Textual

Introduction to Writing Plugins for Version 2.1.1 and Later.

The Fundamentals

- Plugins are written in Objective-C. The same language that powers Textual.
- A plugin has full access to the codebase of Textual. It can subclass any part of the Textual or simply listen for a few distinct calls that will be made to it. Everything is voluntary. A plugin will be loaded even if it does absolutely nothing.
- Plugins can listen for server input or user input. They can also create preference panes or set timers to do jobs.
- Every plugin has access to the entire structure of Textual based around the ***world*** and ***client*** pointers. — All header files are also included.
- Plugins are bundles which can be loaded and unloaded at anytime which makes it so restarts of Textual are not required to apply changes.
- The principal class of a plugin **should always** begin with the TPI_ (Textual Plugin Item) prefix to avoid naming collisions with higher level classes All other classes within a plugin also should use a prefix specific to the plugin.

- A plugin is created as a **Bundle** under the “Framework & Library” section of Xcode’s new project window.

- When a plugin is created, it **must be set to never use only the active architecture of the current machine**. If this setting is not changed, then Textual will reject it during loading for not matching the architecture of Textual itself. This setting and the one described below can be edited in the “Build” tab of Project Settings.

- **The Textual executable should always be set as the Bundle Loader** so Xcode can link against the headers of Textual properly. Normally this executable is in the location: ***/Applications/Textual.app/Contents/MacOS/Textual***

- Plugins should import two items at the start of development. First is the Cocoa framework since Xcode does not do this by default. The other is the folder containing our own headers. — “Show the Package Contents” of Textual and drag the “Headers” folder under “Contents” to Xcode. Save the headers as a reference group. **Do not copy.**

- Finally, `#import` the header ***TextualApplication.h*** to your project’s pre-compiled header (.pch) file to link against all available Textual headers.

Plugin Methods

- Textual provides eight methods to every plugin that it loads. Each is voluntary.
- Two indicate when a plugin has been either allocated into the memory heap or when it is about to be deallocated. It is recommended to place timers and other settings within these calls.
- Another two are used for handling server input. One returns an ***NSArray*** of commands for from the server such as ***PRIVMSG*** or *raw numeric 404*. The second method then processes any data handed down to it. — **Note:** Raw numerics need to be returned in the form ***@"<number>"*** so they are interpreted as an ***NSString***.
- Textual provides another two methods for user input. These methods share the same principals of server input whereas one returns the commands to listen for and the other processes the resulting input.
- The last two methods provides plugins the ability to have their own preference pane. One returns the name to use for a menu item which users will click to get to the preference pane. The other provides the ***NSView*** of the pane itself.

```
- (void)pluginLoadedIntoMemory:(IRCWorld *)world;
```

Called when a plugin has been allocated into memory. Textual passes the ***world*** property which a plugin can store as a pointer for when it executes timers or other jobs. ***world*** is one of our highest level objects.

```
- (void)pluginUnloadedFromMemory;
```

Called before a plugin is removed from the memory heap. Stop timers and save data at this point. This is a poor man's implementation of dealloc.

```
- (void)messageSentByUser:(IRCClient *)client
                  message:(NSString *)messageString
                  command:(NSString *)commandString;
```

Used to process user input. The ***commandString*** property is the command that was invoked. For example, if a user typed "***/test***," then it would equal "test" — The ***messageString*** property is any data presented to the command (string following command). Lastly, the ***client*** is the server that the command was executed on. Higher level objects can be reached through ***client***.

```
- (NSArray *)pluginSupportsUserInputCommands;
```

Returns an ***NSArray*** containing a list of lowercase commands that this plugin will support as user input.

```
- (void)messageReceivedByServer:(IRCClient *)client
                         sender:(NSDictionary *)senderDict
                        message:(NSDictionary *)messageDict;
```

Process server input. The ***senderDict*** property is an ***NSDictionary*** that contains details related to the owner of the message. It could either be the server itself or user details such as nickname, ident, hostmask, etc. — ***messageDict*** is also a dictionary. It contains details related to the actual message that was received.

NSLog() can be used during development to get an idea of data passed to this call.

```
- (NSArray *)pluginSupportsServerInputCommands;
```

Returns an ***NSArray*** containing a list of lowercase commands that this plugin will support as server input.

```
- (NSString *)preferencesMenuItemName;
```

Return ***NSString*** to use as the name of the menu item which a user will click to go to the preference pane for the plugin.

```
- (NSView *)preferencesView;
```

The ***NSView*** that Textual will use as the actual preference pane for the plugin. A minimum width of 540 pixels is recommended.

IMPORTANT: All plugins are ran in background threads.

Before starting development on your Textual plugin we must remind you that any action taking place that involves WebKit such as posting a message to the channel view must be done on the main thread.

Textual will try and recognize when you are not on the main thread, but results are not guaranteed. See ***`DDExtensions.h`*** for a list of methods Textual provides for easy thread communication.

Misc. Notes

- At the time that this document was written the header files of Textual did not contain comments as to what each method does. Textual is open source so it is recommended to look at its own source code to see how it interacts.

- This guide is meant to be a quick introduction to the Textual Plugin API. There is a lot missing from it. I am a developer writing it. Not an English major. Developers who want help can feel free to join the **#textual** channel by clicking “Connect to Help Channel” under the “Help” menu of Textual.

- Textual comes with two handy commands for plugin developers: ***/unload_plugins*** deallocates all loaded plugins so that their contents can be replaced in Finder or erased entirely. After any changes have been made the plugins can be initialized again by typing: ***/load_plugins***

- The GitHub account of Textual contains sample code. The sample code covers different areas of development such as modifying menus, processing user input, creating a preference pane, and more. View these as an example of how a plugin can be created and configured.