SKETCH OF SOME FEATURES OF THE CONTEXT DESIGN

ADAM WALSH

1. Prototype shortcuts

- Data handling still done via the old system, no internal caching.
- No caching or nice handling of the user or server custom defined commands.
- The implementation of the prefix checking gives the bot a lot of work to do when people type for instance the global prefix in a server with a different custom prefix set. Not sure if this can be avoided.
- servconf, userconf and botconf should not exist at bot and context level, at least not in that form. Maybe conf exists and has, say, conf.bot, conf.user and conf.server. The actual configuration could then be in an enumeration, so we have conf.bot.prefix or conf.server.prefix, which are objects with known interfaces (e.g. get, read etc).
- All errors will be handled by attributes of context, which should later be changed to actual exceptions for some things.

2. General Notes

- Custom commands are expected to be handled by custon Command handlers. Commands from a higher priority CommandHandler overwrite command from a lower priority handler. The command handler is expected to implement a method get_cmds(ctx) which returns all the valid commands in the current context. This will mean for custom commands that on every possible command message, the bot has to go out to the data object and ask for commands, however this shouldn't be a problem if we have caching.
- The blacklisted users info goes in the global command handler checks, in the subclassed CommandHandler specifically for para. This is inefficient however we can assume that most messages will not be from blacklisted sources or contexts.

3. Thoughts

- Can eliminate the user and server separate command lists by using different command handlers with loading priorities. When checking for the command iterate through the command handlers.

4. General Docs

4.1. **Error Codes.**

- 0 means there were no errors and the process completed
- 1 means there was a user level error, e.g. permissions
- 2 means there was insufficient data in the context to run the process
- 3 means something bad happened which didn't make sense

4.2. Check functions.

- The check function will return a tuple (n, msg), where n is the error code and msg is the error message to be reported to the user if they requested something which needs this check to pass
- The check function takes a context, and possibly unspecified other keyword arguments

5. Bot

5.1. Initialisation of Bot.

5.2. onmessage event handling.

5.2.1. *Implementation (prototype).*

- Iterate through prefix list, for each prefix check whether message starts with that prefix.
- Note that unless   exists in the prefix list, we want to explicitly ignore the message if it contains . This should save some work for the bot.
- If we find a prefix in the cache, instantiate a message context and ask it what the valid prefixes are, then make sure that we match those.
- If this fails, and if the mention prefix flag is on for the bot, check whether the message starts by mentioning the bot.
- If the message starts by mentioning the bot, then again instantiate a message context, and set mentioned to True.
- If a valid prefix is found, set the prefix variable.
- Now strip out the prefix from the message and grab the first word, which should be the command
- Check the command against the user defined commands and the server defined commands disctionaries, making sure we include a prefix of the user or serverid, in that order. Then check the global command dictionary. If the command does not appear in any of them, if the mentioned flag is on, reply to the message with a meangingless ack, otherwise terminate.
- If we do find a command, then grab the actual command from the dictionary, instantiate a commandcontext, and execute the commands run method in a try/catch block.

5.3. Attributes.

- Prefix_list:
  Cached list of valid prefixes. Load them in the bot initialisation in on_ready.
-

6. Context Classes

6.1. Context.

Base Context class, the methods and attributes here are expected to be available to all snippets, checks, commands, in fact all pluggable bot modules.

6.1.1. *Methods.*

- str async ctx_format(str string):
  Formats the given string by expanding certain keys based on the current context. Current support keys are:

servers	The number of servers the client is in.
users	The number of users the client can see.
channels	The number of channels the client is in.
username	The name of the user in the context, henceforth ctx.user (usually the author in a MessageContext).

mention	A mention for ctx.user.
id	The id of ctx.user.
tag	The tag of ctx.user.
displayname	The displayname of ctx.user.
server	The name of the server context.

- char** get_prefixes():
  Returns a list of valid prefixes in the current context.
  Proper implementation remains TBD.
- dict async get_cmds():
  Returns a dictionary of valid commands in the current context.
  Runs through each command handler loaded into ctx.bot, and requests the command list from the command handler, passing self. Expects the list of handlers in ctx.bot to be sorted by priority, so that higher priority commands will overwrite lower priority ones.
- char** async msg_split(str msg, bool code=False, int MAX_LEN = 1800):
  Takes a string msg, then splits it by newline into blocks of less than MAX_LEN chars.

6.2. **Message Context.**

6.2.1. *Methods.*

- *char** get_prefixes():*
  Returns a list of current valid prefixes for the message. Takes into account the server prefix, the global prefix if the server prefix isn't set, and the user prefix.

7. Server configuration

7.1. **Config settings:**

- Server local prefix(s) (overrides global prefix). Server prefix is treated as global prefix if not set.

8. User configuration

8.1. **Config settings.**

- User bot prefix (does not override server prefix)

9. Data storage

9.1. **Notes.**

- Most data storage will be handled by the database interface.
- Let the database interface handle the caching of common things? For instance, when we retrieve some data, have a cache=False setting which stores the data in memory, next time that request is made, it retrieves it from memory instead. When the data is updated it updates both the cached copy and the actual copy. To do this, we will want some sort of structure held in memory with the same form as the stored data, but obviously a lot smaller. When we request the server prefix for a server for the first time, it gets cached into this data structure. Next time we request it, it is loaded from the structure instead of from the probably slower database.