Ingenieurbüro David Fischer AG | A Company of the Apica Group
http://www.apicasystem.com

ZebraTester

"Recorder Plug-In" Developer Handbook

 February 2, 2016

Table of Contents

1 Overview

1.1 Document Contents

This Handbook consists of two parts.

Part One provides an overview of the Properties, Runtime Behavior, and Configuration of ZebraTester “Recorder Plug-Ins”.

Part Two provides information on how to develop own ZebraTester “Recorder Plug-Ins”.

1.2 Introduction

ZebraTester “Recorder Plug-ins” are Extension Modules to the ZebraTester product and can be automatically executed during the recording of a Web surfing session. Furthermore, “Recorder Plug-ins” can also manually called from the Web Admin GUI or called from the Proxy Recorder REST API.

Such Recorder Plug-Ins have direct access to the recorded session data and also to the internal “Var Handler” which means that such Plug-Ins can modify HTTP Request and Response data before, during and after the recording, and that such Plug-Ins can also create variables, variable extractors and variable assigners. Furthermore such plug-ins can also reconfigure the HTTP response “content test” settings of URL calls.

The main purpose for using Recorder Plug-Ins is to automate some parts of the post-processing steps – after a session has been recorded. Another purpose is to modify HTTP request data during recording on the fly – just before they are sent to the Web server.

Recorder Plug-Ins have the major characteristic of being re-usable. Once developed, a Recorder Plug-In can be re-used and applied for every new recorded session.

Recorder Plug-Ins are supported starting from ZebraTester version 5.2-L.

2 Properties, Runtime Behavior and Configuration

2.1 Installation Directory

Once a Recorder-Plug in was developed its compiled Java class must be placed in the ZebraTester installation directory – at the same place where normally the file prxsniff.jar exists. This is only needed for machines on which the ZebraTester Web GUI is running (or on which the proxy recorder process is running). There is no need to copy such a Plug-In to a load generator.

After that ZebraTester should be triggered to perform at runtime a re-scan for Recorder Plug-Ins. Alternatively you can also restart ZebraTester.

2.2 Active/Inactive State and Manual Calls

The only thing that you can configure at runtime for a loaded Recorder Plug-In is its “active state”. The developer of a Plug-In can choose if the Plug-in is active or inactive by default – and you can normally change that state by using the Web GUI or the REST-API.

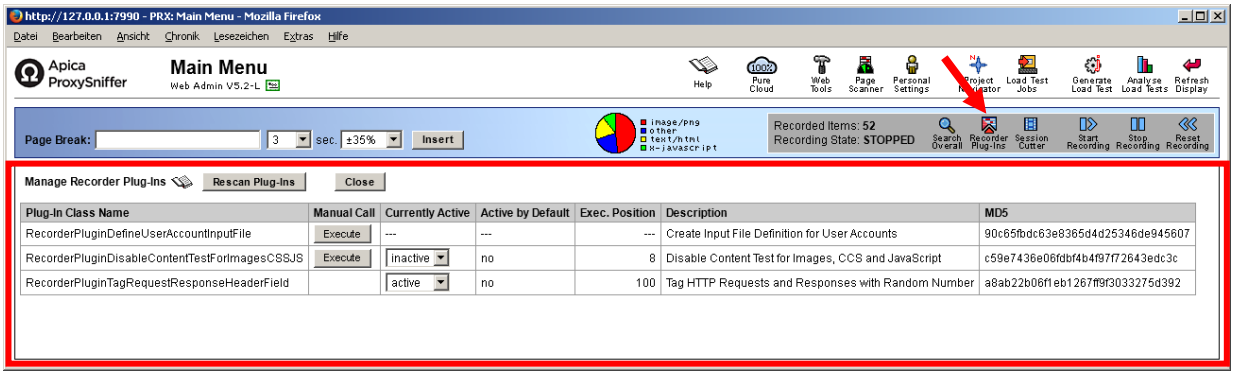
If the current state of a Plug-In is inactive it will not called while recording a Web session. However you can normally still call the Plug-In manually at every time – regardless if its state is inactive or active. So recorder Plug-Ins are often set by default to inactive and the automated post-processing tasks are then performed by a manual call.

As an option the programmer of a Plug-In can also prohibit that a Plug-In can become in active state. This is a common case and shows you that the Plug-In contains no functionality that is needed just-in-time during the recording of a session.

Also the opposite is possible: The programmer of a Plug-In can prohibit that a Plug-In can be manually called. This shows you that the Plug-In contains only just-in-time functionality.

2.3 Plug-in Configuration and Plug-In Execution using the GUI

The management functions for Recorder Plug-ins are directly accessible in the main menu by clicking on the “Recorder Plug-Ins” icon 



You will see a list of all already available Recorder Plug-Ins and you can activate/deactivate them and also call them manually. Furthermore you can also perform a re-scan for Recorder Plug-Ins.

The MD5 checksum shown in the last column in the image above is calculated over the compiled code of the Java class of the Plug-In. This means that the checksum changes if the code of the Plug-In was modified.

All of these function are also available in the REST API of the Proxy Recorder (→ See “Application Reference Manual”).

2.4 Enabling Debug Output for Recorder Plug-Ins

To enable debug output for Recorder Plug-Ins you can set the startup argument **`-debugRecorderPlugins`** for the ProxySniffer Process (→ See “Application Reference Manual”). Example:

```
java -Xmx1024m ProxySniffer -debugRecorderPlugins . . .
```

Alternately, you can also create or edit the configuration file prxsniff.dat which is located in the ZebraTester installation directory. Example:

```
type prxsniff.dat
-debugRecorderPlugins -dnsstatistic
```

The debug output is written to stdout or to the ZebraTester Console:

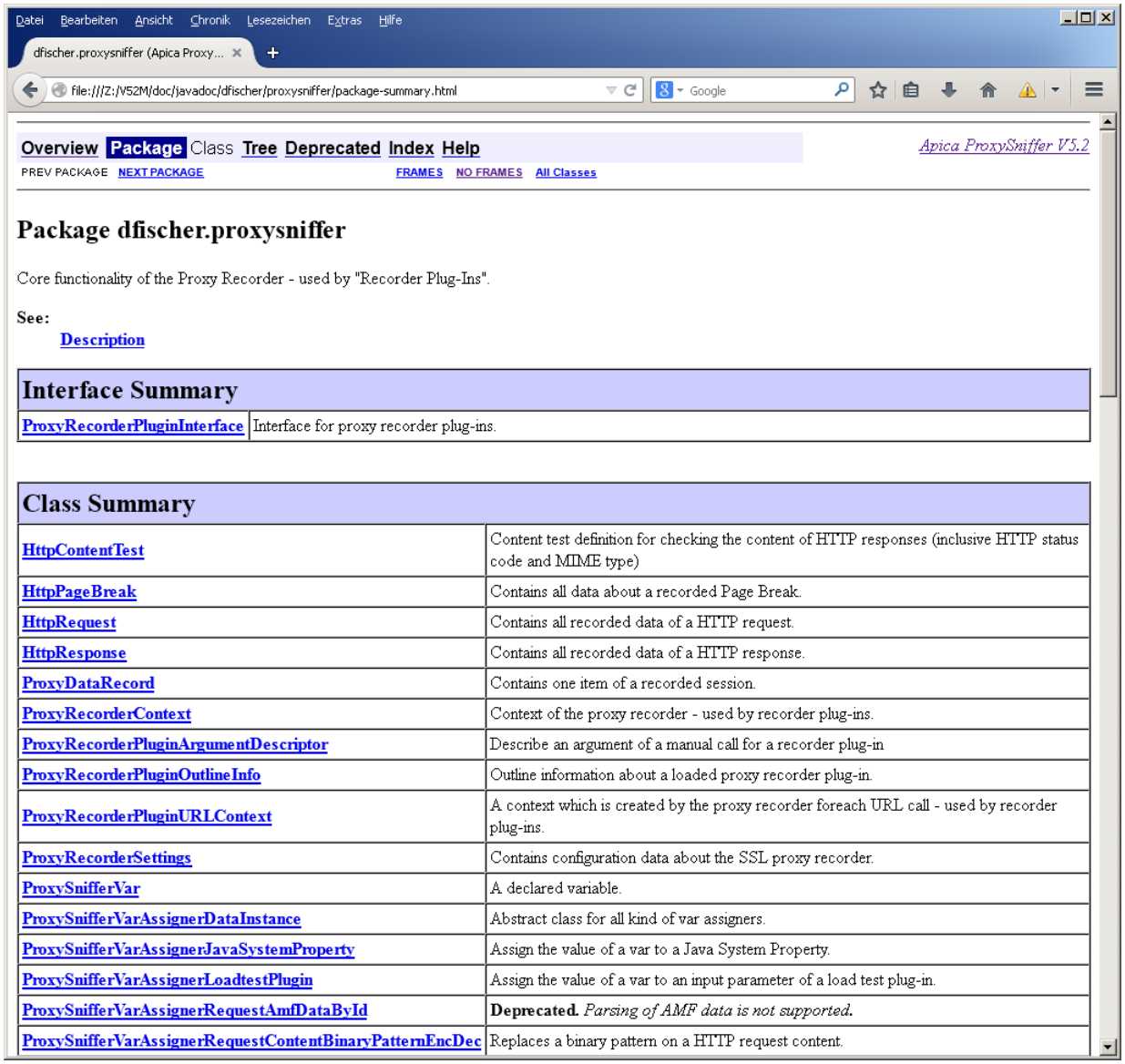
```
22 Dec 2014 23:45:10.539 | main       | --- vvv --- start scanning for recorder plug-ins --- vvv ---
22 Dec 2014 23:45:10.558 | main       | recorder plug-in loaded: class = RecorderPluginDefineUserAccountInputFile, active =
false, position = 1, allow manual ca
ll = true, "Create Input File Definition for User Accounts", MD5 = 90c65fbdc63e8365d4d25346de945607
22 Dec 2014 23:45:10.560 | main       | recorder plug-in loaded: class = RecorderPluginDisableContentTestForImagesCSSJS, active
= false, position = 8, allow man
ual call = true, "Disable Content Test for Images, CCS and JavaScript", MD5 = c59e7436e06fdbf4b4f97f72643edc3c
22 Dec 2014 23:45:10.561 | main       | recorder plug-in loaded: class = RecorderPluginTagRequestResponseHeaderField, active =
false, position = 100, allow manu
al call = false, "Tag HTTP Requests and Responses with Random Number", MD5 = e1e4e2c27ca21db6ea96177fb20edf7c
22 Dec 2014 23:45:10.562 | main       | --- ^^^ --- end scanning for recorder plug-ins --- ^^^ ---
```

3 Developing your own Plug-In

First of all, you cannot use a newer Java version for developing your own Recorder Plug-in as the Java version that is used by your installation of ZebraTester. To have a maximum of compatibility it's recommended to use the older Java version 7.

Although the following examples in this document are relatively simple, programming an own "Recorder Plug-in" might be very complex.
For this reason, we support you only if you have previously visited a training course about developing "Recorder Plug-ins".

During reading this guide it might be helpful also to have a look at the **Java API doc** of the **package dfischer.proxysniffer** which is delivered as a part of the installation kit:



Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe

dfischer.proxysniffer (Apica Proxy...

file:///Z:/V52M/doc/javadoc/dfischer/proxysniffer/package-summary.html

Overview Package Class Tree Deprecated Index Help

Apica ProxySniffer V5.2

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES All Classes

Package dfischer.proxysniffer

Core functionality of the Proxy Recorder - used by "Recorder Plug-Ins".

See:

Description

Interface Summary	
ProxyRecorderPluginInterface	Interface for proxy recorder plug-ins.

Class Summary	
HttpContentTest	Content test definition for checking the content of HTTP responses (inclusive HTTP status code and MIME type)
HttpPageBreak	Contains all data about a recorded Page Break.
HttpRequest	Contains all recorded data of a HTTP request.
HttpResponse	Contains all recorded data of a HTTP response.
ProxyDataRecord	Contains one item of a recorded session.
ProxyRecorderContext	Context of the proxy recorder - used by recorder plug-ins.
ProxyRecorderPluginArgumentDescriptor	Describe an argument of a manual call for a recorder plug-in
ProxyRecorderPluginOutlineInfo	Outline information about a loaded proxy recorder plug-in.
ProxyRecorderPluginURLContext	A context which is created by the proxy recorder foreach URL call - used by recorder plug-ins.
ProxyRecorderSettings	Contains configuration data about the SSL proxy recorder.
ProxySnifferVar	A declared variable.
ProxySnifferVarAssignerDataInstance	Abstract class for all kind of var assigners.
ProxySnifferVarAssignerJavaSystemProperty	Assign the value of a var to a Java System Property.
ProxySnifferVarAssignerLoadtestPlugin	Assign the value of a var to an input parameter of a load test plug-in.
ProxySnifferVarAssignerRequestAmfDataById	**Deprecated.** *Parsing of AMF data is not supported.*
ProxySnifferVarAssignerRequestContentBinaryPatternEncDec	Replaces a binary pattern on a HTTP request content.

3.1 The ProxyRecorderPluginInterface

To develop an own Plug-in you have to create a Java class that implements the **`dfischer.proxysniffer.ProxyRecorderPluginInterface`**. The plug-in itself can't be in a Java package and must be a plain class at top of the Java root.

```
package dfischer.proxysniffer;


/**
 * Interface for proxy recorder plug-ins.
 *
 * @see ProxyRecorderPluginClassLoader
 */
public interface ProxyRecorderPluginInterface
// ================================================================================
{

    // -------- Part 1/3 : information about the plug-in --------

    /**
     * Get a short description about the recorder plug-in.
     *
     * @return  a short description about the recorder plug-in in plain ASCII format.
     */
    public abstract String getDescription();



    /**
     * Get an extended description about the recorder plug-in in HTML formatted text.
     * If the plug-in can be called manually and the manual call has arguments then the
     * extended description should also contain information about the meaning of
     * the arguments.
     *
     * @return the extended description about the recorder plug-in as HTML formatted text,
     * or null if no extended description shall be displayed.
     */
    public abstract String getExtendedHTMLDescription();



    /**
     * Get if the recorder plug-in is active by default (active = called by the proxy recorder
     * during recording a session).
     * Note: if the plug-in allows manual calls only then it will remain always in inactive
     * state, even if this method return true.
     *
     * @return true if the recorder plug-in is active by default. Note: inactive plug-ins can
     * be activated later by using the Web Admin GUI or the REST API.
     *
     * @see #stopRecording(ProxyRecorderContext)
     * @see #stopRecording(ProxyRecorderContext)
     * @see #clearRecording(ProxyRecorderContext)
     * @see #beforeURLTransmitToServer(ProxyRecorderContext, ProxyRecorderPluginURLContext)
     * @see #afterURLReceiveFromServer(ProxyRecorderContext, ProxyRecorderPluginURLContext)
     * @see #allowManualCallOnly()
     */
    public abstract boolean isDefaultActive();


    /**
     * Get a hint about the execution order if multiple active recorder plug-ins are defined.
     * Note: plug-ins with the highest execution position are executed at first for HTTP
     * requests and at last for HTTP responses.
     * If the plug-in method allowManualCallOnly() returns true then the execution order is
     * never used.
     *
     * @return  a hint about the execution order if multiple active recorder plug-ins are
     * defined (can also be a non-unique number)
     *
     * @see #allowManualCallOnly()
     */
    public abstract int getExecutionPosition();
```

```
    /**
     * Get if it's allowed to call the plug-in manually by using the Web Admin GUI or the
     * REST API.
     *
     * @return true = it's allowed to call the plug-in manually
     *
     * @see #getManualCallArgumentDescriptors()
     * @see #manualCall(ProxyRecorderContext, String[])
     */
    public abstract boolean allowManualCall();



    /**
     * Get if the plug-in can be called manually only by using the Web Admin GUI or the REST
     * API.
     * If this method return true then the plug-in remains always in inactive state and cannot
     * be activated.
     *
     * @return true = if the plug-in can be called manually only
     *
     * @see #getManualCallArgumentDescriptors()
     * @see #manualCall(ProxyRecorderContext, String[])
     */
    public abstract boolean allowManualCallOnly();



    /**
     * Get the label text(s) and the default value(s) of the arguments when the plug-in is
     * manually called by using the Web Admin GUI or the REST API.
     *
     * @return  the label text(s) and the default value(s) in plain ASCII format of the
     * arguments Derived from this return information also the number of arguments is
     * implicitly set.
     * You can also return null or an empty array if no arguments are needed when calling the
     * plug-in manually.
     *
     * @see #getManualCallArgumentDescriptors()
     * @see #allowManualCall()
     * @see #manualCall(ProxyRecorderContext, String[])
     */
    public abstract ProxyRecorderPluginArgumentDescriptor[] getManualCallArgumentDescriptors();



    // --- Part 2/3 : manual call of the plug-in by using the Web Admin GUI or the REST API ---

    /**
     * The plug-in is called manually by using the Web Admin GUI or the REST API.
     *
     * @param recorderContext   context of the proxy recorder
     * @param arguments         the arguments passed from the caller (Web Admin GUI or
     *                          REST API)
     *
     * @see #allowManualCall()
     * @see #getManualCallArgumentDescriptors
     */
    public abstract void manualCall(ProxyRecorderContext recorderContext, String[] arguments);
```



```
    // -------- Part 3/3 : proxy recorder calls of the plug-in  --------

    /**
     * Called when the plug-in changes its active state.
     *
     * @param recorderContext   context of the proxy recorder
     * @param isActive          true = the plug-in was inactive and is now active,
     *                          false = the plug-in was active and is now inactive
     */
    public abstract void activeChange(ProxyRecorderContext recorderContext, boolean isActive);



    /**
     * Called when recording is started.
     *
     * @param recorderContext context of the proxy recorder
     */
    public abstract void startRecording(ProxyRecorderContext recorderContext);



     /**
     * Called when recording is stopped.
     *
     * @param recorderContext context of the proxy recorder
     */
    public abstract void stopRecording(ProxyRecorderContext recorderContext);



    /**
     * Called when the recorded session is cleared.
     *
     * @param recorderContext context of the proxy recorder
     */
    public abstract void clearRecording(ProxyRecorderContext recorderContext);



    /**
     * Called just before an URL call is transmitted to the web server. The URL context
     * contains HTTP request data only.
     *
     * @param recorderContext   context of the proxy recorder
     * @param urlContext        context of the URL call
     */
    public abstract void beforeURLTransmitToServer(ProxyRecorderContext recorderContext,
                                                   ProxyRecorderPluginURLContext urlContext);


    /**
     * Called just after an URL call is received from the web server (but before its
     * passed back to the web client of the proxy).
     *
     * @param recorderContext   context of the proxy recorder
     * @param urlContext        context of the URL call
     */
    public abstract void afterURLReceiveFromServer(ProxyRecorderContext recorderContext,
                                                   ProxyRecorderPluginURLContext urlContext);

}
```

3.1.1 Default Constructor

As you see the interface doesn't define a constructor and also your own programmed Plug-In class should not use any constructor (with the exception that a constructor can be programmed that is public and contains no arguments).

Instances of Recorder Plug-ins are created only at product startup, or when a re-scan for plug-ins is made. For each Recorder Plug-In only one instance is created.

3.1.2 Part 1/3: Information about the Plug-In

Method	Programming Hints
getDescription()	The description should be very short (not more than 50 characters recommended).
getExtendedHTMLDescription()	The extended HTML description is shown in the Web GUI when prompting for a change of the active state and when calling the plug-in manually. Note that there are no style sheets (CSS) available. Avoid also any references to images.
isDefaultActive()	Normally you should return always “false”.
getExecutionPosition()	If the logic of your Plug-in is not chained with the logic of another Plug-In you can return any value.
allowManualCall()	Nothing special.
allowManualCallOnly()	Nothing special. If you return true then the methods described later in part 3/3 are never called.
getManualCallArgumentDescriptors()	Here you can specify how many arguments are required for a manual call, and the “labels” and the “default values” of the arguments. This information is shown in the Web GUI. Note that also REST API calls must use the exact number of arguments specified by this method.

3.1.3 Part 2/3: Manual Call of the Plug-In

Method	Programming Hints
manualCall (ProxyRecorderContext recorderContext, String[] arguments);	The ProxyRecorderContext gives you access to the data of the recorded session and to the internal "Var Handler".

3.1.4 Part 3/3: Proxy Recorder Calls of the Plug-In

Method	Programming Hints
activeChange (ProxyRecorderContext recorderContext, boolean isActive)	If the Plug-In is called with isAcitive == false you should free any private resources like DB-connections and open files. And reverse, you should wait with creating private resources until the Plug-In is called with isAcitive == true. Note that this method is not called the first time when isDefaultActive() returns true. This method is called only when a change of the state happens. You can also get the current state of the plug-in in any other method by using ProxyRecorderContext. getRecordState(). The ProxyRecorderContext gives you also access to the data of the recorded session and to the internal "Var Handler".
startRecording	The ProxyRecorderContext gives you access to

(ProxyRecorderContext recorderContext)	the data of the recorded session and to the internal "Var Handler".
stopRecording (ProxyRecorderContext recorderContext)	The ProxyRecorderContext gives you access to the data of the recorded session and to the internal "Var Handler".
clearRecording (ProxyRecorderContext recorderContext);	The ProxyRecorderContext gives you access to the data of the recorded session and to the internal "Var Handler".
beforeURLTransmitToServer (ProxyRecorderContext recorderContext, ProxyRecorderPluginURLContext urlContext	At the time when this method is called the methods ProxyRecorderPluginURLContext. getHttpResponse() and ProxyRecorderPluginURLContext. getRecordData() return always null. You can modify the HTTP request data – before they are sent to the Web server – by using ProxyRecorderPluginURLContext. getHttpRequest(). However, note that you can modify only HTTP request header fields and the HTTP request content-data (not recommended), but **not** network related data like the name of the Web Server or the IP port of the network connection. You can also add any own context information to the ProxyRecorderPluginURLContext which you can retrieve later in the Plug-In method afterURLReceiveFromServer(..). The ProxyRecorderContext gives you access to the data of the recorded session and to the internal "Var Handler".
afterURLReceiveFromServer (ProxyRecorderContext recorderContext, ProxyRecorderPluginURLContext urlContext)	At the time when this method is called the method ProxyRecorderPluginURLContext. getRecordData() returns only valid data if the Proxy Recorder is in recording state. Note that some rare cases can occur where ProxyRecorderPluginURLContext. getHttpResponse() return null. So you have always to check first if this value is not null – before you are using it. If you perform variable declarations by using the "Var Handler" you should get the pointers to the HTTP request and response always from ProxyRecorderPluginURLContext. **getRecordData().**getHttpRequest() and ProxyRecorderPluginURLContext. **getRecordData().**getHttpResponse()

3.1.5 Debug and Log Output

Before any debug output is written in Plug-Ins you should first check if debug output is enabled by calling **`recorderContext.isWriteDebugInfo())`**. To write the debug or log data you should always use one of the methods of **`dfischer.proxysniffer.Stdout.`**

```
if (recorderContext.isWriteDebugInfo())
{
    Stdout.log(”This is a log message”);
}
```

However fatal errors that occur in Plug-Ins should be always logged (or thrown), regardless if the debug output is enabled or not

```
try
{
 …
}
catch (Exception ex)
{
    Stdout.log(ex);       // write the whole stack trace
}
```

3.1.6 Exception Handling

In Plug-Ins you can throw only Exceptions which are from the type **`java.lang.RuntimeException`**.

Two of such exceptions are already defined:

- **`ProxyRecorderPluginIllegalArgumentException`**: you should throw this kind of exception if a parameter of a manual call contains an invalid value. For example when an integer (string-value) is expected in a passed argument that cannot be converted into a Java "int" data type.
- **`ProxyRecorderPluginInternalErrorException`**: Any unexpected error that occur in a Plug-In should converted to this exception. Example:

  ```
  try
  {
   …
  }
  catch (IOException ie)
  {
      throw new ProxyRecorderPluginInternalErrorException("unable to open file", ie);
  }
  ```


3.2 Illustrative Programming Examples

3.2.1 Disable Content Test for Images, CCS and JavaScript

This example of a Plug-In shows how to get access to the “Content Test” settings of URL calls and disables all "Content Tests" for images, CCS and JavaScript .

The state of the Plug-In can be set to active, or alternatively the Plug-In can also manually called.

Source Code of RecorderPluginDisableContentTestForImagesCSSJS.java

```
import dfischer.proxysniffer.HttpContentTest;
import dfischer.proxysniffer.HttpRequest;
import dfischer.proxysniffer.HttpResponse;
import dfischer.proxysniffer.ProxyDataRecord;
import dfischer.proxysniffer.ProxyRecorderPluginArgumentDescriptor;
import dfischer.proxysniffer.ProxyRecorderPluginInterface;
import dfischer.proxysniffer.ProxyRecorderContext;
import dfischer.proxysniffer.ProxyRecorderPluginURLContext;
import dfischer.proxysniffer.Stdout;

import java.util.Vector;


/**
 * A recorder plug-in that disables the content tests for any images as well as for CCS and
 * JavaScript files.
 */
public class RecorderPluginDisableContentTestForImagesCSSJS implements
             ProxyRecorderPluginInterface
{

    // -------- Part 1/3 : information about the plug-in --------

    /**
     * Get a short description about the recorder plug-in.
     *
     * @return  a short description about the recorder plug-in in plain ASCII format.
     */
    public String getDescription()
    {
        return "Disable Content Test for Images, CCS and JavaScript";
    }


    /**
     * Get an extended description about the recorder plug-in in HTML formatted text.

     *
     * @return the extended description about the recorder plug-in as HTML formatted text.
     */
    public String getExtendedHTMLDescription()
    {
        return "Modify the recorded session in such a way that Content Tests for Images, " +
               "CCS Files and JavaScript Files are disabled.<P>" +
               "Note that the HTTP response codes and the Content-Types for such data are "
               "still verified.";
    }


    /**
     * Get if the recorder plug-in is active by default.
     *
     * @return true if the recorder plug-in is active by default
     */
    public boolean isDefaultActive()
    {
        return false;   // default = inactive
    }
```

```
    /**
     * Get a hint about the execution order if multiple active recorder plug-ins are defined.
     *
     * @return  a hint about the execution order if multiple active recorder plug-ins are
     * defined (can also be a non-unique number)
     */
    public int getExecutionPosition()
    {
        return 8;       // any value is ok for this plug-in
    }


    /**
     * Get if it's allowed to call the plug-in manually by using the Web Admin GUI or the REST
     * API.
     *
     * @return true = it's allowed to call the plug-in manually
     */
    public boolean allowManualCall()
    {
        return true;    // allowed
    }


    /**
     * Get if the plug-in can be called manually only by using the Web Admin GUI or the REST
     * API.
     *
     * @return true = if the plug-in can be called manually only
     */
    public boolean allowManualCallOnly()
    {
        return false;   // the plug-in can be set in active state
    }


    /**
     * Get the label text(s) and the default value(s) of the arguments when the plug-in is
     * manually called by using the Web Admin GUI or the REST API.
     *
     * @return the label text(s) and the default value(s) in plain ASCII format of the
     * arguments
     */
    public ProxyRecorderPluginArgumentDescriptor[] getManualCallArgumentDescriptors()
    {
        return null;    // no arguments required for manual call
    }


    // --- Part 2/3 : manual call of the plug-in by using the Web Admin GUI or the REST API ---

    /**
     * The plug-in is called manually by using the Web Admin GUI or the REST API.
     *
     * @param recorderContext   context of the proxy recorder
     * @param arguments         the arguments passed from the caller (Web Admin GUI or REST
     *                          API)
     */
    public void manualCall(ProxyRecorderContext recorderContext, String[] arguments)
    {
        Vector<ProxyDataRecord> recordVector = recorderContext.getRecordVector();

        for (ProxyDataRecord proxyDataRecord : recordVector)
        {
            if (proxyDataRecord.getDataType() == ProxyDataRecord.DATA_TYPE_HTTP_DATA)
            {
                if (proxyDataRecord.hasHttpRequest() && proxyDataRecord.hasHttpResponse() &&
                    proxyDataRecord.hasHttpContentTest())
                {
                    HttpRequest httpRequest = proxyDataRecord.getHttpRequest();
                    HttpResponse httpResponse = proxyDataRecord.getHttpResponse();
                    HttpContentTest httpContentTest = proxyDataRecord.getHttpContentTest();

                    if (isImage(httpResponse) || isCSS(httpRequest, httpResponse) ||
                        isJavaScript(httpRequest, httpResponse))
                    {
                        httpContentTest.applyNoTest();   // disable content test
                        if (recorderContext.isWriteDebugInfo())
                        {
                            Stdout.log("Content test disabled for " + httpRequest.getUrl());
                        }
                    }
                }
```



```
            }
        }
    }
```

The Vector<ProxyDataRecord> contains the whole recorded session. A ProxyDataRecord can be either a “Page Break” or contain “HTTP Data” (an URL Call). If it contains “HTTP Data” you get access to the “HttpContentTest” which is modified → disabled by calling httpContentTest.applyNoTest();

```
// -------- Part 3/3 : proxy recorder calls of the plug-in  --------

/**
 * Called when the plug-in changes its active state.
 */
public void activeChange(ProxyRecorderContext recorderContext, boolean isActive)
{
    // not used in this plug-in
}


/**
 * Called when recording is started.
 */
public void startRecording(ProxyRecorderContext recorderContext)
{
    // not used in this plug-in
}


/**
 * Called when recording is stopped.
 */
public void stopRecording(ProxyRecorderContext recorderContext)
{
    // not used in this plug-in
}


/**
 * Called when the recorded session is cleared.
 */
public void clearRecording(ProxyRecorderContext recorderContext)
{
    // not used in this plug-in
}


/**
 * Called just before an URL call is transmitted to the web server.
 */
public void beforeURLTransmitToServer(ProxyRecorderContext recorderContext,
                                      ProxyRecorderPluginURLContext urlContext)
{
    // not used in this plug-in
}


/**
 * Called just after an URL call is received from the web server
 * (but before its passed back to the web client of the proxy).
 *
 * @param recorderContext   context of the proxy recorder
 * @param urlContext        context of the URL call
 */
public void afterURLReceiveFromServer(ProxyRecorderContext recorderContext,
            ProxyRecorderPluginURLContext urlContext)
{
    if (urlContext.getRecordData() != null)
    {
        HttpRequest httpRequest = urlContext.getRecordData().getHttpRequest();
        HttpResponse httpResponse = urlContext.getRecordData().getHttpResponse();
        HttpContentTest httpContentTest = urlContext.getRecordData().getHttpContentTest();
        if ((httpRequest != null) && (httpResponse != null) && (httpContentTest != null))
        {
            if (isImage(httpResponse) || isCSS(httpRequest, httpResponse) ||
                isJavaScript(httpRequest, httpResponse))
            {
                httpContentTest.applyNoTest();  // disable content test
                if (recorderContext.isWriteDebugInfo())
                {
                    Stdout.log("Content test disabled for " + httpRequest.getUrl());
```



```
                }
            }
        }
    }
}
```

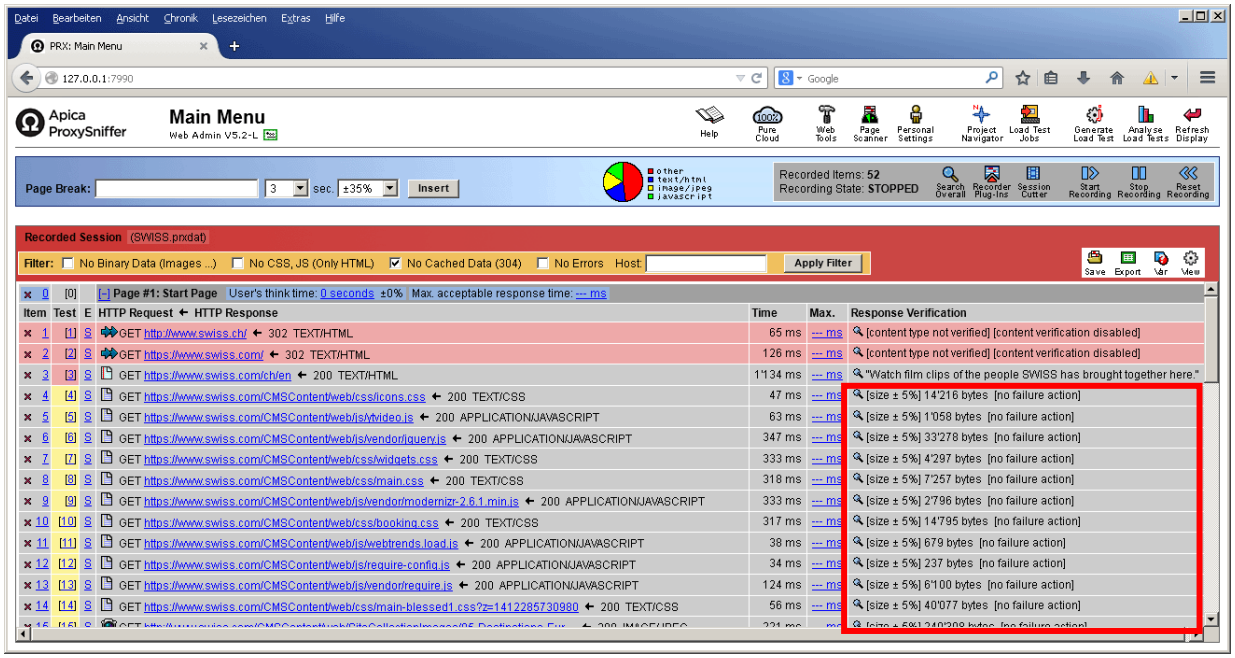
Here happens almost the same as before shown in the Method manualCall(..). The only difference is that you get with urlContext.getRecordData() directly a single ProxyDataRecord (the actual URL call) which is in this case either null (recording disabled) or always from the type “HTTP Data”.

```
// ========= Local Plug-In Methods ================================

/**
 * Check if the HTTP response is an image.
 *
 * @param httpResponse  the HTTP response
 *
 * @return true if the HTTP response is an image
 */
private static boolean isImage(HttpResponse httpResponse)
{
    String contentType = httpResponse.getContentType();
    if (contentType != null)
        return contentType.toLowerCase().startsWith("image/");

    return false;
}


/**
 * Check if the HTTP response is a CSS file.
 *
 * @param httpRequest   the HTTP request
 * @param httpResponse  the HTTP response
 *
 * @return true if the HTTP response is a CSS file
 */
private static boolean isCSS(HttpRequest httpRequest, HttpResponse httpResponse)
{
    String contentType = httpResponse.getContentType();
    if (contentType != null)
    {
        if (contentType.toLowerCase().endsWith("/css"))
            return true;
    }

    // check also extension of request path w/o CGI parameters
    String uri = httpRequest.getPlainFile();
    if (uri.toLowerCase().endsWith(".css"))
        return true;

    return false;
}


/**
 * Check if the HTTP response is a JavaScript file.
 *
 * @param httpRequest   the HTTP request
 * @param httpResponse  the HTTP response
 *
 * @return true if the HTTP response is a JavaScript file
 */
private static boolean isJavaScript(HttpRequest httpRequest, HttpResponse httpResponse)
{
    String contentType = httpResponse.getContentType();
    if (contentType != null)
    {
        if (contentType.toLowerCase().endsWith("/javascript") ||
            contentType.toLowerCase().endsWith("/x-javascript"))
            return true;
    }

    // check also extension of request path w/o CGI parameters
    String uri = httpRequest.getPlainFile();
    if (uri.toLowerCase().endsWith(".js"))
        return true;

    return false;
}
```

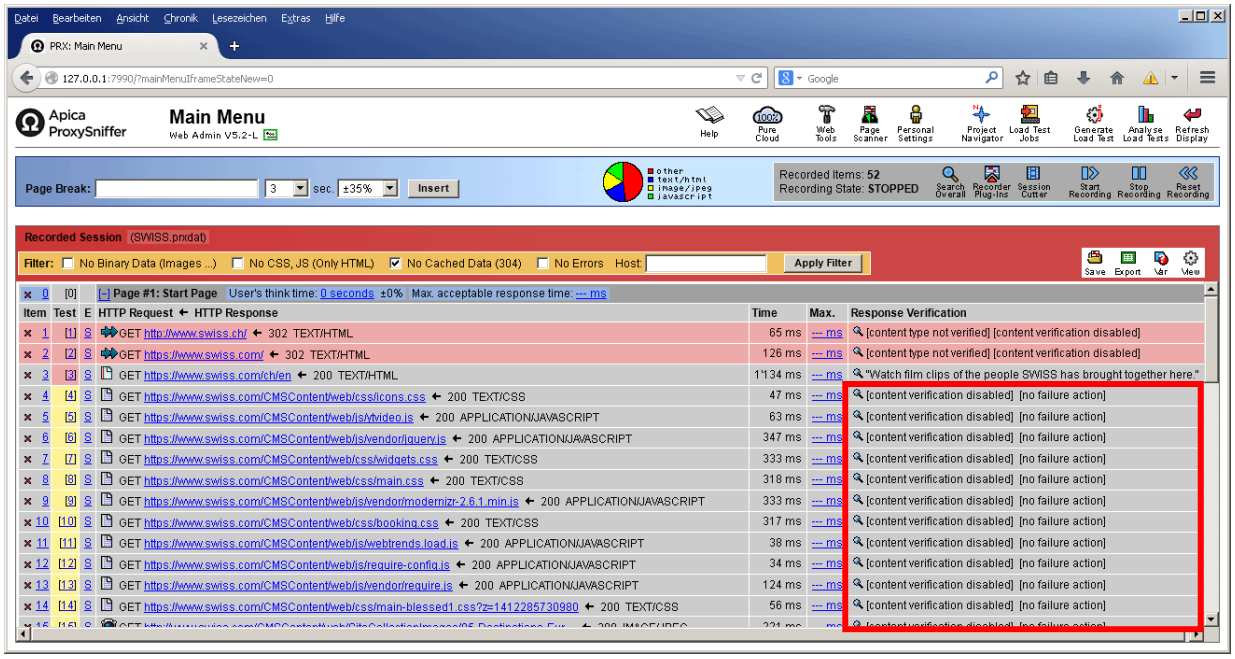


```
} // end of class
```

Example: Recorded session before manual call:



Recorded session after manual call:

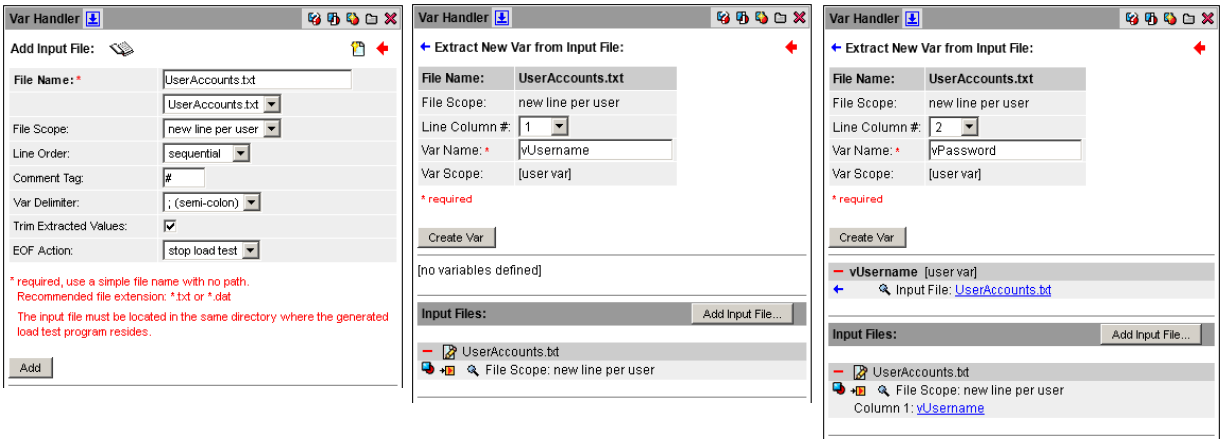


3.2.2 Create Input File Definition for User Accounts

This Plug-In automates the following steps:

- Define an "Input File" for user accounts and create the two variables vUsername and vPassword inclusive the corresponding "var assigners".

If you would do the same thing in the Web GUI you have to make may clicks and to select a couple of options:



Source Code of RecorderPluginDisableContentTestForImagesCSSJS.java

```
import dfischer.proxysniffer.ProxyRecorderContext;
import dfischer.proxysniffer.ProxyRecorderPluginArgumentDescriptor;
import dfischer.proxysniffer.ProxyRecorderPluginIllegalArgumentException;
import dfischer.proxysniffer.ProxyRecorderPluginInterface;
import dfischer.proxysniffer.ProxyRecorderPluginInternalErrorException;
import dfischer.proxysniffer.ProxyRecorderPluginURLContext;
import dfischer.proxysniffer.ProxySnifferVar;
import dfischer.proxysniffer.ProxySnifferVarExtractorInputFile;
import dfischer.proxysniffer.ProxySnifferVarHandler;
import dfischer.proxysniffer.ProxySnifferVarSourceHandler;
import dfischer.proxysniffer.ProxySnifferVarSourceTextfile;
import dfischer.utils.Lib;


/**
 * A recorder plug-in that crates a definition for an input file that contains on each line
 * a username and a password.
 */
public class RecorderPluginDefineUserAccountInputFile implements ProxyRecorderPluginInterface
{
    private static final String VAR_NAME_USERNAME = "vUsername";
    private static final String VAR_NAME_PASSWORD = "vPassword";
    private static final String FILE_LINE_COMMENT_CHAR = "#";
    private static final String FILE_VAR_DELIMITER_CHAR = ";";


    /**
     * Get a short description about the recorder plug-in.
     *
     * @return  a short description about the recorder plug-in in plain ASCII format.
     */
    public String getDescription()
    {
        return "Create Input File Definition for User Accounts";
    }


    /**
     * Get an extended description about the recorder plug-in in HTML formatted text.
     *
     * @return the extended description about the recorder plug-in as HTML formatted text
     */
    public String getExtendedHTMLDescription()
    {
        return "Create an <B>Input File</B> definition for <B>User Accounts</B> that " +
               "contains on each line an username and a password, separated by " +
               "a '" + Lib.toHtmlText(FILE_VAR_DELIMITER_CHAR) + "' character. Use the " +
```

```
                "'" + Lib.toHtmlText(FILE_LINE_COMMENT_CHAR) + "' character to comment out " +
                "lines.<P>This plug-in creates also " +
                "two variables (<B>" + VAR_NAME_USERNAME + "</B> and <B>" + VAR_NAME_PASSWORD +
                "</B>) and the corresponding var extractors. " +
                "The scope of the file is <B>new line per user</B> and the load test is " +
                "<B>aborted when the end of the file is reached</B>.<P>" +
                "Please note that the name of the input file must match the following rules:" +
                "<UL style=\"margin-top:0px\">" +
                "<LI style=\"padding-top:0px\">The name of the input file can contain only " +
                "the following characters: '0'..'9', 'A'..'Z', 'a'..'z', '_' and '.'</LI>" +
                "<LI style=\"padding-top:5px\">The name of the input file must contain a " +
                "point character ('.'), which is not at the first and not at the last " +
                "position</LI>" +
                "<LI style=\"padding-top:5px\">The first character of the input file must " +
                "not be a digit ('0'..'9' is not allowed for the first character)</LI>" +
                 "</UL>" +
                 "The recommended file name extension is '.txt'";
    }


    /**
     * Get if the recorder plug-in is active by default.
     *
     * @return true if the recorder plug-in is active by default.
     */
    public boolean isDefaultActive()
    {
        return false;   // default = inactive
    }


    /**
     * Get a hint about the execution order if multiple active recorder plug-ins are defined.
     *
     * @return  a hint about the execution order
     */
    public int getExecutionPosition()
    {
        return 1;   // any number is ok for this plug-in
    }


    /**
     * Get if it's allowed to call the plug-in manually by using the Web Admin GUI or the REST
     * API.
     *
     * @return true = it's allowed to call the plug-in manually
     */
    public boolean allowManualCall()
    {
        return true;    // yes, allow manual calls
    }


    /**
     * Get if the plug-in can be called manually only by using the Web Admin GUI or the REST
     * API.
     * If this method return true then the plug-in remains always in inactive state and cannot
     * be activated.
     *
     * @return true = if the plug-in can be called manually only
     */
    public boolean allowManualCallOnly()
    {
        return true;    // yes - the plug-in cannot be set to active - manual calls only
    }
```

allowManualCallOnly() returns true – so the Plug-In cannot set in active state.


```
/**
 * Get the label text(s) and the default value(s) of the arguments when the plug-in is
 * manually called by using the Web Admin GUI or the REST API.
 *
 * @return  the label text(s) and the default value(s) in plain ASCII format of the
 * arguments.
 * Derived from this return information also the number of arguments is implicitly set.
 */
public ProxyRecorderPluginArgumentDescriptor[] getManualCallArgumentDescriptors()
{
    // this plug-in has only one argument - the name of the file
    ProxyRecorderPluginArgumentDescriptor[] argumentDescriptors = new
    ProxyRecorderPluginArgumentDescriptor[1];

    argumentDescriptors[0] = new ProxyRecorderPluginArgumentDescriptor("File Name",
                                                                       "UserAccounts.txt");
    return argumentDescriptors;
}
```

The Plug-In declares that one argument must be passed when it is manually called. The label "File Name" and the default value "UserAccounts.txt" is set for that argument.

```
// --- Part 2/3 : manual call of the plug-in by using the Web Admin GUI or the REST API ---

/**
 * The plug-in is called manually by using the Web Admin GUI or the REST API.
 *
 * @param recorderContext   context of the proxy recorder
 * @param arguments         the arguments passed from the caller (Web Admin GUI or REST
 *                          API) - in this case arguments[0] is the file name (only one
 *                          argument)
 */
public void manualCall(ProxyRecorderContext recorderContext, String[] arguments)
{
    // check first if the file name is valid (input verification)
    if (!ProxySnifferVarSourceTextfile.isValidFileName(arguments[0]))
        throw new ProxyRecorderPluginIllegalArgumentException("invalid file name: \"" +
                                                              arguments[0] + "\"");

    // define now the file
    defineUserAccountInputFile(recorderContext, arguments[0]);
}
```

The Plug-In verifies first if the passed argument (the file name) is valid. If this is not the case then an ProxyRecorderPluginIllegalArgumentException is thrown.

After that the local method defineUserAccountInputFile(recorderContext,<file name>) is called


```
// -------- Part 3/3 : proxy recorder calls of the plug-in  --------

/**
 * Called when the plug-in changes its active state.
 */
public void activeChange(ProxyRecorderContext recorderContext, boolean isActive)
{
    // not used in this plug-in
}


/**
 * Called when recording is started.
 */
public void startRecording(ProxyRecorderContext recorderContext)
{
    // not used in this plug-in
}


/**
 * Called when recording is stopped.
 */
public void stopRecording(ProxyRecorderContext recorderContext)
{
    // not used in this plug-in
}


/**
 * Called when the recorded session is cleared.
 */
public void clearRecording(ProxyRecorderContext recorderContext)
{
    // not used in this plug-in
}


/**
 * Called just before an URL call is transmitted to the web server.
 */
public void beforeURLTransmitToServer(ProxyRecorderContext recorderContext,
                                      ProxyRecorderPluginURLContext urlContext)
{
    // not used in this plug-in
}


/**
 * Called just after an URL call is received from the web server (but before its passed
 * back to the web client of the proxy).
 */
public void afterURLReceiveFromServer(ProxyRecorderContext recorderContext,
                                      ProxyRecorderPluginURLContext urlContext)
{
    // not used in this plug-in
}
```



```
    // ========= Local Plug-In Method ==================================

    /**
     * Create a definition for an input file that contains on each line a username and a
     * password.
     *
     * @param recorderContext   the context of the proxy recorder
     * @param inputFileName     the name of the input file
     *
     */
    private static void defineUserAccountInputFile(ProxyRecorderContext recorderContext,
                                                   String inputFileName) throws
                                                   ProxyRecorderPluginInternalErrorException
    {
        ProxySnifferVarSourceHandler varSourceHandler = recorderContext.getVarSourceHandler();
        ProxySnifferVarHandler varHandler = recorderContext.getVarHandler();

        try
        {
            // create the definition for the input file
            ProxySnifferVarSourceTextfile varSourceTextfile = new
             ProxySnifferVarSourceTextfile(inputFileName,
                                           ProxySnifferVar.SCOPE_USER,
                                           ProxySnifferVarSourceTextfile.LINE_ORDER_SEQUENTIAL,
                                           FILE_LINE_COMMENT_CHAR,
                                           FILE_VAR_DELIMITER_CHAR,
                                           true,
                                           ProxySnifferVarSourceTextfile.EOF_STOP_LOAD_TEST);
            varSourceTextfile = (ProxySnifferVarSourceTextfile)
                                varSourceHandler.addVarSource(varSourceTextfile);

            // create the variables for the username and the password
            ProxySnifferVar usernameVar = new ProxySnifferVar(ProxySnifferVar.SCOPE_USER,
                                                              VAR_NAME_USERNAME);
            usernameVar = varHandler.addVar(usernameVar);

            ProxySnifferVar passwordVar = new ProxySnifferVar(ProxySnifferVar.SCOPE_USER,
                                                              VAR_NAME_PASSWORD);
            passwordVar = varHandler.addVar(passwordVar);

            // create the var extractors for the username and the password
            ProxySnifferVarExtractorInputFile usernameVarExtractor = new
             ProxySnifferVarExtractorInputFile(VAR_NAME_USERNAME,
                                               varSourceTextfile.getUniqueKey(), 1);
            usernameVarExtractor = (ProxySnifferVarExtractorInputFile)
                                   usernameVar.addVarExtractor(usernameVarExtractor);

            ProxySnifferVarExtractorInputFile passwordVarExtractor = new
             ProxySnifferVarExtractorInputFile(VAR_NAME_PASSWORD,
                                               varSourceTextfile.getUniqueKey(), 2);
            passwordVarExtractor = (ProxySnifferVarExtractorInputFile)
                                   passwordVar.addVarExtractor(passwordVarExtractor);
        }
        catch (Exception e)
        {
            throw new ProxyRecorderPluginInternalErrorException(e);
        }
    }

}  // end of class
```

This local method implements the inner algorithm of the Plug-In.

At first, a definition for the Input File is created and added to the **varSourceHandler** (which is the central handler for all sources used by any "var extractors").

Next, a variable definition for the "var name" <VAR_NAME_USERNAME>(=" vUsername") is created and added to the **varHandler** (which is the central handler for all "var(s)"). The same is also made for the second variable <VAR_NAME_PASSWORD >(=" vPassword").

Finally, a "var extractor" is created that connects the column number one of the Input File to the value of the variable <VAR_NAME_USERNAME>(=" vUsername") by calling the <var>. addVarExtractor(..) method of the declared "var". The same is the also made for the second variable <VAR_NAME_PASSWORD >(=" vPassword"), but that "var extractor" references column number two of the Input File.

As you see in the code above the code for creating the "var source" and the "vars(s)" and the "var extractors(s)" look a little bit strange:

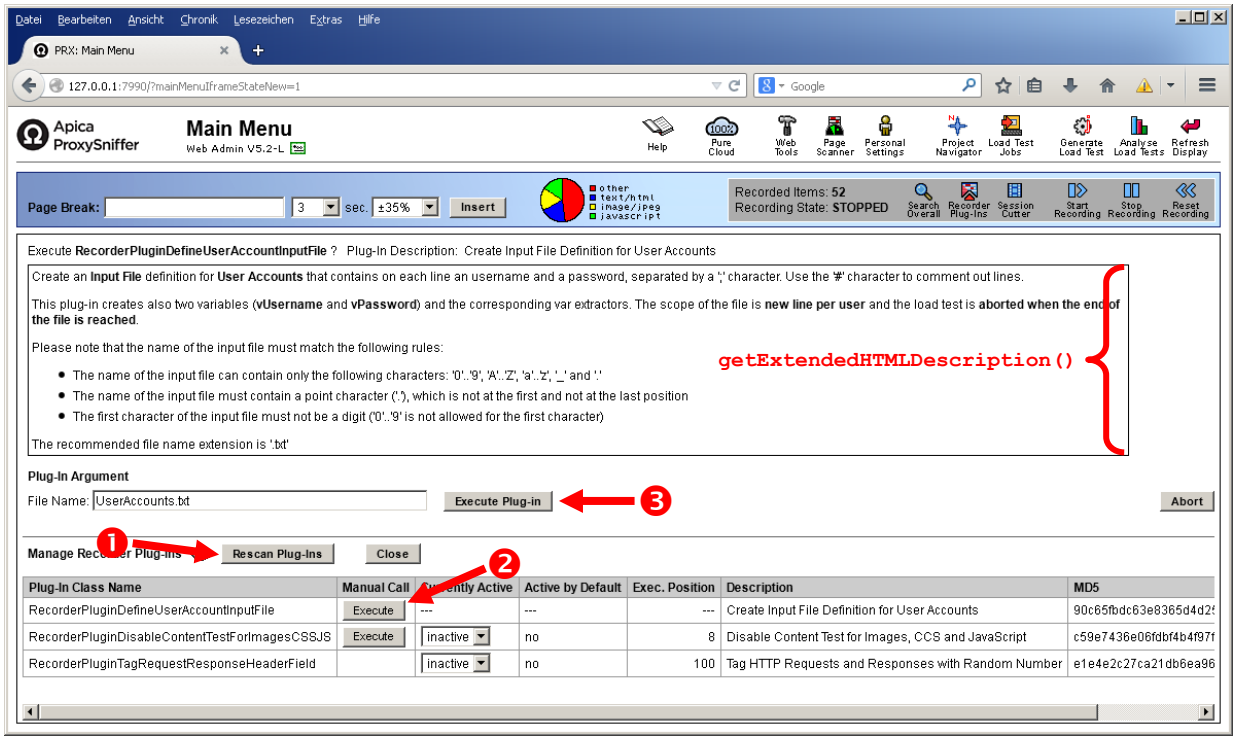
```
•varSourceTextfile = (ProxySnifferVarSourceTextfile)
    varSourceHandler.addVarSource(varSourceTextfile);

•usernameVar = varHandler.addVar(usernameVar);

•passwordVar = varHandler.addVar(passwordVar);

•usernameVarExtractor = (ProxySnifferVarExtractorInputFile)
    usernameVar.addVarExtractor(usernameVarExtractor);

•passwordVarExtractor = (ProxySnifferVarExtractorInputFile)
    passwordVar.addVarExtractor(passwordVarExtractor);
```

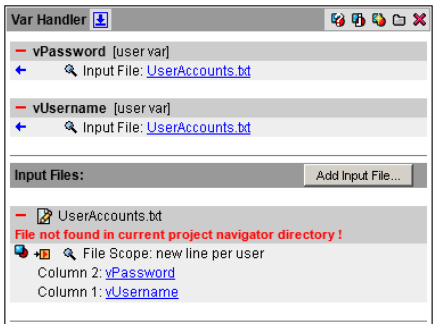
But that way guarantees, that no duplicated declarations occur when exactly the same things are declared once again.

The corresponding methods of **<ProxySnifferVarSourceHandler >.addVarSource(..)**, **<ProxySnifferVarHandler >.addVar(..)**, and **<ProxySnifferVar>. addVarExtractor(..)** are checking first if a duplicated declaration already exists, and – if this is the case – return the pointer of the (already) duplicated declaration, rather than the pointer of your newly created declaration.

It's very important that you follow this programming approach when you write your own Plug-In. You will see the effect also when you manually call this plug-in repeatedly by passing the same argument to the Plug-In: no second input file is created and no extra "vars" and "var assignes" are created. This ensures also that the generated code of the load test program is well optimized.

Once the Plug-In is compiled and its class is placed in the ProxySniffer installation directory you can trigger a re-scan (needed for the first time only) and then manually call the Plug-In:





After the Plug-In was called you can double-check the result in the Web GUI. If the Input File does not yet exists in the "Project Navigator" directory then copy the file to this directory and click on the "Refresh" icon. Alternatively you can also click on the icon and enter the data for the user names and the passwords manually (separated by a semicolon) – line per line.

3.2.3 Tag HTTP Requests and Responses with Random Number

This Plug-In adds a HTTP header field with a random number to each request and response. Per single pair of request/response the same random number is used. Note that the additional header field is also sent to the Web server.

The state of the Plug-In can be set to active, but manual calls are not allowed.

Source Code of RecorderPluginTagRequestResponseHeaderField.java

```
import dfischer.proxysniffer.ProxyRecorderContext;
import dfischer.proxysniffer.ProxyRecorderPluginArgumentDescriptor;
import dfischer.proxysniffer.ProxyRecorderPluginInterface;
import dfischer.proxysniffer.ProxyRecorderPluginURLContext;
import dfischer.proxysniffer.Stdout;

import java.util.Random;


/**
 * A recorder plug-in that adds a HTTP header field with a random number to each request
 * and response.
 * Per single pair of request/response the same random number is used.
 */
public class RecorderPluginTagRequestResponseHeaderField implements
             ProxyRecorderPluginInterface
{
    // the name of the additional header field
    private static final String ADD_HEADER_FIELD_NAME = "RandomNumberTag";

    // initialize the random generator when the plug-in is loaded (scanned).
    private Random rand = new Random();


    // -------- Part 1/3 : information about the plug-in --------

    /**
     * Get a short description about the recorder plug-in.
     *
     * @return  a short description about the recorder plug-in in plain ASCII format.
     */
    public String getDescription()
    {
        return "Tag HTTP Requests and Responses with Random Number";
    }


    /**
     * Get an extended description about the recorder plug-in in HTML formatted text.
     *
     * @return the extended description about the recorder plug-in as HTML formatted text
     */
    public String getExtendedHTMLDescription()
    {
        return "This plug-in adds during recording the HTTP request header field <B>" +
               ADD_HEADER_FIELD_NAME + "</B> " +
               "to each HTTP request and to each HTTP response. Per single pair of " +
               "request/response the same random number is used.<P>" +
               "Note that the additional header field is also sent to the Web server.";
    }


    /**
     * Get if the recorder plug-in is active by default.
     *
     * @return true if the recorder plug-in is active by default.
     */
    public boolean isDefaultActive()
    {
        return false;   // default = inactive
    }


    /**
     * Get a hint about the execution order if multiple active recorder plug-ins are defined.
     *
     * @return  a hint about the execution order if multiple active recorder plug-ins are
     * defined (can also be a non-unique number)
     */
```



```
public int getExecutionPosition()
{
    return 100; // any value is ok for this plug-in
}


/**
 * Get if it's allowed to call the plug-in manually by using the Web Admin GUI or the REST
 * API.
 *
 * @return true = it's allowed to call the plug-in manually
 */
public boolean allowManualCall()
{
    return false;   // manual calls are not allowed
}


/**
 * Get if the plug-in can be called manually only by using the Web Admin GUI or the REST
 * API.
 *
 * @return true = if the plug-in can be called manually only
 */
public boolean allowManualCallOnly()
{
    return false;   // manual calls are not allowed
}


/**
 * Get the label text(s) and the default value(s) of the arguments when the plug-in is
 * manually called by using the Web Admin GUI or the REST API.
 *
 * @return  the label text(s) and the default value(s) in plain ASCII format of the
 * arguments.
 */
public ProxyRecorderPluginArgumentDescriptor[] getManualCallArgumentDescriptors()
{
    // manual calls are not allowed so it makes no sense to specify here any arguments
    return null;
}


// --- Part 2/3 : manual call of the plug-in by using the Web Admin GUI or the REST API ---

/**
 * The plug-in is called manually by using the Web Admin GUI or the REST API.
 */
public void manualCall(ProxyRecorderContext recorderContext, String[] arguments)
{
    // not used in this plug-in, manual calls are not allowed
}


// -------- Part 3/3 : proxy recorder calls of the plug-in  --------

/**
 * Called when the plug-in changes its active state.
 */
public void activeChange(ProxyRecorderContext recorderContext, boolean isActive)
{
    // not used in this plug-in
}


/**
 * Called when recording is started.
 */
public void startRecording(ProxyRecorderContext recorderContext)
{
    // not used in this plug-in
}


/**
 * Called when recording is stopped.
 */
public void stopRecording(ProxyRecorderContext recorderContext)
{
    // not used in this plug-in
}
```



```
    /**
     * Called when the recorded session is cleared.
     */
    public void clearRecording(ProxyRecorderContext recorderContext)
    {
        // not used in this plug-in
    }


    /**
     * Called just before an URL call is transmitted to the web server.
     *
     * @param recorderContext   context of the proxy recorder
     * @param urlContext        context of the URL call
     */
    public void beforeURLTransmitToServer(ProxyRecorderContext recorderContext,
                                          ProxyRecorderPluginURLContext urlContext)
    {
        // generate a new random number
        int posRandInt = rand.nextInt(Integer.MAX_VALUE);

        // add the random number as additional request header field
        urlContext.getHttpRequest().addOrReplaceHeaderField(ADD_HEADER_FIELD_NAME,
                                                            "" + posRandInt);

        // store the random number in the request/response context
        urlContext.setRequestResponseContext(new Integer(posRandInt));

        if (recorderContext.isWriteDebugInfo())
            Stdout.log("Add Random Header Field " + ADD_HEADER_FIELD_NAME + ": " +
                       posRandInt + " to http request header");
    }
```

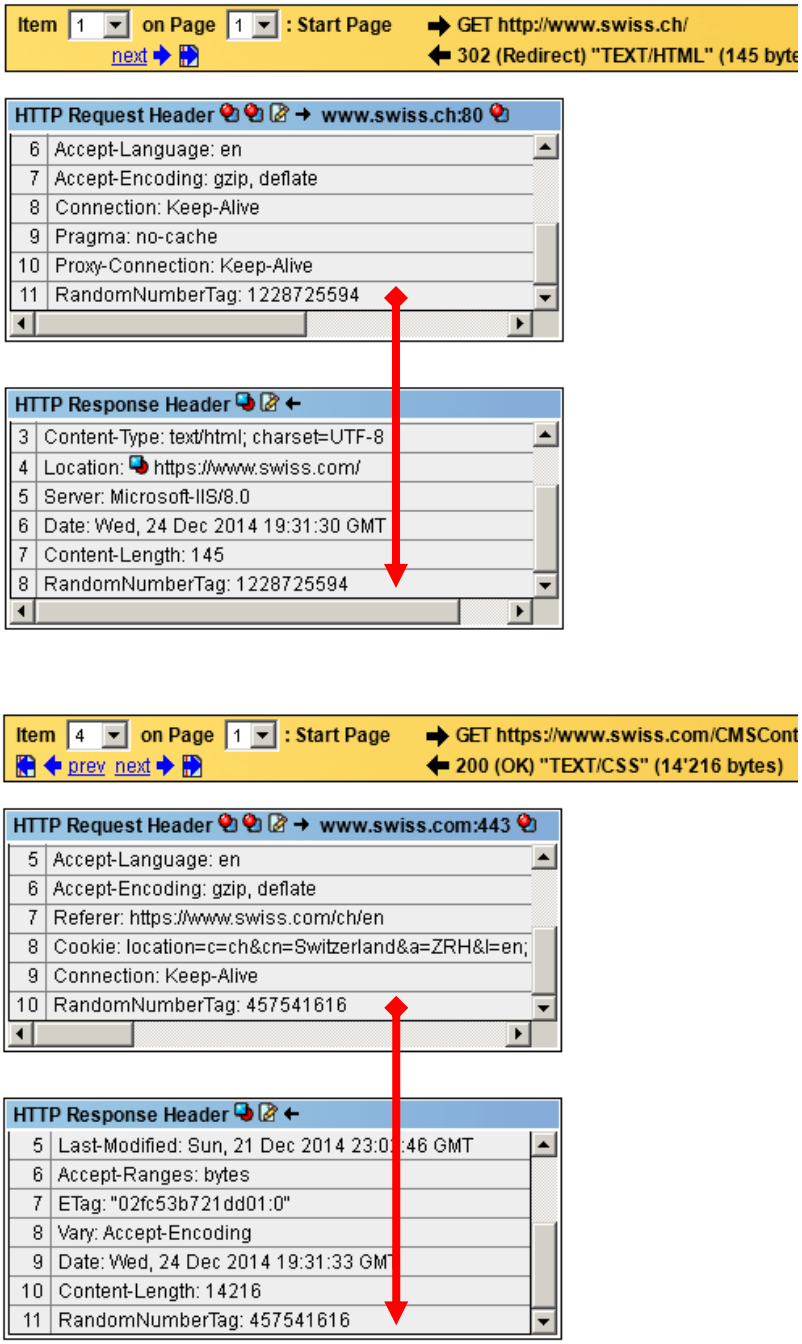
Before each HTTP request that is sent to the Web server the Plug-In generates an new random number and adds the HTTP header field by calling urlContext.getHttpRequest().addOrReplaceHeaderField(..). Then the random number is also stored request/response context by calling urlContext.setRequestResponseContext(..).

```
    /**
     * Called just after an URL call is received from the web server (but before
     * its passed back to the web client of the proxy).
     *
     * @param recorderContext   context of the proxy recorder
     * @param urlContext        context of the URL call
     */
    public void afterURLReceiveFromServer(ProxyRecorderContext recorderContext,
                                          ProxyRecorderPluginURLContext urlContext)
    {
        // get the random number from the request/response context
        // (meaning from the context already set in the corresponding the HTTP request)
        int posRandInt = ((Integer) urlContext.getRequestResponseContext());

        // set the same random number as additional response header field
        urlContext.getHttpResponse().addOrReplaceHeaderField(ADD_HEADER_FIELD_NAME,
                                                             "" + posRandInt);

        if (recorderContext.isWriteDebugInfo())
            Stdout.log("Add Random Header Field " + ADD_HEADER_FIELD_NAME + ": " +
                       posRandInt + " to http response header");
    }
```

After the corresponding HTTP response is received from the Web server the random number is restored from the request/response context by calling urlContext.getRequestResponseContext(). Then the HTTP header field is added by calling urlContext.getHttpResponse().addOrReplaceHeaderField(..);

```
}    // end of class
```


Once the plug-in was set to active the recorded requests and their corresponding responses are tagged with the additional header field "RandomNumberTag ":



3.3 Proxy Recorder Data Structures

3.3.1 The ProxyRecorderContext

The Proxy Recorder Context which is passed to almost all Plug-in methods as an instance of the Java class **dfischer.proxysniffer.ProxyRecorderContext** gives you access to the Recoded Session, to the "Var Handler" and to the "Var Source Handler".

Source Code of dfischer.proxysniffer.ProxyRecorderContext.java

```
package dfischer.proxysniffer;

import dfischer.utils.LoadtestExternalResources;

import java.util.Vector;


/**
 * Context of the proxy recorder - used by recorder plug-ins.
 *
 * @see ProxyRecorderPluginInterface
 */
public class ProxyRecorderContext
// ================================================================================================
{
    private ProxyRecorderSettings recorderSettings;
    private int recordState;
    private Vector<ProxyDataRecord> recordVector;
    private ProxySnifferVarSourceHandler varSourceHandler;
    private ProxySnifferVarHandler varHandler;
    private LoadtestExternalResources externalResources;
    private boolean writeDebugInfo;


    /**
     * Non-public constructor. Create a new instance.
     *
     * @param recorderSettings  configuration data of the proxy recorder
     * @param recordState       the current recorder state (RECORD_STATE_STOPPED or
     *                          RECORD_STATE_STARTED)
     * @param recordVector      the recorded session
     * @param varSourceHandler  the handler of source definitions for extracting values
     *                          into variables
     * @param varHandler        the handler for all variables and variable assigners
     * @param externalResources the handler for external resources
     * @param writeDebugInfo    if true the recorder plug-in shall write debug information
     *                          to stdout - recommended stdout methods:
     *                          dfischer.proxysniffer.Stdout.log(..)
     *
     * @see ProxySniffer#RECORD_STATE_STOPPED
     * @see ProxySniffer#RECORD_STATE_STARTED
     * @see dfischer.proxysniffer.Stdout
     */
    ProxyRecorderContext(ProxyRecorderSettings recorderSettings,
                         int recordState,
                         Vector<ProxyDataRecord> recordVector,
                         ProxySnifferVarSourceHandler varSourceHandler,
                         ProxySnifferVarHandler varHandler,
                         LoadtestExternalResources externalResources,
                         boolean writeDebugInfo)
    // --- constructor ------------------------------------------------------------------
    {
        this.recorderSettings = recorderSettings;
        this.recordState = recordState;
        this.recordVector = recordVector;
        this.varSourceHandler = varSourceHandler;
        this.varHandler = varHandler;
        this.externalResources = externalResources;
        this.writeDebugInfo = writeDebugInfo;
    }


    /**
     * Get the configuration data of the proxy recorder.
     *
     * @return the configuration data of the proxy recorder
     */
    public ProxyRecorderSettings getRecorderSettings()
```



```
// --------------------------------------------------------------------------------
{
    return recorderSettings;
}


/**
 * Get the current recorder state.
 *
 * @return  the current recorder state
 *
 * @see ProxySniffer#RECORD_STATE_STOPPED
 * @see ProxySniffer#RECORD_STATE_STARTED
 */
public int getRecordState()
// --------------------------------------------------------------------------------
{
    return recordState;
}


/**
 * Get the recorded session.
 *
 * @return the recorded session
 */
public Vector<ProxyDataRecord> getRecordVector()
// --------------------------------------------------------------------------------
{
    return recordVector;
}


/**
 * Get the handler of source definitions for extracting values into variables.
 *
 * @return  the handler of source definitions for extracting values into variables
 */
public ProxySnifferVarSourceHandler getVarSourceHandler()
// --------------------------------------------------------------------------------
{
    return varSourceHandler;
}


/**
 * Get the handler for all variables and variable assigners.
 *
 * @return  the handler for all variables and variable assigners
 */
public ProxySnifferVarHandler getVarHandler()
// --------------------------------------------------------------------------------
{
    return varHandler;
}


/**
 * Get the handler for external resources.
 *
 * @return  the handler for external resources
 */
public LoadtestExternalResources getExternalResources()
// --------------------------------------------------------------------------------
{
    return externalResources;
}


/**
 * Get if debug information of the recorder plug-in shall be written to stdout.
 " Recommended stdout methods: dfischer.proxysniffer.Stdout.log(..)
 *
 * @return  true = the recorder plug-in shall write debug information to stdout,
 *          false = don't write any debug information in recorder plug-in
 *
 * @see dfischer.proxysniffer.Stdout
 */
public boolean isWriteDebugInfo()
// --------------------------------------------------------------------------------
{
    return writeDebugInfo;
}
```



```
}:   // end of class
```

3.3.2 The Vector of ProxyDataRecord

The Vector of ProxyDataRecord which you can get from the ProxyRecorderContext contains the whole recorded session.

Source Code of **dfischer.proxysniffer.ProxyDataRecord**.java (an element of the Vector, incomplete compendium):

```
package dfischer.proxysniffer;


/**
 * Contains one item of a recorded session. Such an item is either an URL call or a Page Break.
 */
public class ProxyDataRecord
// ================================================================================
{

    /**
     * Page break data type.
     */
     final public static int          DATA_TYPE_PAGE_BREAK = 2;        // page break



    /**
     * URL call data type.
     */
     final public static int          DATA_TYPE_HTTP_DATA = 10;        // recorded URL data



    /**
     * Modify / Update the content test configuration for the URL call.
     *
     * @param httpContentTest   the new content test configuration
     */
     public void setHttpContentTest(HttpContentTest httpContentTest)
     // ---------------------------------------------------------------------------
     {
          this.httpContentTest = httpContentTest;
     }



     /**
      * Get the unique ID of the item.
      *
      * @return    the unique ID of the item (thread-save timestamp)
      */
     public long getId()
     // ---------------------------------------------------------------------------
     {
          return id;
     }


    /**
     * Get the data type of this item.
     *
     * @return the data type of this item
     *
     * @see    #DATA_TYPE_PAGE_BREAK
     * @see    #DATA_TYPE_HTTP_DATA
     */
     public int getDataType()
     // ---------------------------------------------------------------------------
     {
          return dataType;
     }


     /**
      * Set the maximum acceptable response time. Note: this is only an informative value.
      *
```



```
     * @param maxAcceptableResponseTime the maximum acceptable response time in milliseconds,
     *                                  or -1 if no acceptable response time is configured.
     */
    public void setMaxAcceptableResponseTime(long maxAcceptableResponseTime)
    // ------------------------------------------------------------------------------------
    {
        this.maxAcceptableResponseTime = maxAcceptableResponseTime;
    }


    /**
     * Get the maximum acceptable response time. Note: this is only an informative value.
     *
     * @return    the maximum acceptable response time in milliseconds, or -1 if no acceptable
     *            response time is configured.
     */
    public long getMaxAcceptableResponseTime()
    // ------------------------------------------------------------------------------------
    {
        return maxAcceptableResponseTime;
    }


    /**
     * Mark the data record as non-executable (not executed during a load test).
     *
     * @param     markAsNonexecutable  true = mark as non-executable
     */
    public void setAsNonexecutable(boolean markAsNonexecutable)
    // ------------------------------------------------------------------------------
    {
        this.markAsNonexecutable = markAsNonexecutable;
    }


    /**
     * Get if the data record is marked as non-executable (not executed during the load test).
     *
     * @return true if the data record is marked as non-executable, else false (default value)
     */
    public boolean isMarkedAsNonexecutable()
    // ------------------------------------------------------------------------------
    {
        return markAsNonexecutable;
    }


   /**
    * Get if this item is a Page Break.
    *
    * @return true if this item is a Page Break
    *
    * @see    #DATA_TYPE_PAGE_BREAK
    */
    public boolean isDataTypePageBreak()
    // ------------------------------------------------------------------------------
    {
        return (dataType == DATA_TYPE_PAGE_BREAK);
    }


   /**
    * Get if this item is an URL call.
    *
    * @return true if this item is an URL call
    *
    * @see    #DATA_TYPE_HTTP_DATA
    */
    public boolean isDataTypeHttpData()
    // ------------------------------------------------------------------------------
    {
        return (dataType == DATA_TYPE_HTTP_DATA);
    }


   /**
    * Get the data of a Page Break.
    *
    * @return  the data of a Page Break, or null if this item is not a page break
    *
    */
    public HttpPageBreak getHttpPageBreak()
    // ------------------------------------------------------------------------------
```



```
    {
        return httpPageBreak;
    }


    /**
     * Set the data for a Page Break.
     *
     * @param httpPageBreak the data for a Page Break
     *
     */
    public void setHttpPageBreak(HttpPageBreak httpPageBreak)
    // ------------------------------------------------------------------------
    {
        this.httpPageBreak = httpPageBreak;
    }


    /**
     * Get if data of a HTTP request are available.
     *
     * @return true if data of a HTTP request are available
     */
    public boolean hasHttpRequest()
    // ------------------------------------------------------------------------
    {
        return (httpRequest != null);
    }


    /**
     * Get the data of a HTTP request.
     *
     * @return  the data of a HTTP request, or null if no such data are available
     */
    public HttpRequest getHttpRequest()
    // ------------------------------------------------------------------------
    {
        return httpRequest;
    }


    /**
     * Get if data of a HTTP response are available.
     *
     * @return true if data of a HTTP response are available
     */
    public boolean hasHttpResponse()
    // ------------------------------------------------------------------------
    {
        return (httpResponse != null);
    }


    /**
     * Get the data of a HTTP response.
     *
     * @return  the data of a HTTP response, or null if no such data are available
     */
    public HttpResponse getHttpResponse()
    // ------------------------------------------------------------------------
    {
        return httpResponse;
    }


    /**
     * Get if content test definition data are available.
     *
     * @return true if content test definition data are available
     */
    public boolean hasHttpContentTest()
    // ------------------------------------------------------------------------
    {
        return (httpContentTest != null);
    }


    /**
     * Get the content test definition.
     *
     * @return the content test method, or null if no content test method was set.
     */
```



```
    public HttpContentTest getHttpContentTest()
    // ------------------------------------------------------------------------
    {
        return httpContentTest;
    }


}   // end of class
```

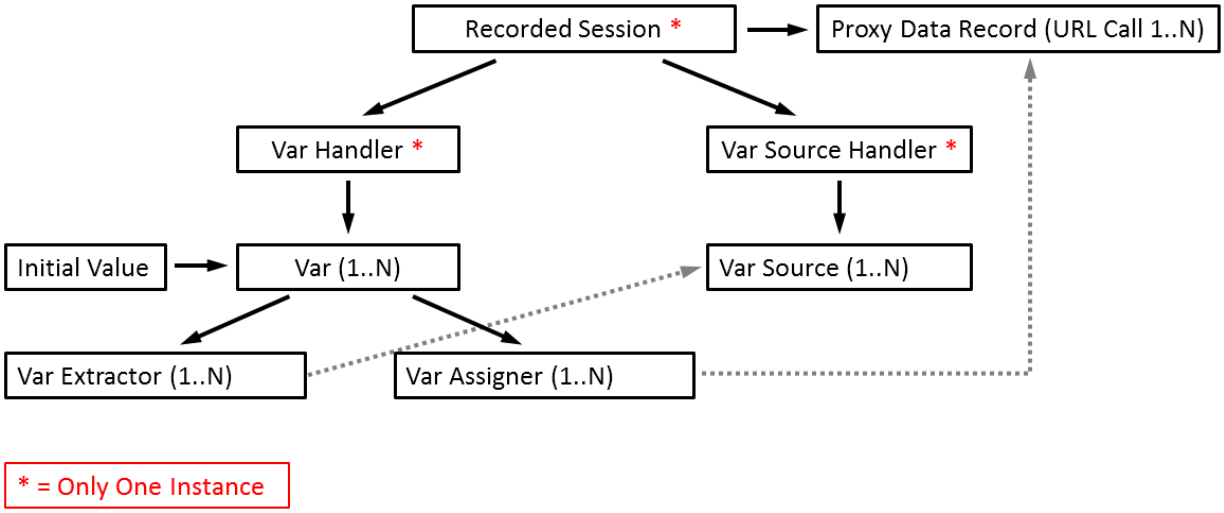

3.3.3 The Var Source Handler and the Var Handler

This subchapter contains information about declaring Proxy Recorder variables (so-called "Var"), declaring sources of values for variables (so-called "Var Source"), declaring extractors for values of variables (so-called "Var Extractor") and declaring targets for assigning the value of variables (so-called "Var Assigner").

There are 6 central data structures:

1. The "**Var Source Handler**" contains a set of all declared "Var Sources".
2. A "**Var Source**" is an object from which the value for a "Var" can be extracted
3. The "**Var Handler**" contains a set of all declared "Vars".
4. For each "**Var**", multiple "Var extractors" and "Var assigners" can be declared.
5. A "**Var Extractor**" is a bridge that connects a "Var Source" with the value of a "Var".
6. A "**Var Assigner**" is a bridge that connects the value of a "Var" to the value of a target-object.

Each recorded session contains only one "Var Source Handler" and only one "Var Handler". So you can imagine as they are static which is of course not really true.



3.3.3.1 Var Source(s)

The following types of Var Sources are supported:

- Class **ProxySnifferVarSourceConstantValue**: The value for the Var(s) is extracted from a simple value. This Var Source is normally not used because a Var can be directly initialized with a constant value.
- Class **ProxySnifferVarSourceLoadtestPlugin**: The value for the Var(s) is extracted from a "Load Test Plug-in".
- Class **ProxySnifferVarSourceRecordedUrl**: The value for the Var(s) is extracted from a recorded URL (ProxyDataRecord).
- Class **ProxySnifferVarSourceTextfile**: The value for the Var(s) is extracted from an "Input File".
- Class **ProxySnifferVarSourceUserInputField**: The value for the Var(s) is extracted from an "User Input Field".
- Class **ProxySnifferVarSourceInlineScript**: The value for the Var(s) is extracted from an "Inline Script".

Once you have created an instance of a "Var Source" type, you have to **add it to the "Var Source Handler"** by calling

```
<ProxySnifferVarSourceType> <Var Source Instance> = (ProxySnifferVarSourceType)
<ProxySnifferVarSourceHandler>.addVarSource(<Var Source Instance>)
```


3.3.3.2 Var(s)

Each Var has a scope that must be defined when the Var is the first time declared. Possible **scopes** are:

- Global
- User
- Loop
- Inner Loop

The "intrinsic" value of a Var is always a String. A Var can be directly **initialized** with one the following values:

- null (Java null – no value).
- A (constant) text.
- The counter of the simulated user (0, 1, 2, ..)
- The counter of the executed loop (0, 1, 2, ..). This value depends also on the scope of the Var.
- The counter of the executed inner loop (0, 1, 2, ..).
- The current time (elapsed milliseconds since January 1, 1970 GMT)
- The source host name used by the simulated user (useful if multiple source IP addresses are defined on the load generator).
- The source IP address used by the simulated user (useful if multiple source IP addresses are defined on the load generator).
- A formatted date and time that can also be some days in the past or in the future. The format as well as the time zone is configurable.
- A random number within a specific range.
- a Java system property

Once you have created an instance of a "Var", you have to **add it to the "Var Handler"** by calling

```
<ProxySnifferVar> <Var Instance> = <ProxySnifferHandler>.addVar(<Var Instance>)
```


3.3.3.3 Var Extractor(s)

The following Var Extractor types are supported to extract a value to a Var:

- from a HTTP redirect (URL)
- from a HTML form action
- from a HTML form target (URL)
- from a HTML form field (for example from a hidden form value)
- from an unique hyperlink (URL)
- from a text-line pattern of a HTTP response (data-protocol-independent Var Extractor)
- from a pattern of a HTTP redirect
- from the host name or from the path of a HTTP redirect
- from XML and from JSON data
- from a HTTP response header field
- from the HTTP status code
- from the content-data of a HTTP response (the whole content-data)
- from Google Protobuf data
- from an "Input File"
- from an "User Input Field"
- from a Java Session ID parameter
- from a "Load Test Plug-In" (output parameter)
- from an "Inline Script" (output parameter)
- from a constant value

Once you have created an instance of a "Var Extractor" type, you have to **add it to the corresponding "Var"** by calling

```
<ProxySnifferVarExtractorType> <Var Extractor Instance> = (ProxySnifferVarExtractorType)
<ProxySnifferVar>.addVarExtractor(<Var Extractor Instance>)
```


3.3.3.4 Var Assigner(s)

The following Var Assigner types are supported to assign a value from a Var:

- to a parameter value of a HTML form (HTTP request content)
- to a parameter value of a HTML "multipart" form (HTTP request content)
- to (replace) a text-pattern in the HTTP request content
- to the path of an URL (HTTP request)
- to the host name of a HTTP request
- to the TCP/IP port of a HTTP request
- to the protocol of a HTTP request ("http" or "https")
- to a field value of a HTTP request header field
- to (replace) a text-pattern in the HTTP request header fields
- to set the value of a CGI parameter in the HTTP request URL
- to (replace) a text-pattern in the HTTP request URL
- to (replace) a binary pattern in the HTTP request content data
- to set a value in XML request content data
- to set a value in Google Protobuf request content data
- to replace or set a value for a Java system property
- to set a value for a "Load Test Plug-In" (input parameter)
- to set a value for an "Inline Script" (input parameter)

Once you have created an instance of a "Var Assigner" type, you have to **add it to the corresponding "Var"** by calling

```
<ProxySnifferVarAssignerType> <Var Assigner Instance> = (ProxySnifferVarAssignerType)
<ProxySnifferVar>.addVarAssigner(<Var Assigner Instance>)
```


4 Advanced Steps - Offline Programming

All of the examples shown before in this handbook require that a ProxySniffer/ZebraTester process is started and running on your machine.

But it's also supported to make any modification to a recorded session offline, without that a ProxySniffer/ZebraTester process is needed. It's even supported to create whole new sessions from scratch – offline – by directly adding the page breaks and the URLs to an "empty" session.

For example, the support of the automatically generated load test programs for JUnit test cases was made by using offline programming. The following two subchapters contain the source code of some of these tools.

4.1 Source Code of CreateJUnitSession.java

```
import java.io.*;
import java.util.ArrayList;
import java.util.Vector;

import dfischer.proxysniffer.HttpPageBreak;
import dfischer.proxysniffer.HttpRequest;
import dfischer.proxysniffer.HttpResponse;
import dfischer.proxysniffer.ProxyDataDump;
import dfischer.proxysniffer.ProxyDataRecord;
import dfischer.proxysniffer.ProxySnifferSettings;
import dfischer.utils.LoadtestExternalResources;


/**
 * Command line tool to support the automation of JUnit load tests.
 * Create a session which contain one as non-executable marked URL and
 * contain all external resources required to execute the JUnit test case(s).
 *
 * @since V5.4-F, back-ported to V5.4-D
 */
public class CreateJUnitSession
// ================================================================================
{

    /**
     * No public constructor - all is static.
     */
    private CreateJUnitSession()
    // --- constructor ------------------------------------------------------------
    {
    }


    /**
     * Create a session which contain one as non-executable marked URL and
     * contain all external resources required to execute the JUnit test case(s). <BR>
     * Note: This method can called directly from a command line of a terminal.
     *
     * @param args  Argument[0]: The project name, or an empty string, <BR>
     *              Argument[1]: The name of the author, or an empty string, <BR>
     *              Argument[2]: A comment about the session, or an empty string,
     *              Argument[3]: The absolute or relative file path of the created session
(*.prxdat output file),
     *              Argument[4..n]: The absolute file paths of the external resources
     */
    public static void main(String[] args)
    // ------------------------------------------------------------------------------
    {
        ArrayList<String> externalResourceFiles = new ArrayList<String>();
        for (int x = 4; x < args.length; x++)
        {
            File f = new File(args[x]);
            if (!f.exists())
            {
                throw new IllegalArgumentException("file not exists: " + f.getPath());
            }
            externalResourceFiles.add(args[x]);
        }
```

```
        try
        {
            execute(args[0], args[1], args[2], args[3], externalResourceFiles);
            System.exit(0);
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.exit(1);
        }
    }


    /**
     * Create a session which contain one as non-executable marked URL and
     * contain all external resources required to execute the JUnit test case(s). <BR>
     * Note: This method can called directly by any (external) Java program.
     *
     * @param projectName   The project name, or an empty string
     * @param author        The name of the author, or an empty string
     * @param comment       A comment about the session, or an empty string
     * @param outputFile    The absolute or relative file path of the created session (*.prxdat
output file)
     * @param externalResourceFiles a list of absolute file paths of the external resources
     */
    public static void execute(String projectName, String author, String comment, String
outputFile, ArrayList<String> externalResourceFiles) throws IOException
    // ------------------------------------------------------------------------------------
    {
        if (!outputFile.endsWith(ProxySnifferSettings.PROXY_SNIFFER_DUMP_FILE_EXTENSION))
            throw new IllegalArgumentException("the output file extension must be " +
ProxySnifferSettings.PROXY_SNIFFER_DUMP_FILE_EXTENSION);

        // create an empty session
        ProxyDataDump proxyDataDump = new ProxyDataDump(projectName, author, comment);

        // add a page break
        HttpPageBreak pageBreak = new HttpPageBreak("Start Page", 0, null, 0);
        ProxyDataRecord pageBreakDataRecord = new ProxyDataRecord(-1, pageBreak);
        proxyDataDump.addProxyDataRecord(pageBreakDataRecord);

        // add a non-executed URL
        Vector<String> requestHeaderVector = new Vector<String>();
        requestHeaderVector.add("GET / HTTP/1.1");
        requestHeaderVector.add("Host: placeholder.noexec");
        long requestStartDate = System.currentTimeMillis();
        HttpRequest httpRequest = new HttpRequest("http://placeholder.noexec/",
requestHeaderVector, null, requestStartDate, -1, 0);
        ProxyDataRecord urlDataRecord = new ProxyDataRecord(-1, httpRequest);

        Vector<String> responseHeaderVector = new Vector<String>();
        responseHeaderVector.add("HTTP/1.1 200 OK");
        HttpResponse httpResponse = new HttpResponse(responseHeaderVector, null, 0, 0, 0,
requestStartDate + 10);
        urlDataRecord.setHttpResponse(httpResponse);

        urlDataRecord.setAsNonexecutable(true);
        proxyDataDump.addProxyDataRecord(urlDataRecord);

        // add external resources to session
        LoadtestExternalResources externalResources = proxyDataDump.getExternalResources();
        for (String externalResourceFile : externalResourceFiles)
        {
            externalResources.addLocalFileResource(externalResourceFile, true, false);
        }

        // store session on disk
        DataOutputStream dout = null;
        try
        {
            dout = new DataOutputStream(new FileOutputStream(outputFile));
            proxyDataDump.writeObject(dout);
        }
        finally
        {
            if (dout != null)
            {
                try { dout.close(); } catch (Exception e) {}
            }
        }
    }

}
```


4.2 Source Code of AddJUnitPlugin.java

```
import java.io.*;
import java.util.Vector;

import dfischer.proxysniffer.ProxyDataDump;
import dfischer.proxysniffer.ProxyDataRecord;
import dfischer.proxysniffer.ProxySnifferVar;
import dfischer.proxysniffer.ProxySnifferVarAssignerLoadtestPlugin;
import dfischer.proxysniffer.ProxySnifferVarExtractorUserInputField;
import dfischer.proxysniffer.ProxySnifferVarHandler;
import dfischer.proxysniffer.ProxySnifferVarSourceHandler;
import dfischer.proxysniffer.ProxySnifferVarSourceLoadtestPlugin;
import dfischer.proxysniffer.ProxySnifferVarSourceTextfile;
import dfischer.proxysniffer.ProxySnifferVarSourceUserInputField;
import dfischer.prxutils.GenericPluginClassLoader;
import dfischer.utils.LoadtestPluginFixedUserInputField;
import dfischer.utils.LoadtestPluginInterface;


/**
 * Command line tool to support the automation of JUnit load tests.
 * Add a JUnit plug-in to a session (*.prxdat file).
 *
 * @see GenerateJUnitPluginCode
 * @see CreateJUnitSession
 *
 * @since V5.4-F, back-ported to V5.4-D
 */
public class AddJUnitPlugin
// ============================================================================================
{
    /**
     * No public constructor - all is static.
     */
    private AddJUnitPlugin()
    // --- constructor ------------------------------------------------------------------
    {
    }


    /**
     * Add a JUnit plug-in to a session at the first non-executable marked URL. <BR>
     * Note: This method can called directly from a command line of a terminal.
     *
     * @param args  Argument[0]: The absolute or relative file path of the session (*.prxdat
file) which contains already all declarations of the required external resources, <BR>
     *              Argument[1]: The absolute or relative file path of the compiled JUnit plug-
in (*.class file)
     */
    public static void main(String[] args)
    // ------------------------------------------------------------------------------------
    {
        try
        {
            execute(args[0], args[1]);
            System.exit(0);
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.exit(1);
        }
    }


    /**
     * Add a JUnit plug-in to a session at the first non-executable marked URL. <BR>
     * Note: This method can called directly by any (external) Java program.
     *
     * @param sessionFile The absolute or relative file path of the session (*.prxdat file)
which contains already all declarations of the required external resources, <BR>
     * @param pluginFile  The absolute or relative file path of the compiled JUnit plug-in
(*.class file)
     */
    public static void execute(String sessionFile, String pluginFile) throws Exception
    // ------------------------------------------------------------------------------------
    {
        DataInputStream din = null;
        DataOutputStream dout = null;
```



```
        try
        {
            // load session
            din = new DataInputStream(new FileInputStream(sessionFile));
            ProxyDataDump proxyDataDump = new ProxyDataDump();
            proxyDataDump.readObject(din);
            din.close();
            din = null;

            // find the data record ID of the non-executed URL
            long urlDataRecordId = -1;
            Vector<ProxyDataRecord> recordVector = proxyDataDump.getProxyData();
            for (ProxyDataRecord dataRecord : recordVector)
            {
                if ((dataRecord.getDataType() == ProxyDataRecord.DATA_TYPE_HTTP_DATA) &&
dataRecord.isMarkedAsNonexecutable())
                {
                    urlDataRecordId = dataRecord.getId();
                    break;
                }
            }
            if (urlDataRecordId == -1)
                throw new IllegalArgumentException("invalid session, non-executable marked URL
missing");

            // load plug-in
            GenericPluginClassLoader genericPluginClassLoader = new
GenericPluginClassLoader(proxyDataDump.getExternalResources());
            Class c = genericPluginClassLoader.defineClass(new File(pluginFile));
            if (c == null)
                throw new IllegalArgumentException("unable to load plug-in");

            boolean pluginValid = genericPluginClassLoader.isValid();
            if (!pluginValid)
                throw new IllegalArgumentException("invalid plug-in");

            String[] pluginInputParameter = genericPluginClassLoader.getInputParameterLabels();
            String[] pluginOutputParameter =
genericPluginClassLoader.getOutputParameterLabels();
            LoadtestPluginFixedUserInputField[] userInputFields =
genericPluginClassLoader.getFixedUserInputFields();

            // dump plug-in data
            /*
            System.out.println(">>> implements additionally Loadtest Plugin Interface = " +
genericPluginClassLoader.implementsLoadtestPluginInterface());
            System.out.println(">>> Plugin Type = " +
genericPluginClassLoader.getPluginType());
            System.out.println(">>> Plugin Name = \"" +
genericPluginClassLoader.getPluginName()+ "\"");
            System.out.println(">>> Plugin Description = \"" +
genericPluginClassLoader.getPluginDescription() + "\"");
            for (int x = 0; x < pluginInputParameter.length; x++)
                System.out.println(">>> Plugin Input Parameter [" + x + "] = " +
pluginInputParameter[x]);
            for (int x = 0; x < pluginOutputParameter.length; x++)
                System.out.println(">>> Plugin Output Parameter [" + x + "] = " +
pluginOutputParameter[x]);
            for (LoadtestPluginFixedUserInputField userInputField : userInputFields)
            {
                System.out.println(">>> Input Field Var Name = " +
userInputField.getVarName());
                System.out.println(">>> Input Field Default Value = " +
userInputField.getDefaultValue());
            }
            */

            // define a global var for the user input field
            ProxySnifferVarHandler varHandler = proxyDataDump.getVarHandler();
            ProxySnifferVarSourceHandler varSourceHandler =
proxyDataDump.getVarSourceHandler();

            ProxySnifferVar inputFieldVar = new ProxySnifferVar(ProxySnifferVar.SCOPE_GLOBAL,
userInputFields[0].getVarName());
            inputFieldVar = varHandler.addVar(inputFieldVar);

            // define the user input field
            ProxySnifferVarSourceUserInputField varSourceUserInputField = new
ProxySnifferVarSourceUserInputField(inputFieldVar.getName(), pluginInputParameter[0],
userInputFields[0].getDefaultValue());
            varSourceUserInputField = (ProxySnifferVarSourceUserInputField)
varSourceHandler.addVarSource(varSourceUserInputField);
```



```
            // create the var extractor for the user input field
            ProxySnifferVarExtractorUserInputField varExtractorUserInputField = new
ProxySnifferVarExtractorUserInputField(inputFieldVar.getName(),
varSourceUserInputField.getUniqueKey());
            varExtractorUserInputField = (ProxySnifferVarExtractorUserInputField)
inputFieldVar.addVarExtractor(varExtractorUserInputField);

            // define the plug-in
            ProxySnifferVarSourceLoadtestPlugin varSourceLoadtestPlugin = new
ProxySnifferVarSourceLoadtestPlugin(varSourceHandler.getTotalPluginNumber() + 1,

genericPluginClassLoader.getClassName(),

genericPluginClassLoader.getAllowedConstructScope(),

genericPluginClassLoader.getAllowedExecScope(),

false,

urlDataRecordId);

varSourceLoadtestPlugin.setPluginCode(genericPluginClassLoader.getBinaryClassData(),
genericPluginClassLoader.getFileLastModified());    // and store also class code
            varSourceLoadtestPlugin = (ProxySnifferVarSourceLoadtestPlugin)
varSourceHandler.addVarSource(varSourceLoadtestPlugin);

            // create the var assigner for the plug-in
            ProxySnifferVarAssignerLoadtestPlugin varAssignerLoadtestPlugin = new
ProxySnifferVarAssignerLoadtestPlugin(inputFieldVar.getName(), urlDataRecordId,
varSourceLoadtestPlugin.getUniqueKey(), 0);
            varAssignerLoadtestPlugin = (ProxySnifferVarAssignerLoadtestPlugin)
inputFieldVar.addVarAssigner(varAssignerLoadtestPlugin);

            // store session on disk
            dout = new DataOutputStream(new FileOutputStream(sessionFile));
            proxyDataDump.writeObject(dout);

        }
        finally
        {
            if (din != null)
            {
                try { din.close(); } catch (Exception e) {}
            }

            if (dout != null)
            {
                try { dout.close(); } catch (Exception e) {}
            }
        }
    }


}
```


5 Manufacturer

Ingenieurbüro David Fischer AG, Switzerland | A company of the Apica Group

Product Web Site: http://www.zebratester.com

Apica AB: http://www.apicasystem.com