Serverless

Serverless computing is a cloud computing execution model in which the cloud provider allocates machine resources on-demand, taking care of the servers on behalf of their customers. When an app is not in use, there are no computing resources allocated to the app. Pricing is based on the actual amount of resources consumed by an application (source).

With a **serverless architecture**, you focus purely on the individual functions in your application code. Services such as AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions take care of all the physical hardware, virtual machine operating system, and web server software management.

> info **Hint** This chapter does not cover the pros and cons of serverless functions nor dives into the specifics of any cloud providers.

Cold start

A cold start is the first time your code has been executed in a while. Depending on a cloud provider you use, it may span several different operations, from downloading the code and bootstrapping the runtime to eventually running your code. This process adds **significant latency** depending on several factors, the language, the number of packages your application require, etc.

The cold start is important and although there are things which are beyond our control, there's still a lot of things we can do on our side to make it as short as possible.

While you can think of Nest as a fully-fledged framework designed to be used in complex, enterprise applications, it is also **suitable for much "simpler" applications** (or scripts). For example, with the use of Standalone applications feature, you can take advantage of Nest's DI system in simple workers, CRON jobs, CLIs, or serverless functions.

Benchmarks

To better understand what's the cost of using Nest or other, well-known libraries (like `express`) in the context of serverless functions, let's compare how much time Node runtime needs to run the following scripts:

```
// #1 Express
import * as express from 'express';

async function bootstrap() {
  const app = express();
  app.get('/', (req, res) => res.send('Hello world!'));
  await new Promise<void>((resolve) => app.listen(3000, resolve));
}
bootstrap();

// #2 Nest (with @nestjs/platform-express)
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
```

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule, { logger: ['error'] });
  await app.listen(3000);
}
bootstrap();

// #3 Nest as a Standalone application (no HTTP server)
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { AppService } from './app.service';

async function bootstrap() {
  const app = await NestFactory.createApplicationContext(AppModule, {
    logger: ['error'],
  });
  console.log(app.get(AppService).getHello());
}
bootstrap();

// #4 Raw Node.js script
async function bootstrap() {
  console.log('Hello world!');
}
bootstrap();
```

For all these scripts, we used the `tsc` (TypeScript) compiler and so the code remains unbundled (`webpack` isn't used).

Express	0.0079s (7.9ms)
Nest with `@nestjs/platform-express`	0.1974s (197.4ms)
Nest (standalone application)	0.1117s (111.7ms)
Raw Node.js script	0.0071s (7.1ms)

> info **Note** Machine: MacBook Pro Mid 2014, 2.5 GHz Quad-Core Intel Core i7, 16 GB 1600 MHz DDR3, SSD.

Now, let's repeat all benchmarks but this time, using `webpack` (if you have Nest CLI installed, you can run `nest build --webpack`) to bundle our application into a single executable JavaScript file. However, instead of using the default `webpack` configuration that Nest CLI ships with, we'll make sure to bundle all dependencies (`node_modules`) together, as follows:

```
module.exports = (options, webpack) => {
  const lazyImports = [
    '@nestjs/microservices/microservices-module',
    '@nestjs/websockets/socket-module',
  ];

  return {
```

```
      ...options,
      externals: [],
      plugins: [
        ...options.plugins,
        new webpack.IgnorePlugin({
          checkResource(resource) {
            if (lazyImports.includes(resource)) {
              try {
                require.resolve(resource);
              } catch (err) {
                return true;
              }
            }
            return false;
          },
        }),
      ],
    };
  };
```

> info **Hint** To instruct Nest CLI to use this configuration, create a new `webpack.config.js` file in the root directory of your project.

With this configuration, we received the following results:

Express	0.0068s (6.8ms)
Nest with `@nestjs/platform-express`	0.0815s (81.5ms)
Nest (standalone application)	0.0319s (31.9ms)
Raw Node.js script	0.0066s (6.6ms)

> info **Note** Machine: MacBook Pro Mid 2014, 2.5 GHz Quad-Core Intel Core i7, 16 GB 1600 MHz DDR3, SSD.

> info **Hint** You could optimize it even further by applying additional code minification & optimization techniques (using `webpack` plugins, etc.).

As you can see, the way you compile (and whether you bundle your code) is crucial and has a significant impact on the overall startup time. With `webpack`, you can get the bootstrap time of a standalone Nest application (starter project with one module, controller, and service) down to ~32ms on average, and down to ~81.5ms for a regular HTTP, express-based NestJS app.

For more complicated Nest applications, for example, with 10 resources (generated through `$ nest g resource` schematic = 10 modules, 10 controllers, 10 services, 20 DTO classes, 50 HTTP endpoints + `AppModule`), the overall startup on MacBook Pro Mid 2014, 2.5 GHz Quad-Core Intel Core i7, 16 GB 1600 MHz DDR3, SSD is approximately 0.1298s (129.8ms). Running a monolithic application as a serverless function typically doesn't make too much sense anyway, so think of this benchmark more as an example of how the bootstrap time may potentially increase as your application grows.

Runtime optimizations

Thus far we covered compile-time optimizations. These are unrelated to the way you define providers and load Nest modules in your application, and that plays an essential role as your application gets bigger.

For example, imagine having a database connection defined as an asynchronous provider. Async providers are designed to delay the application start until one or more asynchronous tasks are completed. That means, if your serverless function on average requires 2s to connect to the database (on bootstrap), your endpoint will need at least two extra seconds (because it must wait till the connection is established) to send a response back (when it's a cold start and your application wasn't running already).

As you can see, the way you structure your providers is somewhat different in a **serverless environment** where bootstrap time is important. Another good example is if you use Redis for caching, but only in certain scenarios. Perhaps, in this case, you should not define a Redis connection as an async provider, as it would slow down the bootstrap time, even if it's not required for this specific function invocation.

Also, sometimes you could lazy-load entire modules, using the `LazyModuleLoader` class, as described in this chapter. Caching is a great example here too. Imagine that your application has, let's say, `CacheModule` which internally connects to Redis and also, exports the `CacheService` to interact with the Redis storage. If you don't need it for all potential function invocations, you can just load it on-demand, lazily. This way you'll get a faster startup time (when a cold start occurs) for all invocations that don't require caching.

```
if (request.method === RequestMethod[RequestMethod.GET]) {
  const { CacheModule } = await import('./cache.module');
  const moduleRef = await this.lazyModuleLoader.load(() => CacheModule);

  const { CacheService } = await import('./cache.service');
  const cacheService = moduleRef.get(CacheService);

  return cacheService.get(ENDPOINT_KEY);
}
```

Another great example is a webhook or worker, which depending on some specific conditions (e.g., input arguments), may perform different operations. In such a case, you could specify a condition inside your route handler that lazily loads an appropriate module for the specific function invocation, and just load every other module lazily.

```
if (workerType === WorkerType.A) {
  const { WorkerAModule } = await import('./worker-a.module');
  const moduleRef = await this.lazyModuleLoader.load(() => WorkerAModule);
  // ...
} else if (workerType === WorkerType.B) {
  const { WorkerBModule } = await import('./worker-b.module');
  const moduleRef = await this.lazyModuleLoader.load(() => WorkerBModule);
  // ...
}
```

Example integration

The way your application's entry file (typically `main.ts` file) is supposed to look like **depends on several factors** and so **there's no single template** that just works for every scenario. For example, the initialization file required to spin up your serverless function varies by cloud providers (AWS, Azure, GCP, etc.). Also, depending on whether you want to run a typical HTTP application with multiple routes/endpoints or just provide a single route (or execute a specific portion of code), your application's code will look different (for example, for the endpoint-per-function approach you could use the `NestFactory.createApplicationContext` instead of booting the HTTP server, setting up middleware, etc.).

Just for illustration purposes, we'll integrate Nest (using `@nestjs/platform-express` and so spinning up the whole, fully functional HTTP router) with the Serverless framework (in this case, targeting AWS Lambda). As we've mentioned earlier, your code will differ depending on the cloud provider you choose, and many other factors.

First, let's install the required packages:

```
$ npm i @vendia/serverless-express aws-lambda
$ npm i -D @types/aws-lambda serverless-offline
```

> info **Hint** To speed up development cycles, we install the `serverless-offline` plugin which emulates AWS λ and API Gateway.

Once the installation process is complete, let's create the `serverless.yml` file to configure the Serverless framework:

```
service: serverless-example

plugins:
  - serverless-offline

provider:
  name: aws
  runtime: nodejs14.x

functions:
  main:
    handler: dist/main.handler
    events:
      - http:
          method: ANY
          path: /
      - http:
          method: ANY
          path: '{proxy+}'
```

> info **Hint** To learn more about the Serverless framework, visit the official documentation.

With this place, we can now navigate to the `main.ts` file and update our bootstrap code with the required boilerplate:

```
import { NestFactory } from '@nestjs/core';
import serverlessExpress from '@vendia/serverless-express';
import { Callback, Context, Handler } from 'aws-lambda';
import { AppModule } from './app.module';

let server: Handler;

async function bootstrap(): Promise<Handler> {
  const app = await NestFactory.create(AppModule);
  await app.init();

  const expressApp = app.getHttpAdapter().getInstance();
  return serverlessExpress({ app: expressApp });
}

export const handler: Handler = async (
  event: any,
  context: Context,
  callback: Callback,
) => {
  server = server ?? (await bootstrap());
  return server(event, context, callback);
};
```

> info **Hint** For creating multiple serverless functions and sharing common modules between them, we recommend using the CLI Monorepo mode.

> warning **Warning** If you use `@nestjs/swagger` package, there are a few additional steps required to make it work properly in the context of serverless function. Check out this thread for more information.

Next, open up the `tsconfig.json` file and make sure to enable the `esModuleInterop` option to make the `@vendia/serverless-express` package load properly.

```
{
  "compilerOptions": {
    ...
    "esModuleInterop": true
  }
}
```

Now we can build our application (with `nest build` or `tsc`) and use the `serverless` CLI to start our lambda function locally:

```
$ npm run build
$ npx serverless offline
```

Once the application is running, open your browser and navigate to `http://localhost:3000/dev/[ANY_ROUTE]` (where `[ANY_ROUTE]` is any endpoint registered in your application).

In the sections above, we've shown that using `webpack` and bundling your app can have significant impact on the overall bootstrap time. However, to make it work with our example, there are a few additional configurations you must add in your `webpack.config.js` file. Generally, to make sure our `handler` function will be picked up, we must change the `output.libraryTarget` property to `commonjs2`.

```
return {
  ...options,
  externals: [],
  output: {
    ...options.output,
    libraryTarget: 'commonjs2',
  },
  // ... the rest of the configuration
};
```

With this in place, you can now use `$ nest build --webpack` to compile your function's code (and then `$ npx serverless offline` to test it).

It's also recommended (but **not required** as it will slow down your build process) to install the `terser-webpack-plugin` package and override its configuration to keep classnames intact when minifying your production build. Not doing so can result in incorrect behavior when using `class-validator` within your application.

```
const TerserPlugin = require('terser-webpack-plugin');

return {
  ...options,
  externals: [],
  optimization: {
    minimizer: [
      new TerserPlugin({
        terserOptions: {
          keep_classnames: true,
        },
      }),
    ],
  },
  output: {
    ...options.output,
    libraryTarget: 'commonjs2',
```

```
  },
  // ... the rest of the configuration
};
```

Using standalone application feature

Alternatively, if you want to keep your function very lightweight and you don't need any HTTP-related features (routing, but also guards, interceptors, pipes, etc.), you can just use `NestFactory.createApplicationContext` (as mentioned earlier) instead of running the entire HTTP server (and `express` under the hood), as follows:

```
@@filename(main)
import { HttpStatus } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { Callback, Context, Handler } from 'aws-lambda';
import { AppModule } from './app.module';
import { AppService } from './app.service';

export const handler: Handler = async (
  event: any,
  context: Context,
  callback: Callback,
) => {
  const appContext = await
NestFactory.createApplicationContext(AppModule);
  const appService = appContext.get(AppService);

  return {
    body: appService.getHello(),
    statusCode: HttpStatus.OK,
  };
};
```

> info **Hint** Be aware that `NestFactory.createApplicationContext` does not wrap controller methods with enhancers (guard, interceptors, etc.). For this, you must use the `NestFactory.create` method.

You could also pass the `event` object down to, let's say, `EventsService` provider that could process it and return a corresponding value (depending on the input value and your business logic).

```
export const handler: Handler = async (
  event: any,
  context: Context,
  callback: Callback,
) => {
  const appContext = await
NestFactory.createApplicationContext(AppModule);
  const eventsService = appContext.get(EventsService);
```

```
  return eventsService.process(event);
};
```

HTTP adapter

Occasionally, you may want to access the underlying HTTP server, either within the Nest application context or from the outside.

Every native (platform-specific) HTTP server/library (e.g., Express and Fastify) instance is wrapped in an **adapter**. The adapter is registered as a globally available provider that can be retrieved from the application context, as well as injected into other providers.

Outside application context strategy

To get a reference to the `HttpAdapter` from outside of the application context, call the `getHttpAdapter()` method.

```
@@filename()
const app = await NestFactory.create(AppModule);
const httpAdapter = app.getHttpAdapter();
```

In-context strategy

To get a reference to the `HttpAdapterHost` from within the application context, inject it using the same technique as any other existing provider (e.g., using constructor injection).

```
@@filename()
export class CatsService {
  constructor(private adapterHost: HttpAdapterHost) {}
}
@@switch
@Dependencies(HttpAdapterHost)
export class CatsService {
  constructor(adapterHost) {
    this.adapterHost = adapterHost;
  }
}
```

> info **Hint** The `HttpAdapterHost` is imported from the `@nestjs/core` package.

The `HttpAdapterHost` is **not** an actual `HttpAdapter`. To get the actual `HttpAdapter` instance, simply access the `httpAdapter` property.

```
const adapterHost = app.get(HttpAdapterHost);
const httpAdapter = adapterHost.httpAdapter;
```

The `httpAdapter` is the actual instance of the HTTP adapter used by the underlying framework. It is an instance of either `ExpressAdapter` or `FastifyAdapter` (both classes extend `AbstractHttpAdapter`).

The adapter object exposes several useful methods to interact with the HTTP server. However, if you want to access the library instance (e.g., the Express instance) directly, call the `getInstance()` method.

```
const instance = httpAdapter.getInstance();
```

Global prefix

To set a prefix for **every route** registered in an HTTP application, use the `setGlobalPrefix()` method of the `INestApplication` instance.

```
const app = await NestFactory.create(AppModule);
app.setGlobalPrefix('v1');
```

You can exclude routes from the global prefix using the following construction:

```
app.setGlobalPrefix('v1', {
  exclude: [{ path: 'health', method: RequestMethod.GET }],
});
```

Alternatively, you can specify route as a string (it will apply to every request method):

```
app.setGlobalPrefix('v1', { exclude: ['cats'] });
```

> info **Hint** The `path` property supports wildcard parameters using the path-to-regexp package. Note: this does not accept wildcard asterisks `*`. Instead, you must use parameters (e.g., `(.*)`, `:splat*`).

Raw body

One of the most common use-case for having access to the raw request body is performing webhook signature verifications. Usually to perform webhook signature validations the unserialized request body is required to calculate an HMAC hash.

> warning **Warning** This feature can be used only if the built-in global body parser middleware is enabled, ie., you must not pass `bodyParser: false` when creating the app.

Use with Express

First enable the option when creating your Nest Express application:

```
import { NestFactory } from '@nestjs/core';
import type { NestExpressApplication } from '@nestjs/platform-express';
import { AppModule } from './app.module';

// in the "bootstrap" function
const app = await NestFactory.create<NestExpressApplication>(AppModule, {
  rawBody: true,
});
await app.listen(3000);
```

To access the raw request body in a controller, a convenience interface `RawBodyRequest` is provided to expose a `rawBody` field on the request: use the interface `RawBodyRequest` type:

```
import { Controller, Post, RawBodyRequest, Req } from '@nestjs/common';
import { Request } from 'express';

@Controller('cats')
class CatsController {
  @Post()
  create(@Req() req: RawBodyRequest<Request>) {
    const raw = req.rawBody; // returns a `Buffer`.
  }
}
```

Registering a different parser

By default, only `json` and `urlencoded` parsers are registered. If you want to register a different parser on the fly, you will need to do so explicitly.

For example, to register a `text` parser, you can use the following code:

```
app.useBodyParser('text');
```

> warning **Warning** Ensure that you are providing the correct application type to the `NestFactory.create` call. For Express applications, the correct type is `NestExpressApplication`. Otherwise the `.useBodyParser` method will not be found.

Body parser size limit

If your application needs to parse a body larger than the default `100kb` of Express, use the following:

```
app.useBodyParser('json', { limit: '10mb' });
```

The `.useBodyParser` method will respect the `rawBody` option that is passed in the application options.

Use with Fastify

First enable the option when creating your Nest Fastify application:

```
import { NestFactory } from '@nestjs/core';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
import { AppModule } from './app.module';

// in the "bootstrap" function
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  new FastifyAdapter(),
  {
    rawBody: true,
  },
);
await app.listen(3000);
```

To access the raw request body in a controller, a convenience interface `RawBodyRequest` is provided to expose a `rawBody` field on the request: use the interface `RawBodyRequest` type:

```
import { Controller, Post, RawBodyRequest, Req } from '@nestjs/common';
import { FastifyRequest } from 'fastify';

@Controller('cats')
class CatsController {
  @Post()
  create(@Req() req: RawBodyRequest<FastifyRequest>) {
    const raw = req.rawBody; // returns a `Buffer`.
  }
}
```

Registering a different parser

By default, only `application/json` and `application/x-www-form-urlencoded` parsers are registered. If you want to register a different parser on the fly, you will need to do so explicitly.

For example, to register a `text/plain` parser, you can use the following code:

```
app.useBodyParser('text/plain');
```

> warning **Warning** Ensure that you are providing the correct application type to the `NestFactory.create` call. For Fastify applications, the correct type is `NestFastifyApplication`. Otherwise the `.useBodyParser` method will not be found.

Body parser size limit

If your application needs to parse a body larger than the default 1MiB of Fastify, use the following:

```
const bodyLimit = 10_485_760; // 10MiB
app.useBodyParser('application/json', { bodyLimit });
```

The `.useBodyParser` method will respect the `rawBody` option that is passed in the application options.

Hybrid application

A hybrid application is one that both listens for HTTP requests, as well as makes use of connected microservices. The `INestApplication` instance can be connected with `INestMicroservice` instances through the `connectMicroservice()` method.

```
const app = await NestFactory.create(AppModule);
const microservice = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.TCP,
});

await app.startAllMicroservices();
await app.listen(3001);
```

To connect multiple microservice instances, issue the call to `connectMicroservice()` for each microservice:

```
const app = await NestFactory.create(AppModule);
// microservice #1
const microserviceTcp = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.TCP,
  options: {
    port: 3001,
  },
});
// microservice #2
const microserviceRedis = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.REDIS,
  options: {
    host: 'localhost',
    port: 6379,
  },
});

await app.startAllMicroservices();
await app.listen(3001);
```

To bind `@MessagePattern()` to only one transport strategy (for example, MQTT) in a hybrid application with multiple microservices, we can pass the second argument of type `Transport` which is an enum with all the built-in transport strategies defined.

```
@@filename()
@MessagePattern('time.us.*', Transport.NATS)
getDate(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
```

```
@MessagePattern({ cmd: 'time.us' }, Transport.TCP)
getTCPDate(@Payload() data: number[]) {
  return new Date().toLocaleTimeString(...);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*', Transport.NATS)
getDate(data, context) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
@Bind(Payload(), Ctx())
@MessagePattern({ cmd: 'time.us' }, Transport.TCP)
getTCPDate(data, context) {
  return new Date().toLocaleTimeString(...);
}
```

> info **Hint** `@Payload()`, `@Ctx()`, `Transport` and `NatsContext` are imported from `@nestjs/microservices`.

Sharing configuration

By default a hybrid application will not inherit global pipes, interceptors, guards and filters configured for the main (HTTP-based) application. To inherit these configuration properties from the main application, set the `inheritAppConfig` property in the second argument (an optional options object) of the `connectMicroservice()` call, as follow:

```
const microservice = app.connectMicroservice<MicroserviceOptions>(
  {
    transport: Transport.TCP,
  },
  { inheritAppConfig: true },
);
```

HTTPS

To create an application that uses the HTTPS protocol, set the `httpsOptions` property in the options object passed to the `create()` method of the `NestFactory` class:

```
const httpsOptions = {
  key: fs.readFileSync('./secrets/private-key.pem'),
  cert: fs.readFileSync('./secrets/public-certificate.pem'),
};
const app = await NestFactory.create(AppModule, {
  httpsOptions,
});
await app.listen(3000);
```

If you use the `FastifyAdapter`, create the application as follows:

```
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  new FastifyAdapter({ https: httpsOptions }),
);
```

Multiple simultaneous servers

The following recipe shows how to instantiate a Nest application that listens on multiple ports (for example, on a non-HTTPS port and an HTTPS port) simultaneously.

```
const httpsOptions = {
  key: fs.readFileSync('./secrets/private-key.pem'),
  cert: fs.readFileSync('./secrets/public-certificate.pem'),
};

const server = express();
const app = await NestFactory.create(
  AppModule,
  new ExpressAdapter(server),
);
await app.init();

const httpServer = http.createServer(server).listen(3000);
const httpsServer = https.createServer(httpsOptions, server).listen(443);
```

Because we called `http.createServer` / `https.createServer` ourselves, NestJS doesn't close them when calling `app.close` / on termination signal. We need to do this ourselves:

```
@Injectable()
export class ShutdownObserver implements OnApplicationShutdown {
  private httpServers: http.Server[] = [];

  public addHttpServer(server: http.Server): void {
    this.httpServers.push(server);
  }

  public async onApplicationShutdown(): Promise<void> {
    await Promise.all(
      this.httpServers.map((server) =>
        new Promise((resolve, reject) => {
          server.close((error) => {
            if (error) {
              reject(error);
            } else {
              resolve(null);
            }
          });
        })
      ),
    );
  }
}

const shutdownObserver = app.get(ShutdownObserver);
shutdownObserver.addHttpServer(httpServer);
shutdownObserver.addHttpServer(httpsServer);
```

> info **Hint** The `ExpressAdapter` is imported from the `@nestjs/platform-express` package. The `http` and `https` packages are native Node.js packages.

> **Warning** This recipe does not work with GraphQL Subscriptions.

Request lifecycle

Nest applications handle requests and produce responses in a sequence we refer to as the **request lifecycle**. With the use of middleware, pipes, guards, and interceptors, it can be challenging to track down where a particular piece of code executes during the request lifecycle, especially as global, controller level, and route level components come into play. In general, a request flows through middleware to guards, then to interceptors, then to pipes and finally back to interceptors on the return path (as the response is generated).

Middleware

Middleware is executed in a particular sequence. First, Nest runs globally bound middleware (such as middleware bound with `app.use`) and then it runs module bound middleware, which are determined on paths. Middleware are run sequentially in the order they are bound, similar to the way middleware in Express works. In the case of middleware bound across different modules, the middleware bound to the root module will run first, and then middleware will run in the order that the modules are added to the imports array.

Guards

Guard execution starts with global guards, then proceeds to controller guards, and finally to route guards. As with middleware, guards run in the order in which they are bound. For example:

```
@UseGuards(Guard1, Guard2)
@Controller('cats')
export class CatsController {
  constructor(private catsService: CatsService) {}

  @UseGuards(Guard3)
  @Get()
  getCats(): Cats[] {
    return this.catsService.getCats();
  }
}
```

`Guard1` will execute before `Guard2` and both will execute before `Guard3`.

> info **Hint** When speaking about globally bound vs controller or locally bound, the difference is where the guard (or other component is bound). If you are using `app.useGlobalGuard()` or providing the component via a module, it is globally bound. Otherwise, it is bound to a controller if the decorator precedes a controller class, or to a route if the decorator proceeds a route declaration.

Interceptors

Interceptors, for the most part, follow the same pattern as guards, with one catch: as interceptors return RxJS Observables, the observables will be resolved in a first in last out manner. So inbound requests will go through the standard global, controller, route level resolution, but the response side of the request (i.e., after returning from the controller method handler) will be resolved from route to controller to global. Also,

any errors thrown by pipes, controllers, or services can be read in the `catchError` operator of an interceptor.

Pipes

Pipes follow the standard global to controller to route bound sequence, with the same first in first out in regards to the `@UsePipes()` parameters. However, at a route parameter level, if you have multiple pipes running, they will run in the order of the last parameter with a pipe to the first. This also applies to the route level and controller level pipes. For example, if we have the following controller:

```
@UsePipes(GeneralValidationPipe)
@Controller('cats')
export class CatsController {
  constructor(private catsService: CatsService) {}

  @UsePipes(RouteSpecificPipe)
  @Patch(':id')
  updateCat(
    @Body() body: UpdateCatDTO,
    @Param() params: UpdateCatParams,
    @Query() query: UpdateCatQuery,
  ) {
    return this.catsService.updateCat(body, params, query);
  }
}
```

then the `GeneralValidationPipe` will run for the `query`, then the `params`, and then the `body` objects before moving on to the `RouteSpecificPipe`, which follows the same order. If any parameter-specific pipes were in place, they would run (again, from the last to first parameter) after the controller and route level pipes.

Filters

Filters are the only component that do not resolve global first. Instead, filters resolve from the lowest level possible, meaning execution starts with any route bound filters and proceeding next to controller level, and finally to global filters. Note that exceptions cannot be passed from filter to filter; if a route level filter catches the exception, a controller or global level filter cannot catch the same exception. The only way to achieve an effect like this is to use inheritance between the filters.

> info **Hint** Filters are only executed if any uncaught exception occurs during the request process. Caught exceptions, such as those caught with a `try/catch` will not trigger Exception Filters to fire. As soon as an uncaught exception is encountered, the rest of the lifecycle is ignored and the request skips straight to the filter.

Summary

In general, the request lifecycle looks like the following:

1. Incoming request

2. Middleware
    - 2.1. Globally bound middleware
    - 2.2. Module bound middleware
3. Guards
    - 3.1 Global guards
    - 3.2 Controller guards
    - 3.3 Route guards
4. Interceptors (pre-controller)
    - 4.1 Global interceptors
    - 4.2 Controller interceptors
    - 4.3 Route interceptors
5. Pipes
    - 5.1 Global pipes
    - 5.2 Controller pipes
    - 5.3 Route pipes
    - 5.4 Route parameter pipes
6. Controller (method handler)
7. Service (if exists)
8. Interceptors (post-request)
    - 8.1 Route interceptor
    - 8.2 Controller interceptor
    - 8.3 Global interceptor
9. Exception filters
    - 9.1 route
    - 9.2 controller
    - 9.3 global
10. Server response

Common errors

During your development with NestJS, you may encounter various errors as you learn the framework.

"Cannot resolve dependency" error

> info **Hint** Check out the NestJS Devtools which can help you resolve the "Cannot resolve dependency" error effortlessly.

Probably the most common error message is about Nest not being able to resolve dependencies of a provider. The error message usually looks something like this:

```
Nest can't resolve dependencies of the <provider> (?). Please make sure
that the argument <unknown_token> at index [<index>] is available in the
<module> context.

Potential solutions:
- Is <module> a valid NestJS module?
- If <unknown_token> is a provider, is it part of the current <module>?
- If <unknown_token> is exported from a separate @Module, is that module
imported within <module>?
  @Module({
    imports: [ /* the Module containing <unknown_token> */ ]
  })
```

The most common culprit of the error, is not having the `<provider>` in the module's `providers` array. Please make sure that the provider is indeed in the `providers` array and following standard NestJS provider practices.

There are a few gotchas, that are common. One is putting a provider in an `imports` array. If this is the case, the error will have the provider's name where `<module>` should be.

If you run across this error while developing, take a look at the module mentioned in the error message and look at its `providers`. For each provider in the `providers` array, make sure the module has access to all of the dependencies. Often times, `providers` are duplicated in a "Feature Module" and a "Root Module" which means Nest will try to instantiate the provider twice. More than likely, the module containing the `<provider>` being duplicated should be added in the "Root Module"'s `imports` array instead.

If the `<unknown_token>` above is the string `dependency`, you might have a circular file import. This is different from the circular dependency below because instead of having providers depend on each other in their constructors, it just means that two files end up importing each other. A common case would be a module file declaring a token and importing a provider, and the provider import the token constant from the module file. If you are using barrel files, ensure that your barrel imports do not end up creating these circular imports as well.

If the `<unknown_token>` above is the string `Object`, it means that you're injecting using an type/interface without a proper provider's token. To fix that, make sure you're importing the class reference or use a custom token with `@Inject()` decorator. Read the custom providers page.

Also, make sure you didn't end up injecting the provider on itself because self-injections are not allowed in NestJS. When this happens, `<unknown_token>` will likely be equal to `<provider>`.

If you are in a **monorepo setup**, you may face the same error as above but for core provider called `ModuleRef` as a `<unknown_token>`:

```
Nest can't resolve dependencies of the <provider> (?).
Please make sure that the argument ModuleRef at index [<index>] is
available in the <module> context.
...
```

This likely happens when your project end up loading two Node modules of the package `@nestjs/core`, like this:

```
.
├── package.json
├── apps
│   └── api
│       └── node_modules
│           └── @nestjs/bull
│               └── node_modules
│                   └── @nestjs/core
└── node_modules
    ├── (other packages)
    └── @nestjs/core
```

Solutions:

- For **Yarn** Workspaces, use the nohoist feature to prevent hoisting the package `@nestjs/core`.
- For **pnpm** Workspaces, set `@nestjs/core` as a peerDependencies in your other module and `"dependenciesMeta": {{ '{' }}"other-module-name": {{ '{' }}"injected": true{{ '}}' }}` in the app package.json where the module is imported. see: dependenciesmetainjected

"Circular dependency" error

Occasionally you'll find it difficult to avoid circular dependencies in your application. You'll need to take some steps to help Nest resolve these. Errors that arise from circular dependencies look like this:

```
Nest cannot create the <module> instance.
The module at index [<index>] of the <module> "imports" array is
undefined.

Potential causes:
- A circular dependency between modules. Use forwardRef() to avoid it.
Read more: https://docs.nestjs.com/fundamentals/circular-dependency
- The module at index [<index>] is of type "undefined". Check your import
```

```
  statements and the type of the module.

  Scope [<module_import_chain>]
  # example chain AppModule -> FooModule
```

Circular dependencies can arise from both providers depending on each other, or typescript files depending on each other for constants, such as exporting constants from a module file and importing them in a service file. In the latter case, it is advised to create a separate file for your constants. In the former case, please follow the guide on circular dependencies and make sure that both the modules **and** the providers are marked with `forwardRef`.

Debugging dependency errors

Along with just manually verifying your dependencies are correct, as of Nest 8.1.0 you can set the `NEST_DEBUG` environment variable to a string that resolves as truthy, and get extra logging information while Nest is resolving all of the dependencies for the application.

In the above image, the string in yellow is the host class of the dependency being injected, the string in blue is the name of the injected dependency, or its injection token, and the string in purple is the module in which the dependency is being searched for. Using this, you can usually trace back the dependency resolution for what's happening and why you're getting dependency injection problems.

"File change detected" loops endlessly

Windows users who are using TypeScript version 4.9 and up may encounter this problem. This happens when you're trying to run your application in watch mode, e.g `npm run start:dev` and see an endless loop of the log messages:

```
XX:XX:XX AM - File change detected. Starting incremental compilation...
XX:XX:XX AM - Found 0 errors. Watching for file changes.
```

When you're using the NestJS CLI to start your application in watch mode it is done by calling `tsc --watch`, and as of version 4.9 of TypeScript, a new strategy for detecting file changes is used which is likely to be the cause of this problem. In order to fix this problem, you need to add a setting to your tsconfig.json file after the `"compilerOptions"` option as follows:

```
  "watchOptions": {
    "watchFile": "fixedPollingInterval"
  }
```

This tells TypeScript to use the polling method for checking for file changes instead of file system events (the new default method), which can cause issues on some machines. You can read more about the `"watchFile"` option in TypeScript documentation.