NYLA  
Java Framework  
Developer Guide

**Author:** Gregory Green

GIT Hub Repository

<https://github.com/ggreen/nyla>

Table of Contents

1 Overview 3

1.1 Target Audience 3

2 Utilities 3

2.1 Config 3

2.2 Cryption 6

2.3 Debugger 10

2.4 BeanComparator 13

2.5 Text 14

3 Searching Patterns 15

3.1 RELookup 15

Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Version Number | Date Updated | Revision Author | Brief Description of Changes |
| 0.1 | 4/25/2012 | Gregory Green | Creation |

# Overview

The purpose of this document is to provide an overview of the NYLA Solutions Global Java API. This API provides support for basic application utilities (application configuration, data encryption, debugger and text processions). It provides wrappers Inversion Control frameworks (Spring, Tuscany), Scripting (SPEL, Groovy), Emailing, HTTP/TCP IP communication, Reflection, JSON/XML and design pattern implementations (Commands, Proxy) and more.

## Target Audience

This document is intended for Java developers and architects. The readers are expected to be familiar with the Java programming language and development design pattern concepts.

# Utilities

See nyla.solutions.global.util

## Config

The Config provides a central mechanism for applications to access key/value property settings and encrypted passwords.

There are several ways to specify the configuration property file location.

1. Add file config.properties to CLASSPATH. This file will be loaded as a Java resource bundle.
2. Add the JVM argument -Dconfig.properties where the value is equal to the location of the configuration file.

Example:

-Dconfig.properties=/dev/configurations/files/system/config.properties

There are methods to get the String value property such as Config.getProperty(key) method. There are also methods to get an expected property value of a type such as Integer, Boolean, etc.

|  |
| --- |
| nyla.solutions.global.util.Config.mergeSystemProperties=false |

It also supports formatting several property values into a single property by adding the following property to the config.properties;

|  |
| --- |
| solutions.global.util.Config.useFormatting=true |

By default the configuration is read only once when the application is initialized. Add the following to the configuration property file to always reload the property whenever a getProperty... method is called. Note that this is a potentially an expensive operation.

|  |
| --- |
| solutions.global.util.Config.alwaysReloadProperties=true |

Note the following is a property file used for the sample usage code below.

|  |
| --- |
| application.name=JUNIT  debug=true  solutions.global.util.ConfigTest.integerProperty=24  password={cryption}102 42 -22 24 12 66 -35 89 50 -15 21 9 -67 73 -128 -105  solutions.global.util.Config.mergeSystemProperties=true  solutions.global.util.Config.useFormatting=true  application.name.debug=${application.name}.${debug}.${user.dir} |

USAGE

|  |
| --- |
| //Get a default string property  //The following assumes;  //application.name=JUNIT  String property = Config.*getProperty*("application.name");  Assert.*assertEquals*("JUNIT",property);    //An exception will be thrown if the referenced property does not exist in the property file  //in this case the ConfigException will be thrown  **try**  {  property = Config.*getProperty*("missing.property");  }  **catch**(ConfigException e)  {  //valid configuration exception  }    //Provide a default value if the default value is missing  property = Config.*getProperty*("missing.property","default");  Assert.*assertEquals*("default", property);    //Properties can be retrieved by type (boolean, Integer, Character, Long, Bytes)  //The following assumes;  //debug=true  **boolean** propertyBoolean = Config.*getPropertyBoolean*("debug");  Assert.*assertTrue*(propertyBoolean);    //Each getProperty<Type> accepts a default value  //The following assumes;  //missing.boolean.property=false  propertyBoolean = Config.*getPropertyBoolean*("missing.boolean.property",**false**);  Assert.*assertFalse*(propertyBoolean);    //Config has a user friendly way to associate properties with classes  //The properties can be prefixed with the class name  //Each getProperty<Type> optional accept the class name as the first argument  //The following assumes the property  //solutions.global.util.ConfigTest.integerProperty=24  **int** integerProperty = Config.*getPropertyInteger*(solutions.global.util.ConfigTest.**class**, "integerProperty");  Assert.*assertEquals*(24, integerProperty);      //Passwords encrypted with the solutions.global.util.Cryption object  //can be retrieved with the Config.getPassword(key) method  //An exception will be thrown if the password is not encrypted correctly in the property file  //The following is example encrypted password stored in the property file  //password={cryption} 2 -21 23 12 2 -21 23 12 2 -21 23 12 2 -21 23 12 2 -21 23 12  **char**[] password = Config.*getPropertyPassword*("password");  Assert.*assertNotNull*(password);      //Properties in the System.getProperties() can be merged with the Config's object properties  //This is done by setting the property  //solutions.global.util.Config.mergeSystemProperties=true  String jvmSystemPropertyName = "user.dir";  property = Config.*getProperty*(jvmSystemPropertyName);  Assert.*assertNotNull*(property);      //solutions.globa.util.Config.useFormatting property can be used to dynamically combine properties.  //This feature uses the solutions.global.patterns.decorator.style package (see Styles interface)  //The value of property surrounded with ${property.name} will be formatted by replacing it with the  //actual value from another property.    //The following is based on the following properties (note this combines the system property "user.dir")  //solutions.global.util.Config.useFormatting=true  //application.name.debug=${application.name}.${debug}.${user.dir}  property = Config.*getProperty*("application.name.debug");  Debugger.*println*(**this**,"property="+property);    Assert.*assertTrue*("All values formatted:"+property, property.indexOf("${") < 0); |

## Cryption

Cryption provides a set of functions to encrypt and decrypt bytes and text. It uses the javax.crypto package.

The default encryption algorithm is the Advanced Encryption Standard (AES).

The default algorithm can be changed with a configuration property named **nyla.solutions.global.util.Cryption.alogorithm**.

|  |
| --- |
| # The following sets the encryption algorithm to Data Encryption Standard (DES)  nyla.solutions.global.util.Cryption.algorithm=DES |

The Cryption object is used by nyla.solutions.global.util.Config object to decrypt properties prefixed with {cryption}. The Cryption class can be used to generate encrypted passwords that can be added to the config.properties file. The Cryption main method accepts a password and will print the encrypted password that can be added to the property file.

The printed password will be prefixed with the value “{cryption}”. Any properties prefixed with {cryption} in the config.properties is an indicator that content is encrypted.

|  |
| --- |
| The follow is a sample Cryption UNIX script:  export LIB\_DIR=put correct directory here  export CP="$LIB\_DIR/solution.global.jar"  java -classpath $CP solutions.global.util.Cryption $1 |

#The following is a sample output of an encrypted password generated by the Cryption main method.

|  |
| --- |
| {cryption}23 4 -3 -77 -128 -88 -34 -105 23 4 -3 -77 -128 -88 -34 -105 |

USAGE

|  |
| --- |
| //Get a default string property  //The following assumes;  //application.name=JUNIT  String property = Config.*getProperty*("application.name");  Assert.*assertEquals*("JUNIT",property);    //An exception will be thrown if the referenced property does not exist in the property file  //in this case the ConfigException will be thrown  **try**  {  property = Config.*getProperty*("missing.property");  }  **catch**(ConfigException e)  {  //valid configuration exception  }    //Provide a default value if the default value is missing  property = Config.*getProperty*("missing.property","default");  Assert.*assertEquals*("default", property);    //Properties can be retrieved by type (boolean, Integer, Character, Long, Bytes)  //The following assumes;  //debug=true  **boolean** propertyBoolean = Config.*getPropertyBoolean*("debug");  Assert.*assertTrue*(propertyBoolean);    //Each getProperty<Type> accepts a default value  //The following assumes;  //missing.boolean.property=false  propertyBoolean = Config.*getPropertyBoolean*("missing.boolean.property",**false**);  Assert.*assertFalse*(propertyBoolean);    //Config has a user friendly way to associate properties with classes  //The properties can be prefixed with the class name  //Each getProperty<Type> optional accept the class name as the first argument  //The following assumes the property  //solutions.global.util.ConfigTest.integerProperty=24  **int** integerProperty = Config.*getPropertyInteger*(solutions.global.util.ConfigTest.**class**, "integerProperty");  Assert.*assertEquals*(24, integerProperty);      //Passwords encrypted with the solutions.global.util.Cryption object  //can be retrieved with the Config.getPassword(key) method  //An exception will be thrown if the password is not encrypted correctly in the property file  //The following is example encrypted password stored in the property file  //password={cryption} 2 -21 23 12 2 -21 23 12 2 -21 23 12 2 -21 23 12 2 -21 23 12  **char**[] password = Config.*getPropertyPassword*("password");  Assert.*assertNotNull*(password);      //Properties in the System.getProperties() can be merged with the Config's object properties  //This is done by setting the property  //solutions.global.util.Config.mergeSystemProperties=true  String jvmSystemPropertyName = "user.dir";  property = Config.*getProperty*(jvmSystemPropertyName);  Assert.*assertNotNull*(property);      //solutions.globa.util.Config.useFormatting property can be used to dynamically combine properties.  //This feature uses the solutions.global.patterns.decorator.style package (see Styles interface)  //The value of property surrounded with ${property.name} will be formatted by replacing it with the  //actual value from another property.    //The following is based on the following properties (note this combines the system property "user.dir")  //solutions.global.util.Config.useFormatting=true  //application.name.debug=${application.name}.${debug}.${user.dir}  property = Config.*getProperty*("application.name.debug");  Debugger.*println*(**this**,"property="+property);    Assert.*assertTrue*("All values formatted:"+property, property.indexOf("${") < 0); |

## Debugger

Debugger provides useful methods for obtaining exception stack traces. It can build reader friendly strings for objects that do not implement their toString method.

It also provides a set of print functions to log DEBUG, INFO, WARN and FATAL level messages using the Debugger.println(...), Debugger.printInfo(...),Debugger.printWarn(...) and Debugger.printFatal(...) methods respectively.

The default log object implementation is solutions.global.operations.Log4J.

Set the configuration property to plug-in another logger (@see Config more information);

|  |
| --- |
| nyla.solutions.global.util.Debugger.logClass=className |

The logClass class name indicated must implement the solutions.global.operations.Log interface.

USAGE

|  |
| --- |
| //The Debugger toString(Object) can be used to debug objects where the toString method is not implemented.  String[] arraysNicely = { "first","second"};  String text = Debugger.*toString*(arraysNicely);  Assert.*assertEquals*("{[0]=first ,[1]=second}", text);  //The print method wraps log levels of DEBUG,INFO,WARN and FATAL  Debugger.*printInfo*("This is a INFO level message");  //Two arguments can be passed where the first is the calling object  //The debugger will prepend the calling objects class name to the logged output  Debugger.*println*(**this**,"This is a DEBUG level message");  //Debugger can be used to efficiently print exception information  text = **null**;  **try**{  text.toString(); //causes a null pointer exception  }  **catch**(NullPointerException e)  {  //Use the stackTrace method to get the string version of the exception call stack  String stackTrace = Debugger.*stackTrace*(e);  //Print warn level  Debugger.*printWarn*(**this**,stackTrace);  //stack trace will be automatically created if the exception object is passed  Debugger.*printError*(e);  Debugger.*printFatal*(**this**,"Stack trace will not be printed, because this is not an exception object.");  } |

Sample Debugger Properties (Config.properties)

|  |
| --- |
| #-------------------------------  # Debugger log instance uses Log4J  # You can directly include the log4J properties in the configuration property files (@see Config object)  # Log4J properties  #  log4j.rootLogger=DEBUG, stdout  log4j.logger.PACKAGE\_NAME=ERROR,file\_error  log4j.logger.YYY=DEBUG, file\_all  log4j.logger.org.apache=ERROR,stdout  log4j.logger.org.springframework=ERROR,stdout    #Standard OUT  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  #log4j.appender.stdout.layout.ConversionPattern=%d [%F:%L] - %x %m%n  #log4j.appender.stdout.layout.ConversionPattern= %p: %d{HH:mm:ss} [%F:%L] - %x %m%n  #log4j.appender.stdout.layout.ConversionPattern=%d{HH:mm:ss} [%c:%L] %m%n  #log4j.appender.stdout.layout.ConversionPattern=%d{HH:mm:ss} %m%n  log4j.appender.stdout.layout.ConversionPattern=%p: %d [%c] - %m%n    #  # FILE Output  log4j.file\_all.category=DEBUG  log4j.appender.file\_all=org.apache.log4j.RollingFileAppender  log4j.appender.file\_all.File=/temp/logs/system.log  log4j.appender.file\_all.MaxFileSize=10MB  log4j.appender.file\_all.MaxBackupIndex=3  log4j.appender.file\_all.layout=org.apache.log4j.PatternLayout  log4j.appender.file\_all.layout.ConversionPattern=%p: %d [%c] - %m%n      #  # FILE Output  #log4j.file\_error.category=ERROR  log4j.appender.file\_error=org.apache.log4j.RollingFileAppender  log4j.appender.file\_error.File=temp/logs/error.log  log4j.appender.file\_error.MaxFileSize=10MB  log4j.appender.file\_error.MaxBackupIndex=3  log4j.appender.file\_error.layout=org.apache.log4j.PatternLayout  log4j.appender.file\_error.layout.ConversionPattern=%p: %d [%c] - %m%n      #Emailing example  #email appender  log4j.appender.mail=org.apache.log4j.net.SMTPAppender  log4j.appender.mail.BufferSize=1  log4j.appender.mail.SMTPHost=smtp.myservername.xx  log4j.appender.mail.From=fromemail@myservername.xx  log4j.appender.mail.To=toemail@myservername.xx  log4j.appender.mail.Subject=Log ...  log4j.appender.mail.threshold=error  log4j.appender.mail.layout=org.apache.log4j.PatternLayout  log4j.appender.mail.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n |

## BeanComparator

BeanComparator is a generic Bean property java.util.Comparator implementation. It compares specified property beans using reflection. This object is internally used by the Organizer.sortByJavaBeanProperty(String,Collection) method.

USAGE

|  |
| --- |
| //The constructor accepts a JavaBean property name  BeanComparator beanComparator = **new** BeanComparator("firstName");  //The following are two sample user profile beans  UserProfile josiah = **new** UserProfile();  josiah.setFirstName("Josiah");    UserProfile nyla = **new** UserProfile();  nyla.setFirstName("Nyla");    //Reflection is used to compare the properties of the beans  Assert.*assertTrue*(beanComparator.compare(josiah, nyla) < 0);    //The following shows how the BeanComparator.sort method can be used    //This method can be used to sort an given collection based on the JavaBean properties  //of objects in the collection  ArrayList<UserProfile> unSorted = **new** ArrayList<UserProfile>();  unSorted.add(0, nyla);  unSorted.add(1, josiah);    //Setting the descending will determine the sort order  beanComparator.setDescending(**true**);  beanComparator.sort(unSorted);  Assert.*assertTrue*(unSorted.get(0) == nyla);    //Changing the descending flag changes the output of the sort method  beanComparator.setDescending(**false**);  beanComparator.sort(unSorted);  Assert.*assertTrue*(unSorted.get(0) == josiah); |

## Text

Text is geared toward string based processing. It includes template engine support like Free Marker that builds composite strings/values dynamically at runtime (see http://freemarker.sourceforge.net/). There are also methods to support complex regular expressions with Boolean AND, OR and NOT logic, numerous string conversions, general text manipulation and parsing methods.

USAGE

|  |
| --- |
| //Format text replacing template place-holders prefixed with ${ and suffixed by }  //with the corresponding values in a map.  String text = "${company} A2D2 Solution Global Application Testings";  Map<String,String> map = **new** HashMap<String,String>();  map.put("company", "EMC");  text = Text.*formatText*(text,map);  Assert.*assertEquals*("EMC A2D2 Solution Global Application Testings", text);  //Use complex text matching boolean logic to regular expressions by adding ${AND}, ${NOT} and $OR} tags  Assert.*assertTrue*(Text.*matches*("Kenya Africa", ".\*Kenya.\*"));  Assert.*assertFalse*(Text.*matches*("Kenya", "${NOT}.\*Kenya.\*"));  Assert.*assertTrue*(Text.*matches*("Kenya", "${NOT}${NOT}.\*Kenya.\*"));  Assert.*assertFalse*(Text.*matches*("America, Kenya, Paris", ".\*Paris.\*${AND}.${NOT}.\*Kenya.\*"));  Assert.*assertFalse*(Text.*matches*("America, Kenya, Paris", "(.\*Paris.\*${AND}.${NOT}.\*Kenya.\*)${OR}(.\*Paris.\*${AND}.${NOT}.\*Kenya.\*)"));  Assert.*assertTrue*(Text.*matches*("United States, Kenya, France", "${NOT}.\*America.\*${AND}.\*Kenya.${NOT}.\*Paris.\*"));  Assert.*assertTrue*(Text.*matches*("United States, Kenya, France", "${NOT}.\*America.\*${AND}.\*Kenya.${NOT}.\*Paris.\*"));  //Use the parse method to retrieve one or more token between a start and end strings  //Note the parse method can be used with non-regular expressions  String start = "Color:";  String end = ";";  Collection collection = Text.*parse*("Color:green; Weight:155oz; Color:Blue; Weight:23oz", start, end);  Assert.*assertEquals*(2,collection.size()); //two color  Iterator i = collection.iterator();  Assert.*assertEquals*("green", i.next()); //first is green  Assert.*assertEquals*("Blue", i.next()); //second is Blue  //There methods to count of a given character  **int** count = Text.*characterCount*('A', "All Apples");  Assert.*assertEquals*(2, count);  //There are methods the get digit counts  count = Text.*digitCount*(text);  Assert.*assertEquals*(2, count);  //Format text numbers/decimals  String format = "#,###,###.###";  String formattedText = Text.*formatNumber*(123443243240.033423,format);  Assert.*assertEquals*("123,443,243,240.033",formattedText);  //Format text currency  formattedText = Text.*formatCurrency*("1000.33");  Assert.*assertEquals*("$1,000.33",formattedText);  //format text percentages  formattedText = Text.*formatPercent*("2.3");  Assert.*assertEquals*("2.3%",formattedText);  //Use grep to search for expressions across multiple lines in a string  text = "Greg on line 1\nGreen on line two";  String results = Text.*grepText*("Green", text);  Assert.*assertEquals*("Green on line two",results); |

# Searching Patterns

## RELookup

The RELookup acts as a lookup table where the key is a complex regular expression. The expression and values are stored in a hash map. Then RELookup.get(argument) method is called, RELookup operation will iterate through the given get argument expressions looking for a match on the corresponding complex regular expression key.

The value column of the lookup table is used if the regular expression matches the argument.

|  |
| --- |
| ReLookup<FaultError> lookup = **new** ReLookup<FaultError>();    *assertTrue*(lookup **instanceof** Map);    lookup.put("(001)\*.\*Green.\*${AND}${NOT}.\*Blue.\*", **new** FaultError("0001","ERROR"));  lookup.put("(002)\*.\*Green.\*${AND}.\*Blue.\*", **new** FaultError("0002","GB"));  lookup.put("(003)\*.\*Blue.\*", **new** FaultError("0003","BLUE"));    *assertEquals*(lookup.get("Green").getCode(), "0001");  *assertEquals*(lookup.get("Blue Green").getCode(), "0002");  *assertEquals*(lookup.get("Blue with Live of pure").getCode(), "0003"); |

Complex Regular Expression (And/Not)

By default, regular expressions do not have an easy way to chain expressions together using AND/NOT logic. The OR logical expression is supported with the character “|”.

The RELookup operation combines regular expressions with a special syntax to support AND/NOT logic.

AND Operation

The RELookup supports chaining expressions together with “AND” logic. This is accomplished by chaining expressions together with “${AND}”. The string “${AND}” can be used to separate two regular expressions. If any of the regular expressions return false then the entire regular expression is false. In the following example, the regular expression “.\*USA.\*${AND}.\*Greece.\*”, only returns true if the text contains both “USA” and “Greece”.

|  |  |  |
| --- | --- | --- |
| Complex Regular Expression | Value | Matches |
| .\*USA.\*${AND}.\*Greece.\* | USA and Greece | TRUE |
| .\*USA.\*${AND}.\*Greece.\* | USA | FALSE |
| .\*USA.\*${AND}.\*Greece.\* | Greece | False |
| .\*USA.\*${AND}.\*Greece.\* | Greece USA | TRUE |

NOT Operation

The RELookup supports negative logic (NOT) for expressions. This is accomplished by prefixing the expressions with “${NOT}”. In the following example, the regular expression ${NOT}.\*USA only returns true if the text does not contain the word “USA”. Note that multiple “${NOT}”(s) can be chained together with “${AND}”(s) (see table below).

|  |  |  |
| --- | --- | --- |
| Complex RE | Value | Matches |
| ${NOT}.\*USA.\* | USA and Greece | False |
| ${NOT}.\*USA.\* | USA | False |
| ${NOT}.\*USA.\* | Greece | True |
| ${NOT}.\*USA.\* | Greece USA | False |
| .\*Greece.\*${AND}${NOT}.\*USA.\*${AND}${NOT}.\*Turkey.\* | Greece Turkey | False |
| .\*Greece.\*${AND}${NOT}.\*USA.\* ${AND}${NOT}.\*Turkey.\* | Greece Africa | True |