YAGPDB

Templates

"Go is all about type... Type is life." // William Kennedy

Preface

All available data that can be used in YAGPDB's templating "engine" which is slightly modified version of Golang's stdlib text/template package; more in depth and info about actions, pipelines and global functions like `printf, index, len,` etc > https://golang.org/pkg/text/template/ . This section is meant to be a concise and to the point reference document for all available templates/functions. **Functions** are covered here. For detailed explanations and syntax guide refer to the learning resource.

Legend: at current state this is still prone to formatting errors, but everything in a `code block` should refer to a function, parts of a template's action-structure or output returned by YAGPDB; single word/literal-structure in *italics* refers to type. Methods and fields (e.g. .Append, .User) are usually kept in standard formatting. If argument for a function is optional, it's enclosed in parenthesis `( )`. If there are many optional arguments possible, it's usually denoted by 3-dot `...` ellipsis.
If functions or methods are denoted with an accent, tilde ~, they are not yet deployed in actual YAGPDB bot but are already in master code branch.

> **Always put curly brackets around the data and "actions you perform" you want to formulate as a template** like this: `{{.User.Username}}`
>
> This `{{ ... }}` syntax of having two curly brackets aka braces around context is necessary to form a template's control structure also known as an action with methods and functions stated below.

> Templating system uses standard ASCII quotation marks:
> 0x22 > `"` for straight double quotes, 0x27 > `'` for apostrophes and 0x60 `` ` `` for backticks/back quotes; so make sure no "smart-quotes" are being used.
>
> The difference between back quotes and double quotes in string literals is covered here.

The Dot and Variables

The dot (also known as cursor) `{{ . }}` encompasses all active data available for use in the templating system, in other words it always refers to current context.

For example .User is a Discord User object/structure of current context, meaning the triggering user. To get

user object for other users, functions `getMember`, `userArg` would help. Same meaning of object/struct applies to other **Fields** with dot prefix. If it is mentioned as a **Method** (for example, .Append for type *cslice*) or as a field on a struct (for example, .User.Bot) then it can not be used alone in template context and always belongs on a parent value. That is, `{{.Bot}}` would return `<no value>` whereas `{{.User.Bot}}` returns *bool* true/false. Another good example is .Reaction.Emoji.MessageFormat, here you can use .MessageFormat every time you get emoji structure of type *discordgo.Emoji*, either using reaction triggers or for example .Guild.Emojis.

From official docs > "Execution of the template walks the structure and sets the cursor, represented by a period `.` and called "dot", to the value at the current location in the structure as execution proceeds." All following fields/methods/objects like User/Guild/Member/Channel etc are all part of that dot-structure and there are some more in tables below.

For commenting something inside a template, use this syntax: `{{/* this is a comment */}}`. May contain newlines. Comments do not nest and they start and end at the delimiters.

To trim spaces use hyphens after/before curyl brackets, for example > `{{- /* this is a multi-line comment with whitespace trimmed from  preceding and following text */ -}}` Using `{{- ... -}}` is also handy inside `range` actions, because whitespaces and newlines are rendered there as output.

`$` has a special significance in templates, it is set to the starting value of a dot. This means you have access to the global context from anywhere - e.g., inside `range`/`with` actions. `$` for global context would cease to work if you redefine it inside template, to recover it `{{ $ := .  }}`.

`$` also denotes the beginning of a variable, which maybe be initialized inside a template action. So data passed around template pipeline can be initialized using syntax > `$variable := value`. Previously declared variable can also be assigned with new data > `$variable = value`, it has to have a white-space before it or control panel will error out. Variable scope extends to the `end` action of the control structure (`if`, `with`, `range`, `etc.`) in which it is declared, or to the end of custom command if there are no control structures - call it global scope.

Pipes

A powerful component of templates is the ability to stack actions - like function calls, together - chaining one after another. This is done by using pipes `|`. Borrowed from Unix pipes, the concept is simple: each pipeline's output becomes the input of the following pipe. One limitation of the pipes is that they can only work with a single value and that value becomes the last parameter of the next pipeline.

Example: `{{randInt 41 | add 2}}` would pipeline `randInt` function's return to addition `add` as second parameter and it would be added to 2; this more simplified would be like `{{40 | add 2}}` with return 42. If written normally, it would be `{{ add 2 (randInt 41) }}`. Same pipeline but using a variable is also useful one - `{{$x:=40 | add 2}}` would not return anything as printout, 40 still goes through pipeline to addition and 42 is stored to variable `$x` whereas `{{($x:=40) | add 2}}` would return 42 and store 40 to `$x`.

> Pipes are useful in select cases to shorten code and in some cases improve readability, but they **should not be overused**. In most cases, pipes are unnecessary and cause a dip in readability that helps nobody.

Context Data

Context data refers to information accessible via the dot, `{{ . }}`. The accessible data ranges from useful constants to information regarding the environment in which the custom command was executed, such as the user that ran it, the channel it was ran in, and so on.

Fields documented as accessible on specific structures, like the context user `.User`, are usable on all values that share the same type. That is, given a user `$user`, `$user.ID` is a valid construction that yields the ID of the user. Similarly, provided a channel `$channel`, `$channel.Name` gives the name of the channel.

Field	Description
.BotUser	Returns bot's user object.
.CCID	The ID of currently executing custom command in type of *int64*.
.CCRunCount	Shows run count of triggered custom command, although this is not going to be 100% accurate as it's cached up to 30 minutes.
.CCTrigger	If trigger type has a printable trigger, prints out its name. For example, if trigger type is `regex` and trigger is set to `\A`, it would print `\A`.
.DomainRegex	Returns string value of in-built domain-matching regular expression.
.IsPremium	Returns boolean true/false whether guild is premium of YAGPDB or not.
.LinkRegex	Returns string value of in-built link-matching regular expression.
.Permissions	Returns all mapped-out permission bits available for Discord in their bitshifted decimal values; e.g. `{{.Permissions.AddReactions}}` would return `64`, same as `{{bitwiseLeftShift 1 6}}`. More here.
.ServerPrefix	Returns server's command-prefix.

Channel

Field	Description
.Channel.Bitrate	Bitrate used; only set on voice channels.
.Channel.GuildID	Guild ID of the channel.
.Channel.ID	The ID of the channel.
.Channel.IsPrivate	Whether the channel is private.
.Channel.IsThread	Whether the channel is a thread.
.Channel.Mention	Mentions the channel object.
.Channel.Name	The name of the channel.
.Channel.NSFW	Outputs whether this channel is NSFW or not.
.Channel.ParentID	The ID of the channel's parent (category), returns if none.
.Channel.PermissionOverwrites	A slice of permission overwrite structures applicable to the channel.
.Channel.Position	Channel position from top-down.
.Channel.Topic	The topic of the channel.
.Channel.Type	The type of the channel. Explained here.

Channel object in Discord documentation.

Channel functions are covered here.

Guild / Server

Field	Description
.Guild.AfkChannelID	Outputs the AFK channel ID.
.Guild.AfkTimeout	Outputs the time when a user gets moved into the AFK channel while not being active.
.Guild.Channels	Outputs a *slice* of channels in the guild with type *[]dstate.ChannelState.*
.Guild.DefaultMessageNotifications	Outputs the default message notification setting fo the guild.
	Outputs a list of emojis in the guild with type

.Guild.Emojis	*discordgo.Emoji.*
.Guild.ExplicitContentFilter	Outputs the explicit content filter level for the guild
.Guild.Features	The list of enabled guild features of type *[]string*.
.Guild.Icon	Outputs the icon hash ID of the guild's icon. Settin full icon URL is explained here.
.Guild.ID	Outputs the ID of the guild.
.Guild.MemberCount	Outputs the number of users on a guild.
.Guild.MfaLevel	The required MFA level for the guild. If enabled, members with moderation powers will be required to have 2-factor authentication enabled in order to exercise moderation powers.
.Guild.Name	Outputs the name of the guild.
.Guild.OwnerID	Outputs the ID of the owner.
.Guild.Roles	Outputs all roles and indexing them gives more information about the role. For example `{{len .Guild.Roles}}` gives you how many roles ar there in that guild. Role struct has following fields.
.Guild.Splash	Outputs the splash hash ID of the guild's splash.
.Guild.SystemChannelID	The ID of the channel where guild notices such as welcome messages and boost events are posted.
.Guild.Threads	Returns all active threads in the guild as a slice of type *[]dstate.ChannelState*.
.Guild.VerificationLevel	Outputs the required verification level for the guild
.Guild.VoiceStates	Outputs a slice of voice states (users connected to VCs) with type *[]discordgo.VoiceState.*

Method	**Description**
`.Guild.GetChannel id`	Gets the channel with the ID provided, returning a **dstate.ChannelState*.
`.Guild.GetEmoji id`	Gets the guild emoji with the ID provided, returnin a **discordgo.Emoji.*
	Calculates full permissions that the member has i

`.Guild.GetMemberPermissions channelID memberID memberRoles`	the channel provided, taking into account the roles of the member. Example: `{{.Guild.GetMemberPermissions .Channel.ID .Member.User.ID .Member.Roles}}` would retrieve the permissions integer the triggering member has in the context/triggering channel.
`.Guild.GetRole id`	Gets the role object with the integer ID provided, returning a struct of type **discordgo.Role.*
`.Guild.GetVoiceState userID`	Gets the voice state of the user ID provided, returning a **discordgo.VoiceState*. Example code to show if user is in VC or not: `{{if .Guild.GetVoiceState .User.ID}} use is in voice channel {{else}} user i not in voice channel {{end}}`

Guild object in Discord documentation.

Member

Field	Description
.Member.Avatar	Member's avatar hash, if it is custom per server, then custom avatar hash.
.Member.GuildID	The guild ID on which the member exists.
.Member.JoinedAt	When member joined the guild/server of type *discordgo.Timestamp*. Method `.Parse` will convert this to of type *time.Time*.
.Member.Nick	The nickname for this member.
.Member.Pending	Returns *bool* true/false, whether member is pending behind Discord's screening process.
.Member.Roles	A *slice* of role IDs that the member has.
.Member.User	Underlying user object on which the member is based on.

Method	Description
.Member.AvatarURL "256"	Gives the URL for member's avatar, argument "256" is the size of the picture and increases/decreses twofold (e.g. 512, 1024 or 128 64 etc.).

Member object in Discord documentation.

Member functions are covered here.

Message

Field	Description
.Message.Attachments	Attachments of this message (*slice* of attachment objects).
.Message.Author	Author of the message (User object).
.Message.ChannelID	Channel ID this message is in.
.Message.Content	Text content of this message.
.Message.ContentWithMentionsReplaced	Replaces all <@ID> mentions with the username the mention.
.Message.EditedTimestamp	The time at which the last edit of the message occurred, if it has been edited. As with .Message.Timestamp, it is of type *discordgo.Timestamp.*
.Message.Embeds	Embeds of this message (*slice* of embed objects).
.Message.GuildID	Guild ID in which the message is.
.Message.ID	ID of the message.
.Message.Link	Discord link to the message. *
.Message.Member	Member object. *
.Message.MentionEveryone	Whether the message mentions everyone, returns *bool* true/false.
.Message.MentionRoles	The roles mentioned in the message, returned as slice of type *discordgo.IDSlice.*
.Message.Mentions	Users this message mentions, returned as a slice of type *[]*discordgo.User.*
.Message.Pinned	Whether this message is pinned, returns *bool* true/false.
.Message.Reactions	Reactions on this message, returned as a slice of type *[]*discordgo.MessageReactions*.
.Message.ReferencedMessage	Message object associated by message_referenc like a message that was replied to.

.Message.Timestamp	Timestamp of the message in type *discordgo.Timestamp* (use .Message.Timestamp.Parse to get type *time.Time* and .Parse.String method returns type *string*).
.Message.Tts	Whether the message is text-to-speech. *

Field	Description
.Args	List of everything that is passed to .Message.Content. .Args is a *slice* of type *string*.
.Cmd	.Cmd is of type *string* and shows all arguments tha trigger custom command, part of .Args. Starting from `{{index .Args 0}}`.
.CmdArgs	List of all the arguments passed after `.Cmd` (`.Cmd` is the actual trigger) `.CmdArgs` is a *slice* of type *string*. For example `{{$allArgs := (joinStr " " .CmdArgs)}}` saves all the arguments after trigger to a variable `$allArgs`.
.StrippedMsg	"Strips" or cuts off the triggering part of the message and prints out everything else after that. Bear in mind, when using regex as trigger, for example `"day"` and input message is `"Have a nice day my dear YAG!"` output will be `"my dear YAG!"` - rest is cut off.

* denotes field that will not have proper return when using `getMessage` function.

Message object in Discord documentation.

Message functions are covered here.

> ⓘ More information about the `Message` object can be found here.

Reaction

This is available and part of the dot when reaction trigger type is used.

Field	Description

.Reaction	Returns reaction object which has following fields `UserID`, `MessageID`, `Emoji.(ID/Name/...)`, `ChannelID`, `GuildID`. The Emoji.ID is the ID of the emoji for custom emojis, and Emoji.Name will hold the Unicode emoji if its a default one. (otherwise the name of the custom emoji)
.Reaction.Emoji.APIName	Returns type *string*, a correctly formatted API nam for use in the MessageReactions endpoints. For custom emojis it is `emojiname:ID`.
.Reaction.Emoji.MessageFormat	Returns a correctly formatted emoji for use in Message content and embeds. It's equal to `<:.Reaction.Emoji.APIName>` and `<a:.Reaction.Emoji.APIName>` for animated emojis.
.ReactionAdded	Returns a boolean type *bool* true/false indicating whether reaction was added or removed.
.ReactionMessage	Returns the message object reaction was added t `{{range .ReactionMessage.Reactions}} {{.Count}} - {{.Emoji.Name}} {{end}}` Returns emoji count and their name. Has an alias `.Message` and it works the same way.

Reaction object in Discord documentation.
Emoji object in Discord documentation.

User

Field	Description
.User	The user's username together with discriminator.
.User.Avatar	The user's avatar hash.
.User.Bot	Determines whether the target user is a bot - if yes it will return `true`.
.User.Discriminator	The user's discriminator/tag (The four digits after a person's username).
.User.ID	The user's ID.

.User.Mention	Mentions user.
.User.String	The user's username together with discriminator a *string* type.
.User.Username	The user's username.
.UsernameHasInvite	Only works with join and leave messages (not joi dms). It will determine does the username contain an invite link.
.RealUsername	Only works with join and leave messages (not joi DMs). This can be used to send the real usernam to a staff channel when invites are censored.

Method	Description
.User.AvatarURL "256"	Gives the URL for user's avatar, argument "256" is the size of the picture and can increase/decrease twofold (e.g. 512, 102 or 128, 64 etc.).

User object in Discord documentation.

User functions are covered here.

Actions

Actions, or elements enclosed in double braces `{{ }}`, are what makes templates dynamic. Without them, templates would be no more than static text. In this section, we introduce several special kinds of actions which affect the control flow of the program. For example, iteration actions like `range` and `while` permit statements to be executed multiple times, while conditional actions like `if` and `with` allow for alteration of what statements are ran or are not ran.

If (conditional branching)

Branching using `if` action's pipeline and comparison operators - these operators don't need to be inside `if` branch. `if` statements always need to have an enclosing `end`.
Learning resources covers conditional branching more in depth.

> `eq`, though often used with 2 arguments (`eq x y`) can actually be used with more than 2. If there are more than 2 arguments, it checks whether the first argument is equal to any one of the following arguments. This behaviour is unique to `eq`.

> ⓘ Comparison operators always require the same type: i.e comparing `1.23` and `1` would throw **`incompatible types for comparison`** error as they are not the same type (one is float, the other int). To fix this, you should convert both to the same type -> for example, `toFloat 1`.

Case	Example

if	`{{if (condition)}} output {{end}}`
	Initialization statement can also be inside `if` statement with conditional statement, limiting the initialized scope to that `if` statement.
	`{{$x := 24}}`
	`{{if eq ($x := 42) 42}} Inside: {{$x}} {{end}}`
	`Outside: {{$x}}`

else if	`{{if (condition)}} output1 {{else if (condition)}} output2 {{end}}` You can have as many `else if` statements as many different conditionals you have.
else	`{{if (condition)}} output1 {{else}} output2 {{end}}`

Boolean Logic	
and	`{{if and (cond1) (cond2) (cond3)}} output {{end}}`
not	`{{if not (condition)}} output {{end}}`
or	`{{if or (cond1) (cond2) (cond3)}} output {{end}}`

Comparison operators	
Equal: `eq`	`{{if eq .Channel.ID ########}} output {{end}}`
Not equal: `ne`	`{{$x := 7}} {{$y := 8}} {{ne $x $y}}` returns `true`
Less than: `lt`	`{{if lt (len .Args) 5}} output {{end}}`
Less than or equal: `le`	`{{$x := 7}} {{$y := 8}} {{le $x $y}}` returns `true`
Greater than: `gt`	`{{if gt (len .Args) 1}} output {{end}}`
Greater than or equal: `ge`	`{{$x := 7}} {{$y := 8}} {{ge $x $y}}` returns `false`

Range

`range` iterates over element values in variety of data structures in pipeline - slices/arrays, maps or

channels. The dot `.` is set to successive elements of those data structures and output will follow execution. If the value of pipeline has zero length, nothing is output or if an `{{else}}` action is used, that section will be executed.

> ⓘ To skip execution of a single iteration and jump to the next iteration, the `{{continue}}` action may be used. Likewise, if one wishes to skip all remaining iterations, the `{{break}}` action may be used. These both are usable also inside `while` action.

Affected dot inside `range` is important because methods mentioned above in this documentation: `.Server.ID`, `.Message.Content` etc are all already using the dot on the pipeline and if they are not carried over to the `range` control structure directly, these fields do not exists and template will error out. Getting those values inside `range` and also `with` action would need `$.User.ID` for example.

`range` on slices/arrays provides both the index and element for each entry; `range` on map iterates over key/element pairs. If a `range` action initializes a variable, that variable is set to the successive elements of the iteration. `range` can also declare two variables, separated by a comma and set by index and element or key and element pair. In case of only one variable, it is assigned the element.

Like `if`, `range` is concluded with `{{end}}` action and declared variable scope inside `range` extends to that point.

```
{{/* range over a slice */}}
{{ range $index, $element := cslice "YAGPDB" "IS COOL!" }}
{{ $index }} : {{ $element }} {{ end }}
{{/* range on a map */}}
{{ range $key, $value := dict "SO" "SAY" "WE" "ALL!" }}
{{ $key }} : {{ $value }} {{ end }}
{{/* range with else and variable scope */}}
{{ range seq 1 1 }} no output {{ else }} output here {{ end }}
{{ $x := 42 }} {{ range $x := seq 2 4 }} {{ $x }} {{ end }} {{ $x }}
```

> ⚠ **Custom command response was longer than 2k (contact an admin on the server...)**
> This is quite common error users will get whilst using range. Simple example to reproduce it:
> *{{ range seq 0 1000 }}*
> *{{ $x := . }}*
> *{{ end }}*
> *HELLO!*
> This will happen because of whitespaces and newlines, so make sure you one-line the range or trim spaces, in this context *{{- $x := . -}}*

Try-catch

Multiple template functions have the possibility of returning an error upon failure. For example, `dbSet` can return a short write error if the size of the database entry exceeds some threshold.

While it is possible to write code that simply ignores the possibility of such issues occuring (letting the error stop the code completely), there are times at which one may wish to write more robust code that handles such errors gracefully. The `try`-`catch` construct enables this possibility.

Similar to an `if` action with an associated `else` branch, the `try`-`catch` construct is composed of two blocks: the `try` branch and the `catch` branch. First, the code in the `try` branch is ran, and if an error is raised by a function during execution, the `catch` branch is executed instead with the context (`.`) set to the offending error.

To check for a specific error, one can compare the result of the `Error` method with a predetermined message. (For context, all errors have a method `Error` which is specified to return a message describing the reason that the error was thrown.) For example, the following example has different behavior depending on whether "Reaction blocked" is in the message of the error caught.

```
{{ try }}
    {{ addReactions " " }}
    added reactions successfully
{{ catch }}
    {{ if in .Error "Reaction blocked" }}
        user blocked YAG :(
    {{ else }}
        different issue occurred: {{ .Error }}
    {{ end }}
{{ end }}
```

While

`while` iterates as long as the specified condition is true, or more generally evaluates to a non-empty value. The dot (`.`) is not affected, unlike with the `range` action. Analogous to `range`, `while` introduces a new scope which is concluded by the `end` action. Within the body of a `while` action, the `break` and `continue` actions can be used to appropriate effect, like in a `range` action.

```
{{/* efficiently search for an element in a sorted slice using binary search */}}
{{ $xs := cslice 1 3 5 6 6 8 10 12 }}
{{ $needle := 8 }}

{{ $lo := 0 }}
{{ $hi := sub (len $xs) 1 }}
{{ $found := false }}
{{/* it's possible to combine multiple conditions using logical operators */}}
{{ while and (le $lo $hi) (not $found) }}
        {{- $mid := div (add $lo $hi) 2 }}
        {{- $elem := index $xs $mid }}
        {{- if lt $elem $needle }}
                {{- $lo = add $mid 1 }}
        {{- else if eq $elem $needle }}
                {{- print "found at index " $mid }}
```

```
                {{- $found = true }}
        {{- else }}
                {{- $hi = sub $mid 1 }}
        {{- end -}}
{{ end }}
{{ if not $found }} not found {{ end }}
```

With

`with` lets you assign and carry pipeline value with its type as a dot (`.`) inside that control structure, it's like a shorthand. If the value of the pipeline is empty, dot is unaffected and when an `else` or `else if` action is used, execution moves on to those branches instead, similar to the `if` action.

Affected dot inside `with` is important because methods mentioned above in this documentation: `.Server.ID` , `.Message.Content` etc are all already using the dot on the pipeline and if they are not carried over to the `with` control structure directly, these fields do not exists and template will error out. Getting those values inside `with` and also `range` action would need `$.User.ID` for example.

Like `if` and `range` actions, `with` is concluded using `{{end}}` and variable scope extends to that point.

```
{{/* Shows the scope and how dot is affected by object's value in pipeline */}}
{{ $x := "42" }} {{ with and ($z:= seq 0 5) ($x := seq 0 10) }}
len $x: `{{ len $x }}`
{{/* "and" function uses $x as last value for dot */}}
same as len dot: `{{ len . }}`
but len $z is `{{ len $z }}` {{ end }}
Outer-scope $x len however: {{ len $x }}
{{/* when there's no value, dot is unaffected */}}
{{ with false }} dot is unaffected {{ else }} printing here {{ .CCID }} {{ end }}
{{/* using else-if chain is possible */}}
{{ with false }}
    not executed
{{ else if eq $x "42" }}
    x is 42, dot is unaffected {{ .User.Mention }}
{{ else if eq $x "43" }}
    x is not 43, so this is not executed
{{ else }}
    branch above already executed, so else branch is not
{{ end }}
```

Associated Templates

Templates (i.e., custom command programs) may also define additional helper templates that may be invoked from the main template. Technically speaking, these helper templates are referred to as *associated*

templates. Associated templates can be used to create reusable procedures accepting parameters and outputting values, similar to functions in other programming languages.

Definition

To define an associated template, use the `define` action. It has the following syntax:

```
{{ define "template_name" }}
    {{/* associated template body */}}
{{ end }}
```

> **Warning:** Template definitions must be at the top level of the custom command program; in other words, they cannot be nested in other actions (for example, an if action.) That is, the following custom command is invalid:
>
> ```
> {{ if $cond }}
>     {{ define "hi" }} hi! {{ end }}
> {{ end }}
> ```

The template name can be any string constant; however, it cannot be a variable, even if said variable references a value of string type. As for the body of the associated template body, it can be anything that is a standalone, syntactically valid template program. Note that the first criterion precludes using variables defined outside of the associated template; that is, the following custom command is invalid, as the body of the associated template references a variable (`$name`) defined in an outer scope:

```
{{ $name := "YAG" }}
{{ define "hello" }}
    Hello, {{ $name }}!
{{ end }}
```

If accessing the value of `$name` is desired, then it needs to be passed as part of the context when executing the associated template.

Within the body of an associated template, the variable `$` and the context dot (`.`) both initially refer to the data passed as context during execution. Consequently, any data on the original context that needs to be accessed must be explicitly provided as part of the context data. For example, if one wishes to access `.User.Username` in an associated template body, it is necessary to pass `.User.Username` as part of the context data when executing said template.

To return a value from an associated template, use the `return` action. Encountering a `return` action will cause execution of the associated template to end immediately and control to be returned to the caller. For example, below is an associated template that always returns `1` :

```
{{ define "getOne" }} {{ return 1 }} {{ end }}
```

Note that it is not necessary for a value to be returned; `{{ return }}` by itself is completely valid.

> **Note:** Since all custom commands are themselves templates, using a `return` action at the top level is perfectly valid, and will result in execution of the custom command being stopped at the point the `return` is encountered.

```
{{ if not .CmdArgs }}
    no arguments passed
    {{ return }} {{/* anything beyond this point is not executed */}}
{{ end }}
{{ $firstArg := index .CmdArgs 0 }}
{{/* safe since .CmdArgs is guaranteed to be non-empty here */}}
```

Execution

To execute a custom command, one of three methods may be used: `template`, `block`, or `execTemplate`.

Template action

`template` is a function-like action that executes the associated template with the name provided, ignoring its return value. Note that the name of the template to execute must be a string constant; similar to `define` actions, a variable referencing a value of string type is invalid. Data to use as the context may optionally be provided following the name.

> While `template` is function-like, it is not an actual function, leading to certain quirks; notably, it must be used alone, not part of another action (like a variable declaration), and the data argument need not be parenthesized. Due to this, it is recommended that `execTemplate`, which has much more intuitive behavior, be used instead of the `template` action if at possible.

Below is an example of the `template` action in action:

```
{{ define "sayHi" }}
    {{- if . -}}
        hi there, {{ . }}
    {{- else }}
        hi there!
    {{- end -}}
{{ end }}
{{ template "sayHi" }} {{/* hi there! */}}
{{ template "sayHi" "YAG" }} {{/* hi there, YAG */}}
```

Trim markers: `{{- ... -}}` were used in above example because whitespace is considered as part of output for associated template definitions (and actions in general).

Block action

`block` has a structure similar to that of a `define` action. It is equivalent to a `define` action followed by a `template` action:

```
{{ $name := "YAG" }}
{{ block "sayHi" $name }}
    hi there, {{ . }}
{{ end }}

{{/* equivalent to above */}}
{{ define "sayHi" }}
    hi there, {{ . }}
{{ end }}
{{ template "sayHi" $name }}
```

execTemplate function

Finally, `execTemplate` is essentially the same as the `template` action, but provides access to the return value of the template and may be used as part of another action. Below is an example using `execTemplate`:

```
{{ define "factorial" }}
    {{- $n := 1 }}
    {{- range seq 2 (add . 1) }}
        {{- $n = mult $n . }}
    {{- end }}
    {{- return $n -}}
{{ end }}

{{ $fac := execTemplate "factorial" 5 }}
2 * 5! = {{ mult $fac 2 }}
```

Custom Types

Golang has built-in primitive data types (*int*, *string*, *bool*, *float64*, ...) and built-in composite data types (*array*, *slice*, *map*, ...) which also are used in custom commands.

YAGPDB's templating "engine" has currently two user-defined, custom data types - *templates.Slice* and *templates.SDict*. There are other custom data types used like *discordgo.Timestamp*, but these are outside of the main code of YAGPDB, so not explained here further. Type *time.Time* is covered in its own section.

Custom Types section discusses functions that initialize values carrying those *templates.Slice* (abridged to *cslice*), *templates.SDict* (abridged to *sdict*) types and their methods. Both types handle type *interface{}* element. It's called an empty interface which allows a value to be of any type. So any argument of any type given is handled. (In "custom commands"-wise mainly primitive data types, but *slices* as well.)

Reference type-like behaviour: Slices and dictionaries in CCs exhibit reference-type like behavior, which may be undesirable in certain situations. That is, if you have a variable `$x` that holds a slice/dictionary, writing `$y := $x` and then mutating `$y` via `Append` / `Set` / `Del` /etc. will modify `$x` as well. For example:

```
{{ $x := sdict "k" "v" }}
{{ $y := $x }}
{{ $y.Set "k" "v2" }} {{/* modify $y */}}
{{ $x }}
{{/* k has value v2 on $x as well -
that is, modifying $y changed $x too. */}}
```

If this behaviour is undesirable, copy the slice/dictionary via `cslice.AppendSlice` or a `range + Set` call .

```
{{ $x := sdict "k" "v" }}
{{ $y := sdict }}
{{ range $k, $v := $x }} {{- $y.Set $k $v -}} {{ end }}
{{ $y.Set "k" "v2" }}
{{ $x }} {{/* $x is unmodified - k still has value v */}}
```

Note that this performs a shallow copy, not a deep copy - if you want the latter you will need to perform the aforementioned operation recursively.

templates.Slice

`templates.Slice` - This is a custom composite data type defined using an underlying data type *[]interface{}* . It is of kind *slice* (similar to *array*) having *interface{}* type as its value and can be initialized using `cslice` function. Retrieving specific element inside *templates.Slice* is by indexing its position number.

Function	Description
`cslice value1 value2 ...`	Function creates a slice of type *templates.Slice* th can be used elsewhere (as an argument for `cembed` and `sdict` for example). Example: `cslice 1 "2" (dict "three" 3) 4.5` returns `[1 2 map[three:3] 4.5]` having length of 4 and index positions from 0 to 3. Notice that thanks to type *interface{}* value, *templates.Slice* elements' inherent type does not change.

Method	Description
.Append arg	Creates a new *cslice* having given argument appended fully by its type to current value. Has

	max size of 10 000 length
.AppendSlice arg	Creates a new *cslice* from argument of type *slice* appended/joined with current value. Has max size of 10 000 length.
.Set int value	Changes/sets given *int* argument as index positio of current *cslice* to new value. Note that .Set can only set indexes which already exist in the slice.
.StringSlice strict-flag	Compares *slice* contents - are they of type *string,* based on the strict-flag which is *bool* and is by default `false`. Under these circumstances if the element is a *string* then those elements will be included as a part of the *[]string* slice and rest simply ignored. Also *time.Time* elements - their default *string* notation will be included. If none are *string* an empty *[]string* slice is returned. If strict-flag is set to `true` it will return *[]string* onl if **all** elements are pure *string*, else `<no value>` is returned. Example in this section's Snippets.

This section's snippets:

- To demonstrate .StringSlice `{{(cslice currentTime.Month 42 "YAPGDB").StringSlice}}` will return a slice `[February YAGPDB]`. If the flag would have been set to true - {{...).StringSlice true}}, all elements in that slice were not strings and `<no value>` is returned.

General example:

```
Creating a new cslice: {{ $x := (cslice "red" "red") }} **{{ $x }}**
Appending to current cslice data
and assigning newly created cslice to same variable:
{{ $x = $x.Append "green" }} **{{ $x }}**
Setting current cslice value in position 1:
{{ $x.Set 1 "blue" }} **{{ $x }}**
Appending a slice to current cslice data
but not assigning newly created cslice to same variable:
**{{ $x.AppendSlice (cslice "yellow" "magenta") }}**
Variable is still: **{{ $x }}**
Type of variable: **{{ printf "%T" $x }}**
```

templates.SDict

`templates.SDict` - This is a custom composite data type defined on an underlying data type *map[string]interface{}.* This is of kind *map* having *string* type as its key and *interface{}* type as that key's

value and can be initialized using `sdict` function. A map is key-value store. This means you store value and you access that value by a key. Map is an unordered list and the number of parameters to form key-value pairs must be even, difference to regular *map* is that `templates.SDict` is ordered by its key. Retrieving specific element inside *templates.Sdict* is by indexing its key.

Function	Description
`sdict "key1" value1 "key2" value2 ...`	Like `dict` function, creating a *templates.SDict* type map, key must be of type *string*. Can be used for example in `cembed`. If only one argument is passed to `sdict` function having type *map[string]interface{};* for example .ExecData and data retrieved from database can be of such type `sdict` was used, it is converted to a new *sdict*. Example: `sdict "one" 1 "two" 2 "three" (cslice 3 4) "five" 5.5` returns unordered `map[five:5.5 one:1 three:[3 4] two:2]`, having length of four and index positions are its keys. Notice that thank to type *interface{}* value, *templates.SDict* elements inherent type does not change.

Method	Description
.Del "key"	Deletes given key from *sdict*.
.Get "key"	Retrieves given key from *sdict*.
.HasKey "key"	Returns *bool* true/false regarding whether the key is set or not e.g. `{{(sdict "YAGPDB" "is cool").HasKey "YAGPDB"}}` would return `true`.
.Set "key" value	Changes/sets given key to a new value or creates new one, if no such key exists in *sdict*.

```
1 Creating sdict: {{ $x := sdict "color1" "green" "color2" "red" }} **{{ $x }}**
2 Retrieving key "color2": **{{ $x.Get "color2" }}**
3 Changing "color2" to "yellow": {{ $x.Set "color2" "yellow" }} **{{ $x }}**
4 Adding "color3" as "blue": {{ $x.Set "color3" "blue" }} **{{ $x }}**
5 Deleting key "color1" {{ $x.Del "color1" }} and whole sdict: **{{ $x }}**
```

> **Tip:** Previously, when saving cslices, sdicts, and dicts into database, they were serialized into their underlying native types - slices and maps. This meant that if you wanted to get the custom type back, you needed to convert manually, e.g. `{{cslice.AppendSlice $dbSlice}}` or

> `{{sdict $dbDict}}`. Recent changes to YAG have changed this: values with custom types are now serialized properly, making manual conversion unnecessary.

Database

You have access to a basic set of Database functions having return of type **customcommands.LightDBEntry* called here DBEntry.
This is almost a key value store ordered by the key and value combined.

You can have max 50 * user_count (or 500 * user_count for premium) values in the database, if you go above this all new write functions will fail, this value is also cached so it won't be detected immediately when you go above nor immediately when you're under again.

Patterns are basic PostgreSQL patterns, not Regexp: An underscore `(_)` matches any single character; a percent sign `(%)` matches any sequence of zero or more characters.

Keys can be max 256 bytes long and has to be strings or numbers. Values can be anything, but if their serialized representation exceeds 100kB a `short write` error gets raised.

You can just pass a `userID` of 0 to make it global (or any other number, but 0 is safe).

There can be 10 database interactions per CC, out of which dbTop/BottomEntries, dbCount, dbGetPattern, and dbDelMultiple may only be run twice. (50,10 for premium users).

Learning resources covers database more in-depth.

Database functions are covered here.

Example here.

DBEntry

Fields	Description
.ID	ID of the entry.
.GuildID	ID of the server.
.UserID	Value of `userID` argument or ID of the user if fo example `.User.ID` was used for `dbSet`.
.User	User object of type *discordgo.User* having only `.ID` field, .Mention is still usable with correct userID field entry.
.CreatedAt	When this entry was created.
.UpdatedAt	When this entry was last updated.

.ExpiresAt	When entry will expire.
.Key	The key of the entry.
.Value	The value of the entry.

Tickets

> Ticket functions are limited to 1 call per custom command for both normal and premium guilds.

Function	Description
`createTicket author topic`	Creates a new ticket with the author and topic provided. Author can be `nil` (to use the triggerin member); user ID in form of a string or an integer; user struct; or a member struct. The topic must be string. Returns a template ticket struct on success

Template Ticket

Field	Description
.AuthorID	Author ID of the ticket.
.AuthorUsernameDiscrim	The Discord tag of the author of the ticket, formatte like `username#discrim`.
.ChannelID	Channel ID of the ticket.
.ClosedAt	Time that the ticket was closed, of type *null.Time.* This is, for the most part, useless in custom commands.
.CreatedAt	Time that the ticket was created.
.GuildID	Guild ID of the ticket.
.LocalID	The ticket ID.
.LogsID	Log ID of the ticket.
.Title	Title of the ticket.

Time

Time and duration types use Golang's time package library and its methods > https://golang.org/pkg/time/#time and also this although slightly different syntax all applies here > https://gobyexample.com/time.

Field	Description
.DiscordEpoch	Gives you Discord Epoch time in *time.Time.* `{{.DiscordEpoch.Unix}}` would return in seconds > 1420070400.
.UnixEpoch	Gives you Unix Epoch time in *time.Time.*
.TimeHour	Variable of *time.Duration* type and returns 1 hour > `1h0m0s`.
.TimeMinute	Variable of *time.Duration* type and returns 1 minut > `1m0s`.
.TimeSecond	Variable of *time.Duration* type and returns 1 secor > `1s`.

Time functions are covered here.