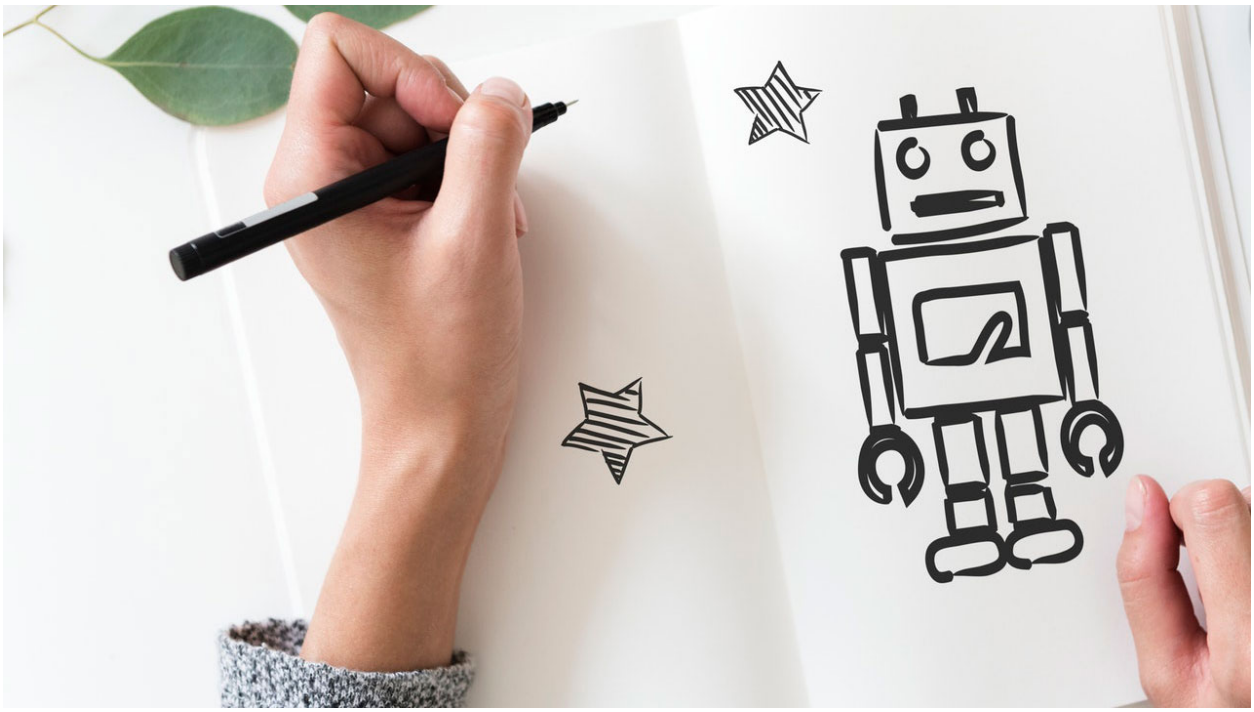
Joeflow

Sep 08, 2020

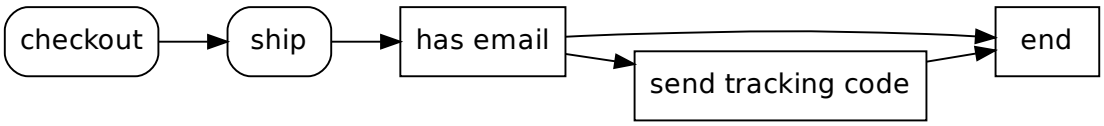
Contents:

The lean workflow automation framework for machines with heart.



Joeflow is a free workflow automation framework designed to bring simplicity to complex workflows. Joeflow written in Python based on the world famous Django web framework.

Here is a little sample of what a workflow or process written with joeflow may look like:




```
from django.core.mail import send_mail
from jowflow.models import Workflow
from joeflow import tasks


class Shipment(Workflow):
    email = models.EmailField(blank=True)
    shipping_address = models.TextField()
    tracking_code = models.TextField()

class ShippingWorkflow(Shipment):
    checkout = tasks.StartView(fields=["shipping_address", "email"])

    ship = tasks.UpdateView(fields=["tracking_code"])

    def has_email(self, task):
        if self.email:
```

(continues on next page)

(continued from previous page)

```
            return [self.send_tracking_code]

    def send_tracking_code(self, task):
        send_mail(
            subject="Your tracking code",
            message=self.tracking_code,
            from_email=None,
            recipient_list=[self.email],
        )

    def end(self, task):
        pass

    edges = [
        (checkout, ship),
        (ship, has_email),
        (has_email, send_tracking_code),
        (has_email, end),
        (send_tracking_code, end),
    ]

    class Meta:
        proxy = True
```

CHAPTER 1

Design Principles

1.1 Common sense is better than convention

Joeflow does not follow any academic modeling notation developed by a poor PhD student who actually never worked a day in their life. Businesses are already complex which is why Joeflow is rather simple. There are only two types of tasks – human & machine – as well as edges to connect them. It's so simple a toddler (or your CEO) could design a workflow.

1.2 Lean Automation (breaking the rules)

Things don't always go according to plan especially when humans are involved. Even the best workflow can't cover all possible edge cases. Joeflow embraces that fact. It allows uses to interrupt a workflow at any given point and modify it's current state. All while tracking all changes. This allows developers to automate the main cases and users handle manually exceptions. This allows you businesses to ship prototypes and MVPs of workflows. Improvements can be shipped in multiple iterations without disrupting the business.

1.3 People

Joeflow is build with all users in mind. Managers should be able to develop better workflows. Users should able to interact with the tasks every single day. And developers should be able to rapidly develop and test new features.

1.4 Free

Joeflow is open source and collaboratively developed by industry leaders in automation and digital innovation.

CHAPTER 2

All Contents

2.1 Tutorial

The following tutorial should give you a quick overview on how to write a workflow, integrate it into your Django application and write robust and automated tests.

Before we get started make you you have the package installed. Simply install the PyPi package...

```
python3 -m pip install "joeflow[reversion,dramatiq,celery]"
```

...and add `joeflow` to the `INSTALLED_APP` setting. You will also need to have celery setup.

See also:

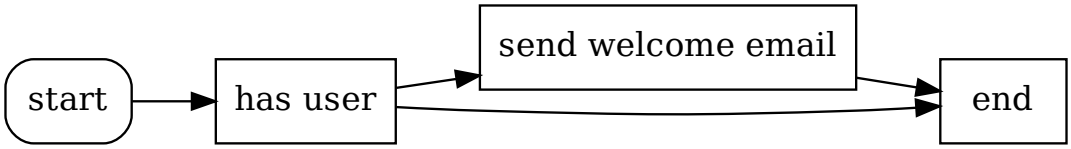
If you don't have celery setup yet, simply follow their setup instructions for Django projects.

https://celery.readthedocs.io/en/latest/django/first-steps-with-django.html

Once the setup is completed you can get started writing your first workflow!

2.1.1 Writing your first Workflow

As an example we will create a simple workflow that sends a welcome email to a user. A human selects the user (or leaves the field blank). If the user is set a welcome emails is being sent. If the user is blank no email will be send and the workflow will end right way.



Let's start with the data structure or workflow state. We need a model that can store a user. Like so:

```
from django.conf import settings
from joeflow.models import Workflow


class WelcomeWorkflowState(Workflow):
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        blank=True, null=True,
    )
```

We keep the model abstract. The abstract model will make it easier to separate state from behavior and therefore easier to read for your fellow developers.

Next we add the behavior:

```
from joeflow import tasks

from . import models


class WelcomeWorkflow(models.WelcomeWorkflowState):
    start = tasks.StartView(fields=['user'])

    def has_user(self, task):
        if self.object.user_id is None:
            return [self.end]
        else:
            return [self.send_welcome_email]

    def send_welcome_email(self, task):
        self.object.user.email_user(
            subject='Welcome',
            message='Hello %s!' % self.object.user.get_short_name(),
        )

    def end(self, task):
        pass

    edges = (
        (start, has_user),
        (has_user, end),
        (has_user, send_welcome_email),
        (send_welcome_email, end),
```

(continues on next page)

(continued from previous page)

```
    )

    class Meta:
        proxy = True
```

We have the tasks `start`, `has_user` `send_welcome_email` and `end` on the top and define all the edges on the bottom. Edges are defined by a set of tuples. Edges are directed, meaning the first item in the tuple is the start tasks and the second item the end tasks.

Note that the `has_user` task has two different return values. A task can return a list of following or child tasks. This is how your workflow can take different paths. If there is no return value, it will simply follow all possible edges defined in `edges`.

The `end` task, does not really do anything. It is also not really needed. It is just added for readability and could be omitted. Any tasks that does not have a child task defined in `edges` or returns an empty list is considered a workflow end.

To make your workflow available to users you will need to add the workflow URLs to your `urls.py`:

```
from django.urls import path, include

from . import workflows

urlpatterns = [
    # ...
    path('welcome/', include(workflows.WelcomeWorkflow.urls())),
]
```

This will add URLs for all human tasks as well as a detail view and manual override view. We will get to the last one later.

That it all the heavy lifting is done. In the next part of tutorial you will learn *how to integrate the tasks into your templates*.

2.1.2 Creating templates

Your human tasks, like your *start* view will need a template. The template name is similar as it is for a `CreateView` but with more options. Default template names are:

```
app_name/welcomeworkflow_start.html
app_name/welcomeworkflow_form.html
app_name/workflow_form.html
```

Django will search for a template precisely that order. This allows you to create a base template for all human tasks but also override override them individually should that be needed.

Following the example please create a file named `app_name/workflow_form.html` in your template folder. The `app_nam` should be replaced by the application name in which you crated your Welcome workflow. Now fill the file with a simple form template:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Welcome Workflow</title>
</head>
```

(continues on next page)

(continued from previous page)

```
<body>
  <form method="POST">
    {% csrf_token %}
    {{ form }}
    <input type="submit">
  </form>
</body>
</html>
```

Of course you can make it prettier, but this will work.

Besides the tasks a workflow comes with two more views by default. A workflow detail view and a view to manually override the current workflow state.

The manual override view will also use the `workflow_form.html` template that you have already created. You can of course create a more specific template. Django will search for templates in the following order:

```
app_name/welcomeworkflow_override.html
app_name/workflow_override.html
app_name/welcomeworkflow_form.html
app_name/workflow_form.html
```

Last but not least you will need a template for the workflow detail view. You don't really need to add anything here, but lets add a little information to make your workflow feel more alive.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Welcome Workflow</title>
</head>
<body>
  {{ object.get_instance_graph_svg }}
  <h1>{{ object }}</h1>
  <table>
    <thead>
    <tr>
      <th>id</th>
      <th>task name</th>
      <th>completed</th>
    </tr>
    </thead>
    <tbody>
    {% for task in object.task_set.all %}
    <tr>
      <td>{{ task.pk }}</td>
      <td>
        {% if task.get_absolute_url %}
        <a href="{{ task.get_absolute_url }}">
          {{ task.name }}
        </a>
        {% else %}
        {{ task.name }}
        {% endif %}
      </td>
      <td>{{ task.completed }}</td>
    </tr>
```

(continues on next page)

(continued from previous page)

```
    {% endfor %}
    </tbody>
  </table>
  <a href="{{ object.get_override_url }}">Override</a>
</body>
</html>
```

You are all set! Spin up your application and play around with it. Once you are done come back to learn *how to write tests in the next part of our tutorial*.

2.1.3 Testing your workflow

Joeflow is designed to make testing as simple as possible. Machine tasks are the simplest to test. You just call the method on the workflow. Following our example, your tests could looks something like this:

```
from django.contrib.auth import get_user_model
from django.core import mail
from django.test import SimpleTestCase, TestCase
from django.urls import reverse

from . import workflows


class WelcomeWorkflowMachineTest(SimpleTestCase):
    def test_has_user__with_user(self):
        user = get_user_model()(
            email="spiderman@avengers.com",
            first_name="Peter",
            last_name="Parker",
            username="spidy",
        )
        workflow = workflows.WelcomeWorkflow(user=user)
        self.assertEqual(workflow.has_user(), [workflow.send_welcome_email])

    def test_has_user__without_user(self):
        workflow = workflows.WelcomeWorkflow()
        self.assertEqual(workflow.has_user(), [workflow.end])

    def test_send_welcome_email(self):
        user = get_user_model()(
            email="spiderman@avengers.com",
            first_name="Peter",
            last_name="Parker",
            username="spidy",
        )
        workflow = workflows.WelcomeWorkflow(user=user)

        workflow.send_welcome_email()

        email = mail.outbox[-1]
        self.assertEqual(email.subject, "Welcome")
        self.assertEqual(email.body, "Hello Peter!")
        self.assertIn("spiderman@avengers.com", email.to)
```

The tests above a regular unit tests covering the machine tasks. Testing the human tasks is similarly simple. Since machine tasks are nothing but views you can use Django's test `Client`. Here an example:

```python
class WelcomeWorkflowHumanTest(TestCase):
    start_url = reverse("welcomeworkflow:start")

    def test_start__get(self):
        response = self.client.get(self.start_url)
        self.assertEqual(response.status_code, 200)

    def test_start__post_with_user(self):
        user = get_user_model().objects.create(
            email="spiderman@avengers.com",
            first_name="Peter",
            last_name="Parker",
            username="spidy",
        )

        response = self.client.post(self.start_url, data=dict(user=user.pk))
        self.assertEqual(response.status_code, 302)
        workflow = workflows.WelcomeWorkflow.objects.get()
        self.assertTrue(workflow.user)
        self.assertTrue(workflow.task_set.succeeded().filter(name="start").exists())

    def test_start__post_without_user(self):
        response = self.client.post(self.start_url)
        self.assertEqual(response.status_code, 302)
        workflow = workflows.WelcomeWorkflow.objects.get()
        self.assertFalse(workflow.user)
        self.assertTrue(workflow.task_set.succeeded().filter(name="start").exists())
```

Note that the start task is somewhat special, since it does not need a running workflow. You can test any other task by simply creating the workflow and task in during test setup. In those cases you will need pass the task primary key. You can find more information about this in the *URLs documentation*.

2.2 Core Components

2.2.1 Workflow

The *`Workflow`* is where all your components come together. It defines the flow overall flow and the different states of your workflow. Workflow are also the vehicle for the other two components `Tasks` and *edges*.

It combines both behavior and state using familiar components. The state is persisted via a Django `Model` for each instance of your workflow.

2.2.2 Task

A task defines the behavior of a workflow. It can be considered as a simple transaction that changes state of a workflow. There are two types of tasks, human and machine tasks.

Human tasks are represented by a Django `View`. A user can change the workflows state via a Django form or a JSON API.

Machine tasks are represented by simple methods on the *`Workflow`* class. They can change the state and perform any action you can think of. They can decide which task to execute next (exclusive gateway) but also start or wait for multiple other tasks (split/join gateways).

Furthermore tasks can implement things like sending emails or fetching data from an 3rd party API. All tasks are executed asynchronously to avoid blocking IO and locked to prevent raise conditions.

2.2.3 Edges

Edges are the glue that binds tasks together. They define the transitions between tasks. They are represented by a simple list of tuples. Edges have no behavior but define the structure of a workflow.

2.2.4 Advanced Workflow API

class joeflow.models.**Workflow**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

The *WorkflowState* object holds the state of a workflow instances.

It is represented by a Django Model. This way all workflow states are persisted in your database.

get_absolute_url()
Return URL to workflow detail view.

classmethod get_graph_svg()
Return graph representation of a model workflow as SVG.

The SVG is HTML safe and can be included in a template, e.g.:

```
<html>
<body>
<!--// other content //-->
{{ workflow_class.get_graph_svg }}
<!--// other content //-->
</body>
</html>
```

Returns SVG representation of a running workflow.

Return type (django.utils.safestring.SafeString)

get_instance_graph_svg(*output_format='svg'*)
Return graph representation of a running workflow as SVG.

The SVG is HTML safe and can be included in a template, e.g.:

```
<html>
<body>
<!--// other content //-->
{{ object.get_instance_graph_svg }}
<!--// other content //-->
</body>
</html>
```

Returns SVG representation of a running workflow.

Return type (django.utils.safestring.SafeString)

get_override_url()
Return URL to workflow override view.

classmethod urls()
Return all URLs to workflow related task and other special views.

Example:

```
from django.urls import path, include

from . import models

urlpatterns = [
    # ...
    path('myworkflow/', include(models.MyWorkflow.urls())),
]
```

Returns Tuple containing aw list of URLs and the workflow namespace.

Return type tuple(list, str)

class joeflow.models.**Task**(*id*, *_workflow*, *content_type*, *name*, *type*, *status*, *completed_by_user*, *created*, *modified*, *completed*, *exception*, *stacktrace*)
Bases: django.db.models.base.Model

enqueue(*countdown=None*, *eta=None*)
Schedule the tasks for execution.

Parameters

- **countdown** (*int*) – Time in seconds until the time should be started.
- **eta** (*datetime.datetime*) – Time at which the task should be started.

Returns Celery task result.

Return type celery.result.AsyncResult

start_next_tasks(*next_nodes: list = None*)
Start new tasks following another tasks.

Parameters

- **self** (Task) – The task that precedes the next tasks.
- **next_nodes** (*list*) – List of nodes that should be executed next. This argument is optional. If no nodes are provided it will default to all possible edges.

2.3 Tasks

A task defines the behavior or a workflow.

A task can be considered as a simple transaction that changes state of a workflow. There are two types of tasks, human and machine tasks.

2.3.1 Human

Human tasks are represented by a Django View.

A user can change the workflows state via a Django form or a JSON API. Anything you can do in a view you can do in a human task. They only difference to machine tasks is that they require some kind of interaction.

You can use view mixins like the `PermissionRequiredMixin` or `LoginRequiredMixin` to create your own tasks that are only available to certain users.

Generic human tasks

Set of reusable human tasks.

class joeflow.tasks.human.**StartView**(***kwargs*)
Start a new workflow by a human with a view.

Starting a workflow with a view allows users to provide initial data.

Similar to Django's `CreateView` but does not only create the workflow but also completes a tasks.

class joeflow.tasks.human.**UpdateView**(***kwargs*)
Modify the workflow state and complete a human task.

Similar to Django's `UpdateView` but does not only update the workflow but also completes a tasks.

2.3.2 Machine

Machine tasks are represented by simple methods on the *`Workflow`* class.

They can change the state and perform any action you can think of. They can decide which task to execute next (exclusive gateway) but also start or wait for multiple other tasks (split/join gateways).

Furthermore tasks can implement things like sending emails or fetching data from an 3rd party API. All tasks are executed asynchronously to avoid blocking IO and locked to prevent raise conditions.

Return values

Machine tasks have three different allowed return values all of which will cause the workflow to behave differently:

None: If a task returns `None` or anything at all the workflow will just proceed as planed and follow all outgoing edges and execute the next tasks.

Iterable: A task can return also an explicit list of tasks that should be executed next. This can be used to create exclusive gateways:

```
from django.utils import timezone
from joeflow.workflows import Workflow
from joeflow import tasks


class ExclusiveWorkflow(Workflow):
    start = tasks.Start()

    def is_workday(self):
        if timezone.localtime().weekday() < 5:
            return [self.work]
        else:
            return [self.chill]

    def work(self):
        # pass time at the water cooler
        pass
```

(continues on next page)

(continued from previous page)

```
    def chill(self):
        # enjoy life
        pass

    edges = (
        (start, is_workday),
        (is_workday, work),
        (is_workday, chill),
    )
```

A task can also return am empty list. This will cause the workflow branch to come to a halt and no further stats will be started.

Warning: A task can not be a generator (*yield* results).

`False`: A task can also return a boolean. Should a task return `False` the workflow will wait until the condition changes to `True` (or anything but `False`):

```
from joeflow import tasks
form joeflow.workflows import Workflow
from django.utils import timezone


class WaitingWorkflow(Workflow):
    start = tasks.Start()

    def wait_for_weekend(self):
        return timezone.now().weekday() >= 5

    def go_home(self):
        # enjoy life
        pass

    edges = (
        (start, wait_for_weekend),
        (wait_for_weekend, go_home),
    )
```

Exceptions

Should a task raise an exception the tasks will change it status to failed. The exception that caused the task to fail will be recorded on the task itself and further propagated. You can find and rerun failed tasks form Django's admin interface.

Generic machine tasks

Set of reusable machine tasks.

class joeflow.tasks.machine.**Start**

Start a new function via a callable.

Creates a new workflow instance and executes a start task. The start task does not do anything beyond creating the workflow.

Sample:

```
from django.db import models
from joeflow.models import Workflow
from joeflow import tasks


class StartWorkflow(Workflow):
    a_text_field = models.TextField()

    start = tasks.Start()

    def end(self):
        pass

    edges = (
        (start, end),
    )

workflow = StartWorkflow.start(a_text_field="initial data")
```

class joeflow.tasks.machine.**Join**(**parents*)

Wait for all parent tasks to complete before continuing the workflow.

Parameters ***parents** (*str*) – List of parent task names to wait for.

Sample:

```
from django.db import models
from joeflow.models import Workflow
from joeflow import tasks


class SplitJoinWorkflow(Workflow):
    parallel_task_value = models.PositiveIntegerField(default=0)

    start = tasks.Start()

    def split(self):
        return [self.batman, self.robin]

    def batman(self):
        self.parallel_task_value += 1
        self.save(update_fields=['parallel_task_value'])

    def robin(self):
        self.parallel_task_value += 1
        self.save(update_fields=['parallel_task_value'])

    join = tasks.Join('batman', 'robin')

    edges = (
        (start, split),
        (split, batman),
        (split, robin),
        (batman, join),
        (robin, join),
    )
```

class joeflow.tasks.machine.**Wait**(*duration: datetime.timedelta*)
Wait for a certain amount of time and then continue with the next tasks.

Parameters **duration** (*datetime.timedelta*) – Time to wait in time delta from creation of task.

Sample:

```
import datetime

from django.db import models
from joeflow.models import Workflow
from joeflow import tasks


class WaitWorkflow(Workflow):
    parallel_task_value = models.PositiveIntegerField(default=0)

    start = tasks.Start()

    wait = tasks.Wait(datetime.timedelta(hours=3))

    def end(self):
        pass

    edges = (
        (start, wait),
        (wait, end),
    )
```

2.4 Settings

class joeflow.conf.**JoeflowAppConfig**(***kwargs*)
List of available settings.

To change the default values just set the setting in your settings file.

JOEFLOW_CELERY_QUEUE_NAME = 'yoloflow'
Queue name in which all machine tasks will be queued.

JOEFLOW_TASK_RUNNER = 'joeflow.runner.celery.task_runner'
Task runner is used to execute machine tasks.

JoeFlow supports two different asynchronous task runners – Dramatiq and Celery.

To use either of the task runners change this setting to:

- `joeflow.runner.dramatiq.task_runner`
- `joeflow.runner.celery.task_runner`

2.5 Management Commands

2.5.1 render_workflow_graph

Render workflow graph to file:

```
usage: manage.py render_workflow_graph [-h] [-f {svg,pdf,png}] [-d DIRECTORY]
                                       [-c] [model [model ...]]

Render workflow graph to file.

positional arguments:
  workflow              List of workflow to render in the form
                        app_label.workflow_name

optional arguments:
  -h, --help            show this help message and exit
  -f {svg,pdf,png}, --format {svg,pdf,png}
                        Output file format. Default: svg
  -d DIRECTORY, --directory DIRECTORY
                        Output directory. Default is current working
                        directory.
  -c, --cleanup         Remove dot-files after rendering.
```

2.6 URLs

Should you ever need to get the URL – like for a test – for a task you can use Django's `reverse`. All users follow a simple pattern consisting of the workflow name (lowercase) and task name, e.g.:

```
>>> from django.urls import reverse
>>> reverse("workflow_name:task_name", args=[task.pk])
'/url/to/workflow/task/1'
```

All task URLs need the *.Task* primary key as an argument. There are some special views that do not like the workflow detail and override view, e.g.:

```
>>> reverse('welcomeworkflow:start')
'/welcome/start/'
>>> reverse('welcomeworkflow:detail', args=[workflow.object.pk])
'/welcome/1/'
>>> reverse('welcomeworkflow:override', args=[workflow.object.pk])
'/welcome/1/override'
```

The first example does not need a primary key, since it is a *StartView* and the workflow is not created yet. The latter two examples are workflow related views. The need the `WorkflowState` primary key as an argument.

Note: The workflow detail view is also available via *Workflow.get_absolute_url()*. The override view is available via *Workflow.get_override_url()*.

Photo by rawpixel.com from Pexels

Python Module Index

j

Index