**🗂️ Directory Structure Overview**

Organizing your asynchronous tasks into a dedicated jobs/ directory is a best practice. Here's a suggested structure 

src/

└── jobs/

├── background/

│ ├── email/

│ │ ├── email.processor.ts

│ │ ├── email.service.ts

│ │ └── email.module.ts

│ └── report/

│ ├── report.processor.ts

│ ├── report.service.ts

│ └── report.module.ts

├── cron/

│ ├── cron.service.ts

│ └── cron.module.ts

└── jobs.module.ts

```

- \*\*`background/`\*\*:Contains modules for background tasks like sending emails or generating reports

- \*\*`cron/`\*\*:Houses scheduled tasks using decorators like `@Cron()` from the `@nestjs/schedule` package

- \*\*`jobs.module.ts`\*\*:Aggregates all job-related modules for easy import into the root module

---

## 📦 Installation

Install the necessary packages:



```bash

npm install --save @nestjs/schedule @nestjs/bull bull ioredis

npm install --save-dev @types/bull

``



Ensure that Redis is installed and running, as both Bull and distributed locking mechanisms rely on i.

---

## 🧩 Background Job Example: Email Queue

\*\*`email.service.ts`\*



```typescript

import { Injectable } from '@nestjs/common';

import { InjectQueue } from '@nestjs/bull';

import { Queue } from 'bull';

@Injectable()

export class EmailService {

constructor(@InjectQueue('emailQueue') private emailQueue: Queue) {}

async sendWelcomeEmail(userEmail: string) {

await this.emailQueue.add('sendEmail', { email: userEmail });

}

}

``



\*\*`email.processor.ts`\*



```typescript

import { Processor, Process } from '@nestjs/bull';

import { Job } from 'bull';

@Processor('emailQueue')

export class EmailProcessor {

@Process('sendEmail')

async handleSendEmail(job: Job) {

const { email } = job.data;

// Logic to send email

console.log(`Sending email to ${email}`);

}

}

``



\*\*`email.module.ts`\*



```typescript

import { Module } from '@nestjs/common';

import { BullModule } from '@nestjs/bull';

import { EmailService } from './email.service';

import { EmailProcessor } from './email.processor';

@Module({

imports: [

BullModule.registerQueue({

name: 'emailQueue',

}),

],

providers: [EmailService, EmailProcessor],

exports: [EmailService],

})

export class EmailModule {}

``



---

## ⏰ Cron Job Example: Scheduled Tasks

\*\*`cron.service.ts`\*



```typescript

import { Injectable, Logger } from '@nestjs/common';

import { Cron, CronExpression } from '@nestjs/schedule';

@Injectable()

export class CronService {

private readonly logger = new Logger(CronService.name);

// Runs every hour

@Cron(CronExpression.EVERY\_HOUR)

handleCron() {

this.logger.log('Running hourly task...');

// Task logic here

}

}

``



\*\*`cron.module.ts`\*



```typescript

import { Module } from '@nestjs/common';

import { ScheduleModule } from '@nestjs/schedule';

import { CronService } from './cron.service';

@Module({

imports: [ScheduleModule.forRoot()],

providers: [CronService],

})

export class CronModule {}

``



---

## 🔐 Handling Multi-Instance Environments

In a horizontally scaled application, cron jobs might execute multiple times across instances. To prevent this:

1. \*\*Implement Distributed Locks\*: Use Redis to acquire a lock before executing a job. If the lock is acquired, proceed; otherwise, skip executon.

2. \*\*Unique Job IDs\*: When adding jobs to Bull queues, assign unique IDs to prevent duplicate processng.

For a detailed guide on managing distributed cron jobs in NestJS, refer to this artile:

linkturn0search0

---

## 🧠 Concluson

By structuring your `jobs/` directory to include both background and cron jobs, and implementing distributed locking mechanisms, you ensure that your NestJS application handles asynchronous tasks efficiently and relibly.