Events & Worker Queues

Software Architecture

April 5, 2023

Evan Hughes & Brae Webb

1 This Week

Our goal is to:

- Explore events and worker queues in the context of AWS.
- Deploy our own TaskOverflow container on AWS Elastic Container Registry (ECS).
- Develop an event-based architecture to generate a calendar from tasks in TaskOverflow.

2 Event Processing

As we saw in Event-Driven Architectures [1], event processing enable us to build highly scalable and extensible systems. In this practical we will get our hands dirty with event processing using AWS SQS. AWS

SQS is a service which acts as an event broker.

2.1 Technologies

There are other technologies that can be useful in developing an event-based architecture. The following is a non-exhaustive list of services native to AWS.

2.1.1 AWS SQS

AWS provides the Simple Queue Service, SQS, which offers a simple and fully managed message queue service. There are two flavours of SQS to be aware of.

Standard message queues allow for greater scalability by providing higher through-put. However, standard message queues in SQS are not exactly queues, messages are not first in first out, they are best-effort ordered.

FIFO message queues guarantees that messages are First in First Out.

2.1.2 AWS SNS

> Amazon Simple Notification Service (Amazon SNS) is a fully managed messaging service for both application-to-application (A2A) and application-to-person (A2P) communication.
>
> The A2A pub/sub functionality provides topics for high-throughput, push-based, many-to-many messaging between distributed systems, microservices, and event-driven serverless applications. Using Amazon SNS topics, your publisher systems can fanout messages to a large number of subscriber systems, including Amazon SQS queues, AWS Lambda functions, HTTPS endpoints, and Amazon Kinesis Data Firehose, for parallel processing. The A2P functionality enables you to send messages to users at scale via SMS, mobile push, and email.
>
> – AWS

2.1.3 AWS MQ / Apache ActiveMQ / RabbitMQ

> Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ that makes it easy to set up and operate message brokers on AWS. Amazon MQ reduces your operational responsibilities by managing the provisioning, setup, and maintenance of message brokers for you. Because Amazon MQ connects to your current applications with industry-standard APIs and protocols, you can easily migrate to AWS without having to rewrite code.
>
> – AWS

Aside

Not available in the lab environments.

2.1.4 AWS MSK (Managed Streaming for Apache Kafka)

> Amazon Managed Streaming for Apache Kafka (Amazon MSK) is a fully managed service that enables you to build and run applications that use Apache Kafka to process streaming data. Amazon MSK provides the control-plane operations, such as those for creating, updating, and deleting clusters. It lets you use Apache Kafka data-plane operations, such as those for producing and consuming data It runs open-source versions of Apache Kafka. This means existing applications, tooling, and plugins from partners and the Apache Kafka community are supported

without requiring changes to application code. You can use Amazon MSK to create clusters that use any of the Apache Kafka versions listed under Supported Apache Kafka versions.

– AWS

Aside

Not available in the lab environments.

2.1.5 Redis

Redis, which stands for Remote Dictionary Server, is a fast, open source, in-memory, key-value data store. The project started when Salvatore Sanfilippo, the original developer of Redis, wanted to improve the scalability of his Italian startup. From there, he developed Redis, which is now used as a database, cache, message broker, and queue.

Redis delivers sub-millisecond response times, enabling millions of requests per second for real-time applications in industries like gaming, ad-tech, financial services, healthcare, and IoT. Today, Redis is one of the most popular open source engines today, named the "Most Loved" database by Stack Overflow for five consecutive years. Because of its fast performance, Redis is a popular choice for caching, session management, gaming, leaderboards, real-time analytics, geospatial, ride-hailing, chat/messaging, media streaming, and pub/sub apps.

AWS offers two fully managed services to run Redis. Amazon MemoryDB for Redis is a Redis-compatible, durable, in-memory database service that delivers ultra-fast performance. Amazon ElastiCache for Redis is a fully managed caching service that accelerates data access from primary databases and data stores with microsecond latency. Furthermore, ElastiCache also offers support for Memcached, another popular open source caching engine.

– AWS

Aside

Not available in the lab environments.

3 Talking to the Simple Queue Service (SQS)

Info

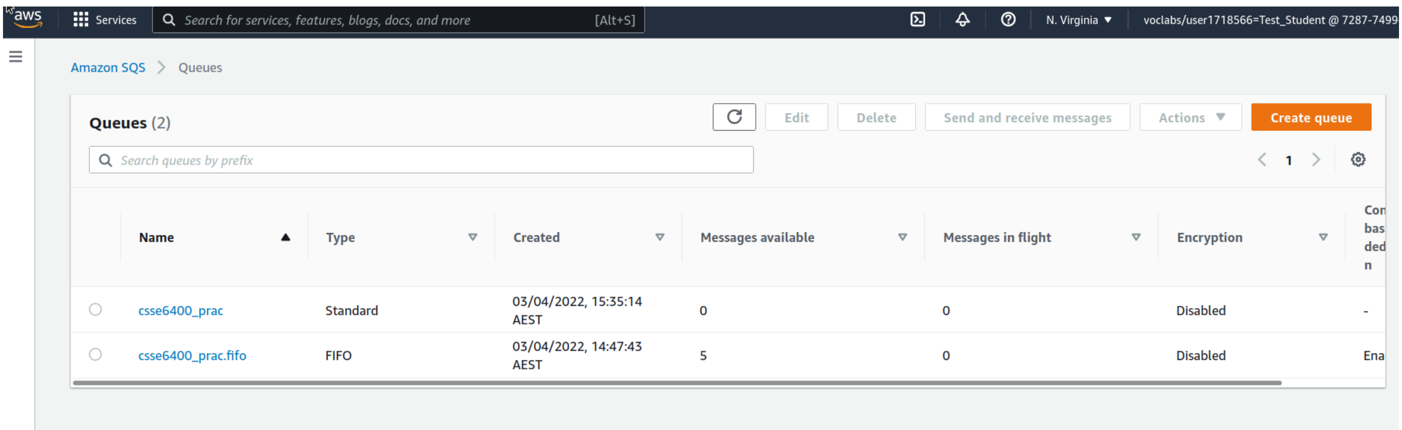
This section is intended to be demonstrated by your practical tutor. You may work through it on your own if you wish.

We will demonstrate working with the two queue flavours of AWS SQS. A standard queue, named `csse6400_prac` and a FIFO queue, named `csse6400_prac.fifo`. The Terraform code below can be used to create these two queues.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
```

```
}

provider "aws" {
  region = "us-east-1"
  shared_credentials_file = "./credentials"
}

resource "aws_sqs_queue" "our_first_mailbox" {
  name = "csse6400_prac"
}

resource "aws_sqs_queue" "our_first_fifo" {
  name = "csse6400_prac.fifo"
  fifo_queue = true
  content_based_deduplication = true
}

output "mailbox" {
  value = aws_sqs_queue.our_first_mailbox.arn
}

output "fifo" {
  value = aws_sqs_queue.our_first_fifo.arn
}
```

Now that we have provisioned the queues we can have a look at them in the AWS Console. In the main AWS dashboard you can search for "SQS" to find these queues. You should reach a page like this:



Like the EC2 and RDS dashboards, we can browse the queue configurations and metrics.

3.1 Queue Command-line Interface

We have provided a small docker container to demonstrate the differnce between the implementations. First we must retrieve our AWS credentials and setup our environment.

Grab the AWS credentials from the Learner Lab but instead of creating a credentials file we will be using environment variables.

Now we need to create an environment file for our docker container to read so that it can access AWS. Create a ".env" file in the current directory and edit the contents so that it looks similar to the below: The AWS keys will be from the credentials shown in your lab environment.

```
TERM=xterm-256color
AWS_ACCESS_KEY_ID=...
AWS_SECRET_ACCESS_KEY=...
AWS_SESSION_TOKEN=...
```

```
> docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "test" --
    client-name "Client 1"



  ----------
 |   \XX/   |
 | T. \/ .T |      University of Queensland
 | XX:  :XX |          Faculty of EAIT
 T L' /\ 'J T
  \  /XX\  /        CSSE6400 Queue Prac
@\_ '____' _/@      csse6400.uqcloud.net
\_X\_ __ _/X_/
 \=/\----/\=/



Unable to find a Queue by this name test
```

3.2 SQS Standard

The "standard" offering of SQS does not guarantee order or "only once delivery". We will create a message publisher and a message subscriber.

Info

For the rest of this practical you will require multiple terminals. Make sure these are all in the same folder so we can reuse the .enf file.

To create the subscriber run the following command:

```
> docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "
    csse6400_prac" --client-name "Client 1" --receive
```

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "csse6400_prac" --cl
ient-name "hello" --receive



      ----------
     |   \XX/   |
     | T. \/ .T |      University of Queensland
     | XX:  :XX |         Faculty of EAIT
     T L' /\ 'J T
      \  /XX\  /        CSSE6400 Queue Prac
    @\_ '____' _/@      csse6400.uqcloud.net
    \_X\_ __ _/X_/
     \=/\----/\=/



Connected to csse6400_prac
⠸ Waiting for messages...
```

Now start a publisher in another terminal.

```
> docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "
   csse6400_prac" --client-name "Client 1"
```

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "csse6400_prac" --cl
ient-name "Client 1"



      ----------
     |   \XX/   |
     | T. \/ .T |      University of Queensland
     | XX:  :XX |         Faculty of EAIT
     T L' /\ 'J T
      \  /XX\  /        CSSE6400 Queue Prac
    @\_ '____' _/@      csse6400.uqcloud.net
    \_X\_ __ _/X_/
     \=/\----/\=/



Connected to csse6400_prac
Sending Messages:
Working... ━━━━━━━━━━────────────────────────────── 25% 0:00:19
```

When the publisher connects to the queue it is going to put 100 messages of increasing increment into the queue. On the subscriber we will be able to see the messages being received, an example is provided below:

```
     | T. \/ .T |      University of Queensland
     | XX:  :XX |          Faculty of EAIT
     T L' /\ 'J T
      \  /XX\  /         CSSE6400 Queue Prac
    @\_ '____' _/@       csse6400.uqcloud.net
    \_X\_ __ _/X_/
     \=/\----/\=/



Connected to csse6400_prac
[11:41:41] Client 1: Message 0                                              main.py:62
           Client 1: Message 1                                              main.py:62
[11:41:42] Client 1: Message 2                                              main.py:62
           Client 1: Message 5                                              main.py:62
           Client 1: Message 6                                              main.py:62
[11:41:43] Client 1: Message 8                                              main.py:62
           Client 1: Message 9                                              main.py:62
[11:41:44] Client 1: Message 12                                             main.py:62
           Client 1: Message 13                                             main.py:62
           Client 1: Message 14                                             main.py:62
[11:41:46] Client 1: Message 16                                             main.py:62
```

Hopefully like our example you can see that some of the messages arrive out of order. Next add more publishers and subscribers and experiment with the different configurations.

> **Info**
>
> When making multiple publishes you may want to change the client-name cli parameter so you can keep track of when the messages arrived at the subscribers.

3.3 SQS FIFO

Now we will experiment with the FIFO based service offered by SQS. Like before, we will start a subscriber but make sure the name of the queue matches the FIFO queue we created in terraform.

```
> docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "
  csse6400_prac.fifo" --client-name "Client 1" --receive
```

Now start a publisher in another terminal.

```
> docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "
  csse6400_prac.fifo" --client-name "Client 1"
```

```
^[[D$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "csse6400_prac.f
ifo" --client-name "Client 2"


      ----------
     |   \XX/   |
     | T. \/ .T |        University of Queensland
     | XX:  :XX |            Faculty of EAIT
     T L' /\ 'J T
      \  /XX\  /           CSSE6400 Queue Prac
    @\_ '____' _/@         csse6400.uqcloud.net
    \_X\_ __ _/X_/
     \=/\----/\=/


Connected to csse6400_prac.fifo
Sending Messages:
Working... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
$
```

On the subscriber we now see the messages arriving in order which is to be expected.

```
$ docker run --rm -it --env-file .env ghcr.io/csse6400/queue:main --name "csse6400_prac.fifo"
 --client-name "hello" --receive


      ----------
     |   \XX/   |
     | T. \/ .T |        University of Queensland
     | XX:  :XX |            Faculty of EAIT
     T L' /\ 'J T
      \  /XX\  /           CSSE6400 Queue Prac
    @\_ '____' _/@         csse6400.uqcloud.net
    \_X\_ __ _/X_/
     \=/\----/\=/


Connected to csse6400_prac.fifo
[11:57:28] Client 1: Message 0                                                    main.py:62
[11:57:29] Client 1: Message 1                                                    main.py:62
           Client 1: Message 2                                                    main.py:62
[11:57:30] Client 1: Message 3                                                    main.py:62
           Client 1: Message 4                                                    main.py:62
```

If we re-run the publisher though we may not see any new messages make it to the consumer. This is because we have asked AWS to de-dupe messages where it can.

As before try experimenting with different publisher / subscriber configurations to see how they behave.

Info

When making multiple publishes you may want to change the client-name cli parameter so you can keep track of when the messages arrived at the subscribers.

Warning

Please remember to terraform destroy to delete your resources

4 Hosting TaskOverflow Images

When we last deployed a container on AWS, we used an existing hosted image. Now, we will be developing our own image, so we will need a mechanism to host the image. For this, we will being using an AWS ECR, Docker, and Terraform. AWS ECR is the Elastic Container Registry, it is a container registry like DockerHub or GitHub. We can use it to host our images, using the process below:

1. Use Terraform to create an ECR repository for our image.
2. Use Terraform to build our Docker image.
3. Use Terraform to push our Docker image.

Info

This is a non-standard process. As you may have seen in the DevOps tutorial, we would ordinarily like our code commits to trigger a CI/CD pipeline which builds the images.
If you would like, you can use GitHub actions to build and push your container to the GitHub container registry and authenticate when you pull the image. However, using ECR simplifies the process, despite the oddities introduced by having a non-persistent ECR repository.

We will setup our initial Terraform configuration. Note that now we introduce a new required provider. This provider is for Docker.

```
terraform {
   required_providers {
      aws = {
         source = "hashicorp/aws"
         version = "~> 4.0"
      }
      docker = {
         source = "kreuzwerker/docker"
         version = "3.0.2"
      }
   }
}

provider "aws" {
   region = "us-east-1"
   shared_credentials_files = ["./credentials"]
}
```

As with our AWS provider, when we initially configure the provider, we want to authenticate so that we can later push to our registry using the Docker provider. We will use the `aws_ecr_authorization_token` data block to get appropriate ECR credentials for Docker.

```
» cat main.tf
data "aws_ecr_authorization_token" "ecr_token" {}

provider "docker" {
  registry_auth {
    address = data.aws_ecr_authorization_token.ecr_token.proxy_endpoint
    username = data.aws_ecr_authorization_token.ecr_token.user_name
    password = data.aws_ecr_authorization_token.ecr_token.password
  }
}
```

We need to use Terraform to create an ECR repository to push to.

```
» cat main.tf
resource "aws_ecr_repository" "taskoverflow" {
  name = "taskoverflow"
}
```

The URL for containers in the ECR following the format below:
`{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME}`
Remember — to push to a container registry we need a local container whose tag matches the remote URL. We could then create and push the contianer locally with:

```
docker build -t {ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME} .
docker push {ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME}
```

However, it would be easier if we could build and push this container from within Terraform. We can use the Docker provider for this.

```
» cat image.tf
resource "docker_image" "taskoverflow" {
  name = "${aws_ecr_repository.taskoverflow.repository_url}:latest"
  build {
    context = "."
  }
}

resource "docker_registry_image" "taskoverflow" {
  name = docker_image.taskoverflow.name
}
```

Notice that we are able to utilize the output of the ECR repository as the URL which resolves to the correct URL for the image.

5 Worker Queues

One good use case for queues is for distributing work to scale to demand. In this exercise we encourage you to have a look at these widly used libraries to see how you could integrate them into a distributed system.

- Python: Celery
- Java: RabbitMQ

The two above libraries are integrated into many popular application frameworks. Today we will be using the Python library Celery to create a simple worker queue for TaskOverflow.

5.1 Celery

Celery is a Python library which allows you to create a worker queue. It is popular and used in many large scale applications. It is also relatively easy to use and has comprehensive documentation.

6 Task Calendar

TaskOverflow will use celery to distribute the work of generating a calendar view of tasks, as this could be a time-intensive task.[1] This will be done by a worker which will run in a separate container. The TaskOverflow server will place job requests on the Celery queue. The worker will pick-up jobs and generate the calendar. The webserver can then serve the calendar to the user.

Calendar generation occurs asynchronously with this architecture. The user will not have to wait for the calendar view to be generated and the webserver can handle other requests.

6.1 API Extension

We will extend our existing API with the following endpoints.

POST /todos/ical Create an asynchronous calendar generation job.[2]

GET /todos/ical/{task_id}/status Check the status of the generation job.

GET /todos/ical/{task_id}/result Download the generated iCal file when it is ready.

An example interaction with the API might look like:

```
> curl -X POST http://localhost:6400/api/v1/todos/ical
{
  "task_id": "2e80aefd-4a69-4a94-b23a-45f2b8110988",
  "task_url": "http://localhost:6400/api/v1/todos/ical/2e80aefd-4a69-4a94-b23a-45
     f2b8110988/status"
}
> curl -X GET http://localhost:6400/api/v1/todos/ical/2e80aefd-4a69-4a94-b23a-45
   f2b8110988/status
```

[1] In reality it is not but this is an example of a common use case for asynchronous tasks — file generation such as PDFs or videos which do take time.

[2] Ideally a POST procedural endpoint should use a verb like `/generate`.

```
{
  "result_url": "http://localhost:6400/api/v1/todos/ical/2e80aefd-4a69-4a94-b23a-45
     f2b8110988/result",
  "task_id": "2e80aefd-4a69-4a94-b23a-45f2b8110988",
  "task_status": "PENDING"
}
> curl -X GET http://localhost:6400/api/v1/todos/ical/2e80aefd-4a69-4a94-b23a-45
   f2b8110988/status
{
  "result_url": "http://localhost:6400/api/v1/todos/ical/2e80aefd-4a69-4a94-b23a-45
     f2b8110988/result",
  "task_id": "2e80aefd-4a69-4a94-b23a-45f2b8110988",
  "task_status": "SUCCESS"
}
> curl -X GET http://localhost:6400/api/v1/todos/ical/2e80aefd-4a69-4a94-b23a-45
   f2b8110988/result
BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//Taskoverflow Calendar//mxm.dk//
BEGIN:VEVENT
SUMMARY:Project Proposal
DTSTART:20230403T140000
UID:1
DESCRIPTION:CSSE6400 Project Proposal Due
END:VEVENT
END:VCALENDAR
```

We will stub out the implementation of these endpoints.

```
» cat todo/views/routes.py
@api.route('/todos/ical', methods=['POST'])
def create_ical():
    pass

@api.route('/todos/ical/<task_id>/status', methods=['GET'])
def get_task(task_id):
    pass

@api.route('/todos/ical/<task_id>/result', methods=['GET'])
def get_calendar(task_id):
    pass
```

6.2 Queue and Workers

Next we will modify our `docker-compose.yml` to include a local queuing service called Redis. We will later use AWS SQS but as Celery supports multiple queues, we can use this to make the local development easier.

```
» cat docker-compose.yml
redis:
  image: redis:latest
  restart: always
  ports:
    - "6379:6379"
```

Now we have a queue, we will create the worker that accepts jobs from the queue. We will create this worker in `todo/tasks/ical.py`.

```
» cat todo/tasks/ical.py
import os

from celery import Celery

celery = Celery(__name__)
celery.conf.broker_url = os.environ.get("CELERY_BROKER_URL")
celery.conf.result_backend = os.environ.get("CELERY_RESULT_BACKEND")

@celery.task(name="ical")
def create_ical(tasks):
    return "Hello World"
```

Note that we are getting a URL from the environment variables: `CELERY_BROKER_URL` and `CELERY_RESULT_BAC` We need to remember to set these variables in whichever environment the service is run in.

We need to install Celery. We will do this in the same pipenv environment as the Flask server as we will later require our Flask server to use Celery to put jobs onto the task queue.

```
$ pipenv install celery redis celery[sqs]
```

The above will install the Celery core and Redis and SQS drivers for communication.

We will now start our local Redis queue and our worker to ensure that they can communicate.

```
$ docker-compose up redis
```

```
> export CELERY_BROKER_URL=redis://localhost:6379
> export CELERY_RESULT_BACKEND=redis://localhost:6379
> pipenv run python3 -m celery --app todo.tasks.ical worker --loglevel=info
```

You should see something like the below. This indicates that the worker has successfully subscribed to the queue and is ready to accept events.

```
 -------------- celery@uqbwebb2-6842 v5.2.7 (dawn-chorus)
--- ***** -----
-- ******* ---- macOS-12.6.1-x86_64-i386-64bit 2023-04-04 18:51:09
- *** --- * ---
- ** ---------- [config]
- ** ---------- .> app: todo.tasks.ical:0x10432e7d0
- ** ---------- .> transport: redis://localhost:6379//
- ** ---------- .> results: redis://localhost:6379/
- *** --- * --- .> concurrency: 8 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
 -------------- [queues]
               .> celery exchange=celery(direct) key=celery


[tasks]
  . ical

[2023-04-04 18:51:10,059: INFO/MainProcess] Connected to redis://localhost:6379//
[2023-04-04 18:51:10,065: INFO/MainProcess] mingle: searching for neighbors
[2023-04-04 18:51:11,200: INFO/MainProcess] mingle: sync with 1 nodes
[2023-04-04 18:51:11,201: INFO/MainProcess] mingle: sync complete
[2023-04-04 18:51:11,240: INFO/MainProcess] celery@uqbwebb2-6842 ready.
```

Finally, we will run Celery inside a new container. For this we will modify our `docker-compose.yml` file.

```
  » cat docker-compose.yml

worker:
  build:
    context: .
    dockerfile: Dockerfile.dev
  restart: always
  environment:
    CELERY_BROKER_URL: redis://redis:6379
    CELERY_RESULT_BACKEND: redis://redis:6379
  command: celery --app todo.tasks.ical worker --loglevel=info
```

Note that we are re-using the same image development image as the Flask server. This makes sense since it shares very similar dependencies. It would make sense to separate the dependencies between the two containers as they grow larger.

We are overriding the command to run when the container starts to use the celery command. We have also remembered to set the environment variables appropriately. We have to use the domain name of the redis image instead of the localhost because redis is not running on our localhost. Earlier we were able to pretend it was because the redis port, 6379, was being forwarded from our localhost to our redis container.

6.3 Creating Tasks

We will now return to our Flask server. We can modify our new endpoints to use Celery to create tasks on the task queue.

```
» cat todo/views/routes.py

from celery.result import AsyncResult
from todo.tasks import ical

...

@api.route('/todos/ical', methods=['POST'])
def create_ical():
    # put all tasks in the database into a tasks variable
    task = ical.create_ical.delay(tasks)
    return jsonify({"task_id": task.id}), 202

@api.route('/todos/ical/<task_id>/status', methods=['GET'])
def get_task(task_id):
    task = AsyncResult(task_id)
    # use task to populate an appropriate JSON response
    return jsonify(result), 200

@api.route('/todos/ical/<task_id>/result', methods=['GET'])
def get_calendar(task_id):
    task_result = AsyncResult(task_id)
    if task_result.status == 'SUCCESS':
        return task_result.result, 200, {'Content-Type': 'text/calendar'}
    else:
        return jsonify({'error': 'Task not finished'}), 404
```

The variable that we create as tasks and pass to the delay method is the JSON data that is stored with the event. It is then deserialized from JSON and passed as a variable to the `create_ical` function. Celery is able to create impressive abstractions so that message passing over a task queue looks almost like regular function calls.

6.4 Creating the Calendar

Once we have confirmed that the task queue can communicate with our worker, we can return to the calendar generation. We will use the `icalendar` library for this.

```
$ pipenv install icalendar
```

```
$ cat todo/tasks/ical.py
import icalendar
import time

...

def create_ical(tasks):
    cal = icalendar.Calendar()
    cal.add("prodid", "-//Taskoverflow Calendar//mxm.dk//")
    cal.add("version", "2.0")

    time.sleep(50)

    for task in tasks:
        event = icalendar.Event()

        event.add("uid", task["id"])
        event.add("summary", task["title"])
        event.add("description", task["description"])
        event.add("dtstart", datetime.datetime.strptime(task["deadline_at"], "%Y-%m-%
            dT%H:%M:%S"))

        cal.add_component(event)

    return cal.to_ical().decode("utf-8")
```

Now you should be able to test your application.

1. Start it with docker-compose.

```
$ docker-compose up --build
```

2. Open the user interface on `http://localhost:6400` and create some tasks.

3. Follow the cURL interaction from Section 6.1 to generate an appropriate iCal.

6.4.1 Design Challenge

Say we observe a user pattern that the majority of users will mark off multiple todo items at once. We have also moved our calendar from a PDF being displayed to the user to an iCal so that its visible in their chosen calendar application.

How could we improve the efficiency of our system to reduce the amount of time it takes to generate the calendar view?

References

[1] R. Thomas, "Event-driven architecture," April 2022. https://csse6400.uqcloud.net/handouts/event.pdf.