Caching

Caching is a great and simple **technique** that helps improve your app's performance. It acts as a temporary data store providing high performance data access.

Installation

First install required packages:

```
$ npm install @nestjs/cache-manager cache-manager
```

> warning **Warning** `cache-manager` version 4 uses seconds for `TTL (Time-To-Live)`. The current version of `cache-manager` (v5) has switched to using milliseconds instead. NestJS doesn't convert the value, and simply forwards the ttl you provide to the library. In other words:
>
> - If using `cache-manager` v4, provide ttl in seconds
> - If using `cache-manager` v5, provide ttl in milliseconds
> - Documentation is referring to seconds, since NestJS was released targeting version 4 of 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, import the `CacheModule` and call its `register()` method.

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

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

Interacting with the Cache store

To interact with the cache manager instance, inject it to your class using the `CACHE_MANAGER` token, as follows:

```
constructor(@Inject(CACHE_MANAGER) private cacheManager: Cache) {}
```

> info **Hint** The `Cache` class is imported from the `cache-manager`, while `CACHE_MANAGER` token from the `@nestjs/cache-manager` package.

The `get` method on the `Cache` instance (from the `cache-manager` package) is used to retrieve items from the cache. If the item does not exist in the cache, `null` will be returned.

```
const value = await this.cacheManager.get('key');
```

To add an item to the cache, use the `set` method:

```
await this.cacheManager.set('key', 'value');
```

The default expiration time of the cache is 5 seconds.

You can manually specify a TTL (expiration time in seconds) for this specific key, as follows:

```
await this.cacheManager.set('key', 'value', 1000);
```

To disable expiration of the cache, set the `ttl` configuration property to `0`:

```
await this.cacheManager.set('key', 'value', 0);
```

To remove an item from the cache, use the `del` method:

```
await this.cacheManager.del('key');
```

To clear the entire cache, use the `reset` method:

```
await this.cacheManager.reset();
```

Auto-caching responses

> warning **Warning** In GraphQL applications, interceptors are executed separately for each field resolver. Thus, `CacheModule` (which uses interceptors to cache responses) will not work properly.

To enable auto-caching responses, just tie the `CacheInterceptor` where you want to cache data.

```
@Controller()
@UseInterceptors(CacheInterceptor)
```

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

> warning**Warning** Only `GET` endpoints are cached. Also, HTTP server routes that inject the native response object (`@Res()`) cannot use the Cache Interceptor. See response mapping for more details.

To reduce the amount of required boilerplate, you can bind `CacheInterceptor` to all endpoints globally:

```
import { Module } from '@nestjs/common';
import { CacheModule, CacheInterceptor } from '@nestjs/cache-manager';
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 AppModule {}
```

Customize caching

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

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

Use module globally

When you want to use `CacheModule` in other modules, you'll need to import it (as is standard with any Nest module). Alternatively, declare it as a global module by setting the options object's `isGlobal` property to `true`, as shown below. In that case, you will not need to import `CacheModule` in other modules once it's been loaded in the root module (e.g., `AppModule`).

```
CacheModule.register({
  isGlobal: true,
});
```

Global cache overrides

While global cache is enabled, cache entries are stored under a `CacheKey` that is auto-generated based on the route path. You may override certain cache settings (`@CacheKey()` and `@CacheTTL()`) on a per-method basis, allowing customized caching strategies for individual controller methods. This may be most relevant while using different cache stores.

```
@Controller()
export class AppController {
  @CacheKey('custom_key')
  @CacheTTL(20)
  findAll(): string[] {
    return [];
  }
}
```

> info **Hint** The `@CacheKey()` and `@CacheTTL()` decorators are imported from the `@nestjs/cache-manager` package.

The `@CacheKey()` decorator may be used with or without a corresponding `@CacheTTL()` decorator and vice versa. One may choose to override only the `@CacheKey()` or only the `@CacheTTL()`. Settings that are not overridden with a decorator will use the default values as registered globally (see Customize caching).

WebSockets and Microservices

You can also apply the `CacheInterceptor` to WebSocket subscribers as well as Microservice's patterns (regardless of the transport method that is being used).

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

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

Additionally, you may specify a cache expiration time (TTL) by using the `@CacheTTL()` decorator, which will override the global default TTL value.

```
@@filename()
@CacheTTL(10)
@UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client: Client, data: string[]): Observable<string[]> {
  return [];
}
@@switch
@CacheTTL(10)
@UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client, data) {
  return [];
}
```

> info **Hint** The `@CacheTTL()` decorator may be used with or without a corresponding `@CacheKey()` decorator.

Adjust tracking

By default, Nest uses the request URL (in an HTTP app) or cache key (in websockets and microservices apps, set through the `@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).

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

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

Different stores

This service takes advantage of cache-manager under the hood. The `cache-manager` package supports a wide-range of useful stores, for example, Redis store. A full list of supported stores is available here. To set

up the Redis store, simply pass the package together with corresponding options to the `register()` method.

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

@Module({
  imports: [
    CacheModule.register<RedisClientOptions>({
      store: redisStore,

      // Store-specific configuration:
      host: 'localhost',
      port: 6379,
    }),
  ],
  controllers: [AppController],
})
export class AppModule {}
```

> warning**Warning** `cache-manager-redis-store` does not support redis v4. In order for the `ClientOpts` interface to exist and work correctly you need to install the latest `redis` 3.x.x major release. See this issue to track the progress of this upgrade.

Async configuration

You may want to asynchronously pass in module options instead of passing them statically at compile time. In this case, use the `registerAsync()` method, which provides several ways to deal with async configuration.

One approach is to use a factory function:

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

Our factory behaves like all other asynchronous module factories (it can be `async` and is able to inject dependencies through `inject`).

```
CacheModule.registerAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
```

```
      ttl: configService.get('CACHE_TTL'),
    }),
    inject: [ConfigService],
  });
```

Alternatively, you can use the `useClass` method:

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

The above construction will instantiate `CacheConfigService` inside `CacheModule` and will use it to get the options object. The `CacheConfigService` has to implement the `CacheOptionsFactory` interface in order to provide the configuration options:

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

If you wish to use an existing configuration provider imported from a different module, use the `useExisting` syntax:

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

This works the same as `useClass` with one critical difference - `CacheModule` will lookup imported modules to reuse any already-created `ConfigService`, instead of instantiating its own.

> info **Hint** `CacheModule#register` and `CacheModule#registerAsync` and `CacheOptionsFactory` has an optional generic (type argument) to narrow down store-specific configuration options, making it type safe.

Example

A working example is available here.