Lazy-loading modules

By default, modules are eagerly loaded, which means that as soon as the application loads, so do all the modules, whether or not they are immediately necessary. While this is fine for most applications, it may become a bottleneck for apps/workers running in the **serverless environment**, where the startup latency ("cold start") is crucial.

Lazy loading can help decrease bootstrap time by loading only modules required by the specific serverless function invocation. In addition, you could also load other modules asynchronously once the serverless function is "warm" to speed-up the bootstrap time for subsequent calls even further (deferred modules registration).

> info **Hint** If you're familiar with the **Angular** framework, you might have seen the "lazy-loading modules" term before. Be aware that this technique is **functionally different** in Nest and so think about this as an entirely different feature that shares similar naming conventions.

> warning **Warning** Do note that lifecycle hooks methods are not invoked in lazy loaded modules and services.

Getting started

To load modules on-demand, Nest provides the `LazyModuleLoader` class that can be injected into a class in the normal way:

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(private lazyModuleLoader: LazyModuleLoader) {}
}
@@switch
@Injectable()
@Dependencies(LazyModuleLoader)
export class CatsService {
  constructor(lazyModuleLoader) {
    this.lazyModuleLoader = lazyModuleLoader;
  }
}
```

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

Alternatively, you can obtain a reference to the `LazyModuleLoader` provider from within your application bootstrap file (`main.ts`), as follows:

```
// "app" represents a Nest application instance
const lazyModuleLoader = app.get(LazyModuleLoader);
```

With this in place, you can now load any module using the following construction:

```
const { LazyModule } = await import('./lazy.module');
const moduleRef = await this.lazyModuleLoader.load(() => LazyModule);
```

info **Hint** "Lazy-loaded" modules are **cached** upon the first `LazyModuleLoader#load` method invocation. That means, each consecutive attempt to load `LazyModule` will be **very fast** and will return a cached instance, instead of loading the module again.

```
Load "LazyModule" attempt: 1
time: 2.379ms
Load "LazyModule" attempt: 2
time: 0.294ms
Load "LazyModule" attempt: 3
time: 0.303ms
```

Also, "lazy-loaded" modules share the same modules graph as those eagerly loaded on the application bootstrap as well as any other lazy modules registered later in your app.

Where `lazy.module.ts` is a TypeScript file that exports a **regular Nest module** (no extra changes are required).

The `LazyModuleLoader#load` method returns the module reference (of `LazyModule`) that lets you navigate the internal list of providers and obtain a reference to any provider using its injection token as a lookup key.

For example, let's say we have a `LazyModule` with the following definition:

```
@Module({
  providers: [LazyService],
  exports: [LazyService],
})
export class LazyModule {}
```

info **Hint** Lazy-loaded modules cannot be registered as **global modules** as it simply makes no sense (since they are registered lazily, on-demand when all the statically registered modules have been already instantiated). Likewise, registered **global enhancers** (guards/interceptors/etc.) **will not work** properly either.

With this, we could obtain a reference to the `LazyService` provider, as follows:

```
const { LazyModule } = await import('./lazy.module');
const moduleRef = await this.lazyModuleLoader.load(() => LazyModule);

const { LazyService } = await import('./lazy.service');
const lazyService = moduleRef.get(LazyService);
```

> warning **Warning** If you use **Webpack**, make sure to update your `tsconfig.json` file - setting `compilerOptions.module` to `"esnext"` and adding `compilerOptions.moduleResolution` property with `"node"` as a value:
>
> ```
> {
>   "compilerOptions": {
>     "module": "esnext",
>     "moduleResolution": "node",
>     ...
>   }
> }
> ```
>
> With these options set up, you'll be able to leverage the code splitting feature.

Lazy-loading controllers, gateways, and resolvers

Since controllers (or resolvers in GraphQL applications) in Nest represent sets of routes/paths/topics (or queries/mutations), you **cannot lazy load them** using the `LazyModuleLoader` class.

> error **Warning** Controllers, resolvers, and gateways registered inside lazy-loaded modules will not behave as expected. Similarly, you cannot register middleware functions (by implementing the `MiddlewareConsumer` interface) on-demand.

For example, let's say you're building a REST API (HTTP application) with a Fastify driver under the hood (using the `@nestjs/platform-fastify` package). Fastify does not let you register routes after the application is ready/successfully listening to messages. That means even if we analyzed route mappings registered in the module's controllers, all lazy-loaded routes wouldn't be accessible since there is no way to register them at runtime.

Likewise, some transport strategies we provide as part of the `@nestjs/microservices` package (including Kafka, gRPC, or RabbitMQ) require to subscribe/listen to specific topics/channels before the connection is established. Once your application starts listening to messages, the framework would not be able to subscribe/listen to new topics.

Lastly, the `@nestjs/graphql` package with the code first approach enabled automatically generates the GraphQL schema on-the-fly based on the metadata. That means, it requires all classes to be loaded beforehand. Otherwise, it would not be doable to create the appropriate, valid schema.

Common use-cases

Most commonly, you will see lazy loaded modules in situations when your worker/cron job/lambda & serverless function/webhook must trigger different services (different logic) based on the input arguments (route path/date/query parameters, etc.). On the other hand, lazy-loading modules may not make too much sense for monolithic applications, where the startup time is rather irrelevant.