What’s New in 9.1

2025-11-07

Table of Contents

Jakarta EE 11

Hibernate Validator 9.1 continues to target Jakarta Validation 3.1.1 and requires a minimum Java version of 17.

Performance improvements

In this version, we have changed the `jakarta.validation.Path` implementation, approach to processed bean tracking and slightly modified the message interpolation to improve both overall performance and performance of cascading validation when beans require a lot of cascading operations. We plan to publish a more detailed report with benchmark results, so stay tuned if you are interested in the details and numbers.

Deprecating the use of `@Valid` at the container level

Since Bean Validation 2.0, which introduced type argument constraints, the approach of placing the `@Valid` annotation at the container level to apply cascading validation to container elements has been considered a "legacy" approach. Users have been encouraged to move their constraints to the type argument level.

Example 1: Legacy approach to cascading validation for container elements

```
class MyBean {
    @Valid ①
    List<MyContainerElement> list;
}
```

① Cascading validation is requested at the container level, whereas the expectation is that container elements will be considered for cascading, not the container itself.

Example 2: Current approach to cascading validation for container elements

```
class MyBean {
    List<@Valid MyContainerElement> list; ①
}
```

① Cascading validation is requested at the container element level. This clearly communicates the intention that elements are expected to be cascaded into and not the container itself.

Starting with 9.1, Hibernate Validator will produce warnings during the metadata building step if it detects the legacy approach to cascading validation of container elements. In the future version, this support of Bean Validation 1.0/1.1 legacy behaviour will be dropped. Placing the `@Valid` at the container level would result in cascading into the container itself rather than into its elements.

There is still some work ahead of us before we can turn off this legacy behaviour. For example, we must address the case where the container does not have type arguments but still requires cascading into its elements. This is currently tracked at the Jakarta Validation specification level through the following issue.

We encourage users to review and update their validation mapping where necessary. Additionally, if you encounter a particular use case that you believe is not covered, please don't hesitate to contact us.

Extended validation path

This version introduces another extension of the `jakarta.validation.Path`: `org.hibernate.validator.path.RandomAccessPath`. There are scenarios where the first couple of nodes have to be inspected to determine how to process the constraint violation. For cases when the path is represented by an array or some other collection that allows easy random access to the nodes, it would be simple enough to expose the access to the nodes by index:

Example 3: Random access to the path nodes

```
Path path = constraintViolation.getPropertyPath();
if ( path instanceof org.hibernate.validator.path.RandomAccessPath hvPath ) {
    Node rootNode = hvPath.getRootNode();
    // ...
    int index = ...
    Node someNode = hvPath.getNode(index);
    // ...
    for(int i; i < hvPath.length(); i++) {
        hvPath.getNode(i);
    }
}
```

Constraint initialization shared data

Constraint initialization shared data opens up a way for constraint validators to access a shared instance within the `initialize(..)`. This can be used to cache and reuse elements required to construct a constraint validator. For example, internally, this mechanism is used by the pattern constraint validator to reuse the `java.util.regex.Pattern` instances

Example 4: Accessing the constraint validator's lazy initialization shared data in a constraint validator

```
public class ParsableDateTimeFormatValidator
    implements HibernateConstraintValidator<ParsableDateTimeFormat, String> { ①

    private DateTimeFormatter formatter;

    @Override
    public void initialize(ConstraintDescriptor<ParsableDateTimeFormat> constraintDescriptor,
            HibernateConstraintValidatorInitializationContext initializationContext) {
        formatter = initializationContext.getSharedData( DateTimeFormatterCache.class, DateTimeFormatterCache:
:new ) ②
                .get( constraintDescriptor.getAnnotation().dateFormat() ); ③
    }

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

        try {
            formatter.parse( dateTime );
        }
        catch (DateTimeParseException e) {
            return false;
        }
        return true;
    }

    private static class DateTimeFormatterCache { ④
        private final Map<String, DateTimeFormatter> cache = new ConcurrentHashMap<>();

        DateTimeFormatter get(String format) {
            return cache.computeIfAbsent( format, DateTimeFormatter::ofPattern );
        }
    }
}
```

① Implement the Hibernate Validator `HibernateConstraintValidator` extension to have access to the initialization context.

② Retrieve the shared data from the initialization context, providing the supplier that will be executed if the `DateTimeFormatterCache` is not yet available in the current initialization context.

③ Perform some actions with the shared data instance.

④ A simple wrapper around the map to cache the formatters. Compared to the use of a static field cache, using the shared data has the benefit that it is tied to the initialization context and will be garbage collected along with it.

`IpAddress` constraint

The new `@IpAddress` constraint validates that the corresponding string is a well-formed IP address. This constraint provides a `IpAddress.Type` enum with the IP address types it can validate: `IPv4`, `IPv6` or `ANY`. By default, `IpAddress.ANY` is used, which allows validating all the other address types listed in the `IpAddress.Type` enum.

```
@IpAddress ①
String address;
// ...
@IpAddress(type = Type.IPv6) ②
private String address;
```

① Using a default configuration of the `@IpAddress` constraint, where both `IPv4` and `IPv6` address types are considered valid.
② Applying the `@IpAddress` constraint, where only the `IPv6` addresses are considered valid.