Adjutant Documentation

Release

Catalyst IT Ltd

Mar 15, 2018

Contents

CHAPTER 1

Development

Adjutant is built around tasks and actions.

Actions are a generic database model which knows what 'type' of action it is. On pulling the actions related to a Task from the database we wrap it into the appropriate class type which handles all the logic associated with that action type.

An Action is both a simple database representation of itself, and a more complex in memory class that handles all the logic around it.

Each action class has the functions "pre_approve", "post_approve", and "submit". These relate to stages of the approval process, and any python code can be executed in those functions, some of which should ideally be validation.

Multiple actions can be chained together under one Task and will execute in the defined order. Actions can pass information along via an in memory cache/field on the task object, but that is only safe for the same stage of execution. Actions can also store data back to the database if their logic requires some info passed along to a later step of execution.

See `actions.models` and `actions.v1` for a good idea of Actions.

Tasks originate at a TaskView, and start the action processing. They encompass the user side of interaction.

The main workflow consists of three possible steps which can be executed at different points in time, depending on how the TaskView and the actions within it are defined.

The base use case is three stages:

- **Recieve Request**
  - Validate request data against action serializers.
  - If valid, setup Task to represent the request, and the Actions specified for that TaskView.
  - The service runs the pre_approve function on all actions which should do any self validation to mark the actions themselves as valid or invalid, and populating the nodes in the Task based on that.
- **Admin Approval**
  - An admin looks at the Task and its notes.
  - **If they decide it is safe to approve, they do so.**

        * If there are any invalid actions approval will do nothing until the action data is updated and initial validation is rerun.
    - The service runs the post_approve function on all actions.
    - If any of the actions require a Token to be issued and emailed for additional data such as a user password, then that will occur.
    - If no Token is required, the Task will run submit actions, and be marked as complete.
- **Token Submit**
    - User submits the Token data.
    - The service runs the submit function on all actions, passing along the Token data, normally a password.
    - The action will then complete with the given final data.
    - Task is marked as complete.

There are cases and TaskViews that auto-approve, and thus automatically do the middle step right after the first. There are also others which do not need a Token and thus run the submit step as part of the second, or even all three at once. The exact number of 'steps' and the time between them depends on the definition of the TaskView.

Actions themselves can also effectively do anything within the scope of those three stages, and there is even the ability to chain multiple actions together, and pass data along to other actions.

Details for adding taskviews and actions can be found on the *Creating Plugins for Adjutant* page.

1.1 What is an Action?

Actions are a generic database model which knows what 'type' of action it is. On pulling the actions related to a Task from the database we wrap it into the appropriate class type which handles all the logic associated with that action type.

An Action is both a simple database representation of itself, and a more complex in memory class that handles all the logic around it.

Each action class has the functions "pre_approve", "post_approve", and "submit". These relate to stages of the approval process, and any python code can be executed in those functions.

1.2 What is a Task?

A task is a top level model representation of the request. It wraps the request metadata, and based on the TaskView, will have actions associated with it.

1.3 What is a Token?

A token is a unique identifier linking to a task, so that anyone submitting the token will submit to the actions related to the task.

1.4 What is an TaskView?

TaskViews are classes which extend the base TaskView class and use its inbuilt functions to process actions. They also have actions associated with them and the inbuilt functions from the base class are there to process and validate those against data coming in.

The TaskView will process incoming data and build it into a Task, and the related Action classes.

The base TaskView class has three functions:

- **get**
  - just a basic view function that by default returns list of actions, and their required fields for the action view.
- **process_actions**
  - needs to be called in the TaskView definition
  - A function to run the processing and validation of request data for actions.
  - Builds and returns the task object, or the validation errors.

At their base TaskViews are django-rest ApiViews, with a few magic functions to wrap the task logic.

CHAPTER 2

Deploying Adjutant in Devstack

This is a guide to setting up Adjutant in a running Devstack environment close to how we have been running it for development purposes.

This guide assumes you are running this in a clean ubuntu 16.04 virtual machine with sudo access.

2.1 Deploy Devstack

Grab the Devstack repo:

```
git clone https://github.com/openstack-dev/devstack.git
```

And then define a basic localrc file with the password set and place that in the devstack folder (adjutant's default conf assumes 'openstack' as the admin password):

```
ADMIN_PASSWORD=openstack
MYSQL_PASSWORD=openstack
DATABASE_PASSWORD=openstack
RABBIT_PASSWORD=openstack
SERVICE_PASSWORD=openstack
```

Run the devstack build:

```
./devstack/stack.sh
```

Provided your VM has enough ram to handle a devstack install this should take a while, but go smoothly. Ideally give your VM 5gb or more of ram, any less can cause the devstack build to fail.

2.2 Deploy Adjutant

Grab the Adjutant repo:

```
git clone https://github.com/openstack/adjutant.git
```

Then you'll want to setup a virtual environment:

```
cd adjutant
virtualenv venv
source venv/bin/activate
```

Once that is done you can install Adjutant and its requirements:

```
pip install -r requirements.txt
python setup.py develop
```

If you prefer you can install it fully, but using develop instead allows you update the Adjutant code and have the service reflect that without rerunning the install.

2.3 Configure Adjutant

Most of the default conf values should work fine against devstack, but one thing that you will need to change is the uuid for the public network in *DEFAULT_ACTION_SETTINGS* for the actions NewDefaultNetworkAction and NewProjectDefaultNetworkAction. If you don't set this correctly, then signups or tasks using those actions will not be able to correctly create a default network as they cannot find the correct external public network.

On a fresh devstack there is only one public network so to find the public network uuid you can to run:

```
openstack network show public
```

And then grab the id value and put that into the Adjutant conf.

2.3.1 Username is email

The example conf for Adjutant is setup with *USERNAME_IS_EMAIL = TRUE* which works on the assumption that usernames are emails. This is easy to change in the conf, but a fairly useful way of avoiding username clashes. If you set this to *False* then usernames will be required as well as emails for most tasks that deal with user creation.

Migrating between the two states hasn't yet been handled entirely, so once you pick a value for *USERNAME_IS_EMAIL* stick with it, or clear the database inbetween.

2.4 Running Adjutant

Still in the Adjutant repo directory, you will now need to run the migrations to build a basic database. By default this will use sqlite3.:

```
adjutant-api migrate
```

Now the that the migrations have been setup and the database built run the service from the same directory, and it will revert to using the config file at 'conf/conf.yaml':

```
adjutant-api runserver 0.0.0.0:5050
```

Note: The port doesn't matter, but 5050 is a safe bet as it isn't used by any other DevStack services and we can then safely assume you will be using the same url for the rest of the guide.

Now you have Adjutant running, keep this window open as you'll want to keep an eye on the console output.

2.5 Add Adjutant to Keystone Catalogue

In a new SSH termimal connected to your ubuntu VM setup your credentials as environment variables:

```
export OS_USERNAME=admin
export OS_PASSWORD=openstack
export OS_PROJECT_NAME=demo
export OS_USER_DOMAIN_NAME=default
export OS_PROJECT_DOMAIN_NAME=default
export OS_AUTH_URL=http://localhost/identity
export OS_IDENTITY_API_VERSION=3
export OS_REGION_NAME=RegionOne
```

If you used the localrc file as given above, these should work.

Now setup a new service in Keystone for Adjutant and add an endpoint for it:

```
openstack service create registration --name adjutant
openstack endpoint create adjutant public http://0.0.0.0:5050/v1 --region RegionOne
```

2.6 Adjutant specific roles

To allow certain actions, Adjutant requires two special roles to exist. You can create them as such:

```
openstack role create project_admin
openstack role create project_mod
```

Also because Adjutant by default also adds the role, you will want to create 'heat_stack_owner' which isn't by default present in devstack unless you install Heat:

```
openstack role create heat_stack_owner
```

2.7 Testing Adjutant via the CLI

Now that the service is running, and the endpoint setup, you will want to install the client and try talking to the service:

```
sudo pip install python-adjutantclient
```

In this case the client should be safe to install globally with sudo, but you can also install it in the same virtualenv as Adjutant itself, or make a new virtualenv.

Now lets check the status of the service:

```
openstack adjutant status
```

What you should get is:

```
{
    "error_notifications": [],
    "last_completed_task": null,
    "last_created_task": null
}
```

Seeing as we've done nothing to the service yet this is the expected output.

To list the users on your current project (admin users are hidden by default):

```
openstack project user list
```

The above action is only possibly for users with the following roles: 'admin', 'project_admin', 'project_mod'

Now lets try inviting a new user:

```
openstack project user invite bob@example.com project_admin
```

You will then get a note saying your invitation has been sent. You can list your project users again with 'openstack project user list' to see your invite.

Now if you look at the log in the Adjutant terminal you should still have open, you will see a print out of the email that would have been sent to bob@example.com. In the email is a line that looks like this:

```
http://192.168.122.160:8080/token/e86cbfb187d34222ace90845f900893c
```

Normally that would direct the user to a Horizon dashboard page where they can submit their password.

Since we don't have that running, your only option is to submit it via the CLI. This is cumbersome, but doable. From that url in your Adjutant output, grab the values after '.../token/'. That is bob's token. You can submit that via the CLI:

```
openstack admin task token submit <token> <json_data>
openstack admin task token submit e86cbfb187d34222ace90845f900893c '{"password":
↪"123456"}'
```

Now if you get the user list, you will see bob is now active:

```
openstack project user list
```

And also shows up as a user if you do:

```
openstack user list
```

And since you are an admin, you can even take a look at the tasks themselves:

```
openstack admin task list
```

The topmost one should be your invite, and if you then do a show using that id you can see some details about it:

```
openstack admin task show <UUID>
```

2.8 Setting Up Adjutant on Horizon

Adjutant has a Horizon UI plugin, the code and setup instructions for it can be found here

If you do set this up, you will want to edit the default Adjutant conf to so that the TOKEN_SUBMISSION_URL is correctly set to point at your Horizon.

CHAPTER 3

Configuring Adjutant

Adjutant is designed to be highly configurable for various needs. The goal of Adjutant is to provide a variety of common tasks and actions that can be easily extended or changed based upon the needs of your OpenStack.

The default Adjutant configuration is found in conf/conf.yaml, and but will be overriden if a file is placed at `/etc/adjutant/conf.yaml`.

The first part of the configuration file contains standard Django settings.

```
SECRET_KEY:

ALLOWED_HOSTS:
    - "*"

ADDITIONAL_APPS:
    - adjutant.api.v1
    - adjutant.actions.v1

DATABASES:
    default:
        ENGINE: django.db.backends.sqlite3
        NAME: db.sqlite3

LOGGING:

EMAIL_SETTINGS:
    EMAIL_BACKEND: django.core.mail.backends.console.EmailBackend
```

If you have any plugins, ensure that they are also added to **ADDITIONAL_APPS**.

The next part of the confirguration file contains a number of settings for all taskviews.

```
USERNAME_IS_EMAIL: True

KEYSTONE:
    username:
    password:
```

```
    project_name:
    auth_url: http://localhost:5000/v3
    domain_id: default

TOKEN_SUBMISSION_URL: http://192.168.122.160:8080/token/

# time for the token to expire in hours
TOKEN_EXPIRE_TIME: 24

ROLES_MAPPING:
    admin:
        - project_admin
        - project_mod
        - _member_
    project_admin:
        - project_admin
        - project_mod
        - _member_
    project_mod:
        - project_mod
        - heat_stack_owner
        - _member_
```

USERNAME_IS_EMAIL impacts account creation, and email modification actions. In the case that it is true, any task passing a username and email pair, the username will be ignored. This also impacts where emails are sent to.

The keystone settings must be for a user with administrative privileges, and must use the Keystone V3 API endpoint.

If you have Horizon configured with adjutant-api **TOKEN_SUBMISSION_URL** should point to that.

ROLES_MAPPING defines which roles can modify other roles. In the default configuration a user who has the role project_mod will not be able to modify any of the roles for a user with the project_admin role.

3.1 Standard Task Settings

```
ACTIVE_TASKVIEWS:
    - UserRoles
    - UserDetail
    - UserResetPassword
    - UserSetPassword
    - UserList
    - RoleList
    - SignUp
    - UserUpdateEmail
```

All in use taskviews, including those that are from plugins must be included in this list. If a task is removed from this list its endpoint will not be accessable however users who have started tasks will still be able submit them.

```
DEFAULT_TASK_SETTINGS:
    duplicate_policy: null
    emails:
        initial:
            subject: Initial Confirmation
            reply: no-reply@example.com
            from: bounce+%(task_uuid)s@example.com
            template: initial.txt
```

```
            # html_template: initial.txt
        token:

        completed:
    notifications:
        EmailNotification:
            standard:
                emails:
                    - example@example.com
                reply: no-reply@example.com
                from: bounce+%(task_uuid)s@example.com
                template: notification.txt
                # html_template: completed.txt
            error:
```

The default settings can be overridden for individual tasks in the TASK_SETTINGS configuration, these are cascading overrides. Two additional options are available, overriding the default actions or adding in additional actions. These will run in the order specified.

```
TASK_SETTINGS:
    signup:
        default_actions:
             - NewProjectAction
    invite_user:
        additional_actions:
            - SendAdditionalEmailAction
```

By default duplicate tasks will be marked as invalid, however the duplicate policy can be set to 'cancel' to cancel duplicates and start a new class.

You can also here at the task settings layer ensure that the task is never auto approved by it's underlying actions.

```
TASK_SETTINGS:
    update_quota:
        allow_auto_approve: False
```

3.1.1 Email Settings

The `initial` email will be sent after the user makes the request, the `token` email will be sent after approval steps are run, and the `completed` email will be sent after the token is submitted.

The emails will be sent to the current user, however this can be changed at the action level with the `get_email()` function.

3.1.2 Notification Settings

The type of notifications can be defined here for both standard notifications and error notifications:

```
notifications:
    EmailNotification:
        standard:
            emails:
                - example@example.com
            reply: no-reply@example.com
            template: notification.txt
```

```
        error:
            emails:
                - errors@example.com
            reply: no-reply@example.com
            template: notification.txt
    <other notification engine>:
```

Currently EmailNotification is the only available notification engine however new engines can be added through plugins and may have different settings.

3.2 Action Settings

Default action settings. Actions will each have their own specific settings, dependent on what they are for. The standard settings for a number of default actions are below:

An action can have it's settings overridden in the settings for it's task. This will only effect when the action is called through that specific task Overriding action settings for a specific task.

3.3 Email Templates

Additional templates can be placed in `/etc/adjutant/templates/` and will be loaded in automatically. A plain text template and an HTML template can be specified separately. The context for this will include the task object and a dictionary containing the action objects.

3.4 Additional Emails

The SendAdditionalEmailAction is designed to be added in at configuration for relevant tasks. It's templates are also passed a context dictionary with the task and actions available. By default the template is null and the email will not send.

The settings for this action should be defined within the action_settings for it's related task view.

```
additional_actions:
  - SendAdditionalEmailAction
action_settings:
    SendAdditionalEmailAction:
        initial:
            subject: OpenStack Email Update Requested
            template: email_update_started.txt
            email_current_user: True
```

The additional email action can also send to a subset of people.

The user who made the request can be emailed with

```
email_current_user: true
```

Or the email can be sent to everyone who has a certain role on the project. (Multiple roles can also be specified)

```
email_roles:
  - project_admin
```

Or an email can be sent to a specified address in the task cache (key: `additional_emails`)

```
email_in_task_cache: true
```

Or sent to an arbitrary administrative email address(es):

```
email_additional_addresses:
   - admin@example.org
```

This can be useful in the case of large project affecting actions.

CHAPTER 4

Creating Plugins for Adjutant

As Adjutant is built on top of Django, we've used parts of Django's installed apps system to allow us a plugin mechanism that allows additional actions and views to be brought in via external sources. This allows company specific or deployer specific changes to easily live outside of the core service and simply extend the core service where and when need.

An example of such a plugin is here: https://github.com/catalyst/adjutant-odoo

New TaskViews should inherit from adjutant.api.v1.tasks.TaskView can be registered as such:

```
from adjutant.api.v1.models import register_taskview_class,

from myplugin import tasks

register_taskview_class(r'^openstack/sign-up/?$', tasks.OpenStackSignUp)
```

Actions must be derived from adjutant.actions.v1.base.BaseAction and are registered alongside their serializer:

```
from adjutant.actions.v1.models import register_action_class

register_action_class(NewClientSignUpAction, NewClientSignUpActionSerializer)
```

Serializers can inherit from either rest_framework.serializers.Serializer, or the current serializers in adjutant.actions.v1.serializers.

A task must both be registered with a valid URL and specified in ACTIVE_TASKVIEWS in the configuration to be accessible.

A new task from a plugin can effectively 'override' a default task by registering with the same URL, and sharing the task type. However it must have a different class name and the previous task must be removed from ACTIVE_TASKVIEWS.

4.1 Building Taskviews

Examples of taskviews can be found in adjutant.api.v1.openstack

Minimally they can look like this:

```
class NewCreateProject(TaskView):

    task_type = "new_create_project"

    default_actions = ["NewProjectWithUserAction", ]

    def post(self, request, format=None):
        processed, status = self.process_actions(request)

        errors = processed.get('errors', None)
        if errors:
            self.logger.info("(%s) - Validation errors with task." %
                             timezone.now())
            return Response(errors, status=status)

        return Response(response_dict, status=status)
```

Access can be restricted with the decorators mod_or_admin, project_admin and admin decorators found in adjutant.api.utils. The request handlers are fairly standard django view handlers and can execute any needed code. Additional information for the actions should be placed in request.data.

4.2 Building Actions

Examples of actions can be found in adjutant.actions.v1.

Minimally actions should define their required fields and implement 3 functions:

```
required = [
    'user_id',
    'value1',
]

def _pre_approve(self):
    self.perform_action('initial')

def _post_approve(self):
    self.perform_action('token')
    self.action.task.cache['value'] = self.value1

def _submit(self, data):
    self.perform_action('completed')
    self.add_note("Submit action performed")
```

Information set in the action task cache is available in email templates under task.cache.value, and the action data is available in action.ActionName.value.

If a token email is needed to be sent the action should also implement:

```
def _get_email(self):
    return self.keystone_user.email
```

If an action does not require outside approval this function should be run at the pre-approval stage:

```
self.set_auto_approve(True)
```

If an action requires a token this should be set at the post approval stage:

```
self.action.need_token = True
self.set_token_fields(["confirm"])
```

All actions must be paired with a serializer to do basic data structure checking, but should also check data validity during the action. Serializers are django-rest-framework serializers, but there are also two base serializers available in adjutant.actions.v1.serializers, BaseUserNameSerializer and BaseUserIdSerializer.

All fields required for an action must be placed through the serializer otherwise they will be inaccessible to the action.

Example:

```
from adjutant.actions.v1.serializers import BaseUserIdSerializer
from rest_framework import serializers

class NewActionSerializer(BaseUserIdSerializer):
    value_1 = serializers.CharField()
```

4.3 Building Notification Engines

Notification Engines can also be added through a plugin:

```
from adjutant.notifcations.models import NotificationEngine
from django.conf import settings

class NewNotificationEngine(NotificationEngine):

    def _notify(self, task, notification):
        if self.conf.get('do_this_thing'):
          # do something with the task and notification


settings.NOTIFICATION_ENGINES.update(
  {'NewNotificationEngine': NewNotificationEngine})
```

They should be then refered to in conf.yaml:

```
TASK_SETTINGS:
    signup:
        notifications:
            NewNotificationEngine:
                standard:
                    do_this_thing: True
                error:
                    do_this_thing: False
```

4.4 Using the Identity Manager, and Openstack Clients

The Identity Manager is designed to replace access to the Keystone Client. It can be imported from `adjutant.actions.user_store.IdentityManager` . Functions for access to some of the other Openstack Clients are in `adjutant.actions.openstack_clients`.

CHAPTER 5

Quota Management

The quota API will allow users to change their quota values in any region to a number of preset quota definitions. If a user has updated their quota in the past 30 days or are attempting to jump across quota values, administrator approval is required. The exact number of days can be modified in the configuration file.

Adjutant will assume that you have quotas setup for nova, cinder and neutron. Adjutant offers deployers a chance to define what services they offer in which region that require quota updates. At present Adjutant does not check the catalog for what services are available, but will in future, with the below setting acting as an override.

The setting `QUOTA_SERVICES` can be modified to include or remove a service from quota listing and updating, and it is a mapping of region name to services with `*` acting as a wildcard for all regions:

```
QUOTA_SERVICES:
    "*":
        - cinder
        - neutron
        - nova
        - octavia
    RegionThree:
        - nova
        - cinder
        - neutron
```

A new service can be added by creating a new helper object, like `adjutant.common.quota.QuotaManager.ServiceQuotaCinderHelper` and adding it into the `_quota_updaters` class value dictionary. The key being the name that is specified in `QUOTA_SERVICES` and on the quota definition.

CHAPTER 6

Project Guide Lines

Because of the extremely vague scope of the Adjutant project, we need to have some sensible guides lines to help us define what isn't part of it, and what should or could be.

Adjutant is a service to let cloud providers build workflow around certain actions, or to build smaller APIs around existing things in OpenStack. Or even APIs to integrate with OpenStack, but do actions in external systems.

Ultimately Adjutant is a Django project with a few limitations, and the plugin system probably exposes too much extra functionality which can be added by a plugin. Some of this we plan to cut down, and throw in some explicitly defined limitations, but even with the planned limitations the framework will always be very flexible.

6.1 Should a feature become part of core

Core Adjutant is mostly two parts. The first is the underlying workflow system, the APIs associated that, and the notifications. The second is the provider configurable APIs. This separation will increase further as we try and distance the workflow layer away from having one task linked to a view.

Anything that is a useful improvement to the task workflow framework and the associated APIs and notifications system, is always useful for core. As part of this we do include, and plan to keep adding to, a collection of generally useful actions and tasks. For those we need to be clear what should be part of core.

1. Is the action one that better makes sense as a feature in one of the existing services in OpenStack? If so, we should add it there, and then build an action in Adjutant that calls this new API or feature.
2. Is the action you want to add one that is useful or potentially useful to any cloud provider? If it is too specific, it should not be added to core.
3. Is the action you want to add talking to system outside of Adjutant itself or outside of OpenStack? If either, then it should not be added to core.
4. Is the task (a combination of actions), doing something that is already in some fashion in OpenStack, or better suited to be a feature in another OpenStack service. If so, it does not belong in core.

In addition to that, we include a collection of generally useful API views which expose certain underlying tasks as part of the workflow framework. These also need clarification as to when they should be in core. These are mostly a way

to build smaller APIs that cloud users can use consume that underneath are using Adjutant's workflow framework. Or often build APIs that expose useful wrappers or supplementary logic around existing OpenStack APIs and features.

1. Is the API you are building something that makes better sense as a feature in one of the other existing OpenStack services? If so, it doesn't belong in Adjutant core.
2. Does the API query systems outside of Adjutant or OpenStack? Or rely on actions or tasks that also need to consume systems outside of Adjutant or OpenStack.

Note: If an action, task, or API doesn't fit in core, it may fit in a plugin, potentially even one that is maintained by the core team. If a feature isn't yet present in OpenStack that we can build in Adjutant quickly, we can do so as a semi-official plugin with the knowledge that we plan to deprecate that feature when it becomes present in OpenStack proper. In addition this process allows us to potentially allow providers to expose a variant of the feature if they are running older versions of OpenStack that don't entirely support it, but Adjutant could via the plugin mechanism. This gives us a large amount of flexibility, while ensuring we aren't reinventing the wheel.

6.2 Appropriate locations for types of logic in Adjutant

In Adjutant there are different elements of the system that are better suited to certain types of logic either because of what they expose, or what level of auditability is appropriate for a given area.

6.2.1 Actions and Tasks

Actions and Tasks (collections of actions), have no real constraint. An action can do anything, and needs a high level of flexibility. Given that is the cases they should ideally have sensible validation built in, and should log what they'd done so it can be audited.

6.2.2 Pluggable APIs

Within the pluggable APIs, there should never be any logic that changes resources outside of Adjutant. They should either only change Adjutant internal resources (such as cancel a task), or query and return data. Building an API which can return a complex query across multiple OpenStack services is fine, but if a resource in any of those services needs to be changed, that should always be done by triggering an underlying task workflow. This keeps the logic clean, and the changes auditable.

> **Warning:** Anyone writing API plugins that break the above convention will not be supported. We may help and encourage you to move to using the underlying workflows, but the core team won't help you troubleshoot any logic that isn't in the right place.

CHAPTER 7

Project Features

To be clear, Adjutant doesn't really have features. It's a framework for deployer defined workflow, and a service to expose those workflows on configurable APIs, and supplementary micro APIs. This provides useful ways to extend some functionality in OpenStack and wrap sensible business logic around it, all while providing a clear audit trail for all tasks that Adjutant handles.

Adjutant does have default implementations of workflows and the APIs for them. These are in part meant to be workflow that is applicable to any cloud, but also example implementations, as well as actions that could potentially be reused in deployer specific workflow in their own plugins. If anything could be considered a feature, it potentially could be these. The plan is to add many of these, which any cloud can use out of the box, or augment as needed.

To enable these they must be added to *ACTIVE_TASKVIEWS* in the conf file.

For most of these there are matching panels in Horizon.

7.1 Built in Tasks and APIs

7.1.1 UserList

List the users on your project if you have the *project_admin* or *project_mod* role, and allow the invitiation of additional members (via email) to your project.

Note: Adjutant-UI exposes this with the Project Users panel for Horizon.

7.1.2 UserRoles

Allows the editing of roles for users in the same project provided you have the *project_admin* or *project_mod* role.

Note: Adjutant-UI exposes this with the Project Users panel for Horizon.

7.1.3 RoleList

Mirco API to list roles that can be managed by your current user for users on your project.

Note: Adjutant-UI exposes this with the Project Users panel for Horizon.

7.1.4 UserDetail

Just a micro API to show details of users on your project, and cancel invite user tasks.

Note: Adjutant-UI exposes this with the Project Users panel for Horizon.

7.1.5 UserResetPassword

An unauthenticated API that allows password reset request submissions. Will check if user exists, and email user with password reset token to reset password. This token is usable in Horizon, or via the API directly.

Note: Adjutant-UI exposes this with the Forgot Password panel for Horizon.

7.1.6 SignUp

An unauthenticated API that allows prospective users to submit requests to have a project and account created. This will then notify an admin as configured and an admin can approve or cancel the request.

This is mostly built as a basic example of a signup workflow. Most companies would use this only as a template and expand on the actions to talk to external systems and facilitate much more complex validation.

A more complex example of a signup process built on top of the defaults is Catalyst Cloud's own one: https://github.com/catalyst-cloud/adjutant-odoo

Note: Adjutant-UI exposes this with the Sign Up panel for Horizon.

7.1.7 UserUpdateEmail

A simple task that allows a user to update their own email address (or username if username==email). An email is sent to the old email informing them of the change, and a token to the new email so that the user must confirm they have correctly given their email.

Note: Adjutant-UI exposes this with the Update Email Address panel for Horizon.

7.1.8 UpdateProjectQuotas

A way for users to request quota changes between given sizes. These requests are either automatically approved if configured as such, or require an admin to approve the quota change.

Note: Adjutant-UI exposes this with the Quota Management panel for Horizon.

CHAPTER 8

Project History

Adjutant was started by CatalystCloud to fill our needs around missing features in OpenStack. CatalystCloud is public cloud provider based in New Zealand with a strong focus on opensource, with a team of operators and developers who are all contributors to OpenStack itself.

Early prototyping for Adjutant began at the end of of 2015 to fill some of the missing pieces we needed as a public cloud provider. It was initially something we had started designing as far back as early 2014, with the scope and design changing many times until initial prototyping and implementation was started in late 2015.

Originally it was designed to act as a service to manage customers, their users, projects, quotas, and to be able to process signups and initial resource creation for new customers. It would act as a layer above OpenStack and most non-authentication based identity management from a user perspective would happen through it, with the service itself making the appropriate changes to the underlying OpenStack services and resources. The reason why it didn't end up quite so complex and so big is because OpenStack itself, and many of the services (and the future roadmap) had planned solutions to many of the things we wanted, and our business requirements changed to storing our customer information in an eternal ERP system (Odoo/OpenERP at the time) rather than a standalone service.

So instead of something so grand, we tried smaller, a service that handles our unique public cloud requirements for signup. It should take in signup data from some source such as a public API that our website posts to, then validate it, give us those validation notes, and lets us decide if we wanted that customer, or even have the system itself based on certain criteria approve that customer itself. It would then create the user in Keystone, create a project, give them access, create a default network, and then also link and store the customer data in our ERP and billing systems. This was the initial 'openstack-registration' project, and the naming of which is present in our git history, and where the initial service-type name comes from.

As we prototyped this we realised that it was just a workflow system for business logic, so we decided to make it flexible, and found other things we could use it for:

- Allowing non-admin users to invite other users to their project.
- Let users reset their password by sending an email token to them.
- Manage and create child-projects in a single domain environment.
- Request quota increases for their projects.

All with the optional step of actually requiring an admin to approve the user request if needed. And with a good audit trail as to where the actions came from, and who approved them.

Eventually it also got a rename, because calling it OpenStack Registration got dull and wasn't accurate anymore. The name it got at the time was StackTask, and there are still elements of that naming in our systems, and plenty in the git history. Eventually we would rename it again because the name still being feel right, and was too close to StackTach.

Around that time we also added plugin support to try and keep any company specific code out of the core codebase, and in the process realised just how much further flexibility we'd now added.

The service gave us an easy way to build APIs around workflow we wanted our customers to be able to trigger around larger normally unsafe admin APIs in OpenStack itself. With the ability to have those workflows do changes to our ERP system, and other external systems. It gave us the missing glue we needed to make our public cloud business requirements and logic actually work.

But we were always clear, that if something made better sense as a feature in another service, we should implemented in that other service. This was meant to be a glue layer, or potentially for mini API features that don't entirely have a good place for them, or just a wrapper around an existing OpenStack feature that needs organisation specific logic added to it.

Throughout all this, the goal was always to keep this project fully opensource, to invite external contribution, to do our planning, bug tracking, and development where the OpenStack community could see and be transparent about our own internal usage of the service and our plans for it. The code had been on our company github for a while, but it was time to move it somewhere better.

So we renamed again, and then finally moved all the core repos to OpenStack infrastructure, as well as the code review, bug, and spec tracking.

Adjutant, in it's current form is the culmination of that process, and while the core driving force behind Adjutant was our own needs, it always was the intention to provide Adjutant for anyone to build and use themselves so that their effort isn't wasted threading the same ground.

A basic workflow framework built using Django and Django-Rest-Framework to help automate Admin tasks within an OpenStack cluster.

The goal of Adjutant is to provide a place and standard actions to fill in functionality missing from Keystone, and allow for the easy addition of business logic into more complex tasks, and connections with outside systems.

Tasks are built around three states of initial submission, admin approval and token submission. All of the states are not always used in every task, but this format allows the easy implementation of systems requiring approval and checks final user data entry.

While this is a Django application, it does not follow the standard Django folder structure because of certain packaging requirements. As such the project does not have a manage.py file and must be installed via setup.py or pip.

Once installed, all the normal manage.py functions can be called directly on the 'adjutant-api' commandline function.

The command `tox -e venv {your commands}` can be used and will setup a virtual environment with all the required dependencies for you.

For example, running the server on port 5050 can be done with:: tox -e venv adjutant-api runserver 0.0.0.0:5050

CHAPTER 9

Client and UI Libraries

Both a commandline/python and a horizon plugin exist for adjutant:

- python-adjutantclient
- adjutant-ui

CHAPTER 10

Tests and Documentation

Tests and documentation are managed by tox, they can be run simply with the command `tox`.

To run just action unit tests:

```
tox adjutant.actions
```

To run a single api test:

```
tox adjutant.api.v1.tests.test_api_taskview.TaskViewTests.test_duplicate_tasks_new_
↪user
```

Tox will run the tests in Python 2.7, Python 3.5 and produce a coverage report.

Api reference can be generated with the command `tox -e api-ref`. This will be placed in the `api-ref/build` directory, these docs can be generated with the command `tox -e docs`, these will be placed inside the `doc/build` directory.

CHAPTER 11

Contributing

Bugs and blueprints for Adjutant, its ui and client are managed here on launchpad.

Changes should be submitted through the OpenStack gerrit, the guide for contributing to OpenStack projects is here .