator` allows checking constraints on a specific object.

Constraints

In this application, we are going to test an elementary object, but we support complicated constraints

and can validate graphs of objects. Create the `org.acme.validation.Book` class with the following content:

```
package org.acme.validation;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Min;

public class Book {

    @NotBlank(message="Title may not be blank")
    public String title;

    @NotBlank(message="Author may not be blank")
    public String author;

    @Min(message="Author has been very lazy", value=1)
    public double pages;
}
```

Constraints are added on fields, and when an object is validated, the values are checked. The getter and setter methods are also used for JSON mapping.

JSON mapping and validation

Back to the `BookResource` class. Add the following method:

```
@Path("/manual-validation")
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Result tryMeManualValidation(Book book) {
    Set<ConstraintViolation<Book>> violations =
validator.validate(book);
    if (violations.isEmpty()) {
        return new Result("Book is valid! It was validated by
manual validation.");
    } else {
        return new Result(violations);
    }
}
```

Yes it does not compile, `Result` is missing, but we will add it very soon.

The method parameter (`book`) is created from the JSON payload automatically.

The method uses the `Validator` to check the payload. It returns a set of violations. If this set is

empty, it means the object is valid. In case of failures, the messages are concatenated and sent back to the browser.

Let's now create the `Result` class as an inner class:

```java
public static class Result {

    Result(String message) {
        this.success = true;
        this.message = message;
    }

    Result(Set<? extends ConstraintViolation<?>> violations) {
        this.success = false;
        this.message = violations.stream()
                .map(cv -> cv.getMessage())
                .collect(Collectors.joining(", "));
    }

    private String message;
    private boolean success;

    public String getMessage() {
        return message;
    }

    public boolean isSuccess() {
        return success;
    }

}
```

The class is very simple and only contains 2 fields and the associated getters and setters. Because we indicate that we produce JSON, the mapping to JSON is made automatically.

REST end point validation

While using the `Validator` manually might be useful for some advanced usage, if you simply want to validate the parameters or the return value or your REST end point, you can annotate it directly, either with constraints (@NotNull, @Digits…) or with @Valid (which will cascade the validation to the bean).

Let's create an end point validating the Book provided in the request:

```
@Path("/end-point-method-validation")
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Result tryMeEndPointMethodValidation(@Valid Book book) {
    return new Result("Book is valid! It was validated by end point
method validation.");
}
```

As you can see, we don't have to manually validate the provided Book anymore as it is automatically validated.

If a validation error is triggered, a violation report is generated and serialized as JSON as our end point produces a JSON output. It can be extracted and manipulated to display a proper error message.

Service method validation

It might not always be handy to have the validation rules declared at the end point level as it could duplicate some business validation.

The best option is then to annotate a method of your business service with your constraints (or in our particular case with @Valid):

```
package org.acme.validation;

import javax.enterprise.context.ApplicationScoped;
import javax.validation.Valid;

@ApplicationScoped
public class BookService {

    public void validateBook(@Valid Book book) {
        // your business logic here
    }
}
```

Calling the service in your rest end point triggers the Book validation automatically:

```
@Inject BookService bookService;

@Path("/service-method-validation")
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Result tryMeServiceMethodValidation(Book book) {
    try {
        bookService.validateBook(book);
        return new Result("Book is valid! It was validated by
service method validation.");
    } catch (ConstraintViolationException e) {
        return new Result(e.getConstraintViolations());
    }
}
```

Note that, if you want to push the validation errors to the frontend, you have to catch the exception and push the information yourselves as they will not be automatically pushed to the JSON output.

Keep in mind that you usually don't want to expose to the public the internals of your services - and especially not the validated value contained in the violation object.

A frontend

Now let's add the simple web page to interact with our `BookResource`. Quarkus automatically serves static resources contained in the `META-INF/resources` directory. In the `src/main/resources/META-INF/resources` directory, replace the `index.html` file with the content from this index.html file in it.

Run the application

Now, let's see our application in action. Run it with:

```
./mvnw compile quarkus:dev
```

Then, open your browser to http://localhost:8080/:

1. Enter the book details (valid or invalid)

2. Click on the *Try me...* buttons to check if your data is valid using one of the methods we presented above.

As usual, the application can be packaged using `./mvnw clean package` and executed using the `-runner.jar` file. You can also build the native executable using `./mvnw package -Pnative`.

Going further

Hibernate Validator extension and CDI

The Hibernate Validator extension is tightly integrated with CDI.

Configuring the ValidatorFactory

Sometimes, you might need to configure the behavior of the `ValidatorFactory`, for instance to use a specific `ParameterNameProvider`.

While the `ValidatorFactory` is instantiated by Quarkus itself, you can very easily tweak it by declaring replacement beans that will be injected in the configuration.

If you create a bean of the following types in your application, it will automatically be injected into the `ValidatorFactory` configuration:

- `javax.validation.ClockProvider`
- `javax.validation.ConstraintValidator`
- `javax.validation.ConstraintValidatorFactory`
- `javax.validation.MessageInterpolator`
- `javax.validation.ParameterNameProvider`
- `javax.validation.TraversableResolver`
- `org.hibernate.validator.spi.properties.GetterPropertySelectionStrategy`
- `org.hibernate.validator.spi.scripting.ScriptEvaluatorFactory`

You don't have to wire anything.

> ⚠️ Obviously, for each listed type, you can declare only one bean.
>
> These beans should be declared as `@ApplicationScoped`.

Constraint validators as beans

You can declare your constraint validators as CDI beans:

```java
@ApplicationScoped
public class MyConstraintValidator implements
ConstraintValidator<MyConstraint, String> {

    @Inject
    MyService service;

    @Override
    public boolean isValid(String value, ConstraintValidatorContext
context) {
        if (value == null) {
            return true;
        }

        return service.validate(value);
    }
}
```

When initializing a constraint validator of a given type, Quarkus will check if a bean of this type is available and, if so, it will use it instead of instantiating one.

Thus, as demonstrated in our example, you can fully use injection in your constraint validator beans.

> ℹ️ Except in very specific situations, it is recommended to make the said beans `@ApplicationScoped`.

Validation and localization

By default, constraint violation messages will be returned in the build system locale.

You can configure this behavior by adding the following configuration in your `application.properties`:

```
# The default locale to use
quarkus.default-locale=fr-FR
```

If you are using RESTEasy, in the context of a JAX-RS endpoint, Hibernate Validator will automatically resolve the optimal locale to use from the `Accept-Language` HTTP header, provided the supported

locales have been properly specified in the `application.properties`:

```
# The list of all the supported locales
quarkus.locales=en-US,es-ES,fr-FR
```

Hibernate Validator Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.hibernate-validator.fail-fast` Enable the fail fast mode. When fail fast is enabled the validation will stop on the first constraint violation detected.	boolean	`false`

Method validation	Type	Default
🔒 `quarkus.hibernate-validator.method-validation.allow-overriding-parameter-constraints` Define whether overriding methods that override constraints should throw a `ConstraintDefinitionException`. The default value is `false`, i.e. do not allow. See Section 4.5.5 of the JSR 380 specification, specifically "In sub types (be it sub classes/interfaces or interface implementations), no parameter constraints may be declared on overridden or implemented methods, nor may parameters be marked for cascaded validation. This would pose a strengthening of preconditions to be fulfilled by the caller."	boolean	`false`
🔒 `quarkus.hibernate-validator.method-validation.allow-parameter-constraints-on-parallel-methods` Define whether parallel methods that define constraints should throw a `ConstraintDefinitionException`. The default value is `false`, i.e. do not allow. See Section 4.5.5 of the JSR 380 specification, specifically "If a sub type overrides/implements a method originally defined in several parallel types of the hierarchy (e.g. two interfaces not extending each other, or a class and an interface not implemented by said class), no parameter constraints may be declared for that method at all nor parameters be marked for cascaded validation. This again is to avoid an unexpected strengthening of preconditions to be fulfilled by the caller."	boolean	`false`

🔒 `quarkus.hibernate-validator.method-validation.allow-multiple-cascaded-validation-on-return-values` Define whether more than one constraint on a return value may be marked for cascading validation are allowed. The default value is `false`, i.e. do not allow. See Section 4.5.5 of the JSR 380 specification, specifically "One must not mark a method return value for cascaded validation more than once in a line of a class hierarchy. In other words, overriding methods on sub types (be it sub classes/interfaces or interface implementations) cannot mark the return value for cascaded validation if the return value has already been marked on the overridden method of the super type or interface."	boolean	`false`