**Task Schedule**

1. **Khái niệm**

Task Schedule cho phép chúng ta lập lịch mã (phương pháp / chức năng) tùy ý để thực thi vào một ngày / giờ cố định, theo khoảng thời gian định kỳ hoặc một lần sau một khoảng thời gian xác định.

Trong thế giới Linux, điều này thường được xử lý bởi các gói như cron ở cấp hệ điều hành. Đối với ứng dụng Node.js, có một số gói mô phỏng chức năng giống như cron. Nest cung cấp gói @nestjs/schedule, tích hợp với gói Node.js node-cron phổ biến.

1. **Installation**

npm install --save @nestjs/schedule

1. **Usage**

Để kích hoạt tính năng lập lịch công việc, nhập ScheduleModule vào AppModule gốc và chạy phương thức tĩnh forRoot() như hiển thị bên dưới:

|  |
| --- |
| import { Module } from '@nestjs/common';  import { ScheduleModule } from '@nestjs/schedule';  @Module({  imports: [  ScheduleModule.forRoot()  ],  })  export class AppModule {} |

Lệnh gọi .forRoot() khởi tạo trình lập lịch và registers bất kỳ cron job khai báo, timeouts và intervals nào tồn tại trong ứng dụng. Registration xảy ra khi onApplicationBootstrap lifecycle hook xảy ra, đảm bảo rằng tất cả các modules đã loaded và declared bất kỳ scheduled jobs nào.

**Cron Jobs**

Một cron job lên lịch một hàm tùy ý (gọi phương thức) để chạy tự động. Cron job có thể chạy:

+ Một lần, vào một ngày/giờ cụ thể.

+ Định kỳ, các công việc định kỳ có thể chạy tại một thời điểm cụ thể trong một khoảng thời gian cụ thể (ví dụ: một lần mỗi giờ, một lần mỗi tuần, một lần sau mỗi 5 phút)

Khai báo một cron job với decorator @Cron() đứng trước định nghĩa phương thức có chứa code sẽ được thực thi, như sau:

|  |
| --- |
| import { Injectable, Logger } from '@nestjs/common';  import { Cron } from '@nestjs/schedule';  @Injectable()  export class TasksService {  private readonly logger = new Logger(TasksService.name);  @Cron('45 \* \* \* \* \*')  handleCron() {  this.logger.debug('Called when the current second is 45');  }  } |

Trong ví dụ này, phương thức handleCron() sẽ được gọi mỗi lần giây hiện tại là 45. Nói cách khác, phương thức này sẽ được chạy một lần mỗi phút, tại mốc giây thứ 45. Decorator @Cron() hỗ trợ các cron pattern chuẩn sau:

* Asterisk (e.g. \*)
* Ranges (e.g. 1-3,5)
* Steps (e.g. \*/2)

Trong ví dụ trên, truyền 45 \* \* \* \* \* cho decorator. Key sau đây cho thấy cách diễn giải từng vị trí trong chuỗi mẫu cron:

|  |
| --- |
| \* \* \* \* \* \*  | | | | | |  | | | | | day of week  | | | | months  | | | day of month  | | hours  | minutes  seconds (optional) |

Một số cron pattern là:

|  |  |
| --- | --- |
| \* \* \* \* \* \* | mỗi giây |
| 45 \* \* \* \* \* | mỗi phút, vào giây thứ 45 |
| 0 10 \* \* \* \* | mỗi giờ, vào lúc bắt đầu phút thứ 10 |
| 0 \*/30 9-17 \* \* \* | cứ 30 phút một lần từ 9 giờ đến 17 giờ |
| 0 30 11 \* \* 1-5 | Thứ Hai đến Thứ Sáu lúc 11:30 sáng |

Gói @nestjs/schedule cung cấp một enum tiện lợi với các mẫu cron thường dùng. Bạn có thể sử dụng enum này như sau:

|  |
| --- |
| import { Injectable, Logger } from '@nestjs/common';  import { Cron, CronExpression } from '@nestjs/schedule';  @Injectable()  export class TasksService {  private readonly logger = new Logger(TasksService.name);  @Cron(CronExpression.EVERY\_30\_SECONDS)  handleCron() {  this.logger.debug('Called every 30 seconds');  }  } |

Trong ví dụ này, phương thức handleCron() sẽ được gọi sau mỗi 30 giây.

Ngoài ra, bạn có thể cung cấp đối tượng Date JavaScript cho decorator @Cron(). Làm như vậy sẽ khiến công việc thực thi chính xác một lần, vào ngày đã chỉ định.

VD: Sử dụng phép tính ngày tháng của JavaScript để lên lịch các tác vụ liên quan đến ngày hiện tại. Ví dụ: @Cron(new Date(Date.now() + 10 \* 1000)) để lên lịch một tác vụ chạy 10 giây sau khi ứng dụng khởi động.

Ngoài ra, có thể cung cấp các tùy chọn bổ sung làm tham số thứ hai cho decorator @Cron().

|  |  |
| --- | --- |
| name | Hữu ích để truy cập và kiểm soát tác vụ cron sau khi tác vụ đó được khai báo. |
| timeZone | Chỉ định múi giờ để thực hiện. Điều này sẽ sửa đổi thời gian thực tế so với múi giờ của bạn. Nếu múi giờ không hợp lệ, lỗi sẽ được đưa ra. Bạn có thể kiểm tra tất cả các múi giờ có sẵn tại trang web Moment Timezone |
| utcOffset | Điều này cho phép bạn chỉ định độ lệch múi giờ thay vì sử dụng tham số timeZone. |
| disabled | Điều này cho biết liệu công việc có được thực hiện hay không. |

|  |
| --- |
| import { Injectable } from '@nestjs/common';  import { Cron, CronExpression } from '@nestjs/schedule';  @Injectable()  export class NotificationService {  @Cron('\* \* 0 \* \* \*', {  name: 'notifications',  timeZone: 'Europe/Paris',  })  triggerNotifications() {}  } |

Bạn có thể truy cập và kiểm soát tác vụ cron sau khi tác vụ đó được khai báo hoặc tạo dynamic cron job (trong đó cron pattern được xác định khi chạy) bằng API động. Để truy cập công việc cron khai báo thông qua API, bạn phải liên kết công việc với tên bằng cách truyền thuộc tính ‘name’ trong đối tượng tùy chọn làm đối số thứ hai của decorator.

**Declarative intervals**

Để khai báo một phương thức sẽ chạy ở một khoảng thời gian (lặp lại) được chỉ định, hãy thêm tiền tố cho định nghĩa phương thức bằng trình trang trí @Interval(). Truyền giá trị khoảng thời gian, dưới dạng số tính bằng mili giây, cho decorator như được hiển thị bên dưới:

|  |
| --- |
| @Interval(10000)  handleInterval() {  this.logger.debug('Called every 10 seconds');  } |

Cơ chế này sử dụng hàm JavaScript setInterval() bên dưới. Bạn cũng có thể sử dụng cron job để lên lịch các công việc định kỳ.

**Queue**

Queue là một mẫu thiết kế mạnh mẽ giúp chúng ta đối phó với các thách thức về quy mô và hiệu suất ứng dụng phổ biến.

Trong NestJS, Queues là một mô hình thiết kế mạnh mẽ giúp bạn giải quyết các thách thức về hiệu suất và mở rộng ứng dụng, sử dụng @nestjs/bull. Dưới đây là một số ứng dụng chính của Queues trong NestJS:

1. **Giảm tải xử lý đỉnh điểm**: Khi người dùng có thể khởi tạo các tác vụ tốn tài nguyên vào bất kỳ thời điểm nào, bạn có thể thêm các tác vụ này vào hàng đợi thay vì thực hiện chúng đồng bộ. Điều này giúp xử lý các tác vụ một cách có kiểm soát và tránh quá tải hệ thống.
2. **Phân tán và xử lý không đồng bộ**: Queues cho phép bạn phân tán các tác vụ và xử lý chúng không đồng bộ. Ví dụ, một tác vụ yêu cầu xử lý CPU nặng như chuyển đổi âm thanh có thể được ủy thác cho các quy trình khác, giúp các quy trình giao diện người dùng vẫn duy trì được tính phản hồi.
3. **Kênh giao tiếp đáng tin cậy giữa các dịch vụ**: Bạn có thể xếp hàng các tác vụ (jobs) trong một quy trình hoặc dịch vụ và tiêu thụ chúng trong một quy trình hoặc dịch vụ khác. Điều này cung cấp một kênh giao tiếp đáng tin cậy giữa các dịch vụ khác nhau.
4. **Tự động khởi động lại khi gặp lỗi**: Khi các nhà sản xuất hoặc người tiêu thụ hàng đợi gặp lỗi, trạng thái của chúng được bảo toàn và việc xử lý tác vụ có thể tự động khởi động lại khi các nút được khởi động lại.

Bull là một thư viện Node triển khai một hệ thống queue nhanh chóng và mạnh mẽ dựa trên Redis. Mặc dù có thể triển khai queue trực tiếp chúng tag lệnh Redis, Bull là một trừu tượng / trình bao bọc (abstraction/wrapper) trên Redis. Nó cung cấp một API xử lý tất cả các chi tiết cấp thấp và làm phong phú thêm chức năng cơ chúng ta của Redis để có thể dễ dàng xử lý các trường hợp sử dụng phức tạp hơn. Nói một cách dễ hiểu hơn thì...

Bull là một trong số các thư viện hỗ trợ message queue rất tốt trên NodeJS (bên cạnh RabbitMQ, Kafka, rsmq, agenda,...), và dựa trên Redis để lưu dữ liệu.

Message queue là giải pháp để giải quyết các vấn đề trong công nghệ, thường là backend:

● Xử lý background.

● Gửi thông tin theo hàng đợi.

● Xử lý đồng bộ, concurrency.

Bull là thư viện có thể giúp cả 3 vấn đề trên, và chúng ta thân cũng có 3 thành phần chính:

● Producer: Thành phần tạo ra các công việc (job) và ném chúng vào queue (hàng đợi).

● Consumer: Thành phần nhận việc từ queue.

● Listener: Thành phần lắng nghe các sự kiện diễn ra trong hàng đợi (completed, failed, stalled).

Về cơ chúng ta, message queue là cơ chế đưa các thông tin vào dãy để xử lý dần, và mặc định với Bull thì hàng đợi là dạng FIFO (dữ liệu vào trước thì ra trước).