Textual

Introduction to Writing Plugins for Version 3.0.0 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 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.
- 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.

Example Video

The best way to teach how to create a plugin is to just show one being made step by step. The following series of videos demonstrates the creation of a plugin named “Example” which takes user input from the command ***`/example`*** and echoes it back to the user.

Each video contains a slide with notes related to the video.

These videos as well as associated documentation targets version 3.0.0 of Textual and later. There is absolutely no guarantee any of what is demonstrated will work on an earlier version of Textual.

It is also assumed that you, the reader/viewer, has a basic knowledge of the Objective-C programming language. Plugins are not designed for beginners. Turn back now. You have been warned!

Step 1: Create Project & Configure

See the video "Step 1" — these videos are actually not included in this PDF because that would require this documentation to be exported as a QuickTime movie and that would have looked like trash.

Step 1: Notes

- A plugin is created as a **Bundle**.
- The **Cocoa** environment is required.
- 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***
- In order to write a plugin, Xcode first needs to know where to search for the frameworks and headers used by Textual. These are packaged with the actual application. After defining Textual as the bundle loader, update the configured framework and header search paths.

Step 2: Importing Textual's Headers

See the video "Step 2"

Step 2: Notes

This step is actually not required. Defining the search path for the headers in the project's configuration is enough for Xcode to find everything it needs, but it is still recommended to add a reference to the folder in which the header files are stored. This makes it easier to view what is defined by Textual by having the headers in the project itself.

Do not mistake the process of adding a reference folder as an alternative to defining the header search paths shown in Step 1. Those are required regardless of whether a reference group is created. Listen, bro!

Step 3: Creating First File

See the video "Step 3"

Step 3: Notes

- 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 that plugin.

- Additionally, if the principal class of the plugin is not defined, Textual will not be able to load it. **This setting in your bundle's Info.plist is required.**

Step 4: Defining the Protocol

See the video "Step 4"

Step 4: Notes

Textual defines the protocol ***THOPluginProtocol*** for each plugin loaded. Every method implemented by this protocol is optional. The following describes the methods defined by this protocol.

```
- (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 and controls all servers.

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

The name of this method is actually misleading. It is called when the plugin is being unloaded, **not** after the fact. 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***.

This method is only invoked for commands defined by the plugin. See slide 18 for information about being passed everything that user enters.

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

Returns an ***NSArray*** containing a list of lowercase commands that this plugin will support as user input. If a plugin tries to override a command used by Textual, it will not work. The custom command will be ignored completely.

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

Process server input. The ***senderDict*** property is a ***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 the data passed.

This method does not have the ability to ignore specific server input. It is processed inline with Textual. See slide 17 for more information.

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

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

If a raw numeric is being asked for, then insert it into the array as a ***NSString***.

```
- (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 points is recommended.

```
- (NSString *)processInlineMediaContentURL:(NSString *)resource;
```

resource is a URL that was detected in a message being rendered and Textual is asking the plugin whether the URL can be resolved to an image which can be displayed inline.

This method is called on each plugin loaded into Textual in the order that the plugin was loaded. Return ***nil*** if the URL is not resolvable. The first plugin to return a result will break the loop of plugins being processed.

The actual return value is not validated as a URL. The length of the value however has to be a **minimum of fifteen (15) characters** to allow room for a URL scheme, valid domain, and a filename.

```
- (IRCMessage *)interceptServerInput:(IRCMessage *)input
                                for:(IRCClient *)client;
```

This method is passed a copy of the ***IRCMessage*** class which is an internal representation of the parsed input line. This method can be used to have certain events ignored completely based on what the plugin functions as.

Expected result is the same ***IRCMessage*** item with parameters and information manipulated as needed. Or, ***nil*** for the item to be ignored.

This method is called on each plugin in the order loaded. This method does not stop for the first result returned so the value being passed may have been modified by a plugin above the one being called. This needs to be kept in mind. Try and use some form of input validation when processing the data to overcome this.

```
- (id)interceptUserInput:(id)input
                 command:(NSString *)command;
```

Process user input before Textual does. Return ***nil*** to have it ignored.

This command may be fed a ***NSAttributedString*** or ***NSString***. If it is an ***NSAttributedString***, it most likely contains user defined text formatting. Honor that formatting. Do not turn a ***NSAttributedString*** into an ***NSString***.

This method should only be used if **absolutely required**. Many users will not expect a plugin to modify their input. At least try and inform the user before making any modifications that they are unaware of.

This method is called on each plugin in the order loaded. This method does not stop for the first result returned so the value being passed may have been modified by a plugin above the one being called. This needs to be kept in mind. Try and use some form of input validation when processing the data to overcome this.

See the video “Step 5”

See the video "Step 6"

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.

- 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.