jDTO Binder 1.4 User's Guide

Juan Alberto López Cavallotti

January 6, 2013

http://www.jdto.org
jdto@jdto.org

Contents

1 About jDTO Binder

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.

DTOs (Data Transfer Objects) are part of the API of a service oriented architecture. When this API is loosely coupled, the services can be exposed in various ways including RMI, SOAP and RESTful; and still be used directly. This loose coupling is needed when the application needs to scale up and work over the cloud.

The main reason why developers have to not used DTOs at the initial phases of the development of an application is that the use of DTOs takes valuable time, specially adds extra lines of code to make the copy process from and to domain objects. jDTO Binder takes care of this concern by letting small applications be scalable from the very beginning.

This approach also enables developers to use another declarative tools like JSR-303 (bean declarative validation), and hooks perfectly into model 2 web frameworks.

jDTO Binder has illustrated these concepts in very simple sample projects which are available as open source at https://github.com/jDTOBinder/jdto-sample-projects. These sample projects serve both as a starting point and as proof of concept so they're definitely worth looking at.

1.1 What's new in 1.4

This release includes a great number of new features and improvements.

jDTO Binder 1.4 introduces new features:

- Added support for DTO binding lifecycle with two specific lifecycle hooks, `beforePropertiesSet` and `afterPropertiesSet`. Issue #27.

- Property value mergers are looked into the spring context and then the internal instance pool as a fallback mechanism.

- Added convenience API to DTOBinderFactory to load XML configuration from the classpath.

- Added null to 0 conversion when dealing with primitives for compatibility logic.

Issues solved:

- Fixed `NullPointerException` when loading empty XML configuration. Issue #25

- Fixed `NullPointerException` on cascade when the source value is null. Issue #26.

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/jDTOBinder/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/home/view/contribute.

2 Quick Start

2.1 Getting The Framework

You may download the framework's JAR files and add them to your project's classpath by using our download page. Also jDTO Binder is deployed on the central maven repository.

2.1.1 Downloading the JARs

JAR files with the binaries, source and javadoc are available for download at the jDTO Binder home page http://www.jdto.org/home/view/download you may also want to download separately the jars for the commons-lang and slf4j projects:

- Commons Lang (min version 2.4): http://commons.apache.org/lang/

- Simple Logging Facade for Java (min version 1.6): http://www.slf4j.org/

2.1.2 Maven Dependencies

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:

```xml
<dependencies>
...
    <dependency>
        <groupId>org.jdto</groupId>
        <artifactId>jdto</artifactId>
        <version>1.4</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>2.4</version>
    </dependency>
...
<dependencies>
```

Beginning from jDTO Binder 1.1, maven artifacts are deployed on the central maven repository so you don't need to add any extra repositories as before.

2.2 Bootstrapping the Binder

In order to bind DTOs, a binder instance must be bootstrapped. jDTO Binder can be used as a standalone library but also it is integrated with the Spring Framework[1] and CDI[2] (JSR-330 Contexts and Dependency Injection).

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

2.3 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());
```

---

[1]In order to use the spring framework binder, you should add the spring framework dependency.
[2]In order to use the CDI integration, you should add the jdto-cdi artifact to your dependencies.

The relevant entity and DTO declarations are:

```java
public class MixedEntity {
    private String someString;
    private int someInt;
    private double someDouble;
    private Date someDate;

    ... //getters and setters
}

public class BasicDTO {
    private String someString;

    @Source("someInt")
    private int personAge;

    ... //getters and setters
}
```

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;
```

3.1.1 Property Value Mergers Lookup

Starting from version 1.4, jDTO Binder looks up on the spring registry for property value mergers before looking for them on the internal instance pool, this allows the user to access the Spring Context and features within these mergers and so, making easy to implement functionality otherwise difficult to achieve.

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>
```

9

```
                <groupId>org.jdto</groupId>
                <artifactId>jdto−cdi</artifactId>
                <version>1.3</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.

3.3 Scripting Language Integration

jDTO Binder ships with a scripting module. You may use scripting languages such as Groovy to create simple but powerful field bindings, the binding logic can be configured directly as a merger param, and many things can be achieved with just one line of code.

In order to use the scripting module, you need to add the following dependency to your pom.xml file:

```
        <dependency>
                <groupId>org.jdto</groupId>
                <artifactId>jdto−scripting</artifactId>
                <version>1.3</version>
        </dependency>
```

In the following example is shown how to use the groovy mergers to perform complex bindings:

```
1
2  @SourceNames({"bean1", "bean2"})
3  public class GroovyDTO implements Serializable {
4      private static final long serialVersionUID = 1L;
5
6      @Source(value="myString", merger=GroovyMerger.class,
7              mergerParam="sourceValue == null ? 'is null' : 'is not
                  null'")
8      private String singleSource;
9
10     @Sources(value = {@Source("myString"), @Source(value = "myString"
           , sourceBean="bean2")},
11             merger = MultiGroovyMerger.class,
12             mergerParam = "sourceValues[0] + ' and ' + sourceValues
                  [1]")
13     private String multipleSource;
```

```
14
15      ... // getters and setters
16
17  }
```

Some special features have been enabled on the Groovy runtime for convenience:

- A variable called `logger` has been introduced, this variable may be used for logging values.

- The packages java.math.*, java.util.* and org.apache.commons.lang.* have been automatically imported.

For more information and examples, please read the corresponding Javadoc API.

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: An array of strings 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.

As part of jDTO Binder 1.3, a special keyword has been introduced to reference the root object instead of one of its properties, `+rootObject`, this keyword is also represented as an API constant, the following example demonstrates the usage:

```
1
2   public class WrapperDTO<T> implements Serializable {
3
4       private static final long serialVersionUID = 1L;
5
6       /**
7        * Get the root object.
8        */
9       @Source(Source.ROOT_OBJECT)
10      private T wrapped;
11
12      ... //Getters and setters
13
14  }
```

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. This merger can be used to restore the values it converts.

- `DecimalFormatMerger`: Format any number by applying a format String. This merger can be used to restore the values it converts.

- `EnumMerger`: Convert an enum literal to it's String representation.

- `GroovyMerger`: Merge the objects by evaluating a Groovy expression. (Requires jdto-scripting module).

- `IdentityPropertyValueMerger`: Default merger, returns the same instance of the value. This merger can be used to restore the values it converts.

- `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).

- `MethodCallMerger`: Call a no-arg non-void method and use the result as the merged value. This merger can be used to restore the values it converts in two ways, refer to javadoc for more information.

- `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".

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

4.2 Using the Mergers from outside the Framework

There are some cases in which the developer may need some functionality that is provided by some merger. Building merger instances have some steps and it is likely to become quite complex on the future. For this matter, jDTO Binder provides one method on the `DTOBinder` instance:

```
1
2  DTOBinder binder = DTOBinderFactory.buildBinder();
3  IdentityPropertyValueMerger merger = binder.getPropertyValueMerger(
       IdentityPropertyValueMerger.class);
```

The user may customize also how the mergers are located, cached and dependency-injected by implementing the interface `PropertyValueMergerInstanceManager` and configuring the DTOBinder object to use it.

4.3 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 Collections of Business Objects

jDTO Binder is capable of binding whole collections of business objects to collections of DTOs. To do this, the binder has two utility methods to bind business objects:

- `bindFromBusinessObjectCollection`: This method returns a collection of the same type as the one of the argument but filled with the DTOs. If the source collection cannot be instantiated (for example is immutable), then depending on the case `ArrayList` or `HashSet` will be returned.

- `bindFromBusinessObjectList`: This method is intended now for multi-source DTO Bindings, it has a var-args argument for lists of objects that need to be related by index to build the new DTO.

The first method will not modify the order of the objects if the used collection used doesn't. The second method will always preserve the order because of the nature of the list but does not ensure the returned list is of the same type of the source. In both cases the original collection is not altered.

The following example shows how a list of business objects is converted into a List of DTOs.

```
1  LinkedList<SimpleEntity> simpleEntities =
2      new LinkedList<SimpleEntity>();
3  simpleEntities.add(new SimpleEntity("simple 1", 12, 45.56, true));
4  simpleEntities.add(new SimpleEntity("simple 2", 34, 56.67, false));
5
6  List<FormatDTO> dtos = binder.bindFromBusinessObjectList(FormatDTO.
       class, simpleEntities);
```

6 Cascade Binding

6.1 Cascade Basics

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  }
```

6.2 Circular References and Recursion

Cascade configuration is in some sense a recursive process but there may be situations where it is convenient to have cycles in your DTO objects graph (to represent bidirectional relationships is one example).

You need to keep in mind that you may configure mutable and immutable DTOs with cascading.

The Way jDTO Binder stops the recursion is by caching the DTO objects it instantiates before resolving the values that they will carry, in this sense memory gets optimized because one DTO instance will be generated per domain object and also creates a barrier for cascade recursion.

Since is not the aim jDTO Binder to use dynamic proxy objects (or cglib enhancers or similar), immutable DTOs cannot be cached in advance and therefore infinite recursion may happen when combining these two features.

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.

- `MultiGroovyMerger`: Merge the objects by evaluating a Groovy expression. (Requires jdto-scripting module).

- `StringFormatMerger` as described before, this merger uses the String.format method to merge all the provided values into a single formatted string.

- `TimeBetweenDatesMerger`: This merger merges two dates or calendars by calculating the time between them, by default the time unit is hours but you may specify seconds, minutes, hours, days or weeks.

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[3] or define it into the XML settings.

The following is an example of a typical immutable DTO:

```java
public final class SimpleImmutableDTO {
    private final String firstString;
    private final String secondString;

    //make this the DTO constructor.
    @DTOConstructor
    public SimpleImmutableDTO(@Source("myString") String firstString,
            @Source("related.aString") String secondString) {
        this.firstString = firstString;
        this.secondString = secondString;
    }

    public SimpleImmutableDTO(String firstString, String secondString
            , String thirdString) {
        this.firstString = firstString;
        this.secondString = secondString;
    }

    public String getFirstString() {
        return firstString;
    }

    public String getSecondString() {
        return secondString;
    }
}
```

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.

---

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

10   Reverse Binding: DTO To Business Object

10.1   Reverse Binding Basics

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 restore 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   AnnotatedEntity entity = new AnnotatedEntity("Jones", "Tom", "NO");
3
4   //try and build a DTO out of the same entity.
5   AnnotatedEntity dto = binder.bindFromBusinessObject(AnnotatedEntity.
        class, entity);
6
7   //change things on the dto
8   dto.setFirstString("Myers");
9   dto.setSecondString("Mike");
10
11  entity = binder.extractFromDto(AnnotatedEntity.class, dto);
```

10.2   Value Restauration

In some cases, a merger may be able to restore a converted value to its original state, sometimes this can be achieved in a generic way and other times is domain-specific.

`SinglePropertyValueMerger` have two methods that the framework uses to check if value restoration is possible or not:

- `boolean isRestoreSupported(String[] params)` checks if with the given information the value can be restored.

- `public S restoreObject(R object, String[] params)` restores the value if the previous check returns `true`.

Some mergers have already built-in support for restoring the values, for more information refer to the javadoc of each merger.

If you want to perform domain-specific value restoration, you are encouraged to subclass the merger and override the previously mentioned methods.

```
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 DTO Binding Lifecycle

Starting from version 1.4, jDTO Binder has incorporated the concept of "Binding Life-cycle" and it provides a set of notifications which might be used by the developer to customize the way that DTO's are populated or to trigger custom code on the process. The current lifecycle has the following stages:

- Before Properties Set: This phase is activated after the DTO instance is created and the framework will call by convention a method with name `beforePropertiesSet` which takes as argument a `org.jdto.DTOBindingContext` instance, this method, if present, is not called on immutable DTOs.

- After Properties Set: This phase is activated after the DTO instance has been populated and the framework will call by convention a method with name `afterPropertiesSet` which takes as argument a `org.jdto.DTOBindingContext` instance, this method, if present, is called regardless the type of DTO.

These conventional methods must be `public` or otherwise the framework will ignore them.

Through the `DTOBindingContext` instance, the developer has access to the `DTOBinder` involved on the process as well as the resulting metadata from the initial bean inspection process and the source values used to populate the DTO.

This kind of lifecycle handlers are only applied when populating DTOs from business objects and not the inverse, this is mainly because we want to keep domain objects as clean as possible.

The following is an example of how the developer might take advantage of this lifecycle:

```java
1  import java.io.Serializable;
2  import org.jdto.DTOBindingContext;
3
4  public class MyDTO implements Serializable {
5
6      //... normal DTO implementation
7
8      public void afterPropertiesSet(DTOBindingContext context) {
9          //TODO - Add your own logic.
10     }
11 }
```

It is a design decision not to add conventions without parameters which might affect code readability and also could be confused with lifecycle calls of other frameworks such as the Spring Framework.

12 Value Compatibilization

When try to set a value on a target DTO, jDTO Binder will try to make that value compatible so there is no need for extra configuration.

The following list describes the changes jDTO Binder does to the values so they're compatible:

- If the target type is a `String`, then call `toString` on the source object before attempting to set it.

- If the target type is `Date` or `Calendar`, then try to convert between one another (normally by calling the method `getTime()` and `setTime()` on a calendar instance).

- If the target type is `Enum` and the source type is `String`, then try to resolve the enum constant from the given string.

- If the target type is primitive and the source value is `null`, then it is converted to `false` or 0 depending on the type.

More rules may be added in the future for developer convenience.

13 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.

13.1 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
<?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>
```

13.2 Simple XML Binding

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

```xml
<!-- 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>
```

13.3 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>
```

13.4 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>
```

13.5 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".`

13.6   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>
```

13.7   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.

14 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).

14.1 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.

14.2 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.

14.2.1 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.

14.3 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:

```
/**
 * Implementations should know how to merge a list of objects into a
     single object. <br />
 * This interface is meant to be used to create a single value out of
     a multi-source
 * property configuration, see {@link org.jdto.annotation.Sources}.
 * @param <R> The result type of the merged parameters.
 * @author Juan Alberto Lopez Cavallotti
 */
public interface MultiPropertyValueMerger<R> {

    /**
     * Merge the list of objects into a single object.
     * @param values the values to be merged.
     * @param extraParam metadata that may help the merger to build
         the result.
     * @return the merge resulting object.
     */
    public R mergeObjects(List<Object> values, String[] extraParam);
}
```

Same considerations for single property value mergers should be taken.

14.4 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.

14.5 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.