
Orchestra

Release 0.1.2

Feb 02, 2018

Contents

1	Getting Started with Orchestra in 5 Minutes	3
1.1	Install Dependencies	3
1.2	Create a Django Project	3
1.3	Install and Configure Orchestra	3
1.4	Run Orchestra	5
1.5	Run the Example Project Demo	5
2	Example Use Case: The Newsroom	7
2.1	The workflow	7
2.2	Running the workflow	8
2.3	The code	11
3	Key Concepts	17
3.1	Workflows	18
3.2	Project Distribution	18
3.3	Hierarchical Review	18
3.4	Worker Certification	18
3.5	Life of a Task	19
4	Feature Walkthrough	21
4.1	Time Tracking	21
5	Orchestra Bots	25
5.1	StaffBot	25
5.2	SanityBot	29
6	Our Motivation	31
7	How to contribute to Orchestra	33
7.1	Getting involved	33
7.2	Setting up for Development	33
7.3	Development Workflow	33
7.4	Quick Style Guide	34
8	API Reference	37
8.1	Client API	37
9	Core Reference	41

9.1	Models	41
9.2	Task Lifecycle	41
10	Indices and tables	43
	HTTP Routing Table	45

Orchestra is an open source system to orchestrate teams of experts as they complete complex projects with the help of automation.

Getting Started with Orchestra in 5 Minutes

What follows is a simple 5-minute guide to getting up and running with Orchestra that assumes some basic Python and Django experience, but not much else. For a deeper introduction, you might want to check out our [Key Concepts](#), and for in-depth information on using and developing with Orchestra, take a look at our [API documentation](#).

1.1 Install Dependencies

Orchestra requires Python 3 and Django version 1.8 or higher to run, so make sure you [have them installed](#). We recommend setting up a [virtual environment](#) to isolate your Python dependencies, and we're fond of [virtualenvwrapper](#) to make that process easier. Make sure to create your virtual environment with Python 3 by passing `--python=/path/to/bin/python3` if it isn't your default development setup.

Orchestra requires a number of Python dependencies to run. You can install them by simply pulling down and installing our `requirements.txt` file:

```
wget https://raw.githubusercontent.com/b12io/orchestra/stable/requirements.txt
pip install -r requirements.txt
```

1.2 Create a Django Project

Orchestra is a Django app, which means that it must be run within a Django project (for more details, read [the Django tutorial](#) on this topic). Start a project with `django-admin startproject your_project`, replacing `your_project` with your favorite project name (but don't name it `orchestra`, which will conflict with our namespace). From here on out, this document will assume that you stuck with `your_project`, and you should replace it appropriately.

1.3 Install and Configure Orchestra

Next, let's get Orchestra installed and running. To get the code, just install using `pip`: `pip install orchestra`.

Orchestra has a number of custom settings that require configuration before use. First, download the default Orchestra settings file and place it next to the project settings file:

```
wget https://raw.githubusercontent.com/bl2io/orchestra/stable/example_project/example_
→project/orchestra_settings.py
mv orchestra_settings.py your_project/your_project
```

Next, edit the `orchestra_settings.py` file:

- Add 'simple_workflow' to `settings.ORCHESTRA_WORKFLOWS` in the “General” section if you want to run the demo workflow (*instructions below*), and add 'journalism_workflow' if you want to run the *journalism workflow*.
- Adjust your [email settings](#). By default, Orchestra will direct all messages to the console, but for a realistic registration workflow you'll want to set up a real mail server that can actually send emails.
- Change settings like the `ORCHESTRA_PROJECT_API_SECRET` from 'CHANGEME' to more appropriate values.
- Optionally, add 3rd party credentials in the “3rd Party Integrations” section so that Orchestra can store files on [Amazon S3](#), use [Google Apps](#) and [Slack](#) to help communicate with expert workers, and track usage in [Google Analytics](#).

Then, at the bottom of your existing settings file (`your_project/your_project/settings.py`), import the Orchestra settings:

```
from .orchestra_settings import setup_orchestra
setup_orchestra(__name__)
```

You'll also need to set up Orchestra's URLs, so that Django knows where to route users when they view Orchestra in the browser. If you don't have any URLs of your own yet, you can just download our bare-bones example file with `wget https://raw.githubusercontent.com/bl2io/orchestra/stable/example_project/example_project/urls.py`.

Alternatively, just make sure to add the following code inside the `urlpatterns` variable in `your_project/your_project/urls.py`:

```
# Admin Views
url(r'^orchestra/admin/',
    include(admin.site.urls)),

# Registration Views
# Eventually these will be auto-registered with the Orchestra URLs, but for
# now we need to add them separately.
url(r'^orchestra/accounts/',
    include('registration.backends.default.urls')),

# Optionally include these routes to enable user hijack functionality.
url(r'^orchestra/switch/', include('hijack.urls')),

# Logout then login is not available as a standard django
# registration route.
url(r'^orchestra/accounts/logout_then_login/$',
    auth_views.logout_then_login,
    name='logout_then_login'),

# Orchestra URLs
url(r'^orchestra/',
    include('orchestra.urls', namespace='orchestra')),
```



```
# Beanstalk Dispatch URLs
url(r'^beanstalk_dispatch/',
    include('beanstalk_dispatch.urls')),
```

Finally, you'll need to get the database set up. Create your database with `python manage.py migrate`. You'll also want to make sure you have loaded our example workflows and set up some user accounts to try them out. To load the workflows, run:

```
python manage.py loadworkflow <APP_LABEL> <WORKFLOW_VERSION>
```

Each of our example workflows provides a set of sample users already configured with proper certifications. To load them, run:

```
python manage.py loadworkflowsampleddata <WORKFLOW_SLUG>/<WORKFLOW_VERSION>
```

If you would like to load all of the workflows, then run:

```
python manage.py loadallworkflows
```

The example workflows we currently release with Orchestra are:

- A *simple demo workflow* with one human and one machine step. Its app label is `simple_workflow`, its workflow slug is `simple_workflow`, and the latest version is `v1`.
- A more complicated *journalism workflow*. Its app label is `journalism_workflow`, its workflow slug is `journalism`, and the latest version is `v1`.

In addition, you can use the Orchestra admin (<http://127.0.0.1:8000/orchestra/admin>) to create new users and certifications of your own at any time once Orchestra is running. If you haven't created an admin account for your Django project, you can load a sample one (username: `admin`, password: `admin`) with `python manage.py loaddata demo_admin`.

We provide the option to use the third-party package `django-hijack` to act on behalf of users. To enable this setting, ensure that the following setting is set `HIJACK_ALLOW_GET_REQUESTS = True`, in addition to including the urls referenced above.

Now Orchestra should be ready to go! If you're confused about any of the above, check out our barebones [example project](#).

1.4 Run Orchestra

Now that Orchestra is configured, all that remains is to fire it up! Run your Django project with `python manage.py runserver` (you'll want to switch to something more robust in production, of course), and navigate to `http://127.0.0.1:8000/orchestra/app` in your favorite browser.

If you see the Orchestra sign-in page, your setup is working! If you loaded the simple workflow's sample data above, logging in as its user (username `demo`, password `demo`) should show you a dashboard with no available tasks.

1.5 Run the Example Project Demo

To give you a feel for what it means to run an Orchestra workflow from end to end, we've included a very simple example workflow with two steps, one machine and one human. The machine step takes a URL and extracts a random image from the page. The human step asks an expert to rate how "awesome" the image is on a scale from one to five. If

you're interested in how we defined the workflow, take a look at [the code](#), though we walk through a more interesting example in [this documentation](#).

We've written an interactive script to walk through this simple workflow. To run it:

- Make sure you added `simple_workflow` to your `ORCHESTRA_WORKFLOWS` setting following the previous section.
- Make sure you loaded the workflow and its sample data following the previous section. This should have created a user with username `demo` and password `demo`.
- Run the interactive walkthrough:

```
python manage.py interactive_simple_workflow_demo
```

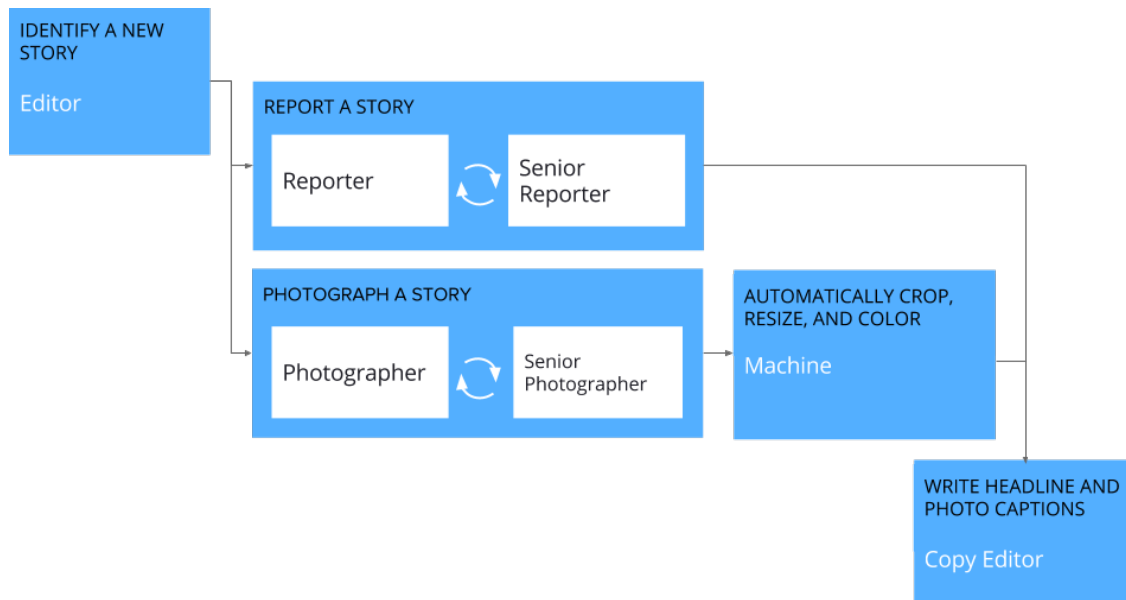
The script will walk you through using [the *Orchestra Client API*](#) to create a new project based on the simple workflow, explaining which API calls to use, what their output looks like, and how machine steps interact with human steps and pass data back and forth.

If you don't want to go to the trouble of running the script yourself, take a look at the transcript of expected output.

Example Use Case: The Newsroom

Below we'll walk you through an example of how Orchestra could be used in a newsroom by journalists, editors, and photographers to craft a story. The code for this example can be found [in our github repo](#).

2.1 The workflow



The image above depicts our example workflow, which is composed of the following steps:

- An editor finds a good story and sends a reporter off to investigate.
- The reporter writes up a draft article.
- A more experienced reporter then reviews the article and suggests improvements.

- In parallel with the reporting step, a photographer captures photos for the story.
- A senior photographer reviews the photos and selects the best ones.
- The selected photos are resized and recolored for display across different media.
- Finally, a copy editor adds headlines and photo captions to complete the story.

To make things work in practice, there's also a hidden machine step at the beginning of the workflow to set up some google documents and folders for article writing and image storage.

2.2 Running the workflow

2.2.1 Setup

If you haven't followed the [getting started guide](#) to set up Orchestra yet, you should do that now. Also, make sure that 'journalism_workflow' is in your `INSTALLED_APPS` setting, and that you have loaded the workflow into the database (`python manage.py loadworkflow journalism_workflow v1`).

The journalism workflow requires Google Apps integration to run, so make sure in `orchestra_settings.py` you set `settings.GOOGLE_APPS` to `True`, and fill in values for `settings.GOOGLE_SERVICE_EMAIL`, `settings.GOOGLE_P12_PATH`, and `settings.GOOGLE_PROJECT_ROOT_ID`. Set up and correct values for those settings are described in [the Google Apps documentation](#).

Next, make sure you have the journalism workflow sample data installed by running (if you haven't already) `python manage.py loadworkflowsampleddata journalism/v1`. This will create the following accounts:

- username: journalism-editor, password: editor. A worker with editor certification.
- username: journalism-reporter-1, password: reporter. A worker with entry-level reporter certification.
- username: journalism-reporter-2, password: reporter. A worker with review-level reporter certification.
- username: journalism-photographer-1, password: photographer. A worker with entry-level photographer certification.
- username: journalism-photographer-2, password: photographer. A worker with review-level photographer certification.
- username: journalism-copy-editor, password: copy-editor. A worker with copy_editor certification.

2.2.2 Start the workflow

The journalism workflow comes with a management script to start and monitor the workflow. To start the workflow:

- Make sure Orchestra is running with `python manage.py runserver`.
- In another tab, run:

```
python manage.py journalism_workflow_ctl --new
```

This will take a bit (because it is automatically running the document creation workflow step), but will eventually return a project id (probably 1), which you should store for future use, and output JSON info about the project.

2.2.3 Complete the steps

To navigate the workflow, first log in as the `journalism-editor` user and request a new task. The interface should look like the image below:

ORCHESTRA journalism-editor -

Journalism Workflow A test run of our journalism workflow **Article Planning**

Plan out the story idea

Who: The Pope

What: Visiting New York, attending the UN

When: September 24-27th, 2015

Where: New York City, NY

Notes: Traffic affected

Submit **Save** Saved at 1:07:31 PM

Fill out the high-level story idea and submit the task.

Next, log in as the `journalism-reporter-1` worker, and you should now have a reporting task available. The interface looks like the image below—use the google doc to write your article.

ORCHESTRA journalism-reporter-1 -

Journalism Workflow A test run of our journalism workflow **Reporting**

Draft the article based on the editor's story plan

Editor's story plan

Who: The Pope

What: Visiting New York, attending the UN

When: September 24-27th, 2015

Where: New York City, NY

Notes: Traffic affected

Article draft [Open in a new window](#)

Article Draft

Write your article below, taking into account the Editor's notes:

When you submit, you'll note that the task appears in the 'Awaiting Review' section. That's your cue to log in as `journalism-reporter-2` and review the work. Once you're satisfied with it, accept it.

In parallel to logging in as a reporter, you can log in as `journalism-photographer-1` and `journalism-photographer-2` to take and review photographs relevant to the article. You should see an in-

terface like the image below, which encourages you to add photos to a shared ‘Raw Photos’ folder. The interface should look like the below:

ORCHESTRA

journalism-photographer-1

Take and upload photos for the article

Editor's story plan

Who

What

When

Where

Notes

Photographer instructions

Read the editor's notes carefully, then take photographs that will go well with the article. When you are finished, upload the photographs to the "Raw Photos" subfolder of the "Project Drive Folder" above.

Submit

Save

Once you've accepted the photography as `journalism-photographer-2`, the machine task to auto-process the photos should run. Our implementation simply makes any images in ‘Raw Photos’ greyscale, but you could imagine more complicated adjustments.

Finally, log in as `journalism-copy-editor` to give the article a headline and caption the photos. You should observe that your photos have been greyscaled as desired, as in the image below:

ORCHESTRA


journalism-copy-editor

Finalize the article copy


Write a headline for the article:

Headline


Caption the photos taken for the article:



Caption:



Caption:



Caption:

Once you submit the task, the workflow is done! You've successfully coordinated 6 expert workers and 2 machine tasks to tell a story.

2.2.4 Verify the final JSON output

You'll note that our workflow didn't actually lay the article out in its final print or electronic form. That's because, in reality, this workflow would have been kicked off by a newsroom's content management system with auto-layout capabilities based on the JSON the project produced. To see the JSON that the workflow produces for input into such a system, run:

```
python manage.py journalism_workflow_ctl --finish -p <PROJECT_ID>
```

where <PROJECT_ID> is the project id you were given when you created the project.

You should see output like:

```
{'articleDocument': 'https://docs.google.com/document/d/someid',
 'headline': 'Your Headline',
 'photos': [{'caption': 'Your Caption 1',
               'src': 'https://docs.google.com/uc?id=someid'},
             {'caption': 'Your Caption 2',
               'src': 'https://docs.google.com/uc?id=someid2'},
             ...
            ]
}
```

which summarizes all of the work accomplished in the workflow.

2.3 The code

All of the code used to create the new room workflow is located in [our github repo](#). There are three main components to the code: The workflow definition, the interface implementations for the human steps, and the python code for the machine steps.

2.3.1 The workflow definition

The workflow is defined in `journalism_workflow/workflow.json`, and its latest version (version 1) is defined in `journalism_workflow/v1/version.json`. These files declaratively defines the steps listed above, in programmatic form.

`workflow.json` defines the workflow with a name and short description:

```
{
  "slug": "journalism",
  "name": "Journalism Workflow",
  "description": "Create polished newspaper articles from scratch.",
}
```

It also describes certifications required by the workflow:

```
{
  "certifications": [
    {
      "slug": "editor",
      "name": "Editor",
      "description": "Trained in planning story ideas"
    },
    {
      "slug": "reporter",
    }
  ]
}
```

```
    "name": "Reporter",
    "description": "Trained in researching and writing articles"
  },
  {
    "slug": "photographer",
    "name": "Photographer",
    "description": "Trained in taking photos for articles"
  },
  {
    "slug": "copy_editor",
    "name": "Copy Editor",
    "description": "Trained in assembling photos and text into article layout"
  }
]
```

And provides the location of a python function to load sample data:

```
{
  "sample_data_load_function": {
    "path": "journalism_workflow.load_sample_data.load"
  }
}
```

`version.json` defines the steps of the workflow. Check out [the source](#) for all of the step definitions, but here we'll list two.

Below is the definition of the human step that takes an editor's story idea and asks a reporter to write an article based on it:

```
{
  "slug": "reporting",
  "name": "Reporting",
  "description": "Research and draft the article text",
  "is_human": true,
  "creation_depends_on": [
    "article_planning"
  ],
  "required_certifications": [
    "reporter"
  ],
  "review_policy": {
    "policy": "sampled_review",
    "rate": 1,
    "max_reviews": 1
  },
  "creation_policy": {
    "policy_function": {
      "path": "orchestra.creation_policies.always_create",
    }
  },
  "user_interface": {
    "angular_module": "journalism_workflow.v1.reporter",
    "angular_directive": "reporter",
    "javascript_includes": [
      "journalism_workflow/v1/reporter/js/modules.js",
      "journalism_workflow/v1/reporter/js/controllers.js",
      "journalism_workflow/v1/reporter/js/directives.js"
    ]
  }
}
```



```

    ]
  }
}

```

Note that we've specified step dependencies with `creation_depends_on`, required worker skills with `required_certifications`, and user interface javascript files with `user_interface`. In addition, we've asked that all reporters have their work reviewed by a senior reporter by specifying a sampled `review_policy` with a rate of 100% (rate goes from 0 to 1). We've also specified a `creation_policy`. Creation policies can be used to conditionally create a task based on previous step and project information.

Next, we show a machine step, in this case the step that takes our photographers' output (a directory of images), and processes those images for layout:

```

{
  "slug": "photo_adjustment",
  "name": "Photo Adjustment",
  "description": "Automatically crop and rescale images",
  "is_human": false,
  "creation_depends_on": [
    "photography"
  ],
  "execution_function": {
    "path": "journalism_workflow.v1.adjust_photos.autoadjust_photos"
  }
}

```

The basic arguments are similar, but we specify the step type as not human (`"is_human": false`), and instead of a user interface, we pass a python function to execute (`autoadjust_photos` here).

2.3.2 The interface implementations

In order for our workflows to be usable by experts, we need to display an interface for each human step. Orchestra uses `angular.js` for all of our interfaces. The interfaces all live under `journalism_workflow/static/journalism_workflow`.

Remember that in our *workflow definition*, we specified user interfaces with JSON that looked like this:

```

{
  "angular_module": "journalism_workflow.v1.editor",
  "angular_directive": "editor",
  "javascript_includes": [
    "journalism_workflow/v1/editor/js/modules.js",
    "journalism_workflow/v1/editor/js/controllers.js",
    "journalism_workflow/v1/editor/js/directives.js"
  ],
  "stylesheet_includes": []
}

```

Orchestra will automatically inject the specified `angular_directive` into the website, which should be implemented in the files listed in `javascript_includes`. To have CSS available in your interface, just list the file paths in `stylesheet_includes`.

An angular interface is composed of a few things: a `controller` that sets up state for the interface, a `directive` that injects the interface into a website, a `module` that registers the controllers and directives, and a `partial` that contains an html template for the interface. The `angular docs` do a better job of explaining these than we will, but here are a couple of things to keep in mind:

- In our directives, we use:

```
scope: {
  taskAssignment: '=',
}
```

to ensure that the input data for a step is available (it will be accessible at `taskAssignment.task.data`

- In our controllers, we use:

```
MyController.$inject = ['$scope', 'orchestraService'];
```

to ensure that the task data is passed to the controller. `orchestraService` has useful convenience functions for dealing with the task data like `orchestraService.taskUtils.prerequisiteData($scope.taskAssignment, stepSlug, dataKey)`, which will get the taskAssignment for the previous step called `step_slug` (and optionally the data specified by `data_key`).

And of course, please refer to [the newsroom workflow step interfaces](#) as examples.

2.3.3 The machine steps

Our workflow has two machine steps, one for creating documents and folders, and one for adjusting images.

A machine step is just a Python function with a simple signature:

```
def my_machine_step(project_data, prerequisites):
    # implement machine-y goodness
    return { 'output_data_key': 'value' }
```

It takes two arguments, a python dictionary containing global project data and a python dictionary containing state from all prerequisite workflow steps (and their prerequisites, and so on). The function can do whatever it likes, and returns a JSON-encodable dictionary containing state that should be made available to future steps (in the `prerequisites` argument for a machine step, and in the angular scope for a human interface).

For example, our image adjustment step (in [journalism_workflow/v1/adjust_photos.py](#)) gets the global project directory from `project_data`, uses Orchestra's Google Apps integration to create a new subfolder for processed photos, downloads all the raw photos, uses `pillow` to process them (for now it just makes them greyscale), then re-uploads them to the new folder.

2.3.4 Providing sample data

In the [workflow definition](#), we specified a module and function name for loading sample data with JSON that looked like:

```
{
  "sample_data_load_function": {
    "path": "journalism_workflow.load_sample_data.load"
  }
}
```

This function should create Django model objects (typically `Users`, `Workers`, and `WorkerCertifications`) that are helpful for a sample run through the workflow. The function has a simple signature, and might look like (for example):

```
from django.contrib.auth.models import User

def load(workflow_version):
    user = User.objects.update_or_create(
```

```
username='test_user',
defaults={
    'first_name': 'Test',
    'last_name': 'User',
})
user.set_password('test')
```

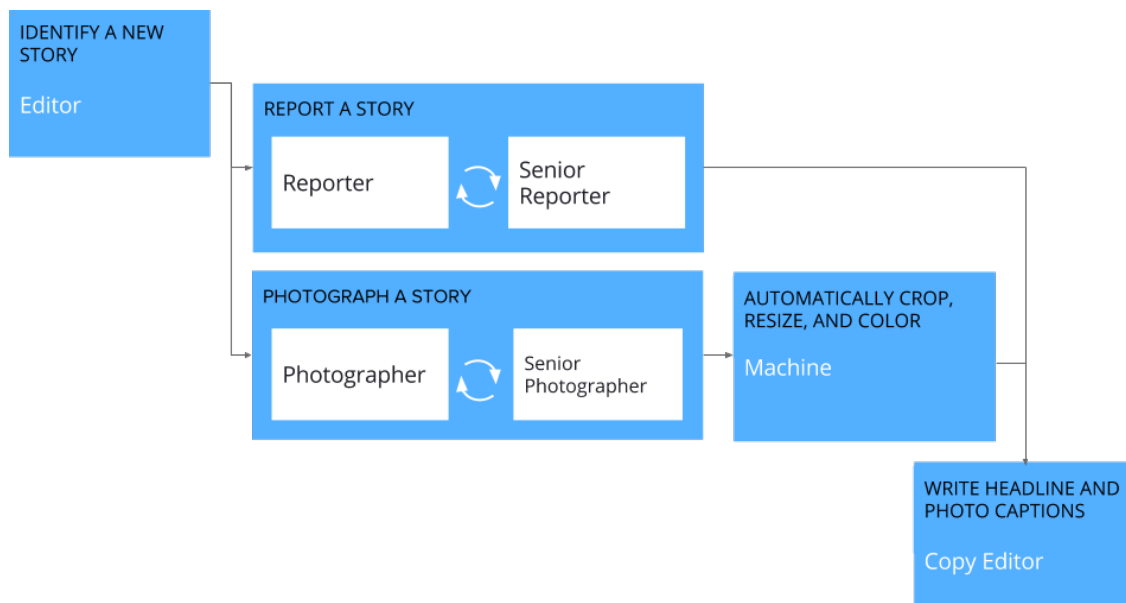
Once that function is defined, sample data can be loaded with:

```
python manage.py loadworkflowsampleddata <WORKFLOW_SLUG>/<WORKFLOW_VERSION>
```


CHAPTER 3

Key Concepts

Let's first recap our example *reporting workflow*:



- An editor finds a good story and sends a reporter off to investigate.
- The reporter writes up a draft article.
- A more experienced reporter then reviews the article and suggests improvements.
- In parallel with the reporting step, a photographer captures photos for the story.
- A senior photographer reviews the photos and selects the best ones.
- The selected photos are resized and recolored for display across different media.
- Finally, a copy editor adds headlines and photo captions to complete the story.

We'll now walk you through major Orchestra concepts based on the example above.

3.1 Workflows

- The entire process above is called a **workflow**, comprised of five component **steps**.
- Two of these steps require **review**, where more experienced experts review the original work performed. Custom review policies (e.g., sampled or systematic review) for tasks can be easily created in Orchestra.
- The photo resizing step is a **machine step**, completed by automation rather than by experts.
- Each step emits a JSON blob with structured data generated by either humans or machines.
- Steps have access to data emitted by previous steps that they depend on. In the example, the copy editor has access to both the story and the resized photos.

3.2 Project Distribution

- **Projects** are a series of interconnected **tasks**. A project is an instance of a workflow; a task is an instance of a step.
 - *An editor with a story about local elections would create an elections project, with tasks for a reporter/photographer/copy editor.*
- **Tasks** are carried out by an **expert** or by a **machine**.
 - *Photographers capture the story.*
 - *Machines resize and recolor the photos.*
- Experts can come from anywhere, from a company's employees to freelancers on platforms like Upwork.

3.3 Hierarchical Review

- **Core experts** do the initial work on a task.
- **Reviewers** provide feedback to other experts to make their work even better.
- The core expert **submits** the task when their work is complete.
- The reviewer can choose to **accept** the task, which is either selected for further review or marked as complete.
- They could also choose to **return** the task, requesting changes from and giving feedback to the worker they are reviewing.

3.4 Worker Certification

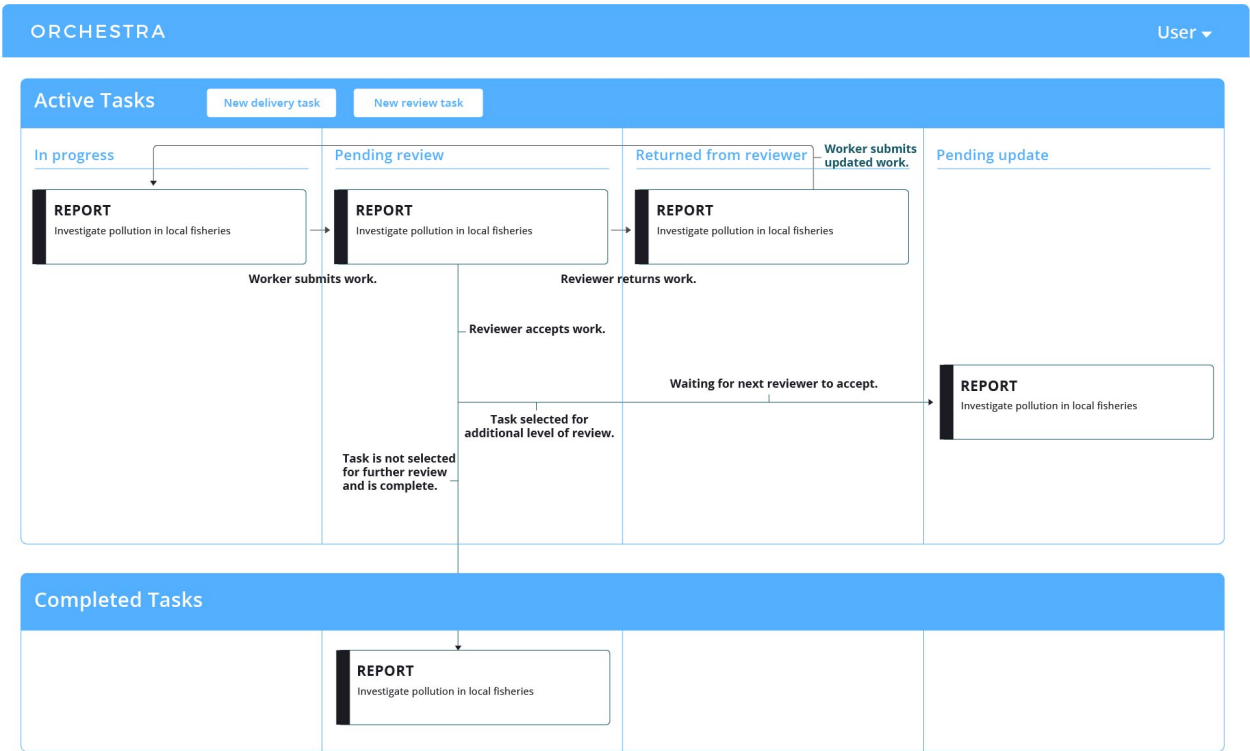
- Certifications allow experts to work on tasks they're great at.
- Experts can work toward all sorts of certifications, picking up practice tasks to build experience.
 - *Joseph is a solid reporter but needs a little more practice as a photographer—let's give him some simple tasks so he can improve!*
- Experts need additional certification to work in a reviewer role.
 - *Amy has been reporting for quite some time and would be great at mentoring new reporters.*

3.5 Life of a Task

Below are two images of the Orchestra dashboard, the launching point for expert workers. Click to see how tasks move differently across the dashboard for core workers and reviewers.

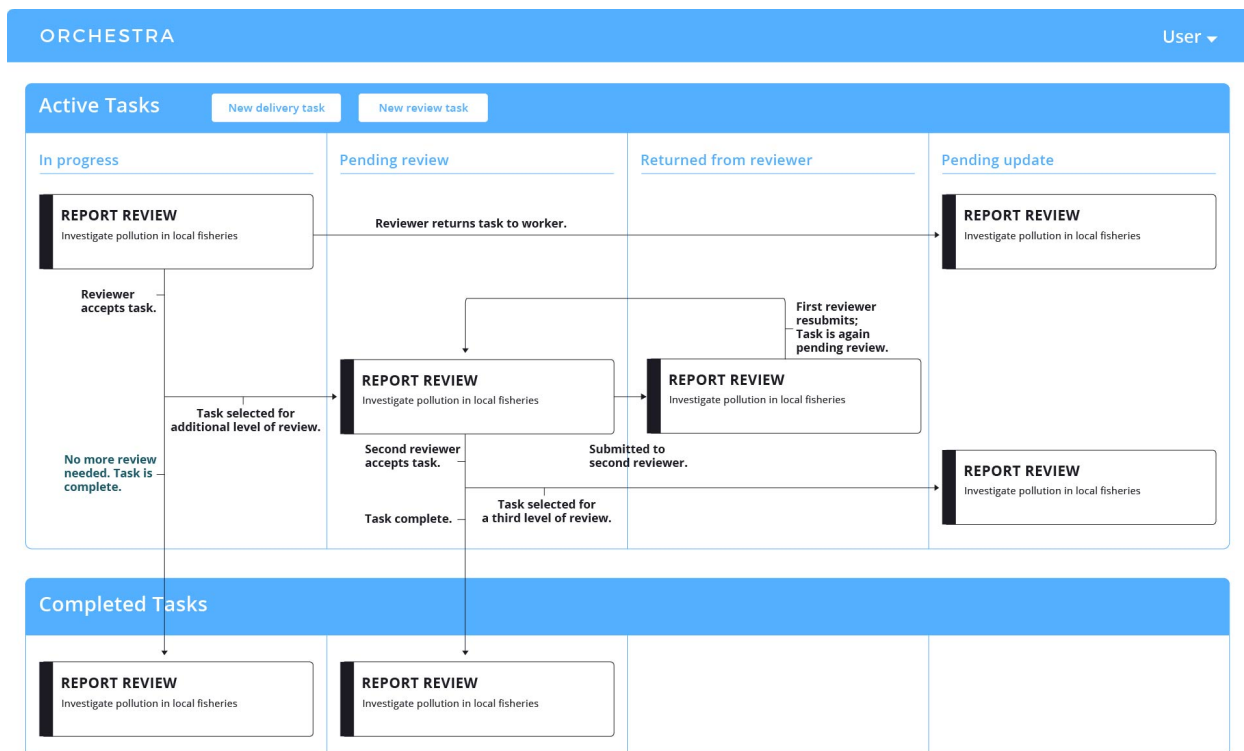
3.5.1 Core Expert

A core expert performs initial task work which will later be reviewed. The diagram below shows a task’s movement through the core worker’s dashboard.



3.5.2 Reviewer

A reviewer evaluates the core expert’s work and provides feedback. The diagram below shows a task’s movement through a reviewer’s dashboard.



Feature Walkthrough

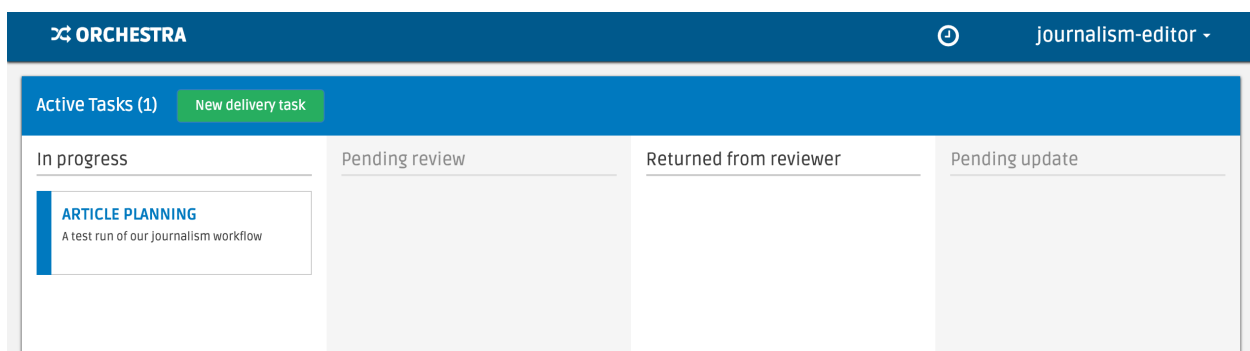
Below, we'll walk through the various features in Orchestra.

4.1 Time Tracking

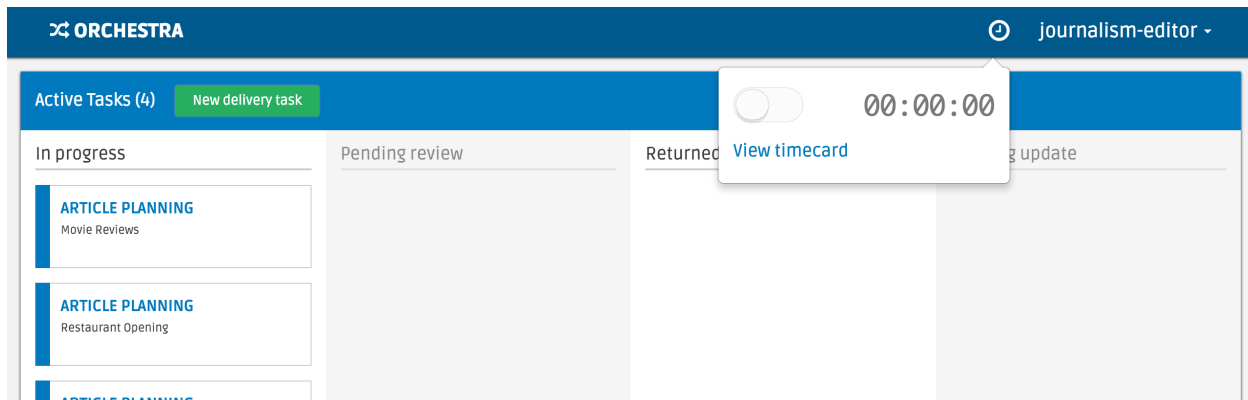
Workers can track time spent on a task using the timer/timecard feature. We provide both a timer widget that workers can turn on to track time, and a timecard page where workers can update and manually create time entries.

4.1.1 Timer

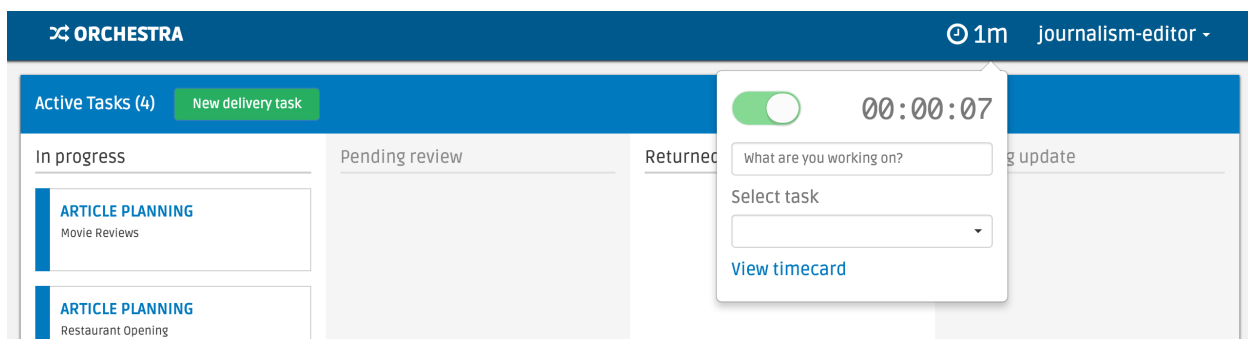
The navigation bar in the task dashboard and task pages has a timer icon.



If you click on the timer icon, a dropdown appears that allows the worker to toggle a timer to track their work time.



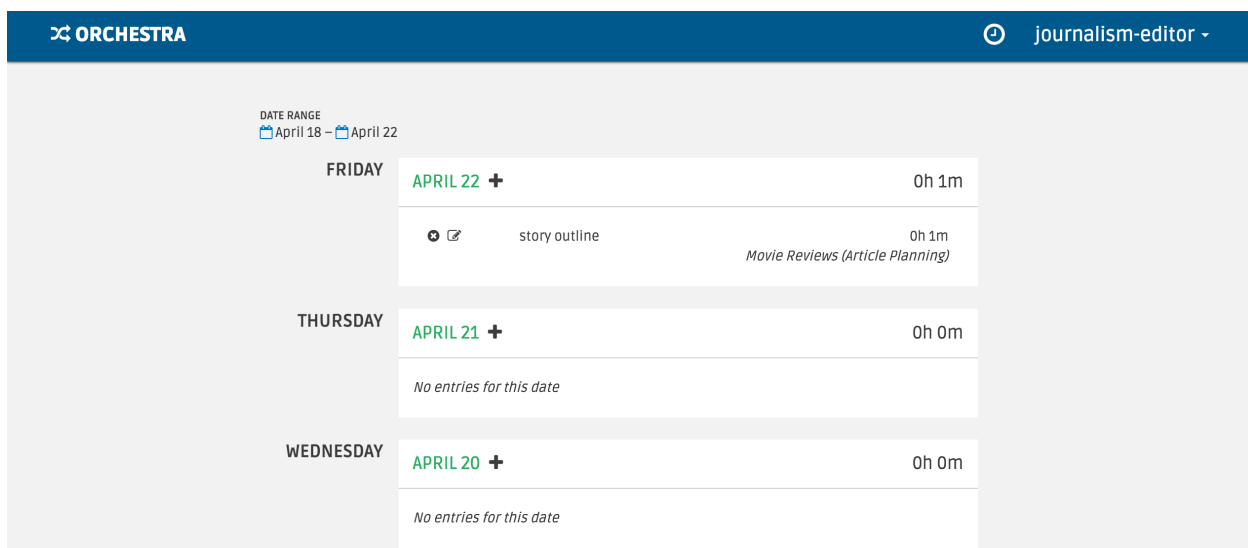
After the timer is started, the worker can add a description and specify the task they are working on. If the timer is started in a task interface, the task field is pre-populated.



When the timer is stopped, a time entry is automatically created for the amount of time worked. A worker can go to the timecard page to edit and manually add time entries.

4.1.2 Timecard

The timecard page by default shows a list of time entries for the current work week, starting on Monday, grouped by date. You can use the date range pickers at the top to change the entries shown.



The time to the right of the date shows the total time worked that day.

A time entry can be manually added (versus automatically added by stopping the timer) by clicking the plus icon next to each date.

The screenshot shows the Orchestra interface with a date range of April 18 to April 22. The 'FRIDAY' section is active, showing 'APRIL 22' with a plus icon and a red 'Invalid Entries' status. A form is open for manual entry with the following fields:

- What did you work on?:** A text input field.
- hours:** A numeric input field with '0' entered.
- minutes:** A numeric input field with '0' entered.
- MOVE TO DATE:** A date selector showing 'April 22'.
- SELECT TASK:** A dropdown menu.
- Buttons:** 'Cancel' and 'Save changes'.

Below the form, a list of tasks is visible:

- story outline** (0h 1m) - *Movie Reviews (Article Planning)*

Each time entry can be edited by clicking the pencil icon, or deleted by clicking the x icon.

If a time entry is missing a description or a task, the missing fields are highlighted and the total hours for the day is hidden.

The screenshot shows the Orchestra interface with the same date range. The 'FRIDAY' section is active, showing 'APRIL 22' with a plus icon and a red 'Invalid Entries' status. A list of time entries is displayed:

- No description** (0h 0m) - *Restaurant Opening (Article Planning)*
- story outline** (0h 1m) - *Movie Reviews (Article Planning)*

4.1.3 Quirks

- Time entries automatically created when the timer is stopped have the date set to the current UTC timestamp date. This means that the date might be different than the date in the worker's time zone. In general, handling time entries across multiple time zones is difficult, and we are still working on a better user experience.
- Time entries automatically created when the timer is stopped have the work time set to the hours:minutes:seconds displayed in the timer. However, the timecard page only shows hours:minutes for time entries for readability. Time entries are rounded up to the minute, and the sum for the day reflects the sum of the *rounded* times, **not** the rounding of the sum of unrounded times.

Below, we'll walk through the various bots available in Orchestra.

5.1 StaffBot

5.1.1 Setup

StaffBot provides a simple way to ask a group of Workers if they would like to work on a particular Task. The goal is reduce the human load when finding a Worker for a Task. StaffBot allows a user to interact with Orchestra via *Slack* or can be configured to automatically find Workers for a Task when it is available by setting up an *Assignment Policy*.

StaffBot works by reaching out to qualified Workers and offering them a Task to work on. Workers can then either accept or reject the task (via Slack or email) and start working on it.

Slack

To interact with StaffBot via Slack, you first need to configure a *Slack Slash Command*. Below is a sample configuration for the `/staffbot` command.

Integration Settings

Command

Commands should be a single word (and can't be blank). Examples: /deploy, /ack

URL

The URL to request when the slash command is run.

Method

POST

Should we do a POST or GET to the above URLs?

Autocomplete help text

You can add this slash command to the autocomplete list and add some usage hints.

Commands matching "/feedb"

Slack

/feedback [your message] Send feedback to Slack

+ /feedback

☒ Show this command in the autocomplete list

Description

A short description of what this slash command does. Example: "Send feedback to Slack"

Usage hint

List any parameters that can be passed. Example: "[your message]"

Once the command is created, copy the token for the command into your project settings as follows: `ORCHESTRA_SLACK_STAFFBOT_TOKEN = 'your-token-here'`. The token is used to authenticated requests sent to the staffbot url. It is important to keep this token secret since otherwise anyone may make a HTTP POST and execute the staffing logic!

In addition to the `ORCHESTRA_SLACK_STAFFBOT_TOKEN` setting, you can further restrict access to the endpoint by specifying the following:

```
STAFFBOT_CONFIG = {
    'allowed_team_ids': ['allowed_team_ids'],
    'allowed_team_domains': ['allowed_domains'],
    'allowed_channel_ids': ['allowed_channel_ids'],
    'allowed_channel_names': ['allowed_channel_names'],
    'allowed_user_ids': ['allowed_user_ids'],
    'allowed_user_names': ['allowed_user_names'],
    'allowed_commands': ['allowed_commands'],
}
```

StaffBot will use the `STAFFBOT_CONFIG` to filter messages that do not match items in the list. If a parameter is not specified, all values are accepted by default.

Once this configuration is complete, you can test it by typing `/staffbot staff <task-id>` where `<task-id>` is an unassigned task id. By default, StaffBot will reach out to Workers who have the appropriate `WorkerCertification` for the task you are staffing, in batches of `settings.ORCHESTRA_STAFFBOT_WORKER_BATCH_SIZE` Workers every `settings.ORCHESTRA_STAFFBOT_BATCH_FREQUENCY` time units (the frequency is a `datetime.timedelta` object).

StaffBot looks at two fields when prioritizing Workers to reach out to when a task is available. The `WorkerCertification.staffbot_enabled` field (True by default) tells StaffBot whether to reach out to a Worker because they have a given certification. If you set it to False, the Worker will still be able to pick up tasks requiring certification (e.g., you can still manually assign tasks to that Worker), but StaffBot will not reach out to them for those tasks. The `WorkerCertification.staffing_priority` integer field (0 by default) helps StaffBot prioritize amongst certified Workers. If Workers have the same `staffing_priority`, StaffBot will prioritize them randomly.

Utility functions

There are several utility functions to help operationalize StaffBot. You should call these through `cron` or some other scheduling utility:

- `orchestra.communication.staffing.send_staffing_requests` looks for tasks that can be staffed and reaches out to the next set of `settings.ORCHESTRA_STAFFBOT_WORKER_BATCH_SIZE` workers every `settings.ORCHESTRA_STAFFBOT_BATCH_FREQUENCY`.
- `orchestra.communication.staffing.remind_workers_about_available_tasks` sends a reminder to any worker who has unclaimed task still available.
- `orchestra.communication.staffing.warn_staffing_team_about_unstaffed_tasks` warns administrators on the internal Slack channel `ORCHESTRA_STAFFBOT_STAFFING_GROUP_ID` about tasks that have not been staffed for more than `ORCHESTRA_STAFFBOT_STAFFING_MIN_TIME`.

Assignment Policy

StaffBot can also automatically staff projects by specifying an Assignment Policy. Orchestra supports custom logic for assigning Workers to tasks, and StaffBot leverages this by asking qualified Workers if they would like to work on a Task as soon as the Task is available. To specify the StaffBot auto-assignment policy, which uses the same logic as the `/staffbot staff` command, add the following to the Step configuration in your `version.json` file. Following the Journalism Workflow Example we have:

```
[...step definition...]
"assignment_policy": {
  "policy_function": {
    "entry_level": {
      "path": "orchestra.bots.assignment_policies.staffbot_autoassign"
    }
  }
},
[...step definition...]
```

Now, for entry-level tasks within the defined step, StaffBot will automatically try to staff this Task. If the task requires review, manual assignment is necessary unless we add a `reviewer` key to the policy function:

```
[...step definition...]
"assignment_policy": {
  "policy_function": {
    "entry_level": {
      "path": "orchestra.bots.assignment_policies.staffbot_autoassign"
    },
    "reviewer": {
      "path": "orchestra.bots.assignment_policies.staffbot_autoassign"
    }
  }
},
[...step definition...]
```

Detailed Description Function

The `detailed_description_function` is used to dynamically describe a Task when StaffBot makes requests to Workers, offering them the opportunity to work on the Task. The function is given a `task_details` dictionary and can be passed extra kwargs as shown below:

```
[...step definition...]
"detailed_description_function": {
  "path": "my_project.orchestra_helpers.get_detailed_description",
  "kwargs": {
    "text": "Task text"
  }
}
[...step definition...]
```

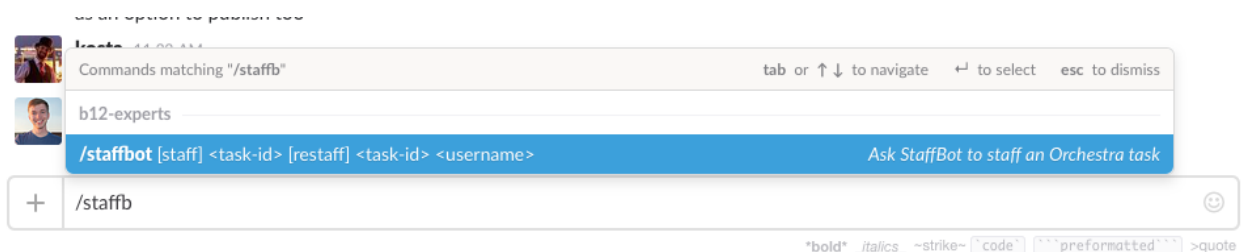
```
# my_project/orchestra_helpers.py

def get_detailed_description(task_details **kwargs):
    return '''A new task is available!
    Find out more about {} at example.com/projects/{}!'''.format(
        kwargs.get('text'), task_details['project']['id'])
```

5.1.2 Usage

Automatic Task Staffing in Orchestra

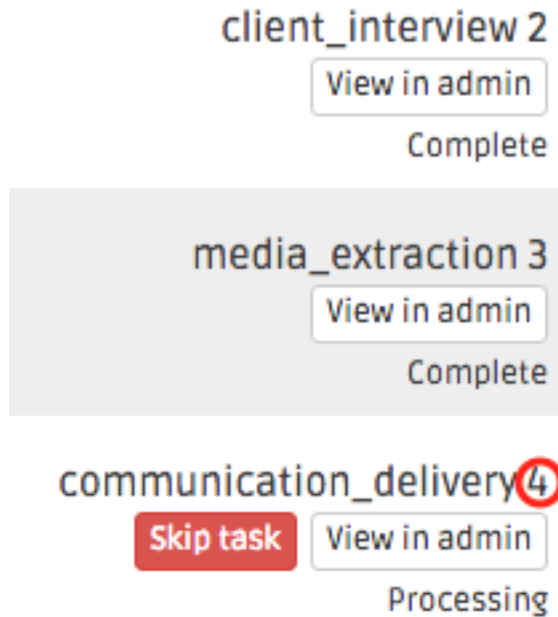
StaffBot allows interaction with Orchestra via Slack to assign or reassign an expert to a task. To use StaffBot, simply type `/staffbot` into your slack window, and will see an autocomplete similar to:



You can send two different commands to StaffBot: 1) `staff`, and 2) `restaff`.

Using the `staff` command

To use the `staff` command, you need to specify a `<task-id>` of a task that is unassigned. You can find the `<task-id>` in the project view (shown below) or from notification emails/Slack messages about a project.



In this example, you have just finished the `client_interview` task and need to add someone to the `communication_delivery` task with id 4 (shown in red), so you can type:

```
/staffbot staff 4
```

Staffbot will then reach out to eligible experts asking them if they would like to work on the task. Once one of them accepts, they will be added to the private Slack channel for the project and can begin working on the task.

If a task has a review step, you can use StaffBot to assign an expert to the review step once the first expert has submitted their work for review.

Using the `restaff` command

You can also use the `restaff` command to offer a task to a different expert. This will be useful if a expert is unable to complete the task. Following the example above, assume that the worker `joshblum` accepted the task 4. To restaff this task you can type:

```
/staffbot restaff 4 joshblum
```

This will offer the task again to eligible experts, and once a new expert accepts, `joshblum` will be removed and the new expert will be added.

5.2 SanityBot

5.2.1 Setup

SanityBot periodically looks at the state of a project and reminds the project team about various things that seem off.

For details and motivation, see the [original project description](#). SanityBot currently warns project team members in the project team's Slack channel.

Project Configuration

To specify which sanity checks to run, and how frequently to run them, update `version.json` for the workflow you are sanity-checking with an optional `sanity_checks` entry. As an example:

```
[...workflow definition...]
"sanity_checks": {
  "sanity_check_function": {
    "path": "path.to.sanity.check.function"
  },
  "check_configurations": {
    "check_slug1": {
      "handlers": [{"type": "slack_project_channel", "message": "<message here>"},
      ↪ "steps": ["step_slug1", ...]],
      "repetition_seconds": 3600
    },
    ...
  },
  ...
}
```

Here's a walkthrough of the configuration above:

- `sanity_check_function` is called periodically and generates `SanityCheck` objects. The function prototype is `def sanity_check_function(project: Project) -> List[SanityCheck]:`.
- `check_configurations` maps `SanityCheck.check_slug` values to a configuration, which consists of a list of handlers and a repetition interval.
- in v1, the only handler is `slack_project_channel`, which messages the team slack project, tagging the experts assigned to the tasks specified by in steps.
- An optional `repetition_seconds` contains the number of seconds to wait before re-issuing/re-handling a `SanityCheck`. If `repetition_seconds` does not appear in the map, that `SanityCheck` is not repeated.

Scheduling function

To operationalize SanityBot, you should call `orchestra.bots.sanitybot.create_and_handle_sanity_checks` through cron or some other scheduling utility. This function will look at all active projects with `sanity_checks` in their workflow definitions, and call the appropriate `sanity_check_function` to trigger sanity checks.

CHAPTER 6

Our Motivation

B12 has open sourced Orchestra as part of our goal to build a brighter future of work.

We are a startup based in NYC that is passionate about improving how people do creative and analytical work. We have a strong team of engineers and designers who have worked extensively on systems that help people work productively online.

Beyond focusing on profit, we believe that the products and experiences we design should be considerate of their greater social context and impact. To stay true to these values, we are in the process of becoming a [B-certified corporation](#).

How to contribute to Orchestra

So you want to get involved in developing Orchestra. Great! We're excited to have your support. This document lays down a few guidelines to help the whole process run smoothly.

7.1 Getting involved

First, if you find a bug in the code or documentation, check out our [open issues](#) and [pull requests](#) to see if we're already aware of the problem. Also feel free to reach out to us on [gitter](#) to answer questions at any time, or [subscribe to the Orchestra mailing list](#) for longer conversations.

If you've uncovered something new, please [create an issue](#) describing the problem. If you've written code that fixes an issue, [create a pull request](#) (PR) so it's easy for us to incorporate your changes.

7.2 Setting up for Development

We have a `.editorconfig` specified in the top level providing editor defaults for our code style. We recommend to install an [EditorConfig plugin](#) so your editor automatically adheres to our style :).

We recommend using a [virtualenv](#) to install the required packages in `requirements.txt`. In addition, we use [Gulp](#) as a frontend build system. To build the frontend resources you can run `make gulp_build` once [npm](#) is installed.

7.3 Development Workflow

Github provides a nice [overview on how to create a pull request](#).

Some general rules to follow:

- Do your work in a [fork](#) of the Orchestra repo.
- Create a branch for each feature/bug in Orchestra that you're working on. These branches are often called "feature" or "topic" branches.

- Use your feature branch in the pull request. Any changes that you push to your feature branch will automatically be shown in the pull request.
- Keep your pull requests as small as possible. Large pull requests are hard to review. Try to break up your changes into self-contained and incremental pull requests, if need be, and reference dependent pull requests, e.g. “This pull request builds on request #92. Please review #92 first.”
- Include unit tests with your pull request. We love tests and use [CircleCI](#) to check every pull request and commit. Check out our tests in `orchestra/tests` to see examples of how to write unit tests. Before submitting a PR, make sure that running `make test` from the root directory of the repository succeeds.
- Additionally, we try to maintain high [code coverage](#). Aim for 100% for every new file you create!
- Once you submit a PR, you’ll get feedback on your code, sometimes asking for a few rounds of changes before your PR is accepted. After each round of comments, make changes accordingly, then squash all changes for that round into a single commit with a name like ‘changes round 0’.
- After your PR is accepted, you should squash all of your changes into a single commit before we can merge them into the main codebase.
- If your feature branch is not based off the latest master, you will be asked to rebase it before it is merged. This ensures that the commit history is linear, which makes the commit history easier to read.
- How do you rebase on to master, you ask? After [syncing your fork against the Orchestra master](#), run:

```
git checkout master
git pull
git checkout your-branch
git rebase master
```

- How do you squash changes, you ask? Easy:

```
git log
<find the commit hash that happened immediately before your first commit>
git reset --soft $THAT_COMMIT_HASH$
git commit -am "A commit message that summarizes all of your changes."
git push -f origin your-branch
```

- Remember to reference any issues that your pull request fixes in the commit message, for example ‘Fixes #12’. This will ensure that the issue is automatically closed when the PR is merged.

7.4 Quick Style Guide

We generally stick to [PEP8](#) for our coding style, use spaces for indenting, and make sure to wrap lines at 79 characters.

We have a linter built in to our test infrastructure, so `make test` won’t succeed until the code is cleaned up. To run the linter standalone, just run `make lint`. Of course, sometimes you’ll write code that will never make the linter happy (for example, URL strings longer than 80 characters). In that case, just add a `# noqa` comment to the end of the line to tell the linter to ignore it. But use this sparingly!

When working on frontend resources, we use [Gulp](#) as a frontend build system. This means that after making any changes to frontend resources, you must run `make gulp_build` to include the modified resources. This moves resources to the `build` folder, compiling `scss` and linting your `javascript`.

For stylesheets we only compile `scss` files so if your file is at `orchestra/common/static/common/scss/example.scss`, to include it in an HTML file you should write `{% static 'common/css/example.css' %} ">` as the static file path.

When including angular templates, we wrap references to static files with the function `$static(static_url_path)`. The `$static` function is defined in the base template, and for development simply returns the url it is given. The purpose is to decouple static file storage from the Django path, so if you host your static files on a CDN, you can simply override this function and put the appropriate urls.

8.1 Client API

Endpoints for communicating with Orchestra.

All requests must be signed using [HTTP signatures](#):

```
from httpsig.requests_auth import HTTPSignatureAuth

auth = HTTPSignatureAuth(key_id=settings.ORCHESTRA_PROJECT_API_KEY,
                          secret=settings.ORCHESTRA_PROJECT_API_SECRET,
                          algorithm='hmac-sha256')
response = requests.get('https://www.example.com/orchestra/api/project/create_project
↪', auth=auth)
```

POST /orchestra/api/project/create_project

Creates a project with the given data and returns its ID.

Query Parameters

- **task_class** – One of *real* or *training* to specify the task class type.
- **workflow_slug** – The slug corresponding to the desired project’s workflow.
- **workflow_version_slug** – The slug corresponding to the desired version of the workflow.
- **description** – A short description of the project.
- **priority** – An integer describing the priority of the project, with higher numbers describing a greater priority.
- **project_data** – Other miscellaneous data with which to initialize the project.

Example response:

```
{
  "project_id": 123,
}
```

POST /orchestra/api/project/project_information

Retrieve detailed information about a given project.

Query Parameters

- **project_id** – The ID for the desired project.

Example response:

```
{
  "project": {
    "id": 123,
    "short_description": "Project Description",
    "priority": 10,
    "team_messages_url": "http://review.document.url",
    "task_class": 1,
    "project_data": {
      "sample_data_item": "sample_data_value_new"
    },
    "workflow_slug": "sample_workflow_slug",
    "workflow_version_slug": "v1",
    "start_datetime": "2015-09-23T20:16:02.667288Z"
  },
  "steps": [
    ["sample_step_slug", "Sample step description"]
  ],
  "tasks": {
    "sample_step_slug": {
      "id": 456,
      "project": 123,
      "status": "Processing",
      "step_slug": "sample_step_slug",
      "latest_data": {
        "sample_data_item": "sample_data_value_new"
      },
      "assignments": [
        {
          "id": 558,
          "iterations": [
            {
              "id": 92134,
              "start_datetime": "2015-09-20T12:02:14.214553",
              "end_datetime": "2015-09-23T20:16:15.821171",
              "submitted_data": {
                "sample_data_item": "sample_data_value_old",
              },
              "status": 'Requested Review'
            }
          ]
        },
        {
          "worker": "sample_worker_username",
          "task": 456,
          "in_progress_task_data": {
            "sample_data_item": "sample_data_value_new"
          },
          "status": "Processing",
        }
      ]
    }
  }
}
```

```

        "start_datetime": "2015-09-23T20:16:17.355291Z"
      }
    ]
  }
}

```

GET /orchestra/api/project/workflow_types

Return all stored workflows and their versions.

Example response:

```

{
  "workflows": {
    "journalism": {
      "name": "Journalism Workflow",
      "versions": {
        "v1": {
          "name": "Journalism Workflow Version 1",
          "description": "Create polished newspaper articles from_
↪scratch."
        },
        "v2": {
          "name": "Journalism Workflow Version 2",
          "description": "Create polished newspaper articles from_
↪scratch."
        }
      }
    },
    "simple_workflow": {
      "name": "Simple Workflow",
      "versions": {
        "v1": {
          "name": "Simple Workflow Version 1",
          "description": "Crawl a web page for an image and rate it."
        }
      }
    }
  }
}

```


CHAPTER 9

Core Reference

Core reference still in progress.

Contents

- *Core Reference*
 - *Models*
 - *Task Lifecycle*

9.1 Models

9.2 Task Lifecycle

CHAPTER 10

Indices and tables

- `genindex`
- `modindex`
- `search`

HTTP Routing Table

/orchestra

GET /orchestra/api/project/workflow_types,
39
POST /orchestra/api/project/create_project,
37
POST /orchestra/api/project/project_information,
38