FastAPI with Background Tasks Project Structure

Async task queue with ARQ and Redis. Worker processes, scheduled jobs, and task monitoring.

#fastapi #python #api #arq #redis #background-tasks #async #workers

PNG PDF Copy Prompt

Project Directory

```
myproject/
    main.py App factory, li…
  app/ Application code
    __init__.py
    config.py Pydantic settin…
    core/
      __init__.py
      database.py Async engine
      redis.py Redis connectio…
    models/
      __init__.py
      base.py
      user.py
      task_log.py Track task runs
    schemas/
      __init__.py
      user.py
      task.py Task request/re…
    api/
      __init__.py
      v1/
        __init__.py
        router.py
        users.py
        tasks.py Enqueue and sta…
    services/
      __init__.py
      user_service.py
      task_service.py Enqueue helpers
  workers/ Background work…
    __init__.py
    settings.py ARQ worker sett…
    tasks/ Task definitions
      __init__.py
      email.py Email sending t…
      reports.py Report generati…
      cleanup.py Scheduled clean…
    cron.py Scheduled job d…
  tests/
    __init__.py
    conftest.py Fixtures, mock …
    api/
      __init__.py
      test_tasks.py
    workers/
      __init__.py
      test_email_tasks.py Test task logic…
  requirements.txt
  requirements-dev.txt
  .env.example
  .gitignore
  pyproject.toml
```

Why This Structure?

This structure separates API endpoints from background workers. ARQ uses the same async/await patterns as FastAPI, so your task code feels native. Workers run as separate processes, scaling independently from your API.

Key Directories

`workers/` - Standalone worker processes with ARQ

`workers/tasks/` - Task functions grouped by domain

`workers/cron.py` - Scheduled jobs (daily reports, cleanup)

`app/services/task_service.py` - API-side enqueue helpers

`app/models/task_log.py` - Optional: track task execution history

Getting Started

1. `python -m venv venv && source venv/bin/activate`
2. `pip install -r requirements.txt`
3. `cp .env.example .env` (set REDIS_URL)
4. Start Redis: `docker run -d -p 6379:6379 redis`
5. Start API: `uvicorn main:app --reload`
6. Start worker: `arq workers.settings.WorkerSettings`

ARQ Task Pattern

ARQ tasks are just async functions. Define in `workers/tasks/`, register in `workers/settings.py`. The API enqueues via `await redis.enqueue_job('task_name', arg1, arg2)`. ARQ handles retries, timeouts, and result storage.

Task Definition

```python
# workers/tasks/email.py
async def send_welcome_email(ctx, user_id: int):
    user = await get_user(user_id)
    await email_client.send(
        to=user.email,
        template="welcome"
    )

# workers/settings.py
class WorkerSettings:
    functions = [send_welcome_email]
    redis_settings = RedisSettings.from_dsn(REDIS_URL)
```

When To Use This

- Email sending, notifications, webhooks
- Report generation and data exports
- Image/video processing pipelines
- Scheduled jobs (daily summaries, cleanup)
- Any operation > 500ms blocking the request

Trade-offs

`Redis dependency` - Requires Redis for task queue (not optional)

`Operational complexity` - Workers need monitoring and restart policies

`Debugging overhead` - Async task failures harder to trace than sync

Testing Strategy

`tests/workers/` - Test task functions directly with mocked deps

`conftest.py` - Use fakeredis for queue tests

`tests/api/` - Mock enqueue, verify job was queued