Tips and Tricks

Explanation

This page is dedicated to things that we haven't found a better place for, but nonetheless people may need to know how to do =]

Getting the URL of the current page

```
#include "DOM/Window.h"
std::string location = m_host->getDOMWindow()->getLocation();
```

This value is considered safe to rely upon but keep in mind that the browser can be tricked by DNS Spoofing, and XSS attacks can inject script from other domains.

Getting the value of GET variables from the query string

```
#include "DOM/Window.h"
#include "URI.h"
FB::URI loc = FB::URI::fromString(m_host->getDOMWindow()->getLocation());
// Get "id" variable from querystring
std::string id = loc.query_data["id"];
```

Getting the path and filename of the plugin (DLL, .so, or .dylib)

```
// From inside your Plugin class (that extends PluginCore)
std::string MyPlugin::getFilesystemPath()
{
    return m_filesystemPath;
}
```

Using `printf(.)`-style debugging

Get Firebug for Firefox or use Chrome's built-in Console. The function you need is:
void FB::BrowserHost::htmlLog (const std::string& str)[virtual]

Example:

```
void MyPluginAPI::do_something()
{
    ...
    m_host->htmlLog("seems to work.");
    ...
}
```

The message will appear in the JavaScript console. Please note that htmlLog(.) is really slow and should not be used in release builds and has been reported to not work on IE.

Storing and using JSAPI pointers

See FireBreath Tips: Dealing with JSAPI objects

Keeping your projects outside of the FireBreath directory

See FireBreath Tips: Working with Source Control (it's covered as part of this short article)

Keeping your FireBreath project in Source Control

See FireBreath Tips: Working with Source Control

Building your plugin on multiple systems

The main issue here that seems to confuse people is that the build/ directory should not be copied between systems!

See FireBreath Tips: Working with Source Control

Drawing and handling WINPROC events on Windows

See FireBreath Tips: Drawing on Windows

MacOSX: Check if event/draw model is available:

```
if( win->validType<FB::PluginWindowMacCocoaCA>())
{

   // got Cocoa + CoreAnimation
}
if( win->validType<FB::PluginWindowMacCarbonCG>())
{

   // got Carbon + CoreGraphics
}
```

Using threads and javascript callbacks to make blocking system calls safe

See FireBreath Tips: Asynchronous Javascript Calls

Building a plugin without drawing support on Linux (removes gtk-dev dependency)

Starting with version v1.4 you can build your plugin without drawing support by adding the following to your PluginConfig.cmake:

```
set(FB_GUI_DISABLED 1)
```

Building a plugin without drawing support on Mac OSX

Starting with version v1.4 you can build your plugin without drawing support by adding the following to your PluginConfig.cmake:

```
set(FB_GUI_DISABLED 1)
set(FBMAC_USE_CARBON 0)
set(FBMAC_USE_COCOA 0)
set(FBMAC_USE_QUICKDRAW 0)
set(FBMAC_USE_COREGRAPHICS 0)
set(FBMAC_USE_COREANIMATION 0)
```

Building FireBreath on Windows using Visual Studio Express edition

See Building with Visual Studio Express

Firefox 4 unpack option when bundling a plugin in an xpi

New in Firefox 4 A true or false value that tells the application whether the extension requires its files be unpacked into a directory in order to work or whether the extension can be loaded direct from the XPI. In versions before Gecko 2.0 all extensions were unpacked, in Gecko 2.0 and later the default is to not unpack. If an extension includes the following then it must request unpacking:

- Binary XPCOM components
- DLLs loaded with ctypes
- Search plugins
- Dictionaries
- Window icons

See install manifest unpack option

MAC OS X: Disabling indexing in XCode 4

If you haven't big amount of RAM (< 8 GB), probably you will get lags during indexing your Firebreath project. Currently we don't know exactly why it's happening, but probably it's due to indexing templates in boost library by clang. One day this bug will be fixed by clang developers. If you can't upgrade your RAM and want to disable indexing, you may just type this in your Terminal:

```
chmod -R +w ~/Library/Developer/Xcode/DerivedData/
```

It will remove write permissions of your indexing directory and Xcode 4 will not be able to write there and indexing will stop automatically. This has not been tested on XCode 4.3 and later!

Blitting an pixel buffer to your plugin window in a platform agnostic way

There is an example of how GradeCam does this that can be found in this gist.

Modal Dialogs

Modal dialogs should never been used on the main thread. The following is a code example of using modal dialogs on windows and mac on an alternate thread. Note that there is some question as to whether doing this on Mac is actually a good idea or not.

https://gist.github.com/1368648

Finding a safe place for file-based logging

On windows vista and 7 IE launches all plugins in protected mode when UAC is on, which prevents you from writing to most places on the drive. You can get access to LocalLow (or similar places on other systems). The following code sample works on Windows and Mac and is a great example of setting up a logfile for log4cplus.

https://gist.github.com/1289031

Rendering a block of pixel data to the screen

Rendering raw pixel data to the screen is a pretty common use-case. Here are simple (non-accelerated) examples for both Mac and Windows:

- Mac/CoreGraphics (CoreGraphics is always windowless)
    - https://gist.github.com/1099740
- Windows (both windowless and windowed)
    - https://gist.github.com/1068352

Using HTTPService to create an embedded web server in the plugin

Not many need this, but for those who need it here is a code sample of setting up a simple embedded web server on your plugin. Note that you need to add the HTTPService firebreath library to use this.

https://gist.github.com/751687

MacOS X: CMake arguments to support 10.4

To deploy your plugin on MacOS X 10.4, add the following arguments to the end of your "prepmac.sh" command line:

```
-D CMAKE_OSX_ARCHITECTURES="ppc;i386" -D CMAKE_OSX_DEPLOYMENT_TARGET=10.4
```

CMAKE_OSX_ARCHITECTURES tells Xcode to compile for 32-bit ppc and i386, and not for 64-bit x86-64. CMAKE_OSX_DEPLOYMENT_TARGET tells Xcode to compile for 10.4 as the minimum acceptable OS version.

If you also want to change the SDK that you use to 10.4, you'll need to add the following:

```
-D CMAKE_OSX_SYSROOT="/Developer/SDKs/MacOSX10.4u.sdk" -D CMAKE_XCODE_ATTRIBUTE_GCC_VERSION="4.0"
```

CMAKE_OSX_SYSROOT tells Xcode to change SDKs. CMAKE_XCODE_ATTRIBUTE_GCC_VERSION tells Xcode to use gcc-4.0 instead of the default compiler; newer versions of the compiler are not compatible with the 10.4u SDK. Note that the CMake documentation says to change the compiler version using "-D CMAKE_C_COMPILER=gcc-4.0 -D CMAKE_CXX_COMPILER=g++-4.0", but this doesn't work.

You'll need to be careful not to use operating system features not supported on 10.4.

Note that there is a space after the "-D".

MacOS X: CMake code to install plugin during build

The documentation discusses linking the plugin to the build location. Xcode can also be configured to install (copy) the binary automatically after each build via the following CMake code, which should be added to the end of your <Project>/Mac/projectDef.cmake file. Select the appropriate "releasePlugin" call to install either per-user or globally.

```
# Copy plugin to Plug-Ins directory:
function(releasePlugin projectName pathToPlugin releaseDirectory)
 ADD_CUSTOM_COMMAND(
 TARGET ${PROJECT_NAME}
 POST_BUILD
 COMMAND mkdir -p ${releaseDirectory}/${projectName}.plugin
 )
 ADD_CUSTOM_COMMAND( TARGET ${PROJECT_NAME}
 POST_BUILD
 COMMAND cp -pr ${pathToPlugin} ${releaseDirectory}
 )
endfunction()

# Current output
set(PBIN "${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_CFG_INTDIR}/${PROJECT_NAME}.plugin")

# Uncomment one of the following releasePlugin() calls to install the plugin

# Copy plugin to ~/Library/Internet Plug-Ins (single user)
# releasePlugin("${PROJECT_NAME}" "${PBIN}" "~/Library/Internet Plug-Ins")

# Copy plugin to /Library/Internet Plug-Ins (all users)
# releasePlugin("${PROJECT_NAME}" "${PBIN}" "/Library/Internet Plug-Ins")
```

Visual Studio 2010: How to use pre-compiled headers for the plugin project

While some of the core modules of FireBreath already use pre-compiled headers ("PCH"), the generated plugin project does not do so. Here's a way how to setup a plugin project to use PCH:

- Create a header file that includes often used modules (Windows API, FireBreath core etc.), that provides generally used macros, typedef's etc. VS2010 defaults to the name "stdafx.h", so that's a good choice for your project (but you can use any other name if you have other preferences).
- Create a corresponding source file named "stdaxf.cpp" (or use the name that matches your choice) which consists of exactly one line of code:

stdafx.cpp

```
// just include the header file to create PCH
 #include "stdafx.h"

 // End Of File //
```

- Make sure that each source file of your plugin project starts with "#include stdafx.h" as first line of code (one exception s.below).
- Now the file "CMakeLists.txt" and "Win/projectDef.cmake" of your plugin project must be modified. Here are examples that work as expected (allthough there may be other ways to achieve the same result - working PCH):

CMakeLists.txt

```
#/*********************************************************\
#
# Auto-generated CMakeLists.txt for the Plugin project
#
#\*********************************************************/

# Written to work with cmake 2.6
cmake_minimum_required (VERSION 2.6)
set (CMAKE_BACKWARDS_COMPATIBILITY 2.6)
Project(${PLUGIN_NAME})

# get all source files
file (GLOB SOURCEFILES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR}
 [^.]*.cpp
 [^.]*.cxx
 )

# get all header files
file (GLOB HEADERFILES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR}
 [^.]*.h
 [^.]*.hxx
 [^.]*.hpp
 )

include_directories(${PLUGIN_INCLUDE_DIRS})

# Generated files are stored in ${GENERATED} by the project configuration
SET_SOURCE_FILES_PROPERTIES(
 ${GENERATED}
 PROPERTIES
 GENERATED 1
 )

# set "filters" for the project file
SOURCE_GROUP("1 Header files" FILES ${HEADERFILES})
SOURCE_GROUP("2 Sources files" FILES ${SOURCEFILES})
SOURCE_GROUP("Y Generated files" FILES ${GENERATED})

# set combined list of sources
SET( SOURCES
 ${SOURCEFILES}
 ${HEADERFILES}
 ${GENERATED}
 )

# This will include Win/projectDef.cmake, X11/projectDef.cmake, Mac/projectDef
# depending on the platform
include_platform()
```

Note: You can change or omit the "SOURCE_GROUP" definitions as you want.

Win/projectDef.cmake

```
#/*********************************************************\
# Auto-generated Windows project definition file for the
# Plugin project
#\*********************************************************/

# Windows template platform definition CMake file
# Included from ../CMakeLists.txt

# remember that the current source dir is the project root; this file is in Win/
file (GLOB PLATFORM RELATIVE ${CMAKE_CURRENT_SOURCE_DIR}
 Win/[^.]*.cpp
 Win/[^.]*.h
 )

# use this to add preprocessor definitions
add_definitions(
 /D "_ATL_STATIC_REGISTRY"
 /D "__YOUR_SPECIFIC_DEFINE"
)

# get PCH related files
file (GLOB PCH RELATIVE ${CMAKE_CURRENT_SOURCE_DIR}
 stdafx*
 )

# get IID instantation file
set (IID "iid.cpp")

# get (C)makefiles etc.
file (GLOB MAKEFILES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR}
 [^.]*.cmake
 Win/[^.]*.cmake
 CMake*.txt
 )

# ---!!!--- important: make sure that PCH-related files and IID instantiating source are NOT part of ${SOURCES}
LIST(REMOVE_ITEM SOURCES ${PCH})
LIST(REMOVE_ITEM SOURCES ${IID})

# set "filters" for the project file
SOURCE_GROUP("3 PCH files" FILES ${PCH})
SOURCE_GROUP("4 IID instantiation files" FILES ${IID})
SOURCE_GROUP("5 Windows specific files" FILES ${PLATFORM})
SOURCE_GROUP("X Makefiles" FILES ${MAKEFILES})

# extend combined list of sources
set (SOURCES
 ${SOURCES}
 ${WIXINS}
 ${PLATFORM}
 ${MAKEFILES}
 )

# set PCH
MACRO(ADD_MSVC_PRECOMPILED_HEADER PrecompiledHeader PrecompiledSource SourcesVar)
 GET_FILENAME_COMPONENT(PrecompiledBasename ${PrecompiledHeader} NAME_WE)
 SET(__PrecompiledBinary "${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_CFG_INTDIR}/${PROJECT_NAME}.pch")

 SET_SOURCE_FILES_PROPERTIES(${PrecompiledSource}
   PROPERTIES COMPILE_FLAGS "/Yc\"${PrecompiledBasename}.h\" /Fp\"${__PrecompiledBinary}\" -Zm160"
   OBJECT_OUTPUTS "${__PrecompiledBinary}")

 foreach(CURFILE ${${SourcesVar}})
   GET_FILENAME_COMPONENT(CURFILE_EXT ${CURFILE} EXT)
   GET_FILENAME_COMPONENT(CURFILE_NAME ${CURFILE} NAME)
   if (CURFILE_EXT STREQUAL ".cpp" AND NOT CURFILE_NAME STREQUAL PrecompiledBasename)
     SET_SOURCE_FILES_PROPERTIES(${CURFILE}
       PROPERTIES COMPILE_FLAGS "/Yu\"${__PrecompiledBinary}\" /FI\"${__PrecompiledBinary}\"
```

```
/Fp\"${__PrecompiledBinary}\" -Zm160"
       OBJECT_DEPENDS "${__PrecompiledBinary}")
   endif()
 endforeach()
 # Add precompiled header to SourcesVar
 LIST(APPEND ${SourcesVar} ${PrecompiledSource})
 LIST(APPEND ${SourcesVar} ${PrecompiledHeader})
ENDMACRO(ADD_MSVC_PRECOMPILED_HEADER)

MACRO(SET_IID_INSTANTIATION SourcesVar IIDSource)
 SET_SOURCE_FILES_PROPERTIES(${IIDSource} PROPERTIES COMPILE_FLAGS "/Y-")
 LIST(APPEND ${SourcesVar} ${IIDSource})
ENDMACRO(SET_IID_INSTANTIATION)

# activate PCH
ADD_MSVC_PRECOMPILED_HEADER("${CMAKE_CURRENT_SOURCE_DIR}/stdafx.h" "${CMAKE_CURRENT_SOURCE_DIR}/stdafx.cpp"
SOURCES)

# set IID instantiation
SET_IID_INSTANTIATION(SOURCES ${IID})

# create plugin project
add_windows_plugin(${PROJECT_NAME} SOURCES)


# This is an example of how to add a build step to sign the plugin DLL before
# the WiX installer builds. The first filename (certificate.pfx) should be
# the path to your pfx file. If it requires a passphrase, the passphrase
# should be located inside the second file. If you don't need a passphrase
# then set the second filename to "". If you don't want signtool to timestamp
# your DLL then make the last parameter "".
#
# Note that this will not attempt to sign if the certificate isn't there --
# that's so that you can have development machines without the cert and it'll
# still work. Your cert should only be on the build machine and shouldn't be in
# source control!
# -- uncomment lines below this to enable signing --
#firebreath_sign_plugin(${PROJECT_NAME}
# "${CMAKE_CURRENT_SOURCE_DIR}/sign/certificate.pfx"
# "${CMAKE_CURRENT_SOURCE_DIR}/sign/passphrase.txt"
# "http://timestamp.verisign.com/scripts/timestamp.dll")
# add library dependencies here; leave ${PLUGIN_INTERNAL_DEPS} there unless you know what you're doing!
target_link_libraries(${PROJECT_NAME}
 ${PLUGIN_INTERNAL_DEPS}
 )
set(WIX_HEAT_FLAGS
 -gg # Generate GUIDs
 -srd # Suppress Root Dir
 -cg PluginDLLGroup # Set the Component group name
 -dr INSTALLDIR # Set the directory ID to put the files in
 )
add_wix_installer( ${PLUGIN_NAME}
 ${CMAKE_CURRENT_SOURCE_DIR}/Win/WiX/PiSAsalesPluginInstaller.wxs
 PluginDLLGroup
 ${FB_BIN_DIR}/${PLUGIN_NAME}/${CMAKE_CFG_INTDIR}/
 ${FB_BIN_DIR}/${PLUGIN_NAME}/${CMAKE_CFG_INTDIR}/${FBSTRING_PluginFileName}.dll
 ${PROJECT_NAME}
 )
# This is an example of how to add a build step to sign the WiX installer
# -- uncomment lines below this to enable signing --
#firebreath_sign_file("${PLUGIN_NAME}_WiXInstall"
# "${FB_BIN_DIR}/${PLUGIN_NAME}/${CMAKE_CFG_INTDIR}/${PLUGIN_NAME}.msi"
# "${CMAKE_CURRENT_SOURCE_DIR}/sign/certificate.pfx"
# "${CMAKE_CURRENT_SOURCE_DIR}/sign/passphrase.txt"
# "http://timestamp.verisign.com/scripts/timestamp.dll")
```

Remarks

- The use of "SOURCE_GROUP" definitions can be modified or omitted as you want.

- The macro "ADD_MSVC_PRECOMPILED_HEADER" was taken from a cmake-file of the FireBreath core and was slightly modified.
- The compiler option "/FI\"${__PrecompiledBinary}\"" forces the PCH file to be included in each source file during the compilation process. This way, one could omit "#include "stdafx.h", but this could confuse Visual Studio if it scans dependencies and builds the IntelleSense database (allthough IntelliSense seems to be unable to deal with FireBreath projects...).

IID Instantiation

If your plugin uses any COM objects, then you'll need all required interface IDs (IIDs) to be instantiated in your plugin code, otherwise you'll get strange linker errors about unresolved references to objects like "_IID_IUnknown", "_IID_IDispatch", "_IID_YourOwnComInterface" etc. All you have to do is to create a simple source file (an appropriate name could be "iid.cpp"), that has two main properties: it defines "INITGUID" in the first line of code and it includes (explicitely or implicitely) all header files that provide the required interfaces IDs (windows.h, objbase.h, shlobj.h, ...). Because you need this specific "#define INITGUID" only here, **the file "iid.cpp" cannot be compiled using PCH**. This is what the macro "SET_IID_INSTANTIATION" in the above example does.

Fix build error with cmake 2.8.7 and Xcode 4.3

** If you're using XCode 4.3 or later and it isn't working, just update to the latest cmake. If you're using 4.5 or later you need cmake 2.8.10 or later.

Add custom build steps

CMake provides the ADD_CUSTOM_COMMAND and ADD_CUSTOM_TARGET to allow attaching pre-build or post-build steps to your build. They can be placed at the end of the platform's ProjectDef.cmake to accommodate the differences between implementations

For Visual Studio, the steps will be placed in the build events properties for the individual project inside the solution.

For Xcode, the command is put inside target_preBuildCommands.make or target_postBuildCommands and shows up in the Build Phases tab.

For Makefiles the ADD_CUSTOM_COMMAND must be paired with a ADD_CUSTOM_TARGET to work.

In most cases the ${MY_TARGET} would be replaced with ${PROJECT_NAME}. It refers to the target (in this case your plugin) that the dependencies are being added to.

This example runs a C program that generates a c++ header. The only difference between Windows and Mac was the requirement of the file extension ".exe". If the call runs a python script, Windows and Mac can be identical. In this example the executable is in a common tools folder by platform. ie MyTools/Win, MyTools/Mac, MyTools/Lin

Pre Build command added to ProjectDef.cmake

```
Windows
        ADD_CUSTOM_COMMAND(
            TARGET "${MY_TARGET}"
            PRE_BUILD
            COMMAND "${MY_BUILD_TOOLS_DIR}/${PLATFORMLIB}/VersionTagger.exe"
            "${FBSTRING_PLUGIN_MAJOR}"
            "${FBSTRING_PLUGIN_MINOR}"
            "${FBSTRING_PLUGIN_REVISION}"
            "${FBSTRING_PLUGIN_BUILD}"
            "${FBSTRING_PLUGIN_OUTPUT}"
        )

Mac
        ADD_CUSTOM_COMMAND(
            TARGET "${MY_TARGET}"
            PRE_BUILD
            COMMAND "${MY_BUILD_TOOLS_DIR}/${PLATFORMLIB}/VersionTagger"
            "${FBSTRING_PLUGIN_MAJOR}"
            "${FBSTRING_PLUGIN_MINOR}"
            "${FBSTRING_PLUGIN_REVISION}"
            "${FBSTRING_PLUGIN_BUILD}"
            "${FBSTRING_PLUGIN_OUTPUT}"
)

Linux
        ADD_CUSTOM_TARGET("${MY_TARGET_prebuild}")
    ADD_DEPENDENCIES("${MY_TARGET}" "${MY_TARGET_prebuild}")

        ADD_CUSTOM_COMMAND(
            TARGET "${MY_TARGET_prebuild}"
            PRE_BUILD
            COMMAND "${MY_BUILD_TOOLS_DIR}/${PLATFORMLIB}/VersionTagger"
            "${FBSTRING_PLUGIN_MAJOR}"
            "${FBSTRING_PLUGIN_MINOR}"
            "${FBSTRING_PLUGIN_REVISION}"
            "${FBSTRING_PLUGIN_BUILD}"
            "${FBSTRING_PLUGIN_OUTPUT}"
)
```

C code for this example.

VersionTagger.cpp

```
// VersionTagger.cpp : Creates a Header file and exports to the environment the version information from
FireBreath
//   input  : major, minor, revision, build, projectname
//   output : projectname_ver.h
#include <stdio.h>
#include <stdlib.h>
#include <string>

int main(int argc, char* argv[])
{
                if (argc < 6) return -1;
                // recive major,minor,revision,build,output
                char *major        = argv[1];
                char *minor        = argv[2];
                char *revision = argv[3];
                char *build    = argv[4];
                char *output = argv[5];

                FILE * pFile;
                std::string fname(output);
                fname += "_ver.h";

                pFile = fopen (fname.c_str(),"w");
                fprintf (pFile, "// AutoGenerated Header for verison information\n");
                fprintf (pFile, "");
                fprintf (pFile, "#ifndef %s_VERSION_HEADER\n",output);
                fprintf (pFile, "#define %s_VERSION_HEADER\n",output);
                fprintf (pFile, "namespace FBVERSION{\n");
                fprintf (pFile, "\n");
                fprintf (pFile, "\n");
                fprintf (pFile, "const char major[]=\"%s\";\n",major);
                fprintf (pFile, "const char minor[]=\"%s\";\n",minor);
                fprintf (pFile, "const char revision[]=\"%s\";\n",revision);
                fprintf (pFile, "const char build[]=\"%s\";\n",build);
                fprintf (pFile, "const char output[]=\"%s\";\n",fname.c_str());
                fprintf (pFile, "}\n");
                fprintf (pFile, "\n");
                fprintf (pFile, "\n");

                fprintf (pFile, "#endif //%s_VERSION_HEADER\n",output);

                fclose (pFile);

                std::string en("major=");
                en = en + major;
                putenv(const_cast<char*>(en.c_str()));

                en = "minor=";
                en = en + minor;
                _putenv(const_cast<char*>(en.c_str()));

                en = "revision=";
                en = en + revision;
                _putenv(const_cast<char*>(en.c_str()));

                en = "build=";
                en = en + build;
                _putenv(const_cast<char*>(en.c_str()));

                return 0;
}
```

Issues With Internet Explorer Enhanced Protected Mode

Some considerations to get your Add On to wotk within IE E.P.M. in Windows 8:

1. Provide (and register) both: 32 and 64 bit DLLs;
2. List **AppContainer** as **Implemented Category** [1] of your ActiveX;
    a. A simple way of doing so is by editing firebreath\gen_templates**FBControl.rgs**, adding after the line containing "'Version' = s " the following
    b.
```
'Implemented Categories'
{
   {59fb2056-d625-48d0-a944-1a85b5ab2640}
}
```
3. Be ready to a very restrictive environment, with access denied to many paths;
    a. Remember you are running within AppContainter, at very **low Integrity**, with all UIPI implications;
4. If you keep your DLL outside Program Files, you must add "*All Application packages*" group to the file's security permissions;
5. Have a look at IE Protected Mode API: http://msdn.microsoft.com/en-us/library/hh802026

References:

1. 64bit explorer bar issue with IE10 in desktop mode with Enhanced Protected Mode enabled - http://social.msdn.microsoft.com/Forums/en-US/ieextensiondevelopment/thread/776ddfb1-93d0-4207-8948-5797dbcb7829/
2. MSDN Blogs > IEBlog > Enhanced Protected Mode - http://blogs.msdn.com/b/ie/archive/2006/02/09/528963.aspx
3. Understanding and Working in Protected Mode Internet Explorer - http://msdn.microsoft.com/en-us/library/bb250462
4. A Developer's Survival Guide to IE Protected Mode By Michael Dunn, 20 May 2007 - http://www.codeproject.com/Articles/18866/A-Developer-s-Survival-Guide-to-IE-Protected-Mode
5. Introduction to the Protected Mode API - http://msdn.microsoft.com/en-us/library/ms537319
6. Protected Mode Windows Internet Explorer Reference - http://msdn.microsoft.com/en-us/library/ms537312
7. Protected Mode Broker Functions - http://msdn.microsoft.com/en-us/library/cc848890
8. MSDN Blogs > IEBlog > Enhanced Protected Mode - http://blogs.msdn.com/b/ie/archive/2012/03/14/enhanced-protected-mode.aspx
9. MSDN Blogs > IEBlog > Enhanced Memory Protections in IE10 - http://blogs.msdn.com/b/ie/archive/2012/03/12/enhanced-memory-protections-in-ie10.aspx
10. MSDN Blogs > IEInternals > Understanding Enhanced Protected Mode - http://blogs.msdn.com/b/ieinternals/archive/2012/03/23/understanding-ie10-enhanced-protected-mode-network-security-addons-cookies-metro-desktop.aspx
11. Microsoft KB 279459 :: BUG: Component Category Registry Entries Not Removed in ATL Component - http://support.microsoft.com/kb/279459
12. Component Categories and How they Work - http://msdn.microsoft.com/en-us/library/windows/desktop/ms694322