Back to "Just Text" with Silverlight

Background 2
Purpose 2
Minimal HTML 3
Automatically adding a Silverlight control 3
Customizing default settings 3
Disabling the "auto-add" Silverlight control 4
Script Tags 5
Inline scripts 5
External scripts 6
XAML 6
Zip files 7
Multiple Silverlight controls 8
Changes to existing behavior 9
Interacting with markup 9
HTML accessors 9
XAML accessors 10
Event handling from markup 11
Event handling from code 11
jQuery-influenced selectors 12
Tiny XAP File 12
AppManifest.xaml 13
languages.config 13
Microsoft.Scripting.slvx 14
Debugging 14
Visual Studio 14
Microsoft.Scripting.Debugging 14
FAQ 15

Background

Dynamic languages were first shown working in Silverlight 1.1 Alpha as part of the Silverlight platform; writing `<x:Code source="app.py" />` in XAML caused Silverlight to download `app.py`, and then called a special hook in the DLR which ran the code contained in `app.py`. Developers would just edit their script files and refresh the browser to see any updates; staying true to the typical browser-development workflow. Silverlight 1.1 Alpha did not have a concept of a XAP file, though it was always planned to have the XAP file option for deployment. However, as Silverlight 1.1 got closer to shipping (and was rebranded Silverlight 2), the XAP file became an absolute requirement for applications. Instead of many JavaScript or XAML files, the XAP file simplified Silverlight's startup dependencies into one compressed package. Coupled with Silverlight 2 moving to a code-first model and the dynamic languages moving outside the Silverlight platform to continue shipping every couple months, a new solution was needed to make dynamic-language-development work as expected.

Silverlight 2 applications written with a dynamic language require the developer to run `Chiron.exe` to get the edit→refresh development experience. `Chiron.exe` is a basic development-time-only web-server which translates all `*.xap` requests into creating a valid XAP file out of the `*` folder, injecting the language's required binaries and `AppManifest.xaml` as needed. So given an `app` directory, Silverlight can be pointed at `app.xap`, and with Chiron.exe running it serves the `app.xap` file as needed. This allows the developer to edit their code, save the files, and then refresh the web browser to see the updated application; Chiron.exe will generate a new XAP on every request. Since this only happens in memory, Chiron.exe can also write the XAP file to disk for deploying on a real web-server, like Apache or IIS.

Since Chiron.exe was introduced, there has been some confusion around how to develop DLR-based Silverlight applications. `Chiron.exe`'s role is so strange to people who use dynamic languages, false information has propagated about what it really does; most notably that Chiron.exe actually performs the compilation and is required for deployment.

The MIX Online Labs team recently released a prototype of this very application thing, called Gestalt. This write-up aims to build upon their prototype's functionality and merge it into the DLR's existing Silverlight integration. In the future the Gestalt team will use this new implementation to build Silverlight application highlighting the benefits of dynamic languages in the browser.

Purpose

This document will detail the changes to the existing `Microsoft.Scripting.Silverlight.dll` to remove the need for `Chiron.exe` to simply achieve an acceptable development mode, make the development model more familiar to browser developers and bring out the "just-text" benefits of dynamic languages.

Making the default close to how browser-JavaScript development works makes Ruby or Python that much more familiar in the browser. This will also remove any obstacles to developing Silverlight applications on any operating system. For example, current Ruby or Python developers who do not use

Windows have a more difficult time developing Silverlight applications, since they have to install Mono to run Chiron.

These changes will introduce no breaking changes to existing DLR-based Silverlight applications; using `Chiron.exe` for development and starting an application with an `app.*` file or the "start" `initParam` will still work. These changes provide a simpler default while preserving the original functionality.

Minimal HTML

`Medium Priority`

`Silverlight.js` is a JavaScript API used to construct the object-tag required to host Silverlight on a HTML page, as well as doing some browser/Silverlight-installation detection. `Silverlight.js` will be part of the default way a Silverlight control for dynamic languages is added to a HTML page. However, the DLR integration needs specific features over-and-above what `Silverlight.js` provides, so those additional features will be merged in with `Silverlight.js` to form `dlr.js`.

Automatically adding a Silverlight control

By default, `dlr.js` does the necessary work to host Silverlight on a page, just by including the file:

```
<script type="text/javascript" src="dlr.js" />
```

This significantly decreases the amount of boilerplate HTML you need to host Silverlight; from about 20 lines of HTML. This adds one hidden Silverlight control for using DLR-based languages to only script the DOM. The XAML section later on will show how to use Silverlight graphics.

Customizing default settings

To customize the default settings, you can do so before `dlr.js` is included. Defaults are merged with any custom settings, so just provide the settings you want to customize.

```
<script type="text/javascript">
  if (!window.DLR)
    window.DLR = {};
  DLR.settings = { width: "100%", height: "100%" };
</script>
<script type="text/javascript" src="dlr.js" />
```

By default the Silverlight control will be created with these settings, which are split between DLR-specific settings and settings passed along to `Silverlight.js`, which all are placed on the Silverlight object tag (default values are in bold):

- `id:` **`"silverlightDLRObject<n>"`** – ID that the Silverlight object-tag will have, with `<n>` being the current count of objects added by `dlr.js`. However, if the first control is autoAdded, its `id` is set to **`"silverlightDlrObject_DOMOnly"`**.
- `width:` **`1`** – Tells Silverlight the `width` of the control. Minimum value is `"1"`. Passed directly to `Silverlight.js`.

- `height: 1` – Tells Silverlight the `height` of the control. Minimum value is `"1"`. Passed directly to `Silverlight.js`.
- `source: "dlr.xap"` – XAP file that Silverlight will use. Passed directly to `Silverlight.js`.
- `onError: Silverlight.default_error_handler` – Default error handler if an exception reaches the browser page. This is passed directly to `Silverlight.js`.

There are also settings that can be set directly on the `DLR` object:

- `DLR.autoAdd: true` – DLR-specific setting to tell `dlr.js` whether or not to auto-add a Silverlight control. Set this to `false` if you need finer control over when and where the control gets created.
- `DLR.path: ""` – Base path where the XAP file is located. By default it will be in the same location as the HTML file.

Documentation on Silverlight-specific settings: http://msdn.microsoft.com/en-us/library/cc838217(VS.95).aspx.

Here are the remaining DLR-specific settings (when omitted, the bold value is default). They are all passed along to `Silverlight.js` as `initParams`:

- `reportErrors: "errorLocation"` – DLR-specific setting that provides the DOM-id to inject a pretty error window if an syntax or compile error occurs.
- `debug`: (`true`|**`false`**) – indicates whether the DLR should produce debug-able code. This makes stack-traces accurate, and allows the Visual Studio debugger to be attached to the Silverlight process to debug the script-code.
- `console`: (`true`|**`false`**) – indicates whether the DLR should display a HTML-based console window, allowing you to switch between all available languages. `stdout` and `stderr` are redirected to this console so common printing operations work.
- `start`: (**`"app.*"`**) – entry-point script; executed after inline script-tags. File must be present in the application's XAP file. Not suggested to use when using script-tags. See FAQ for more info.
- `exceptionDetail`: (`true`|**`false`**) – Shows the entire managed stack trace when an exception is raised. Useful for debugging non dynamic code.
- `xamlid:` Used to scope script-tags to run only in a given control. See the multiple Silverlight controls section.

Disabling the "auto-add" Silverlight control

To get more control over when the first Silverlight control is instantiated, disable the `autoAdd` setting:

```
<script type="text/javascript">
  if (!window.DLR)
    window.DLR = {};
  DLR.autoAdd: false;
</script>
<script type="text/javascript" src="dlr.js" />
```

When you want to add the Silverlight control:

```
<script type="text/javascript">
  DLR.createObject();
</script>
```

The `DLR.createObject` method can be used regardless of `autoAdd`'s value; it simply adds a Silverlight control to the HTML page. By default it uses `DLR.settings`, but you can pass custom settings directly to `DLR.createObject` to override them:

```
DLR.createObject({width: '100%', height: '100%'})
```

Script Tags

Once a Silverlight control is on the page, then DLR-based language code can be placed in script tags, either inline or as an external file:

```
<!-- external script -->
<script type="application/python" src="foo.py"></script>

<!-- inline script -->
<script type="application/python">
  import foo
  window.Alert(foo.say_hello())
</script>
```

This implementation aims to be compliant with the HTML4 specification for scripting in HTML pages: http://www.w3.org/TR/html4/interact/scripts.html

Inline scripts

High Priority

The HTML script-tag is used to embed code directly into the HTML page, and this change allows DLR-based languages to be embedded as well. The `type` attribute defines the mime-type the script code should map to; all DLR-based script code should use the `application/` prefix, but the following prefixes are also allowed: `text/` and `application/x-`. The actual language name will be passed to the DLR hosting API, so the above example could have used `application/ironpython` and it would still work.

For languages that depend on spacing as part of the syntax, like Python, the first line's indent should be considered the baseline for indentation (i.e. no indentation). If a line's indent is smaller than the first line's indent, it should be assumed to have no indent as well.

The inline code will be executed in the order they are defined, but before the "start" script is executed (if one is provided). All inline code is to be executed in the same scope, which will allow methods defined in one scope to be called from another:

```
<script type="application/python">
  def foo():
    return "In Foo"
</script>
...
<script type="application/python">
  window.Alert(foo())
</script>
```

If the `defer` attribute on the script tag is not present, its value is `false`. Otherwise, the value is `true` (even if it's explicitly set to `defer="false"`; this is how all modern browsers behave). If set to `true` the code is not run; but it can be used to evaluate later:

```
<script type="application/ruby" defer="true" id="for_later">
  2 + 2
</script>
<script type="application/ruby">
  puts eval(document.for_later.innerHTML)
</script>
```

External scripts

High Priority

To embed an external script into the HTML page, the `src` attribute is used to specify the path to the script file. The path is relative to the HTML page. The `type` attribute is still required, as it will be passed to the DLR hosting API to identify the language.

The code is executed just by placing the script-tag on the page, just like inline script-tags. Each `<script src="">` tag is downloaded and cached in memory, building a virtual file-system of external script code, replacing the role of the XAP file in the previous application-model. The external code is run in its own DLR `ScriptScope`, allowing proper isolation between scripts.

To just download and cache the file but not run it, set `defer="true"`. This allows another script to "include" it (for example, Python's `import` statement or Ruby's `require` method). The language will load the cached contents of the requested script and run the script as the language sees fit.

```
<script type="application/ruby" src="foo.rb" defer="true"></script>
<script type="application/ruby">require 'foo'</script>
```

XAML

Medium Priority

For applications that want to use Silverlight graphics, XAML content can also be embedded into a script-tag, either inline or as an external file. The `type` attribute should be set to `application/xaml+xml`. Just like DLR-language script-tags, XAML script-tags are executed as they are encountered; can be downloaded ahead-of-time by setting `defer="true"`.

If `defer` is omitted, a XAML script tag creates a totally new Silverlight control just for the XAML content, and sets the RootVisual to the inline XAML (note: you must set width and height to see the XAML contents):

```
<script type="application/xaml+xml" id="xamlContent" width="100" height="100">
  <?xml version="1.0"?>
  <Canvas xmlns="http://schemas.microsoft.com/client/2007" Background="Wheat">
    <TextBlock Canvas.Left="20" FontSize="24" />
  </Canvas>
</script>
```

Setting `defer="true"` will require you to set the `RootVisual` yourself; you can do so by setting the `root_visual` to the script-tag element containing XAML:

```
root_visual = document.xamlContent
```

This is the similar way that Silverlight 1.0 allowed XAML to be embedded:
http://msdn.microsoft.com/en-us/library/cc189016(VS.95).aspx

Zip files

High Priority

The external file can be a `*.zip` file; this is useful for larger libraries where it may be cumbersome to list all the script files out as script-tags. The `type` attribute must be set to `application/x-zip-compressed`. The value of the `src` attribute will be placed on the language’s `path`, and basically treated as a folder. When a script file is requested from any other script, the language will try to find it by using its `path` and checking for the existence of the file. If the path contains a `*.zip` portion of the path, it will continue to look inside the zip file.

```
<script type="application/x-zip-compressed" src="ruby-stdlib.zip"></script>
<script type="application/ruby">
  require 'stringio'
</script>
```

The `defer` attribute toggles whether the zip file is placed on the path: it defaults to `false` which adds it to the path, and `true` will not add it to the path. When `defer="true"` you can always programmatically add it to the path using the language’s path mechanisms:

```
<script type="application/x-zip-compressed" src="ruby-stdlib.zip" defer="true">
</script>
<script type="application/x-ruby">
  $:.unshift "ruby-stdlib.zip"
</script>
<script type="application/x-zip-compressed" src="python-stdlib.zip" defer="true">
</script>
<script type="application/x-python">
  import sys
  sys.path.append "python-stdlib.zip"
</script>
```

Since zip files are treated just like a folder, you can put anything inside the ZIP file; DLLs, XAML files, text files, images, etc, and use them just like you would if they were part of the file-system.

```
<script type="application/x-zip-compressed" src="my-archive.zip"></script>
<script type="application/ruby">
  load_assembly "Foo.dll"
  txt = File.open("my-archive.zip/foo.txt", 'r'){|f| f.read }
</script>
```

The `load_assembly` method works because the zip file is placed on the path, and it will look there for `Foo.dll`. When accessing other files with relative/absolute paths, you can use the zip filename in the path to get to files inside the zip file.

Multiple Silverlight controls

Browsers allow for multiple object-controls to be on a single page, so you could have multiple Silverlight controls on the same page. This introduces an unexpected side-effect to having Silverlight run code inside script-tags; every Silverlight control could have access to run every script-tag. Consider the following:

```
<div id="message"></div>
<script src="dlr.js"></script>
<script type="text/javascript">
  DLR.createObject({width: '100', height: '100'})
</script>
<script type="application/ruby">
  root_visual = UserControl.new
</script>
```

Both Silverlight controls will get their `RootVisuals` set, since the Ruby script-tag is executed twice, once for each Silverlight control. To avoid this, script-tags must be scoped to a specific Silverlight control. `dlr.js` instructs `dlr.xap` to only run “un-scoped” script-tags on the first control added to a page, and only run “scoped” script-tags with subsequent added controls.

To “scope” a script-tag, the `class` attribute contains the same value as its corresponding Silverlight control’s `xamlid` initParam:

```
<script type="text/javascript">
  DLR.createObject({xamlid: 'control1'})
</script>
<script type="application/python" class="control1">
  # will only run in the "control1" object
</script>
```

An “un-scoped” script-tag is simply a script-tag without a class attribute, or having a "*" class attribute. These will run in a Silverlight control that does not have the “xamlid” initParam set; dlr.js does this for only the first control it injects.

If you intend to not use Silverlight graphics through script-tags, or only use them in one control, then you don’t need to worry about scoping; scoping only comes into play when you have multiple controls. If you want to use Silverlight graphics, you can use this same strategy on script-tags containing XAML to make sure the proper `RootVisual` is set.

Changes to existing behavior

Though there are no major breaking changes to any existing behavior of existing applications, there needs to be some changes to existing features to make this new activation-model work properly.

Currently, the "start" `initParam` (entry-point/start-script to the DLR Silverlight app) is required if there is no `app.*` file in the XAP file. If the "start" `initParam` is omitted in this condition, an error is raised complaining about not finding an `app.*` file.

This requirement is now completely relaxed; neither an `app.*` file or a "start" `initParam` is required. If no "start" script or defer=false script-tags exist on the page; then nothing runs and no error is raised. This is relaxed because a Silverlight application can be only inline XAML.

```
<script type="application/python">
  ...
</script>
<object ...>
  <params name="source" value="app.xap" />
  <params name="initParams" value="" /> <!-- no initParams value needed -->
</object>
```

Though these changes are being introduced to remove the need for Chiron, it is still a useful tool for generating XAP files on the fly. Chiron now serves files out of the "externalUrlPrefix" path if it is a relative path, so extensions can be developed locally and Chiron instantly picks them up. Also, Chiron's XAP building features will build an appropriate XAP file depending on whether you're using slvx files or not.

Interacting with markup

Medium Priority

To make accessing the HTML and XAML easier and more like how JavaScript works, variables pointing to them are added to the scope in which script-tags are executed in.

HTML accessors

`document` maps to `System.Windows.Browser.HtmlPage.Document` which is of type `HtmlDocument`, and `window` maps to `System.Windows.Browser.HtmlPage.Window`, which is of type `HtmlWindow`.

When a method is called on an `HtmlDocument` that does not exist, it calls `GetElementById(methodName)`. The following examples are in Python:

```
document.a_div_id
# same as ...
document.GetElementById("a_div_id")
document.doesnotexist # None
```

When a method is called on an `HtmlElement` that does not exist, it should call `GetProperty(methodName)`. When calling the non-existent method as a setter, call `SetProperty(methodName, value)`.

```
document.a_div_id.innerHTML
# same as ...
document.a_div_id.GetProperty("innerHTML")

document.a_div_id.innerHTML = "Hi"
# same as ...
document.a_div_id.SetProperty("innerHTML", "Hi")
```

When an indexer is used on an `HtmlElement`, it should call `GetAttribute(methodName)`. When setting the indexer, call `SetAttribute(methodName, value)`.

```
document.link_id['href']
# same as ...
document.link_id.GetAttribute('href')

document.link_id['href'] = 'http://foo.com'
# same as ...
document.a_div_id.SetAttribute('href', 'http://foo.com')
```

XAML accessors

`root_visual` maps to `System.Windows.Application.Current.RootVisual`, having a base-type of `FrameworkElement`. When a method is called that does not exist on `root_visual`, then `FindName(methodName)` is called. This allows access to any XAML elements with an `x:Name` value to be accessed by the `x:Name` value as a method call.

```
root_visual.Message.Text = "New Message"
```

`load_root_visual` is a method used to set the value of `root_visual` when it is not auto-set. It is a light wrapper around `DynamicApplication#LoadRootVisual`. It takes the following parameters:

- `xaml`: **Required**. Can be the following types:
  - `String`: assumes a URI string, and loads it as XAML using DynamicApplication#LoadRootVisual. This will load xaml files referenced by a script-tag, a file in a zip file, or in the main XAP file.
  - `HtmlElement`: assumes the innerHTML is XAML, and loads it using `XamlReader.Load`
- `element`: **Optional**. Type is `FrameworkElement`. Only used when the `xaml` argument is a `String`. Defaults to `UserControl` when not provided.

```
load_root_visual(document.xamlContent)
# same as ...
DynamicApplication.LoadRootVisual = XamlReader.Load(document.xamlContent.innerHTML)
```

Event handling from markup

Low Priority

HTML events can be hooked both through markup and/or code (for HTML/JavaScript reference: http://www.w3.org/TR/html4/interact/scripts.html#h-18.2.3). Events can be hooked directly from HTML by providing the name of the event as an attribute on an HTML element, whose value is a string of code in the default scripting language. The code is executed when the event fires in the context of the current HTML element:

```
<meta http-equiv="Content-Script-Type" content="application/ruby" />
<a href="javascript:void(0)" onclick="self.innerHTML = 'Clicked!'">Click Me</a>
```

This is accomplished by scanning all HTML elements on the page for attributes which are valid event names (see the HTML4 reference above). For each one found, the event is hooked with a handler which evaluates the attribute's value in the default scripting language in the context of the current HTML element. Not all events will be supported, as some have already fired by the time Silverlight gets control (e.g. `onload`).

Events can be hooked directly from XAML by providing the name of the event as an attribute on a XAML tag, its value being the method name to use as a callback when the event fires. The method should take two arguments: the `sender` and the `event_args`.

```
<script type="application/xaml+xml">
  ...
  <TextBox Click="do_click" Text="Click Me" />
  ...
</script>
<script type="application/python">
  def do_click(sender, event_args):
    sender.Text = "Clicked!"
</script>
```

This is accomplished by scanning all XAML files embedded in script tags, parsing the XML and looking for elements with attributes matching a set of supported events (to be determined). When the event fires, the method name is looked up and called if found, otherwise raises a runtime exception indicating the method does not exist. Event hooking will not be supported in XAML files provided in the XAP or another ZIP file, since Silverlight does not have a way to enumerate zips.

Event handling from code

High Priority

From code, events on both HTML and XAML elements can be hooked via the language's specific .NET event hookup syntax. Given the following HTML:

```
<a id="cm">Click Me</a>
```

You can hook events on it just from Ruby:

```
<script type="application/ruby">
  # either hook with a block
  document.cm.onclick do |link|
    link.innerHTML = "Clicked!"
  end

  # or a method
  def do_c(link)
    link.innerHTML = "Clicked!"
  end
  document.cm.onclick.add method(:do_c)
</script>
```

Or Python:

```
<script type="application/python">
  def do_c(link):
    link.innerHTML = "Clicked!"
  document.cm.onclick += do_c
</script>
```

Or any other scripting language based on the DLR. Hooking XAML events also works:

```
<script type="application/xml+xaml">
  ...
  <TextBox x:Name="xcm" Text="Click Me" />
  ...
</script>
<script type="application/ruby">
  root_visual.xcm.mouse_left_button_down{|s,e| s.text = "Clicked!"}
</script>
```

jQuery-influenced selectors

Out of scope

Though the idea of having a jQuery-like selector API for DLR languages is an attractive idea, it is less feasible since each language will want a different way to specify the syntax. Also, libraries in those languages may exist (eg. Ruby's Hpricot), so it'd be best to use those directly. This might be addressed in a future change, or another library, but is out of scope for this change.

Tiny XAP File

High Priority

With both user scripts and larger libraries outside the main XAP file, the main XAP only serves as a container for the `AppManifest.xaml` and any dynamic language assemblies required by the application.

Silverlight 3 introduced "Transparent Silverlight Extensions", a way to package your own libraries into a `.slvx` (**S**ilver**l**ight **v**ersioned e**x**tension) file (really just zip file) which applications can depend on by referencing it from their `AppManifest.xaml`. Using this feature all the assemblies can be removed from the XAP file, put in a `slvx` file, and hosted on an internet location so other applications can depend on it.

Instead of IronPython and IronRuby releases containing the assemblies built for Silverlight, they will just contain a `dlr.xap` file. This `xap` file will be shared between **all** applications; only advanced scenarios will need to modify the `xap` file. It will only containing just two files:

AppManifest.xaml

The `AppManifest.xaml` file just references the `Microsoft.Scripting.slvx` file, and points the Silverlight application at the static entry point in `Microsoft.Scripting.Silverlight.dll` (included in `Microsoft.Scripting.slvx`):

```
<Deployment
 xmlns="http://schemas.microsoft.com/client/2007/deployment"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 RuntimeVersion="3.0.40624.0"
 EntryPointAssembly="Microsoft.Scripting.Silverlight"
 EntryPointType="Microsoft.Scripting.Silverlight.DynamicApplication">
 <Deployment.ExternalParts>
   <ExtensionPart Source="http://ironpython.net/2.6/Microsoft.Scripting.slvx"/>
 </Deployment.ExternalParts>
</Deployment>
```

languages.config

The `languages.config` file lists the configuration information for DLR languages that can be used in Silverlight. This file can be present in a DLR-based `xap` today for defining configuration information for languages other than Ruby and Python, but now this file must be present if an application depends on the `Microsoft.Scripting.slvx` file. Included in this information is the URL for each language's `slvx` file.

```
<Languages>
    <Language names="IronPython;Python;py"
              extensions=".py"
              languageContext="IronPython.Runtime.PythonContext"
              assemblies="IronPython.dll;IronPython.Modules.dll"
              external="http://ironpython.net/2.6/IronPython.slvx" />
    <Language names="IronRuby;Ruby;rb"
              extensions=".rb"
              languageContext="IronRuby.Runtime.RubyContext"
              assemblies="IronRuby.dll;IronRuby.Libraries.dll"
              external="http://ironpython.net/2.6/IronRuby.slvx" />
</Languages>
```

The language node can have the following attributes:

- `names`: `;`-separated list of names the language can use
- `extensions`: `;`-separated list of file extensions the language can use
- `languageContext`: language's type that inherits from `LanguageContext`
- `assemblies`: URIs to assemblies which make up the language

    - Optional: but if `external` is missing, then this list of assemblies is assumed to be in the XAP
- `external`: SLVX file for all language assemblies

Microsoft.Scripting.slvx

`Microsoft.Scripting.slvx` will contain the following DLLs:

- `Microsoft.Scripting.dll`
- `Microsoft.Dynamic.dll`
- `Microsoft.Scripting.Core.dll`
- `Microsoft.Scripting.ExtensionAttribute.dll`
- `Microsoft.Scripting.Silverlight.dll`

When an application starts up, Silverlight downloads the `Microsoft.Scripting.slvx` file, loads all the assemblies inside it, and then kicks off the static entry point, `Microsoft.Scripting.Silverlight.DynamicApplication`. During its startup logic, it tries to load language configuration from the `languages.config` file; if that fails it looks to already loaded assemblies referenced in the `AppManifest.xaml` and loads the configuration info off the assemblies directly. Because of this, XAP files must have a `languages.config` to download languages on-demand. After the language configuration is loaded, the script-tags on the HTML page are processed; for each language used, the existence of all the language's assemblies in the XAP file is checked, and if they are not all found the language's external-package is downloaded, assemblies inside loaded, and a `ScriptEngine` created for the language. Both the list of assemblies and external-package URI are provided by `languages.config`.

If an application cannot depend on the `slvx` files hosted on the internet, they can be hosted on any machine. Just change the `AppManifest.xaml` and `languages.config` to point to the new location. If Chiron is still being used to generate the XAP file, then the `externalUrlPrefix` in `Chiron.exe.config` is the only setting that needs to be changed.

Debugging

Visual Studio

`High Priority`

When you have debug mode turned on, it will just work as it used to. Attach the debugger to the browser, open the script file in Visual Studio, place a breakpoint, etc. Having the script files in the XAP does not make a difference for debugging; it's all about the debug-able code being generated and having the file open in VS.

Microsoft.Scripting.Debugging

`Low Priority`

TODO: lightweight debugger in HTML page

FAQ

The "start" script referenced in the Inline Scripts section – what is it?

The "start" script is another term for the entry-point script. By default it's `"app.*"`, and `"*"` is used to figure out the correct language to instantiate. However, the user can specify the specific start-script in the `initParams`:

```
<param name="initParams" value="start=myapp.py" />
```

See the original dynamic languages in Silverlight specification for more information **TODO add link**.

Can I write offline Silverlight applications with this?

No. Offline Silverlight applications do not allow using the browser DOM APIs, since they just run the Silverlight control outside the browser. Therefore, offline Silverlight applications cannot use <script> tag code. If you'd like to write a Silverlight application that runs both in the browser and on the desktop, you'll need to keep everything in the XAP file and use the "start" script as the application's entry-point.