

XPages Toolbox

1. Introduction

The XPages Toolbox is a set of web based tools for the XPages developer. It can be used to optimize and debug XPages applications. It currently features the following modules:

- CPU profiler
- Backend classes profiler
- Memory Profiler
- Runtime monitoring
- Java logging groups management

It is based on an open architecture and thus can be extended with extra modules.

Security Warning: The memory profiler can display and store files on the server that contain information coming from some live XPages sessions. You should be careful when profiling memory by:

- > Not using the profiler on production servers
- > Removing the toolbox NSF as soon as you're done with it
- > Protecting its access using ACLs

To prevent any accident, access to sensitive information requires the `java.policy` to explicitly enable the XPagesToolbox application.

2. Installing the toolbox

Because it uses advanced XPages debugging capability, the Toolbox requires a Notes/Domino 8.5.2 runtime at a minimum. It is provided as an NSF that should be installed in the Domino data directory, as any other application.

To fully benefit from all the features, 2 extra steps should be taken:

- A Java profiler agent should be installed in the JVM
This agent is used by the memory profiler to compute the size of objects, using the Java instrumentation API (<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html>). If this agent is not installed, the size being returned by the memory profiler will always be '0'
- Java security policy
Because the profiler can access some sensitive data stored in the other sessions, it must be explicitly authorized in the Java policy file. Not doing that will lead to Java Security Exceptions.

1. Installing the java agent

The java agent is provided as a jar file (`XPagesProfilerAgent.jar`) that must be copied to the `<domino>/xsp` directory:



The jar source file is also provided for your information. Do not change the main class name, nor the field name, as this is looked up by the XPages runtime.

Then, this agent should be activated by passing information to the JVM when started. This is done by copying the `XspProfilerOptionsFile.txt` file to your domino main directory, and by inserting the following lines into `Notes.ini` (assuming that domino is installed in `c:\Domino`):

```
; XPages Profiler
JavaOptionsFile=c:\Domino\XspProfilerOptionsFile.txt
```

If the agent is correctly installed, the Domino console should display the following message when the http task is started:

```
> *** Activating IBM XPages profiler agent
```

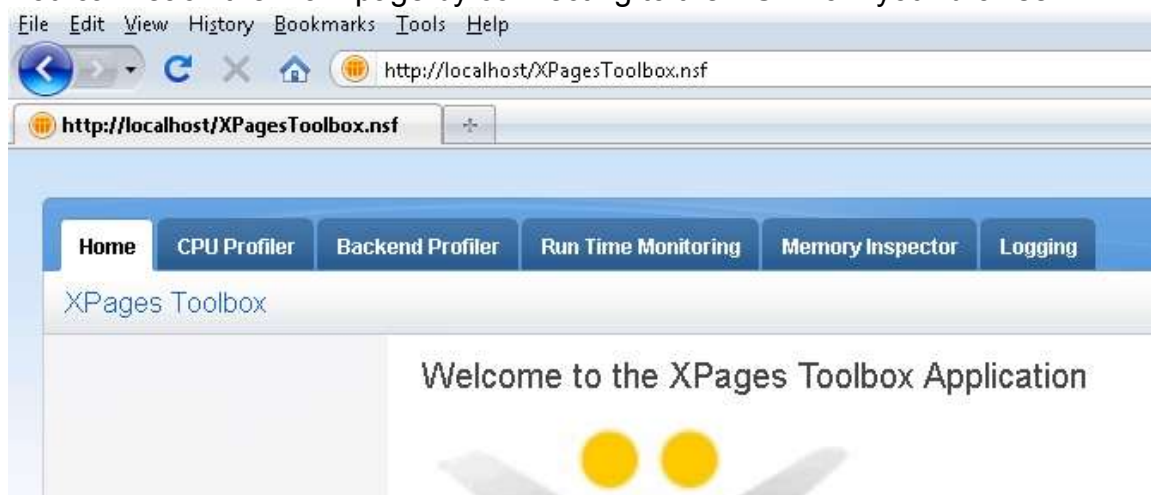
2. Updating the java policy file

The java policy file is located into the `<domino>/jvm/lib/security` directory. Assuming that the NSF is not installed into a sub directory, then the following lines should be added at the end of the `java.policy` file:

```
grant codeBase "xspnsf://server:0/xpagestoolbox.nsf/script/-" {
    permission com.ibm.designer.runtime.domino.adapter.security.AdminPermission
    "AdminPermission.debug";
};
```

Hey, congratulations, you're now ready to go!

You can reach the main page by connecting to the NSF from your browser:



3.Using the toolbox

The Toolbox is featuring different modules, accessible through the top tab bar

1. CPU profiler

The CPU profiler is a high level profiler recording the XPages activity and gathering the time spent in the different parts of the XPages (JSF lifecycle, JavaScript code, custom code...)

On Windows machines, the profiler is using native methods to measure either the elapsed CPU time (the time actually consumed by the thread when running) or the wall time (the whole time spent by the thread, including sleeping/waiting time). On other OS, it only measures the wall time as reported by the JRE System methods.

The profiler starts to gather CPU usage information in memory as soon as it is started, by clicking one of the Start buttons:



When the 'Save Profiler' or 'Stop Profiler' are clicked, then a set of Domino documents are created in the database, after deleting the existing documents.

Once in the database, this information can be accessed using one of the predefined views:



Of course, new views can be created. And even the way the documents are created can be customized. For example, we can choose to ignore some not significant entries (showing less CPU usage than a predefined threshold).

The Java code is located in the JavaSource folder in the NSF.

Note that 'Specific time' means the time spent in the profiled block minus the other profiled blocks it called.

The information gathered by the profiler stays in memory until a click to the 'Reset Profiler' button, or the HTTP task restarted. Click the button to start a brand new session.

Monitoring custom code

The XPages runtime currently monitors some well known operations, like JSF phases or data access. But developers can also monitor their own blocks in either Java or JavaScript.

Each block is defined by a type (a string constant) and an optional content.

The content is used to categorize different instances of the same type. For example, the type can be “JavaScript expression” and the content is the JavaScript code itself.

Monitoring JavaScript blocks

The XPages server side JavaScript interpreter is featuring an extra keyword, defining a block to monitor:

```
__profile(<type>[, content]) {  
    .....  
}
```

Here is an example monitoring a custom block:

```
__profile("MyCode", "myblock1") {  
    java.lang.Thread.sleep(1000)  
}
```

And the result as displayed by the profiler module (using the Wall time option, to take the sleeping time into account):

Type	Content	Count	Total time
▼ XPages Request	/xspetest.nsf/JavaScriptProfile	1	1018
▼ JavaScript expression	__profile("MyCode", "myblock1")	1	1015
	"MyCode"	1	1015

Monitoring Java blocks

As we cannot extend the Java language, this is a little bit more complex. But here is a sample piece of code, to use as a best practice, that shows how to monitor custom Java blocks:

```
import com.ibm.commons.util.profiler.Profiler;  
import com.ibm.commons.util.profiler.ProfilerAggregator;  
import com.ibm.commons.util.profiler.ProfilerType;  
...  
private static final ProfilerType profilerCustom = new ProfilerType("MyJavaType");  
public void myMethod() {  
    if(Profiler.isEnabled()) {  
        ProfilerAggregator agg=Profiler.startProfileBlock(profilerCustom, null);  
        long startProfiler=Profiler.getCurrentTime();  
        try {  
            _myMethod();  
        } finally {  
            Profiler.endProfileBlock(agg, startProfiler);  
        }  
    } else {  
        _myMethod();  
    }  
}  
private void _myMethod() {  
    try {  
        java.lang.Thread.sleep(1000);  
    } catch (InterruptedException ex){}}
```

The goal is to minimize the impact of the profiler when not activated. That's the reason why the actual code is isolated in `_myMethod()`, with a straight call from `myMethod()` when the profiler is not activated.

The profiler type is defined as part of the static `ProfilerType` object, while the optional content is passed to the `startProfileBlock()` method (see the empty content below as we passed null to `startProfileBlock`).

Here is the result:

JavaScript expression	requestBean.myMethod()	1	1009
MyJavaType		1	1002

2. Backend classes profiler

This module gathers, per request, how the backend classes are being called and the time spent in each call. Note that it only measures the wall time.

When enabled, one document is created per XPages request. Each document contains the information within a rich text field:

Previous 1 Next

Date	Subject
Sat Apr 17 04:49:40 CST 2010	renodisc.nsf,/allDocuments.xsp

[Remove All Documents](#) [Refresh](#)

04/17/2010 04:49:40 AM ZE8
Elapsed time: 2355 msec
Methods profiled: 47
Total measured time: 93 msec

Class	Method	Operation	Calls	Time
View	GetAllEntriesByKey		1	62
View	CreateViewNav		1	31
ViewEntry	IsValid	Get	28	0
ViewEntry	GetRead		26	0
ViewColumn	IsResortAscending	Get	16	0
ViewEntry	IsDocument	Get	13	0

3. Runtime monitoring

Runtime monitoring is used to grab a high level view of the memory being used by the JVM., as well as the number of active modules (NSFs) and sessions.

Runtime snapshots can be taken at a regular interval, and then saved into the database (one document per snapshot). Actually, the thread gathering the information cannot save the documents for security reasons. As a result, the documents are saved when the user presses 'Refresh' in the UI or simply refreshes/redispays the page.

Take Snapshot Now!

Start Monitoring Stop Monitoring

Started

Sampling Interval (seconds)

Refresh

Previous 1 Next

Date/time	Modules Count	Sessions Count	Total Memory	Free Memory	Max Memory
Apr 17, 2010 5:16:24 AM	2.0	2.0	41.0	21.0	268.0
Apr 17, 2010 5:16:22 AM	2.0	2.0	41.0	21.0	268.0
Apr 17, 2010 5:16:21 AM	2.0	2.0	41.0	21.0	268.0

Refresh

Remove All Documents

4. Memory profiler

The memory profiler is used to show the memory being used by the XPages runtime, categorized by applications (NSFs) and opened sessions. At anytime, a snapshot of the memory being used can be taken. This creates an XML document, stored in a server temporary directory, which can be accessed from the browser or from the file explorer on the server. The XML format had been chosen because it can easily be processed.

2 options are available for the snapshots:

- a mini dump, that only contains the aggregated size of objects. This is faster and generates small XML files
- a full dump, which dumps each object along with its size. This provides a detailed report on the memory usage, at the expense of a big XML document.

Note: object sizes are just approximations, as there is no reliable way to know if an object is shared or not. The XPages runtime tries to do its best here, by finding the strings that are interned, by skipping objects that are supposed to be shared etc... But it might not be accurate, and should be interpreted cautiously.

A system JVM dump is also available. In this case, it generates a dump file within the Domino directory, and it should be analyzed using one of the existing tools, like the Eclipse Memory Analyzer (<http://www.eclipse.org/mat/>)

5. Java Logging

XPages is taking advantage of the Java Logging API to trace what is happening in the runtime. The logging levels are initially defined in a file located within the jvm directory:

```
<domino>/jvm/lib/logging.properties
```

The toolbox allows you to dynamically see what are the enabled groups and change their level on the fly, without having to restart the JVM.

Note that the toolbox UI only shows the groups that have been created at runtime by the java code. If you want to add you own logging groups, then it is a good practice to get them created as soon as possible, by putting all of them in a single class loaded early. This is what the XPages runtime does.

4. Extending the XPages toolbox

As said earlier, this toolbox is meant to be extended over time, mainly by enhancing the existing functionality, or by providing new capability through new main tabs.

The UI is based on IBM OneUI V2, using the excellent XPages Framework, also available from openNTF (<http://www.openntf.org/Projects/pmt.nsf/0/B5BCFFCB32053E588625765B005846BF>).

ENJOY!