HTTP module

Axios is rich-feature HTTP client that is widely used in dozens of applications. That's why Nest wraps this package and exposes it by default as a built-in `HttpModule`. The `HttpModule` exports `HttpService` that simply exposes axios-based methods to perform HTTP request, but also, transforms return types into `Observables`.

In order to use a `HttpService`, we need to import `HttpModule`.

```
@Module({
  imports: [HttpModule],
  providers: [CatsService],
})
export class CatsModule {}
```

HINT
The `HttpModule` is exposed from `@nestjs/common` package.

Then, you can inject `HttpService`. This class is easily accessible from `@nestjs/common` package.

JS

```
@Injectable()
export class CatsService {
  constructor(private readonly httpService: HttpService) {}

  findAll(): Observable<AxiosResponse<Cat[]>> {
    return this.httpService.get('http://localhost:3000/cats');
  }
}
```

All methods return `AxiosResponse` wrapped with `Observable` object.

Configuration

Axios gives a bunch of options that you may take advantage of to make your `HttpService` even more powerful. Read more about them here. To configure underlying library instance, use `register()` method of `HttpModule`.

```
@Module({
  imports: [
    HttpModule.register({
      timeout: 5000,
      maxRedirects: 5,
    }),
  ],
  providers: [CatsService],
})
export class CatsModule {}
```

All these properties will be passed down to the **axios** constructor.

Async configuration

Quite often you might want to asynchronously pass your module options instead of passing them beforehand. In such case, use `registerAsync()` method, that provides a couple of various ways to deal with async data.

First possible approach is to use a factory function:

```
HttpModule.registerAsync({
  useFactory: () => ({
    timeout: 5000,
    maxRedirects: 5,
  }),
});
```

Obviously, our factory behaves like every other one (might be `async` and is able to inject dependencies through `inject`).

```
HttpModule.registerAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    timeout: configService.getString('HTTP_TIMEOUT'),
    maxRedirects: configService.getString('HTTP_MAX_REDIRECTS'),
  }),
  inject: [ConfigService],
});
```

Alternatively, you are able to use class instead of a factory.

```
HttpModule.registerAsync({
  useClass: HttpConfigService,
});
```

Above construction will instantiate `HttpConfigService` inside `HttpModule` and will leverage it to create options object. The `HttpConfigService` has to implement `HttpModuleOptionsFactory` interface.

```
@Injectable()
class HttpConfigService implements HttpModuleOptionsFactory {
  createHttpOptions(): HttpModuleOptions {
    return {
      timeout: 5000,
      maxRedirects: 5,
    };
  }
}
```

In order to prevent the creation of `HttpConfigService` inside `HttpModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
HttpModule.registerAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

It works the same as `useClass` with one critical difference - `HttpModule` will lookup imported modules to reuse already created `ConfigService`, instead of instantiating it on its own.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more here.

Principal Sponsor

Sponsors / Partners

Become a sponsor