The Gort Guide

Release 0.9

Gort Authors

Aug 01, 2022

CONTENTS

Gort is a chatbot framework designed from the ground up for chatops.

Gort brings the power of the command line to the place you collaborate with your team: your chat window. Its open-ended command bundle support allows developers to implement functionality in the language of their choice, while powerful access control means you can collaborate around even the most sensitive tasks with confidence. A focus on extensibility and adaptability means that you can respond quickly to the unexpected, without your team losing visibility.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Architecture

Gort has several parts:

- The **controller**, which (as its name suggests) acts as the central control point.
- A **data store** which stores all application state.
- One or more **chat services**, such as Slack, which can be used by users to interact with the controller and issue commands.
- One or more **relays**, which execute commands at the direction of the controller.
- A **message bus**, which is used for communication between the controller and the relays.

A high-level view of the relationships between these components is illustrated below.

1.1.1 Gort Controller

The Gort controller proper. This is what you run when you deploy the Gort binary.

It lives in the getgort/gort repository.

1.1.2 Data Store

This stores user, group, and bundle data, as well as a backup of the transaction logs.

Gort currently supports two kinds of data stores:

- External Postgres, intended for production purposes.
- In-memory, intended for trials, testing, and development.

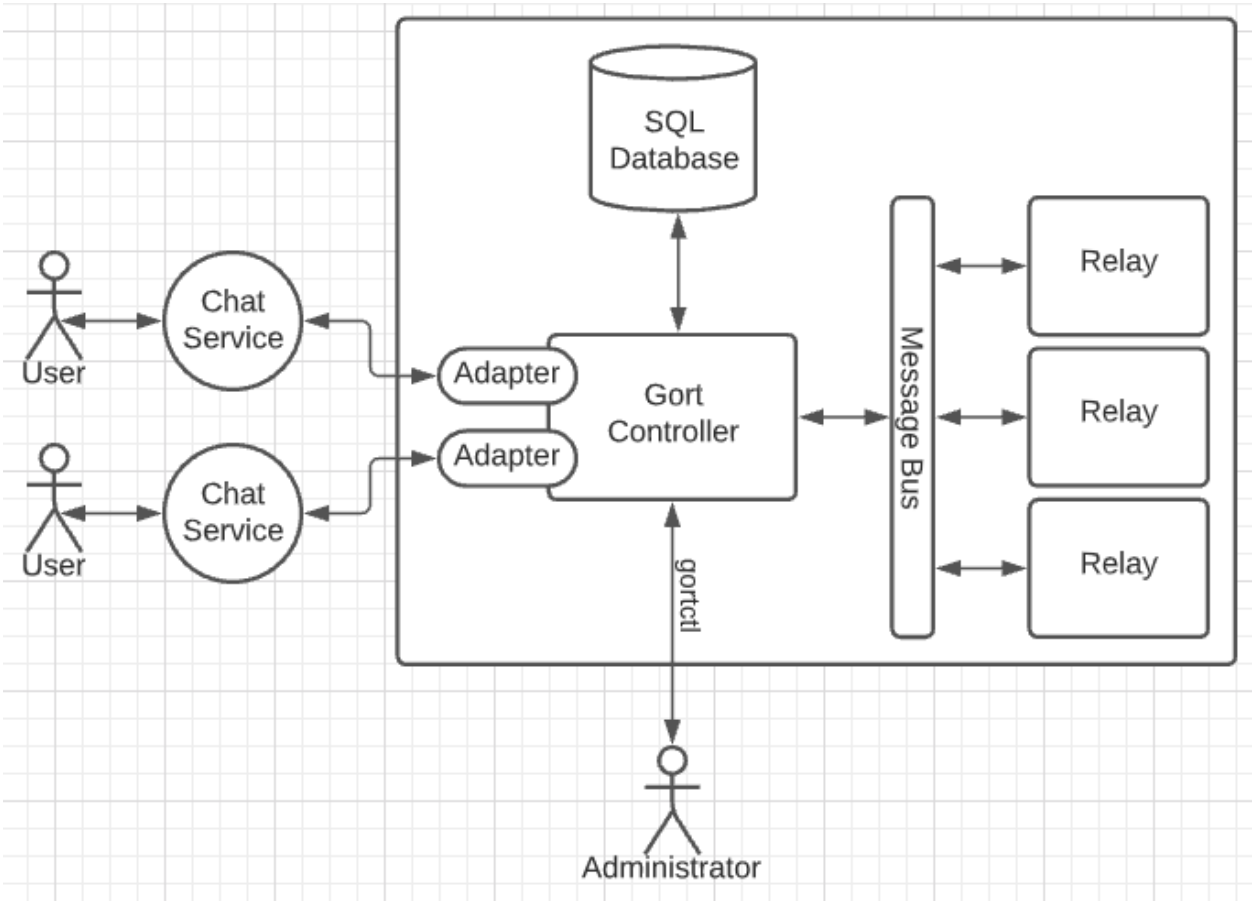


Fig. 1: Gort high-level architecture

1.1.3 Chats

Gort's primary function is to receive messages from users in Slack (and/or other supported chat services) and execute the requested functions.

Currently Gort only supports Slack. It's possible to interact for a single Gort installation to interact with multiple chat services of the same type (multiple Slack workspaces, for example) or different types (Slack and [when supported] Discord, for example).

Adapters

An adapter is a chat-service-specific implementation that receives messages from the service in question, translates them into standard Gort message that can be internally processed, and forwards the message to the Gort system internal for processing. They can then execute the same function in reverse, relaying messages from Gort back to the user(s).

Chat Services

These can be any third-party chat service. Currently only Slack is supported, with more to come soon.

1.1.4 Relays and Commands

Commands triggered by users and conveyed through the adapters are first parsed, compared (by name) against available commands installed as "command bundles", and forwarded to a relay for execution by a worker.

Command Bundles

Command bundles are a set of related commands built into a Docker image or executed natively on the worker. Each bundle includes a list of the specific commands that can be executed, and a set of permission rules required to execute each command.

Command bundles can only be installed by an adequately-privileged user (generally an administrator).

Relays

> **Caution:** This section describes a planned feature that doesn't yet exist.

Optionally, relays can be tagged with identifiers so that commands can be executed preferentially by specific relays installed in specific locations.

Relay Workers

A worker is an ephemeral process executed by a relay to execute a command at the direction of the Gort controller. Upon completion, the process' output and status are conveyed back to the Gort controller via the message bus.

Typically (and per the specific instructions in the corresponding command bundle) a worker will function by pulling a container image and executing the image with the appropriate command and arguments.

Local Command Execution

Caution: This section describes a planned feature that doesn't yet exist.

If so directed in the command bundle (and allowed by the security settings), a worker is capable of executing a command directly on the relay's host.

1.1.5 Message Bus

Caution: This section describes a planned feature that doesn't yet exist.

The Gort controller and the relays communicate via a dedicated message bus, typically Kafka.

1.2 Quick Start

This quick start page will tell you how to quickly get up and running with a very basic "development mode" Gort service running in a local container, communicating with either Slack or Discord.

The resulting service will be suitable for demo purposes only: do not use it in production!

1.2.1 Prerequisites

To install the demo version of Gort, you'll need the following:

- Docker (if using `docker compose`)

1.2.2 Create your Configuration File

Tip: For more information, see: *Configuring Gort*.

1. Copy the example configuration file `config.yml` to `development.yml`.
2. Make the following changes to the `development.yml` file:
    - Remove (or comment out) the `kubernetes` section.
    - If you're using Slack, remove (or comment out) the `discord` section.
    - If you're using Discord, remove (or comment out) the `slack` section.

1.2.3 Create Your Bot User

Whichever chat implementation you use, you'll first have to create a bot user. Choose the section below that's appropriate for your preferred chat provider.

IMPORTANT: Both sections below involve the creation of a secret bot token. Please protect this token carefully: if someone steals it, they can make your bot do whatever they want. You don't want that.

Create a Slack Bot User

1. If you haven't done so already, *create a new Slack workspace <https://slack.com/help/articles/206845317-Create-a-Slack-workspace>*.
2. Use this link to create a new Slack app: *<https://api.slack.com/apps?new_app=1>*. Choose to create your app "From an app manifest".
3. Select your workspace and click "Next".
4. Copy the contents of the manifest file *slackapp.yaml <https://github.com/getgort/gort/blob/main/slackapp.yaml>* into the code box below "Enter app manifest below", replacing the existing content. Click "Next".
5. Review the summary and click "Create" to create your app.
6. On the left-hand bar, under "Settings", click "Basic Information".
7. Under "App-Level Tokens", click "Generate Token and Scopes".
8. Enter a name for your token, click "Add Scope" and select "connections:write". Click "Generate".
9. Copy the app token that starts with `xapp-` and paste it into the `slack` section of your `development.yml` config file as `app_token`. Click "Done".
10. On the left-hand bar, under "Settings", click "Install App".
11. You'll get a screen that says something like "Gort is requesting permission to access the $NAME Slack workspace"; click "Allow"
12. At the top of the screen, you should see "OAuth Tokens for Your Workspace" containing a "Bot User OAuth Token" that starts with `xoxb-`. Copy that value, and paste it into the `slack` section of your `development.yml` config file as `bot_token`.

Create a Discord Bot User

Tip: For a more detailed walk-through, see https://www.writebots.com/discord-bot-token/.

1. Go to the Discord Developer Portal at https://discordapp.com/developers/applications/. This portal shows all of your applications and bots. Click the "New Application" button.
2. Follow the prompt to give your bot a name.
3. (Optional) If you're feeling creative, add an icon and description for your bot.
4. On the menu on the left side of the screen, click "Bot". It's the icon that looks like a little puzzle piece.
5. Click the blue "Add Bot" button, and confirm that you do, in fact, want to add your bot.
6. You should be taken to a page with a green message reading, "A wild bot has appeared!" and a "Token" field. The latter will have and a blue link you can click called "Click to Reveal Token". Click that link, and copy your bot token to your `development.yml` configuration file (specifically the `bot_token` field in your `discord` section).

7. Finally, in order to add your bot to your Discord Server, you'll need to navigate to the "OAuth2" tab on the left-hand menu.
8. Once there, scroll down to the "Oauth2 URL Generator" section. In the "Scopes" section, select the `bot` checkbox. A URL should appear at the bottom on the window: this will be your URL for adding your bot to a server.
9. Finally, scroll down to the "Bot Permissions" section and click the following checkboxes:
    - `Send Messages`
    - `Embed Links`
    - `Read Message History`
10. Scroll up, copy the generated URL, and navigate to it by pasting it into a browser. Select the server to add the bot to, and confirm the selected permissions. If you get an "I'm not a robot" captcha, do what you have to do there (unless you actually are a robot).
11. You should now have successfully created a Discord Bot Token and added your own bot to a server!

1.2.4 Build the Gort Image (Optional)

> **Attention:** This step requires that Docker be installed on your machine.

If you want to use the most absolutely bleeding-edge version of Gort, you can build your own local Gort image. If you don't mind using the stable version, you can skip this step and Docker will automatically download it for you.

To build your own local image you can use the `make` file included in the root of the Gort repository:

```
make image
```

You can verify that this was successful by using the `docker image ls` command:

```
$ docker image ls
REPOSITORY     TAG       IMAGE ID       CREATED         SIZE
getgort/gort   0.9.0     66fca0b90847   5 seconds ago   109MB
getgort/gort   latest    66fca0b90847   5 seconds ago   109MB
```

This should indicate the presence of two images (actually, one image tagged twice) named `getgort/gort`.

1.2.5 Starting Containerized Gort

Finally, from the root of the Gort repository, you can start Gort by using `docker compose` as follows:

```
docker compose up
```

If everything works as intended, you will now be running three containers:

1. Gort
2. Postgres (a database, to store user and bundle data)
3. Jaeger (for storing trace telemetry)

1.2.6 Bootstrapping Gort

Tip: For more information, see: *Bootstrapping Gort*.

Before you can use Gort, you have to bootstrap it by creating the `admin` user.

You can do this using the `gort bootstrap` command and passing it the email address that your Slack provider knows you by, and the URL of the Gort controller API (by default this will be `https://localhost:4000`):

```
$ gort bootstrap --allow-insecure https://localhost:4000
User "admin" created and credentials appended to gort config.
```

Because you haven't installed any TLS/SSL certificates, Gort will generate and use its own. Normally the Gort client will reject self-signed certificates, but the `--allow-insecure` flag turns those safeguards off.

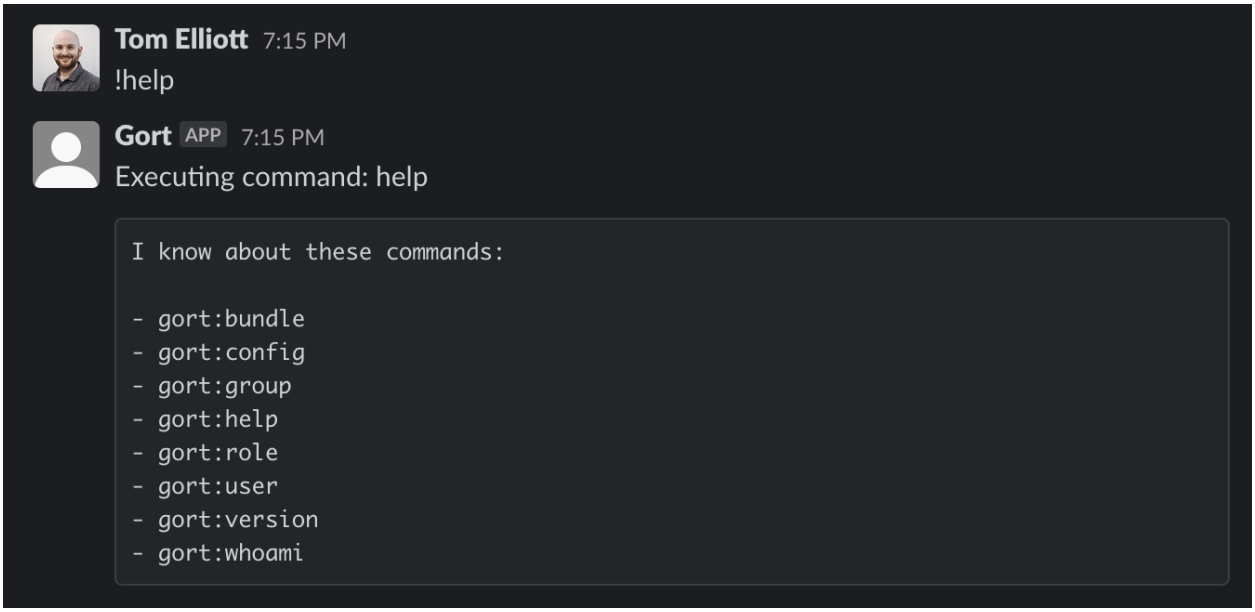
Obviously, do not use this feature in production!

1.2.7 Using Gort

You should now be able to use Gort in any Slack channel that includes your Gort bot. Any Gort commands should be prepended by a `!`. For example, try typing the following in Slack:

```
!help
```

If everything works as expected, you should see an output something like the following:



This instructs Gort to execute the `help` command, which is part of the `gort` bundle. Alternatively, you could have specified the bundle as well by typing something like:

```
!gort:help user
```

1.3 Getting Started

The Gort controller is the core Gort service. It relays messages to and from the chat providers, manages relay command instructions, and exposes the REST administration API. See *Architecture* for more information.

This guide will instruct you through the process of configuring, installing, running, and bootstrapping the Gort controller.

1.3.1 Configuring Gort

Gort is configured via a single YAML-formatted configuration file, typically called `config.yml`, which is used to describe everything from database and chat provider settings to default command bundles.

If the configuration file is changed, Gort can be instructed to "hot reload" its by sending it a SIGHUP or issuing a `GET` request to its `v2/reload/` endpoint. If the new configuration is not well-formed, the changes will not be applied.

Tip: See *Configuring Gort* for more detail.

1.3.2 Deploying Gort

Gort can be installed in a variety of ways: it can be run as a standalone binary, or as a Docker container, or in Kubernetes.

Tip: See *Deploying Gort* for more detail.

1.3.3 Bootstrapping Gort

Once Gort is deployed, the database must be set up and the initial administration user defined, a process referred to as "bootstrapping". Once Gort is properly bootstrapped, the administrator will be able to manage users, install and enable command bundles, and more.

Tip: See *Bootstrapping Gort* for more detail.

1.4 Configuring Gort

Gort is configured via a configuration file, which is used to describe everything from database and chat provider configurations to default command bundles.

Gort can reload its configuration during runtime whenever it detects that its file has been modified. If the new configuration is not well-formed, the changes will not be applied.

1.4.1 Reloading a Configuration

If the configuration file is changed, Gort can be instructed to "hot reload" its by sending it a SIGHUP or issuing a `GET` request to its `v2/reload/` endpoint. If the new configuration is not well-formed, the changes will not be applied.

1.4.2 The Configuration File

Gort configured using a YAML-formatted configuration file. To specify which configuration file to load, use the `--config` flag to the `gort start` command.

The configuration YAML format is defined by the scheme described below. Brackets indicate that a parameter is optional. For non-list parameters the value is set to the specified default.

Generic placeholders are defined as follows:

- <boolean>: a boolean that can take the values `true` or `false`.
- <duration>: a duration string: a sequence of decimal numbers, each with optional fraction and a unit suffix: `1d`, `1h30m`, `5m`, `10s`. Valid units are "ms", "s", "m", "h".
- <filename>: a valid path in the current working directory.
- <host>: a valid string consisting of a hostname or IP (possibly followed by an optional port number).
- <int>: an integer value.
- <path>: a valid URL path.
- <port>: a valid port number ranging from 0 to 65535.
- <secret>: a regular string that is a secret, such as a password.
- <string>: a regular string.
- <template>: a Gort output formatting template.

The other placeholders are specified separately.

A valid example file can be found here.

The global configuration specifies parameters that are valid in all other configuration contexts. They also serve as defaults for other configuration sections.

```
global:
  # How long before a command times out, in seconds. 0 means no timeout.
  [ command_timeout: <int> | default = 60 ]

# Gort controller behavior.
gort:
  [ - <gort> ... ]

# Configures Gort's PostgreSQL database.
database:
  [ - <database> ... ]

# Configures Gort's Docker host data.
docker:
  [ - <docker> ... ]

# Configures Gort's Docker host data.
```

(continues on next page)

(continued from previous page)

```
kubernetes:
  [ - <kubernetes> ... ]

# A list of zero or more configurations that describe Discord servers that
# Gort can relay to and from.
discord:
  [ - <discord> ... ]

# A list of zero or more configurations that describe Slack workspaces that
# Gort can relay to and from.
slack:
  [ - <slack> ... ]

# Configures the Jaeger host, to which Gort sends internal trace telemetry.
jaeger:
  [ - <jaeger> ... ]
```

1.4.3 <gort>

This section contains the settings for the behavior of the Gort controller.

```
# Gort will automatically create accounts for new users when set.
# User accounts created this way will still need to be placed into groups
# by an administrator in order to be granted any permissions.
[ allow_self_registration: <boolean> | default = false ]

# The address to listen on for Gort's REST API.
[ api_address: <host> | default = ":4000" ]

# Controls the prefix of URLs generated for the core API. URLs may contain a
# scheme (either http or https), a host, an optional port (defaulting to 80
# for http and 443 for https), and an optional path.
[ api_url_base: <host> | default = "localhost" ]

# Enables development mode. Currently this only affects log output format.
[ development_mode: <boolean> | default = false ]

# If true, Gort can respond to commands prefixed with !, instead of only
# via direct mentions.
[ enable_spoken_commands: <boolean> | default = true ]

# If set along with tls_key_file, TLS will be used for API connections.
# This parameter specifies the path to a certificate file.
[ tls_cert_file: <filename> ]

# If set along with tls_cert_file, TLS will be used for API connections.
# This parameter specifies the path to a key file.
# The key must not be encrypted with a password.
[ tls_key_file: <filename> ]
```

1.4.4 <database>

The `database` section is used to configure access to Gort's PostgreSQL database.

If this section is absent, the Gort controller will use an "in memory" data model. This is intended for trialing or development and absolutely, positively should not be used in production.

```
# The host where Gort's PostgreSQL database lives.
[ host: <host> | default = "localhost" ]

# The port at which Gort may access its PostgreSQL database.
[ port: <port> | default = 5432 ]

# The user to connect to Gort's PostgreSQL database.
[ user: <string> ]

# The password for connecting to Gort's PostgreSQL database. Alternatively,
# this value can (and should) be specified via the GORT_DB_PASSWORD env var.
[ password: <secret> ]

# Set this to true to have Gort connect to its database using SSL.
[ ssl_enabled: <boolean> | default = false ]

# The maximum amount of time a connection may be idle. Expired connections
# may be closed lazily before reuse. If <= 0, connections are not closed due
# to a connection's idle time.
[ connection_max_idle_time: <duration> | default = 1m ]

# The maximum amount of time a connection may be reused. Expired connections
# may be closed lazily before reuse. If <= 0, connections are not closed due
# to a connection's age.
[ connection_max_life_time: <duration> | default = 10m ]

# Sets the maximum number of connections in the idle connection pool. If
# max_open_connections is > 0 but < max_idle_connections, then this value
# will be reduced to match max_open_connections.
# If n <= 0, no idle connections are retained.
[ max_idle_connections: <int> | default = 2 ]

# The maximum number of open connections to the database. If
# max_idle_connections is > 0 and the new this is less than
# max_idle_connections, then max_idle_connections will be reduced to match
# this value. If n <= 0, then there is no limit on the number of open
# connections.
[ max_open_connections: <int> ]

# How long to wait for execution of a database query to complete.
[ query_timeout: <duration> | default = 15s ]
```

1.4.5 <docker>

This section is used to configure Gort's Docker host data. At the moment it only includes two values (which are likely to move into a relay configuration, when that becomes a thing).

This may not be defined if a `kubernetes` block is also defined.

```
# Defines the location of the Docker port. Required.
host: <path>

# The name of a Docker network. If set, any worker containers will be
# attached to this network. This can be used to allow workers to communicate
# with a containerized Gort controller.
[ network: <string> ]
```

1.4.6 <kubernetes>

This section is used to configure Gort's behavior when deployed in a Kubernetes cluster.

This may not be defined if a `docker` block is also defined.

```
# The label and field selectors for Gort's endpoint resource. These are used
# by Gort to dynamically find its own API endpoint. If both are omitted the
# label selector "app=gort" is used.
[ endpoint_label_selector: <string> ]
[ endpoint_field_selector: <string> ]

# The selectors for Gort's pod resource. Used to dynamically find the
# API endpoint. If both are omitted the label selector "app=gort" is used.

# The label and field selectors for Gort's pod resource. These are used by the
# Gort controller to dynamically find its own pod. If both are omitted the
# label selector "app=gort" is used.
[ pod_field_selector: <string> ]
[ pod_label_selector: <string> ]
```

1.4.7 <discord>

This section is used to describe one or more Discord servers that Gort can receive commands from and relay responses to.

Note that `discord` allows multiple elements, which means that it's possible to configure Gort to interact with multiple Discord servers. It may also be used in additions to one or more `slack` definitions.

```
# An arbitrary name for human labelling purposes. It must be unique among all
# discord and slack definitions.
name: <string>

# The Bot OAuth Token (https://discord.com/developers/docs/topics/oauth2)
# used to connect to Discord.
bot_token: <string>
```

1.4.8 <slack>

This section is used to describe one or more Slack workspaces that Gort can receive commands from and relay responses to.

Note that `slack` allows multiple elements, which means that it's possible to configure Gort to interact with multiple Slack workspaces. It may also be used in addition to one or more `discord` definitions.

```
# An arbitrary name for human labelling purposes. It must be unique among all
# discord and slack definitions.
name: <string>

# App Level Token (https://api.slack.com/authentication/token-types#app)
# used to connect to Slack. You want the one that starts with "xapp".
app_token: <string>

# Bot User OAuth Access Token (https://api.slack.com/docs/token-types#bot)
# used to connect to Slack. You want the one that starts with "xoxb".
bot_token: <string>
```

1.4.9 <jaeger>

This section is used to configures the Jaeger host to which Gort sends internal trace telemetry.

```
# The URL for the Jaeger collector that spans are sent to. If not set then
# no exporter will be created.
[ endpoint: <path> | default = http://localhost:14268/api/traces ]

# The username to be used in the authorization header sent for all requests
# to the collector. If not set no username will be passed.
[ username: <string> | default = gort ]

# The password to be used in the authorization header sent for all requests
# to the collector.
[ password: <secret> ]
```

1.4.10 <templates>

The `templates` section allows the default output formatting templates to be overridden at the application scope. Templates defined here may still be overridden at the bundle and command scopes.

```
# A template used to format the outputs from successfully executed commands.
[ command: <template> ]

# A template used to format the error messages produced by commands that exit
# with a non-zero status.
[ command_error: <template> ]

# A template used to format standard informative (non-error) messages from
# the Gort system (not commands).
[ message: <template> ]
```

(continues on next page)

(continued from previous page)

```
# A template used to format error messages from the Gort system (not commands).
[ message_error: <template> ]
```

1.5 Deploying Gort

This guide will instruct you through the process of installing the Gort controller.

The Gort controller is the core Gort service, which relays messages to and from the chat providers, manages relay command instructions, and exposes the REST administration API. See *Architecture* for more information.

1.5.1 Running Gort in Kubernetes

If you have an existing Kubernetes cluster (or are using something like Minikube or Docker Desktop Kubernetes). you can deploy Gort there. The easiest way to do this is to use the included Helm chart.

Prerequisites

You'll need to have Helm installed. If you don't, you can follow the instructions here.

Deployment

Installing the Helm chart is fairly straight-forward:

```
helm install gort helm/gort
```

1.5.2 Running Gort in Docker

If you don't have Go installed, you can run Gort with Docker.

Building Your Own Gort Image (Optional)

If you want to use the most absolutely bleeding-edge version of Gort, you can build your own local Gort image. If you don't mind using the stable version, you can skip this step and Docker will automatically download it for you.

To build your own local image you can use the `make` file included in the root of the Gort repository:

```
make image
```

You can verify that this was successful by using the `docker image ls` command:

```
$ docker image ls
REPOSITORY     TAG       IMAGE ID       CREATED         SIZE
getgort/gort   0.9.0     66fca0b90847   5 seconds ago   109MB
getgort/gort   latest    66fca0b90847   5 seconds ago   109MB
```

This should indicate the presence of two images (actually, one image tagged twice) named `getgort/gort`.

Now you can check the version of your installation by passing it the `version` command (which is equivalent to executing `gort version`).

```
$ docker run getgort/gort version
Gort ChatOps Engine v0.9.0
```

Starting Containerized Gort

Finally, from the root of the Gort repository, you can start Gort by using `docker compose` as follows:

```
docker compose up
```

If everything works as intended, you will now be running three containers:

1. Gort (listening on port 4000)
2. Postgres (a database, to store user and bundle data)
3. Jaeger (for storing trace telemetry)

Finally, you should see some output similar to the following:

```
INFO  [0000] Loaded configuration file                file=config.yml
INFO  [0000] Starting Gort                            version=0.8.0-alpha.0
INFO  [0000] Installing adapter                       adapter.name=Gort
INFO  [0001] Connection to data source established   type=postgres.PostgresDataAccess
INFO  [0001] Gort controller is starting              address=":4000"
INFO  [0001] Connecting to Slack provider             adapter=Gort provider=Gort
INFO  [0001] Connection established to provider       adapter.name=Gort adapter.
↪type=slack event=connected
```

As you may have noticed, this command opens port 4000. This allows the Gort controller to access the administration API, which is required to bootstrap your Gort instance.

Congratulations, you now have a running Gort controller!

1.5.3 Running Gort as a Native Gort Binary

Installing Gort via `go install`

If you have Go installed, you can build and install Gort in one command using the `go install` command.

```
go install github.com/getgort/gort@latest
```

When installed this way, Gort will be installed to the directory named by the `GOBIN` environment variable, which defaults to `$GOPATH/bin` or `$HOME/go/bin` if the `GOPATH` environment variable is not set.

Building Gort From Source

If you prefer (if you have Go installed), you can also build Gort from the source code.

To do this, you must first clone the getgort/gort repository and `cd` into it.

```
git clone git@github.com:getgort/gort.git
cd gort
```

Once you're in the Gort code repository, you can use `go build` to build the Gort binary.

```
go build
```

You should now have an executable binary named `gort` in your working directory. You can either run it in place, or move it a directory on your command path.

Executing a Native Binary

If you installed or built Gort using `go`, you can run that binary, pointing to the location of the configuration file.

```
gort --config ./config.yml
```

1.6 Bootstrapping Gort

Since interactions with Gort require a user account, you'll need to bootstrap your system before you can do anything interesting with it. This process will create the necessary administrator role and user group, as well as create the first user account and place it into that administrator group. At this point, you can interact with Gort as this first privileged user in order to create other user accounts (to which you can also grant administrative access, if you like), install bundles, and perform other tasks.

The canonical way to do this is to use the `gort bootstrap` command:

```
$ gort bootstrap --allow-insecure https://localhost:4000
User "admin" created and credentials appended to gort config.
```

Note the existence of the `--allow-insecure` flag. This allows you to communicate with the Gort API across an unencrypted connection, which is commonly the case when you're testing locally. _**This state should never, ever be used in production.**_

The `gort bootstrap` command is idempotent: subsequent calls will return an error message, but the Gort system itself will remain unaffected.

If you take a look in `~/.gort/profile`, you'll begin to see what just happened.

```
$ cat ~/.gort/profile

defaults:
    profile: localhost_4000
localhost:
    url: https://localhost:4000
    password: cZO5E4i8+T6vVRO8m4EvYEyGI2fn86iZ
    user: admin
    allow_insecure: true
```

Here, we can see that a user named `admin` has been created for us on the Gort controller. A password has also been generated for this user. Now, whenever we run any `gort` commands from this machine, these credentials will be used by default to make authenticated API calls.

Gort's REST API is guarded by Gort's authorization system, which means that the `admin` user must have permissions to access the API somehow. As detailed in Permissions and Rules, permissions must be attached to a user somehow through a combination of roles and groups. As you can probably guess, the bootstrapping process handles all this. Let's use `gort` to examine what has been done.

First, let's just check which users exist.

```
$ gort user list
USERNAME  FULL NAME           EMAIL ADDRESS
admin     Gort Administrator  admin@gort
```

As you can see, there's just one: `admin`.

Now let's examine the core permissions of the Gort system. These govern fine-grained access to the various REST API endpoints and chat commands.

```
$ gort permission list
NAME
gort:manage_commands
gort:manage_groups
gort:manage_roles
gort:manage_users
```

That's a lot of permissions. Gort helps us out by creating an `admin` role to collect them all together.

```
$ gort role info admin
Name         admin
Permissions  manage_commands, manage_groups, manage_permissions, manage_roles, manage_
↪users
Groups       admin
```

Finally, we have a group that is also named `admin` with the `admin` user as its sole member. This group is granted the `admin` role.

```
$ gort group info admin
Name   admin
Users  admin
Roles  admin
```

Though the Gort `admin` user is named "admin", there's nothing particularly special about that name. As this tour of the bootstrapping process has shown us, the `admin` user functions as an administrator, able to perform any task in the Gort system only because it resides in a group that has been granted all the core permissions. Any user in this group would have the same capabilities.

This also shows how to make any Gort user an administrator by adding them to the `admin` group.

1.7 Commands and Bundles

As a chatops bot, commands are central to Gort. Let's take a look at exactly what commands are, how they're organized, and how they're managed.

Let's start with an example. Entering the following into Slack:

```
!gort:help
```

You should receive a response that looks something like this:

```
I know about these commands:

curl:curl
gort:bundle
gort:group
gort:help
gort:role
gort:user
gort:version
```

Typing `!gort:help` executed the `help` command from the `gort` command bundle. From the response, you can see that the system currently has two bundles installed (`curl`, and `gort`), each of which contains one or more commands. The `curl` bundle contains a single command (also named `curl`), and the `gort` bundle contains several commands, one of which is the `help` command we just invoked.

Try calling `gort:help COMMAND` to find out more about a specific command:

```
!gort:help gort:user
```

This should provide a response like the following:

```
Part of the "gort" bundle.

Allows you to perform user administration.

Use "!gort:user --help" for more information about this command.
```

As indicated in the above output, many commands also support a dedicated `--help` argument (which is handled by the command executable, not by Gort). For example, typing `!gort:help --help` will return the following:

```
Provides information about a command.

If no command is specified, this will list all commands installed in Gort.

Usage:
  !gort:help [flags] [command]

Flags:
  -h, --help   Show this message and exit
```

1.7.1 Commands

If you think of Gort as a "shared command line", then commands are like the executables in your terminal.

A given command may need some additional information that would not be shared on the "shared command line", but will have to be setup by an administrator, such as an OAuth key. See *Dynamic Command Configuration* for more information on how to get this data for command execution.

1.7.2 Bundles

Bundles (or "command bundles") are the packaging unit for collections of one or more commands.

Each references a single Docker container image that contains all the binaries and other dependencies for executing one or more commands. They also include some data about the commands, including a small amount of documentation and other metadata. See *Writing a Command Bundle* for more specifics.

Bundles can be installed into Gort by an administrator (or any user with the `manage_commands` permission) using the `gort` command-line utility. See *Managing Bundles* for more on bundle installation.

Bundle Permissions and Rules

Bundles also contain a set of permissions and authorization rules for their commands. When a bundle is installed, these permissions and accompanying rules are automatically created in the Gort system.

Since permissions are namespaced to the bundle they originate from, installing a bundle's permissions will never conflict with any existing authorization system configurations you may have made. No users are automatically assigned any of these permissions.

Example: The `gort` Bundle

The gort bundle is a unique bundle in that it is effectively built into the bot. All Gort instances will have this bundle installed automatically, which is how the core permissions and authorization rules of the system come to be installed.

1.7.3 Invoking Commands

To invoke a command, like `gort:help`, you actually have a few options.

First, you can use a "command prefix", which defaults to `!`.

```
!gort:help
```

You can also interact with the bot in 1-on-1 chat, in which case you may type commands directly; everything you type to the bot is considered a command.

```
gort:help
```

Shortcuts

Fully-qualifying all command names with their bundle name (i.e., `gort:help`) can get tedious for frequently-used commands.

Fortunately, Gort allows a shortcut: if a command name happens to be unique within a Gort installation (that is, no other bundles are installed that have a command with the same name), you may type the bare command. For example, `gort:help` can be replaced with just `help`, so long as no other bundles have a `help` command.

Triggers

You can also configure a command with "triggers" that will cause it to execute when a message matches a regular expression.

You can define one or more triggers for a command using the triggers field in the bundle yaml:

::

gort_bundle_version: 1 name: my_bundle …

commands:

my_command:
description: Example trigger configuration … triggers:

- match: mytrigger
- match: (?:[0-9]{1,3}.){3}[0-9]{1,3}

The example above will match all messages containing the text "mytrigger" as well as any message containing an IPv4 address.

1.7.4 Implementation Details

Every bundle has a Docker image that contains all of its commands.

By default, the command uses the image's default entrypoint to handle commands. However, if a command has an `executable` defined, then the given binary is used instead (like a Docker –entrypoint parameter).

Any parameters you type into the command line are passed directly to the containerized binary, which can handle them just like a normal command-line execution. This allows you to implement your command using a CLI framework in any language you like.

Tip: See *Commands As Containers* for more details.

1.8 Commands As Containers

As you may recall from *Commands and Bundles*, the *command bundle* is the packaging unit for a collection of one or more actions triggerable by a user, collectively referred to as "commands". Each bundle references a single Docker container image that contains all the binaries and other dependencies for executing one or more commands.

Each command triggered by a user is executed in a container, executed from the image specified in the bundle definition.

This is actually quite a powerful approach, because it allows you to construct commands using whatever language is best suited for the job at hand.

1.8.1 Creating the Container

Each triggered command results in the spawning of a new *worker*, whose job it is to create, manage, and clean up the command container. Each worker is isolated from other workers, even of the same type, and does not share any data with other workers.

The type of worker created depends on the execution environment. In a plain Docker setup, the worker interacts with the Docker daemon to create and execute a container from the image specified in the command's bundle. In Kubernetes, the worker instead creates a Kubernetes Job resource to manage the container.

In both cases, the new command container gets several default Environment Variables that can be useful for command processes.

1.8.2 Executing the Command

However it's created, the new container executes the `executable` specified in the bundle command. If an `executable` isn't specified in the command, the container's default ENTRYPOINT is used instead.

By allowing a different `executable` value to be set for each command in a bundle, it becomes possible for a single container to contain many commands. See *Writing a Command Bundle* for more information.

1.8.3 Command Parameters

When a command is triggered, the entire string following the command is tokenized by splitting on whitespace (quoted words are kept together), and the resulting string array is passed to the container as if the executable was executed on the command line.

For example, the command executed in response to the trigger `!echo I want "to go" home` will receive the string `{ "I", "want", "to go", "home" }`.

The consequence of this design is that Gort has no required libraries or special structures that you need to adhere to. Because command executables receive parameters as a string array, just like any command executed on the command line, you can implement your commands using whatever CLI tooling is most appropriate for your language (Cobra in Go, Argparse in Python, OptionParser in Ruby, etc.).

1.8.4 Termination

When a command process completes (or is forcefully terminated, such as when it times out), Gort captures its exit code, and cleans up (removes) the container or Kubernetes resources.

The value of this exit code is used to determine whether the command was successful: an exit status of 0 indicates success; any other value is assumed to indicate an error.

1.8.5 Output

When the container runs, Gort retrieves its `stdout` and `stderr` as a single stream to form the command output. If the command terminates with an error, this is assumed to be an error message.

1.8.6 Presentation

Finally, the output is presented to the user. If the output consists of valid JSON, it will be sent to the templating engine and transformed according to the appropriate *Output Format Templates* (you don't have to worry about that now – there are perfectly reasonable defaults). Otherwise the output will be formatted for the user as simple, monospaced text.

1.8.7 Additional Reading

Discussions of constructing commands and formatting its output at *Writing a Command Bundle*

1.9 Command Environment Variables

Whenever Gort executes a command, it automatically injects a number of variables into the process' execution environment.

In general, Gort-related environment variables will be prefixed with GORT_.

- `GORT_BUNDLE`: The name of the bundle the current command is a member of.
- `GORT_CHAT_HANDLE`: The chat handle of the user executing the command. This is the bare handle: a Slack user "@user" will result in a value of "user".
- `GORT_COMMAND`: The name of the current command being executed. Does not include the bundle; that is, when executing the command twitter:tweet, for example, GORT_COMMAND will equal "tweet".
- `GORT_INVOCATION_ID`: A unique ID string uniquely identifying the invocation.
- `GORT_ROOM`: The chat room where the command was invoked. This is the bare room name: if it was executed in the "#general" channel, this value would be "general". For direct messages, this will be the string "direct".
- `GORT_SERVICE_TOKEN`: A unique token for accessing Gort's service infrastructure.
- `GORT_SERVICES_ROOT`: Host and port for accessing Gort's service infrastructure.

All other environment variables (e.g., `PATH`, `USER`, etc.) are inherited from the execution container.

1.10 Command Bundles

Bundles (or "command bundles") are the packaging unit for collections of one or more commands. They are sets of related commands that, when installed in Gort, may be executed by users from any connected (and allowed) chat service.

1.10.1 Bundle Configurations

A bundle configuration is a YAML-formatted document that describes a single bundle, including its name, description, version, container image, and one or more commands. Additionally, each command definition includes a name, description, and which binary in the container to execute when the command is triggered by a user.

Tip: See *Bundle Configurations* for more information.

1.10.2 Permissions and Rules

Importantly, each command also includes one or more *rules*, which allows operators fine-grained control over who is able to execute chat commands, extending even to control over particular invocations of chat commands.. Permissions are namespaced to the bundle they originate from, so installing a bundle's permissions will never conflict with any existing rules. Except for `admin`, permissions are never automatically assigned to users.

Tip: See *Permissions and Rules* for more information.

1.10.3 Writing a Command Bundle

Each references a single Docker container image that contains all the binaries and other dependencies for executing one or more commands. They also include some data about the commands, including a small amount of documentation and other metadata.

Tip: See *Writing a Command Bundle* for more information.

1.10.4 Managing Bundles

Bundles can be installed into Gort by an administrator (or any user with the `manage_commands` permission) using the `gort` command-line utility.

Tip: See *Managing Bundles* for more information.

1.11 Bundle Configurations

A command bundle is a set of related commands that, when installed in Gort, may be executed by users from any connected (and allowed) chat service. A bundle configuration specifies which binary to execute, and who may execute the commands (i.e., which users with which permissions).

1.11.1 A Minimal Bundle Configurations

A minimal bundle configuration looks like the following:

```
---
gort_bundle_version: 1

name: my_bundle
description: "Does bundle things"
version: 0.1
image: ubuntu:20.04

commands:
  date:
    executable: [ "/bin/date" ]
```

(continues on next page)

(continued from previous page)

```
    description: "Displays the current date and time"
    rules:
      - allow
```

The `name`, `description`, `version`, `gort_bundle_version` and `commands` fields are all required. Let's go over what these do:

- `gort_bundle_version` is the version of Gort's bundle system that your bundle was built for. The current bundle version is 1, which is used through out the rest of this document.
- `name` is the name of your bundle, which also serves as the namespace under all of the bundle's commands are installed. In this case the date command's fully-qualified name is `my_bundle:date`.
- `description` is a short, one-line description for your bundle. This will be printed along with a list of all installed bundles when a user runs the `!gort:help` command.
- `version` is the semver version number of your bundle. If you want to install a new version of bundle then you first need to uninstall the old one.
- `docker` specifies the Docker image that contains all of the bundle's commands. Limit: one image per bundle.
- `commands` are a map of zero or more commands that can be invoked in the bundle, and their associated executables. The command name, as defined here, will be the command invoked by users; it doesn't have to match the name of the binary.

Commands

Commands are possibly the most complex component of the bundle config.

As an example let's look at an excerpt from the config for an `ec2` bundle.

```
...
commands:
  find:
    executable: [ /usr/local/bin/ec2_find ]
    description: Finds an AWS EC2 instance
    rules:
      - must have ec2:view
...
```

Here you can see the command name, "find", under which are nested several fields.

- `executable` points to the command script or binary that's to be run when the command is executed. Optional. If omitted, this defaults to the image's specified ENTRYPOINT.
- `description` is a short, one-line description for the command. This is the info that will appear along with a list of commands when a user runs the `help` command.
- `rules` is a required list of strings that define what permissions are required to run the command. In this example, the `ec2:view` permission is required. See *Permissions and Rules* to learn more about rules and their construction.

Permissions

Most commands require permissions to run. Permissions are specified by in the bundle config as a list of strings at the top level. Here is another excerpt of the `ec2` config as an example.

```
---
gort_bundle_version: 1

name: ec2
description: Manage EC2 instances and related services
version: 0.4.0
permissions:
- view
- destroy
- create

...
```

In this example, three permissions are defined. When being referenced in a command rule a permission's fully-qualified name must be used: e.g., `ec2:view` or `ec2:destroy`.

Documentation fields

There are a number of fields dedicated to rendering help output via the `help` command, both for the bundle and the command.

Bundle

The following documentation fields can also be used at the top level of a bundle configuration:

- `long_description` is a separate section for a longer form description, which can include things like what configuration is required, how commands should be used, and more details about the underlying implementation.
- `author` is where the bundle author can leave their name and email address if a user needs their contact information.
- `homepage` is a URL for the bundle, typically a GitHub repository.

Command

The following documentation field can also be used in each command configuration:

- `long_description` is a long-form description used to explain details of a command that don't fit into other sections like an explanation of required arguments or about the structure of the output.

1.11.2 Bundle Installation

Command bundles can be explicitly installed using `gort bundle install`. Bundles can only be installed this way by an adequately-privileged user (an administrator or other user with the `gort:manage_bundles` permission), and are disabled by default.

See *Managing Bundles* for more information on how to explicitly install command bundles.

1.11.3 A Complete Bundle Configuration Example

Below is a complete example of a bundle configuration. In fact, it's the default bundle used by Gort to install the `gort` bundle (minus a few commands, cut for brevity).

```
---
gort_bundle_version: 1

name: gort
version: 0.0.1
author: Matt Titmus <matthew.titmus@gmail.com>
homepage: https://guide.getgort.io
description: The default command bundle.
long_description: |-
  The default command bundle, which contains the administrative commands and
  the permissions required to use them.
  Don't change or override this unless you know what you're doing.

permissions:
  - manage_commands
  - manage_groups
  - manage_roles
  - manage_users

docker:
  image: getgort/gort
  tag: v0.9.0

commands:
  bundle:
    description: "Perform operations on bundles"
    long_description: |-
      Allows you to perform bundle administration.

      Usage:
        gort:bundle [command]

      Available Commands:
        disable     Disable a bundle by name
        enable      Enable the specified version of the bundle
        info        Info a bundle
        install     Install a bundle
        list        List all bundles installed
        uninstall   Uninstall bundles
        yaml        Retrieve the raw YAML for a bundle.
```

(continues on next page)

(continued from previous page)

```
    Flags:
      -h, --help   help for bundle
  executable: [ "/bin/gort", "bundle" ]
  rules:
    - must have gort:manage_commands

version:
  description: "Displays version and build information"
  long_description: |-
    Displays version and build information.

    Usage:
      gort:version [flags]

    Flags:
      -h, --help    help for version
      -s, --short   Print only the version number
  executable: [ "/bin/gort", "version" ]
  rules:
    - allow

help:
  description: "Provides information about a command"
  long_description: |-
    Provides information about a command.

    If no command is specified, this will list all commands installed in Gort.

    Usage:
      gort:help [flags] [command]
  executable: [ "/bin/gort", "hidden", "commands" ]
  rules:
    - allow
```

1.12 Permissions and Rules

The Gort chatbot system comes equipped with a comprehensive and flexible authorization system which allows operators fine-grained control over who is able to execute chat commands, extending even to control over particular invocations of chat commands.

In this document, we will discuss the individual pieces of the authorization system and take a look at how it is used in practice.

1.12.1 Authorization Rules

At the core of Gort's authorization system are rules. Each time a user issues a chat command to Gort, rules governing the execution of that command are looked up and applied to the current invocation. If a match is found, the list of permissions the invoking user has is consulted to see if it includes the permission(s) required in the matching rule. If it does, the command is executed; if not, command processing immediately stops.

To make things concrete, we'll start with a simple authorization rule. (There is actually a separate rule language that can be used to make rather complex rules, but for now we'll start simple. Feel free to read more detailed explanations of how rules are formed. In any event, the details of the language are orthogonal to the authorization system itself.)

```
when command is gort:bundle must have gort:manage_commands
```

This is the simplest form of rule, and gives us all we need to discuss the authorization system. This rule states, in English, that for a user to execute the `gort:bundles` chat command (which allows users to enable or disable entire bundles of commands at once), they must have the `gort:manage_commands` permission granted to them.

With that rule in place, let's say I type the following command invocation in my chat application:

```
!gort:bundle disable github
```

I'm telling Gort to disable all the commands in the github bundle. In order for that command to be executed, Gort must verify, according to the rule above, that I have the `gort:manage_commands` permission. It just so happens that I do, so the command succeeds; now nobody can check how many pull requests are open on their favorite repository.

Perhaps you want to restrict the ability to disable a particularly important bundle; perhaps you've written one called prod to help manage your organization's production environment. We can add this with a new rule that matches the invocation

```
!gort:bundle disable prod
```

That rule might look like this:

```
when command is gort:bundle
  with arg[0] == "disable"
    and arg[1] == "prod"
must have site:manage_prod and gort:manage_commands
```

Here we can see a rule that applies to a very specific invocation of a command. If you have the gort:manage_commands permission, you can manipulate bundles in general, but in order to disable the prod bundle, you must have the additional `site:manage_prod` permission.

As you can imagine, the ability to define rules like this offers a lot of power. We'll talk more in depth about rules later; the remainder of this document will delve into the specifics of the authorization system itself, explaining the its components and how they all work together.

Rules can be viewed, created, and deleted using the `gort:rules` command. In particular, simple rules of the form when command is must have can be created thusly:

```
!gort:rule create <COMMAND> <PERMISSION>
```

Note that both and must exist, and be typed as fully-qualified names.

1.12.2 Components of the Authorization System

Permissions

Permissions are at the base of Gort's authorization system. They act as a kind of token; you can carry out certain actions if you possess the correct token(s).

You will notice that permissions have a structure: `gort:manage_commands`, `site:manage_prod`, etc. Permissions are *namespaced*; here we have a `manage_commands` permission in the `operable` namespace, and a `manage_prod` permission in the `site` namespace. In general, every bundle of commands defines its own permission namespace. This allows bundle authors the flexibility to define permissions that are used by commands in the bundle without worrying about conflicting with permissions from any other bundles that happen to be installed on a Gort system. We can have an `gort:manage_commands` permission as well as a `site:manage_commands` permission without any problems.

There are two ways that permissions can be created. The first is through bundle installation. All command bundles have the option to define permissions and authorization rules to help bootstrap the bundle in a Gort system. The operator is not under any obligation to use these rules or permissions, and is free to define their own, but they are always installed with the bundle.

The second way that permissions can be created is directly by the Gort operator. This is where the special site namespace comes into play. site is unique; it is the only permission namespace that is not associated with a corresponding command bundle. All permissions created by operators are part of the site namespace. It is the mechanism by which the permission scheme may be customized to the needs of the operator's unique environment and use cases.

Roles

Moving up from permissions, we arrive at roles, which are collections of associated permissions. While permissions can be created when you install command bundles, roles are something purely under your control as a Gort operator; you create them and you manage them.

Users

In order for permissions to be useful, we have to have a way to associate them with people invoking commands. The Gort user is the unique identity to which permissions are ultimately attached.

Each person that can interact with Gort has an associated user account. This is also the identity with which a person will interact with Gort's REST API.

It is important to understand that this "Gort User" is not the same as a person's "handle" in a particular chat system. In fact, a Gort user can be associated with multiple handles from different chat systems. For instance, I may be `@matt` in Slack, but `mtitmus` in Gort. Gort can recognize this and map these various chat handles back to the same Gort user, allowing authorization to be managed centrally and independently of which chat system is in use.

Users are scoped to the entire Gort installation; that is, there is no higher-level namespace (e.g., "organization") into which users are grouped.

Groups

Finally, Gort groups collect Gort users together. Any number of users may be in a group, but only users may be members of groups.

1.12.3 Bringing It All Together

Now that you know about permissions, roles, users, and groups, how do you use them?

We know that roles are collections of permissions, and groups are collections of users, but that ultimately, somehow, permissions become associated with users. This missing link here is that roles can be granted to groups.

Thus, a user has all the permissions in all the roles granted to all the groups of which she is a member.

To grant a permission to a user, then, the user must be placed into a group that has been granted a role that contains that permission. While this might seem a bit cumbersome from the perspective of a single user and a single permission, it makes global management easier; it frees you to think in terms of the higher-level constructs of roles and groups, without having to worry about "exceptions to the rule" like individual users being directly granted a permission, or potentially complicated group hierarchies.

As an example, let's look at how we might set up a Gort system to grant permissions for the mist EC2 command bundle. For this demonstration, let's say we have three users: Alice, Bob, and Charlie. Furthermore, let's say that Alice is on our Operations team, while Bob and Charlie are on the Development team. Let's also stipulate that everyone on the operations team should be able to perform any action with Mist, while developers start out with read-only permissions.

Looking at Mist's bundle configuration, we see it declares the following permissions:

- `mist:view`
- `mist:change-state`
- `mist:destroy`
- `mist:create`
- `mist:manage-tags`
- `mist:change-acl`

It looks like we'll want to give operations folks all of these permissions, and developers only mist:view. Let's set up some roles to express this.

First a mist_admin role, with all the mist permissions:

```
gort role create mist_admin
gort role grant mist_admin mist:view
gort role grant mist_admin mist:change_state
gort role grant mist_admin mist:destroy
gort role grant mist_admin mist:create
gort role grant mist_admin mist:manage-tags
gort role grant mist_admin mist:change-acl
```

And now, a mist_read_only role:

```
gort role create mist_read_only
gort role grant mist_read_only mist:view
```

Now we have our roles, but we have nothing to grant them to. Let's create some groups.

```
gort group create operations
gort group create developers
```

Now let's grant the roles to our new groups.

```
gort group grant operations mist_admin
gort group grant developers mist_read_only
```

We're almost there. We have the groundwork laid; all that remains is to add our users.

```
gort group add operations alice
gort group add developers bob charlie
```

Any changes to the permission structure take effect immediately. If the `mist:view` permission is removed from the `mist_read_only` role, Bob and Charlie immediately lose the ability to run commands that require that permission (unless they happen to also be members of another group that has the permission via some other role). Similarly, if Danielle is added to the operations group, she immediately has all the mist permissions.

Note also that all authorization rules are written in terms of permissions, and not roles,

1.13 Writing a Command Bundle

Documentation coming soon!

1.14 Installing Your First Command Bundle

Once you have a command bundle, you'll need to install and enable it for it to be useful.

1.14.1 Creating Your Bundle

To create a bundle, you first need a bundle configuration: a YAML-formatted file that supplies Gort with all of the information it needs to install and execute commands in your bundle.

For a detailed description of bundle configurations, go to the *Bundle Configurations* section in the documentation.

For our example we will be using the following config. It's a simple bundle with only one un-enforcing command. Just create a file named `my_bundle.yaml` and paste the contents below into it. It doesn't actually matter what you name the file, just make sure that it is properly-formatted YAML and that it has the correct extension, `.yaml` or `.yml`.

my_bundle.yaml

```
---
gort_bundle_version: 1

name: my_bundle
description: My bundle
version: "0.0.1"
image: ubuntu:20.04

commands:
  date:
```

(continues on next page)

(continued from previous page)

```
    executable: [ "/bin/date" ]
    rules:
      - allow
```

If your commands specify any rules, other than the special "allow" rule, you will need to make sure the proper grants are in place. Check out permissions-and-rules` to learn more.

Bundles are most easily created with Gort's command line interface: `gort`. To create your bundle just type the following at the command prompt. Adjust the `my_bundle.yaml` bit to point to the config file that you created.

```
$ gort bundle install my_bundle.yaml
```

And there you have it! Bundle created. Now let's see about enabling it.

1.14.2 Enabling Your Bundle

By default any bundles added to Gort are added in the disabled state. This way you don't have to worry about accidentally exposing commands that aren't fully configured, or otherwise not ready for production.

Enabling commands is easy though. We'll use the `gort` CLI! By default the highest installed version of a bundle will be enabled. To enable a different version just pass the version to enable to `gort`.

```
$ gort bundle enable my_bundle

$ gort bundle enable my_bundle 0.0.1
```

1.14.3 Success!

That's it! You've successfully installed your first bundle. If everything went properly you should see the new command in Gort's command list.

```
User:
!help

Gort:
I know about these commands:

- gort:bundle
- gort:group
- gort:help
- gort:role
- gort:user
- gort:version
- my_bundle:date

Try calling `gort:help COMMAND` to find out more.
```

Finally, you should be able to run it!

```
User:
!date
```

(continues on next page)

(continued from previous page)

```
Gort:
Wed Nov 17 00:10:24 UTC 2021
```

1.15 Managing Bundles

This document details the commands used to manage bundles.

Currently, all command bundles are executed on the same machine as the Gort Controller. In a future release, support for "relays" will be added, which will allow commands to be executed on different machines running a Relay process.

1.15.1 Prerequisites

For simplicity we will be using the `gort` command-line utility to demonstrate bundle management. Bundle management mostly involves use of the `bundle` subcommand. However, you aren't explicitly required to use `gort` to manage bundles: you can use the `!gort:bundle` command, or you can even make calls directly to the API if you like.

The remainder of this page assumes that you have a working Gort Controller and the `gort` utility.

1.15.2 Installing Bundles

Bundles are installed by uploading bundle configurations to Gort, which then registers the bundle. Registration includes the creation of the permissions declared by the bundle, as well as any default rules specified in the bundle's metadata.

Importantly, after installation your bundle command will be available, but may not be usable yet. Before anyone can execute the new commands, make sure their user permissions are set properly. See *Permissions and Rules* to learn more.

Bundles are installed via the `bundle install` sub-command in `gort`.

```
$ gort bundle install --help
Install a bundle from a bundle file.

When using this command, you must provide the path to the file, as follows:

  gort bundle install /path/to/my/bundle/config.yaml

Usage:
  gort bundle install [flags] config_path

Flags:
  -h, --help   Show this message and exit

Global Flags:
  -P, --profile string   The Gort profile within the config file to use
```

1.15.3 The Bundle Configuration File

The only required argument for `gort bundle install` is the path to the bundle's config file.

All bundles have a `config` file, a yaml-formatted document that contains information for installing and executing commands in your bundle. To learn more about configuration files take a look at Bundle Configurations.

We won't discuss bundle configurations in detail here, but minimally each must contain:

- `name` - The name of your bundle.
- `version` - The version of your bundle in semver format.
- `gort_bundle_version` - The version of the config file format (currently only version 1 is used).
- `commands` - A map of commands to be included in the bundle.

A minimal bundle config might look something like this:

```
---
gort_bundle_version: 1

name: my_bundle
version: 0.0.1
description: My bundle
commands:
  date:
    executable: [ /bin/date ]
    rules:
      - allow
```

The command to install the bundle would be something like `gort bundle install /path/to/my_bundle.yml`.

> **Attention:** Bundles are disabled when first installed. You must enable them before you can use them.

1.15.4 Enabling and Disabling Bundle Versions

When a new version of a bundle is installed, it's disabled by default. Only one version can be enabled at a time and a version must be explicitly enabled before Gort will use it to handle commands.

Enabling and disabling bundle versions is a straight-forward process.

For example, if you already have version 1.0.0 of "my-bundle" installed:

```
$ go run . bundle versions my-bundle
BUNDLE      VERSION    STATUS
my-bundle   1.0.0      Enabled
```

You can install version 2.0.0 in a reasonably straight-forward manner:

```
$ gort bundle install /path/to/my-bundle/v2/config.yaml
$ gort bundle versions my-bundle
BUNDLE      VERSION    STATUS
my-bundle   1.0.0      Enabled
my-bundle   2.0.0      Disabled
```

As always, a newly-installed bundle is disabled by default. At this point, invoking any commands from the "my-bundle" bundle will still execute from version 1.0.0.

```
$ gort bundle info my-bundle
Name: gort
Versions: 1.0.0, 2.0.0
Status: Enabled
Enabled Version: 1.0.0
Commands: date
Permissions:
```

Switching to the new version is as simple as:

```
$ gort bundle enable my-bundle 2.0.0
$ gort bundle versions my-bundle
BUNDLE      VERSION    STATUS
my-bundle   1.0.0      Disabled
my-bundle   2.0.0      Enabled
```

From now on, any "my-bundle" command invocations will execute from version 2.0.0, using whatever access rules have been defined in that version.

And if you decide you don't like version 2.0.0 for any reason, you can always switch back to 1.0.0:

```
$ gort bundle enable my-bundle 1.0.0
$ gort bundle versions my-bundle
BUNDLE      VERSION    STATUS
my-bundle   1.0.0      Enabled
my-bundle   2.0.0      Disabled
```

Assuming that you have the required access, you can also enable and disable bundles using the `gort:bundle` chat command.

1.15.5 Uninstalling Bundles and Bundle Versions

You may uninstall a specific version of a bundle or all versions of a bundle.

Uninstalling a specific version will remove rules and permissions only associated with that version. Uninstalling all bundle versions involves complete removal of all authorization rules governing its commands as well as deletion of all the bundle's permissions. You can re-install to restore the bundle permissions and rules. If you only wish to disable a bundle, see "Enabling and Disabling Bundle Versions" above.

Uninstalling a bundle requires the use of the `gort bundle uninstall` subcommand.

```
$ gort bundle uninstall --help
Uninstall bundles.

Usage:
  gort bundle uninstall [flags] bundle_name version

Flags:
  -c, --clean         Uninstall all disabled bundle versions
  -x, --incompatible  Uninstall all incompatible versions of the bundle
  -a, --all           Uninstall all versions of the bundle
  --help              Show this message and exit.
```

(continues on next page)

(continued from previous page)

```
Global Flags:
  -P, --profile string   The Gort profile within the config file to use
```

Uninstalling a bundle version

Importantly, enabled bundles cannot be uninstalled.

```
$ gort bundle uninstall date 0.1.0
Usage: gort bundle uninstall [OPTIONS] NAME [VERSION]

Error: Invalid value for "version": Cannot uninstall enabled version. Please disable the␣
↪bundle first
```

Before any bundle can be uninstalled, it must first be disabled.

```
$ gort bundle disable my_bundle 0.1.0
$ gort bundle uninstall my_bundle 0.1.0
Uninstalled my_bundle 0.1.0
```

Uninstalling all versions of a bundle

Since uninstalling all versions of a bundle can be very destructive, you must pass the `--all` flag to `gort`, or your request will fail.

```
$ gort bundle uninstall date
Error: Can't uninstall without specifying a version, or --incompatible, --all, --clean
```

It would seem that `gort bundle uninstall` needs either a version number, or an `--all` flag.

```
$ gort bundle uninstall date --all
Usage: gort bundle uninstall [OPTIONS] NAME [VERSION]

Error: date 0.1.0 is currently enabled. Please disable the bundle first.
```

This time the uninstallation failed because the bundle is still enabled.

```
$ gort bundle disable date
Disabled date

$ gort bundle uninstall date --all
Uninstalled date 0.0.1
Uninstalled date 0.0.1
Uninstalled date 0.1.0
```

Success at last.

1.16 Command Execution Rules

1.16.1 Rule Structure

Rules help Gort to determine who is able to perform what task. Gort rules follow a specific format. The rule structure describes what command is executed and what permission is needed in order to execute the command. If a user does not have the specified permission, the user is not able to execute the command.

The general form of a command is:

```
COMMAND [when CONDITIONS] [allow|must have PERMISSION]
```

- Command: The command indicates the command that's affected by the rule. Commands are referred to as `bundle_name:command_name`. For example, the `splitecho` command in the `echo` bundle would be referenced as `echo:splitecho`.
- Conditions: The (optional) conditions clause indicates when the rule should be is applied. It starts with the keyword `when`, and consists of one or more logical statements. See below for more detail. If a rule contains no conditions, it *always* applies when the command is used.
- Permissions: The permissions clause indicates the permissions that a user must have to execute the command when the conditions are met. It begins with the phrase `must have`. Like commands, permissions are namespaced: `bundle_name:permission_name`.
- Allow: The standard permissions clause may be replaced with the `allow` keyword, which can be used to allow a command meeting the rule conditions to be executed by any Gort user. `allow` is used in lieu of a permissions clause, and may not be accompanied with any other keyword or phrase.

A basic example of a rule is:

```
foo:bar with option[delete] == true must have foo:destroy
```

This rule states that a user attempting to use the `bar` command from the `foo` bundle, with the `delete` flag set, must have the `foo:destroy` permission.

Rules can also be used to grant broad permissions by using the `allow` keyword:

```
foo:biz allow
```

This is the simplest possible rule, which allows any user to use the `foo:biz` command under all conditions.

1.16.2 The Conditions Clause

The conditions rule clause begins with the keyword `with`.

The `conditions` clause can match specific command parameter, allowing you to create rules that apply under very specific invocations of a command.

Options and Arguments

Any command can have two kinds of command parameters: *options*, are a general term for command flags and switches, and *arguments*, which are the main inputs into the command.

For example, given the following command:

```
curl -I --capath /home http://example.com
```

The options are `-I` and `--capath /home`, and the parameter is `http://example.com`

Testing Options and Arguments

Each rule can reference two pre-defined two data structures: `option` and `arg`.

- `option`: A map or dict of the commands options. The value of specific options can be accessed using standard map notation.
- `arg`: A (zero-indexed) list of the command arguments. Specific arguments can be accessed using standard map notation.

Logical Operators

Individual (non-collection) values can logically evaluated using the <, >, == and `!=` operators:

- `with option["dry-run"] == true`

Regular expressions may also be used.

- `with option["set"] == /.*/`

Not only can specific `arg` positions be referenced by index, the entire parameter list can also be evaluated as a string by omitting the index. For example, given the following command:

```
echo foo bar
```

The following statements are equivalent:

- `foo:bar with arg[0] == 'foo' and arg[1] == 'bar' allow`
- `foo:bar with arg == 'foo bar' allow`

Sets

Options and arguments can be tested against sets of conditions by using one of the following keywords:

- `in` – Applied to a non-collection value, resolves to true if and only if the value matches a value in the set.
- `any`, `in` – Applied to a collection value, resolves to true if and only if any value in the collection matches a value in the set.
- `all`, `in`– Applied to a collection value, resolves to true if and only if all value in the collection match a value in the set.

Conditional sets can include zero or more values between square brackets. Regular expressions are also legal members and will be evaluated accordingly. Some examples are:

- `foo:bar with arg[0] in ['baz', false, 100] must have foo:read`
- `foo:bar with option["foo"] in ["foo", "bar"] allow`

- `foo:bar with any option == /^prod.*/ must have foo:read`
- `foo:bar with any arg in ['wubba'] must have foo:read`
- `foo:bar with any arg in ['wubba', /^f.*/, 10] must have foo:read`
- `foo:bar with all arg in [10, 'baz', 'wubba'] must have foo:read`
- `foo:bar with all option < 10 must have foo:read`
- `foo:bar with all option in ['staging', 'list'] must have foo:read`

Combining Qualifiers

Arbitrarily long compound qualifiers can be constructed using the `and` and/or `or` keywords, so your rules can be as simple or as complicated as you need them to be. For example, the following rule is legal:

```
foo:bar with arg=="prod" and option["delete"] == true or option["set"] == /.*/ must have␣
↪foo:destroy
```

1.16.3 Permissions

The permissions clause is where you state any permissions that are required to execute the command. The beginning of the permissions clause is indicated by the phrase `must have`.

Like the conditions clause, it can be arbitrarily complex, and can a single permission, a specific combination of permissions combination, or a list of permissions. It supports the same operations as well:

- `or`
- `and`
- `any in`
- `all in`
- `allow`

For example, the following are rule examples with valid permission settings:

- `foo:baz with option[delete] == true must have foo:write and site:admin`
- `foo:export must have all in [foo:write, site:ops] or any in [site:admin, site:management]`
- `foo:bar must have any in [foo:read, foo:write]`
- `foo:qux must have all in [foo:write, site:ops] and any in [site:admin, site:management]`
- `foo:biz allow`

Note the special `allow` keyword, which can be used in lieu of a permissions clause to allow a command to be executed by any registered user in Gort.

1.16.4 Formal Definition

Gort's command execution rule syntax may seem quite English-like, but it's actually a well-structured syntax describable as a formal context-free grammar.

For your reference, we have included the notation for Gort's command execution using Backus–Naur form, a metasyntax notation for context-free grammars that's often used to describe the syntax of computing languages used in computing.

```
<rule>               ::= <arguments> " " <permissions> | <permissions> ;

<arguments>          ::= "with " <argument> ;

<argument>           ::= <argument_part> | <argument_part> " " <conditional> " "
↪<argument> ;

<argument_part>      ::= <argument_single> | <argument_plural> ;

<argument_single>    ::= <variable_single> " " <operator> " " <variable_single> ;

<argument_plural>    ::= "all " <defined_set> " " <operator_set> " " <variable_set> |
↪"any " <defined_set> " " <operator_set> " " <variable_set> ;

<defined_set>        ::= "arg" | "option" ;

<operator_set>       ::= "in" | <operator> ;

<operator>           ::= "==" | "!=" | "<" | "<=" | ">" | ">=" ;

<variable_single>    ::= "arg[" <literal_integer> "]" | "option[" <literal_string> "]" |
↪<literal> ;

<variable_set>       ::= "[" <variable_list> "]" ;

<variable_list>      ::= <variable_single> | <variable_single> "," <variable_set> ;

<conditional>        ::= "and" | "or" ;

<permissions>        ::= "allow" | "must have " <permission>  ;

<permission>         ::= <permission_part> | <permission_part> " " <conditional> " "
↪<permission> ;

<permission_part>    ::= <permission_single> | <permission_plural> ;

<permission_single>  ::= <name> ":" <name> ;

<permission_plural>  ::= "all in " <permission_set> | "any in " <permission_set> ;

<permission_list>    ::= <permission_single> | <permission_single> ", " <permission_list>␣
↪;

<permission_set>     ::= "[" <permission_list> "]" ;

<literal>            ::= <literal_bool> | <literal_string> | <literal_number> | <literal_
```

(continues on next page)

(continued from previous page)

```
↪regex> ;

<literal_bool>      ::= "true" | "false" ;

<literal_string>    ::= '"' <string> '"' | "'" <string> "'" ;

<literal_number>    ::= <literal_integer> | <literal_float> ;

<literal_regex>     ::= "/" <regex> "/" ;

<literal_integer>   ::= <digit>+ ;

<literal_float>     ::= <digit>+ "." <digit>+ ;

<digit>             ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

<letter>            ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K"␣
↪| "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" |
↪"Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n
↪" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;

<symbol>            ::=  "|" | " " | "!" | "#" | "$" | "%" | "&" | "(" | ")" | "*" | "+"␣
↪| "," | "-" | "." | "/" | ":" | ";" | ">" | "=" | "<" | "?" | "@" | "[" | "\" | "]" |
↪"^" | "_" | "`" | "{" | "}" | "~"

<rune>              ::= <letter> | <digit> | <symbol> ;

<string>             ::= <rune>+

<character>         ::= <letter> | <digit> ;

<name>              ::= <character>+ ;
```

1.16.5 Todo

The following list includes some features that are considering adding to the command execution rules language:

1. Built-in/standard permissions (especially for Gort administration actions)
2. Syntax to access user/group/adapter attributes in rule conditions
3. Built-in support functions in conditions?

If any of these is particularly important to you, or if you have an idea for a feature not listed here, please feel free to create an issue.

1.17 Dynamic Command Configuration

Commands often require access to runtime variables, particularly when interacting with external services, ranging from mundane values (like the URL of a downstream resource) to highly sensitive information (like database passwords or access tokens).

One (terrible) way of providing these values to a command might be baking them into the bundle's container image, but that has two problems:

1. It limits command reusability by requiring all users to use the same configuration.
2. It creates a security risk by making potentially sensitive values accessible to anybody who with access to the image.

Dynamic command configuration solves this problem by making it possible to securely store configuration information so that it can be injected into worker containers at runtime as specifically-named environment variables or files.

Note: Dynamic command configuration shouldn't be confused with the *config.yaml* file that defines the commands, rules, and permissions present in each command bundle. That configuration is effectively static. The configuration we are concerned with is for the *execution* of individual commands.

It's also dynamic in the sense that it can be changed on-the-fly by Gort administrators, with the changes taking effect nearly instantaneously without restarting any applications.

Warning: Currently dynamic configurations can only be stored in plain text in the Gort database. While a secure backend is currently in development, it is currently recommended that dynamic configuration not be used to inject highly sensitive values.

1.17.1 Core Concepts

Dynamic configuration allows users to define one or more key-value pairs that are injected as variables into the execution environment of a command. The key is usually a simple name, like "url" or "email".

Note: All configurations belong to a specific bundle. Dynamic configurations cannot be assigned to multiple bundles.

The actual environment variable name is constructed by converting this key into an all-caps name using the pattern `BUNDLE_KEY`. Dashes are also converted into underscores.

For example, a dynamic configuration named "user-email" that belongs to the "testing" bundle will be injected into the command environment as `TESTING_USER_EMAIL`.

A command can then access it as an environment variable (e.g. `ENV['TESTING_USER_EMAIL']` in Ruby, `os.environ['TESTING_USER_EMAIL']` in Python, etc.)

Warning: Each command in a bundle will receive the same dynamic configuration environment. There is not currently a way to allow one command to receive one set of variables while another receives a different set.

1.17.2 Layers

There are four layers:

bundle
: Configurations at the *bundle* layer are applied to all of the commands in its respective *command bundle*. This layer can be overridden by any other layer.

channel
: Configurations made at the *channel* layer are applied to all commands in its bundle executed in a specific channel. This layer can override *bundle* layer configurations, and can in turn be overridden by *group* or *user* layer configurations.

group
: Configurations made at the *group* layer are applied to all commands in its bundle executed by a given *group*. This layer can override *bundle* and *channel* layer configurations, and can be overridden by *user* layer.

user
: Configurations made at the *user* layer are applied to all commands in its bundle executed by a particular user. This layer can be override any other layer.

Layer Overriding

For any given bundle, the same configuration can be defined in multiple layers. In this case, the layer with the highest precedence is the one that's used.

The layer precedence order is as follows:

1. User
2. Group
3. Channel
4. Bundle

This allows you to, for example, define a default set of user credentials at the bundle level while allowing a specific group and even specific users to define their own credentials for more specialized purposes.

1.17.3 Managing Dynamic Configuration Values

Dynamic configurations can be managed using the *gort config* commands. There are three:

1. `gort config get`: Used to retrieve one or more non-secret configuration values.
2. `gort config set`: Used to create or update a configuration value.
3. `gort config delete`: Used to delete a configuration value.

The flags accepted by each of these commands are as follows

Flags	Get	Set	Delete	Description
`--bundle`	R	R	R	The name of the bundle to configure.
`--layer`	O	O	O	One of: `bundle`, `channel`, `group`, `user`. Default: `bundle`.
`--owner`	R	R	R	The owning channel, group, or user.
`--key`	R	R	R	The name of the configuration.
`--secret`	n/a	O	n/a	Makes a configuration value secret. Secret values cannot be read using `gort config get`.

R=Required. O=Optional.

1.17.4 Future Steps

This feature is in a state of minimal viability, and many new features are planned for it. Including:

1. The development of an optional secure backend. Initially this will support Hashicorp Vault.
2. Allowing configuration value to be defined as code.
3. Allowing configuration values to be injected as files (and not just environment variables).

1.18 Output Format Templates

Output format templating is a powerful feature that allows you to control the look and feel (and, to some degree, content) of any information sent to users. Both Gort system messages and command output support templates for customization (within the constraints imposed by a given chat provider).

1.18.1 The four template types

The are four template types:

- *Command templates*, which are used to format the outputs from successfully executed commands.
- *Command error templates*, which are used to format the error messages produced by commands that exit with a non-zero status.
- *Message templates*, which are used to format standard informative (non-error) messages from the Gort system (not commands).
- *Message error templates*, which are used to format error messages from the Gort system (not commands).

Each of these have default values built into Gort, but each may be customized via the `templates` block of the Gort configuration. Furthermore, the *command* and *command error* templates may be further customized per bundle, or even per command.

1.18.2 Template Basic Format

Gort templates use Go's template syntax to format output in a chat-agnostic way.

For example, a very simple *command template* might look something like the following:

```
{{ text | monospace true }}{{ .Response.Out }}{{ endtext }}
```

This template emits the command's response (`.Response.Out`) as monospaced text, which may look something like the following:

A slightly more complicated template, this one a *command error template* (actually the default), is shown below.

```
{{ header | color "#FF0000" | title .Response.Title }}
{{ text }}The pipeline failed planning the invocation:{{ endtext }}
{{ text | monospace true }}{{ .Request.Bundle.Name }}:{{ .Request.Command.Name }} {{ .
↪Request.Parameters }}{{ endtext }}
{{ text }}The specific error was:{{ endtext }}
{{ text | monospace true }}{{ .Response.Out }}{{ endtext }}
```

This one includes a header with a color and title, as well as some alternating monospaced and standard text. In this case, this will format a command error something like the following:

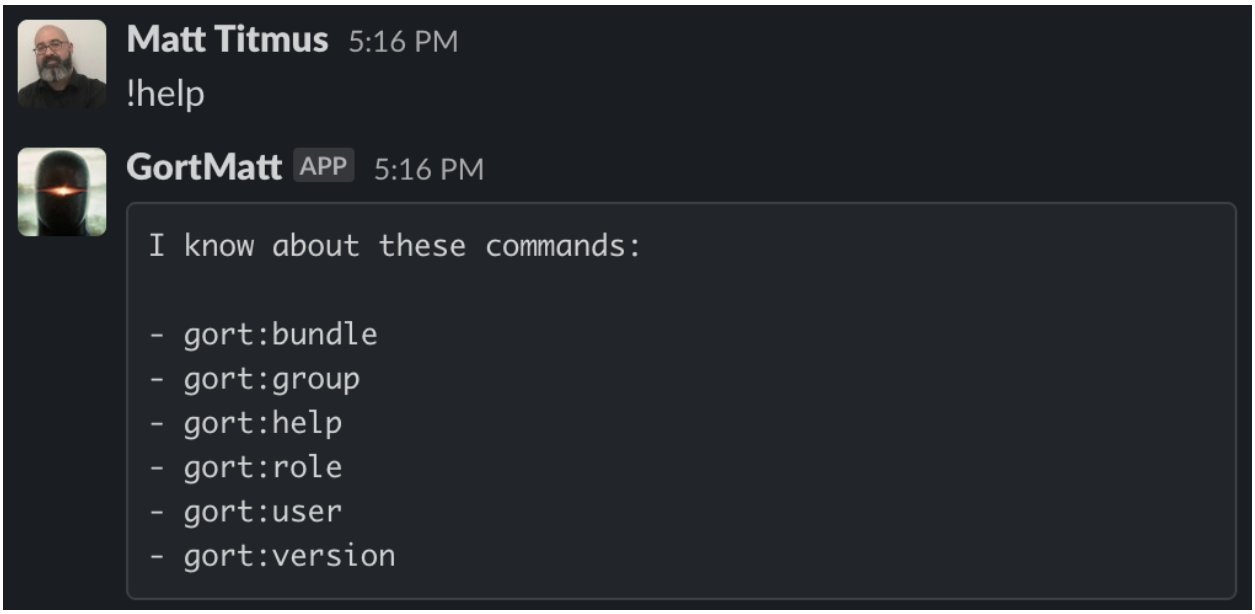
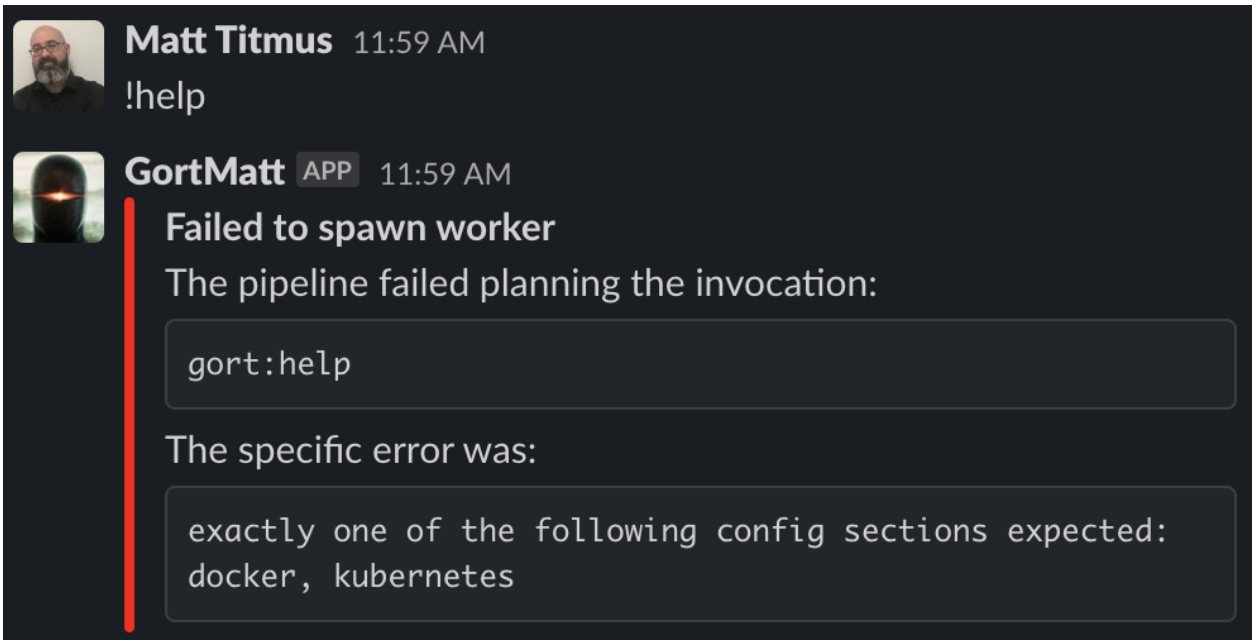


Fig. 2: Monospaced command output



Sure that's nice and all, but what's all this `.Response` stuff? That's part of what's called the "response envelope", a data structure that's accessible from any template, which makes available all of the data and metadata around one command request, execution, and response. The response envelope is discussed in detail in *The Response Envelope*.

The available template tags and functions are also fully presented in *Template Functions*.

1.19 The Response Envelope

The response envelope encapsulates all of the data and metadata around a command request, execution, and response. It's passed into the templating engine, where it can be accessed directly by templates.

The response envelope consists of four major components:

- `.Request` – This describes the original request used to execute the command, and contains all of the original command values, and some data about the user and adapter.
- `.Response` – Contains the textual response emitted by the command.
- `.Data` – This object contains metadata about the command execution, including its duration and exit status.
- `.Payload` – If the command output is valid JSON, it will be unmarshalled and placed here to be accessed by templates. Non-JSON output will also be placed here, as a plain string.

These components (and any sub-components, where relevant) are detailed below.

1.19.1 .Request

The `.Request` object represents the original command request used to execute the command. It contains the original command values as well as the user and adapter data.

Syntax	Type	Description
`.Request.Adapter`	string	The name of the adapter this request originated from
`.Request.ChannelID`	string	The provider ID of the channel that the request originated in
`.Request.Parameters`	[]string	Tokenized command parameters
`.Request.RequestID`	int64	A unique requestID
`.Request.Timestamp`	time.Time	The time this request was triggered
`.Request.UserID`	string	The provider ID of user making this request
`.Request.UserEmail`	string	The email address associated with the user making the request
`.Request.UserName`	string	The Gort username of the user making the request

1.19.2 .Request.Bundle

The `.Request.Bundle` object represents the bundle that owns the command.

1.19.3 .Request.Command

The `.Request.Command` object represents the command definition.

1.19.4 .Response

The `.Response` object contains the response text emitted by an executed command.

Syntax	Type	Description
`.Response.Lines`	[]string	The command output (from both stdout and stderr) as a string slice, delimitted along newlines.
`.Response.Out`	string	The command output as a single block of text, with lines joined with newlines.
`.Response.Structured`	bool	`true` if the command output is valid JSON. If so, then it also be unmarshalled as `.Payload`; else `.Payload` will be a string (equal to `.Response.Out`).
`.Response.Title`	string	A title. Usually only set by the relay for certain internally-detected errors. Generally a short description of the result.

1.19.5 .Data

The `.Data` object contains data about the command execution, including its duration and exit status. If the relay sets an an explicit internal error, it will be here as well.

Syntax	Type	Description
`.Data.Duration`	time.Duration	Duration is how long the command required to execute.
`.Data.ExitCode`	int16	ExitCode is the exit code reported by the command.
`.Data.Error`	error	Error is set by the relay under certain internal error conditions.

1.19.6 .Payload

`.Payload` includes the command output. It's a very special animal, because its contents can very according to the contents and format of the response returned by the command.

Specifically, if the command output is formatted as structured JSON, the output will be unmarshalled and made accessible via `.Payload` as if were any other object. Additionally, the value of `.Response.Structured` will be `true`.

For example, if the contents of the command response are as follows:

```
{
   "User":"Michael Scott",
   "Company":"Dunder Mifflin",
   "Results":[
      {
         "Name":"Bond",
         "Reviews":523,
         "Description":"Bond paper is stronger and more durable than the average sheet␣
↪of paper.",
         "Image":"https://dunder-mifflin/bond.jpg"
      }, {
         "Name":"Gloss coated",
         "Reviews":1234,
         "Description":"Gloss paper is typically used for flyers and brochures as it has␣
↪a high shine.",
         "Image":"https://dunder-mifflin/gloss-coated.jpg"
      }
   ]
}
```

So a template containing the instructions `{{.Payload.User}}, {{.Payload.Company}}` would resolve as `Michael Scott, Dunder Mifflin`.

If the response isn't structured, `.Response.Structured` will be `false`, and `.Payload` will be a standard string equal to `.Response.Out`.

1.20 Template Functions

Output format templating is a powerful feature that allows you to control the look and feel (and, to some degree, content) of any information sent to users. Gort templates are based on Go templates, and support all of their features, including variables, logic, and flow control.

There are three main types of elements used to construct elements:

- **Tags:** Tags (or "actions") represents visual elements or directives, such as `{{ text }}` or `{{ image }}`. All text and directives must be enclosed within (or between) tags.
- **Functions:** Functions are used to modify the contents or behavior of tags. They're called using pipes (`|`) within a tag. For example, `{{ image "foo.jpg" | thumbnail true }}` turns an image into a thumbnail image. Some functions can only be used with specific tags. Sprig functions are also supported.
- **Fields:** A call into a data value, typically the the Response Envelope.

The supported elements are detailed below.

1.20.1 Tags (Actions)

`{{alt}}`

Provides alternative text to be shown if other elements in a message cannot be rendered. Only the first instance of `{{alt}}` will be shown.

Example: `{{ alt "This is alternative text." }}`

`{{divider}}`

Emits a simple divider, used to break up blocks.

`{{ header }}`

Can be used to decorate or modify the behavior of the entire template. If a `{{ header }}` is used, it must be the first line of the template.

Function	Description	Example
`color`	Adds a colored sidebar to the output block. Must be a hexadecimal string with the format `#RRGGBB`.	`{{ header | color "#FF0000" }}`
`title`	Adds a title to the output block.	`{{ header | title "Error?" }}`

{{text}} and {{endtext}}

Used to describe a text block or element. They may be used inside of a {{section}}/{{endsection}} pair.

Example: {{ text }}This is a plain text block.{{ endtext }}

Function	Description	Example
inline	Makes the text "inline" (in the Discord sense). If true, a title is also expected.	{{ text | inline true | title "Favorite Food" }}Pizza{{ endtext }}
monospace	All text in the block is monospaced.	{{ text | monospace true }}THIS IS CODE.{{ endtext }}
title	Adds a title to the text block.	{{ text | title "Favorite Food" }}Pizza{{ endtext }}

{{section}} and {{endsection}}

Sections can be used to group elements together. These are only supported in Slack; they are ignored in Discord.

These are often used with a range loop of some kind over a collection:

```
{{ text }}Here are your results:{{ endtext }}
{{ range $index, $loc := .Payload.Locations }}
{{ section }}
{{ text | title $loc.Title | inline true }}$loc.Name{{ endtext }}
{{ image $loc.Image | thumbnail true }}
{{ endsection }}
{{ end }}
```

{{image}}

Outputs an image whose URL is specified in the argument. They may be used inside of a {{section}}/{{endsection}} pair.

Example: {{ image "https://example.com/img/image1.png" }}

Function	Description	Example
thumbnail	Causes the image to be presented as a thumbnail (usually for a block or section).	{{ image .Payload.Image | thumbnail true }}

1.20.2 Additional Functions

In addition to the above, all of the Sprig library functions are supported as well.

1.21 Going Forward: Features to Look Forward To

The following is a list of major features that are planned for version 1.0:

1. **Triggers.** Allow Gort to execute existing bundled commands in response to non-command chat input. (Priority: very high)
2. **Dynamic command configuration.** Allow dynamic values like tokens and passwords to be securely passed to commands at runtime. (Priority: very high)
3. **Custom webhooks.** Expose custom (auth-gated) RESTful endpoints that can be used to execute existing bundled commands. (Priority: high)
4. **Easing Rule and Permission Design**. Addition of a command-line rule tester. (Priority: high)
5. **Two-factor authentication.** Add support for two-factor authentication. (Priority: high)
6. **Dual authorization.** Add support for dual (two-person) authorization. (Priority: high)
7. **Simplifying audit log access.** Create a `gort logs` command <https://github.com/getgort/gort/issues/156>`__. (Priority: medium-high)
8. **Internal key/value store.** Allow commands access to an internally-scoped key/value store to maintain state, a little like HTTP cookies. (Priority: medium)
9. **Support for other chat platforms.** Microsoft Teams first, others TBD (Priority: medium-low)

Have any other ideas? We want to hear them!

- genindex
- search