Task Scheduling

Task scheduling allows you to schedule arbitrary code (methods/functions) to execute at a fixed date/time, at recurring intervals, or once after a specified interval. In the Linux world, this is often handled by packages like cron at the OS level. For Node.js apps, there are several packages that emulate cron-like functionality. Nest provides the `@nestjs/schedule` package, which integrates with the popular Node.js cron package. We'll cover this package in the current chapter.

Installation

To begin using it, we first install the required dependencies.

```
$ npm install --save @nestjs/schedule
```

To activate job scheduling, import the `ScheduleModule` into the root `AppModule` and run the `forRoot()` static method as shown below:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { ScheduleModule } from '@nestjs/schedule';

@Module({
  imports: [
    ScheduleModule.forRoot()
  ],
})
export class AppModule {}
```

The `.forRoot()` call initializes the scheduler and registers any declarative cron jobs, timeouts and intervals that exist within your app. Registration occurs when the `onApplicationBootstrap` lifecycle hook occurs, ensuring that all modules have loaded and declared any scheduled jobs.

Declarative cron jobs

A cron job schedules an arbitrary function (method call) to run automatically. Cron jobs can run:

- Once, at a specified date/time.
- On a recurring basis; recurring jobs can run at a specified instant within a specified interval (for example, once per hour, once per week, once every 5 minutes)

Declare a cron job with the `@Cron()` decorator preceding the method definition containing the code to be executed, as follows:

```
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');
  }
}
```

In this example, the `handleCron()` method will be called each time the current second is `45`. In other words, the method will be run once per minute, at the 45 second mark.

The `@Cron()` decorator supports all standard cron patterns:

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

In the example above, we passed `45 * * * * *` to the decorator. The following key shows how each position in the cron pattern string is interpreted:

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

Some sample cron patterns are:

`* * * * * *`	every second
`45 * * * * *`	every minute, on the 45th second
`0 10 * * * *`	every hour, at the start of the 10th minute
`0 */30 9-17 * * *`	every 30 minutes between 9am and 5pm
`0 30 11 * * 1-5`	Monday to Friday at 11:30am

The `@nestjs/schedule` package provides a convenient enum with commonly used cron patterns. You can use this enum as follows:

```
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');
  }
}
```

In this example, the `handleCron()` method will be called every `30` seconds.

Alternatively, you can supply a JavaScript `Date` object to the `@Cron()` decorator. Doing so causes the job to execute exactly once, at the specified date.

> info **Hint** Use JavaScript date arithmetic to schedule jobs relative to the current date. For example, `@Cron(new Date(Date.now() + 10 * 1000))` to schedule a job to run 10 seconds after the app starts.

Also, you can supply additional options as the second parameter to the `@Cron()` decorator.

`name`	Useful to access and control a cron job after it's been declared.
`timeZone`	Specify the timezone for the execution. This will modify the actual time relative to your timezone. If the timezone is invalid, an error is thrown. You can check all timezones available at Moment Timezone website.
`utcOffset`	This allows you to specify the offset of your timezone rather than using the `timeZone` param.
`disabled`	This indicates whether the job will be executed at all.

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

@Injectable()
export class NotificationService {
  @Cron('* * 0 * * *', {
    name: 'notifications',
    timeZone: 'Europe/Paris',
  })
  triggerNotifications() {}
}
```

You can access and control a cron job after it's been declared, or dynamically create a cron job (where its cron pattern is defined at runtime) with the Dynamic API. To access a declarative cron job via the API, you

must associate the job with a name by passing the `name` property in an optional options object as the second argument of the decorator.

Declarative intervals

To declare that a method should run at a (recurring) specified interval, prefix the method definition with the `@Interval()` decorator. Pass the interval value, as a number in milliseconds, to the decorator as shown below:

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

> info **Hint** This mechanism uses the JavaScript `setInterval()` function under the hood. You can also utilize a cron job to schedule recurring jobs.

If you want to control your declarative interval from outside the declaring class via the Dynamic API, associate the interval with a name using the following construction:

```
@Interval('notifications', 2500)
handleInterval() {}
```

The Dynamic API also enables **creating** dynamic intervals, where the interval's properties are defined at runtime, and **listing and deleting** them.

Declarative timeouts

To declare that a method should run (once) at a specified timeout, prefix the method definition with the `@Timeout()` decorator. Pass the relative time offset (in milliseconds), from application startup, to the decorator as shown below:

```
@Timeout(5000)
handleTimeout() {
  this.logger.debug('Called once after 5 seconds');
}
```

> info **Hint** This mechanism uses the JavaScript `setTimeout()` function under the hood.

If you want to control your declarative timeout from outside the declaring class via the Dynamic API, associate the timeout with a name using the following construction:

```
@Timeout('notifications', 2500)
handleTimeout() {}
```

The Dynamic API also enables **creating** dynamic timeouts, where the timeout's properties are defined at runtime, and **listing and deleting** them.

Dynamic schedule module API

The `@nestjs/schedule` module provides a dynamic API that enables managing declarative cron jobs, timeouts and intervals. The API also enables creating and managing **dynamic** cron jobs, timeouts and intervals, where the properties are defined at runtime.

Dynamic cron jobs

Obtain a reference to a `CronJob` instance by name from anywhere in your code using the `SchedulerRegistry` API. First, inject `SchedulerRegistry` using standard constructor injection:

```
constructor(private schedulerRegistry: SchedulerRegistry) {}
```

> info **Hint** Import the `SchedulerRegistry` from the `@nestjs/schedule` package.

Then use it in a class as follows. Assume a cron job was created with the following declaration:

```
@Cron('* * 8 * * *', {
  name: 'notifications',
})
triggerNotifications() {}
```

Access this job using the following:

```
const job = this.schedulerRegistry.getCronJob('notifications');

job.stop();
console.log(job.lastDate());
```

The `getCronJob()` method returns the named cron job. The returned `CronJob` object has the following methods:

- `stop()` - stops a job that is scheduled to run.
- `start()` - restarts a job that has been stopped.
- `setTime(time: CronTime)` - stops a job, sets a new time for it, and then starts it
- `lastDate()` - returns a string representation of the last date a job executed
- `nextDates(count: number)` - returns an array (size `count`) of `moment` objects representing upcoming job execution dates.

> info **Hint** Use `toDate()` on `moment` objects to render them in human readable form.

Create a new cron job dynamically using the `SchedulerRegistry#addCronJob` method, as follows:

```
addCronJob(name: string, seconds: string) {
  const job = new CronJob(`${seconds} * * * * *`, () => {
    this.logger.warn(`time (${seconds}) for job ${name} to run!`);
  });

  this.schedulerRegistry.addCronJob(name, job);
  job.start();

  this.logger.warn(
    `job ${name} added for each minute at ${seconds} seconds!`,
  );
}
```

In this code, we use the `CronJob` object from the `cron` package to create the cron job. The `CronJob` constructor takes a cron pattern (just like the `@Cron()` decorator) as its first argument, and a callback to be executed when the cron timer fires as its second argument. The `SchedulerRegistry#addCronJob` method takes two arguments: a name for the `CronJob`, and the `CronJob` object itself.

> warning **Warning** Remember to inject the `SchedulerRegistry` before accessing it. Import `CronJob` from the `cron` package.

Delete a named cron job using the `SchedulerRegistry#deleteCronJob` method, as follows:

```
deleteCron(name: string) {
  this.schedulerRegistry.deleteCronJob(name);
  this.logger.warn(`job ${name} deleted!`);
}
```

List all cron jobs using the `SchedulerRegistry#getCronJobs` method as follows:

```
getCrons() {
  const jobs = this.schedulerRegistry.getCronJobs();
  jobs.forEach((value, key, map) => {
    let next;
    try {
      next = value.nextDates().toDate();
    } catch (e) {
      next = 'error: next fire date is in the past!';
    }
    this.logger.log(`job: ${key} -> next: ${next}`);
  });
}
```

The `getCronJobs()` method returns a `map`. In this code, we iterate over the map and attempt to access the `nextDates()` method of each `CronJob`. In the `CronJob` API, if a job has already fired and has no future firing dates, it throws an exception.

Dynamic intervals

Obtain a reference to an interval with the `SchedulerRegistry#getInterval` method. As above, inject `SchedulerRegistry` using standard constructor injection:

```
constructor(private schedulerRegistry: SchedulerRegistry) {}
```

And use it as follows:

```
const interval = this.schedulerRegistry.getInterval('notifications');
clearInterval(interval);
```

Create a new interval dynamically using the `SchedulerRegistry#addInterval` method, as follows:

```
addInterval(name: string, milliseconds: number) {
  const callback = () => {
    this.logger.warn(`Interval ${name} executing at time
(${milliseconds})!`);
  };

  const interval = setInterval(callback, milliseconds);
  this.schedulerRegistry.addInterval(name, interval);
}
```

In this code, we create a standard JavaScript interval, then pass it to the `SchedulerRegistry#addInterval` method. That method takes two arguments: a name for the interval, and the interval itself.

Delete a named interval using the `SchedulerRegistry#deleteInterval` method, as follows:

```
deleteInterval(name: string) {
  this.schedulerRegistry.deleteInterval(name);
  this.logger.warn(`Interval ${name} deleted!`);
}
```

List all intervals using the `SchedulerRegistry#getIntervals` method as follows:

```
getIntervals() {
  const intervals = this.schedulerRegistry.getIntervals();
```

```
    intervals.forEach(key => this.logger.log(`Interval: ${key}`));
  }
```

Dynamic timeouts

Obtain a reference to a timeout with the `SchedulerRegistry#getTimeout` method. As above, inject `SchedulerRegistry` using standard constructor injection:

```
constructor(private readonly schedulerRegistry: SchedulerRegistry) {}
```

And use it as follows:

```
const timeout = this.schedulerRegistry.getTimeout('notifications');
clearTimeout(timeout);
```

Create a new timeout dynamically using the `SchedulerRegistry#addTimeout` method, as follows:

```
addTimeout(name: string, milliseconds: number) {
  const callback = () => {
    this.logger.warn(`Timeout ${name} executing after
(${milliseconds})!`);
  };

  const timeout = setTimeout(callback, milliseconds);
  this.schedulerRegistry.addTimeout(name, timeout);
}
```

In this code, we create a standard JavaScript timeout, then pass it to the `SchedulerRegistry#addTimeout` method. That method takes two arguments: a name for the timeout, and the timeout itself.

Delete a named timeout using the `SchedulerRegistry#deleteTimeout` method, as follows:

```
deleteTimeout(name: string) {
  this.schedulerRegistry.deleteTimeout(name);
  this.logger.warn(`Timeout ${name} deleted!`);
}
```

List all timeouts using the `SchedulerRegistry#getTimeouts` method as follows:

```
getTimeouts() {
  const timeouts = this.schedulerRegistry.getTimeouts();
```

```
    timeouts.forEach(key => this.logger.log(`Timeout: ${key}`));
  }
```

Example

A working example is available here.