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);
  };
```