Rocoto

v. 2.0

User Guide

The Rocoto Team **2010-01-20**

Table of Contents

1 Home

1.1 Welcome to Rocoto!

Rocoto is a small collection of reusable Modules for *Google Guice* to make easier the task of loading `java.util.Properties` by reading configuration files.

Rocoto is one of the most spicy pepper in South America, very popular in Peru and well known since the age of Incas... it adds some spice to Google Guice through configuration files!

1.2 Before Coding...

To set up your project, configure in your pom.xml the `repository`:

```
<repositories>
    ...
    <repository>
        <id>rocoto-repository</id>
        <name>Rocoto Repository for Maven</name>
        <url>http://rocoto.googlecode.com/svn/repo</url>
        <layout>default</layout>
    </repository>
    ...
</repositories>
```

1.3 Acknowledgements

This work is dedicated to our city, L'Aquila, destroyed by a terrible earthquake the 6th April, 2009... That day more than 300 people were killed because buildings collapsed after a magnitudo 6.3 earthquake at 3:32 am.

We'll never forget that episode.

2 Simple Configuration

2.1 The Simple Configuration module

The *Simple Configuration* module is a dependencies-less, small (jar size is less than ~20KB) light yet powerfull *Guice* module to easy load configuration properties and bing them to the *Guice Binder*.

Users that want ot use the *Simple Configuration* module in their projects, users have to add in their `pom.xml` the following `dependency`:

```
<dependencies>
    ...
    <dependency>
        <groupId>com.google.code.rocoto</groupId>
        <artifactId>rocoto-simple-configuration</artifactId>
        <version>XX.XX</version>
    </dependency>
    ...
</dependencies>
```

Then users are ready to load configuration files, but first create the module; once configured, is it possible to create your injector:

```
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.google.code.rocoto.simpleconfig.SimpleConfigurationModule;
...
SimpleConfigurationModule configurationModule = new SimpleConfigurationModule();
/*
 * adds your configuration files
 */
Injector injector = Guice.createInjector(configurationModule,
    ...
);
```

Finally, users can access to configuration parameters throug the `@com.google.inject.name.Named` annotation.

2.1.1 Adding properties files from Classpath

In many cases developers are used to include their properties file into wars or jars, then reloading them from the classpath.

The `com.google.code.rocoto.simpleconfig.SimpleConfigurationModule` offers a set of methods that simplifies that loading operation, by specifying the full qualified classpath resource name:

```
module.addProperties("com/acme/jdbc.properties");
module.addProperties("/com/acme/ldap.properties");
```

If needed, users can also specify the `ClassLoader` needed to load the pointed resource(s):

```
ClassLoader myClassLoader = [...];
...
module.addProperties("com/acme/jdbc.properties", myClassLoader);
module.addProperties("/com/acme/ldap.properties", myClassLoader);
```

The `com.google.code.rocoto.simpleconfig.SimpleConfigurationModule` also supports the *XML* definition of properties file:

```
module.addXMLProperties("com/acme/jdbc.xml");
module.addXMLProperties("/com/acme/ldap.xml");
```

and, of course, also for *XML* it is possible to specify the `ClassLoader`:

```
ClassLoader myClassLoader = [...];
...
module.addXMLProperties("com/acme/jdbc.xml", myClassLoader);
module.addXMLProperties("/com/acme/ldap.xml", myClassLoader);
```

2.1.2 Adding Properties files from the FileSystem

The `com.google.code.rocoto.simpleconfig.SimpleConfigurationModule` is able to load typical and *XML* properties files from the filesystem by specifying the `java.io.File` location:

```
import java.io.File;
...
module.addProperties(new File("etc/com/acme/jdbc.properties"));
module.addProperties(new File("etc/com/acme/ldap.xml"));
```

By default, using this API, files wich name matches with `**/*.properties` pattern will be threated as typical properties files, `**/*.xml` instead as *XML* properties definition.

If the specified `java.io.File` is a directory, it will be traversed looking for `**/*.properties` and `**/*.xml` properties files and will be loaded in the configuration; given the following directory:

```
etc
|-- com
|   |-- jdbc.properties
|   |-- rocoto
|   |   |-- should-be-ignored.txt
|   |   `-- simpleconfig
|   |       |-- memcached.xml
|   |       `-- should-be-ignored.txt
|   `-- should-be-ignored.txt
|-- ibatis.properties
`-- should-be-ignored.txt
```

all txt files will be ignored, all others will be load as properties files.

Users can customize the default behavior of properties files pattern matching, by specifying their `com.google.code.rocoto.simpleconfig.AbstractPropertiesFileFilter` implementation, where it is possible specifying the files patterns:

```
import com.google.code.rocoto.simpleconfig.AbstractPropertiesFileFilter;
...
module.addProperties(new File("etc"), new AbstractPropertiesFileFilter(
    "**/*.config", // specify here the 'old-style' properties file pattern
    "**/*.p?ml" // specify here the XML properties file pattern
) {});
```

As described earlier, patterns are used for the inclusion and exclusion of files. These patterns look very much like the patterns used in Apache Ant (from wich the code has been kindly borrowed :P):

- `**` matches zero or more 'directories' in a path; * * matches zero or more characters; * ? matches one character.

2.1.3 Adding Properties files from URLs

The `com.google.code.rocoto.simpleconfig.SimpleConfigurationModule` also supports the properties loading from URLs, by specifying the `java.net.URL`:

```
import java.net.URL;
...
module.addProperties(new URL("http://acme.com/config/jdbc.properties"));
module.addXMLProperties(new URL("http://acme.com/config/ldap.xml"));
```

2.1.4 Adding Java System Properties

In many cases users need to acces to Java System Properties by invoking `System.getProperty("java.version")` or `System.getProperties()`.

By enabling them to the configuration module, users can replace the Java System Properties retrieving operation with the Dependency Injection:

```
...
module.addSystemProperties();
```

2.1.5 Adding Environment Variables

In Java5 the Environment Variables are accessible through `System.getenv()` and `System.getenv("JAVA_HOME")`; like for Java System Properties, users can adding Environment Variables simply by invoking:

```
...
module.addEnvironmentVariables();
```

By default Environment Variables will be referenced into the configuration with the `env.` prefix, but users are free to specify their preferred one:

```
...
module.addEnvironmentVariables("environment");
```

2.1.6 Adding existing configurations

The `com.google.code.rocoto.simpleconfig.SimpleConfigurationModule` allows users to plug already existing properties configuration:

```
Properties p = [...]
...
module.addProperties(p);
```

or

```
Map<String, String> m = [...]
...
module.addProperties(m);
```

2.1.7 ${}, the *Apache Ant* variables style

The `com.google.code.rocoto.simpleconfig.SimpleConfigurationModule` supports the well known *${}* expression to define placeholders, which scope is the whole configuration, that means that users can define some commons properties in one properties file:

```
commons.host=localhost
commons.port=8080
...
```

then referencing them in different files, loaded in the same configuration:

```
ldap.host=${commons.host}
ldap.port=${commons.port}
...
```

and

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties version="1.0">
    <entry key="proxy.host">${commons.host}</entry>
    <entry key="proxy.port">${commons.port}</entry>
    ...
</properties>
```

3 Configuration

3.1 The Configuration module

The *Configuration* module is an easy-to-use Apache commons-configurations wrapper, built for users that require binding more complex configuration format.

Users that want ot use the *Configuration* module in their projects, users have to add in their `pom.xml` the following `dependency`:

```
<dependencies>
    ...
    <dependency>
        <groupId>com.google.code.rocoto</groupId>
        <artifactId>rocoto-configuration</artifactId>
        <version>XX.XX</version>
    </dependency>
    ...
</dependencies>
```

Then users are ready to load configuration files, but first create the module; once configured, is it possible to create your injector:

```
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.google.code.rocoto.configuration.ConfigurationModule;
...
ConfigurationModule configurationModule = new ConfigurationModule();
/*
 * adds your configuration files
 */
Injector injector = Guice.createInjector(configurationModule,
    ...
);
```

Finally, users can access to configuration parameters throug the `@com.google.inject.name.Named` annotation.

3.1.1 Adding configuration

Users can easily add existing `org.apache.commons.configuration.Configuration` by invoking the method:

```
import org.apache.commons.configuration.Configuration;
...
Configuration conf = [...]
...
module.addConfiguration(conf);
```

For example, users can add Java System Properties configuration:

```
import org.apache.commons.configuration.Configuration;
import org.apache.commons.configuration.SystemConfiguration;
...
module.addConfiguration(new SystemConfiguration());
```

or the provided `com.google.code.rocoto.configuration.EnvironmentConfiguration` that loads the Environment Variables:

```
import org.apache.commons.configuration.Configuration;
import com.google.code.rocoto.configuration.EnvironmentConfiguration;
...
module.addConfiguration(new EnvironmentConfiguration());
```

In this case, Environment Variables will be prefixed with `env.`, but users are free to customize it:

```
import org.apache.commons.configuration.Configuration;
import com.google.code.rocoto.configuration.EnvironmentConfiguration;
...
module.addConfiguration(new EnvironmentConfiguration("environment"));
```

3.1.2 Adding File based Configurations

Configurations based on textual files are widely supported by *Apache commons-configuration* by the org.apache.commons.configuration.FileConfiguration interface; the `com.google.code.rocoto.configuration.ConfigurationModule` allows loading them by specifying the type, usually a class that implements the `org.apache.commons.configuration.FileConfiguration` interface, the source and the charset encoding, *UTF-8* by default.

3.1.2.1 From a Classpath resource

By specifying the full qualified classpath resource name:

```
Class<? extends FileConfiguration> configurationType = [...]
module.loadConfiguration(configurationType, "com/acme/config.ext");
```

If needed, users can also specify the `ClassLoader` needed to load the pointed resource(s):

```
ClassLoader myClassLoader = [...];
...
Class<? extends FileConfiguration> configurationType = [...]
module.loadConfiguration(configurationType, "com/acme/
config.ext", myClassLoader);
```

3.1.2.2 From a File

```
Class<? extends FileConfiguration> configurationType = [...]
module.loadConfiguration(configurationType, new java.io.File("etc/com/acme/
config.ext"));
```

3.1.2.3 From an URL

```
Class<? extends FileConfiguration> configurationType = [...]
module.loadConfiguration(configurationType, new java.net.URL("http://
acme.com/config/config.ext"));
```

3.1.3 Specifying the encoding

All methods shown above load the specified resources using *UTF-8* encoding, but users are free to changed it if needed through the `java.nio.charset.Charset`:

```
import java.nio.charset.Charset;
...
Class<? extends FileConfiguration> configurationType = [...]
module.loadConfiguration(configurationType, "com/acme/
config.ext", Charset.forName("UTF-16"));
```

4 Converters

4.1 The Converters module

The *Converters* module adds some string-to-type converters not already present in Google Guice, plus a simple to use module that makes easier the converters registration.

Users that want ot use the *Converters* module in their projects, users have to add in their `pom.xml` the following `dependency`:

```
<dependencies>
    ...
    <dependency>
        <groupId>com.google.code.rocoto</groupId>
        <artifactId>rocoto-converters</artifactId>
        <version>XX.XX</version>
    </dependency>
    ...
</dependencies>
```

4.1.1 The ConvertersModule

Thecore class is the `com.google.code.rocoto.converters.ConvertersModule` that's a Guice Module that simplifyes the `com.google.inject.spi.TypeConverter` registration.

Users can write their `com.google.inject.spi.TypeConverter` implementation:

```
import com.google.inject.TypeLiteral;
import com.google.inject.spi.TypeConverter;
public class MyConverter implements TypeConverter {
    public Object convert(String value, TypeLiteral<?> toType) {
       // perform here the conversion
    }
}
```

and registry it into the module by specifying the binding type:

```
import com.google.code.rocoto.converters.ConvertersModule;
...
ConvertersModule convertersModule = new ConvertersModule();
convertersModule.registerConverter(MyType.class, new MyConverter());
```

or bind it to a `com.google.inject.TypeLiteral`:

```
convertersModule.registerConverter(new TypeLiteral<List<MyType>>()
{}, new MyConverter());
```

or bind it to a `com.google.inject.matcher.Matcher`:

```
import com.google.inject.TypeLiteral;
import com.google.inject.matcher.Matcher;
Matcher<? super TypeLiteral<?> myMatcher = new MyMatcher();
convertersModule.registerConverter(myMatcher, new MyConverter());
```

finally, users have just to include the converters module when creating the `com.google.inject.Injector`:

```
import com.google.inject.AbstractModule;
import com.google.inject.Guice;
import com.google.inject.Injector;
...
Injector injector = Guice.createInjector(new ConvertersModule(), new AbstractModule() {
            @Override
            protected void configure() {
                this.bindConstant()
                    .annotatedWith(Names.named("charset"))
                    .to("UTF-8");
            }
        });
```

4.1.2 The Converters

The `com.google.code.rocoto.converters` package comes with default implementations of converers not already included in Google Guice, installed in the `com.google.code.rocoto.converters.ConvertersModule`.

Every converter throws runtime exceptions if invalid input are submitted to the conversion process.

Let's show and explain how they work:

4.1.2.1 BitSetConverter

Is the converter that converts a `java.lang.String` representation to a `java.util.BitSet`.

String representation is typically a CSV String mixed o chars and numbers, i.e:

```
a, 123, ~
```

in the example, `a` is taken in consideration as a `char`, `123` as an `int`, `~` as a `char`.

Note non numerical fragments with length great than 1 are not allowed!!!

4.1.2.2 CharsetConverter

Is the converter that converts a `java.lang.String` representation to a `java.nio.charset.Charset`.

4.1.2.3 CurrencyConverter

Is the converter that converts a `java.lang.String` representation to a `java.util.Currency`.

4.1.2.4 DateConverter

Is the converter that converts a `java.lang.String` representation to a `java.util.Calendar` and to `java.util.Date`.

By default, this converter manages the following ISO Date format representation:

- yyyy;
- yyyy-MM;
- yyyy-MM-dd;
- yyyy-MM-dd'T'hh:mmZ;
- yyyy-MM-dd'T'hh:mm:ssZ;
- yyyy-MM-dd'T'hh:mm:ss.sZ

  If users need to add new supported date formats, first they have to retrieve the *DateConverter*, then add a new pattern:

  ```
  DateConverter dateConverter = module.lookup(Date.class, DateConverter.class);
  ```

```
dateConverter.addPattern("EEE, MMM d, ''yy");
```

If users need to set the `java.util.Locale` and/or the `java.util.TimeZone`, first they have to retrieve the *DateConverter*, then set their preferences:

```
import java.util.Locale;
import java.util.TimeZone;
...
DateConverter dateConverter = module.lookup(Date.class, DateConverter.class);
dateConverter.setLocale(Locale.getDefault());
dateConverter.setTimeZone(TimeZone.getDefault());
```

4.1.2.5 FileConverter

Is the converter that converts a `java.lang.String` representation to a `java.io.File`

4.1.2.6 LocaleConverter

Is the converter that converts a `java.lang.String` representation to a `java.util.Locale`

The converter checks first if the input String matches with the pattern *languageCode_counrtyCode* to create the `java.util.Locale` otherwise will use the input value as locale language.

4.1.2.7 NumberConverter

Is the converter that converts a `java.lang.String` representation to a math number representation, such `java.math.BigDecimal` and `java.math.BigInteger`.

4.1.2.8 PatternConverter

Is the converter that converts a `java.lang.String` representation to a `java.util.regex.Pattern`

4.1.2.9 PropertiesConverter

Is the converter that converts a `java.lang.String` representation to a `java.util.Properties`

Note Input string has to match with the pattern *key1=value1\nkey2=value2...*.

4.1.2.10 SQLDateTimeConverter

Is the converter that converts a `java.lang.String` representation to:

- `java.sql.Date` (input has to match with `yyyy-MM-dd` pattern);
- `java.sql.Time` (input has to match with `HH:mm:ss` pattern);
- `java.sql.Timestamp` (input has to match with `yyyy-MM-dd HH:mm:ss.fffffffff` pattern).

4.1.2.11 TimeZoneConverter

Is the converter that converts a `java.lang.String` representation to a `java.util.TimeZone`.

4.1.2.12 URIConverter

Is the converter that converts a `java.lang.String` representation to a `java.net.URI`.

4.1.2.13 URLConverter

Is the converter that converts a `java.lang.String` representation to a `java.net.URL`, supporting the *classpath://* pseudo protocol, to load resources from the the class path.

Users that need to load classpath resources, have to specify the full qualified name of the resource. For example, given the class path resource:

```
com.acme.myapplication.JDBC.properties
```

following URLs point to the same resource:

```
classpath://com/acme/myapplication/JDBC.properties
classpath:///com/acme/myapplication/JDBC.properties
```

4.1.2.14 UUIDConverter

Is the converter that converts a `java.lang.String` representation to a `java.util.UUID`.

5 System

5.1 The System module

The *System* module is a light and alternative way to bind and inject Java System Properties and Environment Variables.

Users that want ot use the *System* module in their ptojects, users have to add in their `pom.xml` the following `dependency`:

```
<dependencies>
    ...
    <dependency>
        <groupId>com.google.code.rocoto</groupId>
        <artifactId>rocoto-system</artifactId>
        <version>XX.XX</version>
    </dependency>
    ...
</dependencies>
```

5.1.1 Binding Java System Properties

Binding Java System Properties is quick and easy; first of all annotate your POJO fields/method:

```
import com.google.inject.Inject;
import com.google.code.rocoto.system.SystemProperty;
public class MyPojo {
    @Inject
    @SystemProperty("user.home")
    private String userHome;
    @Inject
    public void setUserLanguage(@SystemProperty("user.language") String userLanguage) {
        ...
    }
}
```

then run the injector!

```
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.google.code.rocoto.system.SystemPropertiesModule;
...
Injector injector = Guice.createInjector(new SystemPropertiesModule());
```

5.1.2 Binding Environment Variables

Like Java System Properties, binding Environment Variables is quick and easy; first of all annotate your POJO fields/method:

```
import com.google.inject.Inject;
import com.google.code.rocoto.system.EnvironmentVariable;
public class MyPojo {
    @Inject
    @EnvironmentVariable("HOME")
    private String userHome;
    @Inject
    public void setUserLanguage(@EnvironmentVariable("CATALINA_HOME") String catalinaHo
        ...
    }
}
```

then run the injector!

```
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.google.code.rocoto.system.EnvironmentVariablesModule;
...
Injector injector = Guice.createInjector(new EnvironmentVariablesModule());
```