What’s New in 9.2

2026-11-11

Table of Contents

New constraints . . . 1
Password validation . . . 2
Bean resolution and dependency injection . . . 3
Configurable property access . . . 4
Programmatic constraint mapping . . . 5
Message interpolation improvements . . . 6
`PropertyNodeNameProvider` improvements . . . 7
Validation performance and fixes . . . 8
Message parameter escaping . . . 9

New constraints

Hibernate Validator 9.2 adds the following built-in constraints. Corresponding constraint definitions are also available for programmatic constraint mapping.

Constraint	Purpose
`@Alphabetic`	Checks that every character in a character sequence is a letter, as determined by `Character.isLetter(char)`.
`@Alphanumeric`	Checks that every character in a character sequence is a letter or a digit, as determined by `Character.isLetterOrDigit(char)`.
`@BIC`	Checks Bank Identifier Codes (SWIFT codes).
`@Contains`	Checks for specified substrings. By default all must be present; `minRequired` allows requiring a minimum number of matches.
`@CUIT`, `@CUIL` (`org.hibernate.validator.constraints.ar`)	Checks Argentine tax and labor identification codes, respectively.
`@DateTimeFormat`	Checks that a character sequence represents a date and/or time matching a `DateTimeFormatter` pattern, with configurable locales and lenient parsing.
`@Hexadecimal`	Checks hexadecimal character sequences, with options for letter case, fullwidth characters, empty values and a required prefix.
`@IBAN`	Checks International Bank Account Numbers, including country-specific lengths and check digits. Spaces are ignored; lowercase letters can be accepted with `allowLowercase`.
`@ISSN`	Checks International Standard Serial Numbers.
`@NullOrNotBlank`	Accepts `null` or a character sequence containing at least one non-whitespace character.
`@NullOrNotEmpty`	Accepts `null` or a non-empty character sequence, collection, map or array.
`@Numeric`	Checks that every character in a character sequence is a digit, as determined by `Character.isDigit(char)`.
`@PasswordStrength`, `@NotCompromised`, `@PasswordPolicy`	Checks password strength, exposure in data breaches, or a configurable set of password policy rules. See the next section for configuration requirements.
`@Port`	Checks that a numeric value or character sequence represents a port number between 0 and 65535, inclusive.
`@RENAVAM` (`org.hibernate.validator.constraints.br`)	Checks Brazilian vehicle registration numbers in the current 11-digit or legacy 9-digit representation.
`@RUC` (`org.hibernate.validator.constraints.py`)	Checks Paraguayan taxpayer registration numbers.
`@StartsWith`, `@EndsWith`	Checks for at least one of the specified prefixes or suffixes. Case-insensitive matching is optional.
`@Trimmed`	Checks that a character sequence has no leading or trailing whitespace, using the same whitespace definition as `String.strip()`.
`@UpperCase`, `@LowerCase`	Checks the case of letters in a character sequence, ignoring non-letter characters.

`@NullOrNotBlank` and `@NullOrNotEmpty` are particularly useful for optional values: an empty `Optional` is valid, while a present value must satisfy the constraint. Many of the new constraints are incubating; consult their Javadoc for their API status and configuration options.

`@BitcoinAddress` and `@IpAddress` are now repeatable, allowing multiple declarations on the same element without an explicit container annotation.

Password validation

`@PasswordStrength(min = …)` delegates to a user-provided `PasswordStrengthEstimator`. The `min` attribute is required, and its scale is defined by the estimator implementation. `@NotCompromised` delegates to a user-provided `CompromisedPasswordChecker`. These services must be registered through the bean resolution system described below; Hibernate Validator does not supply a default strength estimator or compromised password checker.

`@PasswordPolicy` references a `PasswordPolicyDefinition`, which uses a builder to combine rules for length, character types, whitespace, sequences, keyboard walks, strength, compromised passwords and custom checks. Each failing rule produces its own constraint violation. The password constraints support `CharSequence` and `char[]` values and accept `null`. `@PasswordPolicy` can also be used on custom types through `AbstractPasswordPolicyValidator`.

Bean resolution and dependency injection

A new bean resolution system supports registering implementations by type and, optionally, by name. Use `addBeanConfigurer()` to register beans programmatically, or expose `BeanConfigurer` implementations through `ServiceLoader`. A custom `BeanProvider` integrates other dependency injection frameworks. With the `hibernate-validator-cdi` extension, a CDI-backed provider is discovered automatically.

Custom constraint validators can access the `BeanResolver` through `HibernateConstraintValidatorInitializationContext#getBeanResolver()`. It is also exposed by `HibernateValidatorFactory#getBeanResolver()`. Existing configuration setters continue to select the built-in pluggable components; registering a bean alone does not override a component with a built-in default.

Configurable property access

Hibernate Validator now uses Hibernate Accessor to read bean properties. The Class-File API factory is the default on JDK 24 and later, with reflection used on earlier JDKs. Optional ASM and ByteBuddy accessor factories can generate property accessors to avoid reflective calls during validation. Configure an `AccessorFactory` using `accessorFactory()` or the `hibernate.validator.accessor_factory` configuration property. The optional factories are provided by `org.hibernate.accessor:hibernate-accessor-asm` and `org.hibernate.accessor:hibernate-accessor-bytebuddy`.

Programmatic constraint mapping

Programmatic getter mappings are now resolved when building the validator factory, using that factory's getter selection strategy and property node name provider. A mapping can therefore be reused across factories with different strategies. Constraint definitions are snapshotted when passed to `constraint()`; subsequent changes to a definition do not alter constraints already added to a mapping.

Message interpolation improvements

EL message expressions can now read record components as properties, for example `${validatedValue.name}` for a record with a `name` component. At the `BEAN_PROPERTIES` feature level, record access is restricted to component accessors and JavaBeans getters; arbitrary record methods require the `BEAN_METHODS` feature level.

Custom message interpolators can retrieve the constraint validator payload through `HibernateMessageInterpolatorContext#getConstraintValidatorPayload(Class)`. This exposes the payload configured for constraint validators to message interpolation as well.

`HibernateConstraintValidatorContext#addMessageParameter()` and `addExpressionVariable()` now accept `null` values.

`PropertyNodeNameProvider` improvements

`PropertyNodeNameProvider` now provides `getName(Property property, PropertyNodeNameProviderContext context)`. The context exposes the current getter property selection strategy, which can help integrations such as Jackson 3 determine the property name. The original `getName(Property)` method is deprecated for removal; existing implementations remain supported through the new method's default implementation.

`org.hibernate.validator.spi.nodenameprovider.JavaBeanProperty#getMemberName()` returns the name of the field or getter backing the property.

Validation performance and fixes

Constraint validator contexts are reused where possible, and interface constraint tracking avoids unnecessary work for beans without interface constraints. Property path parsing now uses a linear scan.

`validateProperty()` and `validateValue()` now check property paths even for unconstrained beans, rejecting invalid paths with `IllegalArgumentException` instead of returning an empty set.

`@CPF` validation now rejects partially formatted identifiers; see the migration guide for the supported representations. The regular expression based URL validator also handles URLs without a host without throwing a `NullPointerException`.

Message parameter escaping

Message parameters added via `HibernateConstraintValidatorContext#addMessageParameter()` are now escaped by default. Characters that have special meaning in message interpolation (`\`, `{`, `}`, `$`) are automatically escaped.

This behavior is controlled by the new `MessageParameterEscaping` enum (`ESCAPE`, `NONE`) and can be configured globally via the `hibernate.validator.message_parameter_escaping` property or the programmatic API on `BaseHibernateValidatorConfiguration`.

A new overload `addMessageParameter(String name, Object value, MessageParameterEscaping escaping)` is also available on `HibernateConstraintValidatorContext` for per-call control over escaping behavior.