(see Constraint metadata request API).

`value` is the value being validated.

`MessageResolver.interpolate()` is invoked once for each constraint whose isValid() method returns false.

A message resolver implementation shall be threadsafe.

The message resolver is injected to the validator instance through `validator.setMessageResolver(MessageResolver)`.

4.3.3. Examples

These examples describe message interpolation based on the default message resolver's built-in messages (see Appendix B), and the `ValidationMessages.properties` file shown in table . The current locale is assumed English.

```
//ValidationMessages.properties
myapp.creditcard.error=Your credit card number is not valid
```

Table 4.1. message interpolation

Failing constraint declaration	interpolated message
@NotNull	may not be null
@Max(30)	must be less than or equal to 30
@Length(min=5, max=15, message="Key must have between {min} and {max} characters")	Key must have between 5 and 15 characters
@Range(min=4, max=40)	must be between 4 and 40
@CreditCard(message={myapp.creditcard.error})	Your credit card number is not valid

4.4. Bootstrapping

The bootstrapping API aims at providing a `ValidatorFactory` object which is used to create `Validator` instances. The bootstrap process is decoupled from the provider implementation initialization: a bootstrap implementation must be able to bootstrap any Bean Validation provider implementation. The bootstrap sequence has been designed to achieve several goals:

- plug multiple implementations
- choose a specific implementation
- extensibility: an application using a specific provider implementation can use specific configurations
- share and reuse of metadata across validators

- leave as much freedom as possible to implementations
- provide integration mechanisms to Java EE 5 and other containers
- type safety

The main artifacts involved in the bootstrap process are:

- `Validation`: API entry point. Lets you optionally define the Bean Validation provider targeted as well as a provider resolution strategy. Validation generates `ValidatorBuilder` objects and can bootstrap any provider implementation.
- `ValidationProvider`: contract between the bootstrap procedure and a Bean Validation provider implementation.
- `ValidationProviderResolver`: returns a list of all Bean Validation providers available in the execution context (generally the classpath).
- `ValidatorBuilder`: collects the configuration details that will be used to build `ValidatorFactory`. A specific sub interface of `ValidatorBuilder` must be provided by Bean Validation providers as a unique identifier. This sub interface typically hosts provider specific configurations.
- `ValidatorFactory`: result of the bootstrap process. Build `Validator` instances from a given Bean Validation provider.

Let's first see the API in action through some examples before diving into the concrete definitions.

4.4.1. Examples

The most simple approach is to use the default Bean Validation provider or the one defined in the XML configuration file. The bootstrap entry point returns a generic `ValidationBuilder` instance on which we apply the build operation. The `ValidatorFactory` is then ready to provide `Validator` instances.

Example 4.1. Simple Bean Validation bootstrap sequence

```
ValidatorFactory factory = Validation.getValidatorBuilder().build();

//cache the factory somewhere
Validator<Address> addressValidator = factory.getValidator(Address.class);
```

The `ValidatorFactory` object is thread-safe contrary to individual `Validators`. Building `Validator` instances is typically a cheap operation. Building a `ValidatorFactory` is typically more expensive. Make sure to check your Bean Validation implementation documentation for more accurate details.

The second example shows how a container can customize some Bean Validator resource handling to match its own behavior.

Example 4.2. Customize message resolution and constraint factory implementation

```
//some customization from a container like Web Beans
ValidatorFactory factory = Validation
       .getValidatorBuilder()
          .messageResolver( new WBMessageResolver() )
          .constraintFactory( new WBComponentConstraintFactory() )
          .build();

//cache the factory somewhere
Validator<Address> addressValidator = factory.getValidator(Address.class);
```

The third example shows how to bootstrap Bean Validation in an environment not following the traditional Java classloader strategies (such as tools or alternative service containers like OSGi). They can provider some alternative provider resolution strategy to discover Bean Validation providers.

Example 4.3. Customize the Bean Validation provider resolution mechanism

```
//osgi environment
ValidatorFactory factory = Validation
       .defineBootstrapState()
          .providerResolver( new OSGiServiceDiscoverer() )
          .build();

//cache the factory somewhere
Validator<Address> addressValidator = factory.getValidator(Address.class);
```

The last example shows how a client can choose a specific Bean Validation provider and configure provider specific properties programmatically in a type-safe way.

Example 4.4. Use a specific provider and add specific configuration

```
ValidatorFactory factory = Validation
       .builderType( ACMEValidatorBuilder.class )
       .getValidatorBuilder()
          .messageResolver( new ContainerMessageResolver() ) //default configuration option
          .addConstaint(Address.class, customConstraintDescriptor) //ACME specific method
          .build();

//same initialization breaking the chaining method use
ACMEValidatorBuilder acmeBuilder = Validation
       .builderType( ACMEValidatorBuilder.class )
       .getValidatorBuilder();

ValidatorFactory factory = acmeBuilder
          .messageResolver( new ContainerMessageResolver() ) //default configuration option
          .addConstaint(Address.class, customConstraintDescriptor) //ACME specific method
          .build();


/**
 * ACME specific validator builder and configuration options
 */
public interface ACMEValidatorBuilder extends ValidatorBuilder<ACMEValidatorBuilder> {
    /**
     * Programmatically add constraints. Specific to the ACME provider.
     */
    ACMEValidatorBuilder addConstraint(Class<?> entity, ACMEConstraintDescriptor constraintDescriptor);
}
```

We will now explore the various interfaces, their constraints and usage. We will go from the `ValidatorFactory` to the `Validation` class walking up the bootstrap chain.

4.4.2. ValidatorFactory

`ValidatorFactory` objects build and provide initialized instances of `Validator<T>` to Bean Validation clients. Clients should cache `ValidatorFactory` objects and reuse them for optimal performances. The API is designed to allow implementors to share constraint metadata in `ValidatorFactory`.

`ValidatorFactory` implementations must be thread-safe (which is not the case of `Validator`). `ValidatorFactory` implementations are allowed to cache and return the same `Validator` instances. Thread safety must be guaranteed to Bean Validation clients as long as they use `Validator` instances coming from the same `ValidatorFactory.getValidator(Class)` call in a non concurrent way.

Example 4.5. ValidatorFactory interface

```
/**
 * Factory returning initialized Validator instances.
 * Implementations are thread-safe
 * This object is typically cached and reused.
 *
 * @author Emmanuel Bernard
 */
public interface ValidatorFactory {
    /**
     * return an initialized Validator instance for the specific class.
     * Validator instances can be pooled and shared by the implementation
     * In this scenario, the implementation must return thread-safe Validator implementations
     *
     */
    <T> Validator<T> getValidator(Class<T> clazz);
}
```

A `ValidatorFactory` is provided by a `ValidatorBuilder`.

4.4.3. ValidatorBuilder

`ValidatorBuilder` collects configuration informations, determines the correct provider implementation and delegates it the `ValidatorFactory` creation. This class lets you define:

- the message resolver strategy instance
- the constraint factory instance
- the configuration `InputStream`

Clients call `ValidatorBuilder.build()` to retrieve the initialized `ValidatorFactory` instance.

Example 4.6. ValidatorBuilder interface

```
/**
 * Receives configuration information, selects the appropriate
 * Bean Validation provider and build the appropriate
 * ValidatorFactory.
 *
 * The provider is selected in the following way:
 *  - if a specific ValidatorBuilder subclass is requested programmatically using Validation.builderType(
 * find the first provider matching it
 *  - if a specific ValidatorBuilder subclass is defined in META-INF/validation.xml,
 * find the first provider matching it/**
 * Receives configuration information, selects the appropriate
 * Bean Validation provider and build the appropriate
 * ValidatorFactory.
 *
 * Usage:
 * <pre>
 * ValidatorBuilder<?> validatorBuilder = //provided by one of the Validation bootstrap methods
 * ValidatorFactory = validatorBuilder
 *         .messageResolver( new CustomMessageResolver() )
 *         .build();
 * </pre>
 *
 * The ValidationProviderResolver is specified at ValidatorBuilder time (see {@link javax.validation.spi.
 * If none is explicitely requested, the default ValidationProviderResolver is used.
 *
 * The provider is selected in the following way:
 *  - if a specific ValidatorBuilder subclass is requested programmatically using Validation.builderType(
 * find the first provider matching it
 *  - if a specific ValidatorBuilder subclass is defined in META-INF/validation.xml,
 * find the first provider matching it
 *  - otherwise, use the first provider returned by the ValidationProviderResolver
 *
 * Implementations are not meant to be thread safe
 *
 * @author Emmanuel Bernard
 */
public interface ValidatorBuilder<T extends ValidatorBuilder> {
    /**
     * Defines the message resolver used. Has priority over the configuration based message resolver.
     *
     * @param resolver message resolver implementation.
     * @return <code>this</code> following the chaining method pattern.
     */
    T messageResolver(MessageResolver resolver);

    /**
     * Defines the constraint factory. Has priority over the configuration based constraint factory.
     *
     * @param constraintFactory constraint factory inmplementation.
     * @return <code>this</code> following the chaining method pattern.
     */
    T constraintFactory(ConstraintFactory constraintFactory);

    /**
     * Configure the ValidatorFactory based on <code>stream</code>
     * If not specified, META-INF/validation.xml is used
     *
     * The stream should be closed by the client API after the ValidatorFactory has been returned
     *
     * @param stream configuration stream.
     * @return <code>this</code> following the chaining method pattern.
     */
    T configure(InputStream stream);

    /**
     * Build a ValidatorFactory implementation.
     *
     * @return ValidatorFactory
     */
    ValidatorFactory build();
}
```

A Bean Validation provider must define a sub interface of `ValidatorBuilder` uniquely identifying the provider. Its `ValidationProvider` implementation must return true when this sub interface type is passed as a parameter, false otherwise. The `ValidatorBuilder` sub interface typically hosts provider specific configuration methods.

To facilitate the use of provider specific configuration methods, `ValidatorBuilder` uses generics: `ValidatorBuilder<T extends ValidatorBuilder<T>>` ; the generic return type `T` is returned by chaining methods. The provider specific sub interface must resolve the generic T as itself as shown in the following example.

Example 4.7. Example of provider specific ValidatorBuilder sub interface

```
/**
 * Unique identifier of the ACME provider
 * also host some provider specific configuration methods
 *
 * @author Emmanuel Bernard
 */
public interface ACMEValidatorBuilder
    extends ValidatorBuilder<ACMEValidatorBuilder> {

    /**
     * Enables contraints implementation dynamic reloading when using ACME
     * default to false
     */
    ACMEValidatorBuilder enableDynamicReloading(boolean);

}
```

When `ValidatorBuilder.build()` is called, the requested Bean Validation provider is determined and the the result of `validationProvider.buildValidatorFactory(ValidatorBuilderImplementor)` is returned. `ValidatorBuilderImplementor` gives access to the configuration artifacts passed to `ValidatorBuilder`. A typical implementation of `ValidatorBuilder` also implements `ValidatorBuilderImplementor`, hence `this` can be passed to `buildValidatorFactory(ValidatorBuilderImplementor)`.

Example 4.8. ValidatorBuilderImplementor interface

```
/**
 * Contract between a <code>ValidationBuilder</code> and a </code>ValidatorProvider</code> to create
 * a <code>ValidatorFactory</code>.
 * The configuration artifacts provided to the <code>ValidationBuilder</code> are passed along.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ValidatorBuilderImplementor {
    /**
     * Message resolver as defined by the client programmatically
     * or null if undefined.
     *
     * @return message provider instance or null if not defined
     */
    MessageResolver getMessageResolver();

    /**
     * Returns the configuration stream defined by the client programmatically
     * or null if undefined.
     *
     * @return the configuration input stream or null
     */
    InputStream getConfigurationStream();

    /**
     * Defines the constraint implementation factory as defined by the client programmatically
     * or null if undefined
     *
     * @return factory instance or null if not defined
     */
    ConstraintFactory getConstraintFactory();
}
```

The correct provider implementation is resolved according to the following rules in the following order:

- Use the provider implementation requested if `ValidatorBuilder` has been created from `Validation.builderType(Class).build()`.
- Use the provider implementation associated with the `ValidatorBuilder` implementation described in the XML configuration (under `validation.provider`) if defined: the value of this element is the fully qualified class name of the `ValidationBuilder` sub interface uniquely identifying the provider.
- Use the first provider implementation returned by `validationProviderResolver.getValidationProviders()`.

The `ValidationProviderResolver` is specified when `ValidatorBuilder` are created (see `ValidationProvider`). If no `ValidationProviderResolver` instance has been specified, the default `ValidationProviderResolver` is used.

`ValidatorBuilder` instances are provided to the Bean Validation client through one of `Validation` methods. `ValidatorBuilder` instances are created by `ValidationProvider`.

Warning

Should we add a ignore XML method? to bypass the XMl file configuration?

4.4.4. ValidationProvider and ValidationProviderResolver

`ValidationProvider` is the contract between the bootstrap process and a Bean Validation provider. `ValidationProviderResolver` can be implemented by any Bean Validation client but is typically implemented by containers having specific classloader structures and restrictions.

4.4.4.1. ValidationProviderResolver

`ValidationProviderResolver` returns the list of Bean Validation providers available at runtime and more specifically a `ValidationProvider` instance for each provider available in the context. This service can be customized by implementing `ValidationProviderResolver`. Implementations must be thread-safe.

Example 4.9. ValidationProviderResolver

```
/**
 * Determine the list of Bean Validation providers available in the runtime environment
 * <p>
 * Bean Validation providers are identified by the presence of META-INF/services/javax.validation.spi.Val
 * files following the Service Provider pattern described
 * <a href="http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Service%20Provider">here</a>
 * <p/>
 * Each META-INF/services/javax.validation.spi.ValidationProvider file contains the list of
 * ValidationProvider implementations each of them representing a provider.
 *
 * Implementations must be thread-safe.
 *
 * @author Emmanuel Bernard
 */
public interface ValidationProviderResolver {
    /**
     * Returns a list of ValidationProviders available in the runtime environment.
     *
     * @return list of validation providers.
     */
    List<ValidationProvider> getValidationProviders();
}
```

By default, providers are resolved using the Service Provider pattern described in http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Service%20Provider. Each Bean Validation provider should supply a service provider configuration file by creating a text file name `javax.validation.spi.ValidationProvider` and placing it in the `META-INF/services` directory of one of its jar files. The content of the the file should be the name of the provider implementation class of the `javax.validation.spi.ValidationProvider` interface.

Persistence provider jars may be installed or made available in the same ways as other service providers, e.g. as extensions or added to the application classpath according to the guidelines in the JAR file speci#cation.

The default `ValidationProviderResolver` implementation will locate all the Bean Validation providers by their provider configuration files visible in the classpath. The default `ValidationProviderResolver` implementation is recommended and custom `ValidationProviderResolver` implementations should be rarely used. A typical use of a custom resolution is resolving providers in a classloader constrained container like OSGi or in a tool environment (IDE).

The default implementation of `ValidationProviderResolver` must be available at `javax.validation.bootstrap.DefaultValidationProviderResolver`. It must contain a public no-arg constructor and must not have any other non private attribute or method besides the method described by `ValidationProviderResolver`.

4.4.4.2. ValidationProvider

`ValidationProvider` represents the SPI (Service Provider Interface) defining the contract between the provider discovery mechanism and initialization and the provider. A `ValidationProvider` does:

- Determine if a provider matches a given `ValidatorBuilder` sub interface. One `ValidatorBuilder` sub interface specifically represent one Bean Validation provider.
- Provide a provider specific `ValidatorBuilder` implementation. This `ValidatorBuilder` will specifically build `ValidatorFactory` though the provider it comes from.
- Build a `ValidatorFactory` object from the configuration provided by `ValidatorBuilder`.

Example 4.10. ValidationProvider

```
/**
 * Contract between the validation bootstrap mechanism and the provider engine.
 *
 * Implementations must have a public no-arg constructor. The construction of a provider should be
 * as "lightweight" as possible.
 *
 * @author Emmanuel Bernard
 * @author Hardy Ferentschik
 */
public interface ValidationProvider {
    /**
     * @param builderClass targeted builder class.
     *
     * @return <code>true</code> if <code>builderClass</code> is the
     *         Bean Validation Provider sub interface for ValidatorBuilder
     *         This sub interface uniquely identify a provider.
     */
    boolean isSuitable(Class<? extends ValidatorBuilder<?>> builderClass);

    /**
     * Returns a ValidatorBuilder instance implementing the <code>builderType</code> interface.
     * The ValidatorBuilder instance uses the current provider to build
     * the ValidatorFactory instance.
     *
     * @param builderClass the Builder class type
     *
     * @param state bootstrap state
     * @return specific validator builder implementation
     */
    <T extends ValidatorBuilder<T>> T createSpecializedValidatorBuilder(BootstrapState state, Class<T> bu

    /**
     * Returns a ValidatorBuilder instance. This instance is not bound to
     * use the current provider. The choice of provider follows the algorithm described
     * in {@link javax.validation.ValidatorBuilder}
     * The ValidationProviderResolver used is provided by <code>state</code>.
     * If null, the default ValidationProviderResolver is used.
     *
     * @param state bootstrap state
     *
     * @return validator builder implementation
     */
    ValidatorBuilder<?> createGenericValidatorBuilder(BootstrapState state);

    /**
     * Build a ValidatorFactory using the current provider implementation. The ValidationFactory
     * is assembled and follow the configuration passed using ValidatorBuilderImplementor.
     * <p>
     * The returned ValidatorFactory is properly initialized and ready for use.
     * </p>
     *
     * @param configuration the configuration descriptor
     *
     * @return the instanciated ValidatorFactory
     */
    ValidatorFactory buildValidatorFactory(ValidatorBuilderImplementor configuration);
}
```

Example 4.11. BootstrapState interface

```
/**
 * Defines the state used to bootstrap the ValidationBuilder
 *
 * @author Emmanuel Bernard
 */
public interface BootstrapState {
    /**
     * returns the user defined ValidationProviderResolver strategy instance or null if undefined
     *
     * @return ValidationProviderResolver instance or null
     */
    ValidationProviderResolver getValidationProviderResolver();
}
```

A client can request a specific Bean Validation provider by using `Validation.builderType(Class<T exptends ValidatorBuilder<T>>)` or by defining the provider in the XML configuration file. The key uniquely identifying a Bean Validation provider is a provider specific sub interface of `ValidatorBuilder`. The sub interface does not have to add any new method but is the natural holder of provider specific methods.

Example 4.12. Example of provider specific ValidationBuilder sub interface

```
/**
 * Unique identifier of the ACME provider
 * also host some provider specific configuration methods
 *
 * @author Emmanuel Bernard
 */
public interface ACMEValidatorBuilder
    extends ValidatorBuilder<ACMEValidatorBuilder> {

    /**
     * Enables contraints implementation dynamic reloading when using ACME
     * default to false
     */
    ACMEValidatorBuilder enableDynamicReloading(boolean);

}
```

Note

`ACMEValidatorBuilder` references itself in the generic definition. Methods of `ValidatorBuilder` will return the `ACMEValidatorBuilder` making the API easy to use even for vendor specific extensions.

The provider discovery mechanism uses the following algorithm:

- Retrieve available providers using `ValidationProviderResolver.getValidationProviders()`.
- The first `ValidationProvider` matching the requested provider is returned. Providers are evaluated in the order they are provided by `ValidationProviderResolver`. A provider is considered matching if `ValidationProvider.isSuitable(Class<T extends ValidatorBuilder<T>>)` returns true when the requested provider specific `ValidationBuilder` sub interface is passed as a parameter.

When the default Bean Validation provider is requested, the first `ValidationProvider` returned by the `ValidationProviderResolver` strategy is returned.

Every Bean Validation provider must provide a `ValidationProvider` implementation containing a public no-arg constructor and add the corresponding `META-INF/services/javax.validation.spi.ValidationProvider` file descriptor in one of its jars.

4.4.5. Validation

The `Validation` class is the entry point used to bootstrap Bean Validation providers. The first entry point, `getValidatorBuilder()`, returns a `ValidatorBuilder` not tied to any provider implementation. The first provider returned by the default `ValidationProviderResolver` is used to build the `ValidationBuilder`. `Validation.getValidatorBuilder()` is equivalent to `Validation.defineBootstrapState().getValidatorBuilder()`.

Warning

Should the resolver strategy be configurable by XML

Example 4.13. Validation methods available

```
/**
 * This class is the entry point for the Bean Validation framework. There are three ways to bootstrap the
 * <ul>
 * <li>
 * The easiest approach is to use the default Bean Validation provider.
 * <pre>
 * ValidatorFactory factory = Validation.getValidatorBuilder().build();
 * </pre>
 * In this case {@link  javax.validation.bootstrap.DefaultValidationProviderResolver  DefaultValidationPr
 * will be used to locate available providers.
 *
 * The chosen provider is defined as followed:
 * <ul>
 * <li>if the XML configuration defines a provider, this provider is used</li>
 * <li>if the XML configuratio does not define a provider or if no XML configuration is present the first
 * returned by the ValidationProviderResolver isntance is used.</li>
 * </ul>
 * </li>
 * <li>
 * The second bootstrap approach allows to choose a custom <code>ValidationProviderResolver</code>. The c
 * <code>ValidationProvider</code> is then determined in the same way as in the default bootstrapping cas
 * <pre>
 * ValidatorBuilder&lt;?&gt; builder = Validation
 *    .defineBootstrapState()
 *    .providerResolver( new MyResolverStrategy() )
 *    .getValidatorBuilder();
 * ValidatorFactory factory = builder.build();
 * </pre>
 * </li>
 *
 * <p/>
 * <li>
 * The third approach allows you to specify explicitly and in a type safe fashion the expected provider b
 * using its specific <code>ValidatorBuilder</code> sub-interface.
 *
 * Optionally you can choose a custom <code>ValidationProviderResolver</code>.
 * <pre>
 * ACMEValidatorBuilder builder = Validation
 *    .builderType(ACMEValidatorBuilder.class)
 *    .providerResolver( new MyResolverStrategy() )  // optionally set the provider resolver
 *    .getValidatorBuilder();
 * ValidatorFactory factory = builder.build();
 * </pre>
 * </li>
 * </ul>
 * Note:<br/>
 * <ul>
 * <li>
 * The ValidatorFactory object built by the bootstrap process should be cached and shared amongst
 * Validator consumers.
 * </li>
 * <li>
 * This class is thread-safe.
 * </li>
 * </ul>
 *
 * @author Emmanuel Bernard
 * @author Hardy Feretnschik
 * @see DefaultValidationProviderResolver
 */
public class Validation {
    public class Validation {

    /**
     * Build a ValidatorBuilder defering the provider choice for later
     * (XML configuration file or default provider)
     *
     * The provider list is resolved using the default strategy.
     *
     * @return ValidatorBuilder instance
     */
    public static ValidatorBuilder<?> getValidatorBuilder() { ... }

    /**
     * Build a generic ValidatorBuilder deferring the provider choice for later
     * (XML configuration file or default provider)
     *
     * <pre>
     * ValidatorBuilder<?> builder = Validation.builderType(ACMEValidatorBuilder.class)
     *     .providerResolver( new MyResolverStrategy() )
     *     .build();
     * </pre>
     *
     * The provider list is resolved using the strategy provided to the bootstrap state.
     *
     * @return instance building a generic ValidatorBuilder compliant with the bootstrap state provided.
     */
    public static GenericBuilderFactory defineBootstrapState() { ... }

    /**
     * Build a <code>ValidatorBuilder</code> for a particular provider implementation.
     * Optionally override the provider resolution strategy used to determine the provider.
     * <p/>
     * Used by applications targeting a specific provider programmatically.
     * <p/>
     * <pre>
     * ACMEValidatorBuilder builder = Validation.builderType(ACMEValidatorBuilder.class)
     *     .providerResolver( new MyResolverStrategy() )
     *     .build();
     * </pre>
     *
     * Where <code>ACMEValidatorBuilder</code> is the <code>ValidatorBuiler</code> sub interface uniquely
     * the ACME Bean Validation provider.
     *
     * @param builderType the ValidatorBuilder sub interface uniquely defining the targeted provider.
     *
     * @return instance building a provider specific ValidatorBuilder sub interface implementation.
     *
     * @see #getValidatorBuilder()
     */
    public static <T extends ValidatorBuilder<T>> SpecializedBuilderFactory<T>
            builderType(Class<T> builderType) { ... }
}
```

The second entry point lets the client provide a custom `ValidationProviderResolution` instance. This instance is passed to `GenericBuilderFactory`. `GenericBuilderFactory` builds a generic `ValidatorBuilder` using the first `ValidationProvider` returned by `ValidationProviderResolution` and calling `ValidatorBuilder<?> createGenericValidatorBuilder(BootstrapState state)`. `BootstrapState` holds the `ValidationProviderResolution` instance passed to `GenericBuilderFactory` and will be used by the `ValidatorBuilder` instance when resolving the provider to use.

Example 4.14. GenericBuilderFactory interface

```
/**
 * Defines the state used to bootstrap Bean Validation and create an appropriate
 * ValidatorBuilder
 *
 * @author Emmanuel Bernard
 */
public interface GenericBuilderFactory {
    /**
     * Defines the provider resolution strategy.
     * This resolver returns the list of providers evaluated
     * to build the ValidationBuilder
     *
     * If no resolver is defined, the default ValidationProviderResolver
     * implementation is used.
     *
     * @return <code>this</code> following the chaining method pattern
     */
    GenericBuilderFactory providerResolver(ValidationProviderResolver resolver);

    /**
     * Returns a generic ValidatorBuilder implementation.
     * At this stage the provider used to build the ValidationFactory is not defined.
     *
     * The ValidatorBuilder implementation is provided by the first provider returned
     * by the ValidationProviderResolver strategy.
     *
     * @return a ValidatorBuilder implementation compliant with the bootstrap state
     */
    ValidatorBuilder<?> getValidatorBuilder();
}
```

The last entry point lets the client define the specific Bean Validation provider requested as well as a custom `ValidationProviderResolver` implementation if needed. The entry point method, `builderType(Class<T> builderType)`, takes the provider specific `ValidationBuilder` sub interface type and returns a `SpecializedBuilderFactory` object that guarantees to return an instance of the specific `ValidationBuilder` sub interface. Thanks to the use of generics, the client API does not have to cast to the `ValidatorBuilder` sub interface.

A `SpecializedBuilderFactory` object can optionally receive a `ValidationProviderResolver` instance.

Example 4.15. SpecializedBuilderFactory interface

```
/**
 * Build implementations of builderType, the specific ValidationBuilder sub interface uniquely identifyin
 * a provider.
 *
 * The requested provider is the first provider suitable for T (as defined in
 * {@link javax.validation.spi.ValidationProvider#isSuitable(Class)}). The list of providers evaluated is
 * returned by {@link ValidationProviderResolver}. If no ValidationProviderResolver is defined, the
 * default ValidationProviderResolver strategy is used.
 *
 *
 * @author Emmanuel Bernard
 */
public interface SpecializedBuilderFactory<T extends ValidatorBuilder<T>> {

    /**
     * Optionally define the provider resolver implementation used.
     * If not defined, use the default ValidationProviderResolver
     *
     * @param resolver ValidationProviderResolver implementation used
     * @return self
     */
    public SpecializedBuilderFactory<T> providerResolver(ValidationProviderResolver resolver);

    /**
     * Determine the provider implementation suitable for builderType and delegate the creation
     * of this specific ValidatorBuilder subclass to the provider.
     *
     * @return a ValidatorBuilder sub interface implementation
     */
    public T getValidatorBuilder();
}
```

`SpecializedBuilderFactory.getValidatorBuilder()` must return the result of `ValidationProvider.createSpecializedValidatorBuilder(BootstrapState state, Class<T extends ValidatorBuilder<T>>)`. The state parameter holds the `ValidationProviderResolver` passed to `SpecializedBuilderFactory`. The builder type passed as a parameter is the builder type passed to `Validation.builderType(Class)`. The validation provider is selected from the builder type according to the algorithm described in (XX).

The `Validation` implementation provided by the Bean Validation provider must not contain any non private attribute or method aside from the three public static bootstrap methods:

- `public static ValidatorBuilder<?> getValidatorBuilder()`
- `public static GenericBuilderFactory defineBootstrapState()`
- `public static <T extends ValidatorBuilder<T>> SpecializedBuilderFactory<T> builderType(Class<T> builderType)`

The bootstrap API is designed to allow complete portability amongst Bean Validation provider implementations. The bootstrap implementation must ensure it can bootstrap third party providers.

4.4.6. Usage

The Bean Validation bootstrap API can be used directly by the application, through the use of a container or by framework in need for validation. In all cases, the following rules apply:

- `ValidatorFactory` is a thread-safe object that should be built once per deployment unit
- `Validator` is not thread-safe and should not be short lived

Containers such as Java EE, Web Bean, dependency injection frameworks, component frameworks are encouraged to propose access to `ValidatorFactory` and `Validator` objects in a way that respects the following rules. For example, injection of `Validator<T>` should be possible.