Triggers Documentation

Release

Phoenix Zerin

Apr 21, 2018

Contents

The Triggers framework is essentially a distributed implementation of the Observer pattern. It keeps track of “triggers” that your application fires, and schedules asynchronous tasks to run in response.

On the surface, this seems simple enough, but what make the Triggers framework so compelling (at least, in the eyes of the person writing this documentation) are:

- It is designed for distributed applications. You can track events that occur across multiple VMs.
- It uses a persistent storage backend to maintain state. It can schedule tasks in response to events, even if they occur days, weeks, even months apart.
- It uses an intuitive JSON configuration schema, allowing administrators to create complex workflows without having to write any (Python) code.

CHAPTER 1

What Are Triggers Useful For

The Triggers framework may be a good fit for your application if:

- It needs to be able to schedule asynchronous tasks when sets of 2 or more conditions are met, and
- You can't predict when, what order, or even if each of the different conditions will be met.

For example, suppose you have a survey application, and you want to schedule an asynchronous task to run after modules 1 and 4 are received from the client.

However, because of the way the internet works, module 4 might never arrive, or perhaps the two modules arrive out-of-order, or even at the same time.

The Triggers framework would be a good fit for this application.

CHAPTER 2

What Are Triggers Not Useful For

If your application:

- Is not distributed (e.g., only has one application server), or
- Does not need to maintain state across requests,

then the Triggers framework might be overkill.

For example, using the survey application from the previous section, suppose that the client always sent the data for modules 1 and 4 in the same web service request.

In this case, you wouldn't need to use the Triggers framework because your application would not need to keep track of which modules were received across multiple web service requests.

CHAPTER 3

Configuration

Let's go back to the example survey application, and see how we might configure the Triggers framework to execute an asynchronous task after modules 1 and 4 are received:

```
{
  // Give your task a name.
  "t_processStepData": {

    // This task runs after these two triggers are fired.
    // Note that order doesn't matter here.
    "after": ["module1Received", "module4Received"],

    // Specify the celery task to run when the above conditions are
    // met.
    "run": "my_app.tasks.ProcessStepData"
  }
}
```

That's it! The Triggers framework will take it from there.

We'll explore exactly what this configuration means, and how to set up more complex workflows in the *Configuration* section.

CHAPTER 4

Basic Concepts

The Triggers framework is loosely based on the Observer pattern, so many of the concepts described here might look familiar.

4.1 Triggers

A trigger is very similar to an event in the Observer pattern. Essentially, it is just a string/identifier, with some optional metadata attached to it.

What makes triggers so important is that **your application decides what they are named, and when to fire them**.

Here's an example of how a survey application might fire a trigger in response to receiving a payload containing data collected by a module:

```
def process_module_1(module_data):
  """
  Processes the data received from module 1.
  """
  #
  # ... process the module data, store results to DB, etc. ...
  #

  # Create a Trigger Manager instance (more on this later).
  trigger_manager = TriggerManager(...)

  # Fire a trigger.
  trigger_manager.fire('module1Received')
```

In this example, the `module1Received` trigger has meaning because your application will only fire it once it finishes processing the data from module 1.

4.1.1 Trigger Kwargs

When your application fires a trigger, it can also attach some kwargs to it. Any task that runs in response to this trigger will have access to these kwargs, so you can use this to provide additional metadata that a task might need.

Using the above example, let's imagine that your application stores the module data to a document database, and you want to add the document ID to the trigger kwargs.

The result might look something like this:

```
def process_module_1(module_data):
  """
  Processes the data received from module 1.
  """
  # Store the module data to a document database.
  document_id = db.store(module_data)

  # Create a Trigger Manager instance (more on this later).
  trigger_manager = TriggerManager(...)

  # Fire a trigger, with kwargs.
  trigger_manager.fire('module1Received', {'document_id': document_id})
```

When the application fires the `module1Received` trigger, it attaches a kwarg for `document_id`. This value will be accessible to any task that runs in response to this trigger, so that it can load the module data from the document database.

Note: Celery schedules tasks by sending messages to a queue in a message broker, so trigger kwargs must be serializable using Celery's task_serializer.

4.2 Tasks

Firing triggers is fun and all, but the whole point here is to execute Celery tasks in response to those triggers!

This is where trigger tasks come into play.

A trigger task acts like a wrapper for a Celery task:

- The Celery task does the actual work.
- The trigger task defines the conditions that will cause the Celery task to get executed.

4.2.1 Task Configurations

Here's an example trigger configuration that defines two tasks, named *t_createApplicant* and *t_computeScore*:

```
{
  "t_createApplicant": {
    "after": ["startSession", "observationsReceived"],
    "run": "applicant_journey.tasks.Import_CreateApplicant"
  },

  "t_computeScore": {
    "after": ["t_createApplicant", "sessionFinalized"],
    "run": "applicant_journey.tasks.Score_ComputePsychometric"
  }
}
```

We can translate the above configuration into English like this:

```
Trigger Task "t_createApplicant":
  After the "startSession" and "observationsReceived" triggers fire,
  Run the "Import_CreateApplicant" Celery task.

Trigger Task "t_computeScore":
  After the "t_createApplicant" and "sessionFinalized" triggers fire,
  Run the "Score_ComputePsychometric" Celery task.
```

We'll explore what this all means in the *How to Configure* section.

Note: Did you notice that one of the triggers for `t_computeScore` (inside its `after` attribute) is the name of another trigger task (`t_createApplication`)?

This takes advantage of a feature called *cascading*, where a trigger task fires its own name as a trigger when its Celery task finishes successfully.

In this way, you can "chain" trigger tasks together.

We will cover cascading in more detail in *Writing Celery Tasks*.

4.3 Task Instances

In certain cases, a task may run multiple times. To accommodate this, the Triggers framework creates a separate task instance for each execution of a task.

Each task instance is named after its task configuration, with an incrementing sequence number (e.g., `t_createApplicant#0`, `t_computeScore#0`, etc.).

4.4 Sessions

A session acts as a container for triggers and trigger task instances. This allows you to maintain multiple states in isolation from each other.

For example, if you maintain a survey application, each survey would have its own session. This way, any triggers fired while processing a particular survey would not interfere with any other surveys.

4.4.1 Session UIDs

Each session should have a unique identifier (UID). This value is provided to the storage backend at initialization, so that the trigger manager can load the saved state for that session.

4.5 Trigger Managers

The trigger manager acts as the controller for the Triggers framework. It is responsible for firing triggers, managing trigger task instances, and so on.

To interact with the Triggers framework in your application, create an instance of the trigger manager class, like this:

```
from triggers import CacheStorageBackend, TriggerManager

# Specify the session UID.
sessionUid = '...'

# Create the trigger manager instance.
trigger_manager = TriggerManager(CacheStorageBackend(sessionUid))

# Fire triggers.
trigger_manager.fire('ventCoreFrogBlasted')
```

4.6 Storage Backends

To maintain state across multiple processes, the trigger manager relies on a storage backend.

The storage backend is responsible for loading and storing the session state.

The Triggers framework comes bundled with a cache storage backend, which stores session state using Django's cache. Additional backends will be added in future versions of the library.

CHAPTER 5

Getting Started

Getting started with the Triggers framework requires a bit of planning.

5.1 Step 1: Define Session UIDs

Sessions limit the context in which the Triggers framework operates. This allows your application to maintain separate state for each user of your application.

In order to integrate the Triggers framework into your application, you will first need to decide what to use for session UIDs.

Depending on your application, you may want to maintain separate state per user ID, or you might want to use the IDs of your application's web sessions, etc.

For example, if we want to integrate the Triggers framework into a questionnaire application, we might opt to create a new session UID each time a user starts a new questionnaire.

5.2 Step 2: Design Your Workflows

Once you've defined the scope of each session, you'll need to think about what workflows you want to support over the course of each session:

1. **What tasks do you want to run?** Figure out what Celery tasks you want to run when certain conditions are met.

   For example, our questionnaire application might have these Celery tasks:

   - `app.tasks.ImportSubject` imports details about the user into a SQL database.
   - `app.tasks.ImportResponses` imports the user's response data into a document database.
   - `app.tasks.ImportBrowserMetadata` sends a request to a 3rd-party web service to download metadata about the user's browser, based on their user agent string.

2. **When do you want to run them?** Decide what triggers have to fire in order for each of those tasks to run. Your application will decide when these happen, so they can correspond to any action or condition evaluated by your code.

   Tip: You can also define triggers that will **prevent** certain tasks from running.

   Going back to the questionnaire application above, we might define our triggers like this:

   - We only want to import data for applicants who successfully complete the questionnaire. `ImportSubject` needs information from the first page of the questionnaire, **but** it shouldn't run until the questionnaire is completed.
   - `ImportResponses` should run **each time** we receive a page of questionnaire responses, **but** it requires a subject ID, so it can only run once `ImportSubject` has finished successfully.
   - `ImportBrowserMetadata` should run **once** after any single page of responses are received, **but** it also requires a subject ID, so it can only run after `ImportSubject` has finished successfully.

     However, if the application detects that the user is completing the questionnaire from an embedded application, then this task should **not** run.

   Tip: The Triggers framework works best when tasks have to wait for multiple asynchronous/unpredictable events in order to run.

   If you find yourself designing tasks that only require a single trigger to run, or if you just want to ensure that tasks run in a specific order, Celery already has you covered.

3. Give each task and trigger a unique name and write them out like this:

```
{
  // Task that runs once.
  "<task name>": {
    "after": ["<trigger>", "<trigger>", ...],
    "run": "<celery task>"
  },

  // Task that can run multiple times:
  "<task name>": {
    "after": ["<trigger>", "<trigger>", ...],
    "andEvery": "<trigger>",
    "run": "<celery task>"
  },

  // Task that will run unless certain condition is met:
  "<task name>": {
    ...
    "unless": ["<trigger>", ...]
  },

  // etc.
}
```

   This will form the starting point for your trigger configuration.

   Here's what the starting configuration looks like for the questionnaire application:

```
{
  "t_importSubject": {
    // Imports data from the first page, but cannot run until
    // the questionnaire is completed.
    "after": ["firstPageReceived", "questionnaireComplete"],

    "run": "app.tasks.ImportSubject"
  },

  "t_importResponses": {
    // Imports response data from EVERY page, but cannot run
    // until the subject data are imported.
    "after": ["t_importSubject"],
    "andEvery": "pageReceived",

    "run": "app.tasks.ImportResponses"
  },

  "t_importBrowserMetadata": {
    // Loads the user agent string from any ONE page of
    // responses (we don't care which one), but cannot run
    // until the subject data are imported...
    "after": ["t_importSubject", "pageReceived"],

    // ... unless the application determines that the requests
    // are coming from an embedded app, in which case, this
    // task should NOT run.
    "unless": ["isEmbeddedApplication"],

    "run": "app.tasks.ImportBrowserMetadata"
  }
}
```

Notice in the above configuration that the trigger task names are distinct from the Celery task names; in some cases, you may have multiple trigger tasks that reference the same Celery task.

Tip: Note that you can also use the name of a trigger task itself as a trigger (this is a technique known as “cascading”, which is described in more detail later on). This allows you to specify that a particular task must finish successfully before another task can run.

In the example configuration, the `t_importResponses` trigger task cannot run until the `t_importSubject` trigger task has finished successfully, so we added `t_importSubject` to `t_importResponses.after`.

To make it easier to identify these cases (and to prevent conflicts in the event that a trigger has the same name as a trigger task), a `t_` prefix is added to trigger task names.

You are recommended to follow this convention, but it is not enforced in the code. You may choose a different prefix, or (at your own risk) eschew prefixes entirely in your configuration.

5.3 Step 3: Select a Storage Backend

In order for the Triggers framework to function, it has to store some state information in a storage backend.

Currently, the only storage backend uses the Django cache. In the future, additional backend will be added to provide more options (e.g., Django ORM, document database, etc.).

Tip: If you use Redis as your cache backend, you can configure the Triggers framework so that it stores values with no expiration time.

You can also *write your own storage backend*.

5.4 Step 4: Fire Triggers

Now it's time to start writing some Python code!

Back in step 2, we defined a bunch of triggers. Now we're going to write the code that fires these triggers.

To fire a trigger, create a trigger manager instance, and provide a storage backend instance, then call the trigger manager's `fire()` method.

It looks like this:

```
from triggers import TriggerManager, CacheStorageBackend

storage_backend =\
  CacheStorageBackend(
    # Session UID (required)
    uid = session_uid,

    # Name of cache to use.
    cache = 'default',

    # TTL to use when setting values.
    # Depending on which cache you use (e.g., Redis), setting
    # ``timeout=None`` may store values permanently, or it
    # may use the cache's default timeout value.
    timeout = 3600,
  )

trigger_manager = TriggerManager(storage_backend)

trigger_manager.fire(trigger_name)
```

In the above code, replace `session_uid` with the Session UID that you want to use (see Step 1 above), and `trigger_name` with the trigger that you want to fire.

Tip: Depending on the complexity of your application, you might opt to use a function and/or Django settings to create the trigger manager instance.

See the *Cookbook* for a sample implementation.

5.4.1 Trigger Kwargs

When your application fires a trigger, it can also attach keyword arguments to that trigger. These arguments will be made available to the Celery task when it runs.

Here's an example of how our questionnaire application might fire the `pageReceived` trigger:

```
def responses(request):
  """
  Django view that processes a page of response data from
  the client.
  """
  responses_form = QuestionnaireResponsesForm(request.POST)
  if responses_form.is_valid():
    trigger_manager = TriggerManager(
      storage = CacheStorageBackend(
        uid      = responses_form.cleaned_data['questionnaire_id'],
        cache    = 'default',
        timeout  = 3600,
      ),
    )

    trigger_manager.fire(
      trigger_name   = 'pageReceived',
      trigger_kwargs = {'responses': responses.cleaned_data},
    )

    ...
```

Caution: Behind the scenes, the trigger kwargs will be provided to the Celery task via the task's `kwargs`, so any values that you use for trigger kwargs must be compatible with Celery's serializer.

5.5 Step 5: Initialize Configuration

Next, you need to write the code that will initialize the configuration for each new session.

This is accomplished by invoking `TriggerManager.update_configuration()`:

```
trigger_manager.update_configuration({
  # Configuration from Step 2 goes here.
})
```

Here's an example showing how we would initialize the trigger configuration at the start of the questionnaire application:

```
def start_questionnaire(request):
  """
  Django view tha processes a request to start a new questionnaire.
  """
  # Create the new questionnaire instance.
  # For this example, we will use the PK value of the
  # new database record as the session UID.
  new_questionnaire = Questionnaire.objects.create()

  trigger_manager = TriggerManager(
    storage = CacheStorageBackend(
      # The session UID must be a string value.
      uid = str(new_questionnaire.pk),
      cache     = 'default',
      timeout   = 3600,
    ),
  )

  trigger_manager.update_configuration({
    't_importSubject': {
      'after': ['firstPageReceived', 'questionnaireComplete'],
      'run': 'app.tasks.ImportSubject',
    },

    't_importResponses': {
      'after': ['t_importSubject'],
      'andEvery': 'pageReceived',
      'run': 'app.tasks.ImportResponses',
    },

    't_importBrowserMetadata': {
      'after': ['t_importSubject', 'pageReceived'],
      'unless': ['isEmbeddedApplication'],
      'run': 'app.tasks.ImportBrowserMetadata',
    },
  })

  ...
```

5.6 Step 6: Write Celery Tasks

The final step is writing the Celery tasks. These will look similar to normal Celery tasks, with a couple of differences:

- The tasks must extend `triggers.task.TriggerTask`.
- Override the `_run` method instead of `run` (note the leading underscore).

For more information about see *Writing Celery Tasks*.

CHAPTER 6

How to Configure

To configure a trigger manager instance, call its `update_configuration` method and provide a dict with the following items:

- Each key is the name of a trigger task. These can be anything you want, but the convention is to start each name with a `t_` prefix.
- Each value is a dict containing that task's configuration.

Here is an example showing how to add 3 tasks to the trigger manager's configuration.

```
trigger_manager.update_configuration({
  't_importSubject':         {...},
  't_importResponses':       {...},
  't_importDeviceMetadata':  {...},
})
```

6.1 Task Configuration

There are many directives you can specify to customize the behavior of each trigger task.

The directives are named and structured in such a way that you should be able to "read" a trigger task configuration like an English sentence.

As an example, consider the following trigger task configuration:

```
trigger_manager.update_configuration({
  't_importResponses': {
    'after': ['t_importSubject'],
    'andEvery': 'pageReceived',
    'run': 'app.tasks.ImportResponses',
  },
})
```

You can "read" the configuration for the `t_importResponses` task as:

After `t_importSubject` fires, **and every** time `pageReceived` fires, **run** the `app.tasks.ImportResponses` Celery task.

6.1.1 Required Directives

The following directives must be provided in each task's configuration.

after

`after` is a list of strings that indicates which triggers must fire in order for the task to be run.

Tip: Recall from *Getting Started* that triggers are fired by your application logic, so you get to decide how triggers are named and what events they represent.

As an example, suppose we want a task to process data from the first page of a questionnaire, but we don't want it to run until the user has completed the questionnaire. We might configure the trigger task like this:

```
trigger_manager.update_configuration({
  't_importSubject': {
    'after': ['firstPageReceived', 'questionnaireComplete'],
    ...
  },
})
```

You can also use task names as triggers. These will fire each time the corresponding task finishes successfully.

Here's an example of a task that processes data from a single page of questionnaire responses, but only after the `t_importSubject` task has finished successfully.

```
trigger_manager.update_configuration({
  't_importDeviceMetadata': {
    'after': ['pageReceived', 't_importSubject'],
    ...
  },
})
```

Tip: The order of values in `after` do not matter.

For compatibility with serialization formats like JSON, `after` is usually expressed as a `list` in Python code, but you can use a `set` if you prefer.

run

`run` tells the trigger manager which Celery task to run once the trigger task's `after` condition is satisfied.

The value should match the `name` of a Celery task, exactly the same as if you were configuring `CELERYBEAT_SCHEDULE`.

As an example, to configure a trigger task to run the `my_app.tasks.ImportSubject` task, the configuration might look like this:

```
from my_app.tasks import ImportSubject

trigger_manager.update_configuration({
  't_importSubject': {
    ...
    'run': ImportSubject.name,
  },
})
```

Important: The trigger manager can only execute Celery tasks that extend the `triggers.task.TriggerTask` class.

See *Writing Celery Tasks* for more information.

6.1.2 Optional Directives

The following optional directives allow you to further customize the behavior of your trigger tasks.

`andEvery`

By default, every trigger task is "one shot". That is, it will only run once, even if the triggers in its `after` directive are fired multiple times.

If you would like a trigger task to run multiple times, you can add the `andEvery` directive to the trigger configuration.

`andEvery` accepts a **single** trigger. Whenever this trigger fires, the trigger manager will create a new "instance" of the trigger task.

For example, suppose we want to configure a trigger task to process data from each page in a questionnaire, but it can only run once the `t_importSubject` trigger task has finished successfully.

The configuration might look like this:

```
trigger_manager.update_configuration({
  't_importResponses': {
    'after': ['t_importSubject'],
    'andEvery': 'pageReceived',
    ...
  },
})
```

Using the above configuration, a new instance of `t_importResponses` will be created, but **they will only run after the t_importSubject task finishes**.

`unless`

`unless` is the opposite of `after`. It defines a condition that will **prevent** the trigger task from running.

Once a task's `unless` condition is satisfied, the trigger manager will not allow that task to run, even if its `after` condition is satisfied later.

Important: This only prevents the trigger manager from scheduling Celery tasks. It will not recall a Celery task that has already been added to a Celery queue, nor will it abort any task that is currently being executed by a Celery worker.

As an example, suppose you wanted to import metadata about the applicant's browser during a questionnaire, but only if the user is completing the questionnaire in a web browser. If the backend detects that the questionnaire is embedded in a mobile application, then this task should not run.

The configuration might look like this:

```
trigger_manager.update_configuration({
  't_importBrowserMetadata': {
    'after': ['t_importSubject', 'pageReceived'],
    'unless': ['isEmbeddedApplication'],
    ...
  },
})
```

If `isEmbeddedApplication` fires before `t_importSubject` and/or `pageReceived`, then the trigger manager will not allow the `t_importBrowserMetadata` task to run.

Caution: Watch out for race conditions!

withParams

When the trigger manager executes a task, it will provide the kwargs that were provided when each of that task's `after` triggers were fired (see *Writing Celery Tasks* for more information).

But, what if you need to inject your own static kwargs?

This is what the `withParams` directive is for.

As an example, suppose you have a generic trigger task that you use to generate a psychometric credit score at the end of a questionnaire, but you have to tell it which model to use.

Using the `withParams` directive, you can inject the name of the model like this:

```
from my_app.tasks import ComputeScore

trigger_manager.update_configuration({
  't_computePsychometricScore': {
    ...
    'run': ComputeScore.name,

    'withParams': {
      'scoring': {'model': 'Psych 01'},
    },
  },
})
```

When the `my_app.tasks.ComputeScore` Celery task runs, it will be provided with the model name `'Psych 01'` so that it knows which model to load.

Important: `withParams` must be a dict of dicts, so that it matches the structure of trigger kwargs (see *Writing Celery Tasks* for more information).

For example, this configuration is **not** correct:

```
trigger_manager.update_configuration({
  't_computePsychometricScore': {
    ...
    'withParams': {
      'model': 'Psych 01',
    },
  },
})
```

using

By default, the trigger manager uses Celery to execute trigger tasks (except during *unit tests*).

However, if you want to use a different *task runner*, you can specify it via the `using` directive.

For example, suppose we created a custom task runner that executes tasks via AWS Lambda. To tell the trigger manager to execute a task using the custom task runner, we might use the following configuration:

```
from my_app.tasks import ComputeScore
from my_app.triggers.runners import AwsLambdaRunner

trigger_manager.update_configuration({
  't_computePsychometricScore': {
    ...
    'run': ComputeScore.name,
    'using': AwsLambdaRunner.name,
  },
})
```

Tip: To change the default task runner globally, override `triggers.runners.DEFAULT_TASK_RUNNER`.

6.1.3 Custom Directives

You can add any additional directives that you want; each will be added to the corresponding task's `extras` attribute.

These aren't used for anything by default, but if you write a *custom trigger manager*, you can take advantage of custom directives to satisfy your application's requirements.

For an example of how to use custom directives, see the "Finalizing a Session" recipe in the *Cookbook*

CHAPTER 7

Writing Celery Tasks

The primary function of the Triggers framework is to execute Celery tasks.

For the most part, these look the same as any other Celery tasks, with two notable differences:

- The tasks must extend `triggers.task.TriggerTask`.
- Override the `_run` method instead of `run` (note the leading underscore).

As an example, consider the following trigger task:

```
trigger_manager.update_configuration({
  't_importSubject': {
    'after': ['firstPageReceived', 'questionnaireComplete'],
    'run': 'app.tasks.ImportSubject',
  },
  ...
})
```

The idea here is that the `ImportSubject` Celery task takes data from the first page of response data and creates a `Subject` record in the database.

The application will help the Celery task by attaching the response data to the `firstPageReceived` trigger when it fires:

```
def first_page_responses(request):
  """
  Django view that processes the first page of response data
  from the client.
  """
  responses_form = QuestionnaireResponsesForm(request.POST)
  if responses.is_valid():
    ...

    trigger_manager.fire(
      trigger_name   = 'firstPageReceived',
      trigger_kwargs = {'responses': responses.cleaned_data},
    )

    ...
```

Note that when the `firstpageReceived` trigger is fired, the response data are attached via `trigger_kwargs`.

Here's what the `ImportSubject` Celery task might look like:

```python
from my_app.models import Subject
from triggers.task import TaskContext, TriggerTask

class ImportSubject(TriggerTask):
  def _run(self, context):
    # type: (TaskContext) -> dict

    # Load kwargs provided when the ``firstPageReceived``
    # trigger was fired by the application.
    page_data =\
      context.trigger_kwargs['firstPageReceived']['responses']

    # Create a new ``subject`` record.
    new_subject =\
      Subject.objects.create(
        birthday = page_data['birthday'],
        name     = page_data['name'],
      )

    # Make the PK value accessible to tasks that are
    # waiting for a cascade.
    return {
      'subjectId': new_subject.pk,
    }
```

The `ImportSubject` task's `_run` method (note the leading underscore) does 3 things:

1. Load the response data from the `firstPageReceived` trigger kwargs.
2. Import the data into a new `Subject` record.
3. Return the resulting ID value so that when the task cascades, other tasks will be able to use it (more on this later).

7.1 Task Context

The only argument passed to the `_run` method is a `triggers.task.TaskContext` object.

The `TaskContext` provides everything that your task will need to interact with the Triggers framework infrastructure:

7.1.1 Trigger Manager

`context.manager` is a trigger manager instance that you can leverage in your task to interact with the Triggers framework. For example, you can use `context.manager` to fire additional triggers as your task runs.

7.1.2 Trigger Kwargs

As noted above, whenever the application fires a trigger, it can attach optional kwargs to that trigger.

These kwargs are then made available to your task in two ways:

- `context.trigger_kwargs` returns the raw kwargs for each trigger that caused your task to run.
- `context.filter_kwargs()` uses the Filters library to validate and transform the `trigger_kwargs`.

The above example shows how to use `context.trigger_kwargs`. Here is an alternate approach that uses `context.filter_kwargs()` instead:

```
import filters as f

 class ImportSubject(TriggerTask):
   def _run(self, context):
     # type: (TaskContext) -> dict

     filtered_kwargs =\
       context.filter_kwargs({
         'firstPageReceived': {
           'responses':
               f.Required
             | f.Type(dict)
             | f.FilterMapper({
                 'birthday':  f.Required | f.Date,
                 'name':      f.Required | f.Unicode,
               }),
         },
       })

     page_data = filtered_kwargs['firstPageReceived']['responses']

     ...
```

Note: If you have worked with FilterMappers in the past, the above structure should look very familiar.

7.2 Cascading

When the Celery task finishes successfully, the trigger manager will cause a "cascade" by firing the corresponding trigger task's name as a trigger.

For example, consider the trigger task from earlier:

```
trigger_manager.update_configuration({
  't_importSubject': {
    'after': ['firstPageReceived', 'questionnaireComplete'],
    'run': 'app.tasks.ImportSubject',
  },
  ...
})
```

The trigger task is named `t_importSubject`, so when the `ImportSubject` Celery task finishes, the trigger manager will automatically fire a trigger named `t_importSubject`.

But, what kwargs are attached to this trigger?

If the Celery task returns a mapping (e.g., dict), then that will be used as the kwargs for the cascading trigger.

Going back to the `ImportSubject` example:

```
class ImportSubject(TriggerTask):
  def _run(self, context):
    ...

    # Make the PK value accessible to tasks that are
    # waiting for a cascade.
    return {
      'subjectId': new_subject.pk,
    }
```

When this task finishes, the trigger manager will cascade like this:

```
trigger_manager.fire(
  trigger_name   = 't_importSubject',
  trigger_kwargs = {'subjectId': new_subject.pk},
)
```

7.3 Logging

If your Celery task needs to use a logger, consider using `context.get_logger_context()`.

The logger instance returned by this method includes a few features that integrate closely with the trigger manager.

See *Logging* for more information.

7.4 Retrying

To retry a Celery task mid-execution, the method looks similar to a regular Celery task:

```
class ImportBrowserMetadata(TriggerTask):
  # Specify the max number of retries allowed.
  max_retries = 3

  def _run(self, context):
    # type: (TaskContext) -> dict
    ...

    try:
      # Try to load data from 3rd-party API...
      metadata = api_client.post(...)
    except HttpError as e:
      # ... but if we are unable to connect,
      # retry after a delay.
      raise self.retry(exc=e, cooldown=10)
```

Note that this retry mechanism works a little differently from Celery's retry:

- You must `raise self.retry()`; it won't raise the exception for you.
- Use `cooldown` instead of `countdown`. `eta` is not supported.

- If desired, you can also specify replacement trigger kwargs to use when retrying the task.

If the Celery task exceeds its `max_retries`, then it will raise a `triggers.task.MaxRetriesExceeded`.

CHAPTER 8

Task Instance Status

Each task instance has a status value associated with it (`TaskInstance.status`).

These are the possible status values:

`abandoned` The task instance will never run, because its `unless` clause was satisfied.

`failed` The Celery task failed due to an exception.

`finished` The Celery task finished successfully.

`replayed` The Celery task failed, and it was replayed.

When a failed instance is replayed, a new task instance is created to rerun the Celery task. This provides a mechanism for recovering from exceptions, while retaining the exception and traceback information for investigation.

`running` A Celery worker is currently executing the task.

`scheduled` The Celery task has been sent to the broker and is waiting for a worker to execute it.

In rare cases, an instance may remain in "scheduled" status for some time (for example, if no Celery workers are available to execute the task, or if the broker becomes unavailable).

`skipped` The Celery task failed, but it was marked as skipped (instead of retrying).

`unstarted` The task instance has been created, but it is not ready to run yet.

This occurs when some – but not all – of the triggers in the task's `after` clause have fired. The instance will remain in "unstarted" status until the remaining triggers have fired.

8.1 Meta-Statuses

`TaskInstance` also defines a few properties that can help your application to make decisions based on an instance's status:

`TaskInstance.can_abandon` Indicates whether the task instance's `unless` condition is satisfied. Returns `False` if the instance already has "abandoned" status.

`TaskInstance.can_run` Indicates whether the instance is ready to run (add a Celery task to the queue).

`TaskInstance.can_schedule` Indicates whether the instance is ready to be scheduled for execution.

This property is generally only used internally.

Important: This property does **not** indicate that the instance is ready to *run*; use `TaskInstance.can_run` for that.

`TaskInstance.can_replay` Indicates whether the instance can be replayed.

`TaskInstance.can_skip` Indicates whether the instance can be skipped.

`TaskInstance.is_resolved` Indicates whether this instance has a "final" status. Once an instance is resolved, no further operations may be performed on it.

Examples of resolved instances include:

- Celery task finished successfully (nothing left to do).
- `unless` clause satisfied (task must not run).
- Celery task failed, but the failed instance was replayed (a new instance was created for the replay).
- Celery task failed, but the failed instance was skipped (nothing left to do).

If an instance's `is_resolved` attribute is `False`, this means that it is currently in progress and/or requires some kind of change before it can be resolved. Some examples include:

- The instance hasn't been run yet because it is waiting for additional triggers (no action necessary).
- The instance has been scheduled for execution, but it is waiting for a Celery worker to become available (no action necessary).
- The instance is currently being executed by a Celery worker (no action necessary).
- The instance is in failed state (needs to be replayed or skipped).

Note that most of the time, an unresolved instance is not a bad thing.

8.2 Checking Instance Status

For more information about how to check an instance's status, see *Inspecting State and Error Recovery*.

CHAPTER 9

Inspecting State and Error Recovery

Each time you create a trigger manager instance, you also assign a storage backend. The storage backend is responsible for maintaining session state, but it also provides a number of methods and attributes that your application can inspect.

9.1 What's In Session State?

Inside of a session's state are 3 objects:

- `tasks` contains the configured trigger tasks.
- `instances` contains instances of each task.
- `metadata` contains internal metadata.

In general, you won't need to interact with these objects directly, but they can be useful for inspecting and troubleshooting sessions.

9.2 Inspecting Session State

To inspect a session's state, your application will interact with the trigger manager's storage backend.

Tip: If you only want to inspect a session's state (i.e., you don't need to fire triggers, change task instance status, etc.), you do not need to create a trigger manager instance; you only need an instance of the storage backend.

9.2.1 Inspecting Task Configuration

To inspect a trigger task's configuration, load it from `tasks`:

```
task = trigger_manager.storage.tasks['t_importSubject']
```

In the above example, `task` is an instance of `triggers.types.TaskConfig`.

9.2.2 Inspecting Instance Configuration

To inspect a trigger instance configuration, load it from `instances`:

```
instance = trigger_manager.storage.instances['t_importSubject#0']
```

In the above example, `instance` is an instance of `triggers.types.TaskInstance`.

Note: To get the instance, you must provide the name of the *instance*, not the name of the *task*:

```
# Using instance name:
>>> trigger_manager.storage.instances['t_importSubject#0']
TaskInstance(...)

# Using task name:
>>> trigger_manager.storage.instances['t_importSubject']
KeyError: 't_importSubject'
```

Finding Instances By Trigger Task

If you want to find all the instances for a particular task, use the `instances_of_task` method:

```
instances =\
  trigger_manager.storage.instances_of_task['t_importSubject']
```

In the above example, `instances` is a list of `TaskInstance` objects.

Finding Unresolved Tasks and Instances

When inspecting the state of a session, one of the most critical pieces of information that applications need is the list of tasks that haven't been finished yet.

The storage backend provides two methods to facilitate this:

`get_unresolved_tasks()` Returns a list of all tasks that haven't run yet, or have one or more unresolved instances.

`get_unresolved_instances()` Returns a list of all unresolved instances.

The difference between these methods is subtle but important.

It is best explained using an example:

```
>>> from uuid import uuid4
>>> from triggers import TriggerManager
>>> from triggers.storages.cache import CacheStorageBackend

>>> trigger_manager =\
...   TriggerManager(CacheStorageBackend(uuid4().hex))
...

>>> trigger_manager.update_configuration({
```

```
...   't_importSubject': {
...     'after': ['firstPageReceived', 'questionnaireComplete'],
...     'run':   '...',
...   },
... })
...

# ``t_importSubject`` hasn't run yet, so it is unresolved.
>>> trigger_manager.storage.get_unresolved_tasks()
[<TaskConfig 't_importSubject'>]

# None of the triggers in ``t_importSubject.after`` have fired
# yet, so no task instance has been created yet.
>>> trigger_manager.storage.get_unresolved_instances()
[]

>>> trigger_manager.fire('firstPageReceived')

# After the trigger fires, the trigger manager creates an
# instance for ``t_importSubject``, but it can't run yet, because
# it's still waiting for the other trigger.
>>> [<TaskInstance 't_importSubject#0'>]
```

Getting the Full Picture

If you want to get a snapshot of the state of every task and instance, invoke the `debug_repr` method:

```
from pprint import pprint
pprint(trigger_manager.storage.debug_repr())
```

Tip: As the name implies, this is intended to be used only for debugging purposes.

If you find yourself wanting to use it as part of normal operations, this likely indicates a deficiency in the Trigger Manager's feature set; please post a feature request on the Triggers Framework Bug Tracker so that we can take a look!

9.3 Error Recovery

On occasion, a trigger task instance may fail (e.g., due to an uncaught exception).

When this happens, you can recover by replaying or skipping the failed instance(s).

Tip: If the instance fails due to an uncaught exception, the exception and traceback will be stored in the failed instance's metadata so that you can inspect them.

To access these values, find the `TaskInstance` and inspect its `metadata` value:

```
failed_instance =\
  trigger_manager.storage.instances['t_importSubject#0']

pprint(failed_instance.metadata)
```

9.3.1 Replaying Failed Task Instances

To replay a failed task invoke the trigger manager's `replay_failed_instance()` method, e.g.:

```
trigger_manager.replay_failed_instance('t_importSubject#0')
```

Note that you must provide the name of the **instance** that failed, not the **task**.

The trigger manager will *clone the failed instance* and schedule it for execution immediately.

The failed instance's status will be changed to "replayed" (see *Task Instance Status*), but otherwise it remains unchanged. This allows you to trace the history of a failed task, retain the original exception details, etc.

If necessary/desired, you may replay the instance with different trigger kwargs:

```
trigger_manager.replay_failed_instance(
  failed_instance = 't_importSubject#0',

  replacement_kwargs = {
    'firstPageReceived':      {'responses': {...}},
    'questionnaireComplete':  {},
  },
)
```

Important: The replacement kwargs will be used *instead of* the trigger kwargs provided to the failed instance. If you only want to change some of the trigger kwargs for the replayed instance, you will need to merge them manually.

Example:

```
failed_instance =\
  trigger_manager.storage.instances['t_importSubject#0']

# Change the ``firstPageReceived`` trigger kwargs
# for the replay, but keep the rest the same.
replacement_kwargs = failed_instance.kwargs
replacement_kwargs['firstPageReceived'] = {'responses': {...}}

trigger_manager.replay_failed_instance(
  failed_instance,
  replacement_kwargs,
)
```

9.3.2 Skipping Failed Task Instances

Sometimes there is just no way to recover a failed task instance, but you still want to mark it as resolved, or to simulate a successful result so that other tasks can still run (i.e., simulate a *cascade*).

To accomplish this, invoke the `skip_failed_instance()` method:

```
trigger_manager.skip_failed_instance('t_importSubject#0')
```

Note that you must provide the name of the **instance** that failed, not the **task**.

The trigger manager will change the status of the instance from "failed" to "skipped" (see *Task Instance Status*).

By default, marking a failed instance as skipped will not cause a cascade, so any tasks that depend on the failed one won't be able to run.

In many cases, this is actually the desired behavior, but if you would like to force a cascade anyway, you can simulate a successful result:

```
trigger_manager.skip_failed_instance(
  failed_instance = 't_importSubject#0',

  # Trigger a cascade.
  cascade = True,

  # Simulate the result from ``t_importSubject#0``.
  result = {'subjectId': 42},
)
```

The above code has basically the same effect as if the `t_importSubject#0` instance finished successfully and caused a cascade:

```
trigger_manager.fire(
  trigger_name    = 't_importSubject',
  trigger_kwargs  = {'subjectId': 42},
)
```

CHAPTER 10

Logging

The trigger task's *Task Context* provides a number of objects and methods that are important to help a trigger task do its job properly.

One of its most critical features is creating a logger, via its `get_logger_context()` method.

Typically, `get_logger_context()` is the first statement in the task body:

```
from triggers.task import TaskContext, TriggerTask

class ImportSubject(TriggerTask):
  def _run(self, context):
    # type: (TaskContext) -> dict
    with context.get_logger_context() as logger:
      ...
```

The resulting `logger` instance acts like a regular `logging.Logger` object, with a couple of notable differences:

- You can attach "context" variables to the logger and log messages.
- The max log level emitted by this logger is recorded by the trigger manager for later reference.

10.1 Context Variables

Oftentimes, it is difficult to convey all of the desired information in a log message. Developers often have to resort to workarounds such as tacking reprs of critical state values onto the end of the log message.

However, this results in long, unformatted text dumps that are a pain to sift through and contribute significantly to warning fatigue.

`get_logger_context()` tackles this problem in a different way.

When emitting a log level, your task may optionally attach a "context" object to the log message, like this:

```
from my_app.models import Subject
from triggers.task import TaskContext, TriggerTask

class ImportSubject(TriggerTask):
  def _run(self, context):
    # type: (TaskContext) -> dict
    with context.get_logger_context() as logger:
      page_data =\
        context.trigger_kwargs['firstPageReceived']['responses']

      given_names = page_data.get('givenNames')
      if not given_names:
        logger.warning(
          'Missing givenNames in response data.',

          # Attach the ``page_data`` to the log message via its context dict.
          extras={'context': {
            'page_data': page_data,
          }},
        )
```

In the above example, a missing or empty `givenNames` value in the response data is notable enough to warrant a warning message, but not an exception.

When troubleshooting this issue, it may be useful for a developer to have a full readout of the page data. Rather than try to include this (potentially massive) value in the log message itself, the code attaches it to the log's `context` dict.

Note: Depending on how your application processes log messages, you may need to configure your log formatter(s) specifically to take advantage of this feature.

Review the logging module documentation for more information.

Tip: You can also provide a dict directly to `get_logger_context()`. These context values will be attached automatically to every log message:

```
from my_app import __version__

class ImportSubject(TriggerTask):
  def _run(self, context):
    # type: (TaskContext) -> dict

    extra_context = {
      "app_version": __version__,
    }

    with context.get_logger_context(extra_context) as logger:
      # The application version number will be attached to every
      # log emitted by ``logger``.
      ...
```

10.1.1 Exception Context

As with log messages, you can also attach context values to exceptions that your task raises.

To use this feature, pass the exception to `triggers.exceptions.with_context()` before raising it.

As an example, suppose we wanted to add some kind of a spam filter to our `ImportSubject` trigger task:

```
from triggers.exceptions import with_context
from triggers.task import TaskContext, TriggerTask

class ImportSubject(TriggerTask):
  def _run(self, context):
    # type: (TaskContext) -> dict
    with context.get_logger_context() as logger:
      ...
      spam_score = ...
      if spam_score < threshold:
        raise with_context(
          exc = ValueError("Response data failed spam check."),

          context = {
            'spam_score': spam_score,
            'threshold': threshold,
          },
        )
```

The actual spam score and threshold are interesting information, but it might not be that helpful to include them in the exception message itself (how often do you check those values when your email application flags an email as spam)?

Still, it's useful to attach them to the exception to assist with any troubleshooting efforts.

`with_context()` facilitates this.

Important: The exception will only get logged if it is raised inside of the `get_logger_context()` block!

10.2 Tracking Log Levels

The logger returned by `get_logger_context()` also keeps track of the max log level emitted inside of that context.

This enables your application to track task instance failure/success with a finer degree of granularity.

For example, if you integrate a custom trigger manager with logic to *"finalize" a session*, you may opt to have it only finalize the session only if none of the task instances emitted log messages with `WARNING` or higher level.

10.2.1 Task Instance Log Level

Once a task instance has finished running (successfully or otherwise), the max log level emitted is stored in its `log_level` property:

```
task_instance = trigger_manager.storage['t_importSubject#0']

task_instance.log_level        # e.g.: logging.INFO
task_instance.log_level_name   # e.g.: 'INFO'
```

Note: If the task instance hasn't finished running yet, its `log_level` will be `NOTSET`.

10.2.2 Resolving Logs

In some cases, it may be necessary to mark a task instance's logs as "resolved".

For example, a task instance may emit a `WARNING` or `ERROR` log, but the application determines that these logs are no longer relevant (e.g., a user reviewed them and addressed any issues manually).

To resolve an instance's logs use the `mark_instance_logs_resolved()` method:

```
trigger_manager.mark_instance_logs_resolved('t_importSubject#0')
```

CHAPTER 11

Testing Trigger Tasks

Writing and running unit tests for trigger tasks can be a bit tricky because they are designed to be executed by Celery.

Fortunately, the Triggers framework comes with a unit testing toolbox that makes it super easy to write tests for your trigger tasks!

11.1 Test Cases

When writing a test case for a trigger task, ensure that it:

1. Derives from `triggers.testing.TriggerManagerTestCaseMixin`, and
2. Initializes `self.manager` in its `setUp()` method.

```
from triggers.testing import TriggerManagerTestCaseMixin
from unittest import TestCase

class ImportSubjectTestCase(TriggerManagerTestCaseMixin, TestCase):
  def setUp(self):
    super(TriggerTaskTestCase, self).setUp()

    self.manager =\
      TriggerManager(CacheStorageBackend(self._testMethodName))
```

Tip: If you are using a persistent storage backend, make sure to clear it before each test.

11.2 Tests

When writing individual tests, they should conform to the following structure:

1. Configure trigger tasks.

2. Fire triggers.
3. Wait for tasks to complete.
4. Perform assertions.

Here's an example:

```
from my_app.models import Subject
from triggers.runners import ThreadingTaskRunner
from triggers.testing import TriggerManagerTestCaseMixin
from unittest import TestCase

class ImportSubjectTestCase(TriggerManagerTestCaseMixin, TestCase):
  def setUp(self):
    super(TriggerTaskTestCase, self).setUp()

    self.manager =\
      TriggerManager(CacheStorageBackend(self._testMethodName))

def test_successful_import(self):
  """
  Successfully importing a new subject record.
  """
  # Configure trigger tasks.
  self.manager.update_configuration({
    't_importSubject': {
      'after': ['firstPageReceived', 'questionnaireComplete'],
      'run': 'app.tasks.ImportSubject',
    },
  })

  responses = {
    'firstName': 'Marcus',
    # etc.
  }

  # Fire triggers (in this case, simulating successful
  # questionnaire completion).
  self.manager.fire(
    trigger_name   = 'firstPageReceived',
    trigger_kwargs = {'responses': responses},
  )

  self.manager.fire('questionnaireComplete')

  # Wait for tasks to complete.
  ThreadingTaskRunner.join_all()

  # Perform assertions.
  subject = Subject.objects.latest()

  self.assertInstanceFinished(
    't_importSubject#0',
    {'subjectId': subject.pk},
  )

  self.assertEqual(subject.firstName, responses['firstName'])
  # etc.
```

11.2.1 1. Configure trigger tasks.

At the start of each test (or in your test case's `setUp()` method), configure the trigger task(s) that you want to execute during the test.

This is done using the trigger manager's `update_configuration()` method. For example:

```
self.manager.update_configuration({
  't_importSubject': {
    'after': ['firstPageReceived', 'questionnaireComplete'],
    'run': 'app.tasks.ImportSubject',
  },
})
```

Note that this is the same code that your application uses to *initialize a triggers session*.

Tip: You can configure multiple trigger tasks in a single test.

This can be used to test entire workflows, not just individual trigger tasks.

11.2.2 2. Fire triggers.

Once the trigger manager has been configured, the next step is to fire triggers that cause your trigger tasks to get run, exactly the same as the application would under normal (or – depending on the test – abnormal) conditions.

For example:

```
self.manager.fire(
  trigger_name   = 'firstPageReceived',
  trigger_kwargs = {'responses': responses},
)

self.manager.fire('questionnaireComplete')
```

11.2.3 3. Wait for tasks to complete.

During unit tests, the trigger manager will automatically use `ThreadingTaskRunner` to execute unit tests. This means that your trigger tasks will be run in separate threads instead of using Celery workers.

This process is still asynchronous, however, so it is very important that your test waits until all of the tasks have finished running (including any tasks that may have been executed as a result of *cascading*) before it begins performing assertions.

To accomplish this, include a call to `ThreadingTaskRunner.join_all()` immediately after firing triggers:

```
from triggers.runners import ThreadingTaskRunner

...

self.manager.fire(...)
self.manager.fire(...)
self.manager.fire(...)
ThreadingTaskRunner.join_all()
```

Tip: You can call `ThreadingTaskRunner.join_all()` multiple times in the same test, if necessary.

11.2.4 4. Perform assertions.

Finally, once all of the trigger tasks have finished, you can begin adding assertions to the test.

There are two things in particular that your test should check:

a. Trigger task instance state.

Because trigger tasks run asynchronously, it is important to first verify that each task instance has the expected status.

For example, if a trigger task fails with an exception or if it didn't get run, it will be easiest to determine this by checking the task instance's status.

To facilitate this, `TriggerManagerTestCaseMixin` provides several custom assertions:

`assertInstanceAbandoned()` Given an instance name, checks that the corresponding instance was abandoned (i.e., its `unless` clause was satisfied before it could be run).

`assertInstanceFailed()` Given an instance name and exception type, checks that the corresponding instance failed with the specified exception type.

`assertInstanceFinished()` Given an instance name and (optional) result dict, checks that the corresponding instance finished successfully and returned the specified result.

`assertInstanceMissing()` Given an instance name, checks that the corresponding instance hasn't been created yet (i.e., none of its triggers have fired yet).

`assertInstanceReplayed()` Given an instance name, checks that the corresponding instance was replayed.

`assertInstanceSkipped()` Given an instance name, checks that the corresponding instance was skipped.

`assertInstanceUnstarted()` Given an instance name, checks that the corresponding instance is in unstarted state (i.e., not all of its triggers have fired yet).

`assertUnresolvedTasks()` Given a list of trigger task (not instance!) names, asserts that the corresponding tasks are unresolved:

- Have one or more instances in an unresolved state (e.g., unstarted, failed, etc.), or
- None of its triggers have fired yet.

`assertUnresolvedInstances()` Given a list of instance names, asserts that the corresponding instances are unresolved.

> **Note:** This method only checks instances where at least one of their triggers have fired.
>
> `assertUnresolvedTasks()` is better at detecting tasks that are unresolved because none of their triggers have fired yet.

Tip: If an instance has the wrong status, the test failure message will include additional information that will make it easier to figure out what went wrong (e.g., traceback from the exception, etc.).

Some examples:

```
# Check that the task instance finished successfully.
# Note that we provide the name of the *instance*, not the *task*
# (hence the ``#0`` suffix):
self.assertInstanceFinished(
  instance_name   = 't_importSubject#0',
  expected_result = {'subjectId': 42},
)

# Check that the task instance failed with the expected error:
from requests.exceptions import Timeout
self.assertInstanceFailed(
   instance_name  = 't_importBrowserMetadata#0',
   exc_type       = Timeout,
)

# Check that an instance retried automatically on error (until it hit
# ``max_retries``):
self.assertInstanceReplayed('t_importBrowserMetadata#0')
self.assertInstanceReplayed('t_importBrowserMetadata#1')
self.assertInstanceFailed('t_importBrowserMetadata#2', Timeout)
```

b. Effects from the trigger tasks.

After checking that all of the trigger tasks finished (or failed) as expected, then add assertions verifying the tasks' effects.

These assertions include tasks such as checking for the presence of database records, checking whether emails were sent, etc.

CHAPTER 12

Trigger Managers

The trigger manager acts as the controller for the Triggers framework. Working in conjunction with a *storage backend*, it provides an interface for effecting changes on a triggers session.

From the *Basic Concepts* documentation, you can see that initializing a trigger manager is fairly straightforward:

```
from triggers import TriggerManager

trigger_manager = TriggerManager(storage_backend)
```

Where `storage_backend` is a *storage backend*.

12.1 Interacting with Trigger Managers

Trigger managers provide the following methods:

`update_configuration(configuration)()` Initializes or updates the trigger task configuration. See *Writing Celery Tasks* for more information.

`fire(trigger_name, [trigger_kwargs])()` Fires a trigger. See *Getting Started* for more information on how to use this method.

`replay_failed_instance(failed_instance, [replacement_kwargs])()` Given a *failed* task instance, creates a copy and attempts to run it. If desired, you can provide replacement kwargs, if the original task failed due to an invalid kwarg value.

See *Replaying Failed Task Instances* for more information.

Note: As the name implies, only failed instances can be replayed.

`skip_failed_instance(failed_instance, [cascade], [result])()` Given a *failed* task instance, marks the instance as skipped, so that it is considered to be *resolved*.

If desired, you may also specify a fake result for the task instance, to trigger a *cascade*.

See *Skipping Failed Task Instances* for more information.

Note: As the name implies, only failed instances can be skipped.

`update_instance_status(task_instance, status, [metadata], [cascade], [cascade_kwargs])()`
Manually changes the status for a task instance. This method can also be used to trigger a *cascade*.

`update_instance_metadata(task_instance, metadata)()` Manually update the metadata for a task instance. This method can be used to attach arbitrary data to a task instance for logging/troubleshooting purposes.

`mark_instance_logs_resolved(task_instance)()` Given a task instance, updates its metadata so that its *log messages are resolved*.

12.2 Writing Custom Trigger Managers

You can customize the behavior of the trigger manager(s) that your application interacts with.

For example, you can write a custom trigger manager that contains additional logic to *finalize sessions*.

Your trigger manager must extend the `triggers.manager.TriggerManager` class.

There is only one attribute that must be implemented in order to create a custom trigger manager:

`name: Text` A unique identifier for your trigger manager.

Generally this matches the name of the trigger manager's entry point in your project's `setup.py` file (see below).

12.2.1 Hooks

Whenever the base trigger manager completes certain actions, it invokes a corresponding hook, which you can override in your custom trigger manager.

The following hooks are supported:

`_post_fire(trigger_name, tasks_scheduled)()` Invoked after processing a call to `fire()`. It receives the name of the trigger that was fired, and a list of any task instances that were scheduled to run as a result.

`_post_replay(task_instance)()` Invoked after processing a call to `replay_failed_instance()`. It receives the **replayed** task instance.

Tip: You can find the failed instance by inspecting the replayed instance's metadata and extracting the `parent` item:

```
def _post_replay(task_instance)
  # type: (TaskInstance) -> NoReturn
  parent_name = task_instance.metadata['parent']  # type: Text
  parent_instance = self.storage[parent_name]  # type: TaskInstance
```

`_post_skip(task_instance, cascade)()` Invoked after processing a call to `skip_failed_instance()`. It receives the skipped task instance, and a boolean indicating whether a cascade was simulated.

Note: This method gets invoked **after** the cascade happens (i.e., after `_post_fire()` is invoked).

12.2.2 Registering Your Trigger Manager

Because of the way *trigger tasks* work, you must register your custom trigger manager in order for it to work correctly.

To do this, you must create a custom entry point.

In your project's `setup.py` file, add a `triggers.managers` entry point for your custom trigger manager.

For example, if you wanted to register `app.triggers.CustomManager`, you would add the following to your project's `setup.py` file:

```
from setuptools import setup

setup(
  ...

  entry_points = {
    'triggers.managers': [
      'custom_manager = app.triggers:CustomManager',
    ],
  },
)
```

Tip: Any time you make changes to `setup.py`, you must reinstall your project (e.g., by running `pip install -e .` again) before the changes will take effect.

Once you've registered your trigger manager, you can then use it in your application:

```
from app.triggers import CustomManager
from triggers import CacheStorageBackend

trigger_manager =\
  CustomManager(CacheStorageBackend(session_uid))
```

Important: Make sure that your application always uses the same trigger manager (unless you are 110% sure you know what you are doing).

CHAPTER 13

Storage Backends

The storage backend's job is to manage the state for a session and to load and save data from a (usually) permanent storage medium, such as a database or filesystem.

The Triggers framework ships with a single storage backend that uses the Django cache to persist data. However, you can write and use your own storage backend if desired.

13.1 Anatomy of a Storage Backend

A session state is comprised of 3 primary components:

`tasks: Dict[Text, TaskConfig]` This is effectively the same dict that was provided to the trigger manager's `update_configuration()` method. Keys are the task names (e.g., `t_importSubject`), and values are `triggers.types.TaskConfig` objects.

`instances: Dict[Text, TaskInstance]` Contains all of the *task instances* that have been created for this session. Keys are the instance names (e.g., `t_importSubject#0`), and values are `triggers.types.TaskInstance` objects.

> **Note:** Task instances may appear in this dict even if they haven't run yet.

`metadata: Dict` Contains any additional metadata that the trigger manager and/or storage backend needs in order to function properly. For example, `metadata` keep track of all of the triggers that have fired during this session, so that the trigger manager can initialize instances for *tasks that run multiple times*.

13.1.1 Working with Session State

The storage backend provides a number of methods for interacting with trigger tasks and instances:

`__getitem__(task_instance)` Returns the task instance with the specified name. For example:

```
task_instance = trigger_manager.storage['t_importSubject#0']
```

`__iter__(task_instance)` Returns an iterator for the task instances. Order is undefined. For example:

```
for task_instance in iter(trigger_manager.storage):
  ...
```

`create_instance(task_config, **kwargs)` Creates a new instance of the specified trigger task.

Additional keyword arguments are passed directly to the `TaskInstance` initializer.

`clone_instance(task_instance)` Given a task instance name or `TaskInstance` object, creates and installs a copy into the trigger session.

Tip: This method is used internally when *replaying a failed task*.

`get_instances_with_unresolved_logs()` Unsurprisingly, returns all task instances with *unresolved logs*.

`get_unresolved_instances()` Returns all task instances with *unresolved status*.

`get_unresolved_tasks()` Returns all trigger tasks that either:

- Do not have any instances yet, or
- Have at least one task instance with unresolved status.

`instances_of_task(task_config)` Returns all task instances that have been created for the specified trigger task.

Note: The storage backend contains several more methods, but they are intended to be used internally.

13.2 Writing Your Own Storage Backend

To create your own storage backend, you only need to define methods to load and save the session data; the base class will take care of everything else for you.

Your backend must extend the `triggers.storages.base.BaseTriggerStorage` class and implement the following attributes/methods:

`name: Text` A unique identifier for your storage backend.

Generally this matches the name of the storage's entry point in your project's `setup.py` file (see below).

`_load_from_backend(self)` Given `self.uid`, loads the corresponding session data from the persistence medium.

This method should return a tuple with three values:

- Item 0 contains the trigger task configurations.
- Item 1 contains the task instances.
- Item 2 contains the session metadata.

Tip: These values do not have to be stored together, as long as the `_load_from_backend()` method knows how to consolidate them.

_save(self) Given `self.uid`, saves the corresponding session data to the persistence medium.

This method should be sure to save the following values:

- `self._configs`: Trigger task configurations.
- `self._instances`: Trigger task instance.
- `self._metas`: Session metadata.

Note the leading underscore on each of these attributes.

Tip: To serialize values for storage, use the `self._serialize()` method.

For more information and examples, look at the implementation of `triggers.storages.cache.CacheStorageBackend`.

13.2.1 Registering Your Storage Backend

As with trigger managers, you must register your custom storage backend before it can be used.

To do this, define a `triggers.storages` entry point in your project's `setup.py` file:

```
from setuptools import setup

setup(
  ...

  entry_points = {
    'triggers.storages': [
      'custom_storage = app.triggers:CustomStorageBackend',
    ],
  },
)
```

Tip: Any time you make changes to `setup.py`, you must reinstall your project (e.g., by running `pip install -e .` again) before the changes will take effect.

Once you've registered your trigger storage backend, you can then use it in your application:

```
from app.triggers import CustomStorageBackend
from triggers import TriggerManager

trigger_manager =\
  TriggerManager(CustomStorageBackend(session_uid))
```

Important: Make sure that your application always uses the same storage backend (unless you are 110% sure you know what you are doing).

CHAPTER 14

Task Runners

When the trigger manager determines that a task instance is ready to run, it instantiates a runner to handle the execution.

By default, this runner uses Celery (`triggers.runners.CeleryTaskRunner`), but you can customize this.

For example, during *unit tests*, the trigger manager will use `triggers.runners.ThreadingTaskRunner` instead.

14.1 CeleryTaskRunner

As the name implies, `CeleryTaskRunner` executes task instances using Celery.

Each trigger task is implemented as a *Celery task*, and when the trigger manager schedules a task instance for execution, the `CeleryTaskRunner` will schedule a matching Celery task for execution.

Tip: You can leverage Celery's router to send tasks to different queues, just like regular Celery tasks.

14.2 ThreadingTaskRunner

`ThreadingTaskRunner` operates completely independently from Celery. Instead of sending tasks to the Celery broker, it executes each task in a separate thread.

Generally, this runner is only used during *testing*, but in certain cases, it may be useful to utilize this runner in other contexts.

Tip: If you need your application to wait for all running tasks to finish before continuing, invoke `ThreadingTaskRunner.join_all()`.

Note that this will wait for *all* running tasks to finish (including any *cascades* that may occur).

It is not possible (nor in line with the philosophy of the triggers framework) to wait for a particular task to finish before continuing. If you need certain logic to run after a particular task finishes, it is recommended that you implement that logic as a separate task that is triggered by a cascade from the first task.

14.3 Writing Your Own Task Runner

As with *Trigger Managers* and *Storage Backends*, you can inject your own task runners into the Triggers framework.

14.3.1 Anatomy of a Task Runner

A task runner must extend `triggers.runners.BaseTaskRunner`. The base class declares the following attributes/methods that you must implement in your custom task runner:

`name: Text` A unique identifier for your task runner.

Generally this matches the name of the task runner's entry point in your project's `setup.py` file (see below).

`run(self, manager: TriggerManager, task_instance: TaskInstance) -> NoReturn`

Given a trigger manager and task instance, finds the correct Celery task (i.e., using the `resolve()` method) and executes it.

Tip: See `ThreadingTaskRunner.run()` for a sample implementation.

14.3.2 Registering Your Task Runner

As with trigger managers, you must register your custom task runner before it can be used.

To do this, define a `triggers.runners` entry point in your project's `setup.py` file:

```
from setuptools import setup

setup(
  ...

  entry_points = {
    'triggers.runners': [
      'custom_runner = app.triggers:CustomRunner',
    ],
  },
)
```

Tip: Any time you make changes to `setup.py`, you must reinstall your project (e.g., by running `pip install -e .` again) before the changes will take effect.

14.3.3 Using Your Task Runner

Unlike *Trigger Managers* and *Storage Backends*, your application does not select the task runner directly.

Instead, the task runner is configured via one of two methods (in descending order of priority):

1. In the trigger task's *using* clause.

   Add your custom task runner to each task's configuration:

   ```
   from app.triggers import CustomRunner

   trigger_manager.update_configuration({
     't_importSubject': {
       ...
       'using': CustomRunner.name,
     },
     ...
   })
   ```

   Tip: This approach is useful if you only want some of your tasks to use the custom task runner (whereas the rest should use e.g., the default `CeleryTaskRunner`).

2. Via the trigger manager's `default_task_runner_name` property.

   In order for this to work correctly, you must subclass `TriggerManager`:

   ```
   class CustomTriggerManager(TriggerManager):
     name = 'custom'
     default_task_runner_name = CustomRunner.name

   trigger_manager = CustomTriggerManager(...)
   tirgger_manager.fire(...)
   ```

   Important: Don't forget to *register your custom trigger manager*!

CHAPTER 15

Cookbook

This page describes some strategies for customizing the behavior of the Triggers framework, depending on the needs of your application.

15.1 Setting the Manager/Storage Type at Runtime

Internally, the Triggers framework uses a library called ClassRegistry to manage the registered trigger managers and storage backends. ClassRegistry works by assigning each class a unique key and adding it to a registry (dict-like object).

You can leverage this feature in your application to make the manager and/or storage type configurable at runtime, by storing the corresponding keys in application settings (e.g., in a Django `settings.py` module).

Here's an example:

First, we set some sensible defaults:

```
# my_app/settings.py

TRIGGER_MANAGER_TYPE = 'default'
TRIGGER_STORAGE_TYPE = 'cache'
```

Tip: The values `'default'` and `'cache'` can be found in the entry point definitions for `TriggerManager` and `CacheStorageBackend`, respectively.

Entry point definitions are set in the library's setup.py; look for the `entry_points` configuration.

See *Registering Your Trigger Manager* for more information.

Next, we'll define a function that will build the trigger manager object from these settings:

```
# my_app/triggers.py
```

```python
from typing import Text
from triggers import TriggerManager
from triggers.manager import trigger_managers
from triggers.storages import storage_backends

from my_app.settings import TRIGGER_MANAGER_TYPE, \
  TRIGGER_STORAGE_TYPE

def get_trigger_manager(uid):
  # type: (Text) -> TriggerManager
  """
  Given a session UID, returns a configured trigger manager.
  """
  storage = storage_backends.get(TRIGGER_STORAGE_TYPE, uid)
  manager = trigger_managers.get(TRIGGER_MANAGER_TYPE, storage)

  return manager
```

Note the use of `triggers.manager.trigger_managers` and `triggers.storages.storage_backends`. These are the registries of trigger managers and storage backends, respectively.

The `get()` method retrieves the class corresponding to the identifier (e.g., `TRIGGER_STORAGE_TYPE` — "cache" in this case) and instantiates it using the remaining arguments (e.g., `uid`).

Finally, call `get_trigger_manager()` in your application wherever you need a `TriggerManager` instance.

By changing the values of the `TRIGGER_MANAGER_TYPE` and/or `TRIGGER_STORAGE_TYPE` settings, you can customize the trigger manager and/or storage backend that your application uses, without having to rewrite any logic.

15.2 Finalizing a Session

In many cases, it is useful to schedule trigger tasks to run when everything else is finished.

For example, we may want to have our questionnaire application set a status flag in the database once the questionnaire is 100% complete and all of the other trigger tasks have finished successfully.

To make this work, we will define a new trigger called `sessionFinalized` that fires when all of the trigger tasks in a session have finished running.

We can detect that a trigger task has finished running by waiting for its *cascade*; that is, we can perform the "is session finalized" check after each trigger fires.

To accomplish this, we must create our own *trigger manager* and override its `_post_fire()` hook.

We will also take advantage of the trigger manager's ability to *find unresolved tasks*, so that we can determine if there are any tasks waiting to run.

The end result looks like this:

```python
class FinalizingTriggerManager(TriggerManager):
  TRIGGER_SESSION_FINALIZED = "sessionFinalized"

  def _post_fire(self, trigger_name, tasks_scheduled):
    # Prevent infinite recursion.
    if trigger_name == self.TRIGGER_SESSION_FINALIZED:
      return

    # A session can only be finalized once.
    if self.TRIGGER_SESSION_FINALIZED in self.storage.latest_kwargs:
      return

    # Check for any unresolved tasks...
    for config in self.get_unresolved_tasks():
      # ... ignoring any that are waiting for session finalized.
      if self.TRIGGER_SESSION_FINALIZED not in config.after:
        return

    # If we get here, we are ready to finalize the session.
    self.fire(self.TRIGGER_SESSION_FINALIZED)
```

Important: Don't forget to *register your trigger manager*!

15.3 Namespaced Session UIDs

Suppose you have a set of related triggers sessions, and you want to schedule some tasks to run in a "super session" of sorts.

For example, let's suppose that our questionnaire application has two different questionnaires: "Flora" and "Fauna". We would like to execute a trigger task after the applicant completes page 3 of the Flora questionnaire and page 6 of the Fauna questionnaire. But, we can't predict what order these events will occur.

To accomplish this, we can create a "namespaced session UID" for the applicant. When the application is processing responses from the applicant's questionnaire, it will actually create *two* trigger managers, each with a separate UID:

```
from my_app.models import Questionnaire

def start_questionnaire(request):
  """
  Django view that is called when the user clicks the "start" button
  on a questionnaire.
  """
  questionnaire = get_object_or_404(
    klass = Questionnaire,
    pk    = request.POST["questionnaire_id"],
  )

  # Prepare our regular triggers session for the questionnaire.
  trigger_manager = TriggerManager(...)
  trigger_manager.update_configuration(...)

  # Prepare our "super session", which will maintain state across
  # multiple questionnaires.
  #
  # Note that the UID is tied to the applicant, not a particular
  # questionnaire.  We also add a prefix, to avoid conflicts with
  # regular trigger session UIDs.
  super_trigger_manager = TriggerManager(
    storage = CacheStorageBackend(
      uid = 'applicant:{}'.format(request.session.applicant_id),
    ),
  )
  super_trigger_manager.update_configuration({
    # This task will run after the applicant completes page 3 in the
    # Flora questionnaire, and page 6 in the Fauna questionnaire.
    't_compareResponses': {
      'after': ['flora_page3', 'fauna_page6'],
      'run': CompareResponses.name,
    },
  })

def responses(request):
  """
  Django view that processes a page of response data from
  the client.
  """
  questionnaire = get_object_or_404(
    klass = Questionnaire,
    pk    = request.POST["questionnaire_id"],
  )

  responses_form = QuestionnaireResponsesForm(request.POST)
  if responses_form.is_valid():
    # Regular triggers session for the questionnaire.
    trigger_manager = TriggerManager(...)
    trigger_manager.fire(...)

    # Fire triggers for "super session".
    super_trigger_manager = TriggerManager(
    storage = CacheStorageBackend(
        uid = 'applicant:{}'.format(request.session.applicant_id),
      ),
    )

    super_trigger_manager.fire(
      # E.g., "fauna_page3", etc.
      trigger_name = '{}_page{}'.format(
        questionnaire.name,
        responses_form.cleaned_data['page_number'],
      ),

      trigger_kwargs = {'responses': responses_form.cleaned_data},
    )
```

CHAPTER 16

Triggers

The Triggers framework is an implementation of the observer pattern, designed for distributed stacks.

It allows you to configure and execute asynchronous tasks based on events that are triggered by your application.

For example, suppose you have a survey application, and you want an asynchronous task to run after the user completes steps 1 and 4.

However, you can't guarantee…

- … that the same server will process both steps.
- … that both steps will arrive in the correct order.
- … whether both steps will arrive separately, or at the same time.

The Triggers framework provides a flexible solution that empowers you to schedule an asynchronous task in such a way that you can guarantee it will be executed after steps 1 and 4 are completed.

But, it doesn't stop there! You can also:

- Configure tasks to wait until other asynchronous tasks have finished.
- Define conditions that will cause a task to run multiple times.
- Define conditions that will prevent a task from running.
- Write functional tests to verify that an entire workflow runs as expected.
- And more!

CHAPTER 17

Prerequisites

The Triggers framework requires:

- Python 2.7, 3.5 or 3.6
- Django (any version, but >= 1.11 preferred)
- Celery (>= 3, but >= 4 preferred)
- django-redis-cache
- python-redis-lock==2.3.0

Currently, the Triggers framework requires Redis in order to function properly, but we are working on removing this requirement in a future version of the framework.

Note that you do not have to use Redis for your primary application cache; you can continue to use your preferred cache backend for your `default` cache in Django. You'll just need to configure a separate cache connection for the Triggers framework.

At the moment, `python-redis-lock` must be at v2.3.0; versions later than this cause deadlocks. We are looking into why this is happening and will remove the version requirement once the issue is resolved.

CHAPTER 18

Installation

Install the Triggers framework using pip:

```
pip install triggers
```

You can also install from source using the following commands:

```
pip install -e git+https://github.com/eflglobal/triggers
```

18.1 Running Unit Tests

To run unit tests after installing from source, you will need to do a little bit of one-time prep:

```
pip install -e '.[test-runner]'
cp tests/settings.py.dist tests/settings.py
```

Tip: By default, the unit tests expect a Redis server listening on `localhost:6379`. If necessary, you can change this by editing `tests/settings.py`.

Once you've set up the test environment, you can run the unit tests with the following command:

```
python manage.py test
```

This project is also compatible with tox, which will run the unit tests in different virtual environments (one for each supported version of Python).

To run the unit tests, it is recommended that you use the detox library. detox speeds up the tests by running them in parallel.

Install the package with the `test-runner` extra to set up the necessary dependencies, and then you can run the tests with the `detox` command:

```
pip install -e '.[test-runner]'
detox -v
```

Important: Currently, `tox.ini` uses `tests/settings.py.dist` for the test settings. In particular, this means that all of the unit tests run by tox depend on having a Redis server listening on `localhost:6379`, and there is currently no way to change this (without voiding the warranty, that is).

This will be fixed in a future version of the library.

CHAPTER 19

Documentation

The Triggers framework documentation is available on ReadTheDocs.

If you are installing from source (see above), you can also build the documentation locally:

1. Install extra dependencies (you only have to do this once):

   ```
   pip install '.[docs-builder]'
   ```

2. Switch to the `docs` directory:

   ```
   cd docs
   ```

3. Build the documentation:

   ```
   make html
   ```