Textual

Introduction to writing plugins for
version 5.0.0 and later.

The Fundamentals

- Plugins are written in Objective-C or Swift.

- Plugins should be written by those who have a basic understanding of either language. It is possible to jump right in, but there will be some roadblocks. This documents assume that, you, the reader know how to at least start a project in Xcode and write a *hello world* statement.

- 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 ***`masterController()`*** and ***`worldController()`*** defines which are available in every class a plugin writes to as long as they import the Textual headers. See ***`TXSharedApplication.h`*** for a list of these common defines.

Example Video

The best way to teach how a plugin is created is to 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 5.0.0 of Textual and later. There is absolutely no guarantee any of what is demonstrated will work on an earlier version of Textual because Textual itself went through a large codebase rewrite in Version 5.0.0.

Step 1: Create Project & Configure

See the video "Step 1" — these videos are not actually 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 5.app/Contents/MacOS/Textual 5`***
- 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. As Textual’s path contain spaces, this value will need to be encapsulated in double quotes.

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.** Your principal class is the only class responsible for ownership of the plugin. It is what will be handed down API calls. It is initialized during loading and maintained as a strong reference until Textual terminates. Of course other classes can be created. However, your principal class is the only one cared about.

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
```

Called when a plugin has been allocated into memory.

```
- (void)pluginWillBeUnloadedFromMemory
```

This method is called when the plugin is being unloaded during application termination. Stop timers and save data at this point. This is a poor man’s implementation of dealloc.

```
- (void)userInputCommandInvokedOnClient:(IRCClient *)client
                          commandString:(NSString *)commandString
                          messageString:(NSString *)messageString
```

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 which was 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 *)subscribedUserInputCommands
```

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)didReceiveServerInputOnClient:(IRCClient *)client
                        senderInformation:(NSDictionary *)senderDict
                       messageInformation:(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, username, address, 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 *)subscribedServerInputCommands
```

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 an ***NSString***.

`- (NSString *)pluginPreferencesPaneMenuItemName`

Return an ***`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 *)pluginPreferencesPaneView`

The ***`NSView`*** that Textual will use as the actual preference pane for the plugin.

A **width of 567 pixels** (points?) will be enforced for all preference panes.

A **minimum height of 406 pixels** is enforced. Anything larger is however discouraged.

```
- (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 non-***nil*** result will break the loop of plugins being processed.

The actual return value must be a valid URL. Textual validates the return by running it through ***NSURL***. If ***NSURL*** returns a ***nil*** object, then it is certain that your plugin returned a bad value.

```
- (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 to 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 to 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 to use some form of input validation when processing the data to overcome this.

```
- (NSString *)willRenderMessage:(NSString *)newMessage
                forViewController:(TVCLogController *)viewController
                         lineType:(TVCLogLineType)lineType
                       memberType:(TVCLogLineMemberType)memberType
```

Passed a message before the internal text to HTML renderer of Textual had a shot at the message. This gives the plugin a chance to manipulate text that will be displayed by replacing certain segments of it with your own.

Under no circumstances should you insert HTML at this point.

Returning ***nil*** or a string with zero length from this method will indicate that the message does not want to be modified and the original ***newMessage*** value will remain in place. There is no way to specify to the renderer that you want to ignore this event. Use the ***intercept**** methods for this purpose.

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 to use some form of input validation when processing the data to overcome this.

```
- (void)didPostNewMessageForViewController:(TVCLogController *)logController
                                   messageInfo:(NSDictionary *)messageInfo
                                 isThemeReload:(BOOL)isThemeReload
                               isHistoryReload:(BOOL)isHistoryReload
```

This method is called the moment a message has been added to the Document Object Model (DOM) of ***logController***.

messageInfo is a dictionary passed down by the renderer to the internals of ***TVCLogController*** which is ultimately passed down to this method call. This dictionary contains several values which would take too long to document here. ***NSLog()*** can be used on the object to inspect it as always.

The values within the ***messageInfo*** may vary between versions of Textual as the actual implementation behind them is internal to Textual. Therefore, it is best to add some error recovery into a plugin when a key may not exist.

Continued on slide 22

isThemeReload informs the call whether the insertion occurred during a style reload. Style reloads occur when a style is changed and the entire view has to have each message repopulated.

isHistoryReload informs the call whether the insertion occurred when the view was first initialized and it was part of the data from previous session being reloaded into the view.

It is NOT recommended to do any heavy work when ***isThemeReload*** and ***isHistoryReload*** is ***YES*** as these events have thousands of messages being processed at the same time.

These events are posted on the dispatch queue associated with the internal plugin manager. They are never received on the main thread. It is extremely important to remember this because WebKit requires work done to it done on the main thread.

Step 5: Writing the Handler

See the video “Step 5”

Step 6: Building & Testing

See the video “Step 6”

IMPORTANT: All plugins are performed 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 to 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 of writing, 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. It even includes Swift sample code. View these as an example of how a plugin can be created and configured.