Caching

Cache is a great and simple **technique** that helps in improving your app's performance. It acts as a temporary data store, which accessing is really performant.

Installation

We need to install the required package at first:

```
$ npm install --save cache-manager
```

In-memory cache

Nest provides a unified API for various cache storage providers. The built-in one is an in-memory data store. However, you can easily switch to a more comprehensive solution, like Redis. In order to enable caching, firstly import `CacheModule` and call its `register()` method.

```
import { CacheModule, Module } from '@nestjs/common';
import { AppController } from './app.controller';

@Module({
  imports: [CacheModule.register()],
  controllers: [AppController],
})
export class ApplicationModule {}
```

Then just tie `CacheInterceptor` somewhere.

```
@Controller()
@UseInterceptors(CacheInterceptor)
export class AppController {
  @Get()
  findAll(): string[] {
    return [];
  }
}
```

WARNING
Only `GET` endpoints are cached.

Global cache

To reduce an amount of the required boilerplate, you can bind `CacheInterceptor` to each existing endpoint at once.

```
import { CacheModule, Module, CacheInterceptor } from '@nestjs/common';
import { AppController } from './app.controller';
import { APP_INTERCEPTOR } from '@nestjs/core';

@Module({
  imports: [CacheModule.register()],
  controllers: [AppController],
  providers: [
    {
      provide: APP_INTERCEPTOR,
      useClass: CacheInterceptor,
    },
  ],
})
export class ApplicationModule {}
```

WebSockets & Microservices

Obviously, you can effortlessly apply `CacheInterceptor` to WebSocket subscribers as well as Microservice's patterns (regardless of transport method that is being used).

JS

```
@CacheKey('events')
@UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client: Client, data: string[]): Observable<string[]> {
  return [];
}
```

HINT
The `@CacheKey()` decorator is imported from `@nestjs/common` package.

However, the additional `@CacheKey()` decorator is required in order to specify a key used to subsequently store and retrieve cached data. Besides, please note that you **shouldn't cache everything**. Actions which responsibility is to perform some business operations rather than simply querying the data should never be cached.

Customize caching

All cached data have its own expiration time (TTL). To customize default values, pass options object to the `register()` method.

```
CacheModule.register({
  ttl: 5, // seconds
  max: 10, // maximum number of items in cache
});
```

Different stores

We take advantage of **cache-manager** under the hood. This package supports a wide-range of useful stores, for example, **Redis** store (full list **here**). To set up the Redis store, simple pass the package together with corresponding options to the `register()` method.

```
import * as redisStore from 'cache-manager-redis-store';
import { CacheModule, Module } from '@nestjs/common';
import { AppController } from './app.controller';

@Module({
  imports: [
    CacheModule.register({
      store: redisStore,
      host: 'localhost',
      port: 6379,
    }),
  ],
  controllers: [AppController],
})
export class ApplicationModule {}
```

Adjust tracking

By default, Nest uses request URL (in HTTP app) or cache key (in websockets and microservices) set through `@CacheKey()` decorator to associate cache records with your endpoints. Nevertheless, sometimes you might want to set up tracking based on different factors, for example, using HTTP headers (e.g. `Authorization` to properly identify `profile` endpoints).

profile endpoints).

In order to accomplish that, create a subclass of `CacheInterceptor` and override `trackBy()` method.

```
@Injectable()
class HttpCacheInterceptor extends CacheInterceptor {
  trackBy(context: ExecutionContext): string | undefined {
    return 'key';
  }
}
```

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:

```
CacheModule.forRootAsync({
  useFactory: () => ({
    ttl: 5,
  }),
});
```

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

```
CacheModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    ttl: configService.getString('CACHE_TTL'),
  }),
  inject: [ConfigService],
});
```

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

```
CacheModule.forRootAsync({
  useClass: CacheConfigService,
});
```

Above construction will instantiate `CacheConfigService` inside `CacheModule` and will leverage it to create options object. The `CacheConfigService` has to implement `CacheOptionsFactory` interface.

```
@Injectable()
class CacheConfigService implements CacheOptionsFactory {
  createCacheOptions(): CacheModuleOptions {
    return {
      ttl: 5,
    };
  }
}
```

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

```
CacheModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

It works the same as `useClass` with one critical difference - `CacheModule` 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

Designed by Jakub Staroń, hosted by Netlify