redocs Documentation

Release 0.0.1

See AUTHORS

September 16, 2014

Contents

Release Engine is a collection of open-source tools that provides the functionality necessary to continuously deploy packaged code from a development team's continuous integration server of choice to all environments and automate certain business processes associated with the release of code.

The home of the Release Engines is on GitHub in the RHInception Organization.

Put very simply: the Release Engine is an orchestration tool (*re-core*) that runs commands on purpose-built workers (*re-worker*). The commands to run (and where to run them) are defined in *playbooks* in YAML or JSON format.

Interaction with the engine happens via a REST interface (*re-rest*). Additional workers exist for the purposes of aggregating logs, as well as sending notifications over any preferred method (such as email, or IRC). A bare-minimal Release Engine installation would require **re-rest**, **re-core**, and any given **re-worker**.

To learn more about the RH Inception group, follow us on the Red Hat Developer Blog under the tag inception.

CHAPTER 1

Introduction

This section provides a very high-level overview of the Release Engine and it's component parts. Let's begin with a high-level overview of the complete system.

1.1 Overview

This section is a narrative, or story, introducing us to the individiual roles each component plays in the Release Engine. At the end of this narrative we'll have learned:

- Overall workflow
- About the key components
- How components communicate

1.1.1 Scenario

We work in a software shop and we have been told to decrease our time to delivery. We took some measurements and realized that even with Jenkins and some home-brewed systems for deployment, we're still spending **20%** of our sprint time on *just deploying to test environments*. Let's focus on getting back that 20%.

How do we approach this? What functionality must be present in any kind of system which can automate deployments? Also consider that we're just one stop in a much larger shop. Given that constraint, it follows that whatever we build should be accessible to outsiders, extensible so other teams can build on it to fit their requirements, as well as have a clear language for describing steps in a release.

- Authentication and authorization
- Storage for deployment playbooks
- Loosey coupled components, so individual installations can scale to meet their owners requirements
- Something to manage all of the actual steps happening
- And, some sort of configurable notification system, so we can get updates in real time.

When used together, the Release Engine provides all these things.

1.2 Components

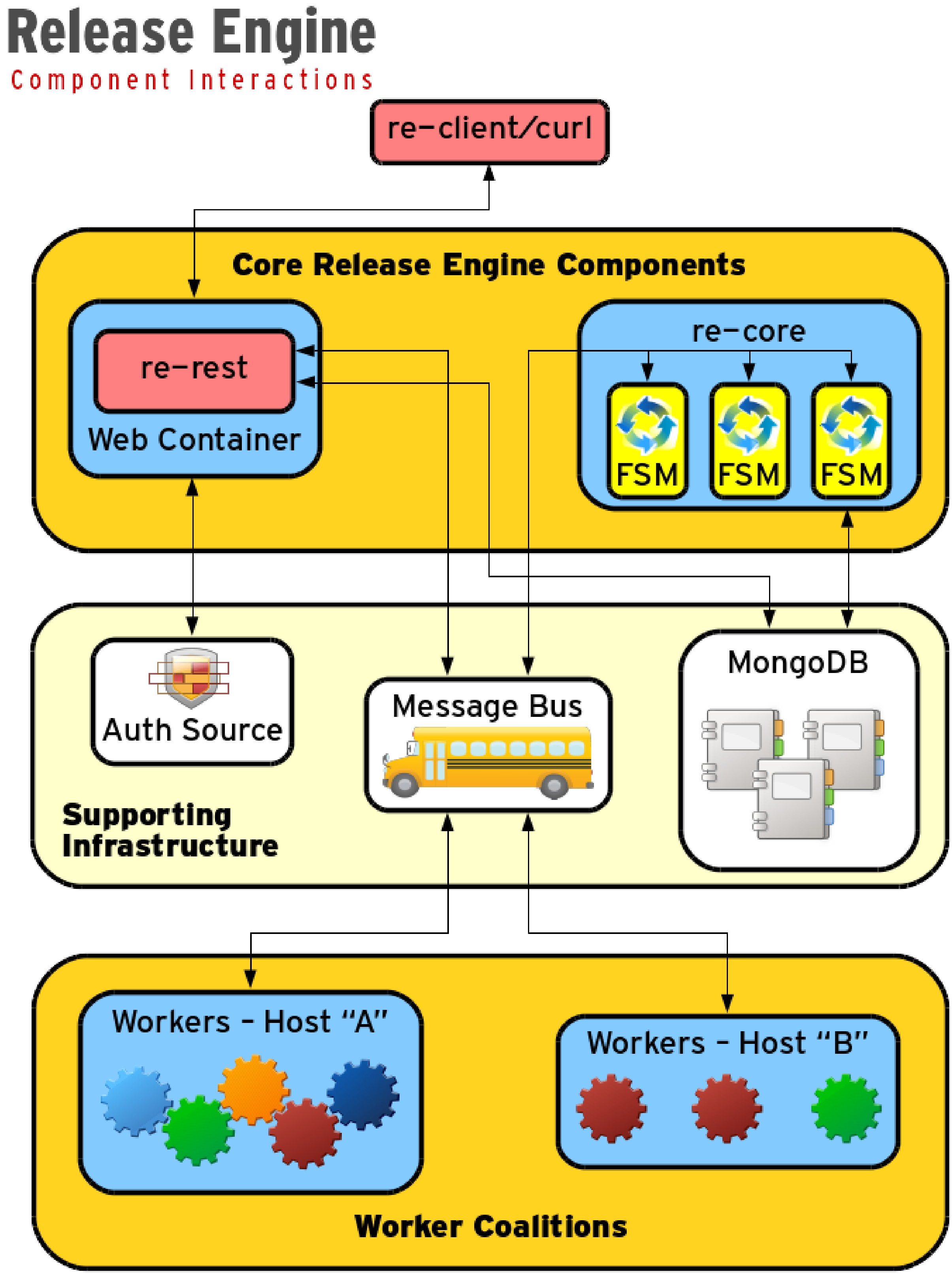
The Release Engine has three required components. Each of which is documented thoroughly in its own separate section. The following are brief descriptions of each component.

- *RE-CORE*
  - A finite state machine which oversees the execution of all steps required to complete a release
- *RE-REST*
  - A REST endpoint which handles authentication/authorization
  - The primary point of interaction for clients
- *One or more workers*
  - Workers are the components which are actually executed as release steps
  - There are several pre-built workers, you can view them on github

In addition to the required components:

- *RE-CLIENT*
  - Command line took for easily interacting with the Release Engine
  - Create, read, update, delete, and run playbooks

1.2.1 Component Diagram



1.2.2 Interactions & Workflow

This section describes how the Release Engine components interact with each other and the *supporting infrastructure*. We’ll review these interaction by examining a typical workflow.

CHAPTER 2

Setting Up

Table of Contents

2.1 Infrastructure Requirements

2.1.1 The Bus

Release Engine requires an AMQP service allowing messages to pass between components. The current, verified to work, AMQP service used with Release Engine is RabbitMQ, an Erlang-based open source messaging service. For more information on setting up a RabbitMQ server please read the project's server documentation.

For security best practices, each component that transmits on the bus should have it's own username and password combination. By enforcing component username/passwords access can be restricted to just what a component needs. This also allows quick deactivation of a component in the event something goes terribly wrong or a service is compromised.

Setup Steps

Note: Provision or utilize an existing server to install RabbitMQ or similar AMQP compliant service. For the rest of this article we will assume that you are running the service on RabbitMQ.

- Install RabbitMQ Server
- Open ports **5672** (*AMQP*) and **15672** (*management*)
- Enable RabbitMQ management via the Management Plugin
- Start RabbitMQ
- Create an exchange called "re" using topics

- Create a user for RE-REST (the rest interface into Release Engine)
- Create a user for RE-CORE (the state machine)
- Create a queue for RE-CORE
- Bind the RE-CORE queue to the re exchange with job.create
- Create a user for each component your instance will support
- Create a queue for each component your instance will support
- Bind the queue for each additional component (not including RE-REST and RE-CORE which are mandatory and separate from any additional components) to the re exchange with connectors that describe the step or plugin that your instance will support.

Todo

List binding instructions for queues

Test Setup

Todo

How to verify it's ready.

2.1.2 The Datastore

Release engine utilizes MongoDB for storing playbooks and other persistent data. Authentication must be turned on and it's highly recommended to create a username/password for every component that requires access to the data store.

Setup Steps

Note: Provision or utilize an existing server to install MongoDB or similar service like Amazon DynamoDB for the NoSQL service. For the rest of this article we will assume that you are running a local MongoDB service.

- Provision or choose a server to utilize for the datastore
- Install MongoDB on the server
- Open port 27017
- Update MongoDB for authentication (see the documentation)
- Start MongoDB
- Create a database called "re"
- Create a user for RE-CORE on database "re"
- Create a user for RE-REST on database "re"
- Import the initial data for the database via MongoDB Command Line Tools or one of the many MongoDB UI Tools.

Todo

Provide a link to the initial database import

Test Setup

Todo

How to verify it's ready.

2.2 Core Component Requirements

There are two components you must have no matter what workers you choose to support. These components are: RE-REST and RE-CORE.

2.2.1 RE-REST

RE-REST is the REST endpoint for interacting with the Release Engine. This is the only interaction point by design. RE-REST is a Flask based application and requires a few libraries before it will work properly.

Setup Steps

- Provision or choose a server to utilize for RE-REST
- Install Python v2
- Install the python v2 libraries listed on the re-rest GitHub page.
- Follow the RE-REST configuration instruction at *RE-REST → Configuration*.
- Choose and implement a RE-REST deployment strategy via *RE-REST Deployment*.

Test Setup

Todo

How to verify it's ready.

2.2.2 RE-CORE

The core is essentially a finite state machine (FSM) hooked into a message bus and a database.

The core oversees the execution of all release steps for any given project. The core is separate from the actual execution of each release step. Execution is delegated to the worker components.

Setup Steps

- Provision or choose a server to utilize for RE-CORE
- Install Python v2
- Install the python v2 libraries listed on the re-core GitHub page.
- Follow the RE-CORE configuration instructions at *RE-CORE Configuration*.
- Choose and implement a RE-CORE deployment strategy via *RE-CORE Deployment*.

Test Setup

Todo

How to verify it's ready.

CHAPTER 3

Components

3.1 RE-CORE

The core is essentially a finite state machine (**FSM**) hooked into a message bus and a database.

The core oversees the execution of all *release steps* for any given project. The core is separate from the actual execution of each release step. Execution is delegated to the **worker** component.

3.1.1 Running From Source

```
$ . ./hacking/setup-env
$ re-core -c ./examples/settings-example.json
```

3.1.2 RE-CORE Configuration

Configuration of the server is done in JSON. You can find an example configuration file in the examples/ directory.

You must point to a specific configuration file using the `-c` command-line option to start the FSM:

```
$ re-core -c settings.json
```

Descriptions of all settings directives:

Name	Type	Parent	Value
LOGFILE	str	None	File name for the application level log
RELEASE_LOG_DIR	str	None	Directory for per-release logging (default: `None`)
MQ	dict	None	Where all of the MQ connection settings are
SERVER	str	MQ	Hostname or IP of the server
NAME	str	MQ	Username to connect with
PASSWORD	str	MQ	Password to authenticate with
QUEUE	str	MQ	Queue on the server to bind
DB	dict	None	Where all the DB connection settings are
SERVERS	list	DB	List of all of the MongoDB hostname/IPs
DATABASE	str	DB	Name of the MongoDB database
NAME	str	DB	Username to connect with
PASSWORD	str	DB	Password to authenticate with
PHASE_NOTIFICATION	dict	None	Notifications that will always happen in a phase
TABOOT_URL	str	PHASE_NOTIFICATION	URL with `%s` to taboot tailer EX: http://example.com/taboot/%s/
TOPIC	str	PHASE_NOTIFICATION	The topic (*routing key*) to send notification on. EX: `notify.irc`.
TARGET	list	PHASE_NOTIFICATION	The targets to send the notification to. EX: `["#mychannel", "auser"]`
PRE_DEPLOY_CHECK	list	None	The yes/no checks to make prior to deployment (see below for more information)

For an example see example-config.json.

3.1.3 RE-CORE Deployment

Note: The release engine is only deployable via source code at this time.

Note: Release engine components are no fully demonized at this time. Therefore, deployment requires running each component in something like **screen**.

- Change into the directory you cloned **re-core** into
- Run **screen**
- Update your *re-core config file* with appropriate values
- Update your paths by running: `. ./hacking/setup-env`
- Run `re-core -c path/to/settings.json`

You should see output similar to the following:

```
[~/release-engine/re-core] $ re-core -c ./real-settings.json
2014-05-19 13:56:00,179 - __init__:start_logging:43 - DEBUG - initialized stdout logger
2014-05-19 13:56:00,180 - __init__:parse_config:53 - DEBUG - Parsed configuration file
```

Additional output will be directed to the log file you configured in the `settings.json` file. The default log file is called `recore.log` and will be in your present working directory.

3.1.4 Per-release Logging

By default, the FSM will log to the console and a single logfile (`LOGFILE`).

Optionally, one may log the FSM activity for **each release** to a separate file. This is done by configuring the re-core `RELEASE_LOG_DIR` setting with the path to the log-holding directory.

If per-release logging is enabled, the log files will be created as: `RELEASE_LOG_DIR/FSM-STATE_ID.log`

> **Warning:** Be sure the FSM has permission to write the specified directory. You won't find out it can't until the first release is attempted.

```
{
    "LOGFILE": "recore.log",
    "RELEASE_LOG_DIR": "/var/log/recore",
    "MQ": {
        "SERVER": "amqp.example.com"
    }
}
```

3.1.5 Pre-Deployment Checks

An re-core instance may be configured to run one or more scripts prior to the deployment of any playbook. Each pre-deployment check defines the command to run and the expected result from the command. If expected equals observed, then the check is considered to have passed. If expected is not equal to observed, then the check has failed and the entire deployment is marked as failed.

> **Important:** These checks apply to *all* deployments

Configuration of pre-deployment checks takes place in the re-core `setting.json` file.

Example settings

```
1  {
2      "LOGFILE": "recore.log",
3      "RELEASE_LOG_DIR": null,
4
5      "PRE_DEPLOY_CHECK": [{
6          "Require Change Record": {
7              "COMMAND": "servicenow",
8              "SUBCOMMAND": "getchangerecord",
9              "PARAMETERS": {},
10             "EXPECTATION": {
11                 "status": "completed",
12                 "data": {
13                     "exists": true
14                 }
15             }
16         }
17     }]
18 }
```

Here we see a new directive, `PRE_DEPLOY_CHECK` (line **5**), this key holds a list whose members are nested dictionaries (lines **6** → **16**). This example has one nested-dictionary. It has one key, that is the name of the check, **Require Change Record**. You can give any name you want to keys as long as it is JSON parsable.

Now let's look at this nested-dictionary closer:

```
{
    "COMMAND": "servicenow",
    "SUBCOMMAND": "getchangerecord",
```

```
    "PARAMETERS": {},
    "EXPECTATION": {
        "status": "completed",
        "data": {
            "exists": true
        }
    }
}
```

- COMMAND - Name of the worker to run the check with, *re-worker-servicenow* in this example
- SUBCOMMAND - The specific sub-command to run on that worker
- PARAMETERS - Dictionary with variable keys depending on what your worker requires
- EXPECTATION - The result we expected to get back from the check.

Pass or fail is determined by comparing the *actual* response against EXPECTATION. If they are the same then the check passes. If they differ then the check fails and the deployment is marked as *failed* and aborted.

3.2 RE-REST

Simple REST Api for the Release Engine. By design RE-REST is the only way to interact with the Release Engine.

Table of Contents

3.2.1 RE-REST Configuration

Configuration of the server is done in JSON and is by default kept in the current directories settings.json file.

You can override the location by setting `REREST_CONFIG` environment variable.

Name	Type	Parent	Value
LOGFILE	str	None	File name for the application level log
LOGLEVEL	str	None	DEBUG, INFO (default), WARN, FATAL
MQ	dict	None	Where all of the MQ connection settings are
SERVER	str	MQ	Hostname or IP of the server
PORT	int	MQ	Port to connect on
USER	str	MQ	Username to connect with
PASSWORD	str	MQ	Password to authenticate with
VHOST	str	MQ	vhost on the server to utilize
MONGODB_SETTINGS	dict	None	Where all of the MongoDB settings live
DB	str	MONGODB_Settings	Name of the database to use
USERNAME	str	MONGODB_Settings	Username to auth with
Password	str	MONGODB_Settings	Password to auth with
HOST	str	MONGODB_Settings	Host to connect to
PORT	int	MONGODB_Settings	Port to connect to on the host
PLAYBOOK_UI	bool	None	Turn's on/off the experimental playbook ui. It's off by default.
AUTHORIZATION_CALLABLE	str	None	module.location:callable. Eg: `rerest.authorization:no_authorization`
AUTHORIZATION_CONFIG	dict	None	Authorization callable specific configuration items

Further configuration items can be found in the Flask Documentation or look at specific `AUTHORIZATION_CALLABLE` documentation.

For an example see example-settings.json

3.2.2 Authentication

re-rest uses a simple decorator which enforces a `REMOTE_USER` be set.

rerest.decorators:remote_user_required

This decorator assumes that re-rest is running behind another web server which is taking care of authentication. If `REMOTE_USER` is passed to re-rest from the web server re-rest assumes authentication has succeeded. If it is not passed through re-rest treats the users as unauthenticated.

> **Warning:** When using this decorator it is very important that re-rest not be reachable by any means other than through the front end webserver!!

3.2.3 Authorization

re-rest uses a decorator which keys off the `AUTHORIZATION_CALLABLE` configuration parameters.

rerest.authroziation.no_authorization

Warning: This should not be used in a production environment**

To use this callable set `AUTHORIZATION_CALLABLE` to `rerest.authorization:no_authorization`.

rerest.authroziation.ldap_search

To use this callable set `AUTHORIZATION_CALLABLE` to `rerest.authorization:ldap_search` and set the following items in your configuration file.

Name	Type	Parent	Value
LDAP_URI	str	AUTHORIZATION_CONFIG	A full ldap URI such as `ldaps://127.0.0.1`
LDAP_USER	str	AUTHORIZATION_CONFIG	User to bind with
LDAP_PASSWORD	str	AUTHORIZATION_CONFIG	Password to bind with
LDAP_SEARCH_BASE	str	AUTHORIZATION_CONFIG	Search base for all queries. Ex: `dc=example,dc=com`
LDAP_MEMBER_ID	str	AUTHORIZATION_CONFIG	The name of the field that houses the username
LDAP_FIELD_MATCH	str	AUTHORIZATION_CONFIG	What field to use against the lookup table
LDAP_LOOKUP_TABLE	dict	AUTHORIZATION_CONFIG	key: list table of `LDAP_FIELD_MATCH` items to allowed groups. A * means all groups.

Here's a command-line example of how the `LDAP_LOOKUP_TABLE` property is used. In this example we will learn how authorization of the user **testuser** is determined.

Our organization has an ldap server at **ldap.example.com**, and groups are organized under the **ou=Groups,dc=example,dc=com** sub-tree. In this example re-rest will not attempt to **bind** (authenticate) with the LDAP server. Here is an example of this configuration:

```
{
    "AUTHORIZATION_CONFIG": {
        "LDAP_URI": "ldap://ldap.example.com",
        "LDAP_USER": "",
        "LDAP_PASSWORD": "",
        "LDAP_SEARCH_BASE": "ou=Groups,dc=example,dc=com",
        "LDAP_MEMBER_ID": "memberUid",
        "LDAP_FIELD_MATCH": "cn",
        "LDAP_LOOKUP_TABLE": {
            "admins": ["prod"],
            "superadmins": ["*"]
        }
    }
}
```

The **admins** group could look like this:

```
dn: cn=admins,ou=Groups,dc=example,dc=com
cn: admins
objectClass: top
objectClass: posixGroup
gidNumber: 1337
```

```
6 memberUid: testuser
7 memberUid: testboss
```

On line **6** we can see that this user is a member of the LDAP group **admins**. We also see here that group membership is denoted by use of the `memberUid` attribute. Note how this matches the the `LDAP_MEMBER_ID` setting we previously mentioned.

Let's pretend **testuser** is attempting to run a playbook with the *group* field set to **prod** (short for **production**). To determine authorization, **re-rest** will perform an LDAP search to query for records which match **two** conditions:

1. A record for a group exists in the `ou=Groups,dc=example,dc=com` sub-tree with a `cn` of **admins**
2. The discovered record has a `memberUid` attribute which matches the user's name: **testuser**

In LDAP search filter syntax, this query would look like the following:

```
(&(cn=admins)(memberUid=testuser))
```

With the `ldapsearch` command-line tool, we can test this authorization with the following command:

```
$ ldapsearch -xLLL -b ou=Groups,dc=example,dc=com -h ldap.example.com '(&(cn=admins)(memberUid=testus
```

If no results are returned, then the user is **not** authorized. If a result is resturned, then the user **is** authorized.

3.2.4 RE-REST Deployment

Apache with mod_wsgi

mod_wsgi can be used with Apache to mount rerest. Example mod_wsgi files are located in contrib/mod_wsgi.

- rerest.conf: The mod_wsgi configuration file. This should be modified and placed in /etc/httpd/conf.d/.
- rerest.wsgi: The WSGI file that mod_wsgi will use. This should be modified and placed in the location noted in rerest.conf

Gunicorn

Gunicorn (http://gunicorn.org/) is a popular open source Python WSGI server. It's still recommend to use Apache (or another web server) to handle auth before gunicorn since gunicorn itself is not set up for it.

```
$ gunicorn --user=YOUR_WORKER_USER --group=YOUR_WORKER_GROUP -D -b 127.0.0.1:5000 --access-logfile=/y
```

3.2.5 Running From Source

To run directly from source in order to test out the server run:

```
$ python rundevserver.py
```

The dev server will allow any HTTP Basic Auth user/password combination.

3.2.6 URLs

/api/v0/*$GROUP*/playbook/*$PLAYBOOKID*/deployment/

- **PUT**: Creates a new deployment.

- **Response Type**: json
- **Response Example**: `{"status": "created", "id": 1}`
- **Input Format**: None
- **Inputs**: optional json

/api/v0/playbooks/

- **GET**: Gets a list of **all** playbooks.
- **Response Type**: json
- **Response Example**: `{"status": "ok", "items": [...]}`
- **Input Format**: None
- **Inputs**: None

/api/v0/*$GROUP*/playbook/

- **GET**: Gets a list of all playbooks for a group.
- **Response Type**: json
- **Response Example**: `{"status": "ok", "items": [...]}`
- **Input Format**: None
- **Inputs**: None
- **PUT**: Creates a new playbook.
- **Response Type**: json
- **Response Example**: `{"status": "created", "id": "53614ccf1370129d6f29c7dd"}`
- **Input Format**: json/yaml
- **Inputs**: Optional format parameter which controls submit type. Can be json or yaml. Default is json.

/api/v0/*$GROUP*/playbook/*$ID*/

- **GET**: Gets a playbooks for a group.
- **Response Type**: json/yaml
- **Response Example**: `{"status": "ok", "item": ...}`
- **Input Format**: None
- **Inputs**: Optional format parameter which controls response type. Can be json or yaml. Default is json.
- **POST**: Replace a playbook in a group.
- **Response Type**: json
- **Response Example**: `{"status": "ok", "id": "53614ccf1370129d6f29c7dd"}`
- **Input Format**: json/yaml
- **Inputs**: Optional format parameter which controls response type. Can be json or yaml. Default is json.
- **DELETE**: Delete a playbook in a group.

- **Response Type**: json
- **Response Example**: `{"status": "gone"}`
- **Input Format**: None
- **Inputs**: None

3.2.7 Platform Gotchas

RHEL 6

You may need to add the following to your PYTHONPATH to be able to use Jinja2:

`/usr/lib/python2.6/site-packages/Jinja2-2.6-py2.6.egg`

3.2.8 What's Happening

1. User requests a new job via the REST endpoint
2. The REST server creates a temporary response queue and binds it to the exchange with the same name.
3. The REST server creates a message with a reply_to of the temporary response queue's topic.
4. The REST server sends the message to the bus on exchange *re* and topic *job.create*. Body Example: {"group": "nameofgroup"}
5. The REST server waits on the temporary response queue for a response.
6. Once a response is returned the REST service loads the body into a json structure and pulls out the id parameter.
7. The REST service then responds to the user with the job id.
8. The temporary response queue then is automatically deleted by the bus.

3.2.9 Usage Example

The authentication mechanism used in the front end webserver could be set up to use vastly different schemes. Instead of covering every possible authentication style which could be used we will work with two common ones in usage examples: htacces and kerberos.

Note: Setting up the front end proxy server for authentication is out of scope for this documentation.

htaccess / HTTP Basic Auth

```
$ curl -X PUT --user "USERNAME" -H "Content-Type: application/json" --data @file.json https://rerest
Password:

... # 201 and json data if exists, otherwise an error code
```

kerberos

```
$ kinit -f USERNAME
Password for USERNAME@DOMAIN:
$ curl --negotiate -u 'a:a' -H "Content-Type: application/json" --data @file.json -X PUT https://rere

... # 201 and json data if exists, otherwise an error code
```

Dynamic Variables

Passing dynamic variables requires two additions

1. We must set the `Content-Type` header (`-H ...` below) to `application/json`
2. We must pass **data** (`-d '{....}'` below) for the `PUT` to send to the server

This example sets the `Content-Type` and passes two **dynamic variables**: `cart` which is the name of a Juicer release cart, and `environment`, which is the environment to push the release cart contents to.

```
$ curl -u "user:passwd" -H "Content-Type: application/json" -d '{"cart": "bitmath", "environment": "r

 ... # 201 and json data if exists, otherwise an error code
```

See also:

- *RE-WORKER-JUICER*
- *Playbooks → Dynamic Variables*

Important: Release Engine workers require the *RE-WORKER* module

3.3 Playbook Workers

These workers are usable in *playbooks*.

3.3.1 RE-WORKER-BIGIP

Release Engine Worker Plugin that interfaces with F5 BigIP devices.

Attention: This plugin is internal to Red Hat only.

This worker takes the normal MQ configuration (`conf/mq_settings.json`) as it's only configuration file:

```
{
  "server": "127.0.0.1",
  "port": 5672,
  "vhost": "/",
  "user": "guest",
  "password": "guest"
}
```

- Set the `MQ config file` parameters to sane values (see also: *Setting Up The Bus*)
- Run the worker: `python ./replugin/emailworker/__init__.py $YOUR_MQ_CONF.json`

We should see output similar to the following if everything well:

```
[user@frober re-worker-bigip]$ python ./replugin/bigipworker/__init__.py
2014-05-19 14:39:47,080 - BigipWorker - WARNING - No app logger passed in. Defaulting to Streamandle
2014-05-19 14:39:47,083 - BigipWorker - INFO - Attempting connection with amqp://inceptadmin:***@mes
2014-05-19 14:39:47,412 - BigipWorker - INFO - Connection and channel open.
2014-05-19 14:39:47,413 - BigipWorker - INFO - Consuming on queue worker.bigip
```

Commands

The BigIP Worker steps are documented in *Worker Steps: BigIP*

3.3.2 RE-WORKER-FUNC

Release Engine Worker Plugin to run commands over FUNC.

What's FUNC?

Func stands for *Fedora Unified Network Controller*. Func allows for running commands on remote systems in a secure way, like SSH, but offers several improvements.

Func is extensible, as such it comes with several modules. Each module gives you more options for what you do with func. Here's a few of the modules which Func ships with:

- Command for running arbitrary commands somewhere remote
- Nagios for handling common tasks in Nagios related to downtime and notifications
- Service for starting, stopping, and checking the status of system services

This plugin allows you to run any number of func worker instances. Each instance is configured to allow for calling specific func module commands through it. However, this requires configuration before it can work.

Note: Installation and configuration of a func infrastructure is outside of the scope of this documentation. Please refer to the upstream documentation for additional information.

Configuration

Each running func worker requires a worker and an MQ configuration file. The worker configuration file defines exactly which func modules and methods the worker is allowed to run.

The configuration file uses the following **pattern** in JSON format:

```
{
    "queue": "QUEUE_NAME",
    "FUNC_MODULE": {
        "METHOD_1": ["REQUIRED", "PARAMETERS"],
        "METHOD_2": ["ONE_ITEM"],
```

```
6           "METHOD_N": []
7   }
```

In this example on line 2 we see a parameter `queue`. This is how we set a specific name for the queue to bind to on the message bus. This parameter is **required** if you are running more than one func worker concurrently. Using a name that defines what the workers does. For example, if you're using the Nagios plugin, you would create and bind to a queue such as `nagios` (which is expanded to `worker.nagios` internally).

The second configuration file is the normal MQ configuration used for connecting to the bus:

```
{
  "server": "127.0.0.1",
  "port": 5672,
  "vhost": "/",
  "user": "guest",
  "password": "guest"
}
```

- Set the `MQ config file` parameters to sane values (see also: *Setting Up The Bus*)
- Run the worker: `python ./replugin/funcworker/__init__.py` -w $YOUR_CONFIG_FILE.json $YOUR_MQ_CONF.json`

We should see output similar to the following if everything well:

```
[root@frober re-worker-func]# python ./replugin/funcworker/__init__.py
2014-05-19 14:39:47,080 - FuncWorker - WARNING - No app logger passed in. Defaulting to Streamandler
2014-05-19 14:39:47,083 - FuncWorker - INFO - Attempting connection with amqp://JoeUser:***@mq.exampl
2014-05-19 14:39:47,412 - FuncWorker - INFO - Connection and channel open.
2014-05-19 14:39:47,413 - FuncWorker - INFO - Consuming on queue worker.nagios
```

Example Configuration

Here is a real-life example of a func worker which may be used to run the `yumcmd` modules `install`, `remove`, and `update` methods.

```
{
    "yumcmd": {
        "install": ["package"],
        "remove": ["package"],
        "update": []
    }
}
```

In the above example we see on the `install` line that there is a list, `["package"]`, with one item in it. This means that when used as a step in a playbook a **single** `package` parameter must also be provided.

In contrast, we can see that the `update` method has an empty list, `[]`, following it. This indicates that the `yumcmd.update` method accepts no parameters. Using this method in a playbook step would update all packages on the target system.

The following is an example using the `yumcmd` module in a playbook step.

Commands

The FUNC Worker steps are documented in *Worker Steps: FUNC*.

Example: Installing a package

The following is an example of a *playbook* which installs a single package:

```
1  ---
2  group: inception
3  name: Setup megafrobber
4  execution:
5    - description: install the megafrobber package
6      hosts:
7        - foo.bar.example.com
8      steps:
9         - yumcmd:install:
10            package: megafrobber
```

Here we can see in lines **9** → **10** how to call the `install` sub-command for the **funcworker**.

Example: Stopping a Service

In this example *playbook* we will use the **service** sub-command to restart the **megafrobber** system service. For reference, first we'll look at the **funcworker** configuration for the **service** module:

```
{
    "service": {
        "stop": ["service"],
        "start": ["service"],
        "restart": ["service"],
        "reload": ["service"],
        "status": ["service"]
    }
}
```

Recall from what we learned in the *configuration* section that this defines one module, `service`.

As we can see above, the `service` module has 5 sub-commands, each requires one parameter, `service`, which is the name of the service to control.

The following example shows how to use the `funcworker.service.restart` method to restart the `megafrobber` service. This happens in lines **9** → **10**:

```
1  ---
2  group: inception
3  name: Setup megafrobber
4  execution:
5    - description: restart the megafrobber service
6      hosts:
7        - foo.bar.example.com
8      steps:
9         - service:restart:
10            service: megafrobber
```

Example: Trying/Checking

We can also add optional parameters `tries` and `check_scripts`. `check_scripts` is an array of scripts that will be run after the command. If they all return success (a zero return value) the whole command is considered successful. However if any return a non zero value the step is considered failed. The `tries` parameter tells the worker to try the step *X* number of times before giving up.

The following example will attempt the restart `megafrobber` and run the check_script `/usr/bin/diditwork`. If the either the restart or the check script return a failure it will try again until it's limit of 5 tries has been hit (at which point it returns failure back to the bus).

```
---
group: inception
name: Setup megafrobber
execution:
  - description: restart the megafrobber service
    hosts:
      - foo.bar.example.com
    steps:
       - service:restart:
           service: megafrobber
           tries: 5
           check_scripts: ["/usr/bin/diditwork"]
```

More Modules

The func worker ships with support for several other func modules out-of-the-box. To see them all, check out GitHub: re-worker-func/conf/

See Func - Module List for more information.

3.3.3 RE-WORKER-JUICER

Release Engine Worker Plugin to run Juicer commands

What's Juicer?

The **Juicer** worker allows you to upload and promote batches of RPMs into Yum repositories. In juicer terminology, these batches of RPMs are referred to as release carts.

About & Setup

A **juicer** worker allows you to upload/promote RPMs as just another step in your release process. **Note** however that the juicer plugin requires that additional information is passed to it when the release is started. See *Dynamic Variables* for more information on this topic.

To run the worker the normal MQ configuration must be defined and used.

```
{
  "server": "127.0.0.1",
  "port": 5672,
  "vhost": "/",
  "user": "guest",
```

```
  "password": "guest"
}
```

- Set the `MQ config file` parameters to sane values (see also: *Setting Up The Bus*)
- Run the worker
  - **From source:** `python ./replugin/juicerworker/__init__.py $YOUR_MQ_CONF.json`
  - **From install:** `re-worker-juicer $YOUR_MQ_CONF.json`

We should see output similar to the following if everything well:

```
[user@frober re-worker-juicerworker]$ re-worker-juicer mq_conf.json
2014-05-19 14:39:47,080 - JuicerWorker - WARNING - No app logger passed in. Defaulting to Streamandle
2014-05-19 14:39:47,083 - JuicerWorker - INFO - Attempting connection with amqp://inceptadmin:***@mes
2014-05-19 14:39:47,412 - JuicerWorker - INFO - Connection and channel open.
2014-05-19 14:39:47,413 - JuicerWorker - INFO - Consuming on queue worker.juicer
```

Juicer Client Configuration

See the upstream juicer configuration documentation for instructions on how to setup a system accounts juicer configuration file.

Dependencies

Use of the juicer worker requires a configured and running Pulp Server installation. Setup and maintenance of pulp servers is out of scope for this documentation. However, they provide detailed setup instructions to help get you started.

Commands

The Juicer Worker steps are documented in *Worker Steps: Juicer*.

3.3.4 RE-WORKER-SERVICENOW

Release Engine Worker Plugin that does basic interaction with Service Now.

This worker takes two configuration files. The first is the SERVICE NOW configuration file. It should look like this:

```
{
    "servicenow_user": "username",
    "servicenow_password": "secret",
    "api_root_url": "https://127.0.0.1/api/now/v1"
}
```

- Set the `MQ config file` parameters to sane values (see also: *Setting Up The Bus*)
- Run the worker
  - **From source:** `python ./replugin/servicenowworker/__init__.py -w $YOUR_SERVICE_NOW_CONF.json $YOUR_MQ_CONF.json`
  - **From install:** `re-worker-servicenow -w $YOUR_SERVICE_NOW_CONF.json $YOUR_MQ_CONF.json`

We should see output similar to the following if everything well:

```
[user@frober]$ re-worker-servicenow -w servicenow.json mq.json`
2014-05-19 14:39:47,080 - ServiceNowWorker - WARNING - No app logger passed in. Defaulting to Streama
2014-05-19 14:39:47,083 - ServiceNowWorker - INFO - Attempting connection with amqp://inceptadmin:***
2014-05-19 14:39:47,412 - ServiceNowWorker - INFO - Connection and channel open.
2014-05-19 14:39:47,413 - ServiceNowWorker - INFO - Consuming on queue worker.servicenow
```

Commands

The ServiceNow Worker steps are documented in *Worker Steps: ServiceNow*.

3.3.5 RE-WORKER-SLEEP

Release Engine Worker Plugin that sleeps for a period of seconds.

This worker takes the normal MQ configuration as it's only configuration file:

```
{
  "server": "127.0.0.1",
  "port": 5672,
  "vhost": "/",
  "user": "guest",
  "password": "guest"
}
```

- Set the `MQ config file` parameters to sane values (see also: *Setting Up The Bus*)
- Run the worker: `python ./replugin/emailworker/__init__.py $YOUR_MQ_CONF.json`

We should see output similar to the following if everything well:

```
[user@frober re-worker-sleep]$ python ./replugin/sleepworker/__init__.py
2014-05-19 14:39:47,080 - SleepWorker - WARNING - No app logger passed in. Defaulting to Streamandler
2014-05-19 14:39:47,083 - SleepWorker - INFO - Attempting connection with amqp://inceptadmin:***@mess
2014-05-19 14:39:47,412 - SleepWorker - INFO - Connection and channel open.
2014-05-19 14:39:47,413 - SleepWorker - INFO - Consuming on queue worker.sleep
```

Commands

The Sleep Worker steps are documented in *Worker Steps: Sleep*.

3.4 Auxiliary Workers

These workers are support workers and handle various other tasks. They are **not** usable in playbooks.

3.4.1 RE-WORKER-EMAILNOTIFY

Release Engine Worker Plugin can send notifications via email.

Note: This is a notification handler and is not meant to be used in steps.

This worker takes two configuration files. The first is the EMAILNOTIFY configuration file. It should look like this:

```
{
  "smtp_host": "127.0.0.1",
  "smtp_port": 25,
  "smtp_from": "noreply@example.com"
}
```

- Set the `EMAILNOTIFY` parameters to match your email server configuration.

The second configuration file is the normal MQ configuration:

```
{
  "server": "127.0.0.1",
  "port": 5672,
  "vhost": "/",
  "user": "guest",
  "password": "guest"
}
```

- Set the `MQ config file` parameters to sane values (see also: *Setting Up The Bus*)
- Run the worker: `python ./replugin/emailworker/__init__.py` -w $YOUR_CONFIG_FILE.json $YOUR_MQ_CONF.json`

We should see output similar to the following if everything well:

```
[user@frober re-worker-emailnotify]$ python ./replugin/emailworker/__init__.py -w myconf.json mq.jso
2014-05-19 14:39:47,080 - EmailNotifyWorker - WARNING - No app logger passed in. Defaulting to Strea
2014-05-19 14:39:47,083 - EmailNotifyWorker - INFO - Attempting connection with amqp://inceptadmin:*
2014-05-19 14:39:47,412 - EmailNotifyWorker - INFO - Connection and channel open.
2014-05-19 14:39:47,413 - EmailNotifyWorker - INFO - Consuming on queue worker.emailnotifyworker
```

3.4.2 RE-WORKER-IRCNOTIFY

Release Engine Worker Plugin can send notifications to IRC.

Note: This is a notification handler and is not meant to be used in steps.

This worker takes two configuration files. The first is the IRCNOTIFY configuration file. It should look like this:

```
{
  "server": "127.0.0.1",
  "port": 6697,
  "ssl": true,
  "channels": ["#release-engine"],
  "nick": "renotify"
}
```

- Set the `IRCNOTIFY` parameters to match your IRC server configuration.

The second configuration file is the normal MQ configuration:

```
{
  "server": "127.0.0.1",
  "port": 5672,
  "vhost": "/",
  "user": "guest",
  "password": "guest"
}
```

- Set the `MQ config file` parameters to sane values (see also: *Setting Up The Bus*)
- Run the worker: `re-worker-ircnotify -w $YOUR_CONFIG_FILE.json $YOUR_MQ_CONF.json`

We should see output similar to the following if everything well:

```
[user@frober re-worker-ircnotify]$ re-worker-ircnotify -w $YOUR_CONFIG_FILE.json $YOUR_MQ_CONF.json
2014-05-19 14:39:47,080 - IRCNotifyWorker - WARNING - No app logger passed in. Defaulting to Streamar
2014-05-19 14:39:47,083 - IRCNotifyWorker - INFO - Attempting connection with amqp://inceptadmin:***(
2014-05-19 14:39:47,412 - IRCNotifyWorker - INFO - Connection and channel open.
2014-05-19 14:39:47,413 - IRCNotifyWorker - INFO - Consuming on queue worker.ircnotifyworker
```

3.4.3 RE-WORKER-OUTPUT

Release Engine Worker Plugin consumes output from other works on the bus and writes it to files.

Note: This is an output consumer and is not meant to be used in steps.

This worker takes two configuration files. The first is the worker configuration file. It should look like this:

```
{
    "queue": "output",
    "output_dir": "/tmp/"
}
```

- Set the `output_dir` to where the output files should reside.

The second configuration file is the normal MQ configuration:

```
{
  "server": "127.0.0.1",
  "port": 5672,
  "vhost": "/",
  "user": "guest",
  "password": "guest"
}
```

- Set the `MQ config file` parameters to sane values (see also: *Setting Up The Bus*)
- Run the worker * **From source:** `python ./replugin/outputworker/__init__.py` -w $YOUR_CONFIG_FILE.json $YOUR_MQ_CONF.json` * **From install:** `re-worker-output -w $YOUR_SERVICE_NOW_CONF.json $YOUR_MQ_CONF.json`

Note: You may need to add the following to your PYTHONPATH to be able to use Jinja2 on RHEL6: `/usr/lib/python2.6/site-packages/Jinja2-2.6-py2.6.egg`

We should see output similar to the following if everything well:

```
[user@frober]$ re-worker-output -w myconf.json mq.json
2014-05-19 14:39:47,080 - IRCNotifyWorker - WARNING - No app logger passed in. Defaulting to Streamar
2014-05-19 14:39:47,083 - IRCNotifyWorker - INFO - Attempting connection with amqp://inceptadmin:***(
2014-05-19 14:39:47,412 - IRCNotifyWorker - INFO - Connection and channel open.
2014-05-19 14:39:47,413 - IRCNotifyWorker - INFO - Consuming on queue worker.output
```

3.5 Libraries/Helpers

3.5.1 RE-CLIENT

Release Engine - Client Tool

Todo

Define what this is better.

Running From Checkout

Todo

How do they install it?

```
$ export PYTHONPATH=`pwd`/src/:$PYTHONPATH
$ ./bin/re-client
```

At this point you'll be prompted to enter some configuration values:

```
Could not load base rerest url from /home/tbielawa/.reclient.conf
Enter the hostname of your rerest endpoint
This will be saved in /home/tbielawa/.reclient.conf for reuse later
Hostname:
```

At this point you would enter the hostname of your re-rest endpoint.

```
Hostname: rerest.example.com

0) Get all playbooks ever
1) Get all playbooks for a project
2) Get a single playbook for a project
3) Update a playbook
4) Delete a playbook
5) Create a new playbook
6) Start a deployment (without any dynamic keys)
7) Quit
command>>
```

Release Engine Client Tools are now setup and ready to use.

Command Line Options

The `re-client` command accepts two optional parameters:

- `--project`, `-p` - Set the default project
- `--id`, `-i` - Set the default playbook ID
- `--format`, `-f` - Select yaml/json for the format. Default: yaml

Usage Example

Let's work with **example project** for the duration of this session:

```
$ ./bin/re-client -p 'example project'
```

Notes

The REPL (command loop) has **readline** history enabled. This means the up/down arrow keys work and you can edit lines of input **like a boss**.

3.5.2 RE-WORKER

This library provides a simple base for release engine workers to build from.

Implementing a Worker

To implement a worker, subclass off of `reworker.worker.Worker` and override the `process` method.

If there are any inputs that need to be passed in, the class level variable `dynamic` should be populated.

```
class MyWorker(Worker):

    dynamic = ('environment', 'cart')

    ...
```

If a `config_file` is passed in on Worker creation it will be loaded as JSON and available as `self._config`. Otherwise `self._config` will be an empty dictionary.

Logging

When implementing your own worker, the `re-worker` base-class provides two mechanisms for logging and reporting worker progress.

Application-level Recording the kind of information system administrators need to see for debugging is facilitated by the `self.app_logger` instance variable. This information is logged to `stdout`.

```
def process(self, channel, basic_deliver, properties, body, output):

    # ...

    self.app_logger.debug("Going to frob the widget.")
```

User-level Reporting progress is facilitated by the `output` parameter which is passed to the `Worker.process` method. This information is send back to the message bus where it is then collected and saved by the *output worker*.

```
def process(self, channel, basic_deliver, properties, body, output):
    self.output = output

    # ...

```

```
    self.output.info("Release step successful with result: %s" % (
        str(result)))
```

Convenience Methods

Worker also provides a few convenience methods to simplify use:

Todo

Change to using actual sphinx API documentation.

Worker.send

Sends a message.

- **Inputs**:
- `topic`: the routing key
- `corr_id`: the correlation id
- `message_struct`: the dict or list to send as the body
- `exchange`: set to **''** to reply back to the FSM
- **Returns**: None

Worker.notify

- **Inputs**:
- `slug`: the short text to use in the notification
- `message`: a string which will be used in the notification
- `phase`: the phase to identify with in the notification
- `corr_id`: the correlation id. Default: **None**
- `exchange`: the exchange to publish on. Default: **re**
- **Returns**: None

Worker.ack

Acks a message.

- **Inputs**:
- `basic_deliver`: pika.Spec.Basic.Deliver instance
- **Returns**: None

Worker.run_forever

Starts the main loop.

- **Inputs**: None
- **Returns**: None

Worker.process

What a worker should do when a message is received. All output should be written to the *output logger*.

- **Inputs**:
- `channel`: pika.channel.Channel instance
- `basic_deliver`: pika.Spec.Basic.Deliver instance
- `properties`: pika.Spec.BasicProperties instance (ex: headers)
- `body`: dict or list that was json loaded off the message
- `output`: logger like instance to send output
- **Returns**: None

Running

Todo

Update this with how to run a custom **non-packaged** worker from source.

To run an instance you will need to make an instance of your worker by passing in a few items.

- **Inputs**:
  - mq_config: should house: user, password, server, port and vhost.
  - config_file: is an optional full path to a json config file
  - logger: is an optional logger. Defaults to a logger to stderr

CHAPTER 4

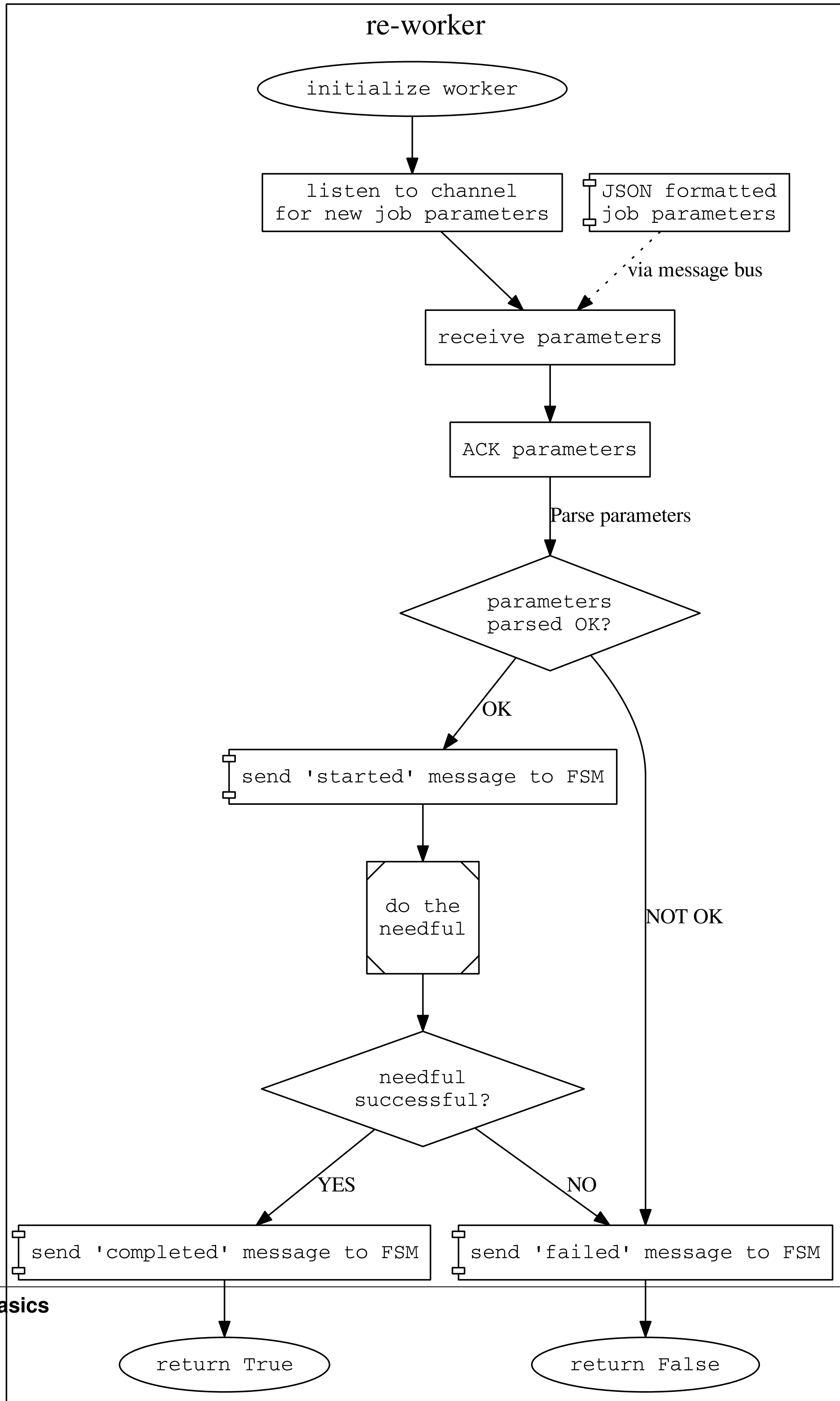
Tutorial: Writing Workers

This section contains tutorials for writing your own workers. If you haven't already, please take a few minutes and familiarize yourself with the *re-worker API documentation*.

4.1 Basics

Simple stuff.

4.1.1 Typical Worker Flow



Note: Not included in the chart are some of the various logging/notification steps which take place in a release.

Now, let's translate what this is saying into human readable words:

initialize worker In this state a process running the code representing our worker has just been started.

listen to channel Once the process has started our worker will open a channel to the message bus and begin consuming from a queue specifically dedicated to that kind of worker.

receive parameters Once our worker is consuming from its queue, it will sit in a waiting state until a message is received from the FSM (*re-core*). This message is only sent when a release is started.

The message will contain the parameters, or more generally speaking, the *configuration* of this step in the release. See the *re-core re-workers* Deployment Message Format documentation for the specifics on what is contained in this information.

Note: The message consumed from the message queue is a serialized JSON datastructure. Most workers will deserialized this message using a JSON library feature, such as the Python json.loads() method.

The **most important** piece of information contained in this first message is the `reply_to` property. This property tells our worker the name of the temporary queue to continue all further communication with the FSM on. Messages sent to *any other* destination will be lost in the message exchange.

Additionally, this first message will contain a **correlation ID**. This information should be recorded by our worker because it is used for logging and communicating back to the FSM.

ACK parameters Now that our worker has received its job parameters, the next step is to **acknowledge** receiving the parameters. Our worker does this using its AMQP librarie's `ack` function or method. If our worker is using the Python Pika AMQP module, it will provide the `delivery tag` as the parameter to the `ack` function.

parse parameters If our worker requires any unique information to do its job, then it must parse that information from the parameters provided by the FSM. This step typically involves verifying everything it needs to operate was provided and is valid.

This information often includes reading which sub-command (if applicable) to run, what (if any) hosts to operate on, parameters to provide to subcommands, and *dynamic values* passed in at deployment-time.

parameters verified If the parameters are parsed and it is verified that all the required information is present, then our worker will *reply* back to the FSM indicating that it is going to start running the step now.

The body of the message sent back to the FSM is a JSON **serialized** datastructure. See the *Response Message Format* documentation in the re-core re-worker docs for more information.

Workers using the *re-worker* library typically respond by calling the *worker.send()* method. When responding they should provide the `reply_to` variable as the `topic` parameter and leave the `exchange` parameter as an empty string.

parameters invalid Our worker must notify the FSM in the unfortunate event that the parameters provided were invalid. Similar to the previous step (valid parameters) our worker will use its `send()` method to send a *job failed* message.

Once the message has been sent our worker will abort all further execution. If the worker is designed such that it runs in a some kind of io-loop (such as in the *re-worker* library), this is as simple as returning `False` while still in the `process()` method.

do the needful At this point our worker has been initialized, received operating parameters from the FSM, and communicated back that it is going to proceed with the release. The next step is for the worker to begin doing what it was instructed to do.

The specifics of what happens in this step are different from worker to worker. The *BigIP* worker, for example, will run one of three sub-commands at this point. The exact sub-command is dictated by the value of the `subcommand` parameter.

step complete If *the needful* was a success, then our worker will reply back to the FSM one last time (again, using its `send()` method) with a JSON serialized datastructure. The message will include a `status` key set to `completed`.

After the message has been sent the worker will return `True` and continue its loop to begin the process all over again.

step failed If *the needful* was **not** a success, then our worker will reply back to the FSM one last time (again, using its `send()` method) with a JSON serialized datastructure. The message will include a `status` key set to `failed` and possibly another key, `data` with various information about the exact nature of the failure.

After the message has been sent the worker will return `False` and continue its loop to begin the process all over again.

4.1.2 Exercise: Write a Worker from Scratch

In this section we will build a worker from scratch. The worker will be written in Python. Additionally, the worker will utilize the *re-worker* library.

To keep things simple, our new worker will pretend to frob ("manipulate or adjust, to tweak") an arbitrary *thing* and then report the results. This worker will be called the **megafrobber** worker. The **megafrobber** worker will have one sub-command: `frob`.

The `frob` sub-command requires no arguments. When the sub-command is ran, it will take no actual actions. It will just randomly pass or fail.

This section is separated into several sub-sections. Each sub-section will incrementally build upon the work of the preceeding sections. At the end, we'll have a deployable worker.

Directory Structure

Workers adhere to the following directory structure:

```
re-worker-megafrobber/     - Top level
-- replugin/              - Python package directory
    -- megafrobberworker/ - Worker code directory
    |   -- __init__.py    - Worker code
    -- __init__.py        - Empty file, Python module requirement
```

In a command-line shell, you could create this structure using the following commands:

```
$ WORKER=megafrobber
$ mkdir -p re-worker-${WORKER}/replugin/${WORKER}worker
$ touch re-worker-${WORKER}/replugin/__init__.py
$ touch re-worker-${WORKER}/replugin/${WORKER}worker/__init__.py
$ find .
.
./re-worker-megafrobber
./re-worker-megafrobber/replugin
./re-worker-megafrobber/replugin/__init__.py
./re-worker-megafrobber/replugin/megafrobberworker
./re-worker-megafrobber/replugin/megafrobberworker/__init__.py
```

Scaffolding: Shebang and Imports

Note: The remainder of this tutorial assumes the present working directory is `re-worker-megafrobber`, the top-level directory

With our directory now created, we can begin filling in some scaffolding for our new worker. All of the following code snippets go into `replugin/megafrobberworker/__init__.py`.

The first things we'll add are the Python shebang and some standard imports:

```
1 #!/usr/bin/env python
2 import reworker.worker
3 import logging
```

The shebang (line **1**) is just there so that this script can be executed from the command line. It tells our shell (ex: BASH) what program to run the rest of the script in.

The import on line **2** will provide the standard **re-worker** library for us. Finally, line **3** will allow us to properly output application behavior.

Scaffolding: Class Definition

Following our imports comes the class definition. As we noted previously, this example worker will use the **re-worker** library. The **re-worker** library includes one class, `reworker.worker.Worker`.

As per the *re-worker* documentation, to *use* this class, our worker must:

- Subclass `reworker.worker.Worker` (line **1**)
- Define a `process` method (line **6**)

As we can see on line **1**, we call our class `MegafrobberWorker`.

```
1 class MegafrobberWorker(reworker.worker.Worker):
2     """
3     Plugin to frob the heck out of something
4     """
5
6     def process(self, channel, basic_deliver, properties, body, output):
```

The parameters that we see defined on line **6** are required. This is because of how the **re-worker** message bus integration code is written.

1. **re-worker** connects to the bus automatically upon startup
2. **re-worker** begins consuming from the workers dedicated queue
3. Upon receiving a message a callback is ran by the AMQP library (we use Pika for this). That callback flows into our `process` method
4. Once in the `process` method, the actual worker **work** happens (that's where we are now)

See also:

The Pika Documentation You can read more about callbacks and their usage on the Pika website.

Scaffolding: Record Job Properties

Our `process` method has a lot of arguments, this can appear overwhelming. Which do we need to care about?

To get us started, here are some common setup actions we might take with these properties.

```
def process(self, channel, basic_deliver, properties, body, output):
    # Output is a logger from the python logger library. This is
    # what we report progress through
    self.output = output

    # This is the ID given to the currently happening deployment. It
    # is a unique ID used to connect all passed messages together and
    # record the deployment state in the database.
    #
    # We use it when responding to the FSM.
    self.corr_id = str(properties.correlation_id)

    # If the FSM passed us any dynamic variables, they will be in
    # the 'dynamic' key of the body parameter
    dynamic = body.get('dynamic', {})

    # reply_to is the temporary message bus queue we respond to the
    # FSM through
    self.reply_to = properties.reply_to
```

Scaffolding: Make It Runnable

There are only two more things we need to add to make our worker runnable from the command line. The first is a `main` function, the second is the code to call that function when requested. These should go at the **end** of the file.

```
1 def main():  # pragma: no cover
2     from reworker.worker import runner
3     runner(MegafrobberWorker)
4
5
6 if __name__ == '__main__':  # pragma: no cover
7     main()
```

Note on line **3** that we pass the name of our class to the `runner` function.

Parse Parameters

Some workers have subcommands which require parameters to run. By default three parameters are always passed to workers: `hosts`, `command`, and `subcommand`. Our worker will not require passing any extra parameters. Therefore, in this tutorial, we will show how to verify that a requested sub-command is valid.

For the cases where input is invalid, we will also demonstrate how to abort the worker.

Note: This is within the `process` method

```
# Begin parameter parsing
#
# It's usually a good idea to record all of your valid
# subcommands somewhere:
self._subcommands = ['frob']

# Grab the REQUESTED subcommand from the 'parameters' dictionary
_subcommand = body['parameters'].get('subcommand', None)

```

```
# Make sure it's recognized
if _subcommand in self._subcommands:
    # This is good, the requested subcommand is valid.
    #
    # ACK the message to make the message bus happy.
    self.ack(basic_deliver)
else:
    # This is bad, the playbook calls for an unknown subcommand
    #
    # Reject the message we received on the message bus
    self.reject(basic_deliver)

    # Output to the console that an error has occurred,
    # include the correlation ID so we can trace the error
    # back to this deployment
    self.app_logger.error(
        "%s - Rejecting message, invalid subcommand requested: %s" % (
            self.corr_id, _subcommand))

    # Use 'notify' to update the output worker of our
    # progress. This output is usually logged to a central
    # location.
    self.notify(
        'Frobbing Failed',
        "Frobbing failed. Invalid subcommand requested: %s" % _subcommand,
        'failed',
        self.corr_id)

    # Send a message to the FSM indicating that the release
    # has failed. This will cause the FSM to stop the
    # deployment.
    self.send(self.reply_to,
              self.corr_id,
              {'status': 'failed',
               "message": "invalid subcommand requested: %s" % _subcommand},
              exchange='')

    # Break out of this job and start over
    return False

# End parameter parsing
```

The `ack`, `notify`, and `send` methods are described in the primary *re-worker* documentation.

Run the Job

At this point we have set up all the usual scaffolding and validated the input parameters for this job. If we haven't aborted by now then we will run the actual `frob` sub-command.

For this tutorial, the `frob` sub-command will just randomly pass or fail. We'll need an additional library for this, `random`, so let's add the import to the top of our file:

```
import random
```

It's a good idea to write each of your sub-commands as a separate method. For the `frob` sub-command it is as simple as returning a random number grabbed from the random number generator:

```python
def _frob(self):
    """
    Frob the random number generator.

    If the result is even then "frob successful". If the result is
    odd, then "frob failed"
    """
    return random.randint(0, 100)
```

And then, back in the `process` method, call this sub-command and process the result:

```python
# Begin the actual job
#
# Let the FSM know we’re starting the job now
self.send(
    self.reply_to, self.corr_id, {’status’: ’started’}, exchange=’’)

self.app_logger.info(’Beginning the frobbing’)

_frob_result = self._frob()

# Process the results
if (_frob_result % 2) == 0:
    _msg = "The frobbing passed, even random number generated: %s" % _frob_result

    self.app_logger.info(_msg)
    self.notify(
        ’Frobbing passed’,
        _msg,
        ’completed’,
        self.corr_id)

    # When a job succeeds, let the FSM know by sending
    # a ’completed’ message
    self.send(self.reply_to,
              self.corr_id,
              {’status’: ’completed’,
               "message": _msg},
              exchange=’’)
    return True
else:
    _msg = ’Frobbing failed, odd random number generated: %s’ % _frob_result

    self.app_logger.error(_msg)
    self.notify(
        ’Frobbing failed’,
        _msg,
        ’failed’,
        self.corr_id)

    # When a job fails, let the FSM know by sending
    # a ’failed’ message
    self.send(self.reply_to,
              self.corr_id,
              {’status’: ’failed’,
               "message": _msg},
              exchange=’’)
    return False
```

Full MegaFrobber Worker Source

```python
#!/usr/bin/env python
import reworker.worker
import logging
import random

class MegafrobberWorker(reworker.worker.Worker):
    """
    Plugin to frob the heck out of something
    """

    def process(self, channel, basic_deliver, properties, body, output):
        # Output is a logger from the python logger library. This is
        # what we report progress through
        self.output = output

        # This is the ID given to the currently happening deployment. It
        # is a unique ID used to connect all passed messages together and
        # record the deployment state in the database.
        #
        # We use it when responding to the FSM.
        self.corr_id = str(properties.correlation_id)

        # If the FSM passed us any dynamic variables, they will be in
        # the 'dynamic' key of the body parameter
        dynamic = body.get('dynamic', {})

        # reply_to is the temporary message bus queue we respond to the
        # FSM through
        self.reply_to = properties.reply_to

        # Begin parameter parsing
        #
        # It's usually a good idea to record all of your valid
        # subcommands somewhere:
        self._subcommands = ['frob']

        # Grab the REQUESTED subcommand from the 'parameters' dictionary
        _subcommand = body['parameters'].get('subcommand', None)

        # Make sure it's recognized
        if _subcommand in self._subcommands:
            # This is good, the requested subcommand is valid.
            #
            # ACK the message to make the message bus happy.
            self.ack(basic_deliver)
        else:
            # This is bad, the playbook calls for an unknown subcommand
            #
            # Reject the message we received on the message bus
            self.reject(basic_deliver)

            # Output to the console that an error has occurred,
            # include the correlation ID so we can trace the error
            # back to this deployment
            self.app_logger.error(
                "%s - Rejecting message, invalid subcommand requested: %s" % (
```

```
                    self.corr_id, _subcommand))

            # Use 'notify' to update the output worker of our
            # progress. This output is usually logged to a central
            # location.
            self.notify(
                'Frobbing Failed',
                "Frobbing failed. Invalid subcommand requested: %s" % _subcommand,
                'failed',
                self.corr_id)

            # Send a message to the FSM indicating that the release
            # has failed. This will cause the FSM to stop the
            # deployment.
            self.send(self.reply_to,
                      self.corr_id,
                      {'status': 'failed',
                       "message": "invalid subcommand requested: %s" % _subcommand},
                      exchange='')

            # Break out of this job and start over
            return False

        # End parameter parsing

        # Begin the actual job
        #
        # Let the FSM know we're starting the job now
        self.send(
            self.reply_to, self.corr_id, {'status': 'started'}, exchange='')

        self.app_logger.info('Beginning the frobbing')

        _frob_result = self._frob()

        # Process the results
        if (_frob_result % 2) == 0:
            _msg = "The frobbing passed, even random number generated: %s" % _frob_result

            self.app_logger.info(_msg)
            self.notify(
                'Frobbing passed',
                _msg,
                'completed',
                self.corr_id)

            # When a job succeeds, let the FSM know by sending
            # a 'completed' message
            self.send(self.reply_to,
                      self.corr_id,
                      {'status': 'completed',
                       "message": _msg},
                      exchange='')
            return True
        else:
            _msg = 'Frobbing failed, odd random number generated: %s' % _frob_result

            self.app_logger.error(_msg)
```

```
            self.notify(
                'Frobbing failed',
                _msg,
                'failed',
                self.corr_id)

            # When a job fails, let the FSM know by sending
            # a 'failed' message
            self.send(self.reply_to,
                      self.corr_id,
                      {'status': 'failed',
                       "message": _msg},
                      exchange='')
            return False

    def _frob(self):
        """
        Frob the random number generator.

        If the result is even then "frob successful". If the result is
        odd, then "frob failed"
        """
        return random.randint(0, 100)


def main():  # pragma: no cover
    from reworker.worker import runner
    runner(MegafrobberWorker)


if __name__ == '__main__':  # pragma: no cover
    main()
```

4.2 Advanced Topics

Hard stuff.

4.2.1 Message Queue Bindings

This section will describe how to configure your message queue bindings so that messages are delivered to the right workers.

FSM - Topics

When the FSM reads a step from a playbook, the destination **topic** is determined by:

- Splitting the execution step name (ex: `juicer::promote`) on the first `::`, and taking the first item (ex: `juicer`)
- This string is then substituted into the string `worker.%s`

Therefore, an execution step of `juicer::promote` would result in the FSM sending messages to the topic `worker.juicer`.

Your message queue must be configured to route messages sent to this topic to somewhere intelligent. Preferably to a queue which matches the same name, i.e.: `worker.juicer`.

Read the next section on how workers select their queue for more information.

Worker - Queues

When a worker using the `re-worker` library first starts, the **default** behavior is to consume from a queue on the message bus whose name matches `worker.CLASS_STR` where `CLASS_STR` is the class name in all lower-case letters. For example, the **juicerworker** worker (from our previous example) would want to consume from the `worker.juicerworker` queue.

Still using the **juicer** worker as reference, if we desired it, this worker could be configured to consume from the `worker.juicer` queue by setting the `queue` parameter in the worker's configuration file to just `juicer`.

4.2.2 Other languages

Nothing is stopping you from writing a worker in any other language of your choice. If you decide to do so, keep a few things in mind:

- Try to follow the `re-worker` reference library as close as possible
- Make sure you ack receipt of the initial message
- The initial message is a dictionary serialized as a JSON string, you'll need to deserialize it
- Talk to the FSM on the temporary queue provided in the `reply_to` property
- Make sure you notify the FSM upon initial failure or start, and final failure or completion

CHAPTER 5

Development

5.1 Build States

Component	State
re-core re-rest re-client re-worker re-worker-bigip re-worker-emailnotify re-worker-func re-worker-ircnotify re-worker-juicer re-worker-output re-worker-sleep	n/a

Note:

More Inception Projects You can find the rest of the Red Hat Inception Team projects on github: GitHub: Red Hat Inception.

5.2 Contributing

This section describes the guidelines for contributing to the Release Engine.

5.2.1 General Guidelines

Please conform to **PEP 0008** for code formatting. This specification outlines the highlights that we require above and beyond. Your code must follow this (or note why it can't) before patches will be accepted. There is one consistent exception to this rule:

E501 Line too long

The `pep8` tests for the Release Engine include a `--ignore` option to specifically exclude **E501** from the tests.

Write unittests for any new functionality, *if you are up to the task*. Not a requirement, but it does get you a lot of karma.

Write intelligent commit messages.

5.2.2 Issue Reporting

If you are reporting an issue with the Release Engine, please use the following template when describing your issue:

```
Description of the issue (include full error messages):

How to reproduce the issue:

How reproducable (every time? intermittently?):

Version of the product effected (git hashes are OK):

Your operating system release-version:

What you expected to happen:

What actually happened:
```

5.3 Testing

All Release Engine code includes unit tests to verify expected functionality. In the rest of this section we'll learn how the unit tests are put together and how to interact with them.

5.3.1 Components

Release Engine unit tests are integrated with/depend on the following items:

- Travis CI - Free online service providing *continuous integration* functionality for open source projects. Tests are ran automatically on every git commit.
- unittest - Python unit testing framework. All Release Engine tests are written using this framework.
- nose - Per the **nose** website: "*extends unittest to make testing easier*". **nose** is used to run our unit tests.

- coverage - A tool for measuring code coverage of Python programs. For the Release Engine we require a minimum test coverage of **80%**. This is invoked by **nose** automatically.
- mock - A library for testing in Python. It allows you to replace parts of your *system under test* with mock objects (such as fake REST endpoints) and make assertions about how they have been used.
- pep8 - A tool to check Python code against some of the style conventions in **PEP 0008**.
- pyflakes - A simple program which checks Python source files for errors.
- virtualenv - A tool to create isolated Python environments. Allows us to install additonal package dependencies without requiring access to the system site-packages directory.
- Makefiles - Utility scripts used for project building and testing. How Release Engine uses **Makefiles** is described later in this section.

5.3.2 Requirements

- `python-nose`
- `python-coverage`
- `python-mock`

Some components may have additional test requirements. For example, *re-worker-func* requires `pyOpenSSL`, which requires `openssl-devel`, `openssl-libs`, and `libffi-devel`. Additionally, *re-rest* requires `openldap-devel` to run its unit tests.

Todo

Document other test dependencies

5.3.3 Targets

In the scope of this document and testing, we use the term *target* in the context of *makefile targets*. For the purpose of this documentation, we can think of these *targets* as pre-defined commands coded in a makefile. Release Engine testing targets include:

- `ci` - Run the tests exactly how they are ran in Travis-CI
- `pep8` - Run **PEP 0008** syntax checks
- `pyflakes` - Run *pyflakes* error checks
- `clean` - Remove temporary files and build artifacts from the checked-out repository.
- `tests` - A quicker version of `ci`. Different from `ci` in that `tests` uses libraries installed on the local development workstation. `tests` runs the **unittests**, **pep8** tests, and **pyflakes** tests automatically.

To ensure the highest degree of confidence in test results you should always use the `ci` target.

When Travis-CI runs an integration test, it calls the `ci` target.

5.3.4 Running the Tests

The Release Engine test suite is invoked via the Makefile. The following is an example of how to run the `ci` test target manually on the *re-core* component.

```
[~/re-core]$ make ci

#############################################
# Creating a virtualenv
#############################################
virtualenv re-coreenv
New python executable in re-coreenv/bin/python
Installing Setuptools..............................................done.
Installing Pip.....................................................done.
. re-coreenv/bin/activate && pip install -r requirements.txt
Downloading/unpacking pika>=0.9.12 (from -r requirements.txt (line 1))

... snip ...

Successfully installed pep8 nose coverage mock
Cleaning up...
#############################################
# Listing all pip deps
#############################################
. re-coreenv/bin/activate && pip freeze
coverage==3.7.1
mock==1.0.1
nose==1.3.3
pep8==1.5.7
pika==0.9.13
pymongo==2.7.1
wsgiref==0.1.2
#############################################
# Running PEP8 Compliance Tests in virtualenv
#############################################
. re-coreenv/bin/activate && pep8 --ignore=E501,E121,E124 src/recore/
#############################################
# Running Unit Tests in virtualenv
#############################################
. re-coreenv/bin/activate && nosetests -v --with-cover --cover-min-percentage=80 --cover-package=reco
Verify using init_amqp provides us with a connection ... ok
Loggers are created with appropriate handlers associated ... ok

... snip ...

Verify create_json_str produces proper json ... ok
Verify load_json_str produces proper structures ... ok
Verify config parsing works as expected. ... ok

Name                  Stmts   Miss  Cover   Missing
-------------------------------------------------
recore                   36      0   100%
recore.amqp              72      4    94%   79, 169-172
recore.constants          1      0   100%
recore.fsm              179     25    86%   97-103, 148-152, 199-249
recore.job                0      0   100%
recore.job.create        25      0   100%
recore.mongo             62      5    92%   92-100
recore.utils             13      0   100%
-------------------------------------------------
TOTAL                   388     34    91%
----------------------------------------------------------------------
Ran 35 tests in 0.047s
```

```
59
60 OK
61 :
```

On line **1** we see how to call a makefile target. In this case it's quite straightforward: `make ci`. Other targets are called in the same way. For example, to run the `clean` target, you run the command `make clean`.

On line **29** we see a header printed, *Running PEP8 Compliance Tests in virtualenv*. By calling the `ci` target, **make** automatically knows what other targets must be called as well, such as `ci-pep8` and `ci-unittests` (seen on line **33**).

5.3.5 Troubleshooting

If you find yourself unable to run the unit tests:

1. Search for relevant error messages
2. **Read** the error message closely. The solution could be hidden in the error message output. The problem could be as simple as a missing dependency
3. If you are unable to figure out all the necessary dependencies to run the tests, file an issue on that specific projects GitHub issue tracker. Include the full error message.

CHAPTER 6

Message Formats

See also:

GitHub: RHInception/re-common The **re-common** repository holds JSON Schema files which can be used for message validation.

6.1 re-rest re-core

re-rest produces two formats: *Deployment Message Format* and *Notification Message Format*. It also receives on message format in return: *Deployment Response Message Format*.

6.1.1 Deployment Message Format

The Deployment Message format comes in two flavors, Simple and Dynamic.

Simple Deployment Message Format

The simple format has two required keys: `group` and `playbook_id`.

Required Keys:

- group: the group's name as a string
- playbook_id: the specific playbook in the group to utilize

Example:

```
{
   "group": "My Group",
   "playbook_id": "1234567890"
}
```

Dynamic Deployment Message Format

The dynamic format adds a key to the simple format: dynamic.

Note: For more information on dynamic variables see *Dynamic Variables*.

Required Keys:

- group: the group's name as a string
- playbook_id: the specific playbook in the group to utilize
- dynamic: a JSON object holding key/values for specific workers

Example:

```
{
   "group": "My Group",
   "playbook_id": "1234567890",
   "dynamic": {
       "cart": "my cart",
       "environment": "QA"
   }
}
```

Deployment Response Message Format

This format is sent from re-core back to re-rest which tells re-rest if the deployment was accepted or not. The message has one key: id. If the deployment was able to start the id will have a deployment id in it. If there was an issue starting a deployment, for example, because the group or playbook doesn't exist, the id will be null.

Required Keys:

- id: the deployment id or null

Example of a Deployment Successfully Starting:

```
{
   "id": "9999999999999999"
}
```

Example of a Deployment Failing to Start:

```
{
  "id": null
}
```

6.2 re-core re-workers

While executing releases, the **re-core** component emits messages in one of three formats. The format selected is determined by the type of *execution step* being ran.

Simple For execution steps which *require no arguments*

Argument For execution steps which require *defined*

Dynamic For execution steps which require *dynamic* arguments

In the rest of this section we will find that these three formats are very similar in structure. Each section will highlight the elements which make the respective format unique from the others.

6.2.1 Deployment Message Formats

This section describes the message formats emitted by **re-core** to workers.

Simple Message Format

On line **8** in the following playbook we can see there is only one step to execute, `bigip:OutOfRotation`. From how the step is written, as a simple string, we know that this step requires no arguments. That is to say, we do not need to define any parameters in the playbook or provide any dynamic data when starting the release.

```
1  {
2      "name": "docs test",
3      "group": "test",
4      "execution": [
5          {
6              "description": "sequence 0",
7              "hosts": [ "host01.example.com" ],
8              "steps": [ "bigip:OutOfRotation" ]
9          }
10     ]
11 }
```

For an execution step like this (line **8**), **re-core** selects the simple message format. The following example shows the format of the message which it would emit to the bus.

```
{
    "parameters": {
        "hosts": [ "host01.example.com" ],
        "command": "bigip",
        "subcommand": "OutOfRotation"
    },
    "group": "test",
    "dynamic": {},
    "notify": {}
}
```

This is the simplest message format used by **re-core** when interacting with simple workers. As such, we can see that it's quite terse. The *Argument* and *Dynamic* message formats use this same structure, however they fill in different items.

On line **2** we have the `parameters` item. This holds the basic information required for this worker to complete its job:

`hosts` An array of hosts to apply the step to

`command` The name of the worker being utilized

`subcommand` The specific action the worker should take

Additionally the `parameters` item has two sibblings: `group` and `dynamic`. These items are always sent to the worker, even if (as in this example), there is no dynamic data to send.

Argument Message Format

On line **10** in the following playbook we can see there is only one step to execute, `service:Restart`. From how the step is written, as a dictionary, we know that this step requires one argument, `service` which is defined as `megafrobber` (line **11**).

```
1  {
2      "name": "docs test",
3      "group": "test",
4      "execution": [
5          {
6              "description": "sequence 0",
7              "hosts": [ "host01.example.com" ],
8              "steps": [
9                  {
10                     "service:Restart": {
11                         "service": "megafrobber"
12                     }
13                 }
14             ]
15         }
16     ]
17 }
```

For an execution step like this (line **10**), **re-core** selects the *argument* message format. The following example shows the format of the message which it would emit to the bus.

```
1  {
2      "parameters": {
3          "service": "megafrobber",
4          "hosts": [
5              "host01.example.com"
6          ],
7          "subcommand": "Restart",
8          "command": "service"
9      },
10     "group": "test",
11     "dynamic": {},
12     "notify": {}
13 }
```

What makes this message format different from the previous format is the presence of an additional key in the `parameters` item. That key is `service` (line **3**). This comes directly from line **11** in the example playbook.

Dynamic Message Format

Still referencing the previous playbook (*Argument Message Format*), let's add an execution step which requires dynamic arguments (this example only shows the additional step).

```
1 {
2     "juicer:Promote": {
3         "dynamic": [
4             "cart",
5             "environment"
6         ]
7     }
8 }
```

See also:

RE-WORKER-JUICER Documentation for the **re-worker-juicer** worker

On line **2** we see that the execution step is called `juicer:Promote`. On the following line we see the dictionary key `dynamic`, and that it's value is a `list` type. The items in the list (lines **4** → **5**) indicate the required **dynamic** variables to run the step. This step requires two such variables, `cart` and `environment`. The user would supply the values for these variables when starting the release.

Note: For more information on dynamic variables see *Dynamic Variables*.

The following example shows the format of the message which **re-core** would emit to the bus.

```
1  {
2      "parameters": {
3          "command": "juicer",
4          "dynamic": [
5              "cart",
6              "environment"
7          ],
8          "subcommand": "Promote",
9          "hosts": [
10             "host01.example.com"
11         ]
12     },
13     "group": "test",
14     "dynamic": {
15         "cart": "bitmath",
16         "environment": "re"
17     },
18     "notify": {}
19 }
```

Here we see a familiar key appearing in the `parameters` item, `dynamic`.

Warning: In future releases, the `dynamic` key will **not** be copied to workers in the `parameters` item. It will only appear as a sibbling of the `parameters` item.

Now, different from the previous format (*argument*), we see the `dynamic` item (sibbling to `parameters`) contains actual key-values (lines **14** → **16**).

`dynamic` A dictionary with the required dynamic variables for a worker to run. The type of each argument is dictated by the respective worker.

Per-Step Notifications

Auxiliary to the formats we've just discussed, are per-step *notifications*. *Per-step notifications* are an **optional** item which may be added to any given step (*simple*, *argument*, and *dynamic*). Using the *previous example playbook* for reference, we would see notifications defined as in lines **6** → **8**, below:

```
1  {
2      "steps": [
3          {
4              "service:Restart": {
5                  "service": "megafrobber",
6                  "notify": {
7                      "started": {
8                          "irc": [ "PHB", "#teamchannel" ]
9                      }
10                 }
11             }
12         }
13     ]
14 }
```

The corresponding message sent to workers, with the additional `notify` item would look like lines **12** → **14** in the following example.

```
1  {
2      "parameters": {
3          "service": "megafrobber",
4          "hosts": [
5              "host01.example.com"
6          ],
7          "subcommand": "Restart",
8          "command": "service"
9      },
10     "group": "test",
11     "dynamic": {},
12     "notify": {
13         "started": {
14             "irc": [ "PHB", "#teamchannel" ]
15         }
16     }
17 }
```

See also:

Playbooks - Notify The documentation for *notify* elements in playbooks. That section defines the allowed syntax for the *notify* item.

6.2.2 Response Message Formats

Complimenting the *Deployment Message Formats* are the *Response Message Formats*. There are three status messages which workers may reply to **re-core** with. This section describes the messages which *workers* send to the *re-core* component.

General Syntax

Status messages are defined as:

- **Type:** `dict`
- **Required Keys:** *status*
  - **Type:** `string`
  - **Allowed Values:**
    - **started**
    - **completed**
    - **failed**
- **Optional Keys:** *data*
  - **Type:** Any JSON Serializable datastructure

Job Started

After a worker has received a message from **re-core**, the message payload is inspected for correctness. If the message payload is successfully verified then the worker will reply to **re-core** with a status update message indicating the job has been started:

```
{
    "status": "started"
}
```

Job Completed

Once a worker has completed the job it was given(without errors), the worker will reply to **re-core** with a status update message indicating success:

```
{
    "status": "completed"
}
```

Optionally a worker may reply to **re-core** with an additional item, `data`. The value of the `data` key may be of any type.

Example

```
1 {
2     "status": "completed",
3     "data": {
4         "items_frobbed": 1337,
5         "avg_time_to_frob_ms": 100
6     }
7 }
```

On line **3** we see the `data` key being defined in the response message. On lines **4** and **5** we see two additional items being reported: `items_frobbed` (the number of items which were frobnicated) and `avg_time_to_frob_ms`, the average amount of time (in miliseconds) it took to *frob* each item.

Important: The `data` item is not currently used by any Release Engine component

Remember that in the previous example, `items_frobbed` and `avg_time_to_frob_ms` are just made-up examples. In reality, workers should use the *notification system* for communicating such information.

Job Failed

If for some reason a worker cannot start a job (for example, due to insufficient or incorrect parameters), or if there is an error while executing the job, then the worker will reply to **re-core** with a status update message:

```
{
    "status": "failed"
}
```

6.3 Notification Message Format

Notifications are sent out by components of the Release Engine and follow a standard message format. This format is then consumed by notification workers who turn the standard format into an external notification of some kind (like email).

The Notification Message Format has 4 required keys: slug, message, phase and target.

Required Keys:

- slug: A "short" message (up to 80 characters).
- message: The message of the notification.
- phase: The phase that the notification occured within: "started", "completed", "failed"
- target: A list denoting where the notification should go. This may be irc nicknames, email address, etc.. and is different for different workers.

Note: Even though slug and message are required it does not mean the notification worker will use them both. Some notification workers will only use one or the other due to space constraints. However, if either key is missing the notification will be rejected as malformed and/or cause problems!

Example:

```
{
    "slug": "RPM's have been promoted",
    "message": "The rpms in deployment 12345667890 have been promoted from DEV to QA and are ready fo
    "phase": "completed",
    "target": ["someone@example.com"]
}
```

6.4 Output Message Format

Notifications are sent out by workers connected to the Release Engine and follow a standard and very simple message format. This format is then consumed by the output worker who turns the standard format into messages in a file.

The Output Message Format has 1 key: message. The only other bit of information needed by an output worker is the correlation id which happens to be stored in the AMQP properties.

Required Keys:

- message: The message which should be written to a file.

Example:

```
{
    "message": "Something happened and you should know about it"
}
```

CHAPTER 7

Playbooks

Playbooks are documents which describe the exact set of steps required to successfully start and finish a given software release. When the Release Engine begins *a deployment*, the actions it takes come directly from playbooks.

A playbook might be ran automatically each time a code builder finishes so that it may deploy the latest snapshots. Alternatively, if more control is desired over the release process, playbooks may be ran by hand. Playbooks may be written in YAML syntax, or optionally in *JSON Syntax*.

In this section we'll learn:

- what a playbook looks like (by reviewing a simple example)
- the major items of playbooks
- the basics of how to describe `execution` steps in your playbooks, including:
- describing a release step
- identifying which worker to use
- passing data to the worker

7.1 Example Playbook

Here is an example of what a **super simple** playbook looks like. The playbook is *owned* by the group called **inception**.

When ran, all this would do is restart the `httpd` service on `foo.bar.example.com`

```
1 ---
2 group: inception
3 name: Simple playbook
4 execution:
5   - description: restart httpd
6     hosts:
7       - foo.bar.example.com
8     steps:
9       - service:Restart: {service: httpd}
```

7.2 Playbook Components

A Release Engine playbook is made up of the following **required** items:

group A short string (acronyms are best) defining the ownership of this playbook. Think of it like the unix group a team might all be members of.

The `group` in our example is **inception**

execution A list of playbook *execution sequences*. These execution sequences are composed of release steps and are accompanied by supporting meta-data. These sequences are explained fully in *Execution Sequences*.

In our example, we have one execution sequence with one release step (`service:Restart`).

Additionally, a playbook may define the optional item:

name A short description of what this playbook accomplishes overall.

In our example the name is **Simple playbook**.

7.3 Execution Sequences

Recall that `execution` items hold a `list` type. Each item in the list is an *execution sequence*. This section describes exactly what execution sequences do, and how to write our own.

In our example, *simple playbook*, the execution sequence is defined in lines **5** → **9**. Let's review those lines again:

```
1 ---
2 group: inception
3 name: Simple playbook
4 execution:
5   - description: restart httpd
6     hosts:
7       - foo.bar.example.com
8     steps:
9       - service:Restart: {service: httpd}
```

Like playbooks themselves, each execution sequence is comprised of several required and optional elements. In *Sample playbook* we can see several items are already being used: `description`, `hosts`, and `steps`. The following sections will describe these, and all other items which are allowed in execution steps.

7.3.1 Required Items

This section describes the **required** items in execution sequences.

Hosts

- **Required**: Yes
- **Argument** type: `list`
- **Default**: None

The `hosts` element is used to describe the target hosts for the script to act on.

```yaml
---
- hosts:
   - www01.web.ext.example.com
   - www02.web.ext.example.com
   - www03.web.ext.example.com
```

Or in YAML shorthand:

```yaml
---
- hosts: [www01.web.ext.example.com, www02.web.ext.example.come, www03.web.ext.example.com]
```

Steps

- **Required**: Yes
- **Argument** type: `list`
- **Default**: None

The `steps` element defines the steps that will be performed on each host in `hosts`. The syntax of each possible step varies, however they will assume one of three legal forms, respective to the information (if any) which the execution step requires to run. In brief, these forms are described below:

- Some steps require no information at all, as such they are given as strings
- Some require explicit parameters given in the form of keyword arguments
- Similar to the previous form, some steps require an enumeration of their *dynamic* parameters

Steps - Strings

An execution step may be a simple string. To us, this means that this given step requires no additional information. The actual *workers* implementing each step may include possible several sub-commands. Because of this it is common to see step names given in colon delimited notation. In this form the string leading up to but not including the colon (":") character refers to the worker or module. The string after the colon character refers to the specific sub-command to run.

OK. Enough of the boring stuff. Let's see some examples.

Assume there is a module called `logrotate`, this module provides one sub-command: `Rotate`.

The documentation for the `logrotate` module tell us that the `Rotate` sub-command requires no keyword parameters. That is to say, we can define it in our execution steps as a simple string. Recall that steps are denoted using colon notation, where the module name comes first, followed by the sub-command. In this case our module is `logrotate` and our sub-command is `Rotate`. We can see this on line **9** in the following example:

```yaml
1 ---
2 group: inception
3 name: Simple playbook
4 execution:
5   - description: Rotate all the megafrobber logs
```

```
6      hosts:
7        - foo.bar.example.com
8      steps:
9        - logrotate:Rotate
```

But what is happening exactly in this example? On line **9** we see the entry: `- logrotate:Rotate`. Recall that each step is given as an item to the `steps` list. This is why line **9** in the previous code example begins with a hyphen character. In YAML this indicates a list item.

Note closely exactly how we gave `logrotate:Rotate`, because in the next example this will change very slightly.

Steps - Keyword Arguments

Now let us assume there is a module called `service` which can control system services. The documentation for this module tells us that there are three sub-commands provided: `Start`, `Stop`, and `Restart`. Additionally, the documentation tells us that each sub-command requires one keyword parameter: `service`. On lines **9** and **10** in the following example, we see how to provide keyword arguments to steps:

```
1  ---
2  group: inception
3  name: Simple playbook
4  execution:
5    - description: Restart the megafrobber service
6      hosts:
7        - foo.bar.example.com
8      steps:
9        - service:Restart:
10           service: megafrobber
```

Or in YAML shorthand (only focusing on the step definition)

```
---
# ...
      - service:Restart: {service: megafrobber}
```

Let's look closer at this and see exactly what is happening.

Recall that playbooks are **YAML Documents**. As such, there are defined ways to describe different datastructures. Review the *dictionary* section in *YAML Scripts* if you need a refresher.

The `service:Restart` sub-command requires one parameter, `service`. You describe parameters in execution steps by using a hash, or dictionary. For our example, a dictionary describing a keyword `service` with value `megafrobber` would look like the following example in YAML:

```
---
service: megafrobber
```

Additionally, recall that you can nest datastructure in YAML. If we wanted to represent a list of dictionaries, we could do that in the following way. Here's an example of a list of nested dictionaries:

```
---
- thingies:
    service: megafrobber
- stuffs:
    penguins: cute
```

Or in alternative representation:

```
---
[{thingies: {service: megafrobber}}, {stuffs: {penguins: cute}}]
```

Now that we know all of this, to give the required parameters to our step we will define the step as a **dictionary key** with a nested-dictionary describing our parameters. This is shown on lines **8** and **9** in the following example:

```
1 ---
2 # ...
3 execution:
4   - description: Restart the megafrobber service
5     hosts:
6       - foo.bar.example.com
7     steps:
8       - service:Restart:
9           service: megafrobber
```

Important:

Note the syntax change In the previous example we only gave the string: `logrotate:Rotate`. Now, instead of a string we're describing a **dictionary key**.

Therefore, the text for this step begins with a hyphen character (to indicate a list item) and ends with a colon character.

Finally, on line **4** you see the provided parameters.

If there were a module which required more than one parameter, the syntax is very similar. Lines **4** → **6** show this in the following example:

```
1 ---
2 # ...
3       - service:Restart:
4           service: megafrobber
5           foo: bar
6           noop: true
```

Steps - Dynamic Arguments

This section is about *dynamic arguments*. Dynamic arguments differ from normal arguments in that their values are not stored in playbooks. Rather, within a playbook, we assert that their values will be provided by the calling client when starting a deployment. The syntax for defining dynamic arguments differs only slightly from how keyword arguments are defined.

The scope of this section is limited to the role of dynamic arguments in **playbooks** only. That is to say, discussion of dynamic arguments in *worker development* will not be covered here. Instead, see the *re-worker documentation* for that information.

> **Caution:** The ability for clients to provide a broad spectrum of dynamic data is both a pro and a con.
> If you're writing a new *worker*, think very carefully before making arguments dynamic. Consider if the there is a *non-interactive* way for the information to be obtained instead.

Use Cases Situations where dynamic arguments may be required are generally limited to actions which require data that changes every, or nearly every, release. Examples might include:

- Change Record IDs
- User Story IDs

  - *or any other agile/scrum related work item*
- A target environment

For a more complete example, imagine our workplace has strict policies around software releases. These policies state that any software release must have an associated change record with it. Additionally the policy states that every time a release happens for a change, an update to the change record document must be recorded. This update must indicate the date of the release.

In this situation, the pragmatic approach to automating this task would be to develop a worker which can interface with the change management system and add updates to the change record over an API. Let's pretend such a worker already exists.

It is clear that we cannot hard-code change record numbers into the worker. And storing this information in the playbook would require a manual update to the playbook every time a new change is created. Furthermore, this limitation effectively nullifies the ability of a playbook to be used for two changes happening in an overlapping time span.

This is an excellent opportunity to use dynamic variables.

The following examples will be using a fictional worker called *change*. This worker has one usable function: `Update`. The **change** worker documentation provides the following API signature for the `change:Update` function:

- **Function:** `change:Update`
  - **Arguments:**
    - None
  - **Dynamic Arguments:**
    - **Name:** id
      - **Required:** `True`
      - **Type:** `string` or `int`
      - **Description:** The ID of the change record to update

For the rest of this section, let's pretend our `id` is **CHG1337**.

Dynamic Argument Syntax Let's begin by considering our example *simple playbook* again.

```
---
group: inception
name: Simple playbook
execution:
  - description: restart httpd
    hosts:
      - foo.bar.example.com
    steps:
      - service:Restart: {service: httpd}
```

However, instead of just restarting the *httpd* service, we have to have an additional step: updating the change record (`CHG1337`). In this section we will learn how add that step.

The general syntax for defining a step with dynamic arguments is shown on lines **4** → **9** in the following example:

```
1 ---
2 # ...
3     steps:
4       - worker:Function:
```

```
5         dynamic:
6            - arg_name_0
7            - arg_name_1
8            - arg_name_2
```

Line 4 As before, we begin by providing the **worker.Function** name, ending with a `:` character

Line 5 We define a dictionary key called `dynamic`. **Again** note, this must end with a `:` character.

Lines 6 → 9 We define the value of the `dynamic` key. This value **must** be a list.

The items in the list are `arg_name_0`, `arg_name_1`, and `arg_name_1`. Each of these is the name of a dynamic variable required by **worker:Function**.

Applying this example to our fictional situation will yield the following playbook:

```
1  ---
2  group: inception
3  name: Simple playbook
4  execution:
5    - description: restart httpd
6      hosts:
7        - foo.bar.example.com
8      steps:
9        - change:Update:
10           dynamic:
11             - id
12       - service:Restart: {service: httpd}
```

Line 9 Insert the new sequence step, `change:Update`

Line 10 Begin the `dynamic` argument dictionary

Line 11 Define the dynamic argument list with one item: `id`

Providing Values Now that we have learned how to add a sequence step that requires dynamic arguments to a playbook, it might be helpful to quickly review how clients can provide the information.

A commonly used command line tool for interacting with REST endpoints (such as *RE-REST*) is `curl`. Put simply, curl allows you to make a request to anything `http`. This is exactly like following a hyperlink in a web page.

The following is an example of how to use the curl command to provide the value of the dynamic argument, `id`, to the release engine and start a deployment.

```
1  $ curl -u "user:passwd" -H "Content-Type: application/json" \
2  -d '{"id": "CHG1337"}' \
3  -X PUT \
4  http://rerest.example.com/api/v0/test/playbook/12345/deployment/
```

Line **2** utilizes the `-d` (or `--data`) option to provide the value of the dynamic argument. When curl is ran in this manner, dynamic arguments are provided by describing a dictionary including *key-value pairs* where the key is the dynamic argument name, and the value is the unique-value of that argument for this particular deployment. In this example the dictionary is `{"id": "CHG1337"}`.

See also:

RE-REST - Dynamic Variables See the *RE-REST → dynamic variables* documentation for a complete review of this topic.

7.3.2 Optional Items

This section describes the **optional** items which are allowed in execution sequences.

Description

Finally, an execution step **may** define a optional `description` item.

- **Required**: No
- **Argument Type**: `string`
- **Default**: None

The `description` element allows you to provide a useful human-readable description of what the step is exactly supposed to do. Use of `description` items are **encouraged**.

Recall our previous example of using `service.Restart` to restart the `megafrobber` service. Below, line **5**, shows an example of how to use the `description` item.

```
1 ---
2 group: inception
3 name: Simple playbook
4 execution:
5   - description: Restart the megafrobber service
6     hosts:
7       - foo.bar.example.com
8     steps:
9       - service.Restart: {service: megafrobber}
```

Because this item is optional, we could just as well have omitted it entirely:

```
---
group: inception
name: Simple playbook
execution:
  - hosts:
      - foo.bar.example.com
    steps:
      - service.Restart: {service: megafrobber}
```

Notify

- **Required**: No
- **Argument type**: `dict`
- **Default**: None

The `notify` element allows you to set custom notification hooks which trigger at different **phases** of each sequence step. For example, you may desire to receive an email every time an RPM promotion step completes, *or fails*.

Here's an example of a notify step that updates IRC when the step has started:

```
---
# ...
    steps:
      - service:Restart:
          service: httpd
          notify:
```

```
7           started:
8             irc: [ "PHB", "#teamchannel" ]
```

Line 6 Shows the beginning of the `notify` syntax.

Line 7 This indicates which **phase** of worker execution this notification is for.

Recognized **phases** include: `started`, `completed`, and `failed`

Line 8 Define the IRC notification parameters.

For the `irc` item we define a list of users/channels for messages to be sent to.

In this example, the user **PHB** would be notified *directly* with status updates. Additionally, a notification would be sent to the channel **#teamchannel**.

See also:

Components For a list of available notification workers, see the *Components* section.

7.4 Putting it all together

To finish up, let's put together everything we've seen up to now. That will include `hosts`, and some example items for `steps`.

```
---
# Playbook owned by group inception
group: inception

# This playbook is clearly awesome:
name: Simple playbook

# This playbook executes **two** sequences of steps for this
# release:
execution:

  #################################################################
  # Sequence 1
  #################################################################
  # Including a description is optional. This must be a string.
  - description: frobnicate these lil guys
    hosts:
      - foo.dev.example.com
      - bar.ops.example.com

    # Install megafrobber on all our hosts ahead of time
    preflight:
      - yumcmd:install:
          package: "megafrobber"

    steps:
      # Some steps don't require parameters:
      - bigip:OutOfRotation

      # Whereas, some require parameters:
      - misc:Echo:
          input: "This is a test message"

      # And some times you just want to tell the world what you're doing
```

```
      - frob:Nicate:
          things: "all the things"
          notify:
              started:
                  irc: [ "PHB", "#myteam" ]

##################################################################
# Sequence 2
##################################################################
- description: then frobnicate the other half
  hosts:
    - dev.foo.example.com
    - ops.bar.example.com

  steps:
    - bigip:OutOfRotation

    # Some may even accept lists as the value of their parameters
    - misc:ListFrob:
        frob_list:
          - item1
          - item2
          - item3
```

Todo

Describe interesting parts of the previous example

CHAPTER 8

Worker Steps

This section documents the *Playbook* syntax for all of the workers included with the Release Engine. What follows includes formal *signatures* of each step, as well as examples of each step in playbooks.

8.1 Juicer

8.1.1 juicer:promote

Promote a release cart to a specified environment. It is recommended that this command is used in an execution sequence only a single dummy host listed in the `hosts` array. This will prevent the step from being ran multiple times.

Parameters

- `dynamic` (type: list)
  - **Required:** True
  - **Items:** The strings `environment` and `cart`

Example

```
hosts: ['localhost']
steps:
    - juicer:promote:
        dynamic:
            - environment
            - cart
```

Note: Recall that playbooks which have steps including *dynamic parameters* require the values for those parameters to be passed when starting the deployment.

8.2 BigIP

8.2.1 bigip:InRotation

Enable the current host in the BigIP.

Example

```
hosts: ["example01.com", "example02.com"]
steps:
    - bigip:InRotation
```

8.2.2 bigip:OutOfRotation

Disable the current host in the BigIP.

Example

```
hosts: ["example01.com", "example02.com"]
steps:
    - bigip:OutOfRotation
```

8.2.3 bigip:ConfigSync

Sync the BigIP configuration from a primary to a secondary.

Parameters

- `envs` (type: list)
  - **Required:** True
  - **Items:** Strings of environment names

Example

```
hosts: ["example01.com", "example02.com"]
steps:
    - bigip:ConfigSync:
        envs: ["qa", "stage", "prod"]
```

8.3 FUNC

The FUNC worker has several commands (with their own subcommands) available. They are all documented on this page.

8.3.1 Puppet

The **puppet** module allows you to interact with the `puppet` service on remote hosts.

puppet:Run

Parameters

The parameters to the `Run` subcommand can be mixed and matched together. That is to say, *none of the parameters given below are mutually exclusive*.

- `noop` (type: boolean)
  - **Required:** False
  - **Default:** `False`
  - **Description:** Set to `True` to enable `noop` mode (show what *would* have happened)
  - **CLI Equivalent:** `puppet agent --test --noop`
- `enable` (type: boolean)
  - **Required:** False
  - **Default:** `False`
  - **Description:** Set to `True` to enable the puppet agent prior to running puppet. **Note** that running puppet will not be attempted if the **enable** command fails
  - **CLI Equivalent:** `puppet agent --enable && puppet agent --test`
- `tags` (type: list of strings)
  - **Required:** False
  - **Default:** `None`
  - **Description:**
  - **CLI Equivalent:** `puppet agent --test --tags sometag anothertag moretags`

Example

```
hosts: ['localhost']
steps:
    # Basic subcommands
    - puppet:Run
    - puppet:Enable
    - puppet:Disable

    # Now with some extra options

    # Run puppet in noop mode
    - puppet:Run:
        noop: True

    # Run puppet in noop mode, and make sure the agent is enabled first
    - puppet:Enable
    - puppet:Run:
        noop: True

    # Run puppet in noop mode, and make sure the agent is enabled
    # first, but as one single step
    - puppet:Run:
        noop: True
        enable: True

    # Equivalent to 'puppet agent --test --tags yum auth package'
    - puppet:Run:
        tags:
          - yum
          - auth
          - package
```

puppet:Enable

Parameters

- The `Enable` subcommand does not accept any parameters

```
hosts: ['localhost']
steps:
    - puppet:Enable
```

puppet:Disable

Parameters

- `motd` (type: string or `False`)
  - **Required:** False
  - **Default:** `puppet disabled by Release Engine at 2014-09-16 16:27:11.075617`
  - **Description:** The `puppet:Disable` sub-command will automatically append a message to `/etc/motd` indicating that the puppet agent has been stopped. This behavior can be disabled by setting `motd` to `False`, or customized by setting `motd` to a message of your choice. Use `%s` to substitute a datestring (per `str(datetime.datetime.now())`) into your message

```
hosts: ['localhost']
steps:
    # Just disable the puppet agent, motd is still updated
    - puppet:Disable

    # Disable the agent, but don't update the motd
    - puppet:Disable
        motd: False

    # Disable the agent, and put a custom message in /etc/motd
    - puppet:Disable
        motd: "Puppet disabled for maintenance on %s"
```

8.3.2 Command

The **command** module allows you to run arbitrary commands on a remote host. It has one sub-command available, **run**.

command:run

Parameters

- `cmd` (type: string)
  - **Required:** True
  - **Description:** The command to run, as it would be typed into a shell prompt

Example

```
hosts: ['localhost']
steps:
    - command:run:
        cmd: puppet agent --test --color=false
```

8.3.3 Service

The **service** module allows you to interact with system services, as you would with the `service` or `systemctl` commands. Only one example is included in this section because the syntax for each of the **service** module steps are nearly identical.

Example

This example demonstrates how to restart the **megafrobber** service (see lines **3** and **4**).

```
1 hosts: ['localhost']
2 steps:
3     - service:restart:
4         service: megafrobber
```

To use any of the other sub-commands, on line **3** in this example we would replace `service:restart` with the desired subcommand. Such as `service:stop` or `service:reload`.

service:stop

Stop a given service.

service:start

Start a given service.

service:restart

Restart a given service.

service:status

Return the status (running, stopped, etc) of a given service.

service:reload

Tell a service to reload it's configuration files.

Note: Not all system services support all the given subcommands. This is especially true for **reload**.

8.3.4 Yum Cmd

yumcmd:install

Foo

yumcmd:remove

Bar

yumcmd:update

Bob

8.3.5 Nagios

The nagios module allows you to perform common tasks in Nagios related to downtime and notifications.

nagios:ScheduleDowntime

Depending on the exact invocation, `nagios:ScheduleDowntime` will schedule downtime for:

- A host
- Services on a host
- A host and it's services

Parameters

- `nagios_url` (type: string)
  - **Description:** Hostname of the nagios server
  - **Required:** True
  - **Default:** None
- `minutes` (type: int)
  - **Description:** Number of minutes to schedule downtime for
  - **Required:** False
  - **Default:** 30
- `service` (type: string or list)
  - **Description:** Service, or services, to schedule downtime for
  - **Required:** False
  - **Default:** Set downtime for the host itself (services on the host will continue to alert like normal)
  - **Extras:** Use the string `ALL` to schedule downtime for the host as well as all services on the host. Use the string `HOST` to explicitly set downtime for just a host. `HOST` and `ALL` are case-insensitive.
- `service_host` (type: string)
  - **Description:** An alternative host to schedule downtime for
  - **Required:** False
  - **Default:** None
  - **Extras:** See example below for **service host**

Example: Schedule Downtime for a host

In this example we set downtime for a host. Because `minutes` is not provided, the duration will be for the default of 30 minutes.

```
hosts: ['localhost']
steps:
    - nagios:ScheduleDowntime:
        nagios_url: nagios.example.com
        service: host
```

As stated in the parameter documentation above, we can give the string **host** in any mix of upper and lower case characters.

Example: Schedule Downtime for a service

In this example we set downtime for 15 minutes (line **5**) for a specific service (`megafrobber`, line **6**).

```
1 hosts: ['localhost']
2 steps:
3     - nagios:ScheduleDowntime:
4         nagios_url: nagios.example.com
5         minutes: 15
6         service: megafrobber
```

Example: Schedule Downtime for several services

Similar to the previous example, here we are setting downtime for several services at once. Note the difference below in syntax on lines **6** → **8** compared to line **6** above. Here we provide the services as a list to the `service` parameter.

```
1 hosts: ['localhost']
2 steps:
3     - nagios:ScheduleDowntime:
4         nagios_url: nagios.example.com
5         minutes: 15
6         service:
7            - megafrobber
8            - httpd
```

Example: Schedule Downtime for a host and all services on the host

In this example we will set an hour of downtime (**60 minutes**, line **5**) for a host and all services running on that host (line **6**).

```
1 hosts: ['localhost']
2 steps:
3     - nagios:ScheduleDowntime:
4         nagios_url: nagios.example.com
5         minutes: 60
6         service: ALL
```

Example: Using service_host to set downtime for an alternative host

In some deployments, **service hosts** are created in nagios to monitor services not exactly tied to a specific host.

For example, you may be using a vendor load balancing solution, like F5 LTM BigIPs. In a situation like this you may monitor the status of all balancer pools so that you can send alerts if members of the pool drop out of rotation unexpectedly.

However, while performing routine maintenance, is it expected for hosts to be taken out of the rotation. That's what `service_host` is for. Instead of setting downtime for a specific host, we might schedule downtime for a service representing a balancer pool on our **service host**.

```
1 hosts: ['localhost']
2 steps:
3     - nagios:ScheduleDowntime:
4         nagios_url: nagios.example.com
5         minutes: 60
6         service_host: lb01.example.com
7         service: megafrobber_pool_prod
```

In the above example on line **6** we tell the nagios worker that instead of setting downtime for `localhost`, instead, set downtime for `lb01.example.com`. Then on the following line (**7**) we indicate we are setting downtime for the production *megafrobber* balancer pool.

8.4 Sleep

8.4.1 sleep:Seconds

Pause all further playbook step execution for the given number of seconds.

Parameters

- `seconds` (type `int`)
  - **Required:** True
  - **Description:** The number of seconds to pause for

Example

To pause a playbook for 1,337 seconds:

```
hosts: ['localhost']
steps:
    - sleep:seconds:
        seconds: 1337
```

Note: If more than one host is given in `hosts`, the playbook will pause again for each host given.

8.5 ServiceNow

8.5.1 servicenow:DoesChangeRecordExist

Checks to see if a change record exists. Resulting data will have an exists key with a bool.

Dynamic Arguments

- `change_record` (type `str`)
  - **Required:** True
  - **Description:** The change record to look for.

Example

To check if a change record exists:

```
hosts: ['localhost']
steps:
    - servicenow:DoesChangeRecordExist:
        dynamic:
            - change_record
```

Note: This check is ran for each host in `hosts`

Note: This step has no direct side-effects. It is more useful as a *Pre-Deployment Step*

Appendices

This section includes all the appendices for the Release Engine Documentation

9.1 JSON Scripts

This page provides a basic overview of correct JSON syntax. Additionally it covers non-task specific modules that are valid in *Release Engine* playbooks.

See also:

Components → Pre-Built Workers For more information on the workers that ship with Release Engine

For the *Release Engine*, every JSON playbook must be a list at it's root-most element. Each item in the list is a dictionary. These dictionaries represent all the options you can use to write a *Release Engine* playbooks.

Tip: With the exception of **strings** all types (arrays, booleans, integers, numbers, nulls and objects) in a JSON list or dictionary are not required to be surrounded by double quotes (`"Foo"`). If added, these types become **strings**. Also, all lines in a list or dictionary must end in a comma `,` **except** the final member in the list or dictionary, which must explicitly **not** end with a comma.

In JSON a list can be represented in two ways. In one way all members of a list are lines beginning at the same indentation level surrounded by square brackets.

```
[
 "Apple",
 "Orange",
 "Strawberry",
 "Mango"
]
```

In the second way a list is represented as comma separated elements surrounded by square brackets. Newlines are permitted between elements:

```
["apple", "orange", "strawberry", "mango"]
```

A dictionary is represented in a simple `key:` and `value` form:

```
{
  "skill": "Elite",
  "job": "Developer",
  "name": "John Eckersberg"
}
```

Like lists, dictionaries can be represented in an abbreviated form:

```
{"skill": "Elite", "job": "Developer", "name": "John Eckersberg"}
```

You can specify a boolean value (true/false) in several forms:

```
{
  "knows_oop": true,
  "likes_emacs": true,
  "uses_cvs": false
}
```

Finally, you can combine these data structures:

```
{
 "name": "John Eckersberg",
 "python": "Elite",
 "job": "Developer",
 "languages": {
   "ruby": "Elite"
 },
 "foods": [
  "Apple",
  "Orange",
  "Strawberry",
  "Mango"
 ],
 "dotnet": "Lame",
 "employed": true,
 "skill": "Elite"
}
```

That's all you really need to know about JSON to get started writing *Release Engine* playbooks.

See also:

JSONLint JSON Lint gets the lint out of your JSON

See also:

Get Deeper into Playbooks

Now that we're comfortable with JSON, let's continue on and read the *Playbooks* section for an in-depth guide of playbooks.

9.2 YAML Scripts

This page provides a basic overview of correct YAML syntax. Additionally it covers non-task specific modules that are valid in *Release Engine* playbooks.

See also:

Components → Pre-Built Workers For more information on the workers that ship with Release Engine

For the *Release Engine*, every YAML playbook must be a list at it's root-most element. Each item in the list is a dictionary. These dictionaries represent all the options you can use to write a *Release Engine* playbook. In addition, all YAML files (regardless of their association with *Release Engine* or not) all YAML documents should start with `---`.

In YAML a list can be represented in two ways. In one way all members of a list are lines beginning at the same indentation level starting with a - character

```yaml
---
# A list of tasty fruits
- Apple
- Orange
- Strawberry
- Mango
```

In the second way a list is represented as comma separated elements surrounded by square brackets. Newlines are permitted between elements

```yaml
---
# A list of tasty fruits
[apple, orange, banana, mango]
```

A dictionary is represented in a simple `key:` and `value` form

```yaml
---
# An employee record
name: John Eckersberg
job: Developer
skill: Elite
```

Like lists, dictionaries can be represented in an abbreviated form

```yaml
---
# An employee record
{name: John Eckersberg, job: Developer, skill: Elite}
```

You can specify a boolean value (true/false) in several forms

```yaml
---
knows_oop: True
likes_emacs: TRUE
uses_cvs: false
```

Finally, you can combine these data structures

```yaml
---
# An employee record
name: John Eckersberg
job: Developer
skill: Elite
employed: True
foods:
    - Apple
    - Orange
    - Strawberry
    - Mango
languages:
    ruby: Elite
python: Elite
dotnet: Lame
```

That's all you really need to know about YAML to get started writing *Release Engine* playbooks.

See also:

YAMLLint YAML Lint gets the lint out of your YAML

See also:

Get Deeper into Playbooks

Now that we're comfortable with YAML, let's continue on and read the *Playbooks* section for an in-depth guide of playbooks.

9.3 Definitions

AMQP See **Message Bus**.

Finite State Machine (FSM) See **RE-CORE**

JSON Javascript Object Notation. Data which can be turned into code. Usually returned from REST APIs. It's also called a *data interchange standard*.

Message Bus Very similar to how the postal service works, but in software. Clients connect to the bus and consume or publish messages. It's like IPC, over the network, with queues, on steroids.

MongoDB A "schemaless" document object collection.

Basically MySQL without all the rigidly defined table structures.

Playbook A document describing a software release. This document is stored in MongoDB. Playbooks consists of three main items: ownership identification, target hosts, and a list of steps (See also: **Playbook Step**) required to finish so that the release can be considered completed.

Playbook Step A playbook step represents a unit of work in your overall release process.

Defining a playbook step is like instantiating a **Worker Plugin**. That is to say, using the api signature of a given Worker Plugin, you fill in the missing parameters.

Python The programming language the Release Engine is primarily written in.

RE-CLIENT The `re-client` tool is how end-users primarily interact with the release engine. The `re-client` tool interfaces with the **RE-REST** component and provides several options for creating, reading, updating, and deleting playbooks.

RE-CORE The ring-leader of the system. Orchestrates the delegation of **playbook steps** to **worker plugins**. Tracks the state of a release in mongo and manupilates the completed/active/remaining job stacks as workers update the FSM.

RE-REST A REST endpoint (see below) which all clients attempting to interact with the Release Engine must proxy their commands and requests through. This component in integrated into the overall authentication/authorization scheme.

Authorized requests made against the REST endpoint result in either: messages having been sent to the RE-CORE component (for example: begin a release for group **foo**), or in database create/relad/update/delete operations.

REST Representational State Transfer. Using the HTTP protocol in a programmatic way to interact with remote systems. Usually supports the basic CRUD operations: Creating, Reaing, Updating, and Deleting records.

Temporary Queue (See also: **Message Bus**) Temporary queues are created by various Release Engine components. These queues are ephemeral and usually automatically clean themselves up after all clients disconnect from them. The purpose of these temporary queues is to enable **direct communication** between two specific components, outside of the pre-defined channels of communication.

Worker (Plugin) Worker plugins do the actual work in a release. This could mean several things: running puppet on a server; restarting a host; uploading RPMs into a YUM repository, scheduling downtime, the possibilities are virtually endless.

It might help to think of Worker Plugins as Class definitions. See **Playbook Step** for the other half of that comparison.

YAML YAML Ain't Markup Language. YAML is an alternative syntax which may be used to write Playbooks in. The normative syntax is JSON.

See also:

YAML Basics Introduction to YAML formatting

See also:

Playbooks Everything you need to know to begin writing playbooks

CHAPTER 10

AGPLv3 License

The Release Engine uses AGPLv3+ for it's license.

```
                    GNU AFFERO GENERAL PUBLIC LICENSE
                       Version 3, 19 November 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU Affero General Public License is a free, copyleft license for
software and other kinds of works, specifically designed to ensure
cooperation with the community in the case of network server software.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
our General Public Licenses are intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  Developers that use our General Public Licenses protect your rights
with two steps: (1) assert copyright on the software, and (2) offer
you this License which gives you legal permission to copy, distribute
and/or modify the software.

  A secondary benefit of defending all users' freedom is that
improvements made in alternate versions of the program, if they
receive widespread use, become available for other developers to
incorporate.  Many developers of free software are heartened and
encouraged by the resulting cooperation.  However, in the case of
software used on network servers, this result may fail to come about.
The GNU General Public License permits making a modified version and
letting the public access it on a server without ever releasing its
source code to the public.
```

The GNU Affero General Public License is designed specifically to ensure that, in such cases, the modified source code becomes available to the community. It requires the operator of a network server to provide the source code of the modified version running there to the users of that server. Therefore, public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version.

An older license, called the Affero General Public License and published by Affero, was designed to accomplish similar goals. This is a different license, not a version of the Affero GPL, but Affero has released a new version of the Affero GPL which permits relicensing under this license.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU Affero General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

```
  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
```

```
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
```

```
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
```

```
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.
```

```
  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.
```

```
  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
```

```
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
```

```
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Remote Network Interaction; Use with the GNU General Public License.

  Notwithstanding any other provision of this License, if you modify the
Program, your modified version must prominently offer all users
interacting with it remotely through a computer network (if your version
supports such interaction) an opportunity to receive the Corresponding
Source of your version by providing access to the Corresponding Source
from a network server at no charge, through some standard or customary
means of facilitating copying of software.  This Corresponding Source
shall include the Corresponding Source for any work covered by version 3
of the GNU General Public License that is incorporated pursuant to the
following paragraph.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the work with which it is combined will remain governed by version
3 of the GNU General Public License.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU Affero General Public License from time to time.  Such new versions
```

```
will be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU Affero General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU Affero General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU Affero General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs
```

```
  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Affero General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Affero General Public License for more details.

    You should have received a copy of the GNU Affero General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If your software can interact with users remotely through a computer
network, you should also make sure that it provides a way for users to
get its source.  For example, if your program is a web application, its
interface could display a "Source" link that leads users to an archive
of the code.  There are many ways you could offer source, and different
solutions will be better for different programs; see section 13 for the
specific requirements.

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU AGPL, see
<http://www.gnu.org/licenses/>.
```