
# Textual

## Writing Plugins

# The Fundamentals

- Plugins are written in Objective-C
- There are eight instance methods Textual uses for plugins. All voluntary.
- Plugins can listen for server input or user input. They can also create preference panes or set timers to do jobs instead of listening for input.
- Every plugin has access to the entire structure of Textual based around “world” and “client” pointers. — All header files are also included.
- Plugins are simply 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** use 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 architectures 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 Textual’s 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 Textual's 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 to listen for from the server such as PRIVMSG or raw numeric 404. The other method then processes any data handed down to it.
- 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 just been allocated into memory. Textual passes the “world” command which a plugin can store as a pointer for when it executes timers or other jobs.

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

Called the exact second before a plugin is removed from Textual’s memory heap. Stop timers and save any existing data at this point.

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

Used to process user input. The `commandString` variable is the command that was invoked. For example, if a user typed "/test" then it would equal "test" — The `messageString` variable is any data passed to the command (string following command). Lastly, the `client` is the server that the command was called on.

```
- (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;
```

Method called to process server input. The `senderDict` variable is an NSDictionary that contains details related to who sent 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 that is used by the plugin.

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

The NSView that Textual will use as the actual preference pane as the plugin. A minimum width of 534 pixels is recommended.

The following is a list of methods declared in the IRCClient.h header file which can be used by a plugin to post data to a server or channel. Timers must get the correct client from the "world" pointer before invoking these calls. When dealing with user interface (UI) changes such as sending messages **always invoke the client on main thread.**

```
[[client invokeOnMainThread] ...];
```

Invoke client on main thread. Replace ... with one of the calls shown below.

```
- (void)connect;
- (void)quit;
- (void)quit:(NSString *)comment;
- (void)cancelReconnect;

- (void)sendWhois:(NSString *)nick;

- (void)changeNick:(NSString *)newNick;
- (void)joinChannels:(NSArray *)chans;
- (void)joinChannel:(IRCChannel *)channel;
- (void)joinChannel:(IRCChannel *)channel password:(NSString *)password;
- (void)partChannel:(IRCChannel *)channel;
- (void)kick:(IRCChannel *)channel target:(NSString *)nick;

- (void)sendCTCPPing:(NSString *)target;
- (void)sendCTCPQuery:(NSString *)target command:(NSString *)command text:(NSString *)text;
- (void)sendCTCPReply:(NSString *)target command:(NSString *)command text:(NSString *)text;

- (void)sendText:(NSString *)s command:(NSString *)command channel:(IRCChannel *)channel;
- (BOOL)sendCommand:(NSString *)s;
- (BOOL)sendCommand:(NSString *)s completeTarget:(BOOL)completeTarget target:(NSString *)target;

- (void)sendLine:(NSString *)str;
- (void)send:(NSString *)str, ...;

- (IRCChannel *)findChannel:(NSString *)name;
- (IRCChannel *)findChannelOrCreate:(NSString *)name;
- (IRCChannel *)findChannelOrCreate:(NSString *)name useTalk:(BOOL)doTalk;

- (void)sendPrivmsgToSelectedChannel:(NSString *)message;

- (BOOL)printRawHTMLToCurrentChannel:(NSString *)text;
- (BOOL)printRawHTMLToCurrentChannelWithoutTime:(NSString *)text ;
- (BOOL)printRawHTMLToCurrentChannel:(NSString *)text withTimestamp:(BOOL)showTime;

- (BOOL)printBoth:(id)chan type:(LogLineType)type text:(NSString *)text;
- (BOOL)printBoth:(id)chan type:(LogLineType)type nick:(NSString *)nick text:(NSString *)text identified:(BOOL)identified;
```

# Quick Notes

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

- It is recommended to use AppleScript scripts whenever possible. That way users who end up using your new features (if you distribute) they will have the ability to customize your work easier.

- This guide is meant to be a quick introduction to the Textual API. There is a lot missing from it. 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 deleted entirely. After any changes have been made the plugins can be initialized again by typing /**load_plugins**

- The GitHub account of Textual contains four example plugins. They cover different areas such as modifying menus, processing user input, creating a preference pane, and more. View these as an example how a plugin should be created and configured.