O'REILLY® ONJava.com™ The Independent Source for Enterprise Java

I18N Messages and Logging

by John Mazzitelli
12/06/2006

For many a software developer, the mere mention of a requirement to support internationalization (aka *i18n*) is sure to elicit groans. Writing code that is targeted towards an international audience does require some forethought, because it is not something very easily introduced into an existing piece of software. If you feel there is even a small possibility that you will need your software to support different locales and languages, it is always more prudent to internationalize your project at the start, rather than attempting to retrofit i18n into it after it has begun.

That said, what does "internationalizing" mean? It is more than just providing translations of your user interface messages into different languages. It involves dealing with different character encodings, localizing date, time, and currency formats, and other things that differ across multiple regions around the world.

Introducing i18nlog

While this article will not attempt to discuss all the facets of internationalization, it will cover how to more easily perform some of the tasks necessary to introduce i18n functionality by examining a new open source project called *I18N Messages and Logging*, or i18nlog for short.

i18nlog allows you to incorporate internationalized messages into your Java applications by providing an API to:

- Annotate Java classes to identify your i18n messages.
- Obtain i18n messages from resource bundles in any supported locale.
- Create localized exceptions whose messages are internationalized.
- Log i18n messages using any logging framework.
- Automatically generate resource bundles in any supported locale.
- Automatically generate help/reference documentation.

Defining i18n Messages

One of the more tedious tasks involved when internationalizing your software is maintaining resource bundles. Resource bundles are properties files that contain "name=value" pairs where "name" is the resource bundle key string and "value" is the localized message string itself. It is customary to store messages in resource bundles such that there is one resource bundle per language and each bundle has an identical set of keys with their associated messages each translated into their locale's language. The name of the resource bundle file dictates which locale it is for; for example, *mybundle_en.properties* denotes the messages in that file are written in English, where *mybundle_de.properties* contain German messages.

i18nlog provides annotations to define your resource bundle messages and their key strings--used in conjunction with i18nlog's custom Ant task, you can automatically generate your resource bundles and not have to do much to ensure consistency between your properties files and the Java code that accesses them.

The @I18NMessage annotation is placed on constants that you use in place of your resource bundle key strings. Using these constants inherently forces compile-time checks; typos introduced in the code (i.e., misspelling a constant name) and usage of obsolete/deleted messages are detected at compile time. An example usage of this annotation is:

```
@I18NMessage( "Hello, {0}. You last visited on {1,date}" )
public static final String MSG_WELCOME = "example.welcome-msg";
```

The above defines one i18n message. The value of the constant defines the resource bundle key string. The value of the annotation is a translation of the actual message. You can place these annotated constants in any class or interface within your application. You can put all of them in a single class or interface (to centralize all of your message definitions in a single location), or you can place these constants in the classes where they are used.

The @I18NResourceBundle annotation defines the resource bundle properties file where your messages are to be placed. This can annotate an entire class or interface, or it can annotate a specific field. If you annotate a class or interface, all @I18NMessage annotations found in that class or interface will be stored, by default, in that resource bundle. If the annotation is on a particular constant field, it will be the bundle for that constant only. An example usage of this annotation would be:

```
@I18NResourceBundle( baseName = "messages",
                     defaultLocale = "en" )
```

which indicates that all associated @I18NMessage-annotated messages will be placed in a resource bundle file named *messages_en.properties*. A more complete example that illustrates an interface that contains a set of i18n messages is shown below:

```
@I18NResourceBundle( baseName = "messages",
                     defaultLocale = "en" )
public interface Messages {
   @I18NMessage( "Hello, {0}. You last visited on {1,date}" )
   String MSG_WELCOME = "welcome-msg";

   @I18NMessage( "An error occurred, please try again" )
   String MSG_ERR = "error-occurred";

   @I18NMessage( "The value is {0}" )
   String MSG_VALUE = "value";
}
```

Retrieving i18n Messages

Now that you have defined your i18n constants, you can use the API provided by the core class in i18nlog: mazz.i18n.Msg. It allows you to load messages from resource bundle properties files. It will also replace the messages' placeholders (e.g. {0}, {1,date}) with the values passed as variable argument parameters. Although the API frees you from having to work directly with some JDK classes, you might want to read the Javadocs on java.text.MessageFormat to get a feel for how i18nlog does what it does, specifically how it replaces the placeholders with localized data.

`Example usages of the Msg class are as follows:`

```
// using a static factory method to create a Msg object
// this assumes the default bundle base name of "messages"
System.out.println( Msg.createMsg( Messages.MSG_WELCOME,
                                   name,
                                   date ) );
```

and

```
// using a constructor to create a Msg object
Msg msg = new Msg( new Msg.BundleBaseName("messages") );
try {
   String hello = msg.getMsg(Messages.MSG_WELCOME, name, date );
   ... do something ...
}
catch (Exception e) {
   throw new RuntimeException( msg.getMsg( Messages.MSG_ERR ) );
}
```

The `name` and `date` arguments are an example of passing an arbitrary variable arguments list--each object represents the value to replace the placeholder in its respective position ({0} and {1,date} respectively, based on the `Messages.MSG_WELCOME` definition given earlier).

The `Msg` object will know which locale's resource bundle to use based on its "bundle base name" and its current locale setting. The "bundle base name" is the name of the bundle file minus the locale specifier and extension (in the example above, the bundle base name is `messages`). You can define the bundle base name the `Msg` object will use by passing it into the `Msg` constructor or one of its static factory methods (if left unspecified, the default is `messages`). While the bundle base name is fixed for the lifetime of the Msg object, you can switch the locale used by the `Msg` object by calling its `setLocale()` method (its default is the JVM's default locale). This is useful if you pool `Msg` objects and have to switch its locale based on the user to which the object is currently assigned.

Localized Exceptions

i18nlog provides two base exception classes (for both checked and unchecked exceptions--LocalizedException and LocalizedRuntimeException) that can be used to create your own subclasses of localized exceptions. These have constructors whose signatures are very similar to the Msg class. They simply allow you to specify your exception message via a resource bundle key and a variable argument list of placeholder values, with the ability to optionally specify the bundle base name and locale. This allows your exception messages to be localized to different languages, just as Msg can retrieve localized messages.

Logging i18n Messages

i18nlog provides a means by which you can log i18n messages. The main class of the logging subsystem is mazz.i18n.Logger. It provides the typical set of trace, debug, info, warn, error, and fatal methods. However, instead of taking a string consisting of the message itself, you pass in the resource bundle key and a variable arguments list for the placeholder values that are to be replaced within the message. It uses the mazz.i18n.Msg class under the covers to get the actual localized message.

You obtain Logger objects by using the factory class mazz.i18n.LoggerFactory in a way that is basically the same as in log4j and the like. But you log messages in a way that is very similar to obtaining messages via the mazz.i18n.Msg object:

```
public static final mazz.i18n.Logger LOG =
               mazz.i18n.LoggerFactory.getLogger(MyClass.class);
...
LOG.debug(Messages.MSG_VALUE, value);
...
try {
   ...
}
catch (Exception e) {
   LOG.warn(e, Messages.MSG_ERR);
}
```

If a log level is not enabled, no resource bundle lookups are performed and no string concatenation is done--effectively making log calls very fast under these conditions. If a log message is to be associated with a particular exception, pass the exception as the first argument to the log method. This will allow the stack trace to be dumped with the message if stack dumps are enabled (see below).

There are additional features that i18nlog adds to its logging framework that go above what the underlying, third-party, logging framework provides. The first is the ability to tell Logger whether or not to dump stack traces of exceptions. You may or may not care to see all the exception stack traces during a particular run. Note that this feature cannot turn off stack dumps for exceptions logged at the FATAL level--fatal exceptions logged with that method always have their stack traces dumped. For all other log levels, the loggers will be told to dump stack traces if the i18nlog.dump-stack-traces system property is set to true (or you programmatically called Logger.setDumpStackTraces(true)).

The second additional feature in the i18nlog logging framework is the ability to log a message's associated resource bundle key along with the message itself. The resource bundle key is the same across all locales--so no matter what language the log messages are in, the keys will always be the same. You can think of these keys as "message IDs" or "error codes." This is very useful if you generate help documentation that references these codes with additional help text for your users to consult that help them decipher what the messages are trying to convey (see below for how to generate this type of documentation). This feature is enabled by default; to disable, set the system property i18nlog.dump-keys to false or programmatically call Logger.setDumpLogKeys(false).

Providing these message IDs and avoiding string concatenation for disabled log levels may be enough to justify using this logging mechanism provided by i18nlog. But some may still argue that internationalizing your log messages (as opposed to just user-interface messages) is overkill and generally not very useful. I, myself, find it hard to argue with this point of view sometimes. You certainly do not have to use i18n logging if your project doesn't warrant it. The other features provided by i18nlog are, of course, still available even if you choose not to use its logging capabilities.

However, I can envision certain cases where i18n logging can be useful. Note that i18nlog makes it possible to define a different locale that the loggers will use (the "log locale"), as compared to the locale Msg instances will use. This is to facilitate the use case where I want to log messages in a language my support group can read, but my user interface is in a language that my users can read (which may be different). For example, my users may be German-speaking, but the software is supported by a group that works in France and is only French-speaking. In this case, when my German users have a problem, they will normally send the logs to the support group in France, so the software could, by default, set its log locale to Locale.FRENCH. On the other hand, if my German users want to try to debug a problem themselves, having the log files contain messages in French isn't helpful to them. In this case, my German users can simply set a system property and have the log messages appear in German. Refer to the mazz.i18n.LoggerLocale Javadoc for more information on how to switch the log locale.

Resource Bundle Auto-Generation

i18nlog provides an Ant task that automatically generates resource bundle properties files so the developer isn't responsible for manually adding new messages to them and manually cleaning up old, obsolete messages that are no longer used.

The Ant task scans your classes looking for @I18N annotations and, based on them, will automatically create your resource bundles for you. This means that as you add more @I18NMessage-annotated fields, they will automatically be added to your resource bundles. If you delete an i18n message constant, that message will be removed from the resulting resource bundle that the Ant task generates. To run the Ant task, you need to have something like the following in your Ant build script:

```
<taskdef name="i18n"
        classpathref="i18nlog-jar.classpath"
        classname="mazz.i18n.ant.I18NAntTask" />

<i18n outputdir="${classes.dir}" verify="true" verbose="true">
  <classpath refid="my.classpath" />
  <classfileset dir="${classes.dir}"/>
</i18n>
```

You must give the Ant task a classpath that can find your I18N-annotated classes and their dependencies (<classpath>) and you have to give a set of class files to the Ant task that contain the list of files that are to be scanned for I18N annotations (<classfileset>). It is recommended that you use the verbose mode the first time you use the Ant task so you can see what its doing. Once you get the build the way you want it, you can turn off verbose mode. After this task executes, your resource bundle properties files will exist in the specified output directory.

Generating Help Documentation

One optional feature you can use with this Ant task is the ability to generate help documentation that consists of a reference of all your resource bundle key names with their messages, along with some additional description of what the message means. There is an optional attribute you can specify in your @I18NMessage annotations--the `help` attribute. Its value can be any string that further describes the message. Think of this as documentation that further describes what situation occurred that the message is trying to convey. The auto-generated help documents can, therefore, provide a cross-reference between the message keys, the messages themselves, and more helpful descriptions of the messages:

```
@I18NMessage( value="The value is {0}",
              help="This will show you the value of your"
                  +" current counter. If this value is over"
                  +" 1000, you should reset it.")
String MSG_VALUE = "value";

@I18NMessage( value="Memory has {0} free bytes left",
              help="The VM is very low on memory. Increase -Xmx"
String MSG_LOW_MEM = "low-memory";
```

Many times, you can use this as a "message code" or "error code" listing, where each of your resource bundle keys can be considered a "message code" or "error code." To generate help documentation, you need to use the <helpdoc> inner tag inside of the <i18n> task:

```
<i18n outputdir="${classes.dir}">
   <classpath refid="my.classpath" />
   <classfileset dir="${classes.dir}"/>
   <helpdoc outputdir="${doc.dir}/help"/>
</i18n>
```

For every resource bundle generated, you will get an additional help document output in the given directory specified in <helpdoc>. The document is generated based on templates that describe what the documents should look like. By default, a simple HTML page with a <table> of message codes and their help documentation is output. There are additional attributes you can specify in the <helpdoc> tag that allow you to define a custom template, should you wish to define your own help documentation look and feel. The Javadocs of the mazz.i18n.ant.Helpdoc class have more information on this.

Once the help documentation generation is complete, you will end up with a document (or documents) containing a list of all your message key codes, their messages, and any "help" attribute text that is defined.

Localization

A lot of what we have discussed deals with how you can internationalize your software by enabling it to obtain translated and localized messages. Once you have enabled your software, it is now a manual process by which your resource bundle properties files (those generated by the `<i18n>` Ant task or those you manually created yourself) have to be localized. You must ensure that all resource bundle messages are translated into languages that you intend to support and the data those messages will contain are localized. A lot of the localization can be done by defining your placeholders properly (i.e. `{0,date}` will output the date string using the locale's localized format and language). But suffice it to say, obtaining the services of a good translation and localization company is a must.

Summary

This article showed you how you can incorporate i18n capabilities into your application using a new open source project, `i18nlog`. This project provides tools and an API that automatically manages your resource bundle files, retrieves and localizes messages from those bundles, and can even generate help documentation for your end users.

Resources

- `i18nlog` user's guide: i18nlog.sourceforge.net/doc/users-guide.html
- `i18nlog` API: i18nlog.sourceforge.net/api
- Placeholder syntax to localize messages: java.sun.com/j2se/1.5.0/docs/api/java/text/MessageFormat.html

John Mazzitelli is a developer for JBoss, a division of Red Hat, currently focusing on the implementation of the JBoss Operations Network management platform.