
Project Documentation: Asynchronous Tasks with Celery & Redis

Purpose

This system was added to handle long-running tasks **in the background** without making the user wait. The primary example is the "Email Summary" feature. Sending an email can be slow, so instead of doing it during the API request, we hand the job off to a background worker. This keeps the API fast and responsive.

The Components 🧩

Our asynchronous system has three main parts that work together.

1. Redis (The Message Broker)

- **What it is:** Redis acts as our "message board" or "post office." It's a very fast, in-memory database.
- **Its Role:** When our Django app needs a background job done, it doesn't do the job itself. It writes a "message" describing the job and posts it to Redis. In our project, we are running Redis inside a **Docker container**.

2. Celery (The Background Worker)

- **What it is:** Celery is the "worker." It's a separate program that constantly watches Redis for new job messages.
- **Its Role:** As soon as a new message appears, the Celery worker picks it up and executes the corresponding task (e.g., sending the summary email). It runs in its own terminal window.

3. Django (The Task Producer)

- **What it is:** Our main Fintech API application.
- **Its Role:** Django is the "producer" of tasks. When a user calls an endpoint like `/api/reports/email-summary/`, the view function's only job is to create the task message and hand it to Redis. It does this by calling the special `.delay()` method on our task function.

The Workflow

Here's the step-by-step process when you request an email report:

1. Your `api.http` client sends a `POST` request to `/api/reports/email-summary/`.
2. The `email_report_view` in `api/views.py` receives the request.
3. The view calls `send_monthly_summary_email.delay(user.id)`.
4. This command packages the function call and its arguments into a message and sends it to the **Redis** message queue.
5. The Django app immediately returns a `202 Accepted` response to you.
6. The **Celery worker**, which is always watching Redis, sees the new message and picks it up.
7. The worker executes the `send_monthly_summary_email` function from `api/tasks.py`, generating the summary and "sending" the email (printing it to the console).

Key Files in the Project

- **`fintech/celery.py`**: Defines and creates the main Celery application instance.
- **`fintech/__init__.py`**: Ensures the Celery app is loaded when Django starts.
- **`fintech/settings.py`**: Contains the `CELERY_` configuration, telling Celery where to find Redis.
- **`api/tasks.py`**: This is where you write your background task functions, decorated with `@shared_task`.
- **`api/views.py`**: The view that triggers a background task by calling `.delay()` on it.

How to Run in Development

To run the full application, you need **three** things running simultaneously in separate terminals:

1. **Start Redis (via Docker):**
2. Bash

```
docker start my-redis-container
```

3.
4.
5. **Start the Django Server:**
6. PowerShell

```
# In Terminal 1
.\venv\Scripts\activate
python manage.py runserver
```

7.
8.
9. **Start the Celery Worker:**

10. PowerShell

```
# In Terminal 2
.\venv\Scripts\activate
celery -A fintech worker -P solo --loglevel=info
```

11.
12.