Orchestra

Release 0.1.0

September 29, 2015

Contents

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

CHAPTER 1

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 virual 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/unlimitedlabs/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. 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/unlimitedlabs/orchestra/stable/example_project/example_project
mv orchestra_settings.py your_project/your_project
```

Next, edit the `orchestra_settings.py` file:

- Add `'simple_workflow'` to `settings.INSTALLED_APPS` 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 and use Google Apps and Slack to help communicate with expert workers.

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 barebones example file with `wget https://raw.githubusercontent.com/unlimitedlabs/orchestra/stable/example_project/example_pr`

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')),

# 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 an initial worker account set up to try out example workflows. We've provided several fixtures relevant for running our examples, which you can load with `python manage.py loaddata <FIXTURE_NAME>`:

- 'demo_admin': creates a single admin account (username: `admin`, password: `admin`) suitable for logging in to the admin and managing the database.
- 'demo_worker': creates a single worker (username: `demo`, password: `demo`) suitable for running the *simple demo workflow*.

- 'journalism_workflow': creates a number of accounts with certifications suitable for running our more complicated journalism workflow.

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.

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! Logging in as the demo user we set up above 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 `INSTALLED_APPS` setting following the previous section.
- Pull down the script into your project's root directory (`your_project`, next to `manage.py`):

```
wget https://raw.githubusercontent.com/unlimitedlabs/orchestra/stable/example_project/interactiv
```

- Run the script:

```
python interactive_simple_workflow_demo.py
```

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.

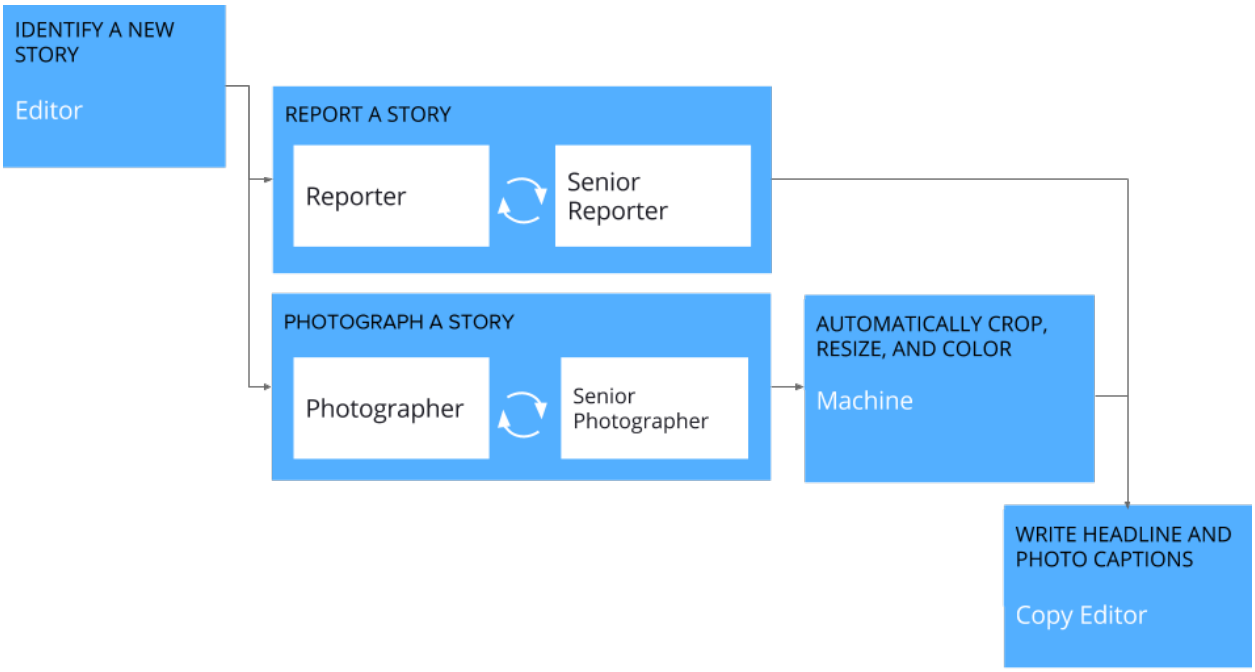
If you don't want to go to the trouble of running the script yourself, take a look at the `transcript of expected output`.

CHAPTER 2

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.

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 fixtures installed by running (if you haven't already) `python manage.py loaddata journalism_workflow`. 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.

Finally, we've included a management script to start and monitor the workflow. Download it to the directory of your project next to `manage.py` with:

```
wget https://raw.githubusercontent.com/unlimitedlabs/orchestra/stable/example_project/journalism_work
```

2.2.2 Start the workflow

To start the workflow:

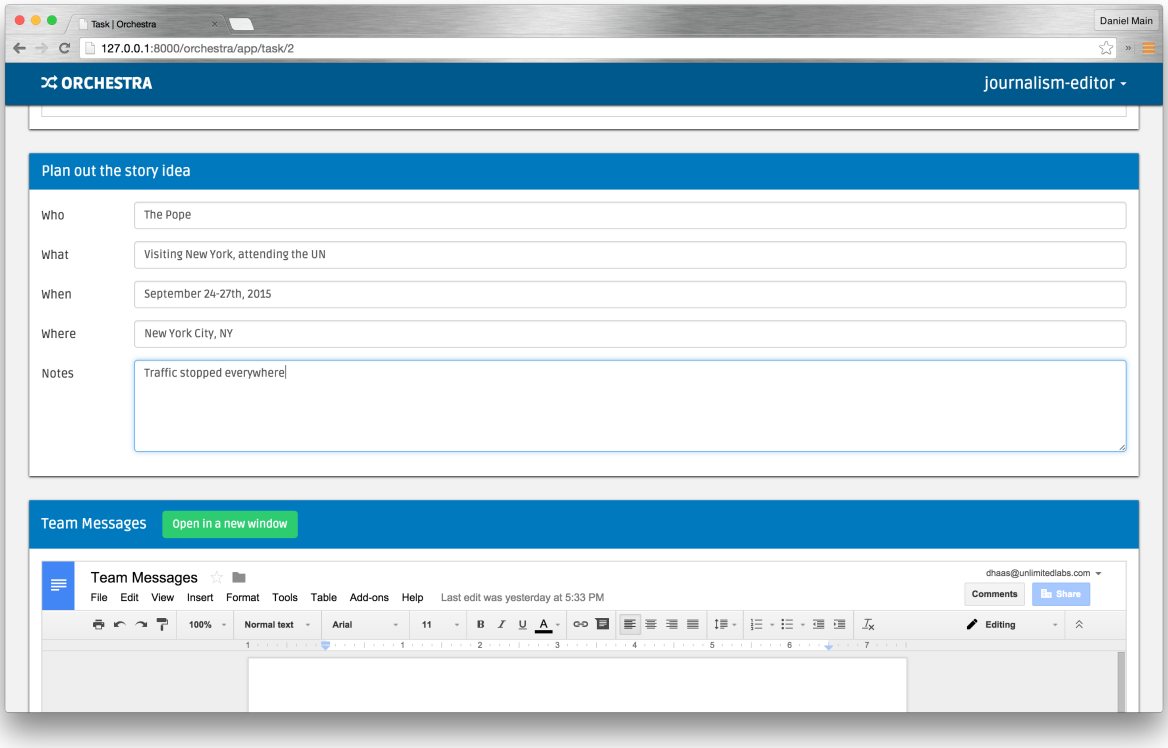
- Make sure Orchestra is running with `python manage.py runserver`.
- In another tab, run:

  python 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.

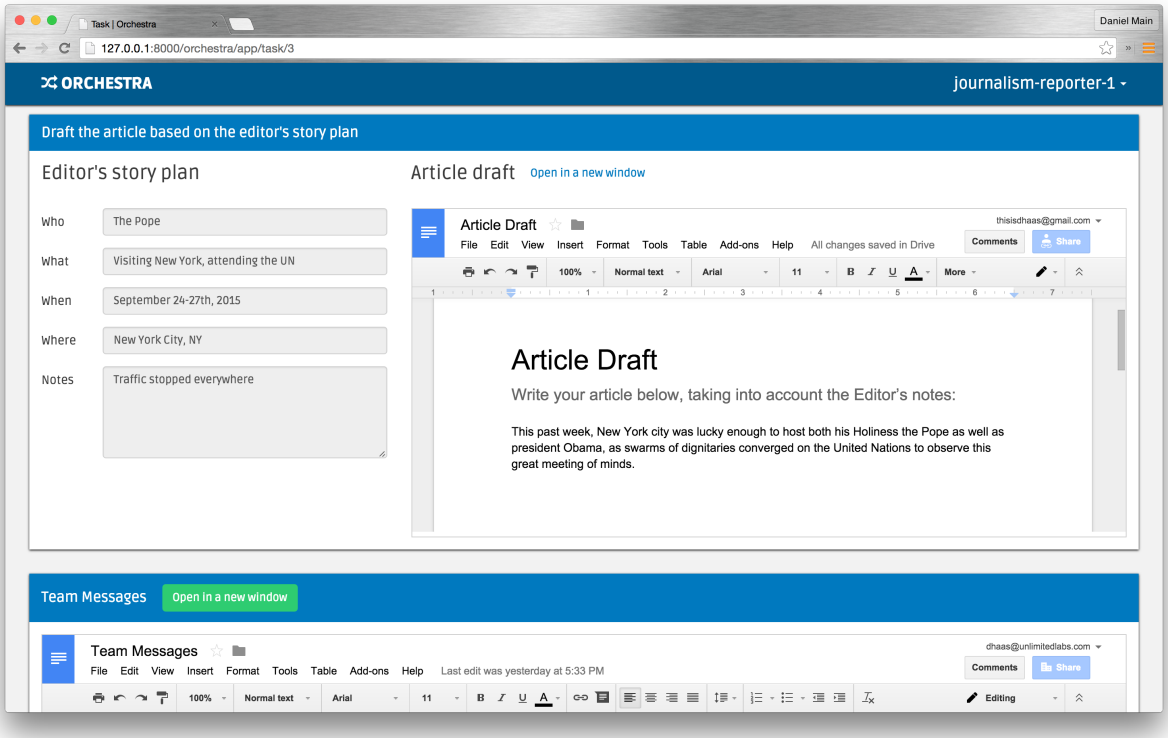
2.2.3 Complete the steps

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:



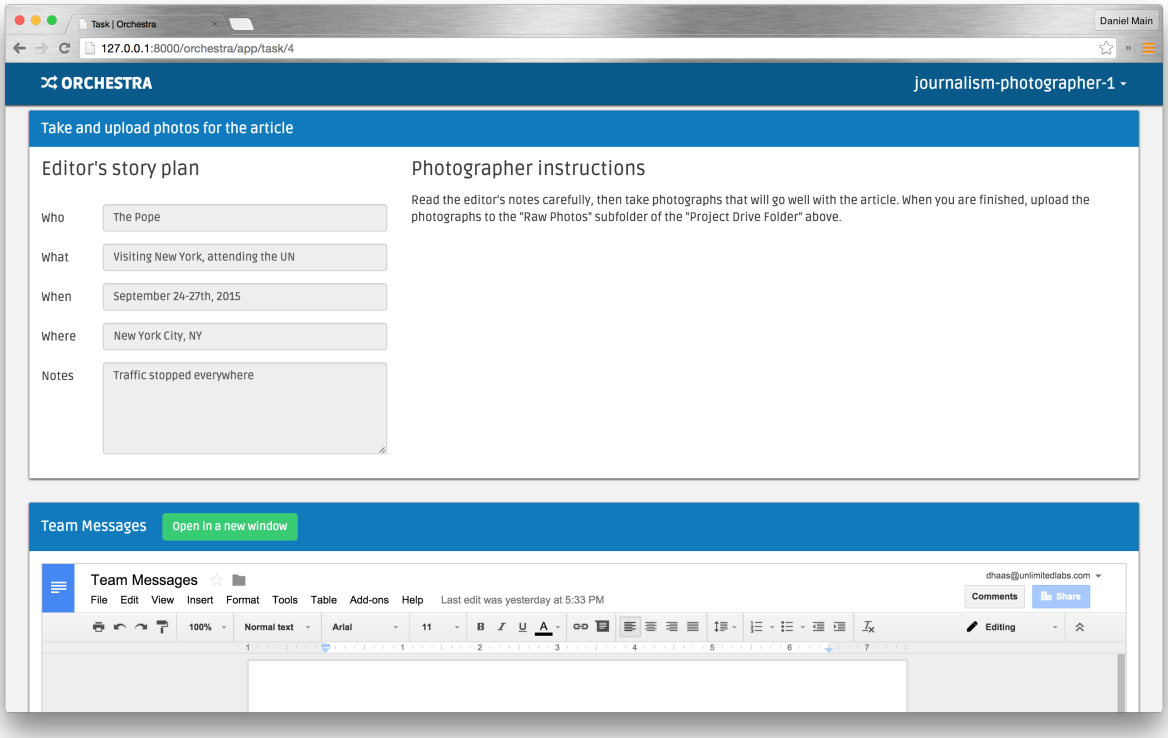
Fill out the high-level story idea and submit the task.

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.



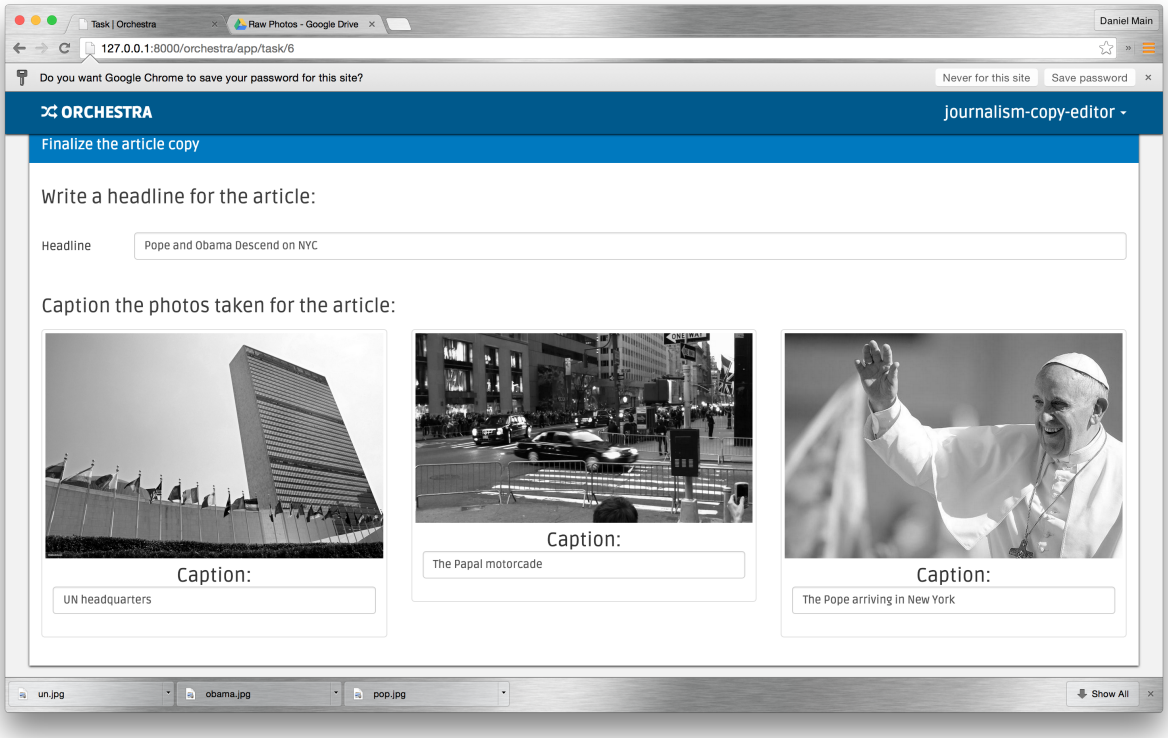
When you submit, you'll note that the task appears in the 'Awaiting Review' section. That's your cue to log in as `journalism-reporter-2` and review the work. Once you're satisfied with it, accept it.

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



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:



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 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.py. It declaratively defines the steps listed above, in programmatic form.

First, we define the workflow with a name and short description:

```
from orchestra.workflow import Workflow

journalism_workflow = Workflow(
  slug='journalism',
  name='Journalism Workflow',
  description='Create polished newspaper articles from scratch.',
)
```

Then, we add 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:

```
from orchestra.workflow import Step

# A reporter researches and drafts an article based on the editor's idea
reporter_step = Step(
  slug='reporting',
  name='Reporting',
  description='Research and draft the article text',
  worker_type=Step.WorkerType.HUMAN,
  creation_depends_on=[editor_step],
  required_certifications=['reporter'],
  user_interface={
    'javascript_includes': [
      '/static/journalism_workflow/reporter/js/modules.js',
      '/static/journalism_workflow/reporter/js/controllers.js',
      '/static/journalism_workflow/reporter/js/directives.js',
    ],
    'stylesheet_includes': [],
    'angular_module': 'journalism_workflow.reporter.module',
    'angular_directive': 'reporter',
  },

  # A senior reporter should review the article text.
  review_policy={
    'policy': 'sampled_review',
    'rate': 1,         # review all tasks
    'max_reviews': 1   # exactly once
  },
)
journalism_workflow.add_step(reporter_step)
```

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). Finally, we add the step to our workflow with `journalism_workflow.add_step(reporter_step)`.

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:

```
photo_adjustment_step = Step(
  slug='photo_adjustment',
  name='Photo Adjustment',
  description='Automatically crop and rescale images',
  worker_type=Step.WorkerType.MACHINE,
  creation_depends_on=[photographer_step],
  function=autoadjust_photos,
)
journalism_workflow.add_step(photo_adjustment_step)
```

The basic arguments are similar, but we specify the step type as `Step.WorkerType.MACHINE`, and insead 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 a JSON object that looked like this:

```
user_interface={
  'javascript_includes': [
    '/static/journalism_workflow/editor/js/modules.js',
    '/static/journalism_workflow/editor/js/controllers.js',
    '/static/journalism_workflow/editor/js/directives.js',
  ],
  'stylesheet_includes': [],
  'angular_module': 'journalism_workflow.editor.module',
  'angular_directive': 'editor',
},
```

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.findPrerequisite($scope.taskAssignment, step_slug)`, which will get the taskAssignment for the previous step called `step_slug`.

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' }
```

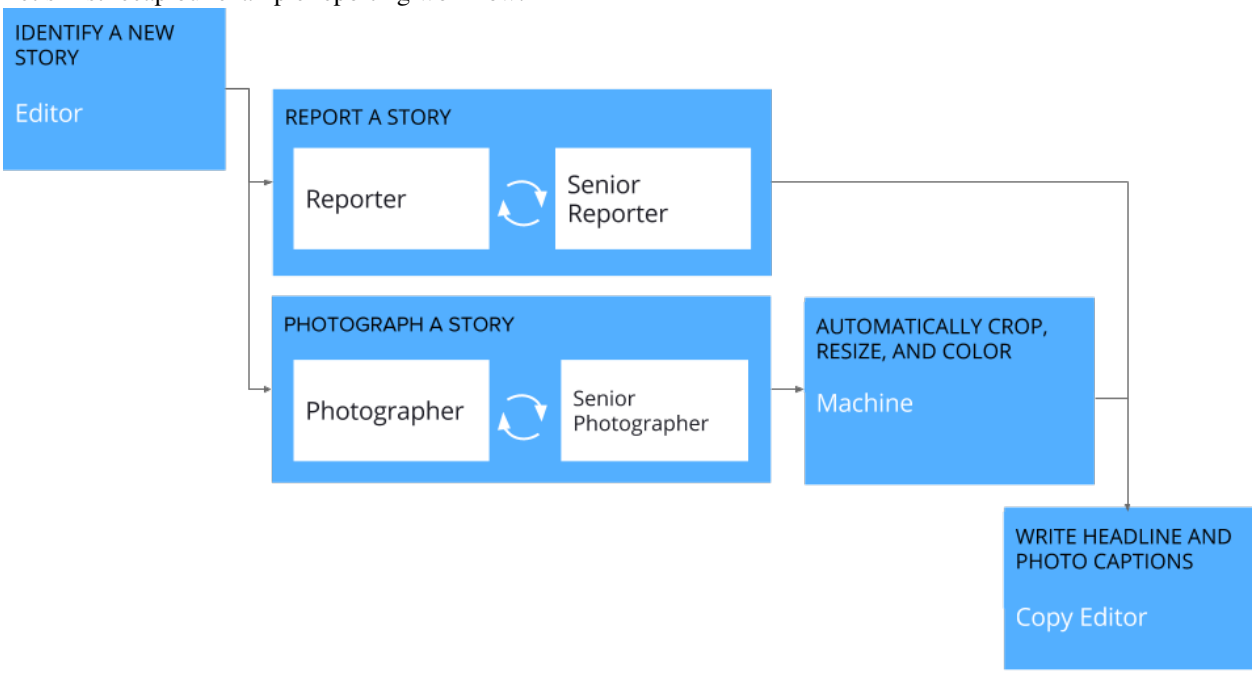
It takes two arguments, a python dictionary containing global project data and a python dictionary containing state from all prerequisite workflow steps (and their prerequisites, and so on). The function can do whatever it likes, and returns a JSON-encodable dictionary containing state that should be 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/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.

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

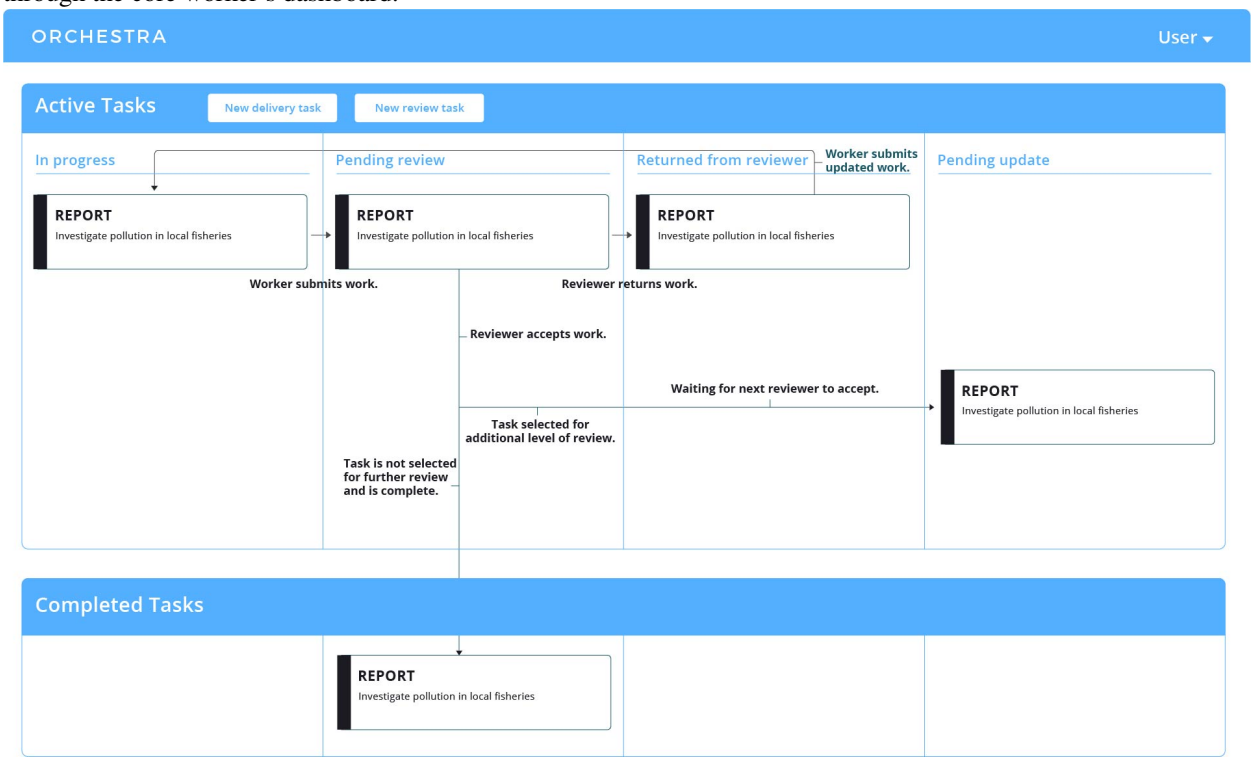
- Certifications allow experts to work on tasks they're great at.
- Experts can work toward all sorts of certifications, picking up practice tasks to build experience.
  - *Joseph is a solid reporter but needs a little more practice as a photographer—let's give him some simple tasks so he can improve!*
- 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.

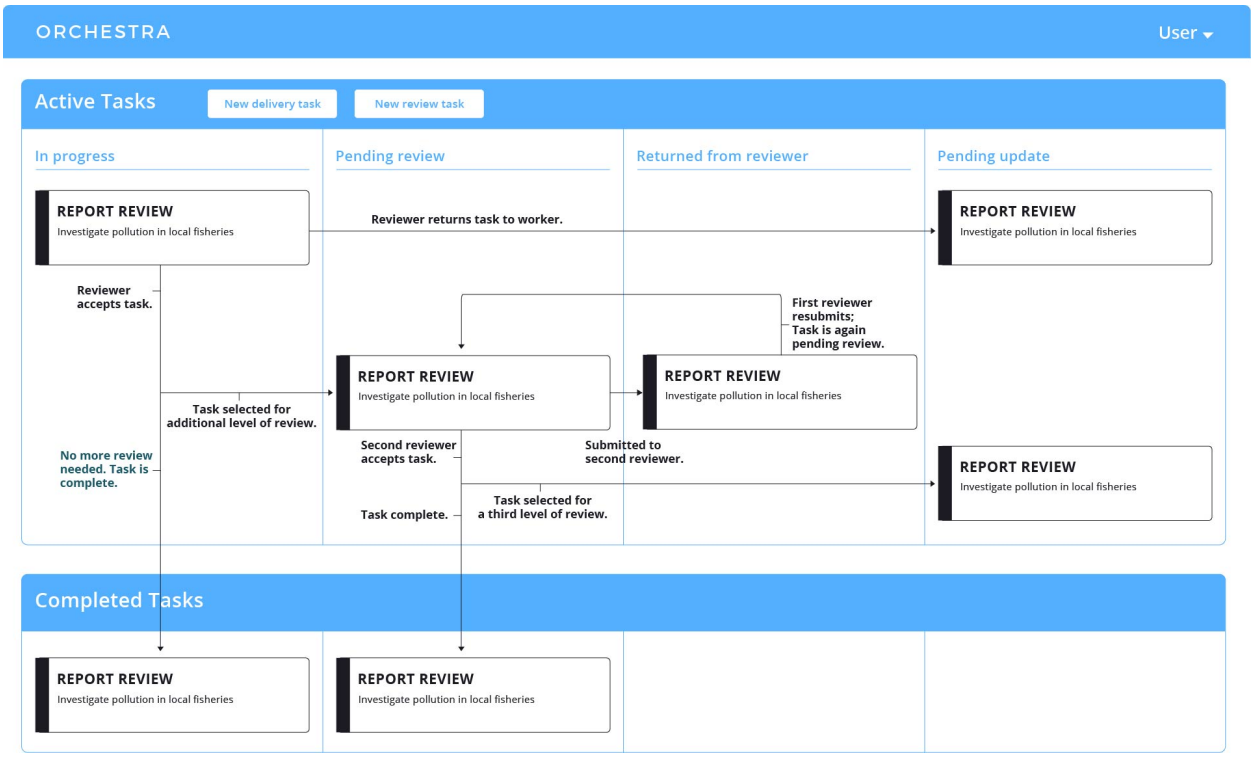
3.5.1 Core Expert

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.


ORCHESTRA
User
Active Tasks
New delivery task
New review task
In progress
Pending review
Returned from reviewer
Pending update
REPORT REVIEW
Investigate pollution in local fisheries
Reviewer returns task to worker.
REPORT REVIEW
Investigate pollution in local fisheries
Reviewer accepts task.
First reviewer resubmits; Task is again pending review.
REPORT REVIEW
Investigate pollution in local fisheries
REPORT REVIEW
Investigate pollution in local fisheries
Task selected for additional level of review.
No more review needed. Task is complete.
Second reviewer accepts task.
Submitted to second reviewer.
REPORT REVIEW
Investigate pollution in local fisheries
Task complete.
Task selected for a third level of review.
Completed Tasks
REPORT REVIEW
Investigate pollution in local fisheries
REPORT REVIEW
Investigate pollution in local fisheries

CHAPTER 4

Our Motivation

Unlimited Labs 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.

CHAPTER 5

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.

5.1 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.

5.2 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. To verify that your changes are well-covered by tests, run `make test_coverage`, which will run the tests, then print out percent coverage for each file in Orchestra. 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.

5.3 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!

CHAPTER 6

API Reference

6.1 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

- **project_id** – The ID for the desired project.
- **task_class** – One of *real* or *training* to specify the task class type.
- **workflow_slug** – The slug corresponding to the desired project's 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.
- **review_document_url** – Team messages Google Doc for 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,
        "review_document_url": "http://review.document.url",
        "task_class": 1,
        "project_data": {
            "sample_data_item": "sample_data_value_new"
        },
        "workflow_slug": "sample_workflow_slug",
        "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,
                    "snapshots": {
                        "__version": 1,
                        "snapshots": [
                            {
                                "work_time_seconds": 3660,
                                "datetime": "2015-09-23T20:16:15.821171",
                                "data": {
                                    "sample_data_item": "sample_data_value_old",
                                    "__version": 1
                                },
                                "type": 0
                            }
                        ]
                    },
                    "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.

Example response:

```
{
  "workflows": {
                  "reporting": "A sample workflow for the newsroom."
                }
}
```

CHAPTER 7

Core Reference

Core reference still in progress.

Contents

7.1 Workflow

class `orchestra.workflow.`**`Step`**(***kwargs*)

Steps represent nodes on a workflow execution graph.

`slug`
str

Unique identifier for the step.

`name`
str

Human-readable name for the step.

`description`
str

A longer description of the step.

`worker_type`
orchestra.workflow.Step.WorkerType

Indicates whether the policy is for a human or machine.

`creation_depends_on`
[str]

Slugs for steps on which this step's creation depends.

`submission_depends_on`
[str]

Slugs for steps on which this step's submission depends.

function
function

Function to execute during step. Should be present only for machine tasks

required_certifications
[str]

Slugs for certifications required for a worker to pick up tasks based on this step.

class WorkerType
Specifies whether step is performed by human or machine

class orchestra.workflow.**Workflow**(**kwargs)
Workflows represent execution graphs of human and machine steps.

slug
str

Unique identifier for the workflow.

name
str

Human-readable name for the workflow.

description
str

A longer description of the workflow.

steps
dict

Steps comprising the workflow.

add_step(*step*)
Add *step* to the workflow.

Parameters **step** (orchestra.workflow.Step) – The step to be added.

Returns None

Raises

- orchestra.core.errors.InvalidSlugValue –

  Step slug should have fewer than 200 characters.

- orchestra.core.errors.SlugUniquenessError –

  Step slug has already been used in this workflow.

get_human_steps()
Return steps from the workflow with a human *worker_type*.

Parameters **None** –

Returns

steps –

Steps from the workflow with a human *worker_type*..

Return type [orchestra.workflow.Step]

get_step(*slug*)
Return the specified step from the workflow.

Parameters **slug** (*str*) – The slug of the desired step.

Returns

step –

The specified step from the workflow.

Return type *orchestra.workflow.Step*

get_step_slugs()
Return all step slugs for the workflow.

Parameters **None** –

Returns

slugs –

List of step slugs for the workflow.

Return type [str]

get_steps()
Return all steps for the workflow.

Parameters **None** –

Returns

steps –

List of steps for the workflow.

Return type [orchestra.workflow.Step]

orchestra.workflow.**get_default_policy**(*worker_type*, *policy_name*)
Return the default value for a specified policy.

Parameters

- **worker_type** (orchestra.workflow.Step.WorkerType) – Indicates whether the policy is for a human or machine.
- **policy_name** (*str*) – The specified policy identifier.

Returns

default_policy –

A dict containing the default policy for the worker type and policy name specified.

Return type dict

orchestra.workflow.**get_step_choices**()
Return step data formatted as *choices* for a model field.

Parameters **None** –

Returns

step_choices –

A tuple of tuples containing each step slug and human-readable name.

Return type tuple

orchestra.workflow.**get_workflow_by_slug**(*slug*)
Return the workflow specified by *slug*.

Parameters **slug** (*str*) – The slug of the desired workflow.

Returns

workflow –

The corresponding workflow object.

Return type *orchestra.workflow.Workflow*

orchestra.workflow.**get_workflow_choices**()
Return workflow data formatted as *choices* for a model field.

Parameters **None** –

Returns

workflow_choices –

A tuple of tuples containing each workflow slug and human-readable name.

Return type tuple

orchestra.workflow.**get_workflows**()
Return all stored workflows.

Parameters **None** –

Returns

workflows –

A dict of all workflows keyed by slug.

Return type [orchestra.workflow.Workflow]

7.2 Models

class orchestra.models.**Certification**(**args*, ***kwargs*)
Certifications allow workers to perform different types of tasks.

slug
str

Unique identifier for the certification.

name
str

Human-readable name for the certification.

description
str

A longer description of the certification.

required_certifications
[orchestra.models.Certification]

Prerequisite certifications for possessing this one.

class orchestra.models.**Project**(**args*, ***kwargs*)
A project is a collection of tasks representing a workflow.

status
orchestra.models.Project.Status

Represents whether the project is being actively worked on.

workflow_slug
str

Identifies the workflow that the project represents.

start_datetime
datetime.datetime

The time the project was created.

priority
int

Represents the relative priority of the project.

task_class
int

Represents whether the project is a worker training exercise or a deliverable project.

review_document_url
str

The URL for the review document to be passed between workers and reviwers for the project's tasks.

slack_group_id
str

The project's internal Slack group ID if Slack integration is enabled.

class orchestra.models.**Task**(**args*, ***kwargs*)
A task is a cohesive unit of work representing a workflow step.

step_slug
str

Identifies the step that the project represents.

project
orchestra.models.Project

The project to which the task belongs.

status
orchestra.models.Task.Status

Represents the task's stage within its lifecycle.

class orchestra.models.**TaskAssignment**(**args*, ***kwargs*)
A task assignment is a worker's assignment for a given task.

start_datetime
datetime.datetime

The time the project was created.

worker
orchestra.models.Worker

The worker to whom the given task is assigned.

task
orchestra.models.Task

The given task for the task assignment.

status
orchestra.models.Project.Status

Represents whether the assignment is currently being worked on.

assignment_counter
int

Identifies the level of the assignment in the given task's review hierarchy (i.e., 0 represents an entry-level worker, 1 represents the task's first reviewer, etc.).

in_progress_task_data
str

A JSON blob containing the worker's input data for the task assignment.

snapshots
str

A JSON blob containing saved snapshots of previous data from the task assignment.

Constraints: *task* and *assignment_counter* are taken to be unique_together.

Task assignments for machine-type tasks cannot have a *worker*, while those for human-type tasks must have one.

class orchestra.models.**Worker**(**args*, ***kwargs*)
Workers are human experts within the Orchestra ecosystem.

user
django.contrib.auth.models.User

Django user whom the worker represents.

start_datetime
datetime.datetime

The time the worker was created.

slack_username
str

The worker's Slack username if Slack integration is enabled.

class orchestra.models.**WorkerCertification**(**args*, ***kwargs*)
A WorkerCertification maps a worker to a certification they possess.

certification
orchestra.models.Certification

Certification belonging to the corresponding worker.

worker
orchestra.models.Worker

Worker possessing the given certification.

task_class
orchestra.models.WorkerCertification.TaskClass

Represents whether the worker is in training for the given certification or prepared to work on real tasks.

role
orchestra.models.WorkerCertification.Role

Represents whether the worker is an entry-level or review worker for the given certification.

Constraints: *certification*, *worker*, *task_class*, and *role* are taken to be unique_together.

Worker must possess an entry-level WorkerCertification before obtaining a reviewer one.

7.3 Task Lifecycle

orchestra.utils.task_lifecycle.**assign_task**(*worker_id*, *task_id*)
Return a given task after assigning or reassigning it to the specified worker.

Parameters

- **worker_id** (*int*) – The ID of the worker to be assigned.
- **task_id** (*int*) – The ID of the task to be assigned.

Returns

task –

The newly assigned task.

Return type *orchestra.models.Task*

Raises

- orchestra.core.errors.TaskAssignmentError –

  The specified worker is already assigned to the given task or the task status is not compatible with new assignment.

- orchestra.core.errors.WorkerCertificationError –

  The specified worker is not certified for the given task.

orchestra.utils.task_lifecycle.**create_subsequent_tasks**(*project*)
Create tasks for a given project whose dependencies have been completed.

Parameters **project** (orchestra.models.Project) – The project for which to create tasks.

Returns

project –

The modified project object.

Return type *orchestra.models.Project*

orchestra.utils.task_lifecycle.**end_project**(*project_id*)
Mark the specified project and its component tasks as aborted.

Parameters **project_id** (*int*) – The ID of the project to abort.

Returns None

orchestra.utils.task_lifecycle.**get_new_task_assignment**(*worker*, *task_status*)
Check if new task assignment is available for the provided worker and task status; if so, assign the task to the worker and return the assignment.

Parameters

- **worker** (orchestra.models.Worker) – The worker submitting the task.
- **task_status** (*orchestra.models.Task.Status*) – The status of the desired new task assignment.

Returns

assignment –

The newly created task assignment.

Return type *orchestra.models.TaskAssignment*

Raises

- `orchestra.core.errors.WorkerCertificationError` –

  No human tasks are available for the given task status except those for which the worker is not certified.

- `orchestra.core.errors.NoTaskAvailable` –

  No human tasks are available for the given task status.

`orchestra.utils.task_lifecycle.`**`get_next_task_status`**(*task*, *snapshot_type*)

Given current task status and snapshot type provide new task status. If the second level reviewer rejects a task then initial reviewer cannot reject it further down, but must fix and submit the task.

Parameters

- **task** (orchestra.models.Task) – The specified task object.
- **task_status** (*orchestra.models.TaskAssignment.SnapshotType*) – The action to take upon task submission (e.g., SUBMIT, ACCEPT, REJECT).

Returns

next_status –

The next status of *task*, once the *snapshot_type* action has been completed.

Return type orchestra.models.Task.Status

Raises `orchestra.core.errors.IllegalTaskSubmission` –

The *snapshot_type* action cannot be taken for the task in its current status.

`orchestra.utils.task_lifecycle.`**`get_task_assignment_details`**(*task_assignment*)

Return various information about the specified task assignment.

Parameters **task_assignment** (orchestra.models.TaskAssignment) – The specified task assignment.

Returns

task_assignment_details –

Information about the specified task assignment.

Return type dict

`orchestra.utils.task_lifecycle.`**`get_task_details`**(*task_id*)

Return various information about the specified task.

Parameters **task_id** (*int*) – The ID of the desired task.

Returns **task_details** – Information about the specified task.

Return type dict

orchestra.utils.task_lifecycle.**get_task_overview_for_worker**(*task_id*, *worker*)
Get information about *task* and its assignment for *worker*.

Parameters

- **task_id** (*int*) – The ID of the desired task object.
- **worker** (orchestra.models.Worker) – The specified worker object.

Returns

task_assignment_details –

Information about *task* and its assignment for *worker*.

Return type dict

orchestra.utils.task_lifecycle.**previously_completed_task_data**(*task*)
Returns a dict mapping task prerequisites onto their latest task assignment information. The dict is of the form: {'previous-slug': {task_assignment_data}, ...}

Parameters **task** (orchestra.models.Task) – The specified task object.

Returns

prerequisites –

A dict mapping task prerequisites onto their latest task assignment information..

Return type dict

orchestra.utils.task_lifecycle.**save_task**(*task_id*, *task_data*, *worker*)
Save the latest data to the database for a task assignment, overwriting previously saved data.

Parameters

- **task_id** (*int*) – The ID of the task to save.
- **task_data** (*str*) – A JSON blob of task data to commit to the database.
- **worker** (orchestra.models.Worker) – The worker saving the task.

Returns None

Raises orchestra.core.errors.TaskAssignmentError –

The provided worker is not assigned to the given task or the assignment is in a non-processing state.

orchestra.utils.task_lifecycle.**submit_task**(*task_id*, *task_data*, *snapshot_type*, *worker*, *work_time_seconds*)
Returns a dict mapping task prerequisites onto their latest task assignment information. The dict is of the form: {'previous-slug': {task_assignment_data}, ...}

Parameters

- **task_id** (*int*) – The ID of the task to submit.
- **task_data** (*str*) – A JSON blob of task data to submit.
- **snapshot_type** (*orchestra.models.TaskAssignment.SnapshotType*) – The action to take upon task submission (e.g., SUBMIT, ACCEPT, REJECT).
- **worker** (orchestra.models.Worker) – The worker submitting the task.
- **work_time_seconds** (*int*) – The time taken by the worker on the latest iteration of their task assignment.

Returns

task –

The modified task object.

Return type *orchestra.models.Task*

Raises

- `orchestra.core.errors.IllegalTaskSubmission` –

  Submission prerequisites for the task are incomplete or the assignment is in a non-processing state.

- `orchestra.core.errors.TaskAssignmentError` –

  Worker belongs to more than one assignment for the given task.

- `orchestra.core.errors.TaskStatusError` –

  Task has already been completed.

`orchestra.utils.task_lifecycle.`**`task_history_details`**(*task_id*)

Return assignment details for a specified task.

Parameters **`task_id`** (*int*) – The ID of the desired task object.

Returns

details –

A dictionary containing the current task assignment and an in-order list of related task assignments.

Return type dict

`orchestra.utils.task_lifecycle.`**`tasks_assigned_to_worker`**(*worker*)

Get all the tasks associated with *worker*.

Parameters **`worker`** (orchestra.models.Worker) – The specified worker object.

Returns

tasks_assigned –

A dict with information about the worker's tasks, used in displaying the Orchestra dashboard.

Return type dict

`orchestra.utils.task_lifecycle.`**`update_related_assignment_status`**(*task*, *assignment_counter*, *data*)

Copy data to a specified task assignment and mark it as processing.

Parameters

- **`task`** (orchestra.models.Task) – The task whose assignments will be updated.
- **`assignment_counter`** (*int*) – The index of the assignment to be updated.
- **`data`** (*str*) – A JSON blob containing data to add to the assignment.

Returns None

`orchestra.utils.task_lifecycle.`**`worker_assigned_to_max_tasks`**(*worker*)

Check whether worker is assigned to the maximum number of tasks.

Parameters **`worker`** (orchestra.models.Worker) – The specified worker object.

Returns

assigned_to_max_tasks –

True if worker is assigned to the maximum number of tasks.

Return type bool

orchestra.utils.task_lifecycle.**worker_assigned_to_rejected_task**(*worker*)
Check whether worker is assigned to a task that has been rejected.

Parameters **worker** (orchestra.models.Worker) – The specified worker object.

Returns

assigned_to_rejected_task –

True if worker is assigned to a task that has been rejected.

Return type bool

orchestra.utils.task_lifecycle.**worker_has_reviewer_status**(*worker*, *task_class=1*)
Check whether worker is a reviewer for any certification for a given task class.

Parameters

- **worker** (orchestra.models.Worker) – The specified worker object.
- **task_class** (*orchestra.models.WorkerCertification.TaskClass*) – The specified task class.

Returns

has_reviwer_status –

True if worker is a reviewer for any certification for a given task class.

Return type bool

orchestra.utils.task_properties.**all_workers**(*task*)
Return all workers for a given task.

Parameters **task** (orchestra.models.Task) – The specified task object.

Returns

all_workers –

A list of all workers involved with *task*.

Return type [orchestra.models.Worker]

orchestra.utils.task_properties.**assignment_history**(*task*)
Return all assignments for *task* ordered by *assignment_counter*.

Parameters **task** (orchestra.models.Task) – The specified task object.

Returns

assignment_history –

All assignments for *task* ordered by *assignment_counter*.

Return type [orchestra.models.TaskAssignment]

orchestra.utils.task_properties.**current_assignment**(*task*)
Return the in-progress assignment for *task*.

Parameters **task** (orchestra.models.Task) – The specified task object.

Returns

current_assignment –

The in-progress assignment for *task*.

Return type *orchestra.models.TaskAssignment*

orchestra.utils.task_properties.**is_worker_assigned_to_task**(*worker*, *task*)

Check if specified worker is assigned to the given task.

Parameters

- **worker** (orchestra.models.Worker) – The specified worker object.
- **task** (orchestra.models.Task) – The given task object.

Returns

worker_assigned_to_task –

True if worker has existing assignment for the given task.

Return type bool

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

O

/orchestra

R

S

T

U

W