## Rate Limiting

A common technique to protect applications from brute-force attacks is **rate-limiting**. To get started, you'll need to install the `@nestjs/throttler` package.

```
$ npm i --save @nestjs/throttler
```

Once the installation is complete, the `ThrottlerModule` can be configured as any other Nest package with `forRoot` or `forRootAsync` methods.

```
@@filename(app.module)
@Module({
  imports: [
    ThrottlerModule.forRoot({
      ttl: 60,
      limit: 10,
    }),
  ],
})
export class AppModule {}
```

The above will set the global options for the `ttl`, the time to live, and the `limit`, the maximum number of requests within the ttl, for the routes of your application that are guarded.

Once the module has been imported, you can then choose how you would like to bind the `ThrottlerGuard`. Any kind of binding as mentioned in the guards section is fine. If you wanted to bind the guard globally, for example, you could do so by adding this provider to any module:

```
{
  provide: APP_GUARD,
  useClass: ThrottlerGuard
}
```

#### Customization

There may be a time where you want to bind the guard to a controller or globally, but want to disable rate limiting for one or more of your endpoints. For that, you can use the `@SkipThrottle()` decorator, to negate the throttler for an entire class or a single route. The `@SkipThrottle()` decorator can also take in a boolean for if there is a case where you want to exclude *most* of a controller, but not every route.

```
@SkipThrottle()
@Controller('users')
export class UsersController {}
```

This `@SkipThrottle()` decorator can be used to skip a route or a class or to negate the skipping of a route in a class that is skipped.

```
@SkipThrottle()
@Controller('users')
export class UsersController {
  // Rate limiting is applied to this route.
  @SkipThrottle(false)
  dontSkip() {
    return "List users work with Rate limiting.";
  }
  // This route will skip rate limiting.
  doSkip() {
    return "List users work without Rate limiting.";
  }
}
```

There is also the `@Throttle()` decorator which can be used to override the `limit` and `ttl` set in the global module, to give tighter or looser security options. This decorator can be used on a class or a function as well. The order for this decorator does matter, as the arguments are in the order of `limit, ttl`. You have to configure it like this:

```
// Override default configuration for Rate limiting and duration.
@Throttle(3, 60)
@Get()
findAll() {
  return "List users works with custom rate limiting.";
}
```

### Proxies

If your application runs behind a proxy server, check the specific HTTP adapter options (express and fastify) for the `trust proxy` option and enable it. Doing so will allow you to get the original IP address from the `X-Forwarded-For` header, and you can override the `getTracker()` method to pull the value from the header rather than from `req.ip`. The following example works with both express and fastify:

```
// throttler-behind-proxy.guard.ts
import { ThrottlerGuard } from '@nestjs/throttler';
import { Injectable } from '@nestjs/common';

@Injectable()
export class ThrottlerBehindProxyGuard extends ThrottlerGuard {
  protected getTracker(req: Record<string, any>): string {
    return req.ips.length ? req.ips[0] : req.ip; // individualize IP
extraction to meet your own needs
  }
}

// app.controller.ts
import { ThrottlerBehindProxyGuard } from './throttler-behind-proxy.guard';

@UseGuards(ThrottlerBehindProxyGuard)
```

> info **Hint** You can find the API of the `req` Request object for express here and for fastify here.

**Websockets**

This module can work with websockets, but it requires some class extension. You can extend the `ThrottlerGuard` and override the `handleRequest` method like so:

```
@Injectable()
export class WsThrottlerGuard extends ThrottlerGuard {
  async handleRequest(context: ExecutionContext, limit: number, ttl: number): Promise<boolean> {
    const client = context.switchToWs().getClient();
    const ip = client._socket.remoteAddress
    const key = this.generateKey(context, ip);
    const { totalHits } = await this.storageService.increment(key, ttl);

    if (totalHits > limit) {
      throw new ThrottlerException();
    }

    return true;
  }
}
```

> info **Hint** If you are using ws, it is necessary to replace the `_socket` with `conn`

There's a few things to keep in mind when working with WebSockets:

- Guard cannot be registered with the `APP_GUARD` or `app.useGlobalGuards()`
- When a limit is reached, Nest will emit an `exception` event, so make sure there is a listener ready for this

> info **Hint** If you are using the `@nestjs/platform-ws` package you can use `client._socket.remoteAddress` instead.

**GraphQL**

The `ThrottlerGuard` can also be used to work with GraphQL requests. Again, the guard can be extended, but this time the `getRequestResponse` method will be overridden

```
@Injectable()
export class GqlThrottlerGuard extends ThrottlerGuard {
  getRequestResponse(context: ExecutionContext) {
    const gqlCtx = GqlExecutionContext.create(context);
    const ctx = gqlCtx.getContext();
    return { req: ctx.req, res: ctx.res };
  }
}
```

#### Configuration

The following options are valid for the `ThrottlerModule`:

| | |
|---|---|
| `ttl` | the number of seconds that each request will last in storage |
| `limit` | the maximum number of requests within the TTL limit |
| `ignoreUserAgents` | an array of regular expressions of user-agents to ignore when it comes to throttling requests |
| `storage` | the storage setting for how to keep track of the requests |

#### Async Configuration

You may want to get your rate-limiting configuration asynchronously instead of synchronously. You can use the `forRootAsync()` method, which allows for dependency injection and `async` methods.

One approach would be to use a factory function:

```
@Module({
  imports: [
    ThrottlerModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (config: ConfigService) => ({
        ttl: config.get('THROTTLE_TTL'),
        limit: config.get('THROTTLE_LIMIT'),
      }),
    }),
  ],
})
export class AppModule {}
```

You can also use the `useClass` syntax:

```
@Module({
  imports: [
    ThrottlerModule.forRootAsync({
      imports: [ConfigModule],
      useClass: ThrottlerConfigService,
    }),
  ],
})
export class AppModule {}
```

This is doable, as long as `ThrottlerConfigService` implements the interface `ThrottlerOptionsFactory`.

#### Storages

The built in storage is an in memory cache that keeps track of the requests made until they have passed the TTL set by the global options. You can drop in your own storage option to the `storage` option of the `ThrottlerModule` so long as the class implements the `ThrottlerStorage` interface.

For distributed servers you could use the community storage provider for Redis to have a single source of truth.

> info **Note** `ThrottlerStorage` can be imported from `@nestjs/throttler`.