Django Task API Documentation

Release 1.0.0

Nikolas Stevenson-Molnar

Jun 22, 2021

Contents

1 What does it look like? 3

2 Next Steps 5

2.1 Getting Started 5

2.2 Guides 9

2.3 References 13

3 Indices and tables 15

Django Task API lets you quickly write background tasks for your Django project and easily call then using the provided REST API, or the included JavaScript library.

CHAPTER 1

What does it look like?

Tasks are defined as classes with typed input and output parameters, and a *run* function with the task implementation, to be called by a worker processes.

```
from task_api.tasks import Task
from task_api.params import ListParameter, NumberParameter

class SumNumbers(Task):
    name = 'sum'

    inputs = {
        'numbers': ListParameter(NumberParameter())
    }

    outputs = {
        'sum': NumberParameter()
    }

    def run(self, numbers):
        return sum(numbers)
```

Tasks are easily called and monitored in front-end code using the included JavaScript API. The API supports both promises (will Polyfill for older browsers) and traditional callbacks.

```
<script src="{% static 'django-task-api.min.js' %}"></script>

<script type="text/javascript">
    function sumTask(numbers) {
        TaskAPI
            .run('sum', {'numbers': numbers})
            .then(function(json) {
                console.log('Sum: ' + json.outputs.sum)
            })
    }
</script>
```

CHAPTER 2

Next Steps

- *Getting Started*
- GitHub

2.1 Getting Started

2.1.1 Install Django Task API

Install the Python library with pip:

```
$ pip install django-task-api
```

Django Task API is compatible and tested with Python versions 3.6 through 3.9, and with Django versions 2.2, 3.1, and 3.2.

2.1.2 Set up Celery

By default, Django Task API uses Celery to manage background tasks. If you're not already using Celery, follow the First steps with Django document to configure Celery for your project.

2.1.3 Create a task

Create a module in your Django app called *background.py* and add a task class to it:

Listing 1: myapp/background.py

```
from time import sleep

from task_api.params import IntParameter
```

(continues on next page)

(continued from previous page)

```
from task_api.tasks import Task

class WaitTask(Task):
    name = 'wait'

    inputs = {
        'seconds': IntParameter(required=False)
    }

    def run(seconds=10):
        self.set_target(10)
        self.set_progress(0)

        for _ in range(seconds):
            sleep(1)
            self.inc_progress(1)
```

The `WaitTask` task accepts an integer value and counts toward that number, one second at a time. Since it updates its progress, we'll be able to monitor it from the front-end.

2.1.4 Configure settings & URLS

Edit `settings.py`, add the `task_api` app, and add tasks to `TASK_API_BACKGROUND_TASKS`.

Listing 2: settings.py

```
INSTALLED_APPS += [
    'task_api'
]

TASK_API_BACKGROUND_TASKS = ['myapp.background.WaitTask']
```

With the app added to settings, run Django's migrate command:

```
$ python manage.py migrate
```

We'll also need a URL route to the task API:

Listing 3: urls.py

```
from django.conf.urls import url
from django.urls import include

urlpatterns = [
    url('^', include('task_api.urls'))
]
```

2.1.5 Add front-end Java Script

Django Task API includes a JS API for starting and monitoring background tasks. If you're using Django to manage your static files, then you can include the library using the `{% static %}` template tag. You can also install the JavaScript library from npm. For purposes of this walk through, let's create a template with some simple HTML and JavaScript to start and monitor a task:

Listing 4: myapp/templates/task.html

```
{% load static %}

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Task API Example</title>

    <script src="{% static "django-task-api.min.js" %}"></script>
    <script type="text/javascript">
        function startTask() {
            TaskAPI.run('wait', {'seconds': 10}, function(json) {
                if (json.target === null || json.progress === null) {
                    return
                }
                document.getElementById('status').innerHTML = 'Progress: ' + json.
↪progress + ' / ' + json.target
            }).then(function() {
                document.getElementById('button').disabled = false
            })

            document.getElementById('button').disabled = true
        }
    </script>
    <style type="text/css">
        .content {
            position: fixed;
            top: 25%;
            left: 50%;
            transform: translate(-50%, -50%);
        }

        .content button {
            margin-top: 5px;
            width: 100px;
        }

        .center {
            text-align: center;
        }
    </style>
</head>
<body>
    <div class="content">
        <div id="status">Click "Run" to start the task.</div>
        <div class="center"><button onclick="startTask()" id="button">Run</button></
↪div>
    </div>
</body>
</html>
```

This gives the user a “Run” button, which when clicked, starts the task defined earlier. As the task counts towards its target, the UI updates to show the current progress.

To finish everything out, we need to add a URL route for this template:

Listing 5: myapp/urls.py

```
from django.conf.urls import url
from django.urls import include
from django.views.generic import TemplateView

url_patterns = [
    # ... other URL patterns
    url('^$', TemplateView.as_view(template_name='example.html'))
]
```

If you haven't already, add your app to your project `urls.py` file:

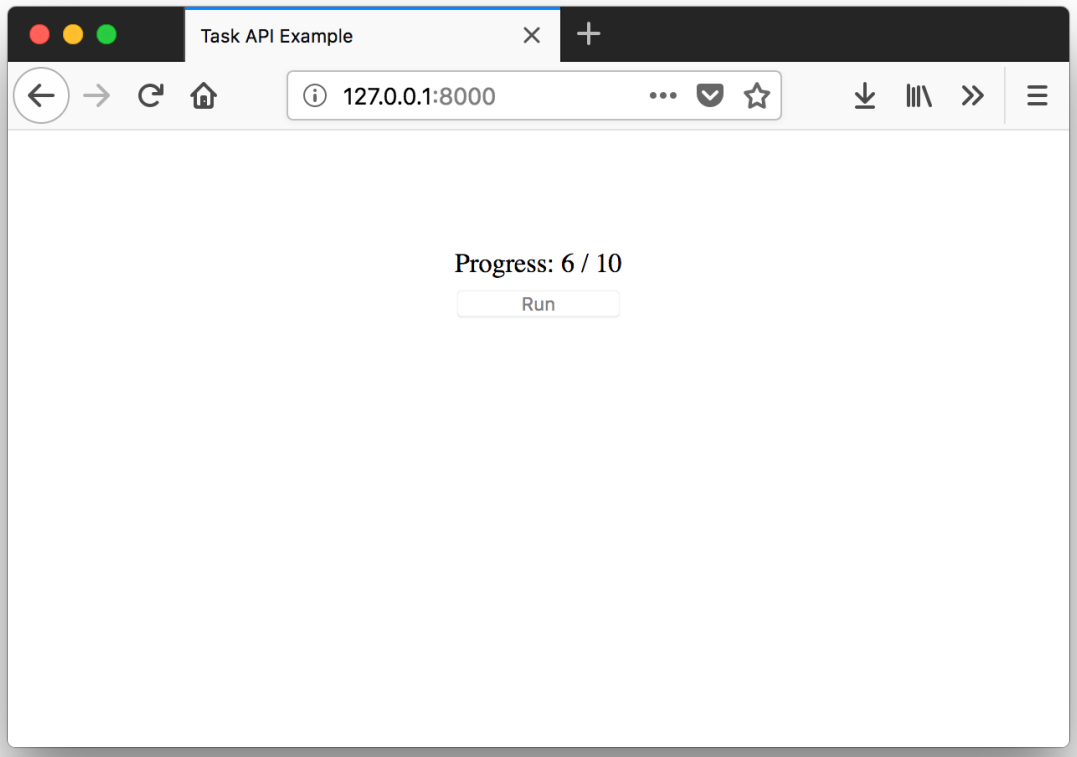
Listing 6: urls.py

```
from django.conf.urls import url
from django.urls import include

urlpatterns = [
    url('^', include('task_api.urls')),
    url('^myapp', include('myapp.urls'))
]
```

2.1.6 Try it out

Your new task should be ready to go. Make sure that the Django debug server and Celery worker process are both running, then open the page in your browser. Click "Run" and watch your task progress.



2.1.7 Problems?

If you run into problems, take a look at the fully-functional example project here.

2.2 Guides

2.2.1 Tasks

Task classes implement code to execute on a background process, and specify input and output names and types. Task classes also provide methods for communicating progress and messages as the task runs.

Here's a basic task:

```
from task_api.tasks import Task
from task_api.params import IntParameter, StringParameter

class MyTask(Task):
    name = 'my-task'

    inputs = {
        'name':  StringParameter(),
```

(continues on next page)

(continued from previous page)

```
        'count': IntParameter(required=False)
    }

    outputs = {'output': StringParameter()}

def run(name, count=5):
    # Do stuff here
    return 'Some output'
```

First, note that all tasks must subclass `task_api.tasks.Task`. All tasks must specify a `name` property, and can optionally specify inputs and outputs. Parameters define input and output types: the determine how values are converted from JSON to Python objects, and they return errors to the client on invalid values.

```
class MyTask(Task):
    name = 'my-task'  # The name used to call this task from the API

    inputs = { ... } # Task inputs
    outputs = { ... } # Task outputs
```

The task logic should be defined in a *run()* method, which accepts inputs named in accordance with the `inputs`. Any inputs marked with `required=False` must specify a default in the function declaration. The above example has a required `name` input and an optional `count` input. Thus the run method looks like this:

```
def run(name, count=5):
    ...
```

Since the task has a single output, the return value of `run()` will be used as the output value. For tasks with multiple outputs, `run()` should return a dictionary with keys matching the `outputs` property.

In order for tasks to be available through the API, they must be added to `TASK_API_BACKGROUND_TASKS` in `settings.py`.

```
TASK_API_BACKGROUND_TASKS = [
    'myapp.background.MyTask'
]
```

Parameters

The `inputs` and `outputs` properties determine which inputs the a client may and must provide to the API, and what outputs it receives back. Parameter types are determined by parameter classes. Parameter provide type conversion and enforcement. E.g., an `IntParameter` given a string input of `"6"` will yield an integer `6`, whereas that same parameter given a string input of `"not a number"` will raise an exception.

When specifying `inputs`, all parameter constructors can optionally be given a `required=` argument. Inputs are required by default, an API call with missing inputs will be rejected. Any parameters with `required=False` can be omitted by the client. Some parameters may have additional optional or required arguments. For example, the `ListParameter` requires an argument with the type of elements in the list:

```
inputs = [
    'items': ListParameter(StringParameter())
]
```

`run()` Method

Your task logic all goes in the `run()` method of your task class. `run()` should accept arguments corresponding your `inputs` property. Django Task API will process each of parameters sent by the client and convert then to Python objects as as specified in `inputs` (e.g., strings, ints, list, etc.). Any optional parameters must be given default values:

```
inputs = [
    'must_have': StringParameter(required=True),
    'nice_to_have': StringParameter(required=False)
]

def run(self, must_have, nice_to_have=None):
    ...
```

Parameters are required by default, so `required=True` isn't strictly necessary.

`run()` must return values in accordance with parameters specified in the `outputs` property. If `outputs` only specifies a single parameter, than `run()` may simply return that parameter:

```
outputs = [
    'out': StringParameter()
]

def run(self):
    return 'Foo'
```

If the task has multiple outputs, then `run()` must return a dictionary of output values:

```
outputs = [
    'out': StringParameter()
    'count': IntParameter()
]

def run(self):
    return {
        'out': 'Foo,
        'count': 5
    }
```

Progress & Messages

Tasks can communicate with front-end code in two ways: updating progress, and adding messages. To use progress, first set a target, then increment progress regularly throughout the task. Target and progress should both be integers.

```
def run(self):
    with open('lines.txt', 'r') as f:
        f.seek(0, os.SEEK_END)
        self.set_target(f.tell())
        f.seek(0, os.SEEK_SET)

        for line in f:
            process_line(line)
            self.set_progress(f.tell())
```

Authorization & Permissions

Django Task API uses Django Rest Framework (DRF) to define its API view. You can specify DRF authorization and permissions classes to be used by the Task API with the `TASK_API_AUTHENTICATION_CLASSES` and `TASK_API_PERMISSIONS_CLASSES` settings.

```
TASK_API_AUTHENTICATION_CLASSES = ['rest_framework.authentication.
↪SessionAuthentication']
TASK_API_PERMISSIONS_CLASSES = ['rest_framework.permissions.IsAuthenticated']
```

The above restricts the Task API to logged in users.

2.2.2 JavaScript Client Library

The built-in JavaScript Client Library lets you run a task, and let's you know when when it's complete using a callback or a promise. You can also use status callbacks to track or communicate task status, such as updating a gauge.

Installing

There are two ways to get the JS library:

1. Include the script from your static files into an HTML template:

```
{% load static %}

{# ... #}

<script src="{% static 'django-task-api.min.js' %}"></script>
```

2. Install with `npm`. If you are using a build process for your front-end code, this is a good option:

```
$ npm i --save django-task-api
```

Basic usage

Start tasks with the *run()* function. How you use this depends on how you installed the library. If you included `django-task-api.min.js` in your HTML template, you can access the library through the `TaskAPI` global variable:

```
TaskAPI.run(/* ... */)
```

Or if you installed the library using `npm`, you can use `import` or `require`:

```
import tasks from 'django-task-api'
// Or
var tasks = require('django-task-api')

tasks.run(/* ... */)
```

The `run()` function takes two required inputs: the task name, and the task inputs. It also accepts an optional progress callback function.

```
TaskAPI.run('my-task', {text: 'Hi'}, function(json) {
    console.log('Status: ' + json.status)
})
```

You can use `run()` with either a promise, or success and error callbacks to receive notification upon a completed task.

```
// Using promise
TaskAPI
    .run('my-task', {text: 'Hi'})
    .then(function(json) { console.log('Success!' )})
    .catch(function(json) { console.log('Error.' )})

// Using callbacks
TaskAPI.run('my-task', {text: 'Hi!'}, null, function(json) {
    console.log('Success!')
}, function(json) {
    console.log('Error!')
})
```

Task API URL

By default, the JS client will use */tasks/* as the base URL for the Django Task API. If you choose to publish the API at a different endpoint, you can change the JS client options to reflect this. For example, if you add the Django Task API urls under `/task-api/`, the full base URL will become `/task-api/tasks`:

Listing 7: urls.py

```
urlpatterns = [
    url(r'^task-api/', include('task_api.urls'))
]
```

Then you can set the `baseURL` option to match:

```
TaskAPI.options.baseURL = '/task-api/tasks/'
TaskAPI.run(/* ... */)
```

Override CSRF names

Django's built-in CSRF protection is a valuable security tool. By default, the Djanto Task API JS library will work with the default CSRF cookie and header names. If you want to change either of those, you can update the JS library to match:

```
TaskAPI.options.csrfCookieName = 'csrf-tok'
TaskAPI.options.csrfHeaderName = 'X-CSRF'
TaskAPI.run(/* ... */)
```

2.3 References

2.3.1 Settings

TASK_API_AUTHENTICATION_CLASSES

Authentication classes to be used by the Task API view. These should be Django Rest Framework (DRF) authentication classes. See DRF's Authentication guide for more info.

TASK_API_BACKEND

The task backend class. Defaults to `task_api.backends.celery.CeleryBackend`, which is the only built-in backend.

TASK_API_BACKGROUND_CLASSES

A list of `Task` classes to be provided by the API. For example:

```
TASK_API_BACKGROUND_CLASSES = [
    'myapp.background.MyTask'
]
```

You can also use class objects directly. For example:

```
from myapp.background import MyTask

TASK_API_BACKGROUND_CLASSES = [MyTask]
```

Note that the above example uses the class definition, *not* a class instance.

TASK_API_PERMISSIONS_CLASSES

Permissions classes to be used by the Task API view. These should be Django Rest Framework (DRF) permissions classes. See DRF's Permissions guide for more info.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search