Serialization

Serialization is a process that happens before objects are returned in a network response. This is an appropriate place to provide rules for transforming and sanitizing the data to be returned to the client. For example, sensitive data like passwords should always be excluded from the response. Or, certain properties might require additional transformation, such as sending only a subset of properties of an entity. Performing these transformations manually can be tedious and error prone, and can leave you uncertain that all cases have been covered.

Overview

Nest provides a built-in capability to help ensure that these operations can be performed in a straightforward way. The `ClassSerializerInterceptor` interceptor uses the powerful class-transformer package to provide a declarative and extensible way of transforming objects. The basic operation it performs is to take the value returned by a method handler and apply the `instanceToPlain()` function from class-transformer. In doing so, it can apply rules expressed by `class-transformer` decorators on an entity/DTO class, as described below.

> info **Hint** The serialization does not apply to StreamableFile responses.

Exclude properties

Let's assume that we want to automatically exclude a `password` property from a user entity. We annotate the entity as follows:

```
import { Exclude } from 'class-transformer';

export class UserEntity {
  id: number;
  firstName: string;
  lastName: string;

  @Exclude()
  password: string;

  constructor(partial: Partial<UserEntity>) {
    Object.assign(this, partial);
  }
}
```

Now consider a controller with a method handler that returns an instance of this class.

```
@UseInterceptors(ClassSerializerInterceptor)
@Get()
findOne(): UserEntity {
  return new UserEntity({
    id: 1,
```

```
    firstName: 'Kamil',
    lastName: 'Mysliwiec',
    password: 'password',
  });
}
```

> **Warning** Note that we must return an instance of the class. If you return a plain JavaScript object, for example, `{{ '{' }} user: new UserEntity() {{ '}' }}`, the object won't be properly serialized.

> info **Hint** The `ClassSerializerInterceptor` is imported from `@nestjs/common`.

When this endpoint is requested, the client receives the following response:

```
{
  "id": 1,
  "firstName": "Kamil",
  "lastName": "Mysliwiec"
}
```

Note that the interceptor can be applied application-wide (as covered here). The combination of the interceptor and the entity class declaration ensures that **any** method that returns a `UserEntity` will be sure to remove the `password` property. This gives you a measure of centralized enforcement of this business rule.

Expose properties

You can use the `@Expose()` decorator to provide alias names for properties, or to execute a function to calculate a property value (analogous to **getter** functions), as shown below.

```
@Expose()
get fullName(): string {
  return `${this.firstName} ${this.lastName}`;
}
```

Transform

You can perform additional data transformation using the `@Transform()` decorator. For example, the following construct returns the name property of the `RoleEntity` instead of returning the whole object.

```
@Transform(({ value }) => value.name)
role: RoleEntity;
```

Pass options

You may want to modify the default behavior of the transformation functions. To override default settings, pass them in an `options` object with the `@SerializeOptions()` decorator.

```
@SerializeOptions({
  excludePrefixes: ['_'],
})
@Get()
findOne(): UserEntity {
  return new UserEntity();
}
```

> info **Hint** The `@SerializeOptions()` decorator is imported from `@nestjs/common`.

Options passed via `@SerializeOptions()` are passed as the second argument of the underlying `instanceToPlain()` function. In this example, we are automatically excluding all properties that begin with the _ prefix.

Example

A working example is available here.

WebSockets and Microservices

While this chapter shows examples using HTTP style applications (e.g., Express or Fastify), the `ClassSerializerInterceptor` works the same for WebSockets and Microservices, regardless of the transport method that is used.

Learn more

Read more about available decorators and options as provided by the `class-transformer` package here.