Trugger User Guide

Marcelo Guimarães

Contents

1 Overview

Trugger is a framework that helps you write code that anyone can read. The idea is to provide a set of fluent interfaces to deal with operations related to reflection (such as creating proxies, reflecting members and scanning classes).

Trugger is intended to be a base for creating infrastructure code. While it is not an IoC container, for example, it could be a part of the core of an IoC container.

1.1 What a hell does the name trugger mean?

When I was learning java, I choosed "Atatec" (*Ataxexe Technology*) to be the name of my fictitious company and then I sent a message to my friend Ives:

> Atatec

And the response was:

> trugger

When I asked him why he came up with this, he said:

> Isn't that the name of that special kick in Street Fighter?

The special kick mentioned is the *Tatsumaki Senpuu Kyaku* and Ken says something like "atatec trugger" (at least in Portuguese) when he does the kick. So the name Trugger became my first choice when I start to write this framework.

1.2 How To Use

Just put the jar file on your **classpath** and you're done. No dependencies are required by Trugger at runtime.

1.3 How To Build

Just make sure you have **Gradle**[1] and execute the `gradle dist` command. The markdown files are converted at build time using **pandoc**[2] so you should have it to build the distribution files. You can also use the command `gradle jar` to build only the binaries.

---

[1] http://gradle.org

[2] http://johnmacfarlane.net/pandoc/README.html

1.4 How To Contribute

Just fork the project, do some stuff and send me a pull request. You can also fire an issue or tell your friends to use Trugger.

1.5 About Maven Repository

Trugger currently is not in the Maven Central Repository, but a pom is generated and distributed in the download so you can import it on your local repository.

2 Reflection

The Reflection module was the first one to show up in Trugger, it allows simple and much less verbose use of the Reflection API to reflect constructors, methods, fields and generic type declarations.

The base class of this module is `Reflection` and you will find a nice fluent interface there with the methods `reflect`, `invoke` and `handle`.

2.1 Fields

2.1.1 Single

Reflecting a field is done with the code `reflect().field(field name)`. You can also apply a filter by using the method `filter` and the target class (or object) by using the method `in`. If you need a recursive search through the entire target class hierarchy, just use the method `deep`.

Here is some examples (assuming a `static import`):

```
Field value = reflect().field("value").in(String.class);

Field name = reflect().field("name").deep().in(MyClass.class);

Field id = reflect().field("id")
    .filter(field -> field.getType().equals(Long.class))
    .in(someInstance);
```

The `filter` method receive a `java.util.function.Predicate` to stay in touch with Java 8.

2.1.2 Multiple

If you need to reflect a set of fields, use the `reflect().fields()`. The same features of a single reflect is available here.

```
List<Field> stringFields = reflect().fields()
    .filter(field -> field.getType().equals(String.class))
    .deep()
    .in(MyClass.class);
```

2.1.3 Predicates

There are some builtin predicates in the class `FieldPredicates`.

```
List<Field> stringFields = reflect().fields()
    .filter(type(String.class)) // a static import
    .deep()
    .in(MyClass.class);
```

2.1.4 Handling Values

A field can have its value manipulated through the method `Reflection#handle`. It will create a handler to set and get the field's value without the verbosity of the Reflection API. To handle static fields, you can call the handler methods directly:

```
String value = handle(field).value();
handle(field).set("new value");
```

For instance fields, just specify an instance using the method `in`:

```
String value = handle(field).in(instance).value();
handle(field).in(instance).set("new value");
```

You can also use a a `FieldSelector`:

```
// static import Reflection#field
String value = handle(field("name")).in(instance).value();
```

2.2 Constructors

2.2.1 Single

To reflect a constructor, use `reflect().constructor()` and specify the parameter types and optionally a filter. If the class has only one constructor, it can be reflected without supplying the parameter types.

```
Constructor constructor = reflect()
  .constructor().withParameters(String.class).in(MyClass.class);

Constructor constructor = reflect().constructor()
  .withoutParameters().in(MyClass.class);

Constructor constructor = reflect().constructor()
  .filter(c -> c.isAnnotationPresent(SomeAnnotation.class))
  .withParameters(String.class).in(MyClass.class);

Constructor constructor = reflect().constructor().in(MyClass.class)
```

2.2.2 Multiple

A set of constructors can be reflected by using `reflect().constructors()`. As in fields, the features in single reflection are present in multiple reflection.

```
List<Constructor> constructors = reflect().constructors().in(MyClass.class);

List<Constructor> constructors = reflect().constructors()
  .filter(c -> c.getParameterCount() == 2)
  .in(MyClass.class);
```

Note that in multiple selection you cannot specify the parameter types directly in the fluent interface (for obvious reasons).

2.2.3 Predicates

A few useful predicates are included in the class `ConstructorPredicates`.

```
List<Constructor> constructors = reflect().constructors()
  .filter(annotated()) // a static import
  .in(MyClass.class);
```

2.2.4 Invocation

To invoke a constructor you need an `Invoker`. The method `Reflection#invoke` returns a Invoker for a constructor. Specify the parameters and the constructor will be invoked.

```
Constructor c = reflect().constructor()
  .withParameters(String.class).in(String.class)
String name = invoke(c).withArgs("Trugger");
```

2.3 Methods

2.3.1 Single

To reflect a single method, just pass it name to `Reflection#method`. Filtering is allowed and you can specify parameter types too.

```
Method toString = reflect().method("toString").in(Object.class);

Method remove = reflect().method("remove")
  .withParameters(Object.class, Object.class)
  .in(Map.class);

Method someMethod = reflect().method("foo")
  .filter(method -> method.isAnnotationPresent(PostConstruct.class))
  .in(instance);
```

As in field reflection, you can do a deep search with `deep`.

```
Method toString = reflect().method("toString").deep().in(MyClass.class);
```

2.3.2 Multiple

A set of methods can be reflected by using `Reflection#methods`:

```
List<Method> methods = reflect().methods().in(Object.class);
```

Deep search and filtering are also supported:

```
List<Method> methods = reflect().methods()
  .filter(method -> method.isAnnotationPresent(PostConstruct.class)
  .deep()
  .in(MyClass.class);
```

2.3.3 Predicates

A set of predicates to deal with methods is in `MethodPredicates`:

```
List<Method> methods = reflect().methods()
  .filter(annotatedWith(PostConstruct.class)) // a static import
  .deep()
  .in(MyClass.class);
```

2.3.4 Invocation

To invoke a method, use the Invoker returned by `Reflection#invoke`. Instance methods needs an instance provided using the method `in`:

```
Method toString = reflect().method("toString").in(String.class);
invoke(toString).in("A string").withoutArgs();
```

Static methods don't need it:

```
Method parseInt = reflect().method("parseInt")
  .withParameters(String.class)
  .in(Integer.class);
int number = invoke(parseInt).withArgs("10");
```

Note that you can also use a `MethodSelector`:

```
invoke(method("toString")).in("A string").withoutArgs();
```

2.4 Generic Type

Generic declarations in a class are present in the bytecode. Trugger can reflect them by using the method `genericType`. Suppose we have this interface:

```
public interface Repository<T> {
  // ... some useful methods here
}
```

A generic base implementation can use that **T**:

```
public class BaseRepository<T> {

  private final Class<T> type;

  protected BaseRepository() {
    this.type = reflect().genericType("T").in(this);
  }
}
```

The constructor was declared `protected` to warn that this will only work for subclasses (it is a Java limitation). A workaround to use this trick in a variable-like way is by declaring an anonymous class:

```
Repository<MyType> repo = new BaseRepository<MyType>(){};
```

I think this is an ugly solution, but works.

3 Class Scanning

Another cool feature Trugger has is the class scanning. Just give a package name and Trugger will scan it for finding classes. The class scanning feature starts at `ClassScan`.

The scanning starts in the method `ClassScan#scan`, which returns a `ClassScanner` that allows changing the ClassLoader and defining the package.

```
List<Class> classes = scan().classes().in("my.package");
```

This will return every class in the package `my.package`. To do a deep scan and also return the classes in subpackages, use the `deep` method:

```
List<Class> classes = scan().classes().deep().in("my.package");
```

You can also filter the classes with the method `filter`:

```
List<Class> classes = scan().classes()
  .deep()
  .filter(c -> c.isAnnotatedWith(Entity.class))
  .in("my.package");
```

3.1 Predicates

In `ClassPredicates` is a set of useful predicates to deal with classes.

```
List<Class> classes = scan().classes()
  .deep()
  .filter(annotatedWith(Entity.class)) //static import
  .in("my.package");

List<Class> classes = scan().classes()
  .deep()
  .filter(subtypeOf(Repository.class)) //static import
  .in("my.package");
```

3.2 Resource Finders for Protocols

The search is performed based on protocols. The basic protocols are `file` and `jar` and are supported by Trugger. More specific protocols can be handled by creating a `ResourceFinder` and registering it with the method `ClassScan#register`. A `ResourceFinder` is responsible for finding any resources in a given package and it must support a protocol. Trugger has a couple of finders registered to the following protocols:

- **jar** - for resources in a jar file
- **file** - for resources in the filesystem
- **vfs** - for resources in jar files deployed on a JBoss AS 7.x
- **vfszip** - for resources in jar files deployed on a JBoss AS 5.x and 6.x
- **vfsfile** - for resources in files deployed on a JBoss AS 5.x and 6.x

Any finder registered to an already supported protocol will override the registered finder.

4 Proxy Creation

A proxy object is created using a interception handler and optionally a fail handler. The DSL exposed starts at `Interception` with the method `intercept` and lets you define one or more interfaces to intercept. Additionally, you can set a target and its interfaces will be used. After the behaviour specification, use the method `proxy` to create the proxy instance.

```
SomeInterface proxy = Interception.intercept(SomeInterface.class)
  .onCall(context -> logger.info("method intercepted: "
    + context.method())
  .proxy();

proxy.doSomething();
```

The interception logic happens in the handler passed through the method `onCall`. The handler receives a context, which contains all information about the intercepted method. A fail handler can also be set using the method `onFail`.

```
SomeInterface proxy = Interception.intercept(SomeInterface.class)
  // sets a target to delegate the call using the context object
  .on(instance)
  // delegates the call to the target (this is the default behaviour)
  .onCall(context -> context.invoke())
  // handles any error occurred
  .onFail((context, throwable) -> handleTheFail(throwable))
  .proxy();

proxy.doSomething();
```

The fail handler has access to the context so you can delegate the method to the target again (if a timeout occurs, for example).

4.1 The Interception Context

The interception context holds everything related to the method interception, included:

- The arguments passed
- The method intercepted
- The declared method intercepted in the target instance
- The proxy instance
- The target instance (may be null if not specified when creating the proxy)

The context can be used to delegate the method call to the target (using `invoke`) or to another instance (using `invokeOn`). The declared method intercepted can be retrieve by using `targetMethod`.

5 Elements of an Object

5.1 What is an Element?

An element is any value that an object holds. It may be accessible through a field, invoking a method (a getter or a setter) or even a specific way like the `Map#get` method.

A basic element in Trugger is a Property or a Field. Trugger tries to find a getter and a setter method for the element name and a field with the same name. This allows manipulate private fields and properties in the same way without bothering you with the way of handling the value.

5.2 Obtaining an Element

An element is obtained using the method `element` in `Elements`. The same features of a field reflection is here with the addition of getting an element without specifying a name. A set of predicates are present in `ElementPredicates`.

```
Element value = element("value").in(MyClass.class);

Element id = element()
  .filter(annotatedWith(Id.class)) // static import
  .in(MyClass.class);

List<Element> strings = elements()
  .filter(type(String.class) // static import
  .in(MyClass.class);
```

5.3 Copying Elements

The elements of an object can be copied to another object, even if they are from different types. The DSL starts at the method `copy`:

```
copy().from(object).to(anotherObject);
```

This will copy every element. To restrict the copy to non null values, use the `notNull` method:

```
copy().from(object).notNull().to(anotherObject);
```

You can also apply a function to transform the values before assigning them to the target object (useful when copying values to a different type of object).

```
copy().from(object)
  .applying(copy -> copy.value().toString())
  .to(anotherObject);
```

To filter the elements to copy, just give a selector to the `copy` method:

```
copy(elements().filter(annotatedWith(MyAnnotation.class)))
  .from(object)
  .to(anotherObject);
```

Or filter the copy directly using `filter`

```
copy(elements().filter(annotatedWith(MyAnnotation.class)))
  .from(object)
  .filter(copy -> copy.dest().isAnnotationPresent(MyAnnotation.class))
  .to(anotherObject);
```

5.4 Nested Elements

Nested elements are supported using a "." to separate the elements:

```
Element element = element("address.street").in(Customer.class);

value = element.in(customer).value();
```

You can use any level of nesting:

```
Element element = element("customer.address.street").in(Response.class);

value = element.in(response).value();
```

5.5 Custom Elements

Some classes have a custom definition of elements. A `Map` has their keys as elements, an `Array` has their indexes as elements an so on. Elements are found by an element finder (a class that implements `Finder<Element>`) and you can write a custom element finder and register it using the registry available through the method `Elements#registry`.

Trugger has custom element finders for a set of java core classes:

- `Map`: keys are used as the elements
- `ResourceBundle`: keys are used as the elements
- `Properties`: keys are used as the elements
- `ResultSet`: the column names are used as the elements
- `Annotation`: the methods as used as elements
- `List`: indexes are used as the elements (and also two special names, *first* and *last*)
- Arrays: indexes are used as the elements (and also two special names, *first* and *last*)

It is important to have clear that since this elements are instance specific, the elements should be queried by passing an instance instead of a class for the method `in` or an empty list will be returned. For a single elements, you may pass a class or an instance but using an instance is better because you can call the handling methods directly.

You can also use this custom element finders to copy elements easily:

```
// this will copy every element from the result set to the instance
copy().from(resultSet).to(myEntity);
```

6 Utilities

6.1 Annotation Mock

Annotations are interfaces and mocking it should be as easy as mocking an interface. The problem is the default values that can be omitted. Trugger has an utility module to help mocking and annotation by using the interception module.

To create a mock, you should start by creating an anonymous class that extends `mock.AnnotationMock` and maps the elements in a block code inside the class using the methods `map` and `to`.

```
Resource resource = new AnnotationMock<Resource>(){{
    map("name").to(annotation.name());
    map(false).to(annotation.shareable());
}}.createMock();

//returns "name"
String name = resource.name();

//return false
boolean shareable = resource.shareable();

//returns "" because it is the default value
String mappedName = resource.mappedName();

// returns javax.annotation.Resource class
Class<? extends Annotation> type = resource.annotationType();
```

If you don't like the anonymous class style, you can still use the classic style.

```
AnnotationMock<Resource> mock = new AnnotationMock<>(Resource.class);

mock.map("name").to(annotation.name());
mock.map(false).to(annotation.shareable());

Resource resource = mock.createMock();
```

6.2 Context Factories

If you need a lightweight component to invoke a constructor with a predicate based logic to resolve the parameter values, you can use the `ContextFactory`.

A `ContextFactory` is a factory that maps a predicate that evaluates parameters to an object or supplier. After creating a `ContextFactory`, you can manipulate the context through the `#context` method and create an object with the `create` method. A set of predicates can be found in `ParameterPredicates` class.

```
ContextFactory factory = new ContextFactory();
factory.context()
  //static imports
  .use(myImplementation)
    .when(type(MyInterface.class))
  .use(someObject)
    .when(named("component"))
  .use(parameter ->
      resolve(parameter.getAnnotation(MyAnnotation.class)))
    .when(annotatedWith(MyAnnotation.class))
  .use(() -> availableWorker())
    .when(type(MyWorker.class));
```

The above factory will:

1. use `myImplementation` for any parameter of the type `MyInterface`
2. use `someObject` for any parameter named *"component"*
3. use the return of `resolve` with the annotation `MyAnnotation` for any parameter annotated with `MyAnnotation`
4. use the return of `availableWorker` to any parameter of type `MyWorker`

These steps will be done with every public constructor of a type, if a constructor has one parameter that cannot be resolved to an object, then the next constructor will be used and if there is no more constructors to use, an exception is thrown.

6.3 Component Factories

Component factories allows creating components defined by annotations. Suppose you have:

```
public @interface ComponentClass {

  Class<? extends Component> value();

}

@ComponentClass(MyComponentImplementation.class)
public @interface MyComponent {

  String name();

}

//inside a class

@MyComponent(name = "myName")
private String aField;
```

The annotation in `aField` can be used to create an instance of `MyComponentImplementation`. The context used to create any components are:

1. Every property of the annotation with their specific types (in that case, the property `name` with the value *"myName"* to a parameter named `name` and of type `String`)
2. The annotation itself with its type (in that case, the `MyComponent` annotation to the type `MyComponent`)

Since the annotation is used as the context, you can have a constructor in the component implementation that receives the annotation instead of its properties. This is useful if you don't want to compile your code with `-parameters` parameter.

This behaviour is completely replaceable by using the method `configureContextWith`. To add behaviour to the default one, compose the `ComponentFactory#defaults` with your behaviour:

```
factory.configureContextWith(
  defaults().andThen(
    (context, annotation) -> yourConfigurations
  )
);
```

To instantiate a component, just use a code like this one:

```
ComponentFactory<ComponentClass, Component> factory =
  new ComponentFactory(ComponentClass.class);

// get the annotation from the field

Component component = factory.create(annotation);
```

Alternatively, you can get a list of components by passing an `AnnotatedElement` to the method `#createAll`:

```
Element = Elements.element("aField").in(myObject);
List<Component> components = factory.createAll(element);
```

Or creating a single one by passing an `AnnotatedElement` to the method `#create`:

```
Element = Elements.element("aField").in(myObject);
Component component = factory.create(element);
```

7 Validation

The validation module is not a replacement for the *Bean Validation*. It combines the utility factories and Element module to provide a basic and simple engine to do only validations (message production is out of scope). This is a good component for backend validations or even to integrate validation in proprietary or old infrastructure codes.

7.1 Basic Validation

To validate an object, you can simply use the Validation class and the provided DSL:

```
ValidationResult result = Validation.engine().validate(object);
```

The result will give all information needed to integrate the validation to almost all frameworks and architectures. (You can access values and use reflection and other DSLs using the invalid elements.)

7.2 Validation using filters

You can pass a selector for restrict the elements to validate:

```
ValidationResult result = Validation.engine()
  .filter(ofType(String.class)))
  .validate(object);
```

The given filter will affect any nested validations that relies on a supplied ValidationEngine.

7.3 Creating Validators

7.3.1 Basic Validators

The validation constraints are defined using two components: an Annotation for defining the constraint and a Validator to implement it.

Example: A basic validator for null objects.

```
@Retention(RetentionPolicy.RUNTIME)
@ValidatorClass(NotNullValidator.class)
public @interface NotNull { }

public class NotNullValidator implements Validator {

  public boolean isValid(Object object) {
    return object != null;
  }

}
```

You can now annotate a field or a getter method:

```
public class Person {

  @NotNull
  private String name;

  //...

}
```

7.3.2 Type Validators

You can create validators for a specific type or a set of types using the validator generic type or the `MultiTypeValidator` component.

Example: a regex validator.

```
@Retention(RetentionPolicy.RUNTIME)
@ValidatorClass(PatternValidator.class)
public @interface Pattern {

  String value();

}

public class PatternValidator implements Validator<CharSequence> {

  private final String pattern;
  private final int flags;

  // the constraint will be automatically injected in the constructor
  public PatternValidator(Pattern constraint) {
    this(constraint.value(), constraint.flags());
  }

  public boolean isValid(CharSequence value) {
    if(value != null) {
      return true;
    }
    java.util.regex.Pattern pattern =
      java.util.regex.Pattern.compile(annotation.value());
    return pattern.matcher(value).matches();
  }
}
```

In this example, everything that is a CharSequence can be validated with `@Pattern`. If you need to change the behaviour of the constraint validation without creating one constraint for each type, just use the `MultiTypeValidator` (as a superclass or using composition) to map each type to a validator.

```
public class NotEmptyValidator extends MultiTypeValidator {

  @Override
  protected void initialize() {
    map(Collection.class).to(this::isCollectionValid);
    mapArray().to(array -> array.length == 0);
    map(Map.class).to(map -> !map.isEmpty());
  }

  private isCollectionValid(Collection collection) {
    return !collection.isEmpty();
  }

}
```

7.3.3 Validation of arguments

Take a look at the first instruction in the validation method of `PatternValidator`. That check can be a pain if you doesn't like boilerplate code. To avoid that you can use validations in parameter declared in `isValid` method.

```
public boolean isValid(@NotNull CharSequence value) {
  java.util.regex.Pattern pattern =
    java.util.regex.Pattern.compile(annotation.value());
  return pattern.matcher(value).matches();
}
```

Much more elegant!

7.3.4 Dependency Injection

7.3.4.1 Constraint values The constraint can be injected in the constructor by passing the entire constraint (like in the PatternValidator example above) or its properties using their names (in case you compile the code using "-parameters") or their types (much less accurate, but works for constraints that has properties with different types).

The `PatternValidator` above may have a constructor `public PatternValidator(String pattern, int flags)` to get rid of the constraint dependency.

7.3.4.2 Validation Engine If your validator requires a ValidationEngine, you can define a parameter in the constructor and an engine will be injected here. The difference between this and calling `Validation.engine()` directly is that the injected engine will have the same filter applied before the validation (using the method `ValidationEngine#filter`).

7.3.4.3 Target Object If your validator requires the object being validated, just declare a parameter annotated with `TargetObject` and the target will be injected there.

7.3.4.4 Element If your validator requires the `Element` being validated, just declare a parameter of type `Element` and it will be injected there.

7.3.4.5 Target Elements This is one of the most useful injections. To explain this injection, suppose you have two dates and one must be after the other.

```
public class Ticket {

  private Date leaving;

  private Date arrival;

  //...

}
```

Validating the `arrival` field will require access to the `leaving` field. This can be done by using element references in the annotation. Let's look the `@After` constraint:

```
@Retention(RetentionPolicy.RUNTIME)
@ValidatorClass(AfterValidator.class)
public @interface After {

  @TargetElement("reference")
  String value();

}

public class AfterValidator implements Validator<Date> {

  private final Date referenceValue;

  public AfterValidator(@NotNull Date reference) {
    this.reference = reference;
  }

  public boolean isValid(@NotNull Date value) {
    return value.after(referenceValue);
  }

}
```

Notice that the references can be validated in constructor. If the reference is invalid the validation will not be processed.

The class will now looks like this:

```
public class Ticket {

  @NotNull
  private Date leaving;

  @NotNull
  @After("leaving")
  private Date arrival;

  //...

}
```

The value that will be passed to the constructor will be the element with the same name as the `value` property in the constraint `After`. The dependency resolution will use the property name (in the example, `value`) and a parameter name (in the example, `reference`) to help finding the value for each constructor parameter (remember to use “-parameters” to compile the code).

7.3.5 Merging Invalid Elements

Sometimes is useful to do a complete validation over the element value and incorporate their invalid elements in the main result. To do such a validation, just add a constructor parameter of type `ValidationEngine`:

```
public class ValidValidator implements Validator {
  private final ValidationEngine engine;

  public ValidValidator(ValidationEngine engine) {
    this.engine = engine;
  }

  public boolean isValid(@NotNull Object value) {
    return !engine.validate(value).isInvalid();
  }

}
```

This solves the validation problem. But the invalid elements are lost in the validation result inside the `ValidValidator`. To incorporate this elements, just annotate the constraint that maps to the `ValidValidator` with `@MergeElements`:

```
@MergeElements
@Retention(RetentionPolicy.RUNTIME)
@ValidatorClass(ValidValidator.class)
public @interface Valid {
}
```

This will make the ValidationEngine passed to the validator to incorporate the invalid elements into the invalid result using nested elements without changing any code in the validator.

7.3.6 Shared Validators

If your validator is thread safe, you can mark it to be shared across every validation by using the annotation `@Shared` on it. Keep in mind that validators that uses constraint properties or dependencies related to the current validation (like a validation engine) will not behave well if they are shared.

7.4 Domain Validations

Suppose you have some properties that requires more than one validation. You can group then into a single constraint (a domain constraint) using the DomainValidator as the @ValidatorClass.

Example:

```
@NotNull
@After("leaving")
@Retention(RetentionPolicy.RUNTIME)
@ValidatorClass(DomainValidator.class)
public @interface ArrivalDate {}

public class Ticket {

  @NotNull
  private Date leaving;

  @ArrivalDate
  private Date arrival;

  //...

}
```

By using the `DomainValidator` you can group validations to create a domain constraint and reuse code. Just keep in mind that merging elements is a main constraint, so you must annotate the domain constraint with `@MergeElements` to merge the invalid elements (if you group a `@Valid` constraint, for example).

8 Extending

8.1 How To Implement the Fluent Interfaces

The fluent interfaces are always defined through java interfaces and may be customized by your own implementation. Trugger uses a `ServiceLoader` to load a factory that knows the implementations to instantiate, so you can override the implementation of any fluent interface by defining a file in your **META-INF/services** directory with the factory implementation.

The factory interfaces that can be customized are listed bellow:

- `ElementFactory`: used for reflecting elements
- `ReflectionFactory`: used for reflection in general
- `ClassScannerFactory`: used for class scanning
- `InterceptorFactory`: used for method interception
- `ValidationFactory`: used for validation

9 Changelog

9.1 Version 5.1

This release marks the reborn of **Validation** module in a different way. The validation module now is a simple solution for manually validating objects (useful in backend processing). It is not an implementation for the *JSR-303* nor a competitor but it includes features that I always wanted in a validation engine to help creating elegant validators with less code.

Also this releases brings a new package name because `trugger` is now in the Maven Central Repository. You should replace your imports from `org.atatec.trugger` to `tools.devnull.trugger`.

9.1.1 New Modules

- Validation (completely from scratch and much better than the old and ugly validation module)

9.1.2 New Features

- `InterceptionHandler` for validating method arguments using the Validation module
- Component `ArgumentsValidator` for validating arguments of constructors and methods
- New element finder for lists behaving as the element finder for arrays

9.1.3 Major Changes

- Predicate for parameters assignable to a given type (use with caution)
- Predicate for primitive array types
- Method `ParameterPredicates#name` renamed to `named`
- `Context` now throws an `UnresolvableValueException` instead of returning null to indicate an unresolvable value.
- Changed `ValueHandler#get` to `#value`
- Renamed `ElementPredicates#type` to `ofType`
- `AnnotationMock` removed because its functionality can be achieved using Mockito and a few lines of code.
- `Registry` and `ImplementationLoader` moved to package **util**

9.1.4 Minor Changes

- Context Factories now sorts the constructors descending using their parameter counts
- EasyMock replaced by Mockito

9.2 Version 5.0

This is a huge update. I'm going to focus on keep this project simple and easy to maintain (since I don't have money to put a great effort on it). Lots of code changed and lots of modules gone to `/dev/null`.

This release is also the first one to have a README with a basic guide.

9.2.1 Major Changes

- Java 8 support
  - Removed **Predicate**, **Iteration** and **Date** modules
  - Removed the tons of selector classes (lambdas are now a good solution) and everything are now centred in `filter` methods
- Renamed `Invoker#handlingExceptionsWith` to `onError`
- Unique searches no longer throws exceptions if two or more results are found
- Reformulated the Interception module, now the DSL implementation can be changed
- Immutable classes
- Element copies only applies functions to non null values (to prevent *NPEs*)
- No more single selections for field and method without specifying a name
- No more exception handlers for invocations (method and constructor)
- Removed `Reflection#newInstanceOf` method
- Changed the return of selections to `List`
- Renamed get and set methods in `ValueHandler`
- Reformulated Class Scan module
- `recursively` changed to `deep` and belongs now to `DeepSelector`
- ContextFactory utility class
- ComponentFactory utility class (a grown up version of the removed `AnnotationBasedFactory`)

9.2.2 Minor Changes

- Removed non used classes
- Renamed some methods and classes to improve code readability

9.3 Version 4.3

9.3.1 Major Changes

- Using `ServiceLoader` for loading implementations

9.4 Version 4.2

Three modules removed since I don't have time/money to maintain them.

9.4.1 Modules removed

- Validation
- Bind
- Annotation (only Domain Annotation was kept)

9.4.2 Minor Improvements

- `Reflection#newInstanceOf` uses `Utils#resolveType`

9.5 Version 4.1.1

9.5.1 Bug Fixes

- Fixed a bug with getter method finder on overrided methods.

9.6 Version 4.1

9.6.1 Major Changes

- Added the possibility to reflect only public members of a class or the declared ones.
- `ImplementationLoader#getInstance` renamed to `instance`
- Removed `AccessSelector`
- Refactor on `ReflectionPredicates`
- Getter and Setter specs changed to include more cases
- Iteration DSL improved
  - Finding operation separated in Find class
- Class scan DSL improved
- Interceptor class simplified and improved DSL to create a proxy
- Reflection DSL improved
- Predicates improved
- Predicable interface removed
- Changed scope of EasyMock to test
  - Mocks for Element and ElementFinder are now in test packages
  - Annotation mock still in the binaries, but is implemented using Interceptor class instead of EasyMock
- Bind DSL improved
- Validation DSL improved
  - Removed pluggable binders and factories (use the composite ones to override the default ValidationFactory

9.6.2 Other Changes

- Using Gradle as the build system (good bye, Maven)

9.7 Version 4.0.1

- Fixed a bug when using a non-named field as selector for binding.

9.8 Version 4.0

9.8.1 New Modules

- Exception Handling

9.8.2 Major Changes

- Packages renamed from **net.sf** to **org.atatec** (back to the origin ^_^)
- pom groupId moved to **org.atatec.trugger**
- Reflection Invokers and Handlers does not throw a NPE if a null object is passed
- Class hierarchy helpers removed. Use `ClassIterator` and `Reflection#hierarchyOf`
- Support for JBoss Virtual Filesystem 3.1 (allows class scanning in JBoss AS 7.x)
- Improved API for registering a ResourceFinder.
- Method `RegistryEntry#registry` renamed to `RegistryEntry#value`
- Method `Predicates#newComposition` removed. Use `Predicates#is` instead.
- ElementFinder for arrays.

9.8.3 Minor Changes

- Using SoftReferences to cache annotation and object elements.

9.9 Version 3.1

9.9.1 Major Changes

- Changed method `PredicateSelector#thatMatches` to `that`
- Transformers to all wrapper classes
- Simplified methods like `elementsMatching` and `allElements` in iteration module.

9.10 Version 3.0

9.10.1 Major Changes

- Removed **cglib** and **commons collections** dependency
- Removed PredicateDSL and TransformerDSL
- Performance improvements in validation
- Interceptor used only for interfaces

9.11 Version 2.8

9.11.1 Major Changes

- Reformulated Reflection DSL

- Code improvements
- Support for **JBoss Virtual Filesystem** (allows class scanning in JBoss AS 5.x and 6.x)

9.11.2 Bug fixes:

- ElementMockBuilder now supports call for getAnnotation method.

9.12 Version 2.7

9.12.1 New Modules:

- Domain Annotations

9.12.2 Major changes:

- Reformulated DSL Criteria.
- Refactor in Interception module
- Non-named element and field selector for bind operations.
- AnnotationBased factory now searchs in the entire annotation levels.
- Renamed method `Bind#newBind` to `newBinder`.
- More operations in Reflector.
- Renamed `BindableElement#getTarget` method to `target`.
- Removed generic type “?” from `ReflectionPredicates` and `ClassScanner`.

9.12.3 Bug fixes:

- Binder can now bind null values.

9.13 Version 2.6

9.13.1 Major changes:

- Validation Binder for using Seam Components inside of a validator.
- Generic type reflection without specifying the generic parameter name.
- Additions to PredicateDSL.
- Renamed the bind(Resolver) method to use(Resolver).
- Merged Field elements and Property elements, so, if you want a property behaviour use the Properties class instead of Elements.
- New DSL for transform operations.

9.13.2 Bug fixes:

- Element type check on element copies.

9.14 Version 2.5

9.14.1 Major changes:

- New Element selection without specifying a name.
- Created a package for holding annotations for general use.
- Reverted method `genericTypeFor` to `genericType` (the shortest name is better - **IMHO**).
- New invocation tracker interceptor.
- New DSL for predicates.
- Scan for one single class.
- New method newInstanceOf in Reflection class.

9.15 Version 2.4

9.15.1 New modules:

- DSL for iterations
  - Refactoring in Collection module to Iteration module. Now the iterations are made using an iterator rather than a collection.

9.15.2 Major changes:

- New ValidatorBinder component for customize binds.
- Exceptions for validation module.
- `@Valid` can now use context and propagates the root context to nested validators.
- `@ValidationContext` for bind the context into a Validator.
- Handler for multiple Element objects.
- Removed the Trugger class (since the project grows, it became very large).

9.15.3 Bug fixes:

- Non-null return if the element is not found in the annotation and the target is not a class.

9.16 Version 2.3.1

9.16.1 Major changes:

- Added a method to remove a registry entry (included in a minor release since its a useful method).

9.17 Version 2.3

9.17.1 New modules:

- DSL for class scan.

9.17.2 Major changes:

- Implementations are now configured through Registry interface.
- Utility class for creating Factories.
- New selection for non-named fields or methods.
- Changed method `Reflector#bridgedMethod` to `bridgedMethodFor`.
- Changed method `Reflector#genericType` to `genericTypeFor`.
- New search operation for CollectionHandler.
- Some validators for use in Brazil.
- Renamed `ValidationStrategy#breakOnFalse` method to `breakOnFirstInvalidObject`.
- Moved the ClassLoader option to ProxyCreator.

9.17.3 Minor changes:

- Source encode changed to UTF-8.

9.18 Version 2.2.1

9.18.1 Minor changes:

- Maven support.

9.19 Version 2.2

9.19.1 Major changes:

- `ValidatorFactory` now receives a context for doing the binds without the `ValidationEngine`.
- New validators.

9.19.2 Bug fixes:

- Valid annotation doesn't validate the mapped context.
- NumberFormatException if `min` or `max` of Range is not defined.

9.20 Version 2.1

9.20.1 New modules:

- DSL for creating proxies.
- DSL for Validation.

9.20.2 Major changes:

- Requirements:
  - cglib (and its dependencies).
  - hibernate-validator (for compile and adapter use).
- Mock for annotations.
- New methods in AnnotationElementSelector.
- Removed the `nonSynthetic` selection (used by fields and methods selectors).
- New `assignableTo` selection in `TypedElementSelector`.
- Predicates for strict and assignable types.
- New interfaces added to `MethodSelector` and `ConstructorSelector`.
- Selection for return type on methods renamed to `returning`.
- Changed selection `SetterMethodSelector#ofType` to `forType`.
- Improved ElementMockBuilder class.
  - Removed notAnnotatedWith method because the class is using now the nice mock instead of the normal mock.
  - The annotations added will be returned by getDeclaredAnnotations and getAnnotations.

9.20.3 Bug fixes:

- Fixed a bug in TruggerElementsSelector that returns a null predicate if no selection is made.

9.21 Version 2.0

9.21.1 New modules:

- DSL for date operations.

9.21.2 Major changes:

- Java 6
- Easy mock dependency (only for mock package).
- Renamed method `getPredicate` to `predicate` in PredicateBuilder class.
- Removed the prefix `all` from `Reflector` and `Elements` methods.
- Corrected misspelling of method `isEmpty` *(epic fail)*.
- Interface `BaseElementSelector` renamed to `ElementSpecifier`.
- New `FieldSpecifier` interface.
- Changed to `in` the name of the method `on` in `MethodInvoker` interface.
- Removed the bind to properties.
- Mocks for helping tests.
- Removed the Alias annotation.
- Refactored `ImplementationLoader` class.

- Changed `getType` to `type`, `getDeclaringClass` to `declaringClass` and `getName` to `name` in `Element` class.
- All references to Property (interface, selector, etc.) are now referenced as Element.
- `AnnotationProperty` migrated to **Element** module.
- Renamed the `ElementCopy` methods `fromElement` and `toElement` for `sourceElement` and `destinationElement`.
- Swapped the src and dest in Element copy.
- Renamed the methods in Properties and Elements class.
- Specific Elements.
- Removed the nested properties, nested elements should be used now.
- Added another way to define custom finders (via “define” method on Elements).
- Added custom element finders for:
  - ResultSet
  - ResourceBundle
  - Properties
  - Map
- Added the Finder interface.
- Removed the possibility of specifying elements from fields or properties.
- New exception hierarchy.
  - Removed `PropertyManipulationException` and added `HandlingException`.
  - Renamed `UnreadablePropertyException` to `UnreadableElementException` and `UnwritablePropertyException` to `UnwritableElementException`

9.21.3 Minor changes:

- Method toString implemented in predicates for better debugging.
- New ofType method in ElementPredicates.
- Changed the return type of ElementPredicates.assignableTo to CompositePredicate.

9.22 Version 1.2

9.22.1 Major changes:

- Removed the listeners in the ImplementationLoader for prevent anonymous changes.
- HierarchySelector replaced by RecursionSelector for better abstraction.
- Merged the property factories (finder, selector and copy) into a new one with the “property” alias.
- Changed the “composite-predicate” alias to “predicate”.
- Renamed the CompositePredicateFactory to PredicateFactory.
- Changed the name of the method “on” (FieldHandler) to “in”.
- Changed the name of PropertyHandler to Properties.

9.22.2 New features and improvements:

- Improved the cache for Object properties (ObjectPropertyFinder).

- New reflection of getter and setter methods for a field object.
- New element module for more flexibility.
- The collection operations now returns the number of affected elements.
- New count operation for collections.
- New Predicable interface for converting objects into predicates.
- Removed the ClassUtils class, its methods are now in the new Utils class.
- Removed the methods to return the handled objects (for Field, Method and Constructor).
- Some improvements in ImplementationLoader class.
- New bind module.
- Added the `nonFinal` selection to the MemberSelector interface.

9.22.3 Bug Fixes:

- Fixed a bug in the CLASS predicate (it allows annotations).

9.23 Version 1.1

9.23.1 New features and improvements:

- Added the AnnotatedElementSelector to the FieldSelector.
- Unnecessary generic types removed.
- Added a package for the selectors.
- Changed the property selectors.
- Replaced the HierarchyResult with the HierarchySelector
- Changed the named member reflection predicate to take only one argument.
- Refactoring in the properties implementation.
- Refactoring in the reflection implementation.
- More selector interfaces:
  - PredicateSelector
  - AccessSelector
  - GetterMethodSelector (used by Reflector)
  - SetterMethodSelector (used by Reflector)

9.24 Version 1.0.1

9.24.1 New features and improvements:

- Improved the object property resolution if a class uses more than one setter to a property.
- Added tests to the reflection predicates.

9.24.2 Bug fixes:

- Fixed a bug in the not assignable class predicate.