Arjuna CLF 2.0

Basic Introduction

CLF-R-08/09/09

Legal Notices

The information contained in this documentation is subject to change without notice.

Arjuna Technologies Limited makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Arjuna Technologies Limited shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Java™ and J2EE is a U.S. trademark of Sun Microsystems, Inc. Microsoft® and Windows NT® are registered trademarks of Microsoft Corporation. Oracle® is a registered U.S. trademark and Oracle9™, Oracle9 Server™ Oracle9 Enterprise Edition™ are trademarks of Oracle Corporation. Unix is used here as a generic term covering all versions of the UNIX® operating system. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Arjuna is a trademark of Hewlett-Packard Company.

Software Version

Arjuna CLF 2.0

Restricted Rights Legend

Use, duplication, or disclosure is subject to restrictions as set forth in contract subdivision (c)(1)(ii) of the Rights in Technical Data and Computer Software clause 52.227-FAR14.

Arjuna Technologies Limited
Nanotechnology Centre
Herschel Building
Newcastle Upon Tyne
NE1 7RU
United Kingdom

Content

Table Of Contents

About This Guide

What This Guide Contains

The Basic Introduction discusses the rationale behind logging and introduces the solutions taken by Arjuna CLF 2.0.

Audience

This guide is most relevant to **unexperienced** programmers who are responsible for using Arjuna CLF 2.0 installations in order to gain a basic understanding of the rationale and concepts behind logging. For a Quick start it's better to read the Programmer's Guide instead.

Organization

This guide contains the following chapters:

1. **Chapter 1, Introduction**
2. **Chapter 2, The Value of Logging**
3. **Chapter 3, Internationalisation**

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
Italic	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
`Code`	Text that represents programming code.
Function | Function	A path to a function or dialog box within an interface. For example, "**Select File | Open**." indicates that you should select the Open function from the File menu.
( ) and |	Parentheses enclose optional items in command syntax. The vertical

	bar separates syntax items in a list of choices. For example, any of the following three items can be entered in this syntax: `persistPolicy (Never | OnTimer | OnUpdate | NoMoreOftenThan)`
Note: and **Caution**:	A note highlights important supplemental information. A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Table 1 Formatting Conventions

Introduction

Introduction

Scope

This document describes the Arjuna Common Logging Framework providing an API to log messages with their importance and to hide the underlying logging implementations. That is it provides portability on top of existing implementations allowing moving an application without changing its source code.

The document is organized as follow; the section 2 describes the concept of internationalisation. Internationalisation is abbreviated as i18n, because there are 18 letters between the first "i" and the last "n". Section 3 gives an overview of the Logging concept, then the logging service considered by the Common Logging Framework are briefly described. Finally the section 4

References

References	Description
CSF Log	http://usage.fc.hp.com/partners/bluestone/common_javadoc/index.html
JDK 1.4	http://java.sun.com/j2se/1.4/docs/guide/util/logging/index.html
Log4j	http://jakarta.apache.org/log4j/docs/index.html

The Value of Logging

The Value of Logging

One can say that using a debugger may help to verify the execution of an application. However, in addition to the fact that a debugger decreases performance of an application, it is difficult to use it in a distributed computing environment.

This most basic form of logging involves developers manually inserting code into their applications to display small (or large) pieces of internal state information to help understand what's going on. It's a useful technique that every developer has used at least once. The problem is that it doesn't scale. Using print statements for a small program is fine, but for a large, commercial-grade piece of software there is far too much labor involved in manually adding and removing logging statements.

C programmers know, of course, that the way to conditionally add and remove code is via the C preprocessor and the #ifdef directive. Unfortunately, Java doesn't have a preprocessor. How can we make logging scale to a useful level in Java?

A simple way to provide logging in your program is to use the Java compiler's ability to evaluate boolean expressions at compile time, provided that all the arguments are known. For example, in this code, the println statements will not be executed if DEBUG not set to true.

```
class foo {
   public bar() {
     if(DEBUG) {
      System.out.println("Debugging enabled.");
      }
  }
}
```

A much better way, and the way that most logging is done in environments where the logged output is important, is to use a logging class.

A logging class collects all the messages in one central place and not only records them, but can also sort and filter them so that you don't have to see every message being generated. A logging class provides more information than just the message. It can automatically add information such as the time the event occurred, the thread that generated the message, and a stack trace of where the message was generated.

Some logging classes will write their output directly to the screen or a file. More advanced logging systems may instead open a socket to allow the log messages to be sent to a separate process, which is in turn responsible for passing those messages to the user or storing them.

The advantage with this system is that it allows for messages from multiple sources to be aggregated in a single location and it allows for monitoring remote systems.

The format of the log being generated should be customisable. This could start from just allowing setting the Log "level" - which means that each log message is assigned a severity level and only messages of greater importance than the log level are logged - to allowing more flexible log file formatting by using some sort LogFormatter objects that do transformations on the logging information.

The logging service should be able to route logging information to different locations based on the type of the information. Examples might be printing certain messages to the console, writing to a flat file, to a number of different flat files, to a database and so on. Examples of different types information could be for example errors, access information etc.

Features

An appropriate logging library should provide these features

- Control over which logging statements are enabled or disabled,
- Define importance or severity for logging statement via a set of levels
- Manage output destinations,
- Manage output format.
- Manage internationalisation (i18n)
- Configuration

Relevant Logging Framework

According to features (described above) a logging framework should provide, we have considering the most common logging service is use.

Overview of Log4j

Categories, Appenders, and Layout

Log4j has three main components:

- Categories
- Appenders
- Layouts

Category Hierarchy

The `org.log4j.Category` class figures at the core of the package. Categories are named entities. In a naming scheme familiar to Java developers, a category is said to be a parent of another category if its name, followed by a dot, is a prefix of the child category name. For example, the category named `com.foo` is a parent of the category named `com.foo.Bar`. Similarly, `java` is a parent of `java.util` and an ancestor of `java.util.Vector`.

The root category, residing at the top of the category hierarchy, is exceptional in two ways:

- It always exists
- It cannot be retrieved by name

In the `Category` class, invoking the static `getRoot()` method retrieves the root category. The static `getInstance()` method instantiates all other categories. `getInstance()` takes the name of the desired category as a parameter. Some of the basic methods in the `Category` class are listed below:

```
package org.log4j;

public Category class {
   // Creation & retrieval methods:
   public static Category getRoot();
   public static Category getInstance(String name);
   // printing methods:
   public void debug(String message);
   public void info(String message);
   public void warn(String message);
   public void error(String message);
   // generic printing method:
   public void log(Priority p, String message);
}
```

Categories *may* be assigned priorities from the set defined by the `org.log4j.Priority` class. Five priorities are defined: FATAL, ERROR, WARN, INFO and DEBUG, listed in decreasing order of priority. New priorities may be defined by subclassing the `Priority` class.

- FATAL: The `FATAL` priority designates very severe error events that will presumably lead the application to abort.
- ERROR: The `ERROR` priority designates error events that might still allow the application to continue running.
- WARN: The `WARN` priority designates potentially harmful situations.
- INFO: The `INFO` priority designates informational messages that highlight the progress of the application.
- DEBUG: The `DEBUG` priority designates fine-grained informational events that are most useful to debug an application.

To make logging requests, invoke one of the printing methods of a category instance. Those printing methods are: `fatal()`, `error()`, `warn()`, `info()`, `debug()`, `log()`.

By definition, the printing method determines the priority of a logging request. For example, if `c` is a category instance, then the statement `c.info("..")` is a logging request of priority INFO.

A logging request is said to be *enabled* if its priority is higher than or equal to the priority of its category. Otherwise, the request is said to be *disabled.* A category without an assigned priority will inherit one from the hierarchy.

Appenders and layouts

Log4j also allows logging requests to print to multiple output destinations called *appenders* in log4j speak. Currently, appenders exist for the console, files, GUI components, remote socket servers, NT Event Loggers, and remote UNIX Syslog daemons.

A category may refer to multiple appenders. Each enabled logging request for a given category will be forwarded to all the appenders in that category as well as the appenders higher in the hierarchy. In other words, appenders are inherited additively from the category hierarchy. For example, if you add a console appender to the root category, all enabled logging requests will at least print on the console. If, in addition, a file appender is added to a category, say *C*, then enabled logging requests for *C* and *C*'s children will print on a file and on the console.

More often than not, users want to customize not only the output destination but also the output format, a feat accomplished by associating a *layout* with an appender. The layout formats the logging request according to the user's wishes, whereas an appender takes care of sending the formatted output to its destination.

For example, the `PatternLayout` with the conversion pattern `%r [%t]%-5p %c - %m%n` will output something like:

```
176 [main] INFO  org.foo.Bar – Hello World.
```

In the output above:

- The first field equals the number of milliseconds elapsed since the start of the program
- The second field indicates the thread making the log request
- The third field represents the priority of the log statement
- The fourth field equals the name of the category associated with the log request

The text after the `-` indicates the statement's message.

Configuration

The log4j environment can be fully configured programmatically. However, it is far more flexible to configure log4j by using configuration files. Currently, configuration files can be written in XML or in Java properties (key=value) format.

Interactions

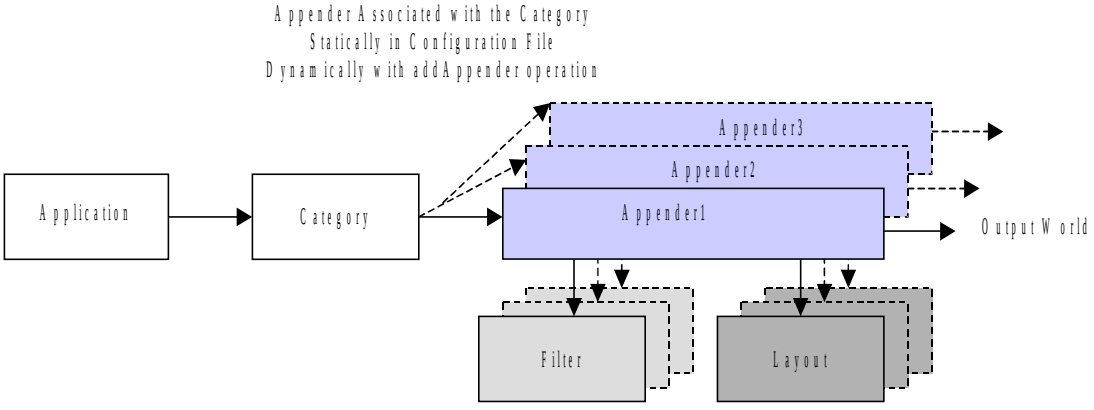


Figure 0-1 Basic Interactions within log4j

The following figure summarizes the different components when using log4j. Applications make logging calls on Category objects. The Category forwards to Appender logging requests for publication. Appender are registered with a Category with the addAppender method on the Category class. Invoking the addAppender method is made either by the Application or by Configurator objects. Log4j provides Configurator such BasicConfigurator, which registers to the category the ConsoleAppender responsible to send logging requests to the console, or the PropertyConfigurator, which registers Appender objects based on Appender classes defined in a configuration file. Both Category and Appender may use logging Priority and (optionally) Filters to decide if they are interested in a particular logging request. An Appender can use a Layout to localize and format the message before publishing it to the output world.

HP Logging Mechanism

The HP Logging Mechanism consists of a log handler, zero or more log writers, and one or more log channels, as illustrated in Figure below.

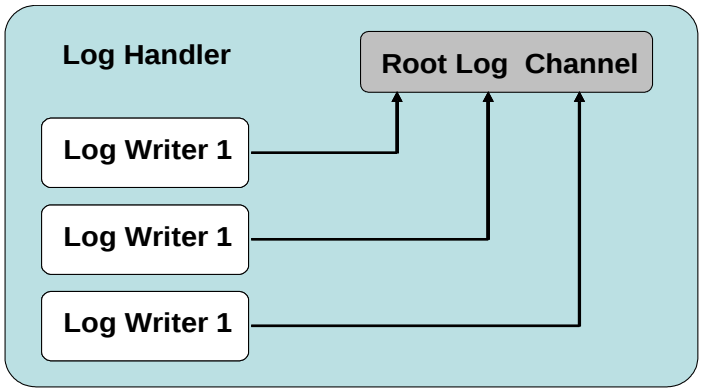


Figure 0-2 xxx

Log Handler

The log handler is implemented as a singleton Java Bean. It is accessible from the `com.hp.mw.common.util.LogHandlerFactory` which returns the single instance of `com.hp.mw.common.util.LogHandler`.

The following code illustrates how to obtain the `LogHandler`:

```
LogHandler handler;
handler = LogHandlerFactory.getHandler();
```

Log Channel

Log channels are virtual destinations; they receive messages and pass them to the log writers that are registered to receive them. They are not aware of the message formatting that might occur and are not aware of the logging tools that are used to view or store the messages. Log writers are registered for channels. When a log channel receives a message, and if that channel has a registered log writer(s), the message is passed along to that writer.

A client may obtain a channel with a specific name as follows.

```
LogChannel channel;
channel = LogChannelFactory.getChannel( "myapplication" );
```

Log Writers

In order to abstract the destination of a log message (e.g., console, file, database), the Logging Mechanism relies on log writers. Log writers are defined by the `com.hp.mw.common.util.logging.LogWriter` interface and are given messages by the channel(s) they service. They are responsible for formatting messages and outputting to the actual destination.

Log Formatters

A log formatter is responsible for formatting a log message into a Java String. Since many log writers do not require the String representation, log formatters are not required for every log writer. As a result, the `com.hp.mw.common.util.logging.LogMessageFormat` interface would be used for formatting messages into Strings when applicable and necessary.

Log Levels and Thresholds

All log channels are created, initially, with a default log threshold. The threshold is the minimum severity of a log message that should be processed for that log channel. The log levels defined by the HP logging mechanisms are as follows:

Log Level Description

- LOG_LEVEL_NONE This log level should be used to turn off all messages to a channel.
- LOG_LEVEL_FLOW Flow messages indicate program flow and can be extremely frequent.
- LOG_LEVEL_DEBUG Debug messages are fairly low-level messages that provide the developer(s) with information about events occurring within the application
- LOG_LEVEL_INFO Informational messages are of higher severity than debug and should provide information that any user could understand, as opposed to debug messages, which provide code-specific information.
- LOG_LEVEL_WARNING Warning messages are typically used to report an unusual or unexpected occurrence from which recovery is possible (e.g., a missing or incorrect configuration value that has a reasonable default).
- LOG_LEVEL_ERROR Error messages are used to report an unusual or unexpected occurrence from which recovery is not possible. This does not indicate that the entire application or framework is incapable of continuing, but that the component involved might be defunct or the operation it was asked to perform is aborted.
- LOG_LEVEL_CRITICAL Critical messages are typically used to report a very unusual or unexpected occurrence. For example, a component that was functioning correctly but suddenly experiences an unrecoverable error that prevents it from continuing should emit a critical message.

Interactions

The following figure summarizes the different components when using log4j. Applications make logging calls on `Channel` objects. The Channel forwards to `LogWriter` logging requests for publication. `LogWriter` are registered with the handler associated to a Channel. Both `LogChannel` and `LogWritter` may use logging `LogLevel` to decide if they are interested in a particular logging request. A LogWriter can use a *LogFormatter* to format the message before publishing it to the output world.

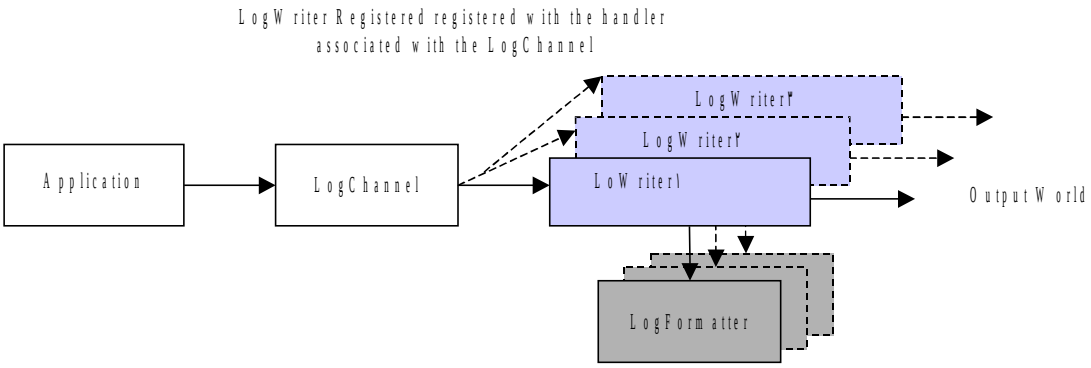


Figure 0-3 Basic interactions within the HP logging Mechanism

Internationalisation

Internationalisation

An application is internationalized, if it can correctly handle different encodings of character data. An application is localized, if it formats and interprets data (dates, times, timezones, currencies, messages and so on) according to rules specific to the user's locale (country and language).

Internationalization (I18N) is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. *Localization* (L10N) is the use of locale-specific language and constructs at run time.

The Java Internationalization API

Java Internationalization shows how to write software that is multi-lingual, using Unicode, a standard system that supports hundreds of character sets.

The Java Internationalization API is a comprehensive set of APIs for creating multilingual applications. The JDK internationalization features, from its version 1.1, include:

- Classes for storing and loading language-specific objects.
- Services for formatting messages, date, times, and numbers.
- Services for comparing and collating text.
- Support for finding character, word, and sentence boundaries.
- Support for display, input, and output of Unicode characters.

Java Interfaces for Internationalization

Users of the Java internationalization interfaces should be familiar with the following interfaces included in the Java Developer's Kit (JDK):

- java.util.Locale
  Represents a specific geographical, political, or cultural region.
- java.util.ResourceBundle
  Containers for locale-specific objects
- java.text.MessageFormat
  A means to produce concatenated messages in a language-neutral way.

Set the Locale

The concept of a *Locale* object, which identifies a specific cultural region, includes information about the country or region. If a class varies its behavior according to `Locale`, it is said to be *locale-sensitive*. For example, the `NumberFormat` class is locale-sensitive; the format of the number it returns depends on the `Locale`. Thus `NumberFormat` may return a number as 902 300 (France), or 902.300 (Germany), or 902,300 (United States). `Locale` objects are only identifiers.

Most operating systems allow to indicate their locale or to modify it. For instance Windows NT does this through the control panel, under the Regional Option icon. In Java, you can get the *Locale* object that matches the user's control-panel setting using *myLocale = Locale.getDefault();*. You can also create *Locale* objects for specific places by indicating the language and country you want, such as *myLocale = new Locale("fr", "CA");* for "Canadian French."

The next example creates `Locale` objects for the English language in the United States and Great Britain:

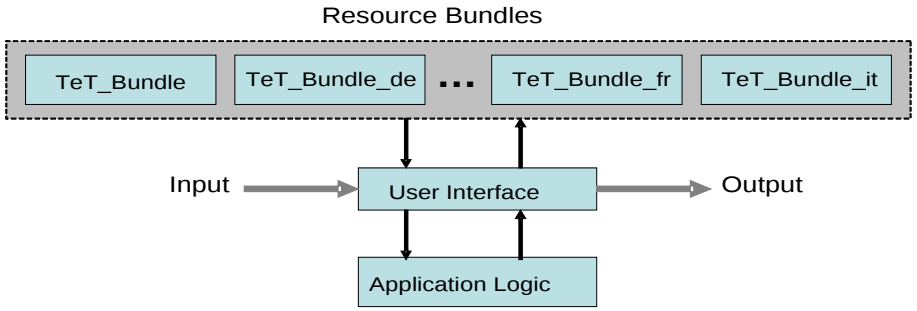
```
bLocale = new Locale("en", "US");

cLocale = new Locale("en", "GB");
```

The strings you pass to the *Locale* constructor are two-letter language and country codes, as defined by ISO standards (put here reference)

Isolate your Locale Data

The first step in making an international Java program is to isolate all elements of your Java code that will need to change in another country. This includes user-interface text -- label text, menu items, shortcut keys, messages, and the like.

The *ResourceBundle* class is an abstract class that provides an easy way to organize and retrieve locale-specific strings or other resources. It stores these resources in an external file, along with a key that you use to retrieve the information. You'll create a *ResourceBundle* for each locale your Java program supports.



The `ResourceBundle` class is an abstract class in the `java.util` package. You can provide your own subclass of `ResourceBundle` or use one of the subclass implementations, as in the case of `PropertyResourceBundle` or `ListResourceBundle`.

Resource bundles inherit from the `ResourceBundle` class and contain localized elements that are stored external to an application. Resource bundles share a base name. The base name `TeT_Bundle`, to display transactional messages such “Transaction Commited”, might be selected because of the resources it contains. Locale information further differentiates a resource bundle. For example, `TeT_Bundle_it` means that this resource bundle contains locale-specific transactional messages for Italian.

To select the appropriate `ResourceBundle`, invoke the `ResourceBundle.getBundle` method. The following example selects the `TeT_Bundle ResourceBundle` for the `Locale` that matches the French language, the country of Canada.

```
Locale currentLocale = new Locale("fr", "CA");
ResourceBundle introLabels =
    ResourceBundle.getBundle("TeT_Bundle", currentLocale);
```

Java loads your resources based on the locale argument to the `getBundle` method. It searches for matching files with various suffixes, based on the language, country, and any variant or dialect to try to find the best match. Java tries to find a complete match first, and then works its way down to the base filename as a last resort.

You should always supply a base resource bundle with no suffixes, so that your program will still work if the user's locale does not match any of the resource bundles you supply. The default file can contain the U.S. English strings. Then you should provide properties files for each additional language you want to support.

Basically, a resource bundle is a container for key/value pairs. The key is used to identify a locale-specific resource in a bundle. If that key is found in a particular resource bundle, its value is returned.

The jdk API defines two kinds of `ResourceBundle` subclasses -- the `PropertyResourceBundle` and `ListResourceBundle`.

A `PropertyResourceBundle` is backed by a properties file. A properties file is a plain-text file that contains translatable text. Properties files are not part of the Java source code, and they can contain values for `String` objects only. A simple default properties file, named `hpts_Bundle.properties`, for messages sent by HPTS could be.

```
# Sample properties file for demonstrating PropertyResourceBundle
# Text to inform on transaction outcomes in English (by default)
trans_committed= Transaction Committed
trans_rolledback= Transaction Rolled Back
# …
```

The equivalent properties file, `hpts_Bundle_fr_FR.properties,` for French would be:

```
# Sample properties file for demonstrating PropertyResourceBundle
# Text to inform on transaction outcomes in French
trans_committed = La Transaction a été Validée
trans_rolledback = La Transaction a été Abandonnée
# …
```

Example

The following example illustrates how to use the internationalization API allowing separating the text with a language specified by the user, from the source code.

```
import java.util.*;
import Demo.*;
import java.io.*;
import com.arjuna.OrbCommon.*;
import com.arjuna.CosTransactions.*;
import org.omg.CosTransactions.*;
import org.omg.*;

public class TransDemoClient
{
  public static void main(String[] args)
  {
    String language;
    String country;
    if (args.length != 2) {
     language = new String("en");
     country = new String("US");
    } else {
     language = new String(args[0]);
     country = new String(args[1]);
    }
    Locale currentLocale;
    ResourceBundle messages;
    currentLocale = new Locale(language, country);
    trans_message = ResourceBundle.getBundle(
                    "hpts_Bundle", currentLocale);

    try
    {
       ORBInterface.initORB(args, null);
  OAInterface.initOA();
  String         ref  = new String();
  BufferedReader file = new BufferedReader(new
                    FileReader("DemoObjReference.tmp"));
  ref = file.readLine();
  file.close();
  org.omg.CORBA.Object obj = ORBInterface.orb().string_to_object(ref);
       DemoInterface  d   = (DemoInterface)
                            DemoInterfaceHelper.narrow(obj);

       OTS.get_current().begin();
  d.work();
  OTS.get_current().commit(true);
     System.out.println(tran_message.getString("trans_committed"));
     }
     catch (Exception e)
     {
       System.out.println(tran_message.getString("trans_rolledback"));
     }
  }
}
```

In the following example the language code is `fr` (French) and the country code is `FR` (France), so the program displays the messages in French:

```
% java TransDemoClient fr FR
La Transaction a été validée
```

Rather to specify explicitly the language to be used to display messages, a property variable can be defined in a properties files (such TransactionService-2.2.properties).

Index