7. Validation, Data Binding, and Type Conversion

7.1 Introduction

> **JSR-303 Bean Validation**
>
> The Spring Framework supports JSR-303 Bean Validation adapting it to Spring's `Validator` interface.
>
> An application can choose to enable JSR-303 Bean Validation once globally, as described in Section 7.8, “Spring 3 Validation”, and use it exclusively for all validation needs.
>
> An application can also register additional Spring `Validator` instances per `DataBinder` instance, as described in Section 7.8.3, “Configuring a DataBinder”. This may be useful for plugging in validation logic without the use of annotations.

There are pros and cons for considering validation as business logic, and Spring offers a design for validation (and data binding) that does not exclude either one of them. Specifically validation should not be tied to the web tier, should be easy to localize and it should be possible to plug in any validator available. Considering the above, Spring has come up with a `Validator` interface that is both basic ands eminently usable in every layer of an application.

Data binding is useful for allowing user input to be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the so-called `DataBinder` to do exactly that. The `Validator` and the `DataBinder` make up the `validation` package, which is primarily used in but not limited to the MVC framework.

The `BeanWrapper` is a fundamental concept in the Spring Framework and is used in a lot of places. However, you probably will not have the need to use the `BeanWrapper` directly. Because this is reference documentation however, we felt that some

explanation might be in order. We will explain the `BeanWrapper` in this chapter since, if you were going to use it at all, you would most likely do so when trying to bind data to objects.

Spring's DataBinder and the lower-level BeanWrapper both use PropertyEditors to parse and format property values. The `PropertyEditor` concept is part of the JavaBeans specification, and is also explained in this chapter. Spring 3 introduces a "core.convert" package that provides a general type conversion facility, as well as a higher-level "format" package for formatting UI field values. These new packages may be used as simpler alternatives to PropertyEditors, and will also be discussed in this chapter.

7.2 Validation using Spring's `Validator` interface

Spring features a `Validator` interface that you can use to validate objects. The `Validator` interface works using an `Errors` object so that while validating, validators can report validation failures to the `Errors` object.

Let's consider a small data object:

```
public class Person {

  private String name;
  private int age;

  // the usual getters and setters...
}
```

We're going to provide validation behavior for the `Person` class by implementing the following two methods of the `org.springframework.validation.Validator` interface:

- `supports(Class)` - Can this `Validator` validate instances of the supplied `Class`?
- `validate(Object, org.springframework.validation.Errors)` - validates the given object and in case of validation errors, registers those with the given `Errors` object

Implementing a `Validator` is fairly straightforward, especially when you know of the `ValidationUtils` helper class that the Spring Framework also provides.

```
public class PersonValidator implements Validator {

    /**
    * This Validator validates *just* Person instances
    */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

As you can see, the `static` `rejectIfEmpty(..)` method on the `ValidationUtils` class is used to reject the `'name'` property if it is `null` or the empty string. Have a look at the Javadoc for the `ValidationUtils` class to see what functionality it provides besides the example shown previously.

While it is certainly possible to implement a single `Validator` class to validate each of the nested objects in a rich object, it may be better to encapsulate the validation logic for each nested class of object in its own `Validator` implementation. A simple example of a *'rich'* object would be a `Customer` that is composed of two `String` properties (a first and second name) and a complex `Address` object. `Address` objects may be used independently of `Customer` objects, and so a distinct `AddressValidator` has been implemented. If you want your `CustomerValidator` to reuse the logic contained within the

`AddressValidator` class without resorting to copy-and-paste, you can dependency-inject or instantiate an `AddressValidator` within your `CustomerValidator`, and use it like so:

```
public class CustomerValidator implements Validator {

    private final Validator addressValidator;


    public CustomerValidator(Validator addressValidator) {
        if (addressValidator == null) {
            throw new IllegalArgumentException(
              "The supplied [Validator] is required and must not be null.");
        }
        if (!addressValidator.supports(Address.class)) {
            throw new IllegalArgumentException(
              "The supplied [Validator] must support the validation of [Address] instances.");
        }
        this.addressValidator = addressValidator;
    }

    /**
    * This Validator validates Customer instances, and any subclasses of Customer too
    */
    public boolean supports(Class clazz) {
        return Customer.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");
        Customer customer = (Customer) target;
        try {
            errors.pushNestedPath("address");
            ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);
        } finally {
            errors.popNestedPath();
```

```
            errors.popNestedPath();
        }
    }
}
```

Validation errors are reported to the `Errors` object passed to the validator. In case of Spring Web MVC you can use `<spring:bind/>` tag to inspect the error messages, but of course you can also inspect the errors object yourself. More information about the methods it offers can be found from the Javadoc.

7.3 Resolving codes to error messages

We've talked about databinding and validation. Outputting messages corresponding to validation errors is the last thing we need to discuss. In the example we've shown above, we rejected the `name` and the `age` field. If we're going to output the error messages by using a `MessageSource`, we will do so using the error code we've given when rejecting the field ('name' and 'age' in this case). When you call (either directly, or indirectly, using for example the `ValidationUtils` class) `rejectValue` or one of the other `reject` methods from the `Errors` interface, the underlying implementation will not only register the code you've passed in, but also a number of additional error codes. What error codes it registers is determined by the `MessageCodesResolver` that is used. By default, the `DefaultMessageCodesResolver` is used, which for example not only registers a message with the code you gave, but also messages that include the field name you passed to the reject method. So in case you reject a field using `rejectValue("age", "too.darn.old")`, apart from the `too.darn.old` code, Spring will also register `too.darn.old.age` and `too.darn.old.age.int` (so the first will include the field name and the second will include the type of the field); this is done as a convenience to aid developers in targeting error messages and suchlike.

More information on the `MessageCodesResolver` and the default strategy can be found online with the Javadocs for MessageCodesResolver and DefaultMessageCodesResolver respectively.

7.4 Bean manipulation and the `BeanWrapper`

The `org.springframework.beans` package adheres to the JavaBeans standard provided by Sun. A JavaBean is simply a class with a default no-argument constructor, which follows a naming convention where (by way of an example) a property named

`bingoMadness` would have a setter method `setBingoMadness(..)` and a getter method `getBingoMadness()`. For more information about JavaBeans and the specification, please refer to Sun's website (java.sun.com/products/javabeans).

One quite important class in the beans package is the `BeanWrapper` interface and its corresponding implementation (`BeanWrapperImpl`). As quoted from the Javadoc, the `BeanWrapper` offers functionality to set and get property values (individually or in bulk), get property descriptors, and to query properties to determine if they are readable or writable. Also, the `BeanWrapper` offers support for nested properties, enabling the setting of properties on sub-properties to an unlimited depth. Then, the `BeanWrapper` supports the ability to add standard JavaBeans `PropertyChangeListeners` and `VetoableChangeListeners`, without the need for supporting code in the target class. Last but not least, the `BeanWrapper` provides support for the setting of indexed properties. The `BeanWrapper` usually isn't used by application code directly, but by the `DataBinder` and the `BeanFactory`.

The way the `BeanWrapper` works is partly indicated by its name: *it wraps a bean* to perform actions on that bean, like setting and retrieving properties.

7.4.1 Setting and getting basic and nested properties

Setting and getting properties is done using the `setPropertyValue(s)` and `getPropertyValue(s)` methods that both come with a couple of overloaded variants. They're all described in more detail in the Javadoc Spring comes with. What's important to know is that there are a couple of conventions for indicating properties of an object. A couple of examples:

Table 7.1. Examples of properties

Expression	Explanation
`name`	Indicates the property `name` corresponding to the methods `getName()` or `isName()` and `setName(..)`
`account.name`	Indicates the nested property `name` of the property `account` corresponding e.g. to the methods `getAccount().setName()` or `getAccount().getName()`

Expression	Explanation
`account[2]`	Indicates the *third* element of the indexed property `account`. Indexed properties can be of type `array`, `list` or other *naturally ordered* collection
`account[COMPANYNAME]`	Indicates the value of the map entry indexed by the key *COMPANYNAME* of the Map property `account`

Below you'll find some examples of working with the `BeanWrapper` to get and set properties.

(This next section is not vitally important to you if you're not planning to work with the `BeanWrapper` directly. If you're just using the `DataBinder` and the `BeanFactory` and their out-of-the-box implementation, you should skip ahead to the section about `PropertyEditors`.)

Consider the following two classes:

```
public class Company {
    private String name;
    private Employee managingDirector;

    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Employee getManagingDirector() {
        return this.managingDirector;
    }
    public void setManagingDirector(Employee managingDirector) {
        this.managingDirector = managingDirector;
```

```
    }
}
```

```
public class Employee {
    private String name;
    private float salary;

    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
}
```

The following code snippets show some examples of how to retrieve and manipulate some of the properties of instantiated `Companies` and `Employees`:

```
BeanWrapper company = BeanWrapperImpl(new Company());
// setting the company name..
company.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue value = new PropertyValue("name", "Some Company Inc.");
company.setPropertyValue(value);

// ok, let's create the director and tie it to the company:
BeanWrapper jim = BeanWrapperImpl(new Employee());
jim.setPropertyValue("name", "Jim Stravinsky");
```

```
company.setPropertyValue("managingDirector", jim.getWrappedInstance());

// retrieving the salary of the managingDirector through the company
Float salary = (Float) company.getPropertyValue("managingDirector.salary");
```

7.4.2 Built-in `PropertyEditor` implementations

Spring uses the concept of `PropertyEditors` to effect the conversion between an `Object` and a `String`. If you think about it, it sometimes might be handy to be able to represent properties in a different way than the object itself. For example, a `Date` can be represented in a human readable way (as the `String` '`2007-14-09`'), while we're still able to convert the human readable form back to the original date (or even better: convert any date entered in a human readable form, back to `Date` objects). This behavior can be achieved by *registering custom editors*, of type `java.beans.PropertyEditor`. Registering custom editors on a `BeanWrapper` or alternately in a specific IoC container as mentioned in the previous chapter, gives it the knowledge of how to convert properties to the desired type. Read more about `PropertyEditors` in the Javadoc of the `java.beans` package provided by Sun.

A couple of examples where property editing is used in Spring:

- *setting properties on beans* is done using `PropertyEditors`. When mentioning `java.lang.String` as the value of a property of some bean you're declaring in XML file, Spring will (if the setter of the corresponding property has a `Class`-parameter) use the `ClassEditor` to try to resolve the parameter to a `Class` object.
- *parsing HTTP request parameters* in Spring's MVC framework is done using all kinds of `PropertyEditors` that you can manually bind in all subclasses of the `CommandController`.

Spring has a number of built-in `PropertyEditors` to make life easy. Each of those is listed below and they are all located in the `org.springframework.beans.propertyeditors` package. Most, but not all (as indicated below), are registered by default by `BeanWrapperImpl`. Where the property editor is configurable in some fashion, you can of course still register your own variant to override the default one:

Table 7.2. Built-in `PropertyEditors`

Class	Explanation
`ByteArrayPropertyEditor`	Editor for byte arrays. Strings will simply be converted to their corresponding byte representations. Registered by default by `BeanWrapperImpl`.
`ClassEditor`	Parses Strings representing classes to actual classes and the other way around. When a class is not found, an `IllegalArgumentException` is thrown. Registered by default by `BeanWrapperImpl`.
`CustomBooleanEditor`	Customizable property editor for `Boolean` properties. Registered by default by `BeanWrapperImpl`, but, can be overridden by registering custom instance of it as custom editor.
`CustomCollectionEditor`	Property editor for Collections, converting any source `Collection` to a given target `Collection` type.
`CustomDateEditor`	Customizable property editor for java.util.Date, supporting a custom DateFormat. NOT registered by default. Must be user registered as needed with appropriate format.
`CustomNumberEditor`	Customizable property editor for any Number subclass like `Integer`, `Long`, `Float`, `Double`. Registered by default by `BeanWrapperImpl`, but can be overridden by registering custom instance of it as a custom editor.
`FileEditor`	Capable of resolving Strings to `java.io.File` objects. Registered by default by `BeanWrapperImpl`.
`InputStreamEditor`	One-way property editor, capable of taking a text string and producing (via an intermediate `ResourceEditor` and `Resource`) an `InputStream`, so `InputStream` properties may be directly set as Strings. Note that the default usage will not close the `InputStream` for you! Registered by default by `BeanWrapperImpl`.

Class	Explanation
`LocaleEditor`	Capable of resolving Strings to `Locale` objects and vice versa (the String format is [language]_[country]_[variant], which is the same thing the toString() method of Locale provides). Registered by default by `BeanWrapperImpl`.
`PatternEditor`	Capable of resolving Strings to JDK 1.5 `Pattern` objects and vice versa.
`PropertiesEditor`	Capable of converting Strings (formatted using the format as defined in the Javadoc for the java.lang.Properties class) to `Properties` objects. Registered by default by `BeanWrapperImpl`.
`StringTrimmerEditor`	Property editor that trims Strings. Optionally allows transforming an empty string into a `null` value. NOT registered by default; must be user registered as needed.
`URLEditor`	Capable of resolving a String representation of a URL to an actual `URL` object. Registered by default by `BeanWrapperImpl`.

Spring uses the `java.beans.PropertyEditorManager` to set the search path for property editors that might be needed. The search path also includes `sun.bean.editors`, which includes `PropertyEditor` implementations for types such as `Font`, `Color`, and most of the primitive types. Note also that the standard JavaBeans infrastructure will automatically discover `PropertyEditor` classes (without you having to register them explicitly) if they are in the same package as the class they handle, and have the same name as that class, with `'Editor'` appended; for example, one could have the following class and package structure, which would be sufficient for the `FooEditor` class to be recognized and used as the `PropertyEditor` for `Foo`-typed properties.

```
com
  chank
    pop
      Foo
      FooEditor   // the PropertyEditor for the Foo class
```

Note that you can also use the standard `BeanInfo` JavaBeans mechanism here as well (described in not-amazing-detail here). Find below an example of using the `BeanInfo` mechanism for explicitly registering one or more `PropertyEditor` instances with the properties of an associated class.

```
com
  chank
    pop
      Foo
      FooBeanInfo   // the BeanInfo for the Foo class
```

Here is the Java source code for the referenced `FooBeanInfo` class. This would associate a `CustomNumberEditor` with the `age` property of the `Foo` class.

```
public class FooBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class, true);
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age", Foo.class) {
                public PropertyEditor createPropertyEditor(Object bean) {
                    return numberPE;
                };
            };
            return new PropertyDescriptor[] { ageDescriptor };

        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

Registering additional custom `PropertyEditors`

When setting bean properties as a string value, a Spring IoC container ultimately uses standard JavaBeans `PropertyEditors` to convert these Strings to the complex type of the property. Spring pre-registers a number of custom `PropertyEditors` (for example, to convert a classname expressed as a string into a real `Class` object). Additionally, Java's standard JavaBeans `PropertyEditor` lookup mechanism allows a `PropertyEditor` for a class simply to be named appropriately and placed in the same package as the class it provides support for, to be found automatically.

If there is a need to register other custom `PropertyEditors`, there are several mechanisms available. The most manual approach, which is not normally convenient or recommended, is to simply use the `registerCustomEditor()` method of the `ConfigurableBeanFactory` interface, assuming you have a `BeanFactory` reference. Another, slightly more convenient, mechanism is to use a special bean factory post-processor called `CustomEditorConfigurer`. Although bean factory post-processors can be used with `BeanFactory` implementations, the `CustomEditorConfigurer` has a nested property setup, so it is strongly recommended that it is used with the `ApplicationContext`, where it may be deployed in similar fashion to any other bean, and automatically detected and applied.

Note that all bean factories and application contexts automatically use a number of built-in property editors, through their use of something called a `BeanWrapper` to handle property conversions. The standard property editors that the `BeanWrapper` registers are listed in the previous section. Additionally, `ApplicationContexts` also override or add an additional number of editors to handle resource lookups in a manner appropriate to the specific application context type.

Standard JavaBeans `PropertyEditor` instances are used to convert property values expressed as strings to the actual complex type of the property. `CustomEditorConfigurer`, a bean factory post-processor, may be used to conveniently add support for additional `PropertyEditor` instances to an `ApplicationContext`.

Consider a user class `ExoticType`, and another class `DependsOnExoticType` which needs `ExoticType` set as a property:

```
package example;

public class ExoticType {

    private String name;
```

```
    public ExoticType(String name) {
        this.name = name;
    }
}

public class DependsOnExoticType {

    private ExoticType type;

    public void setType(ExoticType type) {
        this.type = type;
    }
}
```

When things are properly set up, we want to be able to assign the type property as a string, which a `PropertyEditor` will behind the scenes convert into an actual `ExoticType` instance:

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

The `PropertyEditor` implementation could look similar to this:

```
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}
```

Finally, we use `CustomEditorConfigurer` to register the new `PropertyEditor` with the `ApplicationContext`, which will then be able to use it as needed:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
    <map>
      <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
    </map>
  </property>
</bean>
```

Using `PropertyEditorRegistrars`

Another mechanism for registering property editors with the Spring container is to create and use a `PropertyEditorRegistrar`. This interface is particularly useful when you need to use the same set of property editors in several different situations: write a corresponding registrar and reuse that in each case. `PropertyEditorRegistrars` work in conjunction with an interface called `PropertyEditorRegistry`, an interface that is implemented by the Spring `BeanWrapper` (and `DataBinder`). `PropertyEditorRegistrars` are particularly convenient when used in conjunction with the `CustomEditorConfigurer` (introduced here), which exposes a property called `setPropertyEditorRegistrars(..)`: `PropertyEditorRegistrars` added to a `CustomEditorConfigurer` in this fashion can easily be shared with `DataBinder` and Spring MVC `Controllers`. Furthermore, it avoids the need for synchronization on custom editors: a `PropertyEditorRegistrar` is expected to create fresh `PropertyEditor` instances for each bean creation attempt.

Using a `PropertyEditorRegistrar` is perhaps best illustrated with an example. First off, you need to create your own `PropertyEditorRegistrar` implementation:

```
package com.foo.editors.spring;

public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {

    public void registerCustomEditors(PropertyEditorRegistry registry) {
```

```
        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

        // you could register as many custom property editors as are required here...
    }
}
```

See also the `org.springframework.beans.support.ResourceEditorRegistrar` for an example `PropertyEditorRegistrar` implementation. Notice how in its implementation of the `registerCustomEditors(..)` method it creates new instances of each property editor.

Next we configure a `CustomEditorConfigurer` and inject an instance of our `CustomPropertyEditorRegistrar` into it:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <ref bean="customPropertyEditorRegistrar"/>
        </list>
    </property>
</bean>

<bean id="customPropertyEditorRegistrar"
      class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>
```

Finally, and in a bit of a departure from the focus of this chapter, for those of you using Spring's MVC web framework, using `PropertyEditorRegistrars` in conjunction with data-binding `Controllers` (such as `SimpleFormController`) can be very convenient. Find below an example of using a `PropertyEditorRegistrar` in the implementation of an `initBinder(..)` method:

```
public final class RegisterUserController extends SimpleFormController {

    private final PropertyEditorRegistrar customPropertyEditorRegistrar;
```

```
    public RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {
        this.customPropertyEditorRegistrar = propertyEditorRegistrar;
    }

    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder)
                        throws Exception {
        this.customPropertyEditorRegistrar.registerCustomEditors(binder);
    }

    // other methods to do with registering a User
}
```

This style of `PropertyEditor` registration can lead to concise code (the implementation of `initBinder(..)` is just one line long!), and allows common `PropertyEditor` registration code to be encapsulated in a class and then shared amongst as many `Controllers` as needed.

7.5 Spring 3 Type Conversion

Spring 3 introduces a `core.convert` package that provides a general type conversion system. The system defines an SPI to implement type conversion logic, as well as an API to execute type conversions at runtime. Within a Spring container, this system can be used as an alternative to PropertyEditors to convert externalized bean property value strings to required property types. The public API may also be used anywhere in your application where type conversion is needed.

7.5.1 Converter SPI

The SPI to implement type conversion logic is simple and strongly typed:

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
```

```
    T convert(S source);

}
```

To create your own Converter, simply implement the interface above. Parameterize S as the type you are converting from, and T as the type you are converting to. For each call to convert(S), the source argument is guaranteed to be NOT null. Your Converter may throw any Exception if conversion fails. An IllegalArgumentException should be thrown to report an invalid source value. Take care to ensure your Converter implementation is thread-safe.

Several converter implementations are provided in the `core.convert.support` package as a convenience. These include converters from Strings to Numbers and other common types. Consider `StringToInteger` as an example Converter implementation:

```
package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }

}
```

7.5.2 ConverterFactory

When you need to centralize the conversion logic for an entire class hierarchy, for example, when converting from String to java.lang.Enum objects, implement `ConverterFactory`:

```
package org.springframework.core.convert.converter;

public interface ConverterFactory<S, R> {
```

```
    <T extends R> Converter<S, T> getConverter(Class<T> targetType);

}
```

Parameterize S to be the type you are converting from and R to be the base type defining the *range* of classes you can convert to. Then implement getConverter(Class<T>), where T is a subclass of R.

Consider the `StringToEnum` ConverterFactory as an example:

```
package org.springframework.core.convert.support;

final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

    public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {
        return new StringToEnumConverter(targetType);

    }

    private final class StringToEnumConverter<T extends Enum> implements Converter<String, T> {

        private Class<T> enumType;

        public StringToEnumConverter(Class<T> enumType) {
            this.enumType = enumType;
        }

        public T convert(String source) {
            return (T) Enum.valueOf(this.enumType, source.trim());
        }
    }
}
```

7.5.3 GenericConverter

When you require a sophisticated Converter implementation, consider the GenericConverter interface. With a more flexible but less strongly typed signature, a GenericConverter supports converting between multiple source and target types. In addition, a GenericConverter makes available source and target field context you can use when implementing your conversion logic. Such context allows a type conversion to be driven by a field annotation, or generic information declared on a field signature.

```
package org.springframework.core.convert.converter;

public interface GenericConverter {

    public Set<ConvertiblePair> getConvertibleTypes();

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

To implement a GenericConverter, have getConvertibleTypes() return the supported source->target type pairs. Then implement convert(Object, TypeDescriptor, TypeDescriptor) to implement your conversion logic. The source TypeDescriptor provides access to the source field holding the value being converted. The target TypeDescriptor provides access to the target field where the converted value will be set.

A good example of a GenericConverter is a converter that converts between a Java Array and a Collection. Such an ArrayToCollectionConverter introspects the field that declares the target Collection type to resolve the Collection's element type. This allows each element in the source array to be converted to the Collection element type before the Collection is set on the target field.

Because GenericConverter is a more complex SPI interface, only use it when you need it. Favor Converter or ConverterFactory for basic type conversion needs.

ConditionalGenericConverter

Sometimes you only want a Converter to execute if a specific condition holds true. For example, you might only want to execute a Converter if a specific annotation is present on the target field. Or you might only want to execute a Converter if a specific method, such as static valueOf method, is defined on the target class. ConditionalGenericConverter is an subinterface of GenericConverter that allows you to define such custom matching criteria:

```
public interface ConditionalGenericConverter extends GenericConverter {

    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

A good example of a ConditionalGenericConverter is an EntityConverter that converts between an persistent entity identifier and an entity reference. Such a EntityConverter might only match if the target entity type declares a static finder method e.g. findAccount(Long). You would perform such a finder method check in the implementation of matches(TypeDescriptor, TypeDescriptor).

7.5.4 ConversionService API

The ConversionService defines a unified API for executing type conversion logic at runtime. Converters are often executed behind this facade interface:

```
package org.springframework.core.convert;

public interface ConversionService {

    boolean canConvert(Class<?> sourceType, Class<?> targetType);

    <T> T convert(Object source, Class<T> targetType);

    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
```

```
}
```

Most ConversionService implementations also implement `ConverterRegistry`, which provides an SPI for registering converters. Internally, a ConversionService implementation delegates to its registered converters to carry out type conversion logic.

A robust ConversionService implementation is provided in the `core.convert.support` package. `GenericConversionService` is the general-purpose implementation suitable for use in most environments. `ConversionServiceFactory` provides a convenient factory for creating common ConversionService configurations.

7.5.5 Configuring a ConversionService

A ConversionService is a stateless object designed to be instantiated at application startup, then shared between multiple threads. In a Spring application, you typically configure a ConversionService instance per Spring container (or ApplicationContext). That ConversionService will be picked up by Spring and then used whenever a type conversion needs to be performed by the framework. You may also inject this ConversionService into any of your beans and invoke it directly.

If no ConversionService is registered with Spring, the original PropertyEditor-based system is used.

To register a default ConversionService with Spring, add the following bean definition with id `conversionService`:

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

A default ConversionService can convert between strings, numbers, enums, collections, maps, and other common types. To supplement or override the default converters with your own custom converter(s), set the `converters` property. Property values may implement either of the Converter, ConverterFactory, or GenericConverter interfaces.

```
<bean id="conversionService"
```

```
      class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <list>
            <bean class="example.MyCustomConverter"/>
        </list>
    </property>
</bean>
```

It is also common to use a ConversionService within a Spring MVC application. See Section 7.6.5, “Configuring Formatting in Spring MVC” for details on use with `<mvc:annotation-driven/>`.

In certain situations you may wish to apply formatting during conversion. See Section 7.6.3, “FormatterRegistry SPI” for details on using `FormattingConversionServiceFactoryBean`.

7.5.6 Using a ConversionService programmatically

To work with a ConversionService instance programmatically, simply inject a reference to it like you would for any other bean:

```
@Service
public class MyService {

    @Autowired
    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;
    }

    public void doIt() {
        this.conversionService.convert(...)
    }
}
```

7.6 Spring 3 Field Formatting

As discussed in the previous section, `core.convert` is a general-purpose type conversion system. It provides a unified ConversionService API as well as a strongly-typed Converter SPI for implementing conversion logic from one type to another. A Spring Container uses this system to bind bean property values. In addition, both the Spring Expression Language (SpEL) and DataBinder use this system to bind field values. For example, when SpEL needs to coerce a `Short` to a `Long` to complete an `expression.setValue(Object bean, Object value)` attempt, the core.convert system performs the coercion.

Now consider the type conversion requirements of a typical client environment such as a web or desktop application. In such environments, you typically convert *from String* to support the client postback process, as well as back *to String* to support the view rendering process. In addition, you often need to localize String values. The more general *core.convert* Converter SPI does not address such *formatting* requirements directly. To directly address them, Spring 3 introduces a convenient Formatter SPI that provides a simple and robust alternative to PropertyEditors for client environments.

In general, use the Converter SPI when you need to implement general-purpose type conversion logic; for example, for converting between a java.util.Date and and java.lang.Long. Use the Formatter SPI when you're working in a client environment, such as a web application, and need to parse and print localized field values. The ConversionService provides a unified type conversion API for both SPIs.

7.6.1 Formatter SPI

The Formatter SPI to implement field formatting logic is simple and strongly typed:

```
package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

Where Formatter extends from the Printer and Parser building-block interfaces:

```
public interface Printer<T> {
    String print(T fieldValue, Locale locale);
}
```

```
import java.text.ParseException;

public interface Parser<T> {
    T parse(String clientValue, Locale locale) throws ParseException;
}
```

To create your own Formatter, simply implement the Formatter interface above. Parameterize T to be the type of object you wish to format, for example, `java.util.Date`. Implement the `print()` operation to print an instance of T for display in the client locale. Implement the `parse()` operation to parse an instance of T from the formatted representation returned from the client locale. Your Formatter should throw a ParseException or IllegalArgumentException if a parse attempt fails. Take care to ensure your Formatter implementation is thread-safe.

Several Formatter implementations are provided in `format` subpackages as a convenience. The `number` package provides a NumberFormatter, CurrencyFormatter, and PercentFormatter to format java.lang.Number objects using a java.text.NumberFormat. The `datetime` package provides a DateFormatter to format java.util.Date objects with a java.text.DateFormat. The `datetime.joda` package provides comprehensive datetime formatting support based on the Joda Time library.

Consider `DateFormatter` as an example `Formatter` implementation:

```
package org.springframework.format.datetime;

public final class DateFormatter implements Formatter<Date> {

    private String pattern;

    public DateFormatter(String pattern) {
        this.pattern = pattern;
```

```
    }

    public String print(Date date, Locale locale) {
        if (date == null) {
            return "";
        }
        return getDateFormat(locale).format(date);
    }

    public Date parse(String formatted, Locale locale) throws ParseException {
        if (formatted.length() == 0) {
            return null;
        }
        return getDateFormat(locale).parse(formatted);
    }

    protected DateFormat getDateFormat(Locale locale) {
        DateFormat dateFormat = new SimpleDateFormat(this.pattern, locale);
        dateFormat.setLenient(false);
        return dateFormat;
    }

}
```

The Spring team welcomes community-driven Formatter contributions; see http://jira.springframework.org to contribute.

7.6.2 Annotation-driven Formatting

As you will see, field formatting can be configured by field type or annotation. To bind an Annotation to a formatter, implement AnnotationFormatterFactory:

```
package org.springframework.format;
```

```
public interface AnnotationFormatterFactory<A extends Annotation> {

    Set<Class<?>> getFieldTypes();

    Printer<?> getPrinter(A annotation, Class<?> fieldType);

    Parser<?> getParser(A annotation, Class<?> fieldType);

}
```

Parameterize A to be the field annotationType you wish to associate formatting logic with, for example `org.springframework.format.annotation.DateTimeFormat`. Have `getFieldTypes()` return the types of fields the annotation may be used on. Have `getPrinter()` return a Printer to print the value of an annotated field. Have `getParser()` return a Parser to parse a clientValue for an annotated field.

The example AnnotationFormatterFactory implementation below binds the @NumberFormat Annotation to a formatter. This annotation allows either a number style or pattern to be specified:

```
public final class NumberFormatAnnotationFormatterFactory
        implements AnnotationFormatterFactory<NumberFormat> {

    public Set<Class<?>> getFieldTypes() {
        return new HashSet<Class<?>>(asList(new Class<?>[] {
            Short.class, Integer.class, Long.class, Float.class,
            Double.class, BigDecimal.class, BigInteger.class }));
    }

    public Printer<Number> getPrinter(NumberFormat annotation, Class<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

    public Parser<Number> getParser(NumberFormat annotation, Class<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }
```

```
    private Formatter<Number> configureFormatterFrom(NumberFormat annotation,
                                                     Class<?> fieldType) {
        if (!annotation.pattern().isEmpty()) {
            return new NumberFormatter(annotation.pattern());
        } else {
            Style style = annotation.style();
            if (style == Style.PERCENT) {
                return new PercentFormatter();
            } else if (style == Style.CURRENCY) {
                return new CurrencyFormatter();
            } else {
                return new NumberFormatter();
            }
        }
    }
}
```

To trigger formatting, simply annotate fields with @NumberFormat:

```
public class MyModel {

    @NumberFormat(style=Style.CURRENCY)
    private BigDecimal decimal;

}
```

Format Annotation API

A portable format annotation API exists in the `org.springframework.format.annotation` package. Use @NumberFormat to format java.lang.Number fields. Use @DateTimeFormat to format java.util.Date, java.util.Calendar, java.util.Long, or Joda Time fields.

The example below uses @DateTimeFormat to format a java.util.Date as a ISO Date (yyyy-MM-dd):

```
public class MyModel {

    @DateTimeFormat(iso=ISO.DATE)
    private Date date;

}
```

7.6.3 FormatterRegistry SPI

The FormatterRegistry is an SPI for registering formatters and converters. `FormattingConversionService` is an implementation of FormatterRegistry suitable for most environments. This implementation may be configured programmatically or declaratively as a Spring bean using `FormattingConversionServiceFactoryBean`. Because this implementation also implements `ConversionService`, it can be directly configured for use with Spring's DataBinder and the Spring Expression Language (SpEL).

Review the FormatterRegistry SPI below:

```
package org.springframework.format;

public interface FormatterRegistry extends ConverterRegistry {

    void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);

    void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

    void addFormatterForFieldType(Formatter<?> formatter);

    void addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);

}
```

As shown above, Formatters can be registered by fieldType or annotation.

The FormatterRegistry SPI allows you to configure Formatting rules centrally, instead of duplicating such configuration across your Controllers. For example, you might want to enforce that all Date fields are formatted a certain way, or fields with a specific annotation are formatted in a certain way. With a shared FormatterRegistry, you define these rules once and they are applied whenever formatting is needed.

7.6.4 FormatterRegistrar SPI

The FormatterRegistrar is an SPI for registering formatters and converters through the FormatterRegistry:

```
package org.springframework.format;

public interface FormatterRegistrar {

    void registerFormatters(FormatterRegistry registry);

}
```

A FormatterRegistrar is useful when registering multiple related converters and formatters for a given formatting category, such as Date formatting. It can also be useful where declarative registration is insufficient. For example when a formatter needs to be indexed under a specific field type different from its own <T> or when registering a Printer/Parser pair. The next section provides more information on converter and formatter registration.

7.6.5 Configuring Formatting in Spring MVC

In a Spring MVC application, you may configure a custom ConversionService instance explicitly as an attribute of the `annotation-driven` element of the MVC namespace. This ConversionService will then be used anytime a type conversion is required during Controller model binding. If not configured explicitly, Spring MVC will automatically register default formatters and converters for common types such as numbers and dates.

To rely on default formatting rules, no custom configuration is required in your Spring MVC config XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven/>

</beans>
```

With this one-line of configuration, default formatters for Numbers and Date types will be installed, including support for the @NumberFormat and @DateTimeFormat annotations. Full support for the Joda Time formatting library is also installed if Joda Time is present on the classpath.

To inject a ConversionService instance with custom formatters and converters registered, set the conversion-service attribute and then specify custom converters, formatters, or FormatterRegistrars as properties of the FormattingConversionServiceFactoryBean:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
```

```
    <mvc:annotation-driven conversion-service="conversionService"/>

    <bean id="conversionService"
          class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="org.example.MyConverter"/>
            </set>
        </property>
        <property name="formatters">
            <set>
                <bean class="org.example.MyFormatter"/>
                <bean class="org.example.MyAnnotationFormatterFactory"/>
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean class="org.example.MyFormatterRegistrar"/>
            </set>
        </property>
    </bean>

</beans>
```

See Section 7.6.4, “FormatterRegistrar SPI” and the `FormattingConversionServiceFactoryBean` for more information on when to use FormatterRegistrars.

7.7 Configuring a global date & time format

By default, date and time fields that are not annotated with `@DateTimeFormat` are converted from strings using the the `DateFormat.SHORT` style. If you prefer, you can change this by defining your own global format.

You will need to ensure that Spring does not register default formatters, and instead you should register all formatters manually. Use the `org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar` or `org.springframework.format.datetime.DateFormatterRegistrar` class depending on whether you use the Joda Time library.

For example, the following Java configuration will register a global '`yyyyMMdd`' format. This example does not depend on the Joda Time library:

```
@Configuration
public class AppConfig {

  @Bean
  public FormattingConversionService conversionService() {

    // Use the DefaultFormattingConversionService but do not register defaults
    DefaultFormattingConversionService conversionService = new DefaultFormattingConversionService(false);

    // Ensure @NumberFormat is still supported
    conversionService.addFormatterForFieldAnnotation(new NumberFormatAnnotationFormatterFactory());


    // Register date conversion with a specific global format
    DateFormatterRegistrar registrar = new DateFormatterRegistrar();
    registrar.setFormatter(new DateFormatter("yyyyMMdd"));
    registrar.registerFormatters(conversionService);

    return conversionService;
  }
}
```

If you prefer XML based configuration you can use a `FormattingConversionServiceFactoryBean`. Here is the same example, this time using Joda Time:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd>

    <bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryB
        <property name="registerDefaultFormatters" value="false" />
        <property name="formatters">
        <set>
            <bean class="org.springframework.format.number.NumberFormatAnnotationFormatterFactory" />
        </set>
        </property>
        <property name="formatterRegistrars">
        <set>
          <bean class="org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar">
              <property name="dateFormatter">
                  <bean class="org.springframework.format.datetime.joda.DateTimeFormatterFactoryBean">
                      <property name="pattern" value="yyyyMMdd"/>
                  </bean>
              </property>
          </bean>
      </set>
      </property>
    </bean>
  </beans>
```

Joda Time provides separate distinct types to represent `date`, `time` and `date-time` values. The `dateFormatter`, `timeFormatter` and `dateTimeFormatter` properties of the `JodaTimeFormatterRegistrar` should be used to configure the different formats for each type. The `DateTimeFormatterFactoryBean` provides a convenient way to create formatters.

If you are using Spring MVC remember to explicitly configure the conversion service that is used. For Java based `@Configuration` this means extending the `WebMvcConfigurationSupport` class and overriding the `mvcConversionService()` method. For XML you should use the `'conversion-service'` attribute of the `mvc:annotation-driven` element. See Section 7.6.5, “Configuring Formatting in Spring MVC” for details.

7.8 Spring 3 Validation

Spring 3 introduces several enhancements to its validation support. First, the JSR-303 Bean Validation API is now fully supported. Second, when used programmatically, Spring's DataBinder can now validate objects as well as bind to them. Third, Spring MVC now has support for declaratively validating @Controller inputs.

7.8.1 Overview of the JSR-303 Bean Validation API

JSR-303 standardizes validation constraint declaration and metadata for the Java platform. Using this API, you annotate domain model properties with declarative validation constraints and the runtime enforces them. There are a number of built-in constraints you can take advantage of. You may also define your own custom constraints.

To illustrate, consider a simple PersonForm model with two properties:

```
public class PersonForm {
    private String name;
    private int age;
}
```

JSR-303 allows you to define declarative validation constraints against such properties:

```
public class PersonForm {

    @NotNull
```

```
    @Size(max=64)
    private String name;

    @Min(0)
    private int age;

}
```

When an instance of this class is validated by a JSR-303 Validator, these constraints will be enforced.

For general information on JSR-303, see the Bean Validation Specification. For information on the specific capabilities of the default reference implementation, see the Hibernate Validator documentation. To learn how to setup a JSR-303 implementation as a Spring bean, keep reading.

7.8.2 Configuring a Bean Validation Implementation

Spring provides full support for the JSR-303 Bean Validation API. This includes convenient support for bootstrapping a JSR-303 implementation as a Spring bean. This allows for a `javax.validation.ValidatorFactory` or `javax.validation.Validator` to be injected wherever validation is needed in your application.

Use the `LocalValidatorFactoryBean` to configure a default JSR-303 Validator as a Spring bean:

```
<bean id="validator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

The basic configuration above will trigger JSR-303 to initialize using its default bootstrap mechanism. A JSR-303 provider, such as Hibernate Validator, is expected to be present in the classpath and will be detected automatically.

Injecting a Validator

`LocalValidatorFactoryBean` implements both `javax.validation.ValidatorFactory` and `javax.validation.Validator`, as well as Spring's `org.springframework.validation.Validator`. You may inject a

reference to either of these interfaces into beans that need to invoke validation logic.

Inject a reference to `javax.validation.Validator` if you prefer to work with the JSR-303 API directly:

```
import javax.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
```

Inject a reference to `org.springframework.validation.Validator` if your bean requires the Spring Validation API:

```
import org.springframework.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;

}
```

Configuring Custom Constraints

Each JSR-303 validation constraint consists of two parts. First, a @Constraint annotation that declares the constraint and its configurable properties. Second, an implementation of the `javax.validation.ConstraintValidator` interface that implements the constraint's behavior. To associate a declaration with an implementation, each @Constraint annotation references a corresponding ValidationConstraint implementation class. At runtime, a `ConstraintValidatorFactory` instantiates the referenced implementation when the constraint annotation is encountered in your domain model.

By default, the `LocalValidatorFactoryBean` configures a `SpringConstraintValidatorFactory` that uses Spring to create ConstraintValidator instances. This allows your custom ConstraintValidators to benefit from dependency injection like any other Spring bean.

Shown below is an example of a custom @Constraint declaration, followed by an associated `ConstraintValidator` implementation that uses Spring for dependency injection:

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=MyConstraintValidator.class)
public @interface MyConstraint {
}
```

```
import javax.validation.ConstraintValidator;

public class MyConstraintValidator implements ConstraintValidator {

    @Autowired;
    private Foo aDependency;

    ...
}
```

As you can see, a ConstraintValidator implementation may have its dependencies @Autowired like any other Spring bean.

Additional Configuration Options

The default `LocalValidatorFactoryBean` configuration should prove sufficient for most cases. There are a number of other configuration options for various JSR-303 constructs, from message interpolation to traversal resolution. See the JavaDocs of `LocalValidatorFactoryBean` for more information on these options.

7.8.3 Configuring a DataBinder

Since Spring 3, a DataBinder instance can be configured with a Validator. Once configured, the Validator may be invoked by calling `binder.validate()`. Any validation Errors are automatically added to the binder's BindingResult.

When working with the DataBinder programmatically, this can be used to invoke validation logic after binding to a target object:

```
Foo target = new Foo();
DataBinder binder = new DataBinder(target);
binder.setValidator(new FooValidator());

// bind to the target object
binder.bind(propertyValues);

// validate the target object
binder.validate();

// get BindingResult that includes any validation errors
BindingResult results = binder.getBindingResult();
```

A DataBinder can also be configured with multiple `Validator` instances via `dataBinder.addValidators` and `dataBinder.replaceValidators`. This is useful when combining globally configured JSR-303 Bean Validation with a Spring `Validator` configured locally on a DataBinder instance. See the section called “Configuring a Validator for use by Spring MVC”.

7.8.4 Spring MVC 3 Validation

Beginning with Spring 3, Spring MVC has the ability to automatically validate @Controller inputs. In previous versions it was up to the developer to manually invoke validation logic.

Triggering @Controller Input Validation

To trigger validation of a @Controller input, simply annotate the input argument as @Valid:

```
@Controller
public class MyController {

    @RequestMapping("/foo", method=RequestMethod.POST)
    public void processFoo(@Valid Foo foo) { /* ... */ }
```

Spring MVC will validate a @Valid object after binding so-long as an appropriate Validator has been configured.

The @Valid annotation is part of the standard JSR-303 Bean Validation API, and is not a Spring-specific construct.

Configuring a Validator for use by Spring MVC

The Validator instance invoked when a @Valid method argument is encountered may be configured in two ways. First, you may call binder.setValidator(Validator) within a @Controller's @InitBinder callback. This allows you to configure a Validator instance per @Controller class:

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(new FooValidator());
    }

    @RequestMapping("/foo", method=RequestMethod.POST)
    public void processFoo(@Valid Foo foo) { ... }

}
```

Second, you may call setValidator(Validator) on the global WebBindingInitializer. This allows you to configure a Validator instance across all @Controllers. This can be achieved easily by using the Spring MVC namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven validator="globalValidator"/>

</beans>
```

To combine a global and a local validator, configure the global validator as shown above and then add a local validator:

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

Configuring a JSR-303 Validator for use by Spring MVC

With JSR-303, a single `javax.validation.Validator` instance typically validates *all* model objects that declare validation constraints. To configure a JSR-303-backed Validator with Spring MVC, simply add a JSR-303 Provider, such as Hibernate Validator, to your classpath. Spring MVC will detect it and automatically enable JSR-303 support across all Controllers.

The Spring MVC configuration required to enable JSR-303 support is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- JSR-303 support will be detected on classpath and enabled automatically -->
    <mvc:annotation-driven/>

</beans>
```

With this minimal configuration, anytime a @Valid @Controller input is encountered, it will be validated by the JSR-303 provider. JSR-303, in turn, will enforce any constraints declared against the input. Any ConstraintViolations will automatically be exposed as errors in the BindingResult renderable by standard Spring MVC form tags.

Prev Up Next

6. Resources Home 8. Spring Expression Language (SpEL)