jDTO Binder 1.0-RC User's Guide

Juan Alberto López Cavallotti

March 25, 2012

http://www.jdto.org

Contents

1 Introduction

Welcome to jDTO Binder framework, the main goal of jDTO Binder is to leverage the repetitive work it takes to use the DTO pattern for robust software architectures.

jDTO Binder transforms manual shallow and deep copy processes into a declarative process. By default, the framework takes the shallow copy approach, but with mechanisms such as cascading and cloning the user may achieve deep copying.

Mainly the incorrect use of DTOs have some strong disadvantages:

- Multi powerful DTOs and the lost of lazy loading.
- High memory footprint (and unnecessary).
- Hard to maintain service layer API's.

The discussion of wether the DTOs are useful or not is left out of the picture. It depends specially on the architect and the size of the application. As a personal opinion I like mixed architectures, there are moments where DTOs are useful and there are moments where DTOs are a heavy load.

1.1 What's new in 1.0-RC

jDTO Binder 1.0-RC introduces new features:

- New namespace in packaging to match the framework website.
- New maven artifact namespace to match the framework website.
- Updated documentation heading towards the final release.
- Updated the location of the maven repository.

Issues solved:

- With the new maven repository, the server answers with a 404 error when looking for missing resources so maven won't blacklist it.

1.2 Getting the Source

Currently, jDTO Binder is hosted as an open source GitHub project and licensed with the Apache 2 open source license. You can find the source code deployed on jDTO Binder maven repository or in GitHub. The GitHub project url is:

https://github.com/juancavallotti/jDTO-Binder

You may contribute by submitting a bug or extending the framework in different ways, if you want to contribute please take a look at http://www.jdto.org/#contribute.

2 Quick Start

To start using jDTO Binder you'll have to add it to your maven dependencies, this can be done by adding it to your pom.xml under the dependencies section, you also want to add commons-lang and slf4j:

```
<dependencies>
...
    <dependency>
        <groupId>org.jdto</groupId>
        <artifactId>jdto</artifactId>
        <version>1.0-RC</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.6.2</version>
    </dependency>
    <dependency>
        <groupId>commons-lang</groupId>
        <artifactId>commons-lang</artifactId>
        <version>2.6</version>
    </dependency>
...
<dependencies>
```

You also want to add the jDTO Binder repository to your pom.xml file:

```
<repositories>
    <repository>
        <id>org.jdto</id>
        <name>jdto</name>
        <url>http://maven2.jdto.org/</url>
        <layout>default</layout>
    </repository>
</repositories>
```

This's all what's required in terms of dependencies, so now we can start binding DTOs.

2.1 Bootstrapping the Binder

In order to bind DTOs, a binder must be bootstrapped. At the moment two types of binders are supported: the core binder, and the spring framework binder[1].

The following snippet shows how to bootstrap the core binder, the core binder is

[1] In order to use the spring framework binder, you should add the spring framework dependency.

kept as a singleton instance, so it's safe to call the `getBinder()` method at any time, and the same instance will be returned.

```
1 //init the binder as a singleton.
2 DTOBinder binder = DTOBinderFactory.getBinder();
```

You may also bootstrap the DTO binder based on an XML configuration file, for this matter you'll have to provide an `InputStream` instance pointing to the XML configuration file, the following is an example of how you can bootstrap the binder this way:

```
1 DTOBinder binder = DTOBinderFactory.buildBinder(
2     DTOBinder.class.getResourceAsStream("/xmlmappingtest.xml"));
```

By default, the framework looks for a file named /jdto-mappings.xml on the classpath, if this file is present and no other XML file has been specified, then it will use XML configuration instead of annotations with the configuration specified in /jdto-mappings.xml file.

The bean analysis over XML file is kind of expensive (not much but kind of) so to get the best performance you should keep that bean as a singleton.

2.2 Basic DTO Binding

Once we have our binder instance bootstrapped, we can start binding objects. jDTO Binder uses the "convention over configuration" methodology, therefore if you don't add any kind of mapping it will assume default values as a convention.

In the following example there's one property bound by convention and the other one bound by configuration.

```
1 //bind entities
2 MixedEntity entity = new MixedEntity();
3 entity.setSomeString("hello world!");
4 entity.setSomeInt(15);
5
6 BasicDTO dto = binder.bindFromBusinessObject(BasicDTO.class, entity);
7
8 logger.info(dto.toString());
```

The relevant entity and DTO declarations are:

```
1  public class MixedEntity {
2     private String someString;
3     private int someInt;
4     private double someDouble;
5     private Date someDate;
```

```
6
7       ... //getters and setters
8   }
9
10  public class BasicDTO {
11      private String someString;
12
13      @Source("someInt")
14      private int personAge;
15
16      ... //getters and setters
17  }
```

You may want to keep track of the `MixedEntity` structure for it will be used on the future to demonstrate features of jDTO Binder.

The first attribute `someString` is bound by convention, and the second one is bound by configuration using the `@Source` annotation.

If the object passed as entity is an implementation of `java.util.Map`, then the values will be read as keys of the map instead of calling the object's getters, this means you can populate an object from a map, this also applies if a cascaded property is a map.

3 Integration

3.1 Integration with the Spring Framework

jDTO Binder provides out-of-the-box integration with the spring framework. By default jDTO Binder uses annotation-based configuration for the binder, here is how to bootstrap it to be used within the spring framework xml configuration file:

```
<!-- THE DTO BINDER BEAN -->
<bean id="dtobinder"
      class="org.jdto.spring.SpringDTOBinder" />
```

You may want to use an XML configuration file instead of regular annotations. The spring framework integration provides a way to configure the DTO Binder instance to read the xml configuration file. By default, it will try to look for the /jdto-mappings.xml file on the class path, this can be changed by implicitly specifying a configuration file:

```
<!-- THE DTO BINDER BEAN -->
<bean class="org.jdto.spring.SpringDTOBinder">
    <property name="xmlConfig" value="classpath:/dtos.xml" />
</bean>
```

Note: The binder instance is of the kind of instances you want to keep as singleton. This is because even though it can analyze a bean on the fly, it caches it's metadata to

have a quicker access on the future.

Once configured, you can inject the binder bean as usual, for example:

```
1 @Autowired
2 private DTOBinder binder;
```

3.2 Integration with CDI

jDTO Binder library provides a CDI jar and also a way to inject the DTO binder instance into your own beans. Due to some restrictions on the design of CDI, it was a decision to make this integration as a separate library, so you must change your pom dependencies to:

```
<dependency>
    <groupId>org.jdto</groupId>
    <artifactId>jdto-cdi</artifactId>
    <version>1.0-RC</version>
</dependency>
```

In the following example is shown the typical case of injection via CDI:

```
1 @Inject
2 private DTOBinder binder;
```

The jdto-cdi dependency has as implicit dependency the jdto framework so you should not have to add it explicitly, nevertheless you may add it if that is your taste.

4 Simple Field Binding

To bind simple fields you want to use the `@Source` annotation type. This annotation type can take four parameters but only three are commented in this section:

- value: Indicates the source field to read from, can be a property path.
- merger: An implementation of `SinglePropertyValueMerger` which will take care of the transformation of this item as a single thing.
- mergerParam: A string param which may help the merger to decide how to convert the value.

Users are encouraged to create their own implementations. **Important Note**: The user should see the value mergers as singleton, therefore the use of instance variables is discouraged unless you know what you're doing.

4.1 List of Built-in field Mergers

The following is a complete list of the built-in single field mergers and a brief explanation:

- `AgeMerger`: Evaluates the age in days, weeks or years of a date or calendar instance.
- `CloneMerger`: Call clone in cloneable objects.
- `DateFormatMerger`: Formats a Date or Calendar instance by applying a format String.
- `DecimalFormatMerger`: Format any number by applying a format String.
- `EnumMerger`: Convert an enum literal to it's String representation.
- `IdentityPropertyValueMerger`: Default merger, returns the same instance of the value.
- `StringFormatMerger`: Format the value by using a format string (String.format).
- `ExpressionMerger`: Evaluate a math expression out out literal values and properties of the input bean (or the actual value if the input does not represent a bean).
- `PropertyCollectionMerger`: Converts a collection of objects into a collection of one property of those objects.
- `SumMerger`: Add all of the items of a collection (or some property of it) into a single double.
- `SumExpressionMerger`: Add all the results of an expression evaluation for each value of a bean collection or array.
- `SumProductMerger`: Add all the results of a multiplication between properties for each value of a bean collection or array, for example an SQL equivalent would be: `SELECT sum(itemPrice * amount * taxRate) FROM billItems` . This is a convenience implementation which should be picked instead of `SumExpressionMerger` because is more efficient.
- `ToStringMerger`: Convert any object into it's string representation by calling toString.

Note about expression evaluation: Currently the expression evaluation API supports only five operators: addition (+), subtraction (-), multiplication (*), division(/), and pow (^). Also it supports negative numbers and any amount of balanced parenthesis. If the expression is not well formed, the framework will throw an `IllegalArgumentException`.

Here is an example usage of the `DateFormatMerger`, it will output something like "2011/10/11".

```
@Source(value = "someDate",
    merger=DateFormatMerger.class, mergerParam="yyyy/MM/dd")
private String formattedDate;
```

4.2 Transient Values

There are cases when we want to ignore some of the fields of a DTO. In these cases the user may add the `@DTOTransient` annotation to the field and it will be ignored by the binder.

5 Binding a List of Business Objects

jDTO Binder is capable of binding whole lists of business objects to DTOs. To do this, the binder has an utility method to bind a list of business objects: `bindFromBusinessObjectList`.

In the following sample code you can see how a list of business objects is converted into a List of DTOs. jDTO Binder has adopted the `List` collection as favorite type because it has the concept of insertion-oder, so if you had the business objects previously sorted, then the order remains. Also the framework needs a collection that has the possibility to access elements by index, so `List` was the right choice.

```
LinkedList<SimpleEntity> simpleEntities =
    new LinkedList<SimpleEntity>();
simpleEntities.add(new SimpleEntity("simple 1", 12, 45.56, true));
simpleEntities.add(new SimpleEntity("simple 2", 34, 56.67, false));

List<FormatDTO> dtos = binder.bindFromBusinessObjectList(FormatDTO.
    class, simpleEntities);
```

6 Cascade Binding

jDTO Binder by default copies values, it does not clone instances so is up to the value merger object to decide wether to clone, format, duplicate, or anything else. There are situations where you build a DTO (for example a Bill DTO) which is related to a single or a list of other DTOs. In this case the deep copy process will fail producing unexpected results. For this cases the framework provides an annotation type `DTOCascade` to instruct the binder it should build a DTO related instance.

The DTO Cascading feature supports different kinds of source fields:

- Single Value: A single association can be used as a source.
- Collection: Any type of collection can be used as source.
- Array: Any array can be used as a source.

The target DTO type is inferred by convention or configuration. By convention the following rules apply:

- If the target field is not a collection or array, then its type is used as the resulting DTO type.
- If the target field is a collection, then the generic type parameter is used as the resulting DTO type. If the generic type parameter is not present, the user will have to provide it as a configuration option.
- If the target field is an array, then the type of the components is used to create a DTO.
- For both collection and array targets, the source must be a collection or the ValueMerger must produce a collection.

The target DTO type can be configured as a parameter of the `DTOCascade` annotation type. The following example illustrates some usage of DTO Cascading.

```
1  public class ComplexArrayDTO {
2
3      @DTOCascade
4      @Source("sourceList")
5      private FormatDTO[] formatDtos;
6      ... // GETTERS AND SETTERS
7  }
```

7 Compound Source Binding

jDTO Binder supports composing the value of a target field out of multiple sources. For this purpose it provides the `@Sources` annotation type and the `MultiPropertyValueMerger` interface to merge the sources. The user can safely rely on the parameters sent to the value merger are in the same order as defined on the `@Sources` annotation type.

The default value merger for the `@Sources` annotation type does not merge values, it just returns the first not-null element received or null if none.

7.1 List of Built-in Multi Property Value Mergers

- `FirstObjectPropertyValueMerger`: This is the default merger, it returns the first non-null value.
- `StringFormatMerger` as described before, this merger uses the String.format method to merge all the provided values into a single formatted string.

The following example illustrates how Multi Property Value Mergers can be used for both, single and multiple source configurations:

```
1  public class FormatDTO {
2
3      @Source(value="aDouble", merger=StringFormatMerger.class,
4          mergerParam="$ %.2f")
5      private String price;
6
7      @Sources(value={@Source("aDouble"), @Source("anInt")},
8              merger=StringFormatMerger.class, mergerParam="%.2f %08d")
9      private String compound;
10
11     ... // GETTERS AND SETTERS
12 }
```

8 Multiple Business Objects Sources

jDTO Binder supports merging values for multiple source beans. In order to use this feature the framework provides the `@SourceNames` annotation type. The following example illustrates the basic usage of this feature:

```
1  @SourceNames({"bean1", "bean2", "bean3"})
2  public class MultiSourceDTO {
3
4      @Source(value="aString") // default bean1
5      private String source1;
6      @Source(value="aString", sourceBean="bean2")
7      private String source2;
8      @Source(value="aString", sourceBean="bean3")
9      private String source3;
10
11     ... // GETTERS AND SETTERS
12 }
```

The `@SourceNames` annotation can be used either on class level or in property level. When used at the class level acts as the default setting for all source fields. When used at the property level it overrides the settings for the class.

All of the methods on the binder instance are varargs and the parameters order must match the bean names order for the framework to read the source properties the right way.

Multi source properties also support multi bean sources and all the features it implies, the following example illustrates the power of multi source, multi bean DTO binding:

```
1  @SourceNames({"bean1", "bean2"})
2  public class MultiSourceDTO2 {
3      @Source("aString") //using bean1 as default
4      private String string1;
```

```
5
6       @Sources(value={@Source("anInt"),
7               @Source(value = "theDate",
8               sourceBean="bean2",
9               merger=DateFormatMerger.class,
10              mergerParam="dd/MM/yyyy")},
11          merger=StringFormatMerger.class, mergerParam="%02d %s")
12      private String string2;
13
14      @Source(value = "theCalendar", sourceBean="bean2",
15          merger=DateFormatMerger.class, mergerParam="dd/MM/yyyy")
16      private String string3;
17
18      ... //GETTERS AND SETTERS
19  }
```

9 Immutable DTOs

jDTO Binder is capable of building instances of the DTOs using non default constructors, this brings you the possibility of creating immutable instances which are objects whose state doesn't change.

In order to use this feature, your class must not have a default constructor and may have more than one constructors. In order to choose which constructor you want the framework to use, you must annotate it with the `@DTOConstructor` annotation[2] or define it into the XML settings.

The following is an example of a typical immutable DTO:

```
1   public final class SimpleImmutableDTO {
2       private final String firstString;
3       private final String secondString;
4
5       //make this the DTO constructor.
6       @DTOConstructor
7       public SimpleImmutableDTO(@Source("myString") String firstString,
            @Source("related.aString") String secondString) {
8           this.firstString = firstString;
9           this.secondString = secondString;
10      }
11
12      public SimpleImmutableDTO(String firstString, String secondString
            , String thirdString) {
13          this.firstString = firstString;
14          this.secondString = secondString;
15      }
```

[2]Not required when there's just one constructor.

```
16
17      public String getFirstString() {
18          return firstString;
19      }
20
21      public String getSecondString() {
22          return secondString;
23      }
24  }
```

Since there is no reliable way of getting the parameter names using the Java Reflection API (and therefore no safe way of creating a default configuration), the user must specify settings for each constructor argument. Failing to provide configuration for these arguments will cause a `RuntimeException` to be thrown.

You can't configure a constructor argument to be transient since the class is immutable there won't be a chance to change it later.

Some of these behaviors may change on the future.

10 Reverse Binding: DTO To Business Object

jDTO Binder is capable of reading DTO data and extract a business object of a given type by using the source fields (in the mapping) as target fields for the business object. Even though no extra configuration is required, this process is not as powerful as the original conversion. There are some hidden tricks here, for example, suppose your original business object had four integer fields that where added by some custom field merger. On the reverse process, how would the merger know how to unmerge these values?.

By design, jDTO Binder sacrificed this capability of going back and forth 100% for the flexibility of populating DTOs in a complex way. If you would like a more robust reverse conversion, you could add binding annotations to the business object and treat is as if it was a DTO helping the merge process by writing your own custom reverse-mergers.

The following snippet demonstrates how to apply reverse binding to extract a business object out of a DTO.

```
1  //create a basic entity
2  SimpleEntity entity = new SimpleEntity("test", 123, 345.35, true);
3
4  //try and build a DTO out of the same entity.
5  SimpleEntity dto =
6      binder.bindFromBusinessObject(SimpleEntity.class, entity);
7
8  //change things on the dto
9  dto.setAnInt(10);
10 dto.setaBoolean(false);
11 dto.setaDouble(20.20);
```

```
12 dto.setaString("Changed!");
13
14 entity = binder.extractFromDto(SimpleEntity.class, dto);
15
16
17 assertEquals(10, entity.getAnInt());
18 assertEquals(false, entity.isaBoolean());
19 assertEquals(20.20, entity.getaDouble(), 0.0001);
20 assertEquals("Changed!", entity.getaString());
```

If the entity class argument is an implementation of `java.util.Map`, then the value returned will be a map, and the keys of the map will match the DTO mapping for the object passed as an argument. This means you can populate a map out of an object.

11 XML Configuration

jDTO Binder by default binds the DTOs using annotations but this is not the only option, there are some cases in which annotations are not convenient or even not available. For this cases, jDTO Binder provides a way to configure the DTO binding on a convenient XML file.

As this framework is built with the convention over configuration philosophy, you can start working with practically no configuration and customize just some things. By default, if there is a file on the default package called /jdto-mappings.xml, then the framework will use it and disable the annotations config (which is actually the default when the file is not present). All the properties of a DTO are taken in account unless you explicitly declare them as transient. All DTOs declared on the XML file are loaded and analyzed eagerly, and non-configured DTOs are analyzed lazily the first time they're used.

Finally, the XML configuration file currently supports all the features the Annotation configuration support. There may be some additions on the future to make simpler the configuration.

Rather than explaining again the whole feature set, some examples will be shown and with the hope they're clear enough. Nevertheless a XML schema is available and most popular IDEs allow auto completion out of the schema.

11.1 Basic XML Setup

It is recommended to create an xml file on the default package called /jdto-mappings.xml, and in its empty form should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<dto-mapping
    xmlns="http://jdto.org/jdto/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jdto.org/jdto/1.0
```

```
                        http://jdto.org/jdto-1.0.xsd">

</dto-mapping>
```

11.2 Simple XML Binding

The following snippet demonstrates a DTO mapped in a really simple fashion:

```
<!-- to test simple binding -->
<dto type="org.jdto.dtos.XMLTesterDTO">

    <field name="aString" transient="true" />

    <!-- A field bound to another field. -->
    <field name="dtoName">
        <source name="aString" />
    </field>
</dto>
```

11.3 Navigating Associations

The following snippet demonstrates a mapping which goes through property paths:

```
<!-- test the association feature -->
<dto type="org.jdto.dtos.SimpleAssociationDTO">
    <field name="firstString">
        <source name="myString" />
    </field>
    <field name="secondString">
        <source name="related.aString" />
    </field>
</dto>
```

11.4 Cascaded Mapping

The following snippet demonstrates how the cascade logic can be mapped:

```
<!-- to test cascade logic -->
<dto type="org.jdto.dtos.ComplexDTO">
    <field name="cascadedField" cascade="true">
        <source name="association" />
    </field>
    <field name="stringField">
        <source name="name" />
```

```
    </field>
</dto>
```

11.5 Property Mergers

The following snippet demonstrates how various property mergers can be configured:

```
<!-- test the compound merger feature -->
<dto type="org.jdto.dtos.FormatDTO">
    <!-- single field merger test -->
    <field  name="price">
        <source name="aDouble"
        merger="org.jdto.mergers.StringFormatMerger"
        mergerParam="$ %.2f" />
    </field>
    <!-- multiple field merger test -->
    <field name="compound" mergerParam="%.2f %08d"
    merger="org.jdto.mergers.StringFormatMerger">
        <source name="aDouble" />
        <source name="anInt" />
    </field>
</dto>
```

In order to support multiple merger parameters without adding an excessive load of XML configuration, it's a design decision to separate those parameters in the same XML "mergerParam" attribute with semicolons ";", so for example if the value merger takes two parameters "first" and "second", the XML snippet would look like:
`mergerParam="first;second"`.

11.6 Multiple Source Beans

The following snippet demonstrates how you can configure mappings with multiple bean sources:

```
<!-- test the multi source feature -->
<dto type="org.jdto.dtos.MultiSourceDTO">
    <sourceNames>
        <beanName>bean1</beanName>
        <beanName>bean2</beanName>
        <beanName>bean3</beanName>
    </sourceNames>
    <field name="source1">
        <source name="aString" bean="bean1" />
    </field>
    <field name="source2">
        <source name="aString" bean="bean2" />
```

```
    </field>
    <field name="source3">
        <source name="aString" bean="bean3" />
    </field>
</dto>
```

11.7 Immutable DTOs

The following snippet demonstrates how you can configure a constructor to be used by the jDTO Binder framework:

```
<dto type="org.jdto.dtos.SimpleImmutableDTO">
    <immutableConstructor>
        <arg order="0" type="java.lang.String">
            <source name="test" bean="bean1" />
        </arg>
        <arg order="1" type="java.lang.Number">
            <source name="pepe" />
        </arg>
    </immutableConstructor>
</dto>
```

The order attribute is optional, if not present, the declaration order will be taken into account. The "arg" XML element is very similar to the "field" element, but is different in the way that it doesn't have a name but defined by an order and a type and it can't be transient.

All of the constructor arguments must be declared and also they must have at least one source property configured. This is mainly because there's no reliable way to read the argument names of one method in the java reflection API so it is impossible to assume a default configuration.

12 Extending the Framework

There are some ways to extend or customize the framework, the main way is to write custom property value mergers which will let you customize how the values are copied from the original object to the DTO. Another way of customizing the framework is by implementing a custom bean modifier (which will not be covered).

12.1 Implementing Custom Property Value Mergers

jDTO Binder has two main types of property value mergers: Those who merge values from a single source and those who merge values from multiple sources. Both kinds of mergers are applied on the binding process but in different stages, for those fields annotated with `@Source` (or its XML equivalent), just the single property value merger

is applied; for those fields annotated with `@Sources` (or its XML equivalent), one single property value merger is applied for each source, and then a multiple property value merger is applied to the results of the previous.

12.2 Single Property Value Mergers

In order to implement a single property value merger, you need to create a class that implements the interface `SinglePropertyValueMerger`.

The interface `SinglePropertyValueMerger` looks like the following, the generic type variables are added for developer convenience:

```
1
2  /**
3   * Merge a property into another type / form by applying a
        transformation. <br />
4   * Transformations can be hinted by the extra param attribute.
5   * @param R the type of the resulting property.
6   * @param S the type of the source property, for developer
        convenience.
7   * @author Juan Alberto Lopez Cavallotti
8   */
9  public interface SinglePropertyValueMerger<R, S> {
10
11     /**
12      * Merge the value of type S into another object of type R.
13      * @param values the value to be merged.
14      * @param extraParam metadata that may help the merger to build
            the result.
15      * @return the merged object.
16      */
17     public R mergeObjects(S value, String[] extraParam);
18 }
```

Each implementation of a property merger is kept as a singleton, therefore is not safe to use instance variables for user functionally.

12.2.1 Utility Merger Templates

The framework provides some utility template classes to implement property value mergers, some of these are:

- `AbstractCalulationCollectionMerger`: provides functionality to perform calculations on a given property of a collection or array.

12.3 Multiple Property Value Mergers

In order to implement a multiple property value merger, you need to create a class that implements the interface `MultiPropertyValueMerger`.

The interface `MultiPropertyValueMerger` looks like the following, the generic type variables are added for developer convenience:

```
1  /**
2   * Implementations should know how to merge a list of objects into a
        single object. <br />
3   * This interface is meant to be used to create a single value out of
        a multi-source
4   * property configuration, see {@link com.juancavallotti.jdto.
        annotation.Sources}.
5   * @param <R> The result type of the merged parameters.
6   * @author Juan Alberto Lopez Cavallotti
7   */
8  public interface MultiPropertyValueMerger<R> {
9
10     /**
11      * Merge the list of objects into a single object.
12      * @param values the values to be merged.
13      * @param extraParam metadata that may help the merger to build
            the result.
14      * @return the merge resulting object.
15      */
16     public R mergeObjects(List<Object> values, String[] extraParam);
17 }
```

Same considerations for single property value mergers should be taken.

12.4 Accessing the Bean Modifier

In some cases, the developer must read safely properties from the source objects (the same as the framework does to create the DTOs), for this purpose, you need to instruct the framework that you wish to work with a `BeanModifier` instance, to do this just implement the `BeanModifierAware` interface so the framework injects (by setter injection) the bean modifier to your property value merger before it calls the `mergeObjects` method, the interface looks like the following:

```
1  /**
2   * Makes this object aware of the current bean modifier used by the
        DTO binder
3   * instance. Injection is performed by setter dependency injection.
4   * @author Juan Alberto Lopez Cavallotti
5   */
6  public interface BeanModifierAware {
```

```
7
8       /**
9        * Expose the BeanModifer to the implementing class.
10       * @param modifier the bean modifier instance.
11       */
12      void setBeanModifier(BeanModifier modifier);
13  }
```

In this case it is safe to save the modifier instance in an instance variable of the property value merger.

12.5 Math Expressions API

jDTO Binder ships with a simple-to-use mathematic expression API you can take advantage, the variable resolution process can be either event-driven (lazy loaded) or the developer can provide a map matching each variable with its value. Please see javadoc for more information.