

How to Write a plugin for JMeter

Table of Contents

How to write a plugin for JMeter

Introduction from Peter Lin

On more than one occasion, users have complained JMeter's developer documentation is out of date and not very useful. In an effort to make it easier for developers, I decided to write a simple step-by-step tutorial. When I mentioned this to mike, he had some ideas about what the tutorial should cover.

Before we dive into the tutorial, I'd like to say writing a plugin isn't necessarily easy, even for someone with several years of java experience. The first extension I wrote for JMeter was a simple utility to parse HTTP access logs and produce requests in XML format. It wasn't really a plugin, since it was a stand alone command line utility. My first real plugin for JMeter was the webservice sampler. I was working on a .NET project and needed to stress test a webservice. Most of the commercial tools out there for testing .NET webservices suck and cost too much. Rather than fork over several hundred dollars for a lame testing tool, or a couple thousand dollars for a good one, I decided it was easier and cheaper to write a plugin for JMeter.

After a two weeks of coding on my free time, I had a working prototype using Apache Soap driver. I submitted it back to JMeter and mike asked me if I want to be a committer. I had contributed patches to Jakarta JSTL and tomcat in the past, so I considered it an honor. Since then, I've written the access log sampler, Tomcat 5 monitor and distribution graph. Mike has since then improved the access log sampler tremendously and made it much more useful.

Introduction from Mike Stover

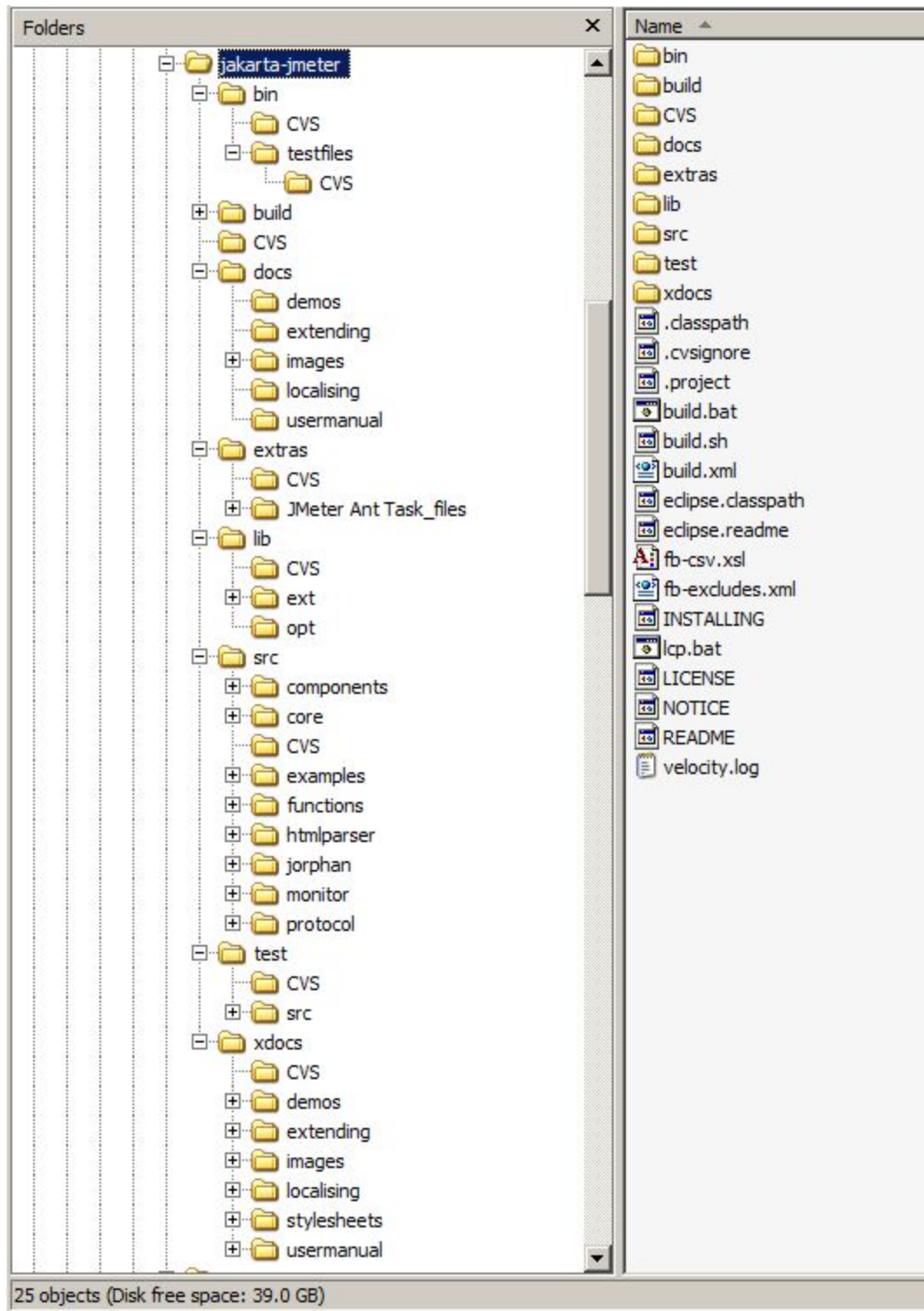
One of my primary goals in designing JMeter was to make it easy to write plugins to enhance as many of JMeter's features as possible. Part of the benefit of being open-source is that a lot of people could potentially lend their efforts to improve the application. I made a conscious decision to sacrifice some simplicity in the code to make plugin writing a way of life for a JMeter developer.

While some folks have successfully dug straight into the code and made improvements to JMeter, a real tutorial on how to do this has been lacking. I tried a long time ago to write some documentation about it, but most people did not find it useful. Hopefully, with Peter's help, this attempt will be more successful.

Basic structure of JMeter

JMeter is organized by protocols and functionality. This is done so that developers can build new jars for a single protocol without having to build the entire application. We'll go into the details of building JMeter later in the tutorial. Since most of the jmeter developers use eclipse, the article will use eclipse directory as a reference point.

Root directory - /eclipse/workspace/jakarta-jmeter/



jakarta-jmeter

- bin – contains the .bat and .sh files for starting JMeter
- docs – directory contains the JMeter documentation files
- extras – ant related extra files
- lib – contains the required jar files for JMeter
- src – contains subdirectory for each protocol and component
- test – unit test related directory
- xdocs – Xml files for documentation. JMeter generates documentation from Xml.

As the tutorial progresses, an explanation of the subdirectories will be provided. For now, let's focus on "src" directory. From the screen capture, we see the following directories.

Src

- components – contains non-protocol-specific components like visualizers, assertions, etc..
- core – the core code of JMeter including all core interfaces and abstract classes.
- examples – example sampler demonstrating how to use the new bean framework
- functions – standard functions used by all components
- htmlparser – a snapshot of HtmlParser, donated by HtmlParser project on sourceforge
- jorphan – utility classes providing common utility functions
- monitor – tomcat 5 monitor components
- protocol – contains the different protocols JMeter supports

Within "protocol" directory, are the protocol specific components.

Protocol

- ftp – components for load testing ftp servers
- http – components for load testing web servers
- java – components for load testing java components
- jdbc – components for load testing database servers using jdbc
- jndi – components for load testing jndi
- ldap – components for load testing LDAP servers
- mail – components for load testing mail servers
- tcp – components for load testing TCP services

As a general rule, all samplers related to HTTP will reside in "http" directory. The exception to the rule is the Tomcat5 monitor. It is separate, because the functionality of the monitor is slightly different than stress or functional testing. It may eventually be reorganized, but for now it is in its own directory. In terms of difficulty, writing visualizers is probably one of the harder plugins to write.

Writing a Visualizer

Of the component types, visualizers require greater depth in Swing than something like controllers, functions or samplers. You can find the full source for the distribution graph in "components/org/apache/jmeter/visualizers/". The distribution graph visualizer is divided into two classes.

- DistributionGraphVisualizer – visualizer which Jmeter instantiates
- DistributionGraph – Jcomponent which draws the actual graph

The easiest way to write a visualizer is to do the following:

1. extend org.apache.jmeter.visualizers.gui.AbstractVisualizer
2. implement any additional interfaces need for call back and event notification. For example, the

DistributionGraphVisualizer implements the following interfaces:

1. ImageVisualizer
2. ItemListener – according to the comments in the class, ItemListener is out of date and isn't used anymore.
3. GraphListener
4. Clearable

AbstractVisualizer provides some common functionality, which most visualizers like “graph results” use. The common functionality provided by the abstract class includes:

- configure test elements – This means it create a new ResultCollector, sets the file and sets the error log
- create the stock menu
- update the test element when changes are made
- create a file panel for the log file
- create the title panel

In some cases, you may not want to display the menu for the file textbox. In that case, you can override the “init()” method. Here is the implementation for DistributionGraphVisualizer.

```
1  /**
2   * Initialize the GUI.
3   */
4   private void init()
5   {
6       this.setLayout(new BorderLayout());
7
8       // MAIN PANEL
9       Border margin = new EmptyBorder(10, 10, 5, 10);
10
11      this.setBorder(margin);
12
13      // Set up the graph with header, footer, Y axis and graph display
14      JPanel graphPanel = new JPanel(new BorderLayout());
15      graphPanel.add(createGraphPanel(), BorderLayout.CENTER);
16      graphPanel.add(createGraphInfoPanel(), BorderLayout.SOUTH);
17
18      // Add the main panel and the graph
19      this.add(makeTitlePanel(), BorderLayout.NORTH);
20      this.add(graphPanel, BorderLayout.CENTER);
21  }
```

The first thing the init method does is create a new BorderLayout. Depending on how you want to layout the widgets, you may want to use a different layout manager. Keep mind using different layout managers is for experts.

The second thing the init method does is create a border. If you want to increase or decrease the border, change the four integer values. Each integer value represents pixels. If you want your visualizer to have no border, skip lines 9 and 11. Line 15 calls “createGraphPanel,” which is responsible for configuring and adding the DistributionGraph to the visualizer.

```
1  private Component createGraphPanel()
2  {
3      graphPanel = new JPanel();
4      graphPanel.setBorder(BorderFactory.createBevelBorder(
5          BevelBorder.LOWERED, Color.lightGray, Color.darkGray));
6      graphPanel.add(graph);
7      graphPanel.setBackground(Color.white);
```

```

8   return graphPanel;
9 }

```

At line 6, the graph component is added to the graph panel. The constructor is where a new instance of `DistributionGraph` is created.

```

public DistributionGraphVisualizer()
{
    model = new SamplingStatCalculator("Distribution");
    graph = new DistributionGraph(model);
    graph.setBackground(Color.white);
    init();
}

```

The constructor of `DistributionGraphVisualizer` is responsible for creating the model and the graph. Every time a new result is complete, the engine passes the result to all the listeners by calling `add(SampleResult res)`. The visualizer passes the new `SampleResult` to the model.

```

1 public synchronized void add(SampleResult res)
2 {
3     model.addSample(res);
4     updateGui(model.getCurrentSample());
5 }

```

In the case of the `DistributionGraphVisualizer`, the “add” method doesn't actually update the graph. Instead, it calls “updateGui” in line four.

```

public synchronized void updateGui(Sample s)
{
    // We have received one more sample
    if (delay == counter){
        updateGui();
        counter = 0;
    } else {
        counter++;
    }
}

```

Unlike `GraphVisualizer`, the distribution graph attempts to show how the results clump; therefore the `DistributionGraphVisualizer` delays the update. The default delay is 10 samples.

```

1 public synchronized void updateGui()
2 {
3     if (graph.getWidth() < 10){
4         graph.setPreferredSize(new Dimension(getWidth() - 40, getHeight() - 160));
5     }
6     graphPanel.updateUI();
7     graph.repaint();
8 }

```

Lines 3 to 4 are suppose to resize the graph, if the user resizes the window or drags the divider. Line 6 updates the panel containing the graph. Line 7 triggers the update of the `DistributionGraph`. Before we cover writing graphs, there are couple of important methods visualizer must implement.

```

public String getLabelResource()
{
    return "distribution_graph_title";
}

```

```
}
```

The label resource retrieves the name of the visualizer from the properties file. The file is located in “core/org/apache/jmeter/resources”. It’s best not to hardcode the name of the visualizer. Message.properties file is organized alphabetically, so adding a new entry is easy.

```
public synchronized void clear()
{
    this.graph.clear();
    model.clear();
    repaint();
}
```

Every component in Jmeter should implement logic for “clear()” method. If this isn’t done, the component will not clear the UI or model when the user tries to clear the last results and run a new test. If clear is not implemented, it can result in a memory leak.

```
public JComponent getPrintableComponent(){
    return this.graphPanel;
}
```

The last method visualizers should implement is “getPrintableComponent()”. The method is responsible for returning the Jcomponent that can be saved or printed. This feature was recently added so that users can save a screen capture of any given visualizer.

GraphListener

Visualizers should implement GraphListener. This is done to make it simpler to add new Sample instances to listeners. As a general rule, if the a custom graph does not plot every single sample, it does not need to implement the interface.

```
public interface GraphListener
{
    public void updateGui(Sample s);
    public void updateGui();
}
```

The important method in the interface is “updateGui(Sample s)”. From DistributionGraphVisualizer, we see it calls graph.repaint() to refresh the graph. In most cases, the implementation of updateGui(Sample s) should do just that.

ItemListener

Visualizers generally do not need to implement this interface. The interface is used with combo boxes, checkbox and lists. If your visualizer uses one of these and needs to know when it has been updated, the visualizer will need to implement the interface. For an example of how to implement the interface, please look at GraphVisualizer.

Writing Custom Graphs

For those new to Swing and haven’t written custom Jcomponents yet, I would suggest getting a book on Swing and get a good feel for how Swing widgets work. This tutorial will not attempt to

explain basic Swing concepts and assumes the reader is already familiar with the Swing API and MVC (Model View Controller) design pattern. From the constructor of DistributionGraphVisualizer, we see a new instance of DistributionGraph is created with an instance of the model.

```
public DistributionGraph(SamplingStatCalculator model)
{
    this();
    setModel(model);
}
```

The implementation of “setModel” method is straight forward.

```
private void setModel(Object model)
{
    this.model = (SamplingStatCalculator) model;
    repaint();
}
```

Notice the method calls “repaint” after it sets the model. If “repaint” isn't called, it can cause the GUI to not draw the graph. Once the test starts, the graph would redraw, so calling “repaint” isn't critical.

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    final SamplingStatCalculator m = this.model;
    synchronized (m)
    {
        drawSample(m, g);
    }
}
```

The other important aspect of updating the widget is placing the call to drawSample within a synchronized block. If drawSample wasn't synchronized, Jmeter would throw a ConcurrentModificationException at runtime. Depending on the test plan, there may be a dozen or more threads adding results to the model. The synchronized block does not affect the accuracy of each individual request and time measurement, but it does affect Jmeter's ability to generate large loads. As the number of threads in a test plan increases, the likelihood a thread will have to wait until the graph is done redrawing before starting a new request increases. Here is the implementation of drawSample.

```
private void drawSample(SamplingStatCalculator model, Graphics g)
{
    width = getWidth();
    double height = (double)getHeight() - 1.0;

    // first lets draw the grid
    for (int y=0; y < 4; y++){
        int q1 = (int)(height - (height * 0.25 * y));
        g.setColor(Color.lightGray);
        g.drawLine(xborder,q1,width,q1);
        g.setColor(Color.black);
        g.drawString(String.valueOf((25 * y) + "%"),0,q1);
    }
    g.setColor(Color.black);
    // draw the X axis
    g.drawLine(xborder,(int)height,width,(int)height);
    // draw the Y axis
```



```

g.drawLine(xborder,0,xborder,(int)height);
// the test plan has to have more than 200 samples
// for it to generate half way decent distribution
// graph. the larger the sample, the better the
// results.
if (model != null && model.getCount() > 50){
    // now draw the bar chart
    Number ninety = model.getPercentPoint(0.90);
    Number fifty = model.getPercentPoint(0.50);

    total = model.getCount();
    Collection values = model.getDistribution().values();
    Object[] objval = new Object[values.size()];
    objval = values.toArray(objval);
    // we sort the objects
    Arrays.sort(objval,new NumberComparator());
    int len = objval.length;
    for (int count=0; count < len; count++){
        // calculate the height
        Number[] num = (Number[])objval[count];
        double iper = (double)num[1].intValue()/(double)total;
        double iheight = height * iper;
        // if the height is less than one, we set it
        // to one pixel
        if (iheight < 1){
            iheight = 1.0;
        }
        int ix = (count * 4) + xborder + 5;
        int dheight = (int)(height - iheight);
        g.setColor(Color.blue);
        g.drawLine(ix -1,(int)height,ix -1,dheight);
        g.drawLine(ix,(int)height,ix,dheight);
        g.setColor(Color.black);
        // draw a red line for 90% point
        if (num[0].longValue() == ninety.longValue()){
            g.setColor(Color.red);
            g.drawLine(ix,(int)height,ix,55);
            g.drawLine(ix,(int)35,ix,0);
            g.drawString("90%",ix - 30,20);
            g.drawString(String.valueOf(num[0].longValue()),ix + 8, 20);
        }
        // draw an orange line for 50% point
        if (num[0].longValue() == fifty.longValue()){
            g.setColor(Color.orange);
            g.drawLine(ix,(int)height,ix,30);
            g.drawString("50%",ix - 30,50);
            g.drawString(String.valueOf(num[0].longValue()),ix + 8, 50);
        }
    }
}
}
}

```

In general, the rendering of the graph should be fairly quick and shouldn't be a bottleneck. As a general rule, it is a good idea to profile custom plugins. The only way to make sure a visualizer isn't a bottleneck is to run it with a tool like Borland Optimizelt. A good way to test a plugin is to create a simple test plan and run it. The heap and garbage collection behavior should be regular and predictable.

Building JMeter

Like other Jakarta projects, Jmeter uses ANT to compile and build the distribution. Jmeter has several tasks defined, which make it easier for developers. For those unfamiliar with ANT, it's a build tool similar to make on Unix. A list of the ANT tasks with a short description is provided below.

All – builds all components and protocols
check-libs – checks which optional libraries are missing
check-anakia – checks anakia
compile – compiles all the directories and components
compile-core – compiles the core directory and all dependencies
compile-components – compiles the components directory and all dependencies
compile-ftp – compiles the samples in ftp directory and all dependencies
compile-functions – compiles the functions and all dependencies
compile-htmlparser – compiles htmlparser and all dependencies
compile-http – compiles the samplers in http directory and all dependencies
compile-java – compiles the samplers in java directory and all dependencies
compile-jdbc – compiles the samplers in jdbc directory and all dependencies
compile-jorphan – compiles the jorphan utility classes
compile-ldap – compiles the samplers in ldap directory and all dependencies
compile-monitor – compiles the sampler in monitor directory and all dependencies
compile-rmi – compiles the samplers in rmi directory and all dependencies
compile-tests – compiles the tests and all dependencies
docs-api – creates the javadocs
docs-all – generates all docs.
package – creates jar files of the compiled packages or protocols
package-only – creates jar files of the compiled components

Here are some example commands.

<i>Command</i>	<i>description</i>
Build compile-http	Compiles just http components
Build package	Creates the jar files
Build all-docs	Generates the html documentation and javadocs