Healthchecks (Terminus)

Terminus integration provides you with **readiness/liveness** health checks. Healthchecks are crucial when it comes to complex backend setups. In a nutshell, a health check in the realm of web development usually consists of a special address, for example, `https://my-website.com/health/readiness`. A service or a component of your infrastructure (e.g., Kubernetes) checks this address continuously. Depending on the HTTP status code returned from a `GET` request to this address the service will take action when it receives an "unhealthy" response. Since the definition of "healthy" or "unhealthy" varies with the type of service you provide, the **Terminus** integration supports you with a set of **health indicators**.

As an example, if your web server uses MongoDB to store its data, it would be vital information whether MongoDB is still up and running. In that case, you can make use of the `MongooseHealthIndicator`. If configured correctly - more on that later - your health check address will return a healthy or unhealthy HTTP status code, depending on whether MongoDB is running.

Getting started

To get started with `@nestjs/terminus` we need to install the required dependency.

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

Setting up a Healthcheck

A health check represents a summary of **health indicators**. A health indicator executes a check of a service, whether it is in a healthy or unhealthy state. A health check is positive if all the assigned health indicators are up and running. Because a lot of applications will need similar health indicators, `@nestjs/terminus` provides a set of predefined indicators, such as:

- `HttpHealthIndicator`
- `TypeOrmHealthIndicator`
- `MongooseHealthIndicator`
- `SequelizeHealthIndicator`
- `MikroOrmHealthIndicator`
- `PrismaHealthIndicator`
- `MicroserviceHealthIndicator`
- `GRPCHealthIndicator`
- `MemoryHealthIndicator`
- `DiskHealthIndicator`

To get started with our first health check, let's create the `HealthModule` and import the `TerminusModule` into it in its imports array.

> info **Hint** To create the module using the Nest CLI, simply execute the `$ nest g module health` command.

```
@@filename(health.module)
import { Module } from '@nestjs/common';
import { TerminusModule } from '@nestjs/terminus';

@Module({
  imports: [TerminusModule]
})
export class HealthModule {}
```

Our healthcheck(s) can be executed using a controller, which can be easily set up using the Nest CLI.

```
$ nest g controller health
```

> info **Info** It is highly recommended to enable shutdown hooks in your application. Terminus integration makes use of this lifecycle event if enabled. Read more about shutdown hooks here.

HTTP Healthcheck

Once we have installed `@nestjs/terminus`, imported our `TerminusModule` and created a new controller, we are ready to create a health check.

The `HTTPHealthIndicator` requires the `@nestjs/axios` package so make sure to have it installed:

```
$ npm i --save @nestjs/axios axios
```

Now we can setup our `HealthController`:

```
@@filename(health.controller)
import { Controller, Get } from '@nestjs/common';
import { HealthCheckService, HttpHealthIndicator, HealthCheck } from
'@nestjs/terminus';

@Controller('health')
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private http: HttpHealthIndicator,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.http.pingCheck('nestjs-docs', 'https://docs.nestjs.com'),
    ]);
  }
```

```
}
@@switch
import { Controller, Dependencies, Get } from '@nestjs/common';
import { HealthCheckService, HttpHealthIndicator, HealthCheck } from
'@nestjs/terminus';

@Controller('health')
@Dependencies(HealthCheckService, HttpHealthIndicator)
export class HealthController {
  constructor(
    private health,
    private http,
  ) { }

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.http.pingCheck('nestjs-docs', 'https://docs.nestjs.com'),
    ])
  }
}
```

```
@@filename(health.module)
import { Module } from '@nestjs/common';
import { TerminusModule } from '@nestjs/terminus';
import { HttpModule } from '@nestjs/axios';
import { HealthController } from './health.controller';

@Module({
  imports: [TerminusModule, HttpModule],
  controllers: [HealthController],
})
export class HealthModule {}
@@switch
import { Module } from '@nestjs/common';
import { TerminusModule } from '@nestjs/terminus';
import { HttpModule } from '@nestjs/axios';
import { HealthController } from './health.controller';

@Module({
  imports: [TerminusModule, HttpModule],
  controllers: [HealthController],
})
export class HealthModule {}
```

Our health check will now send a *GET*-request to the `https://docs.nestjs.com` address. If we get a healthy response from that address, our route at `http://localhost:3000/health` will return the following object with a 200 status code.

```
{
  "status": "ok",
  "info": {
    "nestjs-docs": {
      "status": "up"
    }
  },
  "error": {},
  "details": {
    "nestjs-docs": {
      "status": "up"
    }
  }
}
```

The interface of this response object can be accessed from the `@nestjs/terminus` package with the `HealthCheckResult` interface.

`status`	If any health indicator failed the status will be `'error'`. If the NestJS app is shutting down but still accepting HTTP requests, the health check will have the `'shutting_down'` status.	`'error' | 'ok' | 'shutting_down'`
`info`	Object containing information of each health indicator which is of status `'up'`, or in other words "healthy".	`object`
`error`	Object containing information of each health indicator which is of status `'down'`, or in other words "unhealthy".	`object`
`details`	Object containing all information of each health indicator	`object`

Check for specific HTTP response codes

In certain cases, you might want to check for specific criteria and validate the response. As an example, let's assume `https://my-external-service.com` returns a response code `204`. With `HttpHealthIndicator.responseCheck` you can check for that response code specifically and determine all other codes as unhealthy.

In case any other response code other than `204` gets returned, the following example would be unhealthy. The third parameter requires you to provide a function (sync or async) which returns a boolean whether the response is considered healthy (`true`) or unhealthy (`false`).

```
@@filename(health.controller)
// Within the `HealthController`-class

@Get()
@HealthCheck()
check() {
  return this.health.check([
    () =>
```

```
      this.http.responseCheck(
        'my-external-service',
        'https://my-external-service.com',
        (res) => res.status === 204,
      ),
    ]);
  }
```

TypeOrm health indicator

Terminus offers the capability to add database checks to your health check. In order to get started with this health indicator, you should check out the Database chapter and make sure your database connection within your application is established.

> info **Hint** Behind the scenes the `TypeOrmHealthIndicator` simply executes a `SELECT 1`-SQL command which is often used to verify whether the database still alive. In case you are using an Oracle database it uses `SELECT 1 FROM DUAL`.

```
@@filename(health.controller)
@Controller('health')
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private db: TypeOrmHealthIndicator,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.db.pingCheck('database'),
    ]);
  }
}
@@switch
@Controller('health')
@Dependencies(HealthCheckService, TypeOrmHealthIndicator)
export class HealthController {
  constructor(
    private health,
    private db,
  ) { }

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.db.pingCheck('database'),
    ])
  }
}
```

If your database is reachable, you should now see the following JSON-result when requesting `http://localhost:3000` with a `GET` request:

```
{
  "status": "ok",
  "info": {
    "database": {
      "status": "up"
    }
  },
  "error": {},
  "details": {
    "database": {
      "status": "up"
    }
  }
}
```

In case your app uses multiple databases, you need to inject each connection into your `HealthController`. Then, you can simply pass the connection reference to the `TypeOrmHealthIndicator`.

```
@@filename(health.controller)
@Controller('health')
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private db: TypeOrmHealthIndicator,
    @InjectConnection('albumsConnection')
    private albumsConnection: Connection,
    @InjectConnection()
    private defaultConnection: Connection,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.db.pingCheck('albums-database', { connection:
this.albumsConnection }),
      () => this.db.pingCheck('database', { connection:
this.defaultConnection }),
    ]);
  }
}
```

Disk health indicator

With the `DiskHealthIndicator` we can check how much storage is in use. To get started, make sure to inject the `DiskHealthIndicator` into your `HealthController`. The following example checks the storage used of the path `/` (or on Windows you can use `C:\\`). If that exceeds more than 50% of the total storage space it would response with an unhealthy Health Check.

```
@@filename(health.controller)
@Controller('health')
export class HealthController {
  constructor(
    private readonly health: HealthCheckService,
    private readonly disk: DiskHealthIndicator,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.disk.checkStorage('storage', { path: '/',
thresholdPercent: 0.5 }),
    ]);
  }
}
@@switch
@Controller('health')
@Dependencies(HealthCheckService, DiskHealthIndicator)
export class HealthController {
  constructor(health, disk) {}

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.disk.checkStorage('storage', { path: '/',
thresholdPercent: 0.5 }),
    ])
  }
}
```

With the `DiskHealthIndicator.checkStorage` function you also have the possibility to check for a fixed amount of space. The following example would be unhealthy in case the path `/my-app/` would exceed 250GB.

```
@@filename(health.controller)
// Within the `HealthController`-class

@Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.disk.checkStorage('storage', {  path: '/', threshold: 250 *
```

```
  1024 * 1024 * 1024, })
    ]);
  }
```

Memory health indicator

To make sure your process does not exceed a certain memory limit the `MemoryHealthIndicator` can be used. The following example can be used to check the heap of your process.

> info **Hint** Heap is the portion of memory where dynamically allocated memory resides (i.e. memory allocated via malloc). Memory allocated from the heap will remain allocated until one of the following occurs:
>
> - The memory is *free*'d
> - The program terminates

```
@@filename(health.controller)
@Controller('health')
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private memory: MemoryHealthIndicator,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
    ]);
  }
}
@@switch
@Controller('health')
@Dependencies(HealthCheckService, MemoryHealthIndicator)
export class HealthController {
  constructor(health, memory) {}

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
    ])
  }
}
```

It is also possible to verify the memory RSS of your process with `MemoryHealthIndicator.checkRSS`. This example would return an unhealthy response code in case your process does have more than 150MB

allocated.

> info **Hint** RSS is the Resident Set Size and is used to show how much memory is allocated to that process and is in RAM. It does not include memory that is swapped out. It does include memory from shared libraries as long as the pages from those libraries are actually in memory. It does include all stack and heap memory.

```
@@filename(health.controller)
// Within the `HealthController`-class

@Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.memory.checkRSS('memory_rss', 150 * 1024 * 1024),
  ]);
}
```

Custom health indicator

In some cases, the predefined health indicators provided by `@nestjs/terminus` do not cover all of your health check requirements. In that case, you can set up a custom health indicator according to your needs.

Let's get started by creating a service that will represent our custom indicator. To get a basic understanding of how an indicator is structured, we will create an example `DogHealthIndicator`. This service should have the state `'up'` if every `Dog` object has the type `'goodboy'`. If that condition is not satisfied then it should throw an error.

```
@@filename(dog.health)
import { Injectable } from '@nestjs/common';
import { HealthIndicator, HealthIndicatorResult, HealthCheckError } from
'@nestjs/terminus';

export interface Dog {
  name: string;
  type: string;
}

@Injectable()
export class DogHealthIndicator extends HealthIndicator {
  private dogs: Dog[] = [
    { name: 'Fido', type: 'goodboy' },
    { name: 'Rex', type: 'badboy' },
  ];

  async isHealthy(key: string): Promise<HealthIndicatorResult> {
    const badboys = this.dogs.filter(dog => dog.type === 'badboy');
    const isHealthy = badboys.length === 0;
    const result = this.getStatus(key, isHealthy, { badboys:
```

```
badboys.length });

    if (isHealthy) {
      return result;
    }
    throw new HealthCheckError('Dogcheck failed', result);
  }
}
@@switch
import { Injectable } from '@nestjs/common';
import { HealthCheckError } from '@godaddy/terminus';

@Injectable()
export class DogHealthIndicator extends HealthIndicator {
  dogs = [
    { name: 'Fido', type: 'goodboy' },
    { name: 'Rex', type: 'badboy' },
  ];

  async isHealthy(key) {
    const badboys = this.dogs.filter(dog => dog.type === 'badboy');
    const isHealthy = badboys.length === 0;
    const result = this.getStatus(key, isHealthy, { badboys:
badboys.length });

    if (isHealthy) {
      return result;
    }
    throw new HealthCheckError('Dogcheck failed', result);
  }
}
```

The next thing we need to do is register the health indicator as a provider.

```
@@filename(health.module)
import { Module } from '@nestjs/common';
import { TerminusModule } from '@nestjs/terminus';
import { DogHealthIndicator } from './dog.health';

@Module({
  controllers: [HealthController],
  imports: [TerminusModule],
  providers: [DogHealthIndicator]
})
export class HealthModule { }
```

> info **Hint** In a real-world application the `DogHealthIndicator` should be provided in a separate module, for example, `DogModule`, which then will be imported by the `HealthModule`.

The last required step is to add the now available health indicator in the required health check endpoint. For that, we go back to our `HealthController` and add it to our `check` function.

```
@@filename(health.controller)
import { HealthCheckService, HealthCheck } from '@nestjs/terminus';
import { Injectable, Dependencies, Get } from '@nestjs/common';
import { DogHealthIndicator } from './dog.health';

@Injectable()
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private dogHealthIndicator: DogHealthIndicator
  ) {}

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.dogHealthIndicator.isHealthy('dog'),
    ])
  }
}
@@switch
import { HealthCheckService, HealthCheck } from '@nestjs/terminus';
import { Injectable, Get } from '@nestjs/common';
import { DogHealthIndicator } from './dog.health';

@Injectable()
@Dependencies(HealthCheckService, DogHealthIndicator)
export class HealthController {
  constructor(
    private health,
    private dogHealthIndicator
  ) {}

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.dogHealthIndicator.isHealthy('dog'),
    ])
  }
}
```

Logging

Terminus only logs error messages, for instance when a Healthcheck has failed. With the `TerminusModule.forRoot()` method you have more control over how errors are being logged as well as completely take over the logging itself.

In this section, we are going to walk you through how you create a custom logger `TerminusLogger`. This logger extends the built-in logger. Therefore you can pick and choose which part of the logger you would like to overwrite

> info **Info** If you want to learn more about custom loggers in NestJS, read more here.

```
@@filename(terminus-logger.service)
import { Injectable, Scope, ConsoleLogger } from '@nestjs/common';

@Injectable({ scope: Scope.TRANSIENT })
export class TerminusLogger extends ConsoleLogger {
  error(message: any, stack?: string, context?: string): void;
  error(message: any, ...optionalParams: any[]): void;
  error(
    message: unknown,
    stack?: unknown,
    context?: unknown,
    ...rest: unknown[]
  ): void {
    // Overwrite here how error messages should be logged
  }
}
```

Once you have created your custom logger, all you need to do is simply pass it into the `TerminusModule.forRoot()` as such.

```
@@filename(health.module)
@Module({
imports: [
  TerminusModule.forRoot({
    logger: TerminusLogger,
  }),
],
})
export class HealthModule {}
```

To completely suppress any log messages coming from Terminus, including error messages, configure Terminus as such.

```
@@filename(health.module)
@Module({
imports: [
  TerminusModule.forRoot({
    logger: false,
  }),
],
})
export class HealthModule {}
```

Terminus allows you to configure how Healthcheck errors should be displayed in your logs.

Error Log Style	Description	Example
`json` (default)	Prints a summary of the health check result in case of an error as JSON object	
`pretty`	Prints a summary of the health check result in case of an error within formatted boxes and highlights successful/erroneous results	

You can change the log style using the `errorLogStyle` configuration option as in the following snippet.

```
@@filename(health.module)
@Module({
  imports: [
    TerminusModule.forRoot({
      errorLogStyle: 'pretty',
    }),
  ]
})
export class HealthModule {}
```

More examples

More working examples are available here.