Debugging with Fiddler

The complete reference from the creator of the Fiddler Web Debugger

This is a SAMPLE containing the **Table of Contents** and a bit of content so you can decide whether the book meets your needs and renders nicely on your device.

Buy the book in paperback or ebook format at
http://www.fiddlerbook.com

Eric Lawrence

Debugging with Fiddler
Cover Photo: Nicholas Wave; ©IStockPhoto.com/@by_nicholas

Sample Version LULU 1.00 / Fiddler Version 2.3.9.9

e_lawrence@hotmail.com
@ericlaw on Twitter

Table of Contents

Introduction

About this book

After nearly 9 years and one hundred version updates, Fiddler has evolved into a powerful utility and platform that can perform a wide variety of tasks. It has a rich extensibility model and a community of add-on developers who have broadened its usefulness as a performance, security, and load-testing tool. Questions in email, online discussion groups, and numerous conferences over the years made it overwhelmingly apparent that most users only exploit a tiny fraction of Fiddler's power. I came to realize that thousands of users would get a lot more out of Fiddler if there were a complete reference to the tool available. This book is the product of that realization.

As Fiddler's developer, I've found it both easy and challenging to write this book. It's easy, because I understand Fiddler deeply, down to its very foundation, and can consult the source code to research obscure details. On the other hand, it's been very challenging, as every time I choose an interesting scenario or feature to write about, I'm forced to think deeply about that scenario or feature. Commonly, I've found myself developing improvements to revise Fiddler and minimize or eliminate the need to write about the topic in the first place. As a result, I've rewritten large portions of both this book and Fiddler itself. It's been a slow process, but both projects have benefitted.

Publication of this book will roughly coincide with the release of Fiddler version 2.4.0.0 in the early summer of 2012. If you're using a later version of Fiddler, you will find some minor differences, but the core concepts will remain the same.

This book is deliberately limited in scope—it covers nearly every aspect of Fiddler and FiddlerCore, but it is not a tutorial on HTTP, SSL, HTML, Web Services or the myriad other topics you may want to understand to fully exploit Fiddler's feature set. If you want a deeper understanding of web protocols, I can recommend the references I consulted during the development of Fiddler:

- Hypertext Transfer Protocol -- HTTP/1.1 from `http://www.ietf.org/rfc/rfc2616.txt`
- HTTP: The Definitive Guide by David Gourley
- Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement by Balachander Krishnamurthy and Jennifer Rexford
- SSL & TLS Essentials: Securing the Web by Stephen A. Thomas

This book can be read either "straight through" or you can use the Table of Contents and Index to find the topics most interesting to you. Please consider skimming all of the chapters, even those that don't seem relevant to your needs, because each chapter often contains tips and tricks you might not find elsewhere.

I encourage you to begin by reading the primer in the next chapter, which lays out some terminology and the basic concepts that you'll need to understand to get the most out of Fiddler and this book.

Enjoy!

Understanding Icons and Colors

The default text coloring of each row in the Web Sessions list derives from the HTTP Status (red for errors, yellow for authentication demands), traffic type (`CONNECT`s appear in grey), or response type (CSS in purple, HTML in blue; script in green, images in grey). You can override the font color by setting the Session's `ui-color` flag from FiddlerScript.

Each row is also marked with an icon for quick reference as to the Session's progress, Request type, or Response type:

	The Request is being sent to the server.
	The Response is being downloaded from the server.
	The Request is paused at a breakpoint to allow tampering.
	The Response is paused at a breakpoint to allow tampering.
	The Request used the `HEAD` or `OPTIONS` methods, or returned a `HTTP/204` status code. The `HEAD` and `OPTIONS` methods allow the client to acquire information about the target URL or server without actually downloading the specified content. The `HTTP/204` status code indicates that there is no response body for the specified URL.
	The Request used the `POST` method to send data to the server.
	The Response is HTML content.
	The Response is an image file.
	The Response is a script file.
	The Response is a Cascading Style Sheet (CSS) file.
	The Response is formatted as Extensible Markup Language (XML).
	The Response is formatted using JavaScript Object Notation (JSON).
	The Response is an audio file.
	The Response is a video file.
	The Response is a Silverlight applet.
	The Response is a Flash applet.
	The Response is a font file.
	The Response's `Content-Type` is not a type for which a more specific icon is available.
	The Request used the `CONNECT` method. This method is used to establish a tunnel through which encrypted HTTPS traffic flows.

	The Session wraps a HTML5 WebSocket connection.
	The Response is a `HTTP/3xx` class redirect.
	The Response is a `HTTP/401` or `HTTP/407` demand for client credentials, or a `HTTP/403` error indicating that access was denied.
	The Response has a `HTTP/4xx` or `HTTP/5xx` error status code.
	The Session was aborted by the client application, Fiddler, or the Server. This commonly occurs when the client browser began downloading of a page, but the user then navigated to a different page. The client browser responds by cancelling all in-progress requests, leading to the Aborted Session state.
	The Response is a `HTTP/206` partial response. Such responses are returned as a result of the client performing a `Range` request for only a portion of the file at the target URL.
	The Response is a `HTTP/304` status to indicate that the client's cached copy is fresh.
	The Web Session is unlocked, enabling modification after normal session processing has been completed.

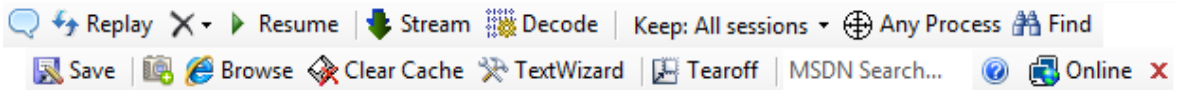
Keyboard Reference

The following keyboard shortcuts are supported by the Web Sessions list:

`Spacebar`	Activate and scroll the currently-focused session into view.
`CTRL+A`	Select all sessions.
`ESC`	Unselect all sessions.
`CTRL+I`	Invert selection; selected sessions are unselected and vice versa.
`CTRL+X`	Remove all sessions (subject to the `fiddler.ui.CtrlX.KeepMarked` preference.)
`Delete`	Remove selected sessions.
`Shift+Delete`	Remove all unselected sessions.
`R`	Replay the current request
`SHIFT+R`	Replay the current request multiple times (specified in the subsequent prompt).
`U`	Unconditionally replay the current request, sending no `If-Modified-Since` and `If-None-Match` headers.
`SHIFT+U`	Unconditionally replay the current request multiple times (the count is specified in the subsequent prompt).
`P`	Attempt to select the "parent" request that triggered this request and set focus to it. This feature depends on the HTTP `Referer` header's value.
`C`	Attempt to select all "child" requests that were provoked by this response. This feature depends on the HTTP `Referer` header's value or the `Location` header on a redirect.

FIDDLER'S TOOLBAR

The Fiddler toolbar provides quick access to popular commands and settings.



The buttons and their functions are:

Comment	Click to add a Comment to all selected Sessions. The comment appears in a column of the Web Sessions list.
Replay	Click to reissue the selected requests to the server again. Hold the `CTRL` key while clicking to reissue the requests without any Conditional Request headers (e.g. `If-Modified-Since` and `If-None-Match`). Hold the `SHIFT` key while clicking to be prompted to specify the number of times each request should be reissued.
Remove	Shows a menu of options for removing Sessions from the Web Sessions list: • **Remove all** removes all Sessions from the list. • **Images** removes all Sessions that returned an image. • **CONNECTs** removes all `CONNECT` tunnels. • **Non-200s** removes all non-`HTTP/200` responses. • **Non-Browser** removes all requests that were not issued by a web browser. • **Complete and Unmarked** removes Sessions which are in the `Done` or `Aborted` state and which are unmarked and have no Comment set. • **Duplicate response bodies** removes any Session which has no response body or has a response body which was received in an earlier Session in the list.
Resume	Resumes all sessions which are currently paused at a Request or Response breakpoint.
Stream	Enable the Stream toggle to disable response buffering for all responses except those for which a breakpoint was set.
Decode	Enable the Decode toggle to remove all HTTP Content and Transfer encodings from requests and responses.
Keep: *value*	The Keep dropdown controls how many Sessions are stored in the Web Sessions list. When the count is reached, Fiddler will begin removing older Sessions to attempt to limit the list to the desired value. Incomplete Sessions and those with comments, markers, or open Inspector windows are not removed.
Process Filter	Drag and drop the Process Filter icon to an application to create a Filter which hides all traffic except for that which originates from the selected process. Right-click the Process Filter icon to clear a previously set filter.
Find	Opens the Find Sessions window.
Save	Saves all Sessions to a SAZ file.
Camera	Adds a JPEG-formatted screenshot of the current desktop to the Web Sessions list.
Browse	If one session is selected, opens Internet Explorer to the target URL. If zero or multiple Sessions are selected, opens Internet Explorer to `about:blank`.

Extending Fiddler's UI - Adding Columns to the Web Sessions List

FiddlerScript can also be used to add new columns to the Web Sessions list, either by using attributes or by making a method call.

Binding Columns using Attributes

The `BindUIColumn` attribute is used to create a new column in the Web Sessions list and bind to it a method in the script that will calculate the text for that column. The method must accept a `Session` object as a parameter, and return a `string` as its result.

The following script adds a new column to the Web Sessions list that shows the HTTP Method for each Session:

```
BindUIColumn("Method", 60)
public static function FillMethodColumn(oS: Session) {
  if ((oS.oRequest != null) && (oS.oRequest.headers != null))
  {
    return oS.oRequest.headers.HTTPMethod;
  }
  return String.Empty;
}
```

After this function is added to the script, a new **Method** column is added to the UI and values are added to the column for each subsequent Session:

#	Method	Protocol	Host
26	GET	HTTP	www.fiddler2.com
27	CONNECT	HTTP	Tunnel to
28	CONNECT	HTTP	Tunnel to
29	POST	HTTPS	beta.urs.microsoft.com
30	POST	HTTPS	beta.urs.microsoft.com
31	GET	HTTP	www.fiddler2.com

Your method must be robust against being called before the data it relies upon is ready. For instance, if you were to add a column that counts the number of times the word `fuzzle` appears in the HTTP response, your method should immediately return an empty string every time it is called until the `responseBodyBytes` array is created after the response is read from the server. Otherwise, the method will throw a Null Reference Exception every time it is called before the server response is completed.

Because your function will run multiple times for each Session as the Session proceeds from one state to the next, you should ensure that it runs as quickly as possible. One strategy to minimize the work of this function is to cache values

BUILDING EXTENSION INSTALLERS

You may install your extensions using any technology you like. Fiddler simply requires that its Assembly `.dll` appear in the correct folder to load it next time that Fiddler launches.

Fiddler and all of the extensions I've written are installed using setup programs built using the Nullsoft Scriptable Install System (NSIS). You can get this great freeware from `http://nsis.sourceforge.net/Download`. NSIS allows you to write a script that is compiled into a compressed executable file containing all of the binaries that make up your project. The resulting setup program is small and works properly across all versions of Windows.

The only significant shortcoming I've encountered with NSIS is that it does not support Unicode, so you may need to use a different technology like WIX (`http://wix.sourceforge.net/`) if you want your installer to use non-Latin characters (e.g. Japanese).

A full explanation of how to use NSIS is beyond the scope of this book—the tool's website offers plenty of documentation at `http://nsis.sourceforge.net/Docs/`. However, I'll share an example setup script you can use to get started.

```
; In a NSIS Script, the semi-colon is a comment operator
Name "MyExtension"

; TODO: Set a specific name for your installer's executable
OutFile "InstallMyExtension.exe"
; Point to an icon to use for the installer, or omit to use the default
Icon "C:\src\MyExt\MyExt.ico"
XPStyle on                      ; Enable visual-styling for a prettier UI

; Explicitly demand admin permissions because we're going to write to
; Program Files. This prevents the "Program Compatibility Assistant" dialog.
; Note, you can use "user" here if you'd like, but then you must only write
; to HKCU and per-user writable locations on disk.
RequestExecutionLevel "admin"

; Maximize compression
SetCompressor /solid lzma

BrandingText "v1.0.1.0"          ; Text shown at the bottom of the Setup window

;
; TODO: Set the install directory to the proper folder.
;
; To install to the Extensions folder, use:
InstallDir "$PROGRAMFILES\Fiddler2\Scripts\"
InstallDirRegKey HKLM "SOFTWARE\Microsoft\Fiddler2" "LMScriptPath"

; To install to the Inspectors folder, use:
;InstallDir "$PROGRAMFILES\Fiddler2\Inspectors\"
```

Inspecting the Session Object

In the original Inspectors API, the Inspectors were never provided a reference to the `Session` object under Inspection-- only the `headers` and `body` would be provided. This provided for a simple, easily understood API contract, but this simplicity presented a number of shortcomings. For instance, it was impossible for an Inspector to get or set flags on the `Session` object, and even examining properties of the Session was impossible. For instance, the Caching Response Inspector was unable to determine whether the inspected traffic used HTTPS because the URL (and thus the protocol scheme) only appears in the *request* headers, which were never available to a Response Inspector.

To resolve these shortcomings, the `Inspector2` base class was augmented with four additional virtual methods:

```
public virtual void AssignSession(Session oS)
public virtual bool CommitAnyChanges(Session oS)
public virtual bool UnsetDirtyFlag()
public virtual InspectorFlags GetFlags()
```

These methods allow an Inspector to be passed a Session object rather than having individual `header` and `body` properties set using the `IRequestInspector2` and `IResponseInspector2` interfaces. If your Inspector *does not* override these virtual methods, Fiddler will simply access the `headers` and `body` properties on the interface, and you need not implement any of the four virtual methods. If your Inspector *does* override the `AssignSession` method, it must still implement all of the legacy properties because not all codepaths in Fiddler call the newer virtual methods. Specifically, when editing a response using the AutoResponder tab, no Session object is available, so the legacy properties will be used.

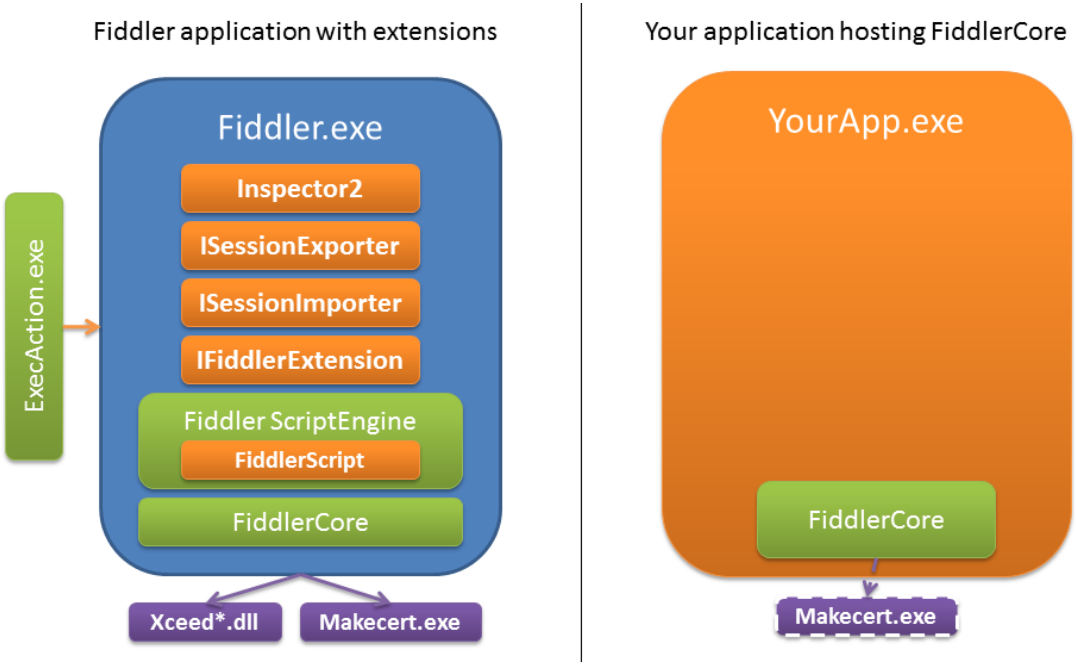
The `AssignSession` method is called when the user selects a session in the Web Sessions list when your Inspector's tab is visible. In your overridden method, your Inspector should update its UI based on the headers and/or body of the session. Note that your Inspector must *itself* examine the Session's `state` to determine whether the Inspector should be readonly, as shown in the following snippet:

```
public override void AssignSession(Session oSession)
{
  if ((null == oSession) || !oSession.bHasResponse)
  {
    Clear();
    return;
  }

  UpdateUIFromHeaders(oSession.oResponse.headers);
  UpdateUIFromBody(oSession.responseBodyBytes);

  bool bIsReadOnly = ((oSession.state != SessionStates.HandTamperResponse)
                  && !oSession.oFlags.ContainsKey("x-Unlocked"));

  UpdateReadOnlyState(bIsReadOnly);
}
```

OVERVIEW

As you've seen in prior chapters, you can extend Fiddler's functionality with both script and .NET code, and this is the best approach for building new functionality for most users. However, in some scenarios, like test automation, it would be more natural to add proxy functionality into an existing tool or test harness instead of using the entirety of Fiddler for the job.

Enter **FiddlerCore**. FiddlerCore is a class library that you can reference in your .NET applications to add Fiddler-like proxy functionality to .NET programs with none of the Fiddler user-interface. This diagram shows the difference between extending Fiddler with your code and extending your code with FiddlerCore:



If you've previously built a Fiddler extension, you'll find that programming against FiddlerCore is an easy adjustment. Many FiddlerCore-based applications are first prototyped as a Fiddler extension before being moved into a standalone program. Building your code on Fiddler first allows you to easily see what is happening to web traffic using Fiddler's Inspectors. Once you're using FiddlerCore, you can only see the web traffic by adding logging functionality to your application (unless you chain your FiddlerCore application to an upstream or downstream Fiddler instance!).

APPENDIX C: SESSION FLAGS

A `StringDictionary` field in each Session object contains flags that control the processing or display of the session.

Some flags are set by Fiddler itself, but most are set script or extensions. The list of supported flags grows with each update to Fiddler, and extensions may use their own flags (which have no meaning to Fiddler) to add state information to a given Session.

The flags can be accessed by `oSession.oFlags["flagname"]` or by using the default indexer on the Session object: `oSession["flagname"]`.

Flag names are case-insensitive strings, and most flag values are interpreted case-insensitively. Most of Fiddler's flags are simply checked for their *existence*, such that setting *any* value (even misleading strings like `0`, `false`, and `heck no!`), enables the named behavior. To disable a flag, remove the flag from the Session like so:

```
oSession.oFlags.Remove("flagname");
```

Because most flags are simply tested for existence, a best practice is to use the flag's value to store a terse explanation of *why* the flag was set. For instance:

```
oSession["ui-hide"] = "hidden by Hide Images rule";
```

You can view a Session's flags by using the Properties item on the Web Sessions list's context menu.

Session Display Flags

The following flags control how a session appears within the Web Sessions list.

Flag Name	`ui-hide`
Explanation	The session will not appear within the Web Sessions list. One of the most commonly used flags, `ui-hide` is used by script or extensions to avoid cluttering the Web Sessions list with uninteresting traffic.
Supported Values	Any value will hide the Session. Typically, you should provide a terse explanation of *why* the Session was hidden, so that if the user activates the Troubleshoot Filters feature on the Help menu, the UI will explain why the Session was hidden. By default, Fiddler will not hide requests that it itself generated (e.g. using the Composer). However, if the `ui-hide` flag's value contains the word `stealth`, the Session will be hidden unconditionally.