PENNY ROHR CURICH

ADVANCED JMETER TESTING

Writing Custom Components for Apache JMeter 5.0

Table of Contents

Table of Contents

Introduction

About this Book

About the Author

What is JMeter?

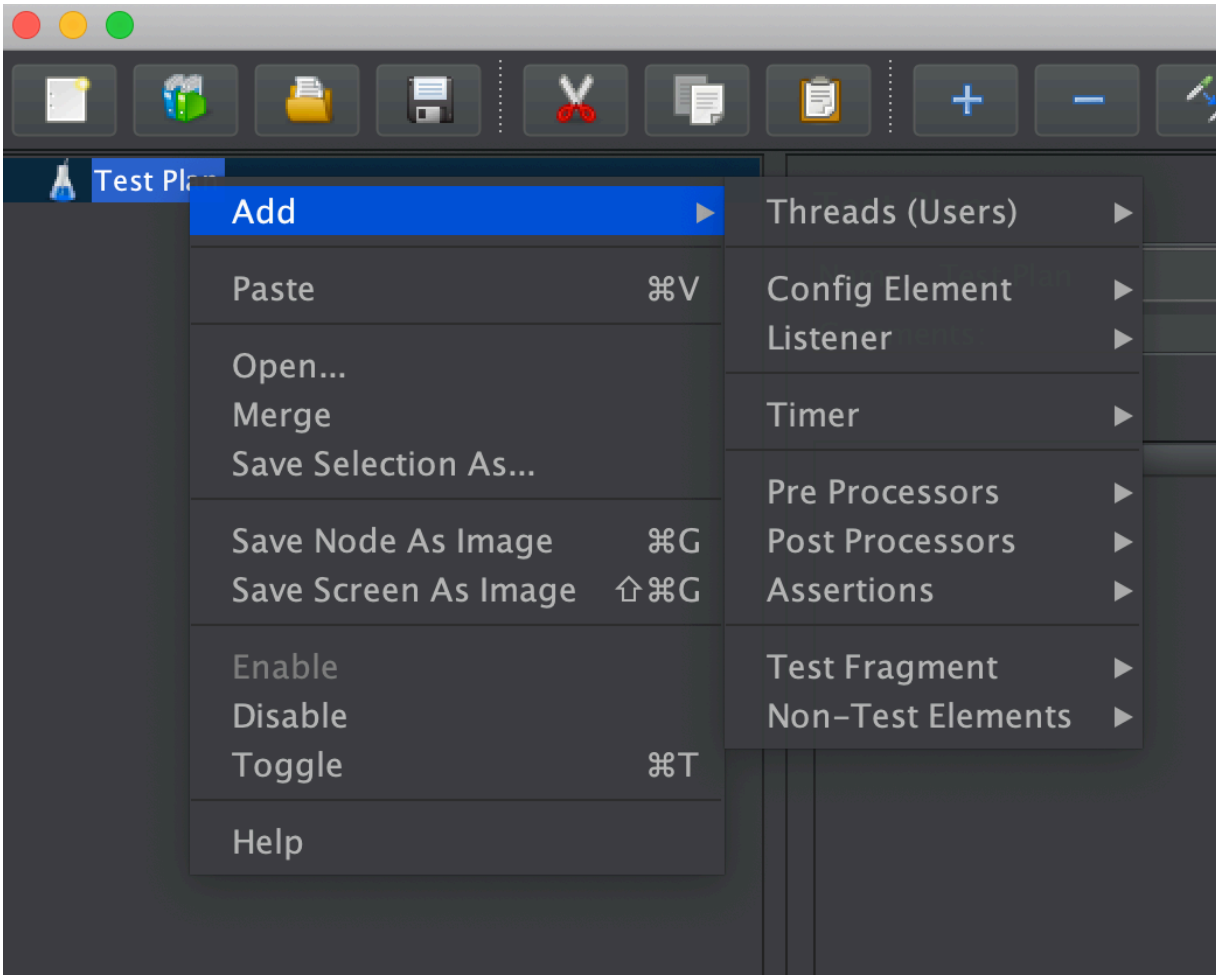
Test Components

Test Element Types

A JMeter test plan is a tree structure of varying test elements implementing interface `org.apache.jmeter.testelement.TestElement`.

Test elements generally implement a subinterface of TestElement (`org.apache.jmeter.control.Controller`, `org.apache.jmeter.samplers.Sampler`) or TestElement and another interface such as `org.apache.jmeter.processor.PreProcessor`.

Available categories of test elements are visible when adding to a test plan through the GUI.



Note that it is possible for a test element to appear under multiple categories when multiple interfaces are implemented by that element.

For more information, reference JMeter's documentation Elements of a Test Plan at https://jmeter.apache.org/usermanual/test_plan.html and the Component Reference at https://jmeter.apache.org/usermanual/component_reference.html.

PreProcessor

Anastasia Golokova succinctly explains the purpose of a PreProcessor in a BlazeMeter article.

> PreProcessors are JMeter elements that are used to execute actions before the sampler requests are executed in the test scenario. PreProcessors can be used for different performance testing needs, like fetching data from a database, setting a timeout between sampler execution or before test data generation.[1]

PreProcessor API documentation is available at https://jmeter.apache.org/api/org/apache/jmeter/processor/PreProcessor.html.

When to Create a PreProcessor

While a PreProcessor is useful for providing objects to a Sampler, because they can be defined outside the scope of a thread group at the plan level, they can also be used to manage suite setup and tear down activities by implementing interface `org.apache.jmeter.testelement.TestStateListener` and other lifecycle interfaces.

Incorporating Spring Framework

Spring is a very powerful tool used in conjunction with custom JMeter components. Complex concerns such as scoped variables, lifecycle management, aspect-oriented code, etc. may be offloaded to objects managed by a Spring application context. The customized JMeter component is then free to reference these objects and concentrate on the business of JMeter execution.

---

[1] BlazeMeter blog entry A Quick Guide to JMeter PreProcessors, March 13 2017. https://www.blazemeter.com/blog/quick-guide-jmeter-preprocessors

Imagine an existing functional test suite that already incorporates Spring Framework, implementing browser-based testing via Selenium. By loading the functional suite in JMeter, it is possible to leverage the existing code and execute it repeatedly and in parallel as a load test with minimal additional code.

How-To

A Spring application context can be managed by a PreProcessor at the test-plan level before any dependent thread groups are defined.

The Martini-JMeter bridge library contains a class that does precisely this: `guru.qas.martini.jmeter.preprocessor.SpringPreProcessor`.

The configuration accepts optional environment variables and requires at least one Spring configuration XML file present on the classpath.

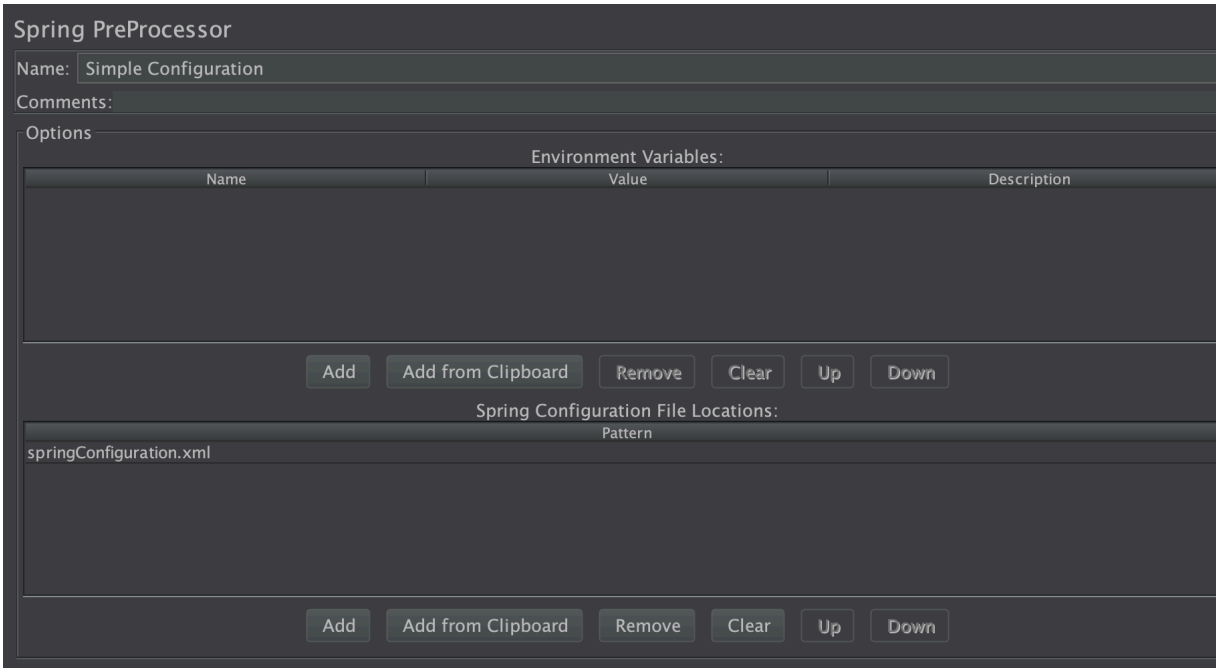
On test start, a Spring ClassPathXmlApplicationContext is loaded, provided with environment variables, registered with a shutdown hook and set as a ThreadContext variable referenced by constant `guru.qas.martini.jmeter.preprocessor.SpringPreProcessor.THREAD_CONTEXT_VARIABLE` for use by other elements implementing `org.apache.jmeter.testelement.TestStateListener`.

If the Spring application context is unable to be started for any reason, an error message is produced and the test suite is immediately halted. Note that more than one configured and active Spring PreProcessor is considered an error condition, and the suite will be halted.

During test execution, the application context is again exposed to test elements as a thread variable set in method `org.apache.jmeter.testlement.TestIterationListener.testIterationStart(LoopIterationEvent)`. It is also exposed to samplers via the sampler context map referenced by constant

`guru.qas.martini.jmeter.preprocessor.SpringPreProcessor.SAMPLER_CONTEXT_KEY` set in the process() method.

Upon test suite or normal JVM exit, the PreProcessor gracefully closes the Spring application context.



Creating Custom Components

Internationalization (i18n)

When software may be used in more than one locale, it is far easier to build internationalized messaging into the initial release than to attempt a retrofit at a later time.

Fortunately, JMeter's UI uses the JavaBeans API [2] in conjunction with Java internationalization [3] support. JMeter also ships with Simple Logging Facade for Java (slf4j) [4] which can leverage the Compiler Assisted Localization library (ch.qos.cal10n) [5] for internationalized logging.

---

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/java/beans/package-summary.html

[3] https://openjdk.java.net/groups/i18n/#i18n

[4] https://www.slf4j.org

[5] http://cal10n.qos.ch

JMeter GUI

There are two general approaches to building a JMeter component's GUI interface:

1. Create Java Swing classes.
2. Implement `org.apache.jmeter.testbeans.TestBean` and extend `org.apache.jmeter.testbeans.BeanInfoSupport`.

I strongly recommend approach number two for a variety of reasons including code clarity, maintainability and forward compatibility.

Implementation Steps

There are three steps to creating a JMeter component with internationalization support using the TestBean interface.

Step One: Implement TestBean

After determining which JMeter component interface [6] to implement, create a new package to house the component.

Supposing a company name of acme.com and the need for a customized JMeter controller, then a fully-qualified class name of `com.acme.jmeter.controller.CustomController` might be appropriate.

Generally speaking, components should at a minimum also be declared to implement and provide implementations of `java.io.Serializable`, `java.lang.Cloneable` and `org.apache.jmeter.testbeans.TestBean`.

[6] https://jmeter.apache.org/usermanual/component_reference.html

Components also generally extend `org.apache.jmeter.testelement.AbstractTestElement` or one of its extensible subclasses, such as `org.apache.jmeter.control.GenericController`.

Logging should be performed using slf4j [7] implementations.

Any member variables that are to be accessible via JMeter's UI should follow the JavaBeans API contract for introspection.

The class will need to have a no-argument constructor, and should call its superclass constructor.

A simple implementation might look like the following.

```
package com.acme.jmeter.controller;

import java.io.Serializable;

import org.apache.jmeter.control.GenericController;
import org.apache.jmeter.samplers.Sampler;
import org.apache.jmeter.testbeans.TestBean;

public class CustomController extends GenericController
   implements Serializable, Cloneable, TestBean {

   private static final long serialVersionUID = 389177611519492768L;

   private String myVariable;

   public String getMyVariable() {
      return myVariable;
   }

   public void setMyVariable(String s) {
      this.myVariable = s;
   }
```

[7] https://www.slf4j.org

```
    public CustomController() {
       super();
    }

    @Override
    public Sampler next() {
       return isSomeCondition() ? super.next() : null;
    }

    private boolean isSomeCondition() {
       return true;
    }
}
```

Step Two: Extend BeanInfoSupport

In the same package location as the new component class, create another class with the same name, appending BeanInfo to the class name.

The class should be public, should extend `org.apache.jmeter.testbeans.BeanInfoSupport` and should call its superclass with the implementation of the TestBean class supported.

The constructor should also define `java.beans.PropertyDescriptor` instances for each member variable accessed through the UI. *Note that the name of the property descriptor exactly matches the name of the base bean introspection method in the supported class.*

Expanding on the previous TestBean example, we might implement the following BeanInfo class.

```
package com.acme.jmeter.controller;

import java.beans.PropertyDescriptor;

import org.apache.jmeter.testbeans.BeanInfoSupport;

public class CustomControllerBeanInfo extends BeanInfoSupport {
```

```
    public CustomControllerBeanInfo() {
       super(CustomController.class);

       PropertyDescriptor p = property("myVariable");
       p.setValue(NOT_UNDEFINED, Boolean.TRUE);
       p.setValue(DEFAULT, "my default value");
    }
}
```

Step Three: Create Message Resource Bundle

Create a new resource directory with the same naming convention as the custom component. At a minimum, create a default .properties resource file.

Assuming the TestBean developed in the last couple sections, a Maven project structure would look like the following. *Note the matching capitalization on the .properties filename, with "Resources" appended to the classname.*

```
/src
   /main
      /resources
         /com
            /acme
               /jmeter
                     /controller
                           /CustomControllerResources.properties
```

The properties file should contain key `displayName` for UI rendering of the component name, and each introspected bean property should be present as a key ending in `.displayName.`

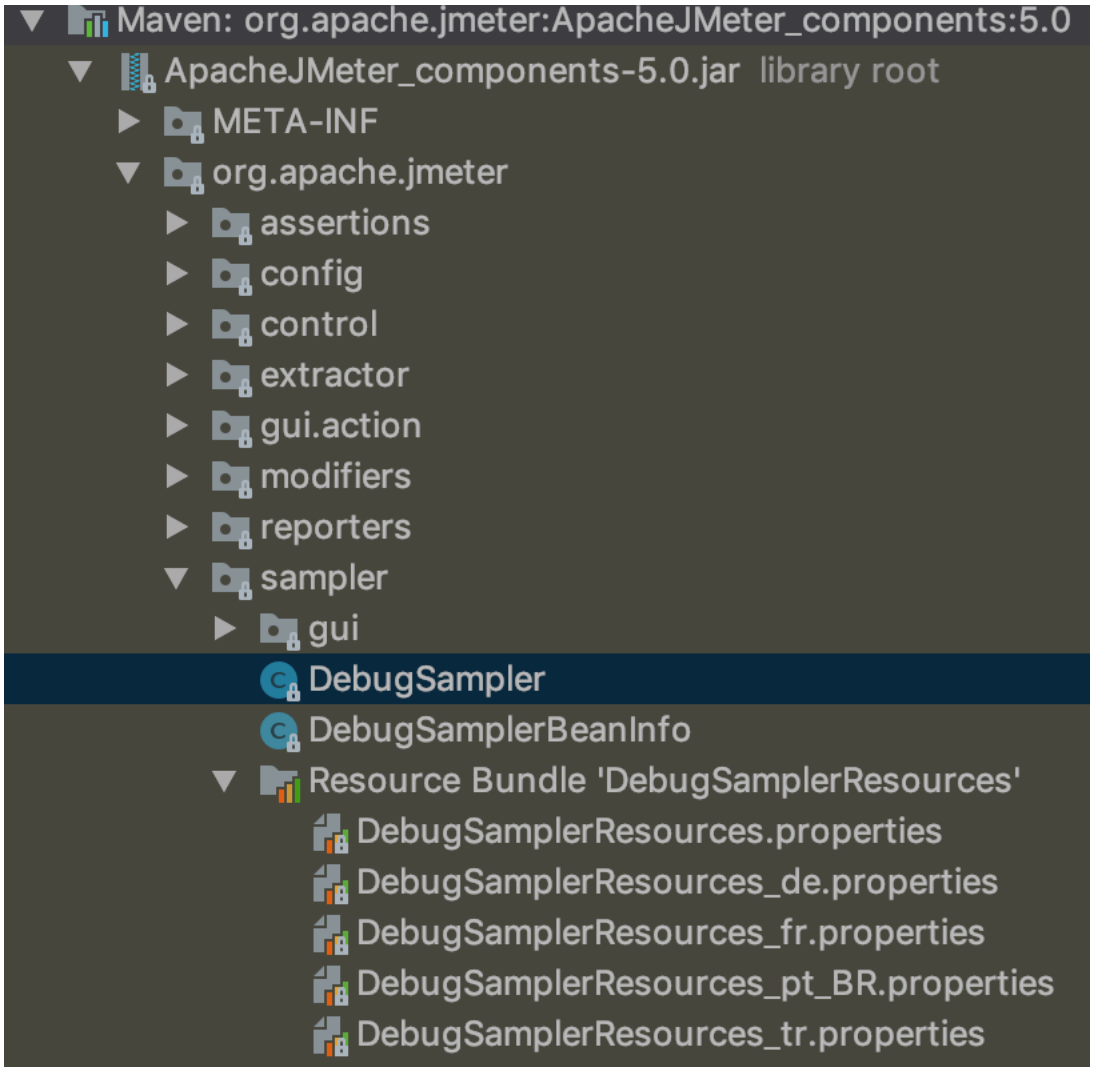
A reasonable implementation of CustomControllerResources.properties might look like the following.

```
displayName=Acme Custom Controller
myVariable.displayName=My Configurable Variable
```

Once the project artifact is built and included in JMeter's classpath, the results of internationalization efforts are visible through the JMeter UI.

DebugSampler Example

For an example of internationalization support, examine the DebugSampler classes in package `org.apache.jmeter.sampler`.



Examining class `org.apache.jmeter.sampler.DebugSampler`, we can see that it extends class `org.apache.jmeter.testbeans.TestBean` and contains private variables that are managed by bean introspection methods, e.g. private boolean displayJMeterVariables is retrieved via method `public boolean isDisplayJMeterVariables() and set via method` public void setDisplayJMeterVariables(boolean).

```java
/**
 * The Debug Sampler can be used to "sample" JMeter variables, JMeter properties and System Propertie
 */
@GUIMenuSortOrder(2)
public class DebugSampler extends AbstractSampler implements TestBean {

    private static final long serialVersionUID = 232L;

    private static final Set<String> APPLIABLE_CONFIG_CLASSES = new HashSet<>(
            Arrays.asList("org.apache.jmeter.config.gui.SimpleConfigGui"));

    private boolean displayJMeterVariables;
    private boolean displayJMeterProperties;
    private boolean displaySystemProperties;

    @Override
```

```java
    public boolean isDisplayJMeterVariables() { return displayJMeterVariables; }

    public void setDisplayJMeterVariables(boolean displayJMeterVariables) {
        this.displayJMeterVariables = displayJMeterVariables;
    }
```

Examining sibling class `org.apache.jmeter.sampler.DebugSamplerBeanInfo`, we can see that it extends class `org.apache.jmeter.testbeans.BeanInfoSupport` and provides property descriptors for the variables in class DebugSampler. *Note that the property descriptors precisely match the naming used in class DebugSampler.*

```java
public class DebugSamplerBeanInfo extends BeanInfoSupport {
    public DebugSamplerBeanInfo() {
        super(DebugSampler.class);
        PropertyDescriptor p;

        p = property("displayJMeterVariables");
        p.setValue(NOT_UNDEFINED, Boolean.TRUE);
        p.setValue(NOT_EXPRESSION, Boolean.TRUE);
        p.setValue(NOT_OTHER, Boolean.TRUE);
        p.setValue(DEFAULT, Boolean.TRUE);

        p = property("displayJMeterProperties");
```

We can also see that several internationalization message resources have been created using ISO 639 and ISO 3166 codes to support DebugSampler in various languages and locales [8]. *Note that each resource bundle matches the name of the DebugSampler class and ends in Resources with appended locale information.*

[8] https://www.oracle.com/technetwork/java/javase/documentation/java11locales-5069639.html

- Default: `DebugSamplerResources.properties`
- German: `DebugSamplerResources_de.properties`
- French: `DebugSamplerResources_fr.properties`
- Portugese, `Brazilian locale: DebugSamplerResources_pt_BR.properties`
- Turkish: `DebugSamplerResources_tr.properties`

JMeter's language and locale preferences are set by the end-user by system properties `user.language` and `user.region` and/or JMeter properties `language` and `locales.add` as documented in the User Manual [9].

If no language or locale is specified, default DebugSamplerResources.properties will be used to render messages, e.g. `displayName=Debug Sampler` and `displayJMeterVariables.displayName=JMeterVariables`.

The default will also be utilized when the language or locale specified is not supported.

Note that the property keys for variables exactly match the variable name with .displayName appended.

```
displayName=DebugSampler
displayJMeterVariables.dispayName=JMeter variables
…
```

[9] http://jmeter.apache.org/usermanual/properties_reference.html#language

Supposing JMeter's language preference has been changed to Canadian French by editing JMeter's jmeter.properties setting.

```
36 #Preferred GUI language. Comment out to use the JVM default locale's language.
37 language=fr
38
39
40 # Additional locale(s) to add to the displayed list.
41 # The current default list is: en, fr, de, no, es, tr, ja, zh_CN, zh_TW, pl, pt_BR
42 # [see JMeterMenuBar#makeLanguageMenu()]
43 # The entries are a comma-separated list of language names
44 locales.add=CA,FR
```

DebugSampler messages will then be rendered in French with display names configured in `DebugSamplerResources_fr.properties`.

```
dispayName=Enchatillon D\u00e9bogate
dispayJMeterVariables.displayName=Variables JMeter
…
```

Missing Message Resources

Imagine JMeter's developers add support specifically for US English in file DebugSamplerResources_en_US.properties, but accidentally left out a variable display.

```
displayName=US English Debug Sampler
#displayJMeterVariables.displayName=forgotten JMeter variables
displayJMeterProperties.dispayName=JMeter properties
…
```

In this case, with language preference set to en and locale to US, the default file will be utilized to display the default message.

Now suppose JMeter's developers remove entry `displayJMeterVariables.displayName` from default `DebugSamplerResources.properties`. With language preference set to en, the actual variable name will be exposed in messages.

If JMeter developers remove the message resource bundle altogether, class and variable names are exposed. *The same behavior will occur if there are variable, class name or property file name mismatches.*

Additionally, JMeter will log an error indicating that internationalization could not be performed.

```
   [AWT-EventQueue-0] Localized strings not available for bean class
org.apache.jmeter.sampler.DebugSampler
   java.util.MissingResourceException: Can't find bundle for base name
org.apache.jmeter.sampler.DebugSamplerResources, locale en
```

```
     at
java.util.ResourceBundle.throwMissingResourceException(ResourceBundle.
java:2055) ~[?:?]
     at java.util.ResourceBundle.getBundleImpl(ResourceBundle.java:
1689) ~[?:?]
     at java.util.ResourceBundle.getBundleImpl(ResourceBundle.java:
1593) ~[?:?]
```

Logging with slf4j and cal10n

The Compiler Assisted Localization Library (cal10n)[10] provides an implementation of Simple Logging Facade (slf4j)[11] library class `org.slfj4.Logger` that can be used to internationalize application logging.

An excellent manual is available for the Compiler Assisted Localization Library at http://cal10n.qos.ch/manual.html.

Note that the Compiler Assisted Localization Library is distributed under the MIT License. Reference https://github.com/qos-ch/cal10n/blob/master/LICENSE.txt.

Step 1: Obtain Locale

JMeter class `org.apache.jmeter.util.JMeterUtils` provides a configurable locale that is statically accessible.

```
Locale locale = JMeterUtils.getLocale();
```

Step 2: Instantiate MessageConveyor

Use the locale to instantiate an instance of `ch.qos.cal10n.MessageConveyor`.

```
MessageConveyor = new MessageConveyor(locale);
```

[10] http://cal10n.qos.ch
[11] https://www.slf4j.org

Step 3: Instantiate LocLoggerFactory

Use the MessageConveyor to instantiate a new instance of class `org.slf4j.cal10n.LocLoggerFactory`.

```
LocLoggerFactory loggerFactory =
   new LocLoggerFactory(messageConveyor);
```

Step 4: Instantiate LocLogger

Use the LocLoggerFactory to instantiate a new instance of class `org.slf4j.cal10n.LocLogger`.

```
LocLogger logger = loggerFactory.getLocLogger(this.getClass());
```

Step 5: Create Message Resources

Reviewing the JavaDocs for LocLogger at https://www.slf4j.org/api/org/slf4j/cal10n/LocLogger.html, we can see that c10n expects an Enum argument.

Method and Description
`debug(Enum<?> key, Object... args)` Log a localized message at the DEBUG level.

This method expects a Java enum implementation with configuration annotations `ch.qos.cal10n.BaseName` and `ch.qos.cal10n.LocaleData` at the class level.

Example Implementation

Following is an example class.

```
package com.acme.jmeter.controller.CustomControllerMessages;
```

```
import ch.qos.cal10n.BaseName;
import ch.qos.cal10n.Locale;
import ch.qos.cal10n.LocaleData;

@BaseName(“com.acme.jmeter.controller.customControllerMessages”)
@LocaleData({@Locale("en_US")})
public enum CustomControllerMessages {
   STARTING,
   UNEXPECTED_ERROR,
   MULTIPLE_INTERPOLATION
}
```

In order to use this enum as an argument to a cal10n LocLogger method, it will be necessary to create a resources directory as defined by the value of the `@BaseName` annotation.

Using a Maven structure:

```
/src
   /main
      /resources
         /com
            /acme
               /jmeter
                  /controller
```

It will also be necessary to create a new file in this directory: `customControllerMessages_en_US.properties` required by the `@LocaleData` annotation.

Note that the first locale specified in the @LocaleData annotation will be considered the default. If for example the user's locale language is French but no @Locale(“fr”) has been specified, then US English will be utilized for message rendering.

If `@LocaleData` resources are not present at compile-time, compilation will fail; this is the aim of Compiler Assisted Localization. Removing the `@LocaleData`

annotation will also result in a failure.

The `.properties files must` contain message resource values for enum keys: `STARTING`, `UNEXPECTED_ERROR` and `MULTIPLE_INTERPOLATINO`. *If any values are missing or other values are specified in the .properties files that aren't referenced by the matching enum class, cal10n will produce a compile-time error.*

Following is an example of a valid .properties file. *Note that Message Resource Bundle interpolations are supported.*

```
STARTING=Starting Custom Controller…
UNEXPECTED_ERROR=encountered unexpected error: {0}
MULTIPLE_INTERPOLATION=see how fancy I am: {1} {0} {1}
```

Step 6: Utilize LocLogger

Utilizing LocLogger is somewhat different than the use of normal slf4j logging.

Note that all of the parameters for LocLogger methods are the same: `.xxx(Enum<?> enum, Object… varargs)`. Unlike slf4j, LocLogger does not expect to be passed a Throwable as a separate argument and will treat the Throwable as a normal vararg for message interpolation.

A call without interpolation may look like this:

```
locLogger.info(CustomControllerMessages.STARTING);
```

A call with multiple, re-used interpolation varargs might look like this:

```
locLogger.warn(
   CustomControllerMessages.MULTIPLE_INTERPOLATION, “zero”, “one”);
```

Logging an exception may look like this:

```
String cause = e.getCause();
locLogger.info(CustomControllerMessages.UNEXPECTED_ERROR, cause);
```

Or like this, utilizing Guava's com.google.common.base.Throwables:

```
String stacktrace = Throwables.getStackTraceAsString(e);
locLogger.info(
   CustomControllerMessages.UNEXPECTED_ERROR, '\n' + stacktrace);
```

Or like this, eventually calling Exception's toString() method.

```
locLogger.info(CustomControllerMessages.UNEXPECTED_ERROR, e);
```

Using cal10n Directly

If localization of messages is desired independently of logging, it is possible to use cal10n directly.

Follow steps one, two and five in previous section Logging with slf4j and cal10n to obtain a MessageConveyor object.

Next, call method `MessageConveyor.getMessage(Enum<?> e, Object… varargs)` to obtain a localized String, similar to step six in previous section Logging with slf4j and cal10n.