# Honeydipper Configurations

***Release 1.0.0***

**Jul 07, 2020**

# Honeydipper

CHAPTER 1

# Honeydipper

CircleCI



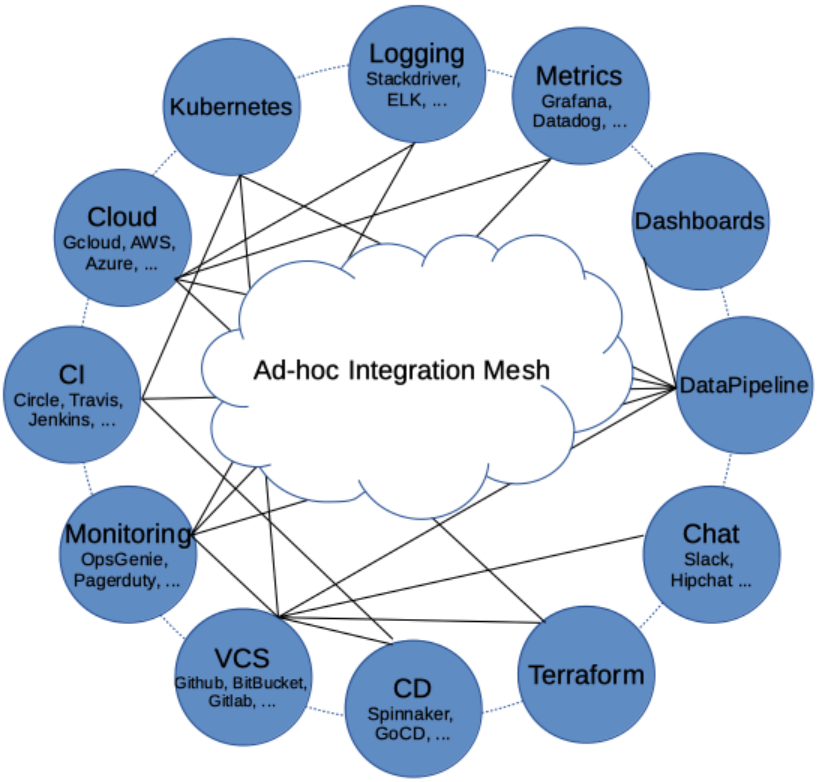
## 1.1 Overview

A IFTTT style event-driven, policy-based orchestration system that, is tailored towards SREs and DevOps workflows, and has a pluggable open architecture. The purpose is to fill the gap between the various components used in DevOps operations, to act as an orchestration hub, and to replace the ad-hoc integrations between the components so that the all the integrations can also be composed as code.

## 1.2 Design

### 1.2.1 Vision

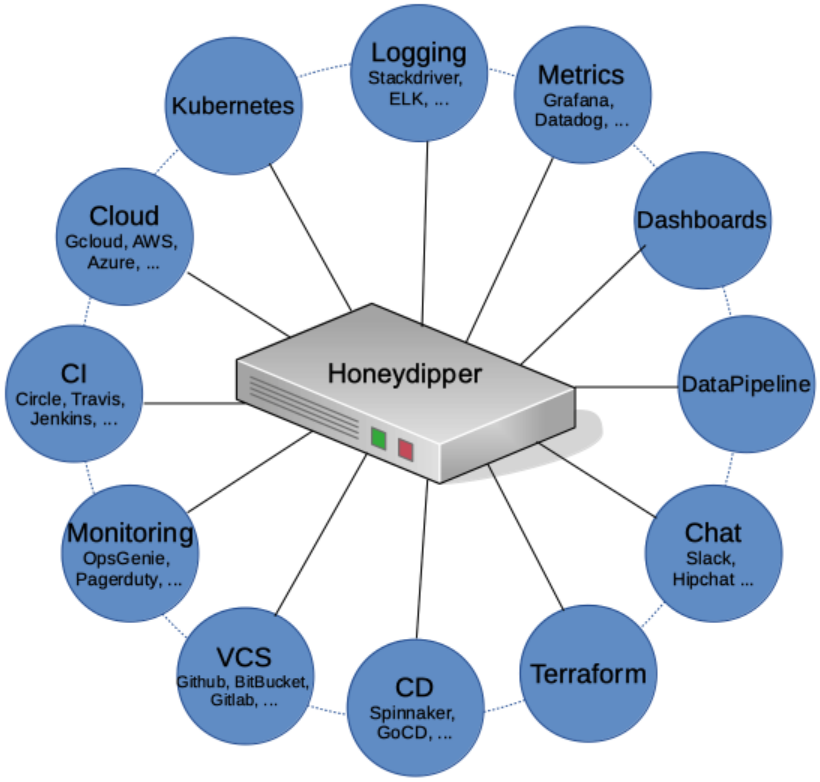
Engineers often use various systems and components to build and deliver services based on software products. The systems and components need to interact with each other to achieve automation. There are usually integration solutions that when implemented the components and systems can operate seamlessly. However, the integrations are usually created or configured in ad-hoc fashion. When the number of systems/components increases, the number of integrations and complexity of the integrations sometimes become unmanageable. All the systems/components are entangled in a mesh like network (See below graph). A lot of redundant configuration or development work become necessary. It is so hard, sometimes even impossible, to switch from one tool to another.




Systems

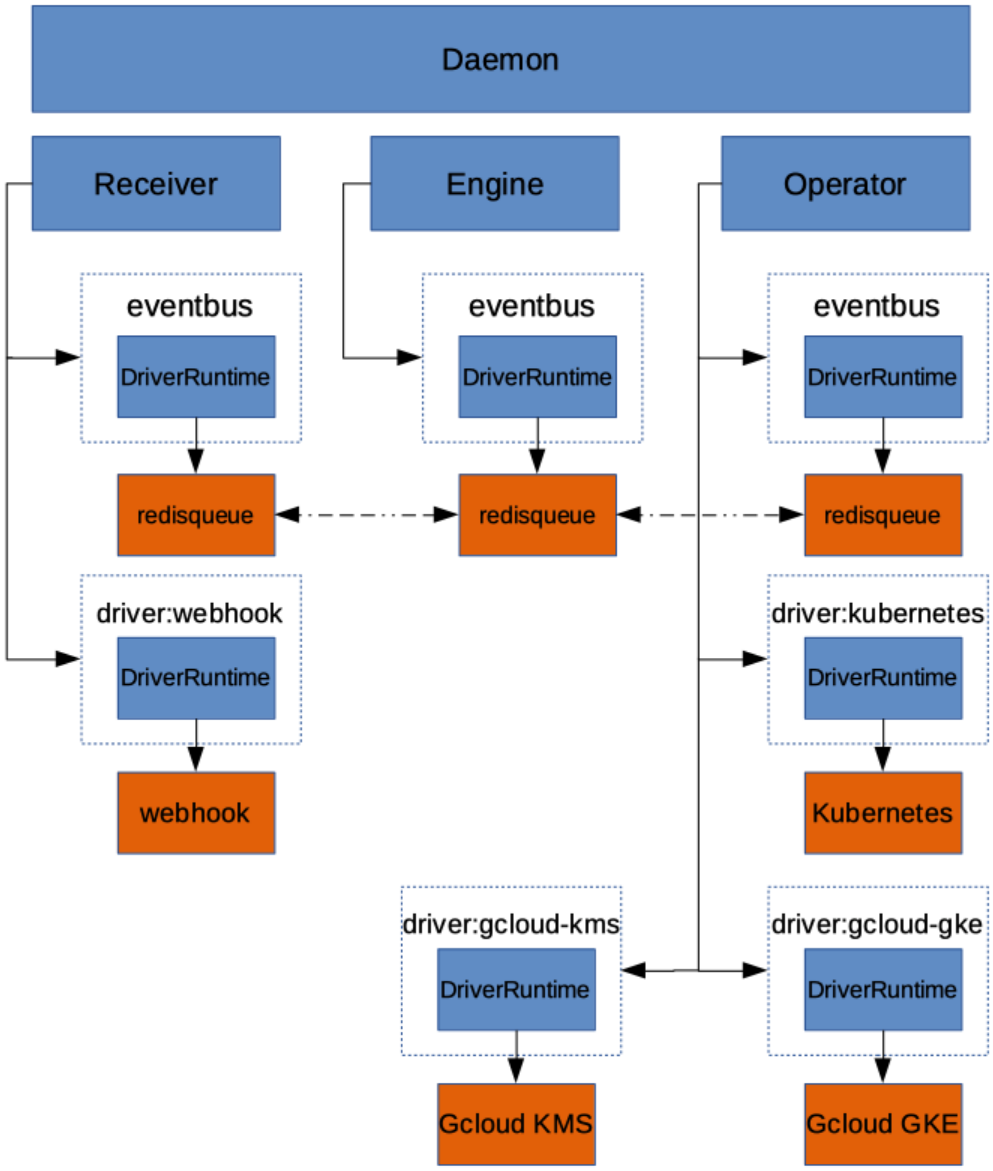
Ad-hoc Integration Mesh

Our vision is for Honeydipper to act as a central hub forming an ecosystem so that the various systems and components can be plugged in together and the integrations can be composed using rules, workflows and abstracted entities like systems, projects and organizations. With a centralized orchestration, redundant configurations and development work can be reduced or eliminated. With the abstraction layer, The underlying tools and solutions become interchangeable.




Systems orchestrated with Honeydipper

The core of Honeydipper is comprised of an event bus, and a rules/workflow engine. Raw events from various sources are received by corresponding event drivers, and then packaged in a standard format then published to the event bus. The rules/workflow engine picks up the event from the bus, and, based on the rules, triggers the actions or a workflow with multiple actions.




Dipper

Daemon

### 1.2.2 Core Concepts

In order for users to compose the rules, a few abstract concepts are introduced:

- Driver (Event)
- Raw Event
- System (Trigger): an abstract entity that groups dipper events and some configurations, metadata together
- Dipper Event (DipperMessage): a data structure that contains information that can be used for matching rules and being processed following the rules
- Rules: if some Dipper Event on some system happens, then start the workflow of actions on certain systems accordingly
- Features: A feature is a set of functions that can be mapped to a running driver, for example, the `eventbus` feature is fulfilled by `redisqueue` driver
- Services: A service is a group of capabilities the daemon provides to be able to orchestrate the plugged systems through drivers
- Workflow: Grouping of the actions so they can be processed, sequentially, parallel, etc
- Dipper Action (DipperMessage): a data structure that contains information that can be used for performing an action
- System (Responder): an abstract entity that groups dipper actions, configurations, metadata together
- Raw Action
- Driver (Action)

As you can see, the items described above follow the order or life cycle stage of the processing of the events into actions. Ideally, anything between the drivers should be composable, while some may tend to focusing on making various systems, Dipper event/actions available, others may want to focus on rules, workflows.

### 1.2.3 Features

#### Embracing GitOps

Honeydipper should have little to no local configuration needed to be bootstrapped. Once bootstrapped, the system should be able to pull configurations from one or more git repo. The benefit is the ease of maintenance of the system and access control automatically provided by git repo(s). The system needs to watch the git repos, one way or another, for changes and reload as needed. For continuous operation, the system should be able to survive when there is a configuration error, and should be able to continue running with an older version of the configuration.

#### Pluggable Architecture

Drivers make up an important part of the Honeydipper ecosystem. Most of the data mutations and actual work processes are handled by the drivers, including data decryption, internal communication, and interacting with external systems. Honeydipper should be able to extend itself through loading external drivers dynamically, and when configurations change, reload the running drivers hot or cold. There should be an interface for the drivers to delegate work to each other through RPCs.

### Abstraction

As mentioned in the concepts, one of Honeydipper's main selling points is abstraction. Events, actions can be defined traditionally using whatever characteristics provided by a driver, but also can be defined as an extension of another event/action with additional or override parameters. Events and actions can be grouped together into systems where data can be shared across. With this abstraction, we can separate the composing of complex workflows from defining low level event/action hook ups. Whenever a low level component changes, the high level workflow doesn't have to change, one only needs to link the abstract events with the new component native events.

## 1.3 More information

Please find the main document index at docs/README.md.

To install, See *Installation Guide*.

For driver developers, please read this guide. *Honeydipper driver developer's guide*

To get started on developing. See *How to setup local test environment*.

## 1.4 TODO

- Python driver library
- API service
- Dashboard webapp
- Auditing/logging driver
- State persistent driver
- Repo jailing
- RBAC

## 1.5 License

Honeydipper is licensed under the MPLv2 License.

## 1.6 Contributing

Thank you for your interest! Please refer to the Code of Conduct for guidance.

CHAPTER 2

# Tutorials

## 2.1 Installing Honeydipper



### 2.1.1 Prerequisites

- A running redis server

### 2.1.2 Step 1: Prepare your bootstrap repo

As described in the *architecture/design document*, Honeydipper loads configurations directly from one or many git repos. You can put the repo locally on the machine or pod where Honeydipper is running, or you can put the repos in GitHub, Bitbucket or Gitlab etc, or even mix them together. Make sure you configuration repo is private, and protected from unauthorized changes. Although, you can store all the sensitive information in encrypted form in the repo, you don't want this to become a target.

Inside your repo, you will need a `init.yaml` file. It is the main entrypoint that Honeydipper daemon seeks in each repo. See the Configuration Guide for detailed explanation. Below is an example of the minimum required data to get the daemon bootstrapped:

```
# init.yaml
---
repos:
  - repo: https://github.com/honeydipper/honeydipper-config-essentials.git

drivers:
  redisqueue:
    connection:
      Addr: <redis server IP>:<port>
      # uncomment below line if your redis server requires authentication
      # Password: xxxxxxxx
  redispubsub:
    connection:
      Addr: <redis server IP>:<port>
      # uncomment below line if your redis server requires authentication
      # Password: xxxxxxxx
```

### 2.1.3 Step 2: Bootstrap your daemon

#### Running in Kubernetes

This is the recommended way of using Honeydipper. Not only this is the easiest way to get Honeydipper started, it also enables Honeydipper to take advantage of the power of Kubernetes.

#### Using helm charts

To pass the information about the bootstrap config repo to Honeydipper daemon, the recommended way is to put all the information in a yaml file rather than use `--values` option during `helm install`. For example:

```
# values.yaml
---
daemon:
  env:
    - name: REPO
      value: git@github.com/example/honeydipper-config.git
    - name: DIPPER_SSH_KEY
      valueFrom:
        secretKeyRef:
          name: example-secret
          key: id_rsa
```

Note that, we need to provide a ssh key for Honeydipper daemon to be able to fetch the private repo using ssh protocol. Make sure that the key exists in your cluster as a `secret`.

Once the values file is prepared, you can run the `helm install` command like below.

```
helm install --values values.yaml orchestrator incubator/honeydipper
```

If you want to use an older version of the chart, (as of now, the latest one is 0.1.3), use `--version` to specify the chart version. By default, the chart uses the latest stable version of the Honeydipper daemon docker image, (latest

is `1.0.0` as of now). You can change the version by specifying `--set daemon.image.tag=x.x.x` in your `helm install` command.

---

Currently, the chart is available from incubator repo, and the honeydipper repo from helm hub as well. You may also choose to customize and build the chart by yourself following below steps.

```
git clone git@github.com:honeydipper/honeydipper-charts.git
cd honeydipper
helm package honeydipper
```

You should see the chart file `honeydipper-x.y.z.tgz` in your current directory.

---

### Create your own manifest file

You can use the below manifest file as a template to create your own. Note that, the basic information needed, besides the docker image for Honeydipper daemon, is the same, `REPO` and `DIPPER_SSH_KEY`.

```
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: honeydipper-daemon
  labels:
    app: honeydipper-daemon
spec:
  template:
    metadata:
      name: honeydipper-daemon
    spec:
      containers:
        - name: honeydipper-daemon
          image: honeydipper/honeydipper:1.0.0
          imagePullPolicy: Always
          env:
            - name: REPO
              value: git@github.com/example/honeydipper-config.git
            - name: DIPPER_SSH_KEY
              valueFrom:
                secretKeyRef:
                  namne: example-secret
                  key: id_rsa
```

For the webhook driver, you will need to create a service.

```
apiVersion: v1
kind: Service
metadata:
  name: honeydipper-webhook
spec:
  type: LoadBalancer
  ports:
  - name: webhook
    targetPort: 8080
```

(continues on next page)

(continued from previous page)

```
    port: 8080
selector:
    app: honeydipper-daemon
```

#### Running as docker container

```
docker run -it -e 'REPO=git@github.com/example/honeydipper-config.git' -e "DIPPER_SSH_
↪KEY=$(cat ~/.ssh/id_rsa)"  honeydipper/honeydipper:1.0.0
```

Replace the repo url with your own, and specify the private key path for accessing the private repo remotely. You may replace the value of `DIPPER_SSH_KEY` with a deploy key for your config repo.

#### Building from source

#### Prerequisites:

- Golang >= 1.11.xx
  - Honeydipper uses go modules
- Git
- Instructions assume POSIX compliant shell

#### Instructions

```
export GO111MODULE=on
git clone https://github.com/honeydipper/honeydipper.git
pushd honeydipper
go install -v ./...
popd
REPO=git@github.com/example/honeydipper-config.git DIPPER_SSH_KEY="$(cat ~/.ssh/id_
↪rsa)" honeydipper
```

*NOTE*: Specifying `GO111MODULE` is not necessary in golang >= 1.13.x

You don't have to specify `DIPPER_SSH_KEY` if the key is used by your ssh client by default.

Alternatively, you can follow the developer setup guide the download and build.

### 2.1.4 Step 3: Hacking away

That's it — your Honeydipper daemon is bootstrapped. You can start to configure it to suit your needs. The daemon pulls your config repos every minute, and will reload when changes are detected. See the *Honeydipper Guides* for more documents, including a way to setup GitHub push event-driven reload.

## 2.2 Honeydipper Configuration Guide

- *Topology and loading order*

### 2.2.1 Topology and loading order

As mentioned in the *Architecture/Design*, Honeydipper requires very little local configuration to bootstrap; it only requires a few environment variables to point it towards the git repo from which the bootstrap configurations are loaded. The bootstrap repo can load other repos using the `repos` section in any of the loaded yaml files. Inside every repo, Honeydipper will first load the `init.yaml`, and then load all the yaml files under `includes` section. Any of the files can also use a `includes` section to load even more files, and so on.

Inside every repo, when loading files, an including file will be loaded after all the files that it includes are loaded. So the including file can override anything in the included files. Similarly, repos are loaded after their dependency repos, so they can override anything in the depended repo.

One of the key selling point of Honeydipper is the ability to reuse and share. The drivers, systems, workflows and rules can all be packaged into repos then shared among projects, teams and organizations. Over time, we are expecting to see a number of reusable public config repos contributed and maintained by communities. The seed of the repos is the honeydipper-config-essentials repo, and the reference document can be found here.

### 2.2.2 Data Set

`DataSet` is the building block of Honeydipper config. Every configuration file contains a `DataSet`. Once all files are loaded, all the `DataSet` will be merged into a final `DataSet`. A `DataSet` is made up with one or more sections listed below.

```
// DataSet is a subset of configuration that can be assembled to the complete final␣
↪configuration.
type DataSet struct {
        Systems   map[string]System      `json:"systems,omitempty"`
        Rules     []Rule                 `json:"rules,omitempty"`
        Drivers   map[string]interface{} `json:"drivers,omitempty"`
        Includes  []string               `json:"includes,omitempty"`
        Repos     []RepoInfo             `json:"repos,omitempty"`
        Workflows map[string]Workflow    `json:"workflows,omitempty"`
        Contexts  map[string]interface{} `json:"contexts,omitempty"`
}
```

While it is possible to fit everything into a single file, it is recommended to organize your configurations into smaller chunks in a way that each chunk contains only relevant settings. For example, a file can define just a system and all its functions and triggers. Or, a file can define all the information about a driver. Another example would be to define a workflow in a file separately.

### 2.2.3 Repos

Repos are defined like below.

```
// RepoInfo points to a git repo where config data can be read from.
type RepoInfo struct {
        Repo   string
        Branch string `json:"branch,omitempty"`
        Path   string `json:"path,omitempty"`
}
```

To load a repo other than the bootstrap repo, just put info in the `repos` section like below.

```
---
repos:
  - repo: <git url to the repo>
    branch: <optional, defaults to master>
    path: <the location of the init.yaml, must starts with /, optional, defaults to />
  ...
```

### 2.2.4 Drivers

The `drivers` section provides driver specific config data, such as webhook listening port, Redis connections etc. It is a map from the names of the drivers to their data. The data element and structure of the driver data is only meaningful to the driver itself. Honeydipper just passes the data as-is, a `map[string]interface{}` in `go`.

#### Daemon configuration

Note that, `daemon` configuration is loaded and passed as a driver in this section.

```
---
drivers:
  daemon:
    loglevel: <one of INFO, DEBUG, WARNING, ERROR>
    featureMap:  # map of services to their defined features
      global:    # all services will recognize these features
        emitter: datadog-emitter
        eventbus: redisqueue
      operator:
        ...
      receiver:
        ...
      engine:
        ...
    features:   # the features to be loaded, mapped features won't be loaded unless␣
↪they are listed here
      global:
        - name: eventbus
          required: true  # will be loaded before other driver, and will rollback if␣
↪this fails during config changes
        - name: emitter
        - name: driver:gcloud-kms  # no feature name, just use the driver: prefix
          required: true
      operator:
```

(continues on next page)

(continued from previous page)

```
        - name: driver:gcloud-gke
        ...
```

### 2.2.5 Systems

As defined, systems are a group of triggers and actions and some data that can be re-used.

```
// System is an abstract construct to group data, trigger and function definitions.
type System struct {
        Data      map[string](interface{}) `json:"data,omitempty"`
        Triggers  map[string]Trigger       `json:"triggers,omitempty"`
        Functions map[string]Function      `json:"functions,omitempty"`
        Extends   []string                 `json:"extends,omitempty"`
}

// Trigger is the datastructure hold the information to match and process an event.
type Trigger struct {
        Driver     string      `json:"driver,omitempty"`
        RawEvent   string      `json:"rawevent,omitempty"`
        Conditions interface{} `json:"conditions,omitempty"`
        // A trigger should have only one of source event a raw event.
        Source Event `json:"source,omitempty"`
}

// Function is the datastructure hold the information to run actions.
type Function struct {
        Driver     string                  `json:"driver,omitempty"`
        RawAction  string                  `json:"rawaction,omitempty"`
        Parameters map[string](interface{}) `json:"parameters,omitempty"`
        // An action should have only one of target action or a raw action.
        Target Action `json:"target,omitempty"`
}
```

A system can extend another system to inherit data, triggers and functions, and then can override any of the inherited data with its own definition. We can create some abstract systems that contains part of the data that can be shared by multiple child systems. A `Function` can either be defined using `driver` and `rawAction` or inherit definition from another `Function` by specifying a `target`. Similarly, a `Trigger` can be defined using `driver` and `rawEvent` or inherit definition from another `Trigger` using `source`.

For example, inheriting the `kubernetes` system to create an instance of `kubernetes` cluster.

```
---
systems:
  my-k8s-cluster:
    extends:
      - kubernetes
    data:
      source:
        type: gcloud-gke
        project: myproject
        location: us-west1-a
        cluster: mycluster
        service_account: ENC[gcloud-kms,...masked...]
```

You can then use `my-k8s-cluster.recycleDeployment` function in workflows or rules to recycle deployments in the cluster. Or, you can pass `my-k8s-cluster` to `run_kubernetes` workflow as `system` context

variable to run jobs in that cluster.

Another example would be to extend the `slack_bot` system, to create another instance of slack integration.

```
---
systems:
  slack_bot: # first slack bot integration
    data:
      token: ...
      slash_token: ...
      interact_token: ...

  my_team_slack_bot: # second slack bot integration
    extends:
      - slack_bot
    data:
      token: ...
      slash_token: ...
      interact_token: ...

rules:
  - when:
      source:
        system: my_team_slack_bot
        trigger: slashcommand
    do:
      call_workflow: my_team_slashcommands
```

### 2.2.6 Workflows

See Workflow Composing Guide for details on workflows.

### 2.2.7 Rules

Here is the definition:

```
// Rule is a data structure defining what action to take when certain event happen.
type Rule struct {
        When Trigger
        Do   Workflow
}
```

Refer to the Systems section for the definition of `Trigger`, and see Workflow Composing Guide for workflows.

### 2.2.8 Config check

Honeydipper 0.1.8 and above comes with a configcheck functionality that can help checking configuration validity before any updates are committed or pushed to the git repos. It can also be used in the CI/CD pipelines to ensure the quality of the configuration files.

You can follow the installation guide to install the Honeydipper binary or docker image, then use below commands to check the local configuration files.

```
REPO=</path/to/local/files> honeydipper configcheck
```

If using a docker image

```
docker run -it -v </path/to/config>:/config -e REPO=/config honeydipper/honeydipper:x.
↪x.x configcheck
```

If your local config loads remote git repos and you want to validate them too, use `CHECK_REMOTE` environment variable.

```
REPO=</path/to/config> CHECK_REMOTE=1 honeydipper configcheck
```

If using docker image

```
docker run -it -v </path/to/config>:/config -e REPO=/config -e CHECK_REMOTE=1␣
↪honeydipper/honeydipper:x.x.x configcheck
```

You can also use `-h` option to see a full list of supported environment variables.

### 2.2.9 References

For a list of available drivers, systems, and workflows that you can take advantage of immediately, see the reference here.

- Honeydipper config essentials

## 2.3 Workflow Composing Guide

*DipperCL* is the control language that `Honeydipper` uses to configure data, assets and logic for its operation. It is basically a YAML with a `Honeydipper` specific schema.

### 2.3.1 Composing Workflows

`workflow` defines what to do and how to perform when an event is triggered. `workflow` can be defined in rules directly in the `do` section, or it can be defined independently with a name so it can be re-used/shared among multiple rules and workflows. A `workflow` can be as simple as invoking a single driver `rawAction`. It can also contains complicate logics, procedures dealing with various scenarios. All `workflows` are built with the same building blocks, follow the same process, and they can be stacked/combined with each other to achieve more complicated goals.

An example of a workflow defined in a rule calling an `rawAction`:

```
---
rules:
  - when:
      driver: webhook
      conditions:
        url: /test1
    do:
      call_driver: redispubsub.broadcast
      with:
        subject: internal
        channel: foo
        key: bar
```

An example of named workflow that can be invoked from other workflows.

```
---
workflows:
  foo:
    call_function: example.execute
    with:
      key1: val2
      key2: val2

rules:
  - when:
      source:
```

(continues on next page)

(continued from previous page)

```
      system: example
      trigger: happened
  do:
    call_workflow: foo
```

### Simple Actions

There are 4 types of simple actions that a workflow can perform.

- `call_workflow`: calling out to another named workflow, taking a string specifying the name of the workflow
- `call_function`: calling a predefined system function, taking a string in the form of `system.function`
- `call_driver`: calling a `rawAction` offered by a driver, taking a string in the form of `driver.rawAction`
- `wait`: wait for the specified amount of time or receive a wake-up request with a matching token. The time should be formatted according to the requirement for function ParseDuration. A unit suffix is required.

They can not be combined.

A function can also have no action at all. `{}` is a perfectly legit no-op workflow.

### Complex Actions

Complex actions are groups of multiple `workflows` organized together to do some complex work.

- `steps`: an array of child workflows that are executed in sequence
- `threads`: an array of child workflows that are executed in parallel
- `switch/cases/default`: taking a piece of contextual data specified in `switch`, chose and execute from a map of child workflows defined in `cases` or execute the child workflow defined in `default` if no branch matches

These can not be combined with each other or with any of the simple actions.

When using `steps` or `threads`, you can control the behaviour of the workflow upon `failure` or `error` status through fields `on_failure` or `on_error`. The allowed values are `continue` and `exit`. By default, `on_failure` is set to `continue` while `on_error` is set to `exit`. When using `threads`, `exit` means that when one thread returns error, the workflow returns without waiting for other threads to return.

### Iterations

Any of the actions can be combined with an `iterate` or `iterate_parallel` field to be executed multiple times with different values from a list. The current element of the list will be stored in a local contextual data item named `current`. Optionally, you can also customize the name of contextual data item using `iterate_as`. The elements of the lists to be iterated don't have to be simple strings, it can be a map or other complex data structures.

For example:

```
---
workflows:
  foo:
    iterate:
      - name: Peter
```

(continues on next page)

(continued from previous page)

```
        role: hero
      - name: Paul
        role: villain
    call_workflow: announce
    with:
      message: '{{ .ctx.current.name }} is playing the role of `{{ .ctx.current.role }
↪}`.'
```

### Conditions

We can also specify the conditions that the workflow checks before taking any action.

- `if_match/unless_match`: specify the skeleton data to match the contextual data
- `if/unless/if_any/unless/unless_all`: specify the list of strings that interpolate to truy/falsy values

Some examples for using skeleton data matching:

```
---
workflows:
  do_foo:
    if_match:
      foo: bar
    call_workflow: do_something

  do_bar:
    unless_match:
      team: :regex:engineering-.*
    call_workflow: complaint
    with:
      message: Only engineers are allowed here.

  do_something:
    if_match:
      user:
        - privileged_user1
        - privileged_user2
    call_workflow: assert
    with:
      message: you are either privileged_user1 or priviledged_user2

  do_some_other-stuff:
    if_match:
      user:
        age: 13
    call_workflow: assert
    with:
      message: .ctx.user matchs a data strucure with age field equal to 13
```

Please note how we use regular expression, list of options to match the contextual data, and how to match a field deep into the data structure.

Below are some examples of using list of conditions:

```
---
workflows:
```

(continues on next page)

(continued from previous page)

```
  run_if_all_meets:
    if:
      - $ctx.exits # ctx.exits must not be empty and not one of such strings `false`,␣
↪`nil`, `{}`, `[]`, `0`.
      - $ctx.also  # ctx.also must also be truy
    call_workflow: assert
    with:
      message: `exits` and `also` are both truy

  run_if_either_meets:
    if_any:
      - '{{ empty .ctx.exists | not }}'
      - '{{ empty .ctx.also | not }}'
    call_workflow: assert
    with:
      message: at least one of `exits` or `also` is not empty
```

### Looping

We can also repeat the actions in the workflow through looping fields

- `while`: specify a list of strings that interpolate into truy/falsy values
- `until`: specify a list of strings that interpolate into truy/falsy values

For example:

```
---
workflows:
  retry_func: # a simple forever retry
    on_error: continue
    on_failure: exit
    with:
      success: false
    until:
      - $ctx.success
    steps:
      - call_function: $ctx.func
      - export:
          success: '{{ eq .labels.status "success" }}'
    no_export:
      - success

  retry_func_count_with_exp_backoff:
    on_error: continue
    on_failure: exit
    with:
      success: false
      backoff: 0
      count-: 2
    until:
      - $ctx.success
      - $ctx.count
    steps:
      - if:
          - $ctx.backoff
```

(continues on next page)

(continued from previous page)

```
        wait: '{{ .ctx.backoff }}s'
      - call_function: $ctx.func
      - export:
          count: '{{ sub (int .ctx.count) 1 }}'
          success: '{{ eq .labels.status "success" }}'
          backoff: '{{ .ctx.backoff | default 10 | int | mul 2 }}'
    no_export:
      - success
      - count
      - backoff
```

### Hooks

Hooks are child workflows executed at a specified moments in the parent workflow's lifecycle. It is a great way to separate auxiliary work, such as sending heartbeat, sending slack messages, making an announcement, clean up, data preparation etc., from the actual work. Hooks are defined through context data, so it can be pulled in through predefined contexts, which makes the actual workflow seems less cluttered.

For example,

```
contexts:
  _events:
    '*':
      hooks:
        - on_first_action: workflow_announcement
  opsgenie:
    '*':
      hooks:
        - on_success:
            - snooze_alert

rules:
  - when:
      source:
        system: foo
        trigger: bar
    do:
      call_workflow: do_something

  - when:
      source:
        system: opsgenie
        trigger: alert
    do:
      context: opsgenie
      call_workflow: do_something
```

In the above example, although not specifically spelled out in the rules, both events will trigger the execution of `workflow_announcement` workflow before executing the first action. And if the workflow responding to the `opsgenie.alert` event is successful, `snooze_alert` workflow will be executed.

The supported hooks:

- on_session: when a workflow session is created, even `{}` no-op session will trigger this hook
- on_first_action: before a workflow performs first simple action

- on_action: before performs each simple action in `steps`
- on_item: before execute each iteration
- on_success: before workflow exit, and when the workflow is successful
- on_failure: before workflow exit, and when the workflow is failed
- on_error: before workflow exit, and when the workflow ran into error
- on_exit: before workflow exit

### 2.3.2 Contextual Data

Contextual data is the key to stitch different events, functions, drivers and workflows together.

#### Sources

Every workflow receives contextual data from a few sources:

- Exported from the event
- Inherit context from parent workflow
- Injected from predefined context, `_default`, `_event` and contexts listed through `context` or `contexts`
- Local context data defined in `with` field
- Exported from previous steps of the workflow

Since the data are received in that particular order listed above, the later source can override data from previous sources. Child workflow context data is independent from parent workflow, anything defined in `with` or inherited will only be in effect during the life cycle of current workflow, except the exported data. Once a field is exported, it will be available to all outer workflows. You can override this by specifying the list of fields that you don't want to export.

Pay attention to the example `retry_func_count_with_exp_backoff` in the previous section. In order to not contaminate parent context with temporary fields, we use `no_export` to block the exporting of certain fields.

#### Interpolation

We can use interpolation in workflows to make the workflow flexible and versatile. You can use interpolation in most of the fields of a workflow. Besides contextual data, other data available for interpolation includes:

- `labels` - string values attached to latest received dipper Message indicating session status, IDs, etc.,
- `ctx` - contextual data,
- `event` - raw unexposed event data from the original event that triggered the workflow
- `data` - raw unexposed payload from the latest received dipper message

It is recommended to avoid using `event` and `data` in workflows, and stick to `ctx` as much as possible. The raw unexposed data might eventually be deprecated and hidden. They may still be available in `system` definition.

*DipperCL* provides following ways of interpolation:

- **path interpolation** - comma separated multiple paths following a dollar sign, e.g. `$ctx.this,ctx.that,ctx.default`, cannot be mixed in strings. can specify a default value using either single, double or tilde quotes if none of the keys are defined in the context, e.g. `$ctx.this,ctx.that,"this value is the default"`. Also, can use `?` following the `$` to indicate that nil value is allowed.

- **inline go template** - strings with go templates that get rendered at time of the workflow execution, requires quoting if template is at the start of the string
- **yaml parser** - a string following a `:yaml:` prefix, will be parsed at the time of the execution, can be combined with go template
- **e-yaml encryption** - a string with `ENC[` prefix, storing base64 encoded encrypted content
- **file attachment** - a relative path following a `@:` prefix, requires quoting

See interpolation guide for detail on how to use interpolation.

### Merging Modifier

When data from different data source is merged, by default, map structure is deeply merged, while all other type of data with the same name is replaced by the newer source. One exception is that if the data in the new source is not the same type of the existing data, the old data stays in that case.

For example, undesired merge behaviour:

```
---
workflows
  merge:
    - export:
        data: # original
          foo: bar
          foo_map:
            key1: val1
          foo_list:
            - item1
            - item2
          foo_param: "a string"
    - export:
        data: # overriding
          foo: foo
          foo_map:
            key2: val2
          foo_list:
            - item3
            - item4
          foo_param: # type inconsistent
            key: val
```

After merging with the second step, the final exported data will be like below. Notice the fields that are replaced.

```
data: # final
  foo: foo
  foo_map:
    key1: val1
    key2: val2
  foo_list:
    - item3
    - item4
  foo_param: "a string"
```

We can change the behaviour by using merging modifiers at the end of the overriding data names.

Usage:

`var` is an example name of the overriding data, the following character indicates what type of merge modifier to use.

- `var-`: only use the new value if the `var` is not already defined and not nil
- `var+`: if the `var` is a list or string, the new value will be appended to the existing values
- `var*`: forcefully override the value

### 2.3.3 Essential Workflows

We have made a few helper workflows available in the `honeydipper-config-essentials` repo. Hopefully, they will make it easier for you to write your own workflows.

#### notify

Sending a chat message using configured system. The chat system can be anything that provides a `say` and a `reply` function.

*Required context fields*

- `chat_system`: system used for sending the message, by default `slack_bot`
- `message`: the text to be sent, do your own formatting
- `message_type`: used for formatting/coloring and select recipients
- `notify`: a list of recipients, slack channel names if using `slack_bot`
- `notify_on_error`: a list of additional recipients if `message_type` is `error` or `failure`

#### workflow_announcement

This workflow is intended to be invoked through `on_first_action` hook to send a chat message to announce what will happen.

*Required context fields*

- `chat_system`: system used for sending the message, by default `slack_bot`
- `notify`: a list of recipients, slack channel names if using `slack_bot`
- `_meta_event`: every events export a `_meta_event` showing the driver name and the trigger name, can be overridden in trigger definition
- `_event_id`: if you export a `_event_id` in your trigger definition, it will be used for display, by default it will be `unspecified`
- `_event_url`: the display of the `_event_id` will be a link to this url, by default `http://honeydipper.io`
- `_event_detail`: if specified, will be displayed after the brief announcement

Besides the fields above, this workflow also uses a few context fields that are set internally from host workflow(not the hook itself) definition.

- `_meta_desc`: the `description` from the workflow definition
- `_meta_name`: the `name` from the workflow definition
- `performing`: what the workflow is currently performing

#### workflow_status

This workflow is intended to be invoked through `on_exit`, `on_error`, `on_success` or `on_failure`. *Required context fields*

- `chat_system`: system used for sending the message, by default `slack_bot`
- `notify`: a list of recipients, slack channel names if using `slack_bot`
- `notify_on_error`: a list of additional recipients if `message_type` is `error` or `failure`
- `status_detail`: if available, the detail will be attached to the status notification message

Besides the fields above, this workflow also uses a few context fields and `labels` that are set internally from host workflow(not the hook itself).

- `_meta_desc`: the `description` from the workflow definition
- `_meta_name`: the `name` from the workflow definition
- `performing`: what the workflow is currently performing
- `.labels.status`: the latest function return status
- `.labels.reason`: the reason for latest failure or error

#### send_heartbeat

This workflow can be used in `on_success` hooks or as a stand-alone step. It sends a heartbeat to the alerting system

*Required context fields*

- `alert_system`: system used for sending the heartbeat, can be any system that implements a `heartbeat` function, by default `opsgenie`
- `heatbeat`: the name of the heartbeat

#### snooze_alert

This workflow can be used in `on_success` hooks or as a stand-alone step. It snooze the alert that triggered the workflow.

- `alert_system`: system used for sending the heartbeat, can be any system that implements a `snooze` function, by default `opsgenie`
- `alert_Id`: the ID of the alert

### 2.3.4 Running a Kubernetes Job

We can use a predefined `run_kubernetes` workflow from `honeydipper-config-essentials` repo to run kubernetes jobs. A simple example is below

```
---
workflows:
  kubejob:
    run_kubernetes:
      system: samplecluster
      steps:
        - type: python
```

(continues on next page)

(continued from previous page)

```
      command: |
        ...python script here...
    - type: bash
      shell: |
        ...shell script here...
```

### Basic of `run_kubernetes`

`run_kubernetes` workflow requires a `system` context field that points to a predefined system. The system must be extended from `kubernetes` system so that it has `createJob`, `waitForJob` and `getJobLog` function defined. The predefined system should also have the information required to connect to the kubernetes cluster, the namespace to use etc.

The required `steps` context field should tell the workflow what containers to define in the kubernetes job. If there are more that one step, the steps before the last step are all defined in `initContainters` section of the pod, and the last step is defined in `containers`.

Each step of the job has its type, which defines what docker image to use. The workflow comes with a few types predefined.

- python
- python2
- python3
- node
- bash
- gcloud
- tf
- helm
- git

A `step` can be defined using a `command` or a `shell`. A `command` is a string or a list of strings that are passed to the default entrypoint using `args` in the container spec. A `shell` is a string or a list of strings that passed to a customized shell script entrypoint.

For example

```
---
workflows:
  samplejob:
    run_kubernetes:
      system: samplecluster
      steps:
        - type: python3
          command: 'print("hello world")'
        - type: python3
          shell: |
            cd /opt/app
            pip install -r requirements.txt
            python main.py
```

The first step uses the `command` to directly passing a python command or script to the container, while the second step uses `shell` to run a script using the same container image.

There is a shared `emptyDir` volumes mounted at `/honeydipper` to every step, so that the steps can use the shared storage to pass on information. One thing to be noted is that the steps don't honour the default `WORKDIR` defined in the image, instead all the steps are using `/honeydipper` as `workingDir` in the container `spec`. This can be customized using `workingDir` in the step definition itself.

The workflow will return `success` in `.labels.status` when the job finishes successfully. If it fails to create a job or fails to get the status or job output, the status will be `error`. If the job is created, but failed to complete or return non-zero status code, the `.labels.status` will be set to `failure`. The workflow will export a `log` context field that contains a map from pod name to a map of container name to log output. A simple string version of the output that contains all the concatenated logs are exported as `output` context field.

### Environment Variables and Volumes

You can define environments and volumes to be used in each step or as a global context field to share them across steps. For example,

```
---
workflows:
  samplejob:
    run_kubernetes:
      system: samplecluster
      env:
        - name: CLOUDSDK_CONFIG
          value: /honeydipper/.config/gcloud
      steps:
        - git-clone
        - type: gcloud
          shell: |
            gcloud auth activate-service-account $GOOGLE_APPLICATION_ACCOUNT --key-
↪file=$GOOGLE_APPLICATION_CREDENTIALS
          env:
            - name: GOOGLE_APPLICATIION_ACCOUNT
              value: sample-service-account@foo.iam.gserviceaccount.com
            - name: GOOGLE_APPLICATION_CREDENTIALS
              value: /etc/gcloud/service-account.json
          volumes:
            - mountPath: /etc/gcloud
              volume:
                name: credentials-volume
                secret:
                  defaultMode: 420
                  secretName: secret-gcloud-service-account

        - type: tf
          shell: |
            terraform plan -no-color
```

Please note that, the `CLOUDSDK_CONFIG` environment is shared among all the steps. This ensures that all steps use the same gcloud configuration directory. The volume definition here is a combining of `volumes` and `volumeMounts` definition from pod `spec`.

### Predefined Step

To make writing kubernetes job workflows easier, we have created a few `predefined_steps` that you can use instead of writing your own from scratch. To use the `predefined_step`, just replace the step definition with the name of the step. See the example from the previous section, where the first step of the job is `git-clone`.

- `git-clone`

This step clones the given repo into the shared volume `/honeydipper/repo` folder. It requires that the `system` contains a few field to identify the repo to be cloned. That includes:

- `git_url` - the url of the repo
- `git_key_secret` - if a key is required, it should be present in the kubernetes cluster as a secret
- `git_ref` - branch

We can also use the predefined step as a skeleton to create our steps by overriding the settings. For example,

```
---
workflows:
  samplejob:
    run_kubernetes:
      system: samplecluster
      steps:
        - use: git-clone
          volumes: [] # no need for secret volumes when cloning a public repo
          env:
            - name: REPO
              value: https://github.com/honeydipper/honeydipper
            - name: BRANCH
              value: DipperCL
        - ...
```

Pay attention to `use` field of the step.

### Expanding `run_kubernetes`

If `run_kubernetes` only supports built-in types or predefined steps, it won't be too useful in a lot of places. Luckily, it is very easy to expand the workflow to support more things.

To add a new step type, just extend the `_default` context under `start_kube_job` in the `script_types` field.

For example, to add a type with the `rclone` image,

```
---
contexts:
  _default:
    start_kube_job:
      script_types:
        rclone:
          image: kovacsguido/rclone:latest
          command_prefix: []
          shell_entry: [ "/bin/ash", "-c" ]
```

Supported fields in a type:

- `image` - the image to use for this type
- `shell_entry` - the customized entrypoint if you want to run shell script with this image
- `shell_prefix` - a list of strings to be placed in `args` of the container `spec` before the actual `shell` script
- `command_entry` - in case you want to customize the entrypoint for using `command`
- `command_prefix` - a list of strings to be placed in `args` before `command`

Similarly, to add a new predefined step, extend the `predefined_steps` field in the same place.

For example, to add a rclone step

```
---
contexts:
  _default:
    start_kube_jobs:
      predefined_steps:
        rclone:
          name: backup-replicate
          type: rclone
          command:
            - copy
            - --include
            - '{{ coalesce .ctx.pattern (index (default (dict) .ctx.patterns)␣
↪(default "" .ctx.from)) "*" }}'
            - '{{ coalesce .ctx.source (index (default (dict) .ctx.sources) (default "
↪" .ctx.from)) }}'
            - '{{ coalesce .ctx.destination (index (default (dict) .ctx.destinations)␣
↪(default "" .ctx.to)) }}'
          volumes:
            - mountPath: /root/.config/rclone
              volume:
                name: rcloneconf
                secret:
                  defaultMode: 420
                  secretName: rclone-conf-with-ca
```

See *Defining steps* on how to define a step

### Using `run_kubernetes` in GKE

GKE is a google managed kubernetes cluster service. You can use `run_kubernetes` to run jobs in GKE as you would any kubernetes cluster. There are a few more helper workflows, predefined steps specifically for GKE.

- **`use_google_credentials` workflow**

If the context variable `google_credentials_secret` is defined, this workflow will add a step in the `steps` list to activate the service account. The service account must exist in the kubernetes cluster as a secret, the service account key can be specified using `google_credentials_secret_key` and defaults to `service-account.json`. This is a great way to run your job with a service account other than the default account defined through the GKE node pool. This step has to be executed before you call `run_kubernetes`, and the following `steps` in the job have to be added through *append modifier*.

For example:

```
---
workflows:
  create_cluster:
    steps:
      - call_workflow: use_google_credentials
      - call_workflow: run_kubernetes
        with:
          steps+: # using append modifier here
            - type: gcloud
              shell: gcloud container clusters create {{ .ctx.new_cluster_name }}
```

- **`use_gcloud_kubeconfig` workflow**

This workflow is used for adding a step to run `gcloud container clusters get-credentials` to fetch the kubeconfig data for GKE clusters. This step requires that the `cluster` context variable is defined and describing a GKE cluster with fields like `project`, `cluster`, `zone` or `region`.

For example:

```
---
workflows:
  delete_job:
    with:
      cluster:
        type: gke # specify the type of the kubernetes cluster
        project: foo
        cluster: bar
        zone: us-central1-a
    steps:
      - call_workflow: use_google_credentials
      - call_workflow: use_gcloud_kubeconfig
      - call_workflow: run_kubernetes:
        with:
          steps+:
            - type: gcloud
              shell: kubectl delete jobs {{ .ctx.job_name }}
```

- **`use_local_kubeconfig` workflow**

This workflow is used for adding a step to clear the kubeconfig file so `kubectl` can use default in-cluster setting to work on local cluster.

For example:

```
---
workflows:
  copy_deployment_to_local:
    steps:
      - call_workflow: use_google_credentials
      - call_workflow: use_gcloud_kubeconfig
        with:
          cluster:
            project: foo
            cluster: bar
            zone: us-central1-a
      - export:
          steps+:
            - type: gcloud
              shell: kubectl get -o yaml deployment {{ .ctx.deployment }} >␣
↪kuberentes.yaml
      - call_workflow: use_local_kubeconfig
      - call_workflow: run_kubernetes
        with:
          steps+:
            - type: gcloud
              shell: kubectl apply -f kubernetes.yaml
```

### 2.3.5 Slash Commands

The new version of `DipperCL` comes with integration with `Slack`, including **slash commands**, right out of the box. Once the integration is setup, we can easily add/customize the slash commands. See integration guide (coming

soon) for detailed instruction. There are a few predefined commands that you can try out without need of any further customization.

### Predefined Commands

- **help** - print the list of the supported command and a brief usage info
- **reload** - force honeydipper daemon to check and reload the configuration

### Adding New Commands

Let's say that you have a new workflow that you want to trigger through slash command. Define or extend a `_slashcommands` context to have something like below.

```
contexts:
  _slashcommands:
    slashcommand:
      slashcommands:
        <command>:
          workflow: <workflow>
          usage: just some brief intro to your workflow
          contexts: # optionally you can run your workflow with these contexts
            - my_context
```

Replace the content in <> with your own content.

### Mapping Parameters

Most workflows expect certain context variables to be available in order to function, for example, you may need to specify which DB to backup or restore using a `DB` context variable when invoking a backup/restore workflow. When a slash command is defined, a `parameters` context variable is made available as a string that can be accessed through `$ctx.parameters` using path interpolation or `{{ .ctx.parameters }}` in go templates. We can use the `_slashcommands` context to transform the `parameters` context variable into the actual variables the workflow requires.

For an simple example,

```
contexts:
  _slashcommands:
    slashcommand:
      slashcommands:
        my_greeting:
          workflow: greeting
          usage: respond with greet, take a single word as greeter

    greeting: # here is the context applied to the greeting workflow
      greeter: $ctx.parameters # the parameters is transformed into the variable␣
↪required
```

In case you want a list of words,

```
contexts:
  _slashcommands:
    slashcommand:
      slashcommands:
```

(continues on next page)

(continued from previous page)

```
        my_greeting:
          workflow: greeting
          usage: respond with greet, take a list of greeters

    greeting: # here is the context applied to the greeting workflow
      greeters: :yaml:{{ splitList " " .ctx.parameters }} # this generates a list
```

Some complex example, command with subcommands

```
contexts:
  _slashcommands:
    slashcommand:
      slashcommands:
        jobs:
          workflow: jobHandler
          usage: handling internal jobs

    jobHandler:
      command: '{{ splitList " " .ctx.parameters | first }}'
      name: '{{ splitList " " .ctx.parameters | rest | first }}'
      jobParams: ':yaml:{{ splitList " " .ctx.parameters | slice 2 | toJson }}'
```

### Messages and notifications

By default, a slashcommand will send acknowledgement and return status message to the channel where the command is launched. The messages will only be visible to the sender, in other words, is `ephemeral`. We can define a list of channels to receive the acknowledgement and return status in addition to the sender. This increases the visibility and auditability. This is simply done by adding a `slash_notify` context variable to the `slashcommand` workflow in the `_slashcommands` context.

For example,

```
contexts:
  _slashcommands:
    slashcommand:
      slash_notify:
        - "#my_team_channel"
        - "#security"
        - "#dont_tell_the_ceo"
      slashcommands:
        ...
```

### Secure the commands

When defining each command, we can use `allowed_channels` field to define a whitelist of channels from where the command can be launched. For example, it is recommended to override the `reload` command to be launched only from the whitelist channels like below.

```
contexts:
  _slashcommands:
    slashcommand:
      slashcommands:
        reload: # predefined
```

(continues on next page)

(continued from previous page)

```
        allowed_channels:
          - "#sre"
          - "#ceo"
```

## 2.4 Honeydipper Interpolation Guide

Tips: use Honeydipper config check feature to quickly identify errors and issues before committing your configuration changes, or setup your configuration repo with CI to run config check upon every push or PR.

- *Prefix interpolation*
  - `ENC[driver,ciphertext/base64==]` *Encrypted content*
  - :regex: *Regular expression pattern*
  - :yaml: *Building data structure with yaml*
  - $ *Referencing context data with given path*
- *Inline go template*
  - *Caveat: What does "inline" mean?*
  - *go template*
  - *Functions offered by Honeydipper*
    - *fromPath*
    - *now*
    - *duration*
    - *ISO8601*
    - *toYaml*
- *Workflow contextual data*
  - *Workflow Interpolation*
  - *Function Parameters Interpolation*
  - *Trigger Condition Interpolation*

Honeydipper functions and workflows are dynamic in nature. Parameters, system data, workflow data can be overridden at various phases, and we can use interpolation to tweak the function calls to pick up the parameters dynamically, or even to change the flow of execution at runtime.

### 2.4.1 Prefix interpolation

When a string value starts with certain prefixes, Honeydipper will transform the value based on the function specified by the prefix.

#### *`ENC[driver,ciphertext/base64==]`* Encrypted content

Encrypted contents are usually kept in system data. The value should be specified in `eyaml` style, start with `ENC[` prefix. Following the prefix is the name of the driver that can be used for decrypting the content. Following the driver name is a ",", and the base64 encoded ciphertext.

Can be used in system data, event conditions.

For example:

```
systems:
  kubenetes:
    data:
      service_account: ENC[gcloud-kms,...]
```

### *:regex:* Regular expression pattern

yaml doesn't have native support for regular expressions. When Honeydipper detects a string value starts with this prefix, it will interpret the following string as a regular expression pattern used for matching the conditions.

Can be used in system data, event conditions.

For example:

```
rules:
  - when:
      driver: webhook
      if_match:
        url: :regex:/test_.*$
  - do:
      ...
```

### *:yaml:* Building data structure with yaml

At first look, It may seem odd to have this prefix, since the config is yaml to begin with. In some cases, combining with the inline Go template, we can dynamically generate complex yaml structure that we can't write at config time.

Can be used in workflow definitions(data, content), workflow condition, function parameters.

For example:

```
workflows:
  create_list:
    export:
      items: |
        :yaml:---
        {{- range .ctx.results }}
        - name: {{ .name }}
          value: {{ .value }}
        {{- end }}
```

### *$* Referencing context data with given path

When Honeydipper executes a `workflow`, some data is kept in the context. We can use either the `$` prefix or the inline go template to fetch the context data. The benefit of using `$` prefix is that we can get the data as a structure such as map or list instead of a string representation.

Can be used in workflow definitions(data, content), workflow condition, function parameters.

For example:

```
workflows:
  next_if_success:
    if:
      - $ctx.result
    call_workflow: $ctx.work
```

The data available for `$` referencing includes

- `ctx` - context data
- `data` - the latest received dipper message payload
- `event` - the original dipper message payload from the event
- `labels` - the latest receive dipper message labels

The `$` reference can be used with multiple data entry separated by `,`. The first non empty result will be used. For example,

```
workflows:
  find_first:
    call_workflow: show_name
    with:
      name: $ctx.name,ctx.full_name,ctx.nick_name # choose the first non empty value␣
↪from the listed varialbes
```

We can also specify a default value with quotes, either single quotes, double quotes or back ticks, if all the listed variables are empty or nil. For example

```
workflows:
  do_something:
    call_workflow: something
    with:
      timeout: $ctx.timeout,ctx.default_timeout,"1800"
```

We can also allow nil or empty value using a `?` mark. For example

```
workflows:
  do_something:
    call_workflow: something
    with:
      timeout: $ctx.timeout,ctx.default_timeout,"1800"
      previous: $?ctx.previous
```

### 2.4.2 Inline go template

Besides the `$` prefix, we can also use inline go template to access the workflow context data. The inline go template can be used in workflow definitions(data, content), workflow condition, and function parameters.

#### Caveat: What does "inline" mean?

Unlike in typical templating languages, where templates were executed before yaml rendering, Honeydipper renders all configuration yaml at boot time or when reloading, and only executes the template when the particular content is needed. This allows Honeydipper to provide runtime data to the template when it is executed. However, that also means that templates can only be stored in strings. You can't wrap yaml tags in templates, unless you store the yaml as text like in the example for `:yaml:` prefix interpolation. Also, you can't use `{{` at the beginning of a string without quoting, because the yaml renderer may treat it as the start of a data structure.

### go template

Here are some available resources for go template:

- How to use go template? https://golang.org/pkg/text/template/
- sprig functions

### Functions offered by Honeydipper

### fromPath

Like the `:path:` prefix interpolation, the `fromPath` function takes a parameter as path and return the data the path points to. It is similar to the `index` built in function, but uses a more condensed path expression.

For example:

```
systems:
  opsgenie:
    functions:
      snooze:
        driver: web
        rawAction: request
        parameters:
          URL: https://api.opsgenie.com/v2/alerts/{{ fromPath . .params.alertIdPath }}
↪/snooze
          header:
            Content-Type: application/json
            Authorization: GenieKey {{ .sysData.API_KEY }}
...

rules:
  - when:
      source:
        system: some_system
        event: some_event
    do:
      target:
        system: opsgenie
        function: snooze
      parameters:
        alertIdPath: event.json.alert.Id
```

### now

This function returns current timestamps.

```
---
workflows:
  do_something:
    call_workflow: something
    with:
      time: '{{ now | toString }}'
```

#### duration

This function parse the duration string and can be used for date time calculation.

```
---
workflows:
  do_something:
    steps:
      - wait: '{{ duration "1m" }}'
      - call_workflow: something
```

#### ISO8601

This function format the timestamps into the ISO8601 format.

```
---
workflows:
  do_something:
    steps:
      - call_workflow: something
        with:
          time_str: '{{ now | ISO8601 }}'
```

#### toYaml

This function converts the given data structure into a yaml string

```
---
workflows:
  do_something:
    steps:
      - call_workflow: something
        with:
          yaml_str: '{{ .ctx.parameters | toYaml }}'
```

### 2.4.3 Workflow contextual data

Depending on where the interpolation is executed, 1) workflow engine, 2) operator (function parameters), the available contextual data is slightly different.

#### Workflow Interpolation

This happens when workflow `engine` is parsing and executing the workflows, but haven't sent the action definition to the `operator` yet.

- **data**: the payload of previous driver function return
- **labels**: the workflow data attached to the dipper.Message
  - **status**: the status of the previous workflow, "success", "failure" (driver failure), "blocked" (failed in daemon)
  - **reason**: a string describe why the previous workflow is not successful

    - **sessionID**
- **ctx**: the data passed to the workflow when it is invoked
- **event**: the event payload that triggered the original workflow

#### Function Parameters Interpolation

This happens at `operator` side, before the final `parameters` are passed to the `action driver`.

- **data**: the payload of previous driver function return
- **labels**: the workflow data attached to the dipper.Message
    - **status**: the status of the previous workflow, "success", "failure" (driver failure), "blocked" (failed in daemon)
    - **reason**: a string describe why the previous workflow is not successful
    - **sessionID**
- **ctx**: the data passed to the workflow when it is invoked
- **event**: the event payload that triggered the original workflow
- **sysData**: the data defined in the system the function belongs to
- **params**: the parameter that is passed to the function

#### Trigger Condition Interpolation

This happens at the start up of the `receiver` service. All the used events are processed into `collapsed` events. The `conditions` in the collapsed events are interpolated before being passed to `event driver`.

- **sysData**: the data defined in the system the event belongs to

## 2.5 Driver Developer's Guide

This document is intended for Honeydipper driver developers. Some programming experience is expected. Theoretically, we can use any programming language, even bash, to develop a driver for honeydipper. For now, there is a go library named honeydipper/dipper that makes it easier to do this in golang.

### 2.5.1 Basics

- Drivers are running in separate processes, so they are executables
- Drivers communicate with daemon through stdin/stdout, logs into stderr
- The name of the service that the driver is working for is passed in as an argument

### 2.5.2 By Example

Below is a simple driver that does nothing but restarting itself every 20 seconds.

```
package main

import (
  "flag"
  "github.com/honeydipper/honeydipper/pkg/dipper"
  "os"
  "time"
)

var driver *dipper.Driver

func main() {
  flag.Parse()

  driver = dipper.NewDriver(os.Args[1], "dummy")
  if driver.Service == "receiver" {
    driver.Start = waitAndSendDummyEvent
  }
  driver.Run()
}

func waitAndSendDummyEvent(msg *dipper.Message) {
  go func() {
    time.Sleep(20 * time.Second)
    driver.SendMessage(&dipper.Message{
      Channel: "eventbus",
      Subject: "message",
      Payload: map[string]interface{}{"data": []string{"line 1", "line 2"}},
    })
    driver.State = "cold"
    driver.Ping(msg)
  }()
}
```

The first thing that a driver does is to parse the command line arguments so the service name can be retrieved through *os.Args[1]*. Following that, the driver creates a helper object with *dipper.NewDriver*. The helper object provides hooks for driver to define the functions to be executed at various stage in the life cycle of the driver. A call to the *Run()* method will start the event loop to receive communication from the daemon.

There are 4 types of hooks offered by the driver helper objects.

- Lifecycle events
- Message handler
- RPC handler

- Command handler

Note that the *waitAndSendDummyEvent* method is assigned to *Start hook*. The *Start hook* needs to return immediately, so the method launches another event loop in a go routine and return the control to the helper object. The second event loop is where the driver actually receives events externally and use *driver.SendMessage* to relay to the service.

In this example, the dummy driver just manifest a fake event with json data as

```
{"data": ["line 1", "line 2"]}
```

The method also sets its status to "cold", meaning cold restart needed, and uses the *Ping* command to send its own state to the daemon, so it can be restarted.

### 2.5.3 Driver lifecycle and states

The driver will be in "loaded" state initially. When the *Run()* method is invoked, it will start fetching messages from the daemon. The first message is always "command:options" which carries the data and configuration required by the driver to perform its job. The helper object has a built-in handler for this and will dump the data into a structure which can later be queried using *driver.GetOption* or *driver.GetOptionStr* method.

Following the "command:options" is the "command:start" message. The helper object also has a built-in handler for the "command:start" message. It will first call the *Start hook* function, if defined, then change the driver state to "alive" then report the state back to daemon with *Ping* method. One important thing here is that if the daemon doesn't receive the "alive" state within 10 seconds, it will consider the driver failed to start and kill the process by closing the stdin/stdout channels. You can see why the Start hook has to return immediately.

When the daemon loads an updated version of the config, it will use "command:options" and "command:start" again to signal the driver to reload. Instead of calling *Start hook*, it will call *Reload hook* for reloading. If *Reload hook* is not defined, it will report to the daemon with "cold" state to demand a cold restart.

There is a handler for "command:stop" which calls the *Stop hook* for gracefully shutting down the driver. Although this is not needed most of time, assuming the driver is stateless, it does have some uses if the driver uses some resources that cannot be released gracefully by exiting.

### 2.5.4 Messages

Every message has an envelope, a list of labels and a payload. The envelope is a string ends with a newline, with fields separated by space(s). An valid envelope has following fields in the exact order:

- Channel
- Subject
- Number of labels
- Size

Following the envelop are a list of labels, each label is made up with a label definition line and a list of bytes as label value. The label definition includes

- Name of the label
- Size of the label in bytes

The payload is usually a byte array with certain encoding. As of now, the only encoding we use is "json". An example of sending a message to the daemon:

```
driver.SendMessage(&dipper.Message{
  Channel: "eventbus",
  Subject: "message",
  Labels: map[string]string{
    "label1": "value1",
  },
  Payload: map[string]interface{}{"data": []string{"line 1", "line 2"},
  IsRaw: false, # default
})
```

The payload data will be encoded automatically. You can also send raw message if use `IsRaw` as `true`, meaning that the driver will not attempt to encode the data for you, instead it will use the payload as bytes array directly. In case you need to encode the message yourself, there are two methods, *dipper.SerializePayload* accepts a *dipper.Message and put the encoded content back into the message, or *dipper.SerializeContent* which accepts bytes array and return the data structure as map.

When a message is received through the *Run()* event loop, it will be passed to various handlers as a *dipper.Message struct with raw bytes as payload. You can call *dipper.DeserializeContent* which accepts a byte array to decode the byte array, and you can also use *dipper.DeserializePayload* which accepts a *dipper.Message and place the decoded payload right back into the message.

Currently, we are categorizing the messages into 3 different channels:

- eventbus: messages that are used by *engine* service for workflow processing, subject could be `message`, `command` or `return`
- RPC: messages that invoke another driver to run some function, subject could be `call` or `return`
- state: the local messages between driver and daemon to manage the lifecycle of drivers

### 2.5.5 RPC

Drivers can make and offer RPC calls to each other. Daemon can also make RPC calls to the drivers. This greatly extends Honeydipper ability to conduct complicated operations. Each driver only need to handle the portion of the work it intends to solve, and outsourcing auxiliary work to other drivers which have the corresponding capabilities.

For example, `kubernetes` driver interacts with kubernetes clusters, but the task of obtaining the credentials and endpoints is outsourced to the vendor drivers, such as `gcloud-gke`, through a RPC call `getKubeCfg`. Another example is how Honeydipper handles encrypted content. Honeydipper supports `eyaml` style of encrypted content in configurations, and the cipher text is prefixed with a driver name. The decryption driver, `gcloud-kms` as an example, must offer a RPC call `decrypt`.

To make a RPC Call, use `Call` or `CallRaw` method, Both method block for return with 10 seconds timeout. The timeout is not tunable at this time. Each of them take three parameters:

- feature name - an abstract feature name, or a driver name with `driver:` prefix
- method name - the name of the RPC method
- parameters - payload of the dipper message constructed for the RPC call, a map or raw bytes

For example, calling the `gcloud-kms` driver for decryption

```
decrypted, err := driver.CallRaw("driver:gcloud-kms", "decrypt", encrypted)
```

There are also two non-blocking methods in the driver, `CallNoWait` or `CallRawNoWait`, to make RPC calls without waiting for any return. For example, making a call to emit a metric to a metrics collecting system, e.g. datadog.

```
err := driver.CallNoWait("emitter", "counter_increment", map[string]interface{}{
  name: "honeydipper.driver.invoked",
  tags: []string{
    "driver:mydriver",
  },
})
```

To offer a RPC method for the system to call, create the function that accept a single parameter `*dipper.Message`. Add the method to `RPCHandlers` map, for example

```
driver.RPCHandler["mymethod"] = MyFunc

func MyFunc(m *dipper.Message) {
  ...
}
```

Feel free to panic in your method, the wrapper will send an error response to the caller if that happens. To return data to the caller use the channel `Reply` on the incoming message. For example:

```
func MyFunc(m *dipper.Message) {
  dipper.DeserializePayload(m)
  if m.Payload != nil {
    panic(errors.New("not expecting any parameter"))
  }
  m.Reply <- dipper.Message{
    Payload: map[string]interface{}{"mydata": "myvalue"},
  }
}
```

### 2.5.6 Driver Options

As mentioned earlier, the driver receives the options / configurations from the daemon automatically through the helper object. As the data is stored in hashmap, the helper method *driver.GetOption* will accept a path and return an *Interface()* object. The path consists of the dot-delimited key names. If the returned data is also a map, you can use *dipper.GetMapData* or *dipper.GetMapDataStr* to retrieve information from them as well. If you are sure the data is a *string*, you can use *driver.GetOptionStr* to directly receive it as *string*.

The helper functions follow the golang convention of returning the value along with a bool to indicate if it is acceptable or not. See below for example.

```
  NewAddr, ok := driver.GetOptionStr("data.Addr")
  if !ok {
    NewAddr = ":8080"
  }

  hooksObj, ok := driver.GetOption("dynamicData.collapsedEvents")
  ...
  somedata, ok := dipper.GetMapDataStr(hooksObj, "key1.subkey")
  ...
```

There is always a *data* section in the driver options, which comes from the configuration file, e.g.:

```
---
...
drivers:
```

(continues on next page)

(continued from previous page)

```
  webhook:
    Addr: :880
...
```

### 2.5.7 Collapsed Events

Usually an event receiver driver just fires raw events to the daemon; it doesn't have to know what the daemon is expecting. There are some exceptions, for example, the webhook driver needs to know if the daemon is expecting some kind of webhook so it can decide what response to send to the web request sender, 200, 404 etc. A collapsed event is an event definition that has all the conditions, including the conditions from events that the current event is inheriting from. Dipper sends the collapsed events to the driver in the options with key name "dynamicData.collapsedEvents". Drivers can use the collapsed events to setup the filtering of the events before sending them to daemon. Not only does this allow the driver to generate meaningful feedback to the external requesters, but it also serves as the first line of defence against DDoS attacks on the daemon.

Below is an example of using the collapsed events data in webhook driver:

```
func loadOptions(m *dipper.Message) {
  hooksObj, ok := driver.GetOption("dynamicData.collapsedEvents")
  if !ok {
    log.Panicf("[%s] no hooks defined for webhook driver", driver.Service)
  }
  hooks, ok = hooksObj.(map[string]interface{})
  if !ok {
    log.Panicf("[%s] hook data should be a map of event to conditions", driver.
↪Service)
  }
  ...
}

func hookHandler(w http.ResponseWriter, r *http.Request) {
  eventData := extractEventData(w, r)

  matched := false
  for SystemEvent, hook := range hooks {
    for _, condition := range hook.([]interface{}) {
      if dipper.CompareAll(eventData, condition) {
        matched = true
        break
      }
    }
    if matched {
      break
    }
  }

  if matched {
    ...
  } else {
    ...
  }
}
```

The helper function *dipper.CompareAll* will try to match your event data to the conditions. Daemon uses the same function to determine if a `rawEvent` is triggering events defined in systems.

### 2.5.8 Provide Commands

A command is a raw function that provides response to an event. The workflow engine service sends "event-bus:command" messages to the operator service, and operator service will map the message to the corresponding driver and raw function, then forward the message to the corresponding driver with all the parameters as a "collapsed function". The driver helper provides ways to map raw actions to the function and handle the communications to back to the daemon.

A command handler is very much like the RPC handler mentioned earlier. All you need to do is add it to the `driver.CommandProvider.Commands` map. The command handler function should always return a value or panic. If it exists without a return, it can block invoking workflow until it times out. If you don't have any data to return, just send a blank message back like below.

```
func main() {
  ...
  driver.Commands["wait10min"] = wait10min
  ...
}

func wait10min(m *dipper.Message) {
  go func() {
    time.Sleep(10 * time.Minute)
    m.Reply <- dipper.Message{}
  }()
}
```

Note that the reply is sent in a go routine; it is useful if you want to make your code asynchronous.

### 2.5.9 Publishing and packaging

To make it easier for users to adopt your driver, and use it efficiently, you can create a public git repo and let users load some predefined configurations to jump start the integration. The configuration in the repo should usually include:

- `driver` definition and `fearture` loading under the `daemon` section;
- some wrapper `system` to define some `trigger`, `function` that can be used in rules;
- some `workflow` to help users use the `functions`, see Workflow composing guide for detail

For example, I created a hypothetical integration for a z-wave switch, the configuration might look like:

```
---
daemon:
  drivers:
    myzwave:
      name: myzwave
      data:
        Type: go
        Package: github.com/example/cmd/myzwave
  features:
    receiver:
      - "driver:myzwave"
    operator:
      - "driver:myzwave"

system:
  lightwitch:
```

(continues on next page)

(continued from previous page)

```
    data:
      token: "placeholder"
    triggers:
      driver: myzwave
      rawEvent: turned_on
      conditions:
        device_id: "placeholder"
        token: "{{ .sysData.token }}"
    functions:
      driver: myzwave
      rawAction: turn_on
      parameters:
        device_id: "placeholder"
        token: "{{ .sysData.token }}"

workflows:
  all_lights_on:
    - content: foreach_parallel
      data:
        items:
          - list
          - of
          - device_ids
          - to_be_override
        work:
          - type: function
            content:
              target:
                system: lightswitch
                function: turn_on
              parameters:
                device_id: '{{ `{{ .wfdata.current }}` }}'
```

Assuming the configuration is in github.com/example/myzwave-config/init.yaml, the users only need to load the below snippet into their bootstrap repo to load your driver and configurations, and start to customizing.

```
repos:
  ...
  - repo: https://github.com/example/myzwave-config
  ...
```

## 2.6 DipperCL Document Automatic Generation



Honeydipper configuration language (`DipperCL`) supports storing meta information of the configurations such as the purpose of the configuration, the fields or parameters definition and examples. The meta information can be used for

automatic document generating and publishing.

The meta information are usually recorded using `meta` field, or `description` field. They can be put under any of the below list of locations

1. `drivers.daemon.drivers.*` - meta information for a `driver`
2. `systems.*` - each system can have its meta information here
3. `systems.*.functions.*` - each system function can have its meta information here
4. `systems.*.triggers.*` - each system triggers can have its meta information here
5. `workflows.*` - each workflow can have its meta information here

The `description` field is usually a simple string that will be a paragraph in the document immediately following the name of the entry. The `meta` field is a map of different items based on what entry the meta is for.

Since the `description` field does not support formatting, and the it could be used for log generating at runtime, it is recommended to not use the `description` field, and instead use a `description` field under the `meta` field.

### 2.6.1 Documenting a Driver

Following fields are allowed under the `meta` field for a driver,

- `description` - A list of items to be rendered as paragraphs following the top level description
- `configurations` - A list of `name`, `description` pairs describing the items needed to be configured for this driver
- `notes` - A list of items to be rendered as paragraphs following the configurations
- `rawActions` - A list of meta information for `rawAction`, see below for detail
- `rawEvents` - A list of meta information for `rawEvents`, see below for detail
- `RPCs` - A list of meta information for `RPCs`, see below for detail

For each of the `rawActions`,

- `description` - A list of items to be rendered as paragraphs following the name of the action
- `parameters` - A list of `name`, `description` pairs describing the context variables needed for this action
- `returns` - A list of `name`, `description` pairs describing the context variables exported by this action
- `notes` - A list of items to be rendered as paragraphs following the above items

For each of the `rawEvents`,

- `description` - A list of items to be rendered as paragraphs following the name of the event
- `returns` - A list of `name`, `description` pairs describing the context variables exported by this event
- `notes` - A list of items to be rendered as paragraphs following the above items

For each of the `RPCs`,

- `description` - A list of items to be rendered as paragraphs following the name of the RPC
- `parameters` - A list of `name`, `description` pairs describing the parameters needed for this RPC
- `returns` - A list of `name`, `description` pares describing the values returned from this RPC
- `notes` - A list of items to be rendered as paragraphs following the above items

For example, to define the meta information for a driver:

```
---
drivers:
  daemon:
    drivers:
      my-driver:
        description: The driver is to enable Honeydipper to integrate with my service␣
↪with my APIs.
        meta:
          description:
            - ... brief description for the system ...

          configurations:
            - name: foo
              description: A brief description for foo
            - name: bar
              description: A brief description for bar
              ...

          rawEvents:
            myEvent:
              description:
                - |
                  paragraph .....
              returns:
                - name: key1
                  description: description for key1
                - name: key2
                  description: description for key2

          notes:
            - some notes as text
            - example:
                ...
        ...
```

### 2.6.2 Document a System

Following fields are allowed under the `meta` field for a `system`,

- `description` - A list of items to be rendered as paragraphs following the top level description
- `configurations` - A list of `name`, `description` pairs describing the items needed to be configured for this in system data
- `notes` - A list of items to be rendered as paragraphs following the configurations

For each of the `functions`,

- `description` - A list of items to be rendered as paragraphs following the name of the function
- `inputs` - A list of `name`, `description` pairs describing the context variables needed for this function
- `exports` - A list of `name`, `description` pares describing the context variables exported by this function
- `notes` - A list of items to be rendered as paragraphs following the above items

For each of the `triggers`,

- `description` - A list of items to be rendered as paragraphs following the name of the trigger

- `exports` - A list of `name`, `description` pares describing the context variables exported by this trigger
- `notes` - A list of items to be rendered as paragraphs following the above items

For example, to define the mata information for a system,

```
---
systems:
  mysystem:
    meta:
      description:
        - ... brief description for the system ...

      configurations:
        - name: key1
          description: ... brief description for key1 ...
        - name: key2
          description: ... brief description for key2 ...

      notes:
        - ... some notes ...
        - example: |
          ... sample in yaml ...

    data:
      ...

    functions:
      myfunc:
        meta:
          description:
            - ... brief description for the function ...
          inputs:
            - name: key1
              description: ... brief description for key1 ...
            - name: key2
              description: ... brief description for key2 ...
          exports:
            - name: key3
              description: ... brief description for key3 ...
            - name: key4
              description: ... brief description for key4 ...
          notes:
            - ... some notes ...
            - example: |
                ... sample in yaml ...
```

### 2.6.3 Document a Workflow

Following fields are allowed under the `meta` field for a `workflow`,

- `description` - A list of items to be rendered as paragraphs following the top level description
- `inputs` - A list of `name`, `description` pairs describing the context variables needed for this workflow
- `exports` - A list of `name`, `description` pares describing the context variables exported by this workflow
- `notes` - A list of items to be rendered as paragraphs following the above items

For example, to define the mata information for a workflow,

```
---
workflows:
  myworkflow:
    meta:
      description:
        - ... brief description for the workflow ...
      inputs:
        - name: key1
          description: ... brief description for key1 ...
        - name: key2
          description: ... brief description for key2 ...
      exports:
        - name: key3
          description: ... brief description for key3 ...
        - name: key4
          description: ... brief description for key4 ...
      notes:
        - ... some notes ...
        - example: |
          ... sample in yaml ...
```

### 2.6.4 Formatting

Both `description` and `notes` fields under `meta` support formatting. They accept a list of items that each will be rendered as a paragraph in the documents. The only difference between them is the location where they will appear in the documents.

Honeydipper `docgen` uses `sphinx` to render the documents, so the source document is in `rst` format. You can use `rst` format in each of the paragraphs. You can also let `docgen` to format your paragraph by specify a data structure as the item instead of plain text.

For example, plain text paragraph,

```
description:
  - This is a plain text paragraph.
```

Highlighting the paragraph, see sphinx document for detail on highlight type.

```
notes:
  - highlight: This paragraph will be highlighted.
  - highlight: This paragraph will be highlighted with type `error`.
    type: error
```

Specify a code block,

```
notes:
  - See below for an example
  - example: | # by default, yaml
      ---
      rules:
        - when: ...
          do: ...
  - example: |
      func dosomething() {
      }
    type: go
```

### 2.6.5 Building

In order to build the document for local viewing, follow below steps.

1. install sphinx following the sphinx installation document
2. install markdown extension for sphinx following the recommonmark installation document
3. install the `read the docs` theme for sphinx following the readthedoc theme installation document
4. clone the `honeydipper-sphinx` repo

```
git clone https://github.com/honeydipper/honeydipper-sphinx.git
```

5. generating the source document for sphinx

```
cd honeydipper-sphinx
docker run -it -v $PWD/docgen:/docgen -v $PWD/source:/source -e DOCSRC=/docgen -e␣
↪DOCDST=/source honeydipper/honeydipper:1.0.0 docgen
```

6. build the documents

```
# cd honeydipper-sphinx
make html
```

7. view your documents

```
# cd honeydipper-sphinx
open build/html/index.html
```

### 2.6.6 Publishing

In order to including your document in the Honeydipper community repo section of the documents, follow below steps.

1. clone the `honeydipper-sphinx` repo

```
git clone https://github.com/honeydipper/honeydipper-sphinx.git
```

2. modify the `docgen/docgen.yaml` to add your repo under the `repos` section
3. submit a PR

# CHAPTER 3

# How-To

## 3.1 Enable Encrypted Config in Honeydipper

Honeydipper outsources encryption/decryption tasks to drivers. In order for Honeydipper to be able to decrypt the encrypted content in the config files, the proper driver needs to be loaded and configured. By default, the `honeydipper-config-essentials` repo `gcloud` bundle comes with a gcloud KMS driver, I will use this as an example to explain how decryption works.

- *Loading the driver*
- *Config the driver*
- *How to encrypt your secret*

### 3.1.1 Loading the driver

When you include the `honeydipper-config-essentials` repo from your bootstrap repo like below:

```
---
repos:
  - repo: https://github.com/honeydipper/honeydipper-config-essentials.git
    path: /gcloud
```

The `gcloud-kms` driver is loaded *automatically* with following daemon configuration.

```
---
drivers:
  ...
  daemon:
    ...
    features:
      global:
        - name: driver:gcloud-kms
          required: true
```

(continues on next page)

(continued from previous page)

```
    ...
  drivers:
    ...
    gcloud-kms:
      name: gcloud-kms
      type: builtin
      handlerData:
        shortName: gcloud-kms
```

Note that the above configuration snippet is for your information purpose, you don't have to manually type them in if you include the `gcloud` bundle from the `honeydipper-config-essential` repo.

### 3.1.2 Config the driver

The `gcloud-kms` driver assumes that there is a default google credential where the daemon is running. This is usually the case when you run Honeydipper in gcloud either in Compute Engine or in Kubernetes clusters. See GCP documentation on how to configure the Compute Engine instance or Kubernetes clusters with a service account. If you are running this from your workstation, make sure you run `gcloud auth login` to authenticate with gcloud. The service account or the credential you are using needs to have `roles/kms.CryptoKeyDecryptor` IAM role. If you are running the Honeydipper in a docker container other than gcloud, you will need to mount a service account key file into the container and set `GOOGLE_APPLICATION_CREDENTIALS` environment variable.

The `gcloud-kms` driver expects a configuration item under `drivers.gcloud-kms` named `keyname`.

For example:

```
---
drivers:
  ...
  gcloud-kms:
    keyname: projects/<your project>/locations/<region>/keyRings/<keyring name>/
↪cryptoKeys/<key name>
  ...
```

Once this is configured in your repo and loaded by the daemon, you can start to use this driver to decrypt content in the configuration files.

### 3.1.3 How to encrypt your secret

Assuming you have `gcloud` command installed, and authenticated, and you have the `roles/kms.CryptoKeyEncryptor` role.

```
echo -n xxxx_your_secret_xxxx |
  gcloud --project=<...> kms encrypt --plaintext-file=- --ciphertext-file=- --keyring=
↪<...> --key=<...> --location=<...> |
  base64
```

Fill in the blank for `project`, `keyring`, `location` and `key` with the same information you configured for the driver. The command will output the base64 encoded cipher text. You can use this in your configuration file with `eyaml` style syntax. For example:

```
---
systems:
  my_system:
```

(continues on next page)

(continued from previous page)

```
    data:
      mysecret: ENC[gcloud-kms,---base64 encoded ciphertext---]
```

See the interpolation guide for more information on eyaml syntax.

## 3.2 Logging Verbosity

Honeydipper uses stdout and stderr for logging. The stdout is used for all levels of logs, while stderr is used for reporting warning or more critical messages. The daemon and each driver can be configured individually on logging verbosity. Just put the verbosity level in `drivers.<driver name>.loglevel`. Use `daemon` as driver name for daemon logging.

For example:

```
---
drivers:
  daemon:
    loglevel: INFO
  web:
    loglevel: DEBUG
  webhook:
    loglevel: WARNING
```

The supported levels are, from most critical to least:

- `CRITICAL`
- `ERROR`
- `WARNING`
- `NOTICE`
- `INFO`
- `DEBUG`

## 3.3 Reload on Github Push

After following this guide, the Honeydipper daemon should not pull from config repo as often as before, and it can reload when there is any change to the remote repo.

- *Github Integration in Honeydipper*
- *Config webhook in Github repo*
- *Configure a reloading rule*
- *Reduce the polling interval*

### 3.3.1 Github Integration in Honeydipper

Create a yaml file in your config repo to store the settings for github integration, and make sure it is loaded through `includes` in `init.yaml`. See the github integration reference for detail on how to config.

For example:

```
# integrations.yaml
---
systems:
  ...
  github:
    token: ENC[gcloud-kms,xxxxxx..]
    oauth_token: ENC[gcloud-kms,xxxxxx...]
```

By configuring the github integration, we enabled a webhook at certain url (by default, `/github/push`, see your infrastructure configuration for the url host and port). As of now, the Honeydipper webhook driver doesn't support authentication using signature header, so we use a token to authenticate requests coming from github.

### 3.3.2 Config webhook in Github repo

Go to your config repo in github, click `settings` => `webhooks`, then add a `webhook` with the webhook url. For example,

```
https://mywebhook.example.com:8443/github/push?token=xxxxxxxx
```

Make sure you select "Pushes" to be sent to the configured webhook.

### 3.3.3 Configure a reloading rule

Create a yaml file in your config repo to store a rule, and make sure it is loaded through `includes` in one of previously loaded yaml file. The rule should look like below

```
# reload_on_gitpush.yaml
---
rules:
  - when:
      source:
        system: github
        trigger: push
    do:
      if_match:
        - git_repo: myorg/myconfig
          git_ref: refs/heads/master
      call_workflow: reload
```

Your repository name and branch name may differ.

After the rule is loaded into the Honeydipper daemon, you should be able to see from the logs, or the slack channel where the daemon is configured to set the status, that the daemon reloads configuration when there is new push to your repo. The `if_match` in the `do` section takes a list, so if you want to watch for more than one repo, just add them into the list.

### 3.3.4 Reduce the polling interval

In the configuration for your daemon, set the `configCheckInterval` to a longer duration. The duration is parsed using ParseDuration API, use 'm' suffix for minutes, 'h' for hours. See below for example:

```
# daemon.yaml
---
drivers:
  daemon:
    configCheckInterval: "60m"
```

## 3.4 Setup a test/dev environment locally

### 3.4.1 Setup Go environment

- Setup a directory as your go work directory and add it to GOPATH. Assuming go 1.13.1 or up is installed, gvm is recommended to manage multiple versions of go. You may want to persist the GOPATH in your bash_profile

```
mkdir ~/go
export GOPATH=$GOPATH:$PWD/go
export PATH=$PATH:$GOPATH/bin
```

### 3.4.2 Clone the code

```
go get github.com/honeydipper/honeydipper
```

or

```
git clone https://github.com/honeydipper/honeydipper.git
```

### 3.4.3 Build and test

- Build

```
go install -v ./...
```

- Run test

```
go test -v ./...
```

- (Optional) For colored test results

```
go get -u github.com/rakyll/gotest
gotest -v ./...
```

- For pre-commit hooks

```
curl -sfL https://install.goreleaser.com/github.com/golangci/golangci-lint.sh | sh -s␣
↪-- -b $(go env GOPATH)/bin v1.15.0
brew install pre-commit
pre-commit install --install-hooks
```

### 3.4.4 Create local config REPO

Run below command to create your local config repo.

```
git init mytest
cd mytest
cat <<EOF > init.yaml
repos:
  - repo: https://github.com/honeydipper/honeydipper-config-essentials.git

drivers:
  redisqueue:
    connection:
      Addr: 127.0.0.1:6379
  redispubsub:
    connection:
      Addr: 127.0.0.1:6379

rules:
  - when:
      driver: webhook
      if_match:
        url: /health
    do: {}
EOF
git add init.yaml
git commit -m 'init' -a
```

### 3.4.5 Start Honeydipper daemon

Before you start your Honeydipper daemon, you need:

1. Have a redis server running locally
2. If you want to use encrypted configuration, make sure your are authenticated with google and having "Cloud KMS Crypto Encryptor/Decryptor" role. See encryption guide for detail

```
REPO=/path/to/mytest LOCALREDIS=1 honeydipper
```

When you use `LOCALREDIS=1` environment vairable, Honeydipper daemon will ignore the connection settings from your repo and use localhost instead.

You can also set envrionment `DEBUG="*"` to enable verbose debug logging for all parts of daemon and drivers.

Once the daemon is running, you can access the healthcheck url like below

```
curl -D- http://127.0.0.1:8080/health
```

You should see a `200` response code. There is no payload in the response.

See configuration guide for detail on how to configure your system.

CHAPTER 4

# Essentials

The essential configurations to bootstrap Honeydipper

## 4.1 Installation

Include the following section in your **init.yaml** under **repos** section

```
- repo: https://github.com/honeydipper/honeydipper-config-essentials
```

## 4.2 Drivers

This repo provides following drivers

### 4.2.1 kubernetes

This driver enables Honeydipper to interact with kubernetes clusters including finding and recycling deployments, running jobs and getting job logs, etc. There a few wrapper workflows around the driver and system functions, see the workflow composing guide for detail. This section provides information on how to configure the driver and what the driver offers as *rawActions*, the information may be helpful for understanding how the kubernetes workflow works.

#### Action: createJob

Start a run-to-complete job in the specified cluster. Although you can, it is not recommended to use this rawAction directly. Use the wrapper workflows instead.

**Parameters**

**type** The type of the kubernetes cluster, basically a driver that provides a RPC call for fetching the kubeconfig from. currently only *gcloud-gke* and *local* is supported, more types to be added in the future.

**source** A list of k/v pair as parameters used for making the RPC call to fetch the kubeconfig. For *local*, no value is required, the driver will try to use in-cluster configurations. For *gcloud-gke* clusters, the k/v pair should have keys including *service_account*, *project*, *zone* and *cluster*.

**namespace** The namespace for the job

**job** the job object following the kubernetes API schema

**Returns**

**metadata** The metadata for the created kubernetes job

**status** The status for the created kuberntes job

See below for a simple example

```
---
workflows:
  call_driver: kubernetes.createJob
  with:
    type: local
    namespace: test
    job:
      apiVersion: batch/v1
      kind: Job
      metadata:
        name: pi
      spec:
        template:
          spec:
            containers:
            - name: pi
              image: perl
              command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
            restartPolicy: Never
        backoffLimit: 4
```

### Action: recycleDeployment

recycle a deployment by deleting the replicaset and let it re-spawn.

**Parameters**

**type** The type of the kubernetes cluster, see **createJob** rawAction for detail

**source** A list of k/v pair as parameters used for getting kubeconfig, see **createJob** rawAction for detail

**namespace** The namespace for the deployment to be recycled, *default* if not specified

**deployment** a label selector for identifying the deployment, e.g. *run=my-app*, *app=nginx*

See below for a simple example

```
---
rules:
  - when:
      source:
```

(continues on next page)

(continued from previous page)

```
        system: alerting
        trigger: fired
    do:
      call_driver: kubernetes.recycleDeployment
      with:
        type: gcloud-gke
        source:
          service_account: ENC[gcloud-kms, ...masked... ]
          zone: us-central1-a
          project: foo
          cluster: bar
        deployment: run=my-app
```

### Action: getJobLog

Given a kubernetes job metadata name, fetch and return all the logs for this job. Again, it is not recommended to use *createJob*, *waitForJob* or *getJobLog* directly. Use the helper workflows instead.

**Parameters**

**type** The type of the kubernetes cluster, see **createJob** rawAction for detail

**source** A list of k/v pair as parameters used for getting kubeconfig, see **createJob** rawAction for detail

**namespace** The namespace for the job

**job** The metadata name of the kubernetes job

**Returns**

**log** mapping from pod name to a map from container name to the logs

**output** with all logs concatinated

See below for a simple example

```
---
workflows:
  run_job:
    steps:
      - call_driver: kubernetes.createJob
        with:
          type: local
          job:
            apiVersion: batch/v1
            kind: Job
            metadata:
              name: pi
            spec:
              template:
                spec:
                  containers:
                  - name: pi
                    image: perl
                    command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
                  restartPolicy: Never
              backoffLimit: 4
      - call_driver: kubernetes.waitForJob
```

(continues on next page)

(continued from previous page)

```
        with:
          type: local
          job: $data.metadta.name
      - call_driver: kubernetes.getJobLog
        with:
          type: local
          job: $data.metadta.name
```

### Action: waitForJob

Given a kubernetes job metadata name, use watch API to watch the job until it reaches a terminal state. This action usually follows a *createJob* call and uses the previous call's output as input. Again, it is not recommended to use *createJob*, *waitForJob* or *getJobLog* directly. Use the helper workflows instead.

**Parameters**

**type** The type of the kubernetes cluster, see **createJob** rawAction for detail

**source** A list of k/v pair as parameters used for getting kubeconfig, see **createJob** rawAction for detail

**namespace** The namespace for the job

**job** The metadata name of the kubernetes job

**timeout** The timeout in seconds

**Returns**

**status** The status for the created kuberntes job

See below for a simple example

```
---
workflows:
  run_job:
    steps:
      - call_driver: kubernetes.createJob
        with:
          type: local
          job:
            apiVersion: batch/v1
            kind: Job
            metadata:
              name: pi
            spec:
              template:
                spec:
                  containers:
                  - name: pi
                    image: perl
                    command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
                  restartPolicy: Never
              backoffLimit: 4
      - call_driver: kubernetes.waitForJob
        with:
          type: local
          job: $data.metadta.name
```

### 4.2.2 redispubsub

redispubsub driver is used internally to facilitate communications between different components of Honeydipper system.

**Configurations**

**connection** The parameters used for connecting to the redis including *Addr*, *Password* and *DB*.

See below for an example

```
---
drivers:
  redispubsub:
    connection:
      Addr: 192.168.2.10:6379
      DB: 2
      Password: ENC[gcloud-kms,...masked]
```

#### Action: send

broadcasting a dipper message to all Honeydipper services. This is used in triggering configuration reloading and waking up a suspended workflow. The payload of rawAction call will used as broadcasting dipper message paylod.

**Parameters**

**broadcastSubject** the subject field of the dipper message to be sent

Below is an example of using the driver to trigger a configuration reload

```
---
workflows:
  reload:
    call_driver: redispubsub.send
    with:
      broadcastSubject: reload
      force: $?ctx.force
```

Below is another example of using the driver to wake up a suspended workflow

```
---
workflows:
  resume_workflow:
    call_driver: redispubsub.send
    with:
      broadcastSubject: resume_session
      key: $ctx.resume_token
      labels:
        status: $ctx.labels_status
        reason: $?ctx.labels_reason
      payload: $?ctx.resume_payload
```

### 4.2.3 redisqueue

redisqueue driver is used internally to facilitate communications between different components of Honeydipper system. It doesn't offer *rawActions* or *rawEvents* for workflow composing.

**Configurations**

**connection** The parameters used for connecting to the redis including *Addr*, *Password* and *DB*.

See below for an example

```
---
drivers:
  redisqueue:
    connection:
      Addr: 192.168.2.10:6379
      DB: 2
      Password: ENC[gcloud-kms,...masked]
```

### 4.2.4 web

This driver enables Honeydipper to make outbound web requests

#### Action: request

making an outbound web request

**Parameters**

**URL** The target url for the outbound web request

**header** A list of k/v pair as headers for the web request

**method** The method for the web request

**content** Form data, post data or the data structure encoded as json for application/json content-type

**Returns**

**status_code** HTTP status code

**cookies** A list of k/v pair as cookies received from the web server

**headers** A list of k/v pair as headers received from the web server

**body** a string contains all response body

**json** if the return is json content type, this will be parsed json data blob

See below for a simple example

```
workflows:
  sending_request:
    call_driver: web.request
    with:
      URL: https://ifconfig.co
```

Below is an example of specifying header for the outbound request defined through a system function

```
systems:
  my_api_server:
    data:
      token: ENC[gcloud-kms,...masked...]
      url: https://foo.bar/api
    function:
      secured_api:
        driver: web
```

(continues on next page)

(continued from previous page)

```
        parameters:
          URL: $sysData.url
          header:
            Authorization: Bearer {{ .sysData.token }}
            content-type: application.json
        rawAction: request
```

### 4.2.5 webhook

This driver enables Honeydipper to receive incoming webhooks to trigger workflows

**Configurations**

**Addr** the address and port the webhook server is listening to

for example

```
---
drivers:
  webhook:
    Addr: :8080 # listening on all IPs at port 8080
```

#### Event: <default>

receiving an incoming webhook

**Returns**

**url** the path portion of the url for the incoming webhook request

**method** The method for the web request

**form** a list of k/v pair as query parameters from url parameter or posted form

**headers** A list of k/v pair as headers received from the request

**host** The host part of the url or the Host header

**remoteAddr** The client IP address and port in the form of *xx.xx.xx.xx:xxxx*

**json** if the content type is application/json, it will be parsed and stored in here

The returns can also be used in matching conditions

See below for a simple example

```
rules:
- do:
    call_workflow: foobar
  when:
    driver: webhook
    if_match:
      form:
        s: hello
      headers:
        content-type: application/x-www-form-urlencoded
      method: POST
      url: /foo/bar
```

Below is an example of defining and using a system trigger with webhook driver

```
systems:
  internal:
    data:
      token: ENC[gcloud-kms,...masked...]
    trigger:
      webhook:
        driver: webhook
        if_match:
          headers:
            Authorization: Bearer {{ .sysData.token }}
          remoteAddr: :regex:^10\.
rules:
  - when:
      source:
        system: internal
        trigger: webhook
      if_match:
        url: /foo/bar
    do:
      call_workflow: do_something
```

## 4.3 Systems

### 4.3.1 github

This system enables Honeydipper to integrate with *github*, so Honeydipper can react to github events and take actions on *github*.

**Configurations**

**oauth_token** The token or API ID used for making API calls to *github*

**token** A token used for authenticate incoming webhook requests, every webhook request must carry a form field **Token** in the post body or url query that matches the value

**path** The path portion of the webhook url, by default `/github/push`

For example

```
---
systems:
  github:
    data:
      oauth_token: ENC[gcloud-kms,...masked...]
      token: ENC[gcloud-kms,...masked...]
      path: "/webhook/github"
```

Assuming the domain name for the webhook server is :code:‘myhoneydipper.com’, you should configure the webhook in your repo with url like below

#### Trigger: hit

This is a catch all event for github webhook requests. It is not to be used directly, instead should be used as source for defining other triggers.

### Trigger: pr_comment

This is triggered when a comment is added to a pull request.

**Matching Parameters**

**.json.repository.full_name** This field is to match only the pull requests from certain repo

**.json.comment.user.login** This is to match only the comments from certain username

**.json.comment.author_association** This is to match only the comments from certain type of user. See github API reference here for detail.

**.json.comment.body** This field contains the comment message, you can use regular express pattern to match the content of the message.

**Export Contexts**

**git_repo** This context variable will be set to the name of the repo, e.g. `myorg/myrepo`

**git_user** This context variable will be set to the user object who made the comment

**git_issue** This context variable will be set to the issue number of the PR

**git_message** This context variable will be set to the comment message

See below snippet for example

```
---
rules:
  - when:
      source:
        system: github
        trigger: pr_commented
      if_match:
        json:
          repository:
            full_name: myorg/myrepo # .json.repository.full_name
          comment:
            autho_association: CONTRIBUTOR
            body: ':regex:^\s*terraform\s+plan\s*$'
    do:
      call_workflow: do_terraform_plan
      # following context variables are available
      #   git_repo
      #   git_issue
      #   git_message
      #   git_user
      #
```

### Trigger: pull_request

This is triggered when a new pull request is created

**Matching Parameters**

**.json.repository.full_name** This field is to match only the pull requests from certain repo

**.json.pull_request.base.ref** This field is to match only the pull requests made to certain base branch, note that the ref value here does not have the `ref/heads/` prefix (different from push event). So to match master branch, just use `master` instead of `ref/heads/master`.

**.json.pull_request.user.login** This field is to match only the pull requests made by certain user

**Export Contexts**

**git_repo** This context variable will be set to the name of the repo, e.g. `myorg/myrepo`

**git_ref** This context variable will be set to the name of the branch, e.g. `mybrach`, no `ref/heads/` prefix

**git_commit** This context variable will be set to the short (7 characters) commit hash of the head commit of the PR

**git_user** This context variable will be set to the user object who created the PR

**git_issue** This context variable will be set to the issue number of the PR

**git_title** This context variable will be set to the title of the PR

See below snippet for example

```
---
rules:
  - when:
      source:
        system: github
        trigger: pull_request
      if_match:
        json:
          repository:
            full_name: myorg/myrepo # .json.repository.full_name
          pull_request:
            base:
              ref: master           # .json.pull_request.base.ref
    do:
      call_workflow: do_something
      # following context variables are available
      #   git_repo
      #   git_ref
      #   git_commit
      #   git_issue
      #   git_title
      #   git_user
      #
```

### Trigger: push

This is triggered when **github** receives a push.

**Matching Parameters**

**.json.repository.full_name** Specify this in the `when` section of the rule using `if_match`, to filter the push events for the repo

**.json.ref** This field is to match only the push events happened on certain branch

**Export Contexts**

**git_repo** This context variable will be set to the name of the repo, e.g. `myorg/myrepo`

**git_ref** This context variable will be set to the name of the branch, e.g. `ref/heads/mybrach`

**git_commit** This context variable will be set to the short (7 characters) commit hash of the head commit of the push

See below snippet for example

```
---
rules:
  - when:
      source:
        system: github
        trigger: push
      if_match:
        json:
          repository:
            full_name: myorg/myrepo # .json.repository.full_name
          ref: ref/heads/mybranch   # .json.ref
    do:
      call_workflow: do_something
      # following context variables are available
      #   git_repo
      #   git_ref
      #   git_commit
      #
```

Or, you can match the conditions in workflow using exported context variables instead of in the rules

```
---
rules:
  - when:
      source:
        system: github
        trigger: push
    do:
      if_match:
        - git_repo: mycompany/myrepo
          git_ref: ref/heads/master
        - git_repo: myorg/myfork
          git_ref: ref/heads/mybranch
      call_workflow: do_something
```

### Function: api

This is a generic function to make a github API call with the configured oauth_token. This function is meant to be used for defining other functions.

**Input Contexts**

**resource_path** This field is used as the path portion of the API call url

### Function: createComment

This function will create a comment on the given PR

**Input Contexts**

**git_repo** The repo that commit is for, e.g. `myorg/myrepo`

**git_issue** The issue number of the PR

**message** The content of the comment to be posted to the PR

See below for example

```
---
rules:
  - when:
      source:
        system: github
        trigger: pull_request
    do:
      if_match:
        git_repo: myorg/myrepo
        git_ref: master
      call_function: github.createComment
      with:
        # the git_repo is available from event export
        # the git_issue is available from event export
        message: type `honeydipper help` to see a list of available commands
```

### Function: createStatus

This function will create a commit status on the given commit.

**Input Contexts**

**git_repo** The repo that commit is for, e.g. `myorg/myrepo`

**git_commit** The short commit hash for the commit the status is for

**context** the status context, a name for the status message, by default `Honeydipper`

**status** the status data structure according github API here

See below for example

```
---
rules:
  - when:
      source:
        system: github
        trigger: push
    do:
      if_match:
        git_repo: myorg/myrepo
        git_ref: ref/heads/testbranch
      call_workflow: post_status

workflows:
  post_status:
    call_function: github.createStatus
    with:
      # the git_repo is available from event export
      # the git_commit is available from event export
      status:
        state: pending
        description: Honeydipper is scanning your commit ...
```

#### Function: getContent

This function will fetch a file from the specified repo and branch.

**Input Contexts**

**git_repo** The repo from where to download the file, e.g. `myorg/myrepo`

**git_ref** The branch from where to download the file, no `ref/heads/` prefix, e.g. `master`

**path** The path for fetching the file, no slash in the front, e.g. `conf/nginx.conf`

**Export Contexts**

**file_content** The file content as a string

See below for example

```
---
workflows:
  fetch_circle:
    call_function: github.getContent
    with:
      git_repo: myorg/mybranch
      git_ref: master
      path: .circleci/config.yml
    export:
      circleci_conf: :yaml:{{ .ctx.file_content }}
```

### 4.3.2 jira

This system enables Honeydipper to integrate with *jira*, so Honeydipper can react to jira events and take actions on jira.

**Configurations**

**jira_credential** The credential used for making API calls to *jira*

**token** A token used for authenticate incoming webhook requests, every webhook request must carry a form field **Token** in the post body or url query that matches the value

**path** The path portion of the webhook url, by default `/jira`

**jira_domain** Specify the jira domain, e.g. `mycompany` for `mycompany.atlassian.net`

For example

```
---
systems:
  github:
    data:
      jira_credential: ENC[gcloud-kms,...masked...]
      jira_domain: mycompany
      token: ENC[gcloud-kms,...masked...]
      path: "/webhook/jira"
```

Assuming the domain name for the webhook server is :code:‘myhoneydipper.com’, you should configure the webhook in your repo with url like below

### Trigger: hit

This is a generic trigger for jira webhook events.

### Function: addComment

This function will add a comment to the jira ticket

**Input Contexts**

**jira_ticket** The ticket number that the comment is for

**comment_body** Detailed description of the comment

See below for example

```
---
workflows:
  post_comments:
    call_function: jira.addComment
    with:
      jira_ticket: $ctx.jira_ticket
      comment_body: |
        Ticket has been created by Honeydipper.
```

### Function: createTicket

This function will create a jira ticket with given information

**Input Contexts**

**jira_project** The name of the jira project the ticket is created in

**ticket_title** A summary of the ticket

**ticket_desc** Detailed description of the work for this ticket

**ticket_type** The ticket type, by default `Task`

**Export Contexts**

**jira_ticket** The ticket number of the newly created ticket

See below for example

```
---
workflows:
  create_jira_ticket:
    call_function: jira.createTicket
    with:
      jira_project: devlops
      ticket_title: upgrading kubernetes
      ticket_desc: |
        Upgrade the test cluster to kubernetes 1.16
```

### 4.3.3 kubernetes

This system enables Honeydipper to interact with kubernetes clusters. This system is intended to be extended to create systems represent actual kubernetes clusters, instead of being used directly.

**Configurations**

**source** The parameters used for fetching kubeconfig for accessing the cluster, should at least contain a `type` field. Currently, only `local` or `gcloud-gke` are supported. For `gcloud-gke` type, this should also include `service_account`, `project`, `zone`, and `cluster`.

**namespace** The namespace of the resources when operating on the resources within the cluster, e.g. deployments. By default, `default` namespace is used.

For example

```
---
systems:
  my_gke_cluster:
    extends:
      - kubernetes
    data:
      source:
        type: gcloud-gke
        service_account: ENC[gcloud-kms,...masked...]
        zone: us-central1-a
        project: foo
        cluster: bar
      namespace: mynamespace
```

#### Function: createJob

This function creates a k8s run-to-completion job with given job spec data structure. It is a wrapper for the kubernetes driver createJob rawAction. It leverages the pre-configured system data to access the kubernetes cluster. It is recommmended to use the helper workflows instead of using the job handling functions directly.

**Input Contexts**

**job** The job data structure following the specification for a run-to-completion job manifest yaml file.

**Export Contexts**

**jobid** The job ID of the created job

See below for example

```
---
workflow:
  create_job:
    call_function: my-k8s-cluster.createJob
    with:
      job:
        apiVersion: batch/v1
        kind: Job
        metadata:
          name: pi
        spec:
          template:
            spec:
```

(continues on next page)

(continued from previous page)

```
            containers:
            - name: pi
              image: perl
              command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
            restartPolicy: Never
          backoffLimit: 4
```

### Function: getJobLog

This function fetch all the logs for a k8s job with the given jobid. It is a wrapper for the kubernetes driver getJobLog rawAction. It leverages the pre-configured system data to access the kubernetes cluster. It is recommmended to use the helper workflows instead of using the job handling functions directly.

**Input Contexts**

**job** The ID of the job to fetch logs for

**Export Contexts**

**log** The logs organized in a map of pod name to a map of container name to logs.

**output** The logs all concatinated into a single string

See below for example

```
---
workflow:
  run_simple_job:
    steps:
      - call_function: my-k8s-cluster.createJob
        with:
          job: $ctx.job
      - call_function: my-k8s-cluster.waitForJob
        with:
          job: $ctx.jobid
      - call_workflow: my-k8s-cluster.getJobLog
        with:
          job: $ctx.jobid
```

### Function: recycleDeployment

This function is a wrapper to the kubernetes driver recycleDeployment rawAction. It leverages the pre-configured system data to access the kubernetes cluster.

**Input Contexts**

**deployment** The selector for identify the deployment to restart, e.g. `app=nginx`

See below for example

```
---
rules:
  - when:
      source:
        system: opsgenie
        trigger: alert
```

(continues on next page)

(continued from previous page)

```
    do:
      steps:
        - if_match:
            alert_message: :regex:foo-deployment
          call_function: my-k8s-cluster.recycleDeployment
          with:
            deployment: app=foo
        - if_match:
            alert_message: :regex:bar-deployment
          call_function: my-k8s-cluster.recycleDeployment
          with:
            deployment: app=bar
```

#### Function: waitForJob

This function blocks and waiting for a k8s run-to-completion job to finish. It is a wrapper for the kubernetes driver waitForJob rawAction. It leverages the pre-configured system data to access the kubernetes cluster. It is recommended to use the helper workflows instead of using the job handling functions directly.

**Input Contexts**

**job** The job id that the function will wait for to reach terminated states

**Export Contexts**

**job_status** The status of the job, either `success` or `failure`

See below for example

```
---
workflow:
  run_simple_job:
    steps:
      - call_function: my-k8s-cluster.createJob
        with:
          job: $ctx.job
      - call_function: my-k8s-cluster.waitForJob
        with:
          job: $ctx.jobid
      - call_workflow: notify
        with:
          message: the job status is {{ .job_status }}
```

### 4.3.4 opsgenie

This system enables Honeydipper to integrate with *opsgenie*, so Honeydipper can react to opsgenie alerts and take actions through opsgenie API.

**Configurations**

**API_KEY** The API key used for making API calls to *opsgenie*

**token** A token used for authenticate incoming webhook requests, every webhook request must carry a form field **Token** in the post body or url query that matches the value

**path** The path portion of the webhook url, by default `/opsgenie`

For example

```
---
systems:
  opsgenie:
    data:
      API_KEY: ENC[gcloud-kms,...masked...]
      token: ENC[gcloud-kms,...masked...]
      path: "/webhook/opsgenie"
```

Assuming the domain name for the webhook server is :code:‘myhoneydipper.com’, you should configure the webhook in your opsgenie integration with url like below

### Trigger: alert

This event is triggered when an opsgenie alert is raised.

**Matching Parameters**

**.json.alert.message** This field can used to match alert with only certain messages

**.json.alert.alias** This field is to match only the alerts with certain alias

**Export Contexts**

**alert_message** This context variable will be set to the detailed message of the alert.

**alert_alias** This context variable will be set to the alias of the alert.

**alert_Id** This context variable will be set to the short alert ID.

**alert_system** This context variable will be set to the constant string, `opsgenie`

**alert_url** This context variable will be set to the url of the alert, used for creating links

See below snippet for example

```
---
rules:
  - when:
      source:
        system: opsgenie
        trigger: alert
      if_match:
        json:
          alert:
            message: :regex:^test-alert.*$
    do:
      call_workflow: notify
      with:
        message: 'The alert url is {{ .ctx.alert_url }}'
```

### Function: heartbeat

This function will send a heartbeat request to opsgenie.

**Input Contexts**

**heartbeat** The name of the heartbeat as configured in your opsgenie settings

**Export Contexts**

**result** The return result of the API call

See below for example

```
---
workflows:
  steps:
    - call_workflow: do_something
    - call_function: opsgenie.heartbeat
      with:
        heartbeat: test-heart-beat
```

### Function: schedules

This function list all on-call schedules or fetch a schedule detail if given a schedule identifier.

**Important:** This function only fetches first 100 schedules when listing.

**Input Contexts**

**scheduleId** The name or ID or the schedule of interest; if missing, list all schedules.

**scheduleIdType** The type of the identifier, `name` or `id`.

**Export Contexts**

**schedule** For fetching detail, the data structure that contains the schedule detail

**schedules** For listing, a list of data structure contains the schedule details

See below for example

```
---
workflows:
  steps:
    - call_function: opsgenie.schedules
```

### Function: snooze

This function will snooze the alert with given alert ID.

**Input Contexts**

**alert_Id** The ID of the alert to be snoozed

**duration** For how long the alert should be snoozed, use golang time format

**Export Contexts**

**result** The return result of the API call

See below for example

```
---
rules:
  - when:
      source:
        system: opsgenie
```

(continues on next page)

(continued from previous page)

```
      trigger: alert
  do:
    if_match:
      alert_message: :regex:test-alert
    call_function: opsgenie.snooze
    #  alert_Id is exported from the event
```

### Function: users

This function gets the user detail with a given ID or list all users

**Input Contexts**

**userId** The ID of the user for which to get details; if missing, list users

**offset** Number of users to skip from start, used for paging

**query** `Field:value` combinations with most of user fields to make more advanced searches. Possible fields are `username`, `fullName blocked`, `verified`, `role`, `locale`, `timeZone`, `userAddress` and `createdAt`

**order** The direction of the sorting, `asc` or `desc`, default is `asc`

**sort** The field used for sorting the result, could be `username`, `fullname` or `insertedAt`.

**Export Contexts**

**user** The detail of user in a map, or a list of users

**users** The detail of user in a map, or a list of users

**opsgenie_offset** The offset that can be used for continue fetching the rest of the users, for paging

See below for example

```
---
workflows:
  steps:
    - call_function: opsgenie.users
      with:
        query: username:foobar
```

### Function: whoisoncall

This function gets the current on-call persons for the given schedule.

**Input Contexts**

**scheduleId** The name or ID or the schedule of interest, required

**scheduleIdType** The type of the identifier, `name` or `id`.

**flat** If true, will only return the usernames, otherwise, will return all including notification, team etc.

**Export Contexts**

**result** the data portion of the json payload.

See below for example

```
---
workflows:
  steps:
    - call_function: opsgenie.whoisoncall
      with:
        scheduleId: sre_schedule
```

### 4.3.5 slack

This system enables Honeydipper to integrate with *slack*, so Honeydipper can send messages to and react to commands from slack channels. This system uses `Custom Integrations` to integrate with slack. It is recommended to use `slack_bot` system, which uses a slack app to integrate with slack.

**Configurations**

**url** The slack incoming webhook integration url

**slash_token** The token for authenticating slash command requests

**slash_path** The path portion of the webhook url for receiving slash command requests, by default `/slack/slashcommand`

For example

```
---
systems:
  slack:
    data:
      url: ENC[gcloud-kms,...masked...]
      slash_token: ENC[gcloud-kms,...masked...]
      slash_path: "/webhook/slash"
```

To configure the integration in slack,

1. select from menu `Administration` => `Manage Apps`
2. select `Custom Integrations`
3. add a `Incoming Webhooks`, and copy the webhook url and use it as `url` in system data
4. create a random token to be used in slash command integration, and record it as `slash_token` in system data
5. add a `Slash Commands`, and use the url like below to send commands

#### Trigger: slashcommand

This is triggered when an user issue a slash command in a slack channel. It is recommended to use the helper workflows and the predefined rules instead of using this trigger directly.

**Matching Parameters**

**.form.text** The text of the command without the prefix

**.form.channel_name** This field is to match only the command issued in a certain channel, this is only available for public channels

**.form.channel_id** This field is to match only the command issued in a certain channel

**.form.user_name** This field is to match only the command issued by a certain user

**Export Contexts**

**response_url** Used by the `reply` function to send reply messages

**text** The text of the command without the slash word prefix

**channel_name** The name of the channel without # prefix, this is only available for public channels

**channel_fullname** The name of the channel with # prefix, this is only available for public channels

**channel_id** The IDof the channel

**user_name** The name of the user who issued the command

**command** The first word in the text, used as command keyword

**parameters** The remaining string with the first word removed

See below snippet for example

```
---
rules:
  - when:
      source:
        system: slack
        trigger: slashcommand
      if_match:
        form:
          channel_name:
            - public_channel1
            - channel2
      steps:
        - call_function: slack.reply
          with:
            chat_colors:
              this: good
            message_type: this
            message: command received `{{ .ctx.command }}`
        - call_workflow: do_something
```

### Function: reply

This function send a reply message to a slash command request. It is recommended to use `notify` workflow instead so we can manage the colors, message types and receipient lists through contexts easily.

**Input Contexts**

**chat_colors** a map from message_types to color codes

**message_type** a string that represents the type of the message, used for selecting colors

**message** the message to be sent

**blocks** construct the message using the slack `layout blocks`, see slack document for detail

See below for example

```
---
rules:
  - when:
      source:
        system: slack
        trigger: slashcommand
```

(continues on next page)

(continued from previous page)

```
    do:
      call_function: slack.reply
      with:
        chat_colors:
          critical: danger
          normal: ""
          error: warning
          good: good
          special: "#e432ad2e"
        message_type: normal
        message: I received your request.
```

#### Function: say

This function send a message to a slack channel slack incoming webhook. It is recommended to use `notify` workflow instead so we can manage the colors, message types and receipient lists through contexts easily.

**Input Contexts**

**chat_colors** A map from message_types to color codes

**message_type** A string that represents the type of the message, used for selecting colors

**message** The message to be sent

**channel_id** The id of the channel the message is sent to. Use channel name here only when sending to a public channel or to the home channel of the webhook.

**blocks** construct the message using the slack `layout blocks`, see slack document for detail

See below for example

```
---
rules:
  - when:
      source:
        system: something
        trigger: happened
    do:
      call_function: slack.say
      with:
        chat_colors:
          critical: danger
          normal: ""
          error: warning
          good: good
          special: "#e432ad2e"
        message_type: error
        message: Something happened
        channel_id: '#public_announce'
```

### 4.3.6 slack_bot

This system enables Honeydipper to integrate with *slack*, so Honeydipper can send messages to and react to commands from slack channels. This system uses slack app to integrate with slack. It is recommended to use this instead of `slack` system, which uses a `Custom Integrations` to integrate with slack.

**Configurations**

**token** The bot user token used for making API calls

**slash_token** The token for authenticating slash command requests

**interact_token** The token for authenticating slack interactive messages

**slash_path** The path portion of the webhook url for receiving slash command requests, by default `/slack/slashcommand`

**interact_path** The path portion of the webhook url for receiving interactive component requests, by default `/slack/interact`

For example

```
---
systems:
  slack_bot:
    data:
      token: ENC[gcloud-kms,...masked...]
      slash_token: ENC[gcloud-kms,...masked...]
      interact_token: ENC[gcloud-kms,...masked...]
      slash_path: "/webhook/slash"
      interact_path: "/webhook/slash_interact"
```

To configure the integration in slack,

1. select from menu `Administration` => `Manage Apps`
2. select `Build` from top menu, create an app or select an exist app from `Your Apps`
3. add feature `Bot User`, and copy the `Bot User OAuth Access Token` and record it as `token` in system data
4. create a random token to be used in slash command integration, and record it as `slash_token` in system data
5. add feature `Slash Commands`, and use the url like below to send commands
6. create another random token to be used in interactive components integration, and record it as `interact_token` in system data
7. add feature `interactive components` and use url like below

### Trigger: interact

This is triggered when an user responds to an interactive component in a message. This enables honeydipper to interactively reacts to user choices through slack messages. A builtin rule is defined to respond to this trigger, so in normal cases, it is not necessary to use this trigger directly.

**Export Contexts**

**slack_payload** The payload of the interactive response

### Trigger: slashcommand

This is triggered when an user issue a slash command in a slack channel. It is recommended to use the helper workflows and the predefined rules instead of using this trigger directly.

**Matching Parameters**

**.form.text** The text of the command without the prefix

**.form.channel_name** This field is to match only the command issued in a certain channel, this is only available for public channels

**.form.channel_id** This field is to match only the command issued in a certain channel

**.form.user_name** This field is to match only the command issued by a certain user

**Export Contexts**

**response_url** Used by the `reply` function to send reply messages

**text** The text of the command without the slash word prefix

**channel_name** The name of the channel without # prefix, this is only available for public channels

**channel_fullname** The name of the channel with # prefix, this is only available for public channels

**channel_id** The IDof the channel

**user_name** The name of the user who issued the command

**command** The first word in the text, used as command keyword

**parameters** The remaining string with the first word removed

See below snippet for example

```
---
rules:
  - when:
      source:
        system: slack
        trigger: slashcommand
      if_match:
        form:
          channel_name:
            - public_channel1
            - channel2
      steps:
        - call_function: slack.reply
          with:
            chat_colors:
              this: good
            message_type: this
            message: command received `{{ .ctx.command }}`
        - call_workflow: do_something
```

### Function: reply

This function send a reply message to a slash command request. It is recommended to use `notify` workflow instead so we can manage the colors, message types and receipient lists through contexts easily.

**Input Contexts**

**chat_colors** a map from message_types to color codes

**message_type** a string that represents the type of the message, used for selecting colors

**message** the message to be sent

**blocks** construct the message using the slack `layout blocks`, see slack document for detail

See below for example

```
---
rules:
  - when:
      source:
        system: slack
        trigger: slashcommand
    do:
      call_function: slack.reply
      with:
        chat_colors:
          critical: danger
          normal: ""
          error: warning
          good: good
          special: "#e432ad2e"
        message_type: normal
        message: I received your request.
```

### Function: say

This function send a message to a slack channel slack incoming webhook. It is recommended to use `notify` workflow instead so we can manage the colors, message types and receipient lists through contexts easily.

**Input Contexts**

**chat_colors** A map from message_types to color codes

**message_type** A string that represents the type of the message, used for selecting colors

**message** The message to be sent

**channel_id** The id of the channel the message is sent to. Use channel name here only when sending to a public channel or to the home channel of the webhook.

**blocks** construct the message using the slack `layout blocks`, see slack document for detail

See below for example

```
---
rules:
  - when:
      source:
        system: something
        trigger: happened
    do:
      call_function: slack.say
      with:
        chat_colors:
          critical: danger
          normal: ""
          error: warning
          good: good
          special: "#e432ad2e"
        message_type: error
        message: Something happened
        channel_id: '#public_announce'
```

#### Function: users

This function queries all users for the team

**Input Contexts**

**cursor** Used for pagination, continue fetching from the cursor

**Export Contexts**

**slack_next_cursor** Used for pagination, used by next call to continue fetch

**members** A list of data structures containing member information

```
---
workflows:
  get_all_slack_users:
    call_function: slack_bot.users
```

## 4.4 Workflows

### 4.4.1 channel_translate

translate channel_names to channel_ids

**Input Contexts**

**channel_names** a list of channel names to be translated

**channel_maps** a map from channel names to ids

**Export Contexts**

**channel_ids** a list of channel ids corresponding to the input names

By pre-populating a map, we don't have to make API calls to slack everytime we need to convert a channel name to a ID.

This is used by `slashcommand` workflow and `notify` workflow to automatically translate the names.

```
---
workflows:
  attention:
    with:
      channel_map:
        '#private_channel1': UGKLASE
        '#private_channel2': UYTFYJ2
        '#private_channel3': UYUJH56
        '#private_channel4': UE344HJ
        '@private_user':     U78JS2F
    steps:
      - call_workflow: channel_translate
        with:
          channel_names:
            - '#private_channel1'
            - '#private_channel3'
            - '@private_user'
            - '#public_channel1'
      - call_workflow: loop_send_slack_message
```

(continues on next page)

(continued from previous page)

```
        # with:
        #   channel_ids:
        #     - UGKLASE
        #     - UYUJH56
        #     - U78JS2F
        #     - '#public_channel1' # remain unchanged if missing from the map
```

### 4.4.2 notify

send chat message through chat system

**Input Contexts**

**chat_system** A system name that supports `reply` and `say` function, can be either `slack` or `slack_bot`, by default `slack_bot`.

**notify** A list of channels to which the message is beng sent, a special name `reply` means replying to the slashcommand user.

**notify_on_error** A list of additional channels to which the message is beng sent if the message_type is error or failure.

**message_type** The type of the message used for coloring, could be `success`, `failure`, `error`, `normal`, `warning`, or `announcement`

**chat_colors** A map from message_type to color codes. This should usually be defined in default context so it can be shared.

This workflow wraps around `say` and `reply` method, and allows multiple recipients.

For example

```
---
workflows:
  attention:
    call_workflow: notify
    with:
      notify:
        - "#honeydipper-notify"
        - "#myteam"
      notify_on_error:
        - "#oncall"
      message_type: $labels.status
      message: "work status is {{ .labels.status }}"
```

### 4.4.3 opsgenie_users

This workflow wraps around the `opsgenie.users` function and handles paging to get all users from Opsgenie.

### 4.4.4 reload

reload honeydipper config

**Input Contexts**

**force** If force is truy, Honeydipper will simply quit, expecting to be re-started by deployment manager.

For example

```
---
rules:
  - when:
      source:
        system: slack_bot
        trigger: slashcommand
    do:
      if_match:
        command: reload
      call_workflow: reload
      with:
        force: $?ctx.parameters
```

### 4.4.5 resume_workflow

resume a suspended workflow

**Input Contexts**

**resume_token** Every suspended workflow has a `resume_token`, use this to match the workflow to be resumed

**labels_status** Continue the workflow with a dipper message that with the specified status

**labels_reason** Continue the workflow with a dipper message that with the specified reason

**resume_payload** Continue the workflow with a dipper message that with the given payload

For example

```
---
rules:
  - when:
      source:
        system: slack_bot
        trigger: interact
    do:
      call_workflow: resume_workflow
      with:
        resume_token: $ctx.slack_payload.callback_id
        labels_status: success
        resume_payload: $ctx.slack_payload
```

### 4.4.6 run_kubernetes

run kubernetes job

**Input Contexts**

**system** The k8s system to use to create and run the job

**steps** The steps that the job is made up with. Each step is an `initContainer` or a `container`. The steps are executed one by one as ordered in the list. A failure in a step will cause the whole job to fail. Each step is defined with fields including `type`, `command`, or `shell`. The `type` tells k8s what image to use, the `command` is the command to be executed with language supported by that image. If a shell script needs to be executed, use `shell` instead of `command`.

Also supported are `env` and `volumes` for defining the environment variables and volumes specific to this step.

**env** A list of environment variables for all the steps.

**volumes** A list of volumes to be attached for all the steps. By default, there will be a `EmptyDir` volume attached at `/honeydipper`. Each item should have a *name* and *volume* and optionally a *subPath*, and they will be used for creating the volume definition and volume mount definition.

**workingDir** The working directory in which the command or script to be exected. By default, `/honeydipper`. Note that, the default `workingDir` defined in the image is not used here.

**script_types** A map of predefined script types. The `type` field in `steps` will be used to select the image here. `image` field is required. `command_entry` is used for defining the entrypoint when using `command` field in step, and `command_prefix` are a list or a string that inserted at the top of container args. Correspondingly, the `shell_entry` and `shell_prefix` are used for defining the entrypoint and argument prefix for running a *shell* script.

Also supported is an optional `securtyContext` field for defining the image security context.

**predefined_steps** A map of predefined steps. Use the name of the predefined step in `steps` list to easily define a step without specifying the fields. This makes it easier to repeat or share the steps that can be used in multiple places. We can also override part of the predefined steps when defining the steps with *use* and overriding fields.

**predefined_env** A map of predefined environment variables.

**predefined_volumes** A map of predefined volumes.

**nodeSelector** See k8s pod specification for detail

**affinity** See k8s pod specification for detail

**tolerations** See k8s pod specification for detail

**timeout** Used for setting the `activeDeadlineSeconds` for the k8s pod

**cleanupAfter** Used for setting the `TTLSecondsAfterFinished` for the k8s job, requires 1.13+ and the feature to be enabled for the cluster.

**Export Contexts**

**log** The logs of the job organized in map by container and by pod

**output** The concatinated log outputs as a string

**job_status** A string indicating if the job is `success` or `failure`

See below for a simple example

```
---
workflows:
  ci:
    call_workflow: run_kubernetes
    with:
      system: myrepo.k8s_cluster
      steps:
        - git_clone # predefined step
        - type: node
          workingDir: /honeydipper/repo
          shell: npm install && npm build && npm test
```

Another example with overrriden predefined step

```
---
workflows:
  make_change:
    call_workflow: run_kubernetes
    with:
      system: myrepo.k8s
      steps:
        - git_clone # predefined step
        - type: bash
          shell: sed 's/foo/bar/g' repo/package.json
        - use: git_clone # use predefined step with overriding
          name: git_commit
          workingDir: /honeydipper/repo
          shell: git commit -m 'change' -a && git push
```

### 4.4.7 send_heartbeat

sending heartbeat to alert system

**Input Contexts**

**alert_system** The alert system used for monitoring, by default `opsgenie`

**heartbeat** The name of the heartbeat

This workflow is just a wraper around the `opsgenie.heartbeat` function.

### 4.4.8 slack_users

This workflow wraps around the `slack_bot.users` function and make multiple calls to stitch pages together.

### 4.4.9 slashcommand

This workflow is used internally to respond to slashcommand webhook events. You don't need to use this workflow directly in most cases. Instead, customize the workflow using `_slashcommands` context.

**Input Contexts**

**slashcommands** A map of commands to their definitions. Each definition should have a brief `usage`, `workflow` `contexts`, and `allowed_channels` fields. By default, two commands are already defined, `help`, and `reload`. You can extend the list or override the commands by defining this variable in `_slashcommands` context.

**slash_notify** A recipient list that will receive notifications and status of the commands executed through slashcommand.

**Export Contexts**

**command** This variable will be passed the actual workflow invoked by the slashcommand. The command is the first word after the prefix of the slashcommand. It is used for matching the definition in `$ctx.slashcommands`.

**parameters** This variable will be passed the actual workflow invoked by the slashcommand. The parameters is a string that contains the rest of the content in the slashcommand after the first word.

You can try to convert the `$ctx.parameters` to the variables the workflow required by the workflow being invoked through the `_slashcommands` context.

```
---
contexts:
  _slashcommands:

######## definition of the commands ##########
    slashcommand:
      slashcommands:
        greeting:
          usage: just greet the requestor
          workflow: greet

######## setting the context variable for the invoked workflow ##########
    greet:
      recipient: $ctx.user_name # exported by slashcommand event trigger
      type: $ctx.parameters     # passed from slashcommand workflow
```

### 4.4.10 slashcommand/announcement

This workflow sends a announcement message to the channels listed in `slash_notify`. Used internally.

### 4.4.11 slashcommand/help

This workflow sends a list of supported commands to the requestor. Used internally.

### 4.4.12 slashcommand/status

This workflow sends a status message to the channels listed in `slash_notify`. Used internally.

### 4.4.13 snooze_alert

snooze an alert

**Input Contexts**

**alert_system** The alert system used for monitoring, by default `opsgenie`

**alert_Id** The Id of the alert, usually exported from the alert event

**duration** How long to snooze the alert for, using golang time format, by default `20m`

This workflow is just a wraper around the `opsgenie.snooze` function. It also sends a notification through chat to inform if the snoozing is success or not.

For example

```
---
rules:
  - when:
      source:
        system: opsgenie
        trigger: alert
    do:
      steps:
```

(continues on next page)

(continued from previous page)

```
      - call_workflow: snooze_alert
      - call_workflow: do_something
```

### 4.4.14 start_kube_job

This workflow creates a k8s job with given job spec. It is not recommended to use this workflow directly. Instead, use `run_kubernetes` to leverage all the predefined context variables.

### 4.4.15 use_local_kubeconfig

This workflow is a helper to add a step into `steps` context variable to ensure the in-cluster kubeconfig is used. Basically, it will delete the kubeconfig files if any presents. It is useful when switching from other clusters to local cluster in the same k8s job.

```
---
workflows:
  copy_deployment_to_local:
    steps:
      - call_workflow: use_google_credentials
      - call_workflow: use_gcloud_kubeconfig
        with:
          cluster:
            project: foo
            cluster: bar
            zone: us-central1-a
      - export:
          steps+:
            - type: gcloud
              shell: kubectl get -o yaml deployment {{ .ctx.deployment }} >␣
↪kuberentes.yaml
      - call_workflow: use_local_kubeconfig # switching back to local cluster
      - call_workflow: run_kubernetes
        with:
          steps+:
            - type: gcloud
              shell: kubectl apply -f kubernetes.yaml
```

### 4.4.16 workflow_announcement

This workflow sends announcement messages to the slack channels. It can be used in the hooks to automatically announce the start of the workflow executions.

```
---
workflows:
  do_something:
    with:
      hooks:
        on_first_action:
          - workflow_announcement
    steps:
      - ...
      - ...
```

### 4.4.17 workflow_status

This workflow sends workflow status messages to the slack channels. It can be used in the hooks to automatically announce the exit status of the workflow executions.

```
---
workflows:
  do_something:
    with:
      hooks:
        on_exit:
          - workflow_status
    steps:
      - ...
      - ...
```

CHAPTER 5

# Gcloud

Contains drivers that interactive with gcloud assets

## 5.1 Installation

Include the following section in your **init.yaml** under **repos** section

```
- repo: https://github.com/honeydipper/honeydipper-config-essentials
  path: /gcloud
```

## 5.2 Drivers

This repo provides following drivers

### 5.2.1 gcloud-dataflow

This driver enables Honeydipper to run dataflow jobs

#### Action: createJob

creating a dataflow job using a template

**Parameters**

**service_account** A gcloud service account key (json) stored as byte array

**project** The name of the project where the dataflow job to be created

**location** The region where the dataflow job to be created

**job** The specification of the job see gcloud dataflow API reference CreateJobFromTemplateRequest for detail

**Returns**

**job** The job object, see gcloud dataflow API reference Job for detail

See below for a simple example

```
---
workflows:
  start_dataflow_job:
    call_driver: gcloud-dataflow.createJob
    with:
      service_account: ...masked...
      project: foo
      location: us-west1
      job:
        gcsPath: ...
        ...
```

### Action: updateJob

updating a job including draining or cancelling

**Parameters**

**service_account** A gcloud service account key (json) stored as byte array

**project** The name of the project where the dataflow job to be created

**location** The region where the dataflow job to be created

**jobSpec** The updated specification of the job see gcloud dataflow API reference Job for detail

**jobID** The ID of the dataflow job

**Returns**

**job** The job object, see gcloud dataflow API reference Job for detail

See below for a simple example of draining a job

```
---
workflows:
  find_and_drain_dataflow_job:
    with:
      service_account: ...masked...
      project: foo
      location: us-west1
    steps:
      - call_driver: gcloud-dataflow.findJobByName
        with:
          name: bar
      - call_driver: gcloud-dataflow.updateJob
        with:
          jobID: $data.job.Id
          jobSpec:
            currentState: JOB_STATE_DRAINING
      - call_driver: gcloud-dataflow.waitForJob
```

(continues on next page)

(continued from previous page)

```
        with:
          jobID: $data.job.Id
```

### Action: waitForJob

This action will block until the dataflow job is in a terminal state.

**Parameters**

**service_account** A gcloud service account key (json) stored as byte array

**project** The name of the project where the dataflow job to be created

**location** The region where the dataflow job to be created

**jobID** The ID of the dataflow job

**interval** The interval between polling calls go gcloud API, 15 seconds by default

**timeout** The total time to wait until the job is in terminal state, 1800 seconds by default

**Returns**

**job** The job object, see gcloud dataflow API reference Job for detail

See below for a simple example

```
---
workflows:
  run_dataflow_job:
    with:
      service_account: ...masked...
      project: foo
      location: us-west1
    steps:
      - call_driver: gcloud-dataflow.createJob
        with:
          job:
            gcsPath: ...
            ...
      - call_driver: gcloud-dataflow.waitForJob
        with:
          interval: 60
          timeout: 600
          jobID: $data.job.Id
```

### Action: findJobByName

This action will find an active job by its name

**Parameters**

**service_account** A gcloud service account key (json) stored as byte array

**project** The name of the project where the dataflow job to be created

**location** The region where the dataflow job to be created

**name** The name of the job to look for

**Returns**

**job** A partial job object, see gcloud dataflow API reference Job for detail, only `Id`, `Name` and `CurrentState` fields are populated

See below for a simple example

```
---
workflows:
  find_and_wait_dataflow_job:
    with:
      service_account: ...masked...
      project: foo
      location: us-west1
    steps:
      - call_driver: gcloud-dataflow.findJobByName
        with:
          name: bar
      - call_driver: gcloud-dataflow.waitForJob
        with:
          jobID: $data.job.Id
```

### Action: waitForJob

This action will block until the dataflow job is in a terminal state.

**Parameters**

**service_account** A gcloud service account key (json) stored as byte array

**project** The name of the project where the dataflow job to be created

**location** The region where the dataflow job to be created

**jobID** The ID of the dataflow job

**interval** The interval between polling calls go gcloud API, 15 seconds by default

**timeout** The total time to wait until the job is in terminal state, 1800 seconds by default

**Returns**

**job** The job object, see gcloud dataflow API reference Job for detail

See below for a simple example

```
---
workflows:
  wait_for_dataflow_job:
    with:
      service_account: ...masked...
      project: foo
      location: us-west1
    steps:
      - call_driver: gcloud-dataflow.createJob
        with:
          job:
            gcsPath: ...
            ...
      - call_driver: gcloud-dataflow.waitForJob
        with:
```

(continues on next page)

(continued from previous page)

```
          interval: 60
          timeout: 600
          jobID: $data.job.Id
```

#### Action: getJob

This action will get the current status of the dataflow job

**Parameters**

**service_account** A gcloud service account key (json) stored as byte array

**project** The name of the project where the dataflow job to be created

**location** The region where the dataflow job to be created

**jobID** The ID of the dataflow job

**Returns**

**job** The job object, see gcloud dataflow API reference Job for detail

See below for a simple example

```
---
workflows:
  query_dataflow_job:
    with:
      service_account: ...masked...
      project: foo
      location: us-west1
    steps:
      - call_driver: gcloud-dataflow.createJob
        with:
          job:
            gcsPath: ...
            ...
      - call_driver: gcloud-dataflow.getJob
        with:
          jobID: $data.job.Id
```

### 5.2.2 gcloud-gke

This driver enables Honeydipper to interact with GKE clusters.

Honeydipper interact with k8s clusters through `kubernetes` driver. However, the `kubernetes` driver needs to obtain kubeconfig information such as credentials, certs, API endpoints etc. This is achieved through making a RPC call to k8s type drivers. This driver is one of the k8s type driver.

#### RPC: getKubeCfg

Fetch kubeconfig information using the vendor specific credentials

**Parameters**

**service_account** Service account key stored as bytes

**project** The name of the project the cluster belongs to

**location** The location of the cluster

**regional** Boolean, true for regional cluster, otherwise zone'al cluster

**cluster** The name of the cluster

**Returns**

**Host** The endpoint API host

**Token** The access token used for k8s authentication

**CACert** The CA cert used for k8s authentication

See below for an example usage on invoking the RPC from k8s driver

```
func getGKEConfig(cfg map[string]interface{}) *rest.Config {
  retbytes, err := driver.RPCCall("driver:gcloud-gke", "getKubeCfg", cfg)
  if err != nil {
    log.Panicf("[%s] failed call gcloud to get kubeconfig %+v", driver.Service, err)
  }

  ret := dipper.DeserializeContent(retbytes)

  host, _ := dipper.GetMapDataStr(ret, "Host")
  token, _ := dipper.GetMapDataStr(ret, "Token")
  cacert, _ := dipper.GetMapDataStr(ret, "CACert")

  cadata, _ := base64.StdEncoding.DecodeString(cacert)

  k8cfg := &rest.Config{
    Host:        host,
    BearerToken: token,
  }
  k8cfg.CAData = cadata

  return k8cfg
}
```

To configure a kubernetes cluster in Honeydipper configuration yaml `DipperCL`

```
---
systems:
  my-gke-cluster:
    extends:
      - kubernetes
    data:
      source:  # all parameters to the RPC here
        type: gcloud-gke
        service_account: ...masked...
        project: foo
        location: us-central1-a
        cluster: my-gke-cluster
```

Or, you can share some of the fields by abstracting

```
---
systems:
  my-gke:
```

(continues on next page)

(continued from previous page)

```
    data:
      source:
        type: gcloud-gke
        service_account: ...masked...
        project: foo

  my-cluster:
    extends:
      - kubernetes
      - my-gke
    data:
      source:  # parameters to the RPC here
        location: us-central1-a
        cluster: my-gke-cluster
```

### 5.2.3 gcloud-kms

This driver enables Honeydipper to interact with gcloud KMS to descrypt configurations.

In order to be able to store sensitive configurations encrypted at rest, Honeydipper needs to be able to decrypt the content. `DipperCL` uses e-yaml style notion to store the encrypted content, the type of the encryption and the payload/parameter is enclosed by the square bracket `[]`. For example.

```
mydata: ENC[gcloud-kms,...base64 encoded ciphertext...]
```

**Configurations**

**keyname** The key in KMS key ring used for decryption. e.g. `projects/myproject/locations/us-central1/keyRings/myring/cryptoKeys/mykey`

#### RPC: decrypt

Decrypt the given payload

**Parameters**

* The whole payload is used as a byte array of ciphertext

**Returns**

* The whole payload is a byte array of plaintext

See below for an example usage on invoking the RPC from another driver

```
retbytes, err := driver.RPCCallRaw("driver:gcloud-kms", "decrypt", cipherbytes)
```

### 5.2.4 gcloud-pubsub

This driver enables Honeydipper to receive and consume gcloud pubsub events

**Configurations**

**service_account** The gcloud service account key (json) in bytes. This service account needs to have proper permissions to subscribe to the topics.

For example

```
---
drivers:
  gcloud-pubsub:
    service-account: ENC[gcloud-gke,...masked...]
```

#### Event: <default>

An pub/sub message is received

**Returns**

**project** The gcloud project to which the pub/sub topic belongs to

**subscriptionName** The name of the subscription

**text** The payload of the message, if not json

**json** The payload parsed into as a json object

See below for an example usage

```
---
rules:
  - when:
      driver: gcloud-pubsub
      if_match:
        project: foo
        subscriptionName: mysub
        json:
          datakey: hello
    do:
      call_workflow: something
```

## 5.3 Workflows

### 5.3.1 use_gcloud_kubeconfig

This workflow will add a step into `steps` context variable so the following `run_kubernetes` workflow can use `kubectl` with gcloud service account credential

**Input Contexts**

**cluster** A object with `cluster` field and optionally, `project`, `zone`, and `region` fields

The workflow will add a step to run `gcloud container clusters get-credentials` to populate the kubeconfig file.

```
---
workflows:
  run_gke_job:
    steps:
      - call_workflow: use_google_credentials
      - call_workflow: use_gcloud_kubeconfig
        with:
          cluster:
            cluster: my-cluster
```

(continues on next page)

(continued from previous page)

```
    - call_workflow: run_kubernetes
      with:
        steps+:
          - type: gcloud
            shell: kubectl get deployments
```

### 5.3.2 use_google_credentials

This workflow will add a step into `steps` context variable so the following `run_kubernetes` workflow can use default google credentials or specify a credential through a k8s secret.

**Important:** It is recommended to always use this with `run_kubernetes` workflow if `gcloud` steps are used

#### Input Contexts

**google_credentials_secret** The name of the k8s secret storing the service account key, if missing, use default service account

For example

```
---
workflows:
  run_gke_job:
    steps:
      - call_workflow: use_google_credentials
        with:
          google_credentials_secret: my_k8s_secret
      - call_workflow: run_kubernetes
        with:
          steps+:
            - type: gcloud
              shell: gcloud compute disks list
```

CHAPTER 6

# Datadog

This repo offers a way to emit Honeydipper internal metrics to datadog

## 6.1 Installation

Include the following section in your **init.yaml** under **repos** section

```
- repo: https://github.com/honeydipper/honeydipper-config-essentials
  path: /datadog
```

## 6.2 Drivers

This repo provides following drivers

### 6.2.1 datadog-emitter

This driver enables Honeydipper to emit internal metrics to datadog so we can monitor how Honeydipper is performing.

**Configurations**

**statsdHost** The host or IP of the datadog agent to which the metrics are sent to, cannot be combined with `useHostPort`

**useHostPort** boolean, if true, send the metrics to the IP specified through the environment variable `DOGSTATSD_HOST_IP`, which usually is set to k8s node IP using `fieldRef`.

**statsdPort** string, the port number on the datadog agent host to which the metrics are sent to

For example

```
---
drivers:
  datadog-emitter:
    useHostPort: true
    statsdPort: "8125"
```

### RPC: counter_increment

Increment a counter metric

**Parameters**

**name** The metric name

**tags** A list of strings to be attached as tags

For example, calling from a driver

```
driver.RPC.Caller.CallNoWait(driver.Out, "emitter", "counter_increment",␣
↪map[string]interface{}{
  "name": "myapp.metric.counter1",
  "tags": []string{
    "server1",
    "team1",
  },
})
```

### RPC: gauge_set

Set a gauge value

**Parameters**

**name** The metric name

**tags** A list of strings to be attached as tags

**value** String, the value of the metric

For example, calling from a driver

```
driver.RPC.Caller.CallNoWait(driver.Out, "emitter", "gauge_set", map[string]interface
↪{}{
  "name": "myapp.metric.gauge1",
  "tags": []string{
    "server1",
    "team1",
  },
  "value": "1000",
})
```