NESTJS SESSIONS

The **ultimate** guide to implementing sessions in NestJs

by Vladimir Agaev

Thank you for downloading this eBook!
My name is Vladimir, founder of **Code with Vlad**. I make complicated programming subjects easy to understand.,

You can find me on my youtube channel, or on my website https://www.codewithvlad.com

If you have Twitter, say hi @vladimir_agaev

Contents

Introduction

When I was an aspiring developer, I had a hard time with sessions and authentications. It took me a lot of trial and error to get it right.

As I grew up as a developer, I noticed that this subject was particularly complicated for a lot of people. Lately, a lot of my viewers have requested more guidance on sessions with NestJs.

Because NestJs documentation is quite light on the subject, I decided to do something about it.
This is how this ebook was born :)

In this ebook, I will guide you through the process of adding session-based authentication to your NestJs application, **the right way**.

We will start with a bit of theory so everyone is on the same page, then we'll jump into a practical example.

What we are going to build won't be out of the ordinary (it's something that is quite hard to do in an eBook anyway) but I will give you the necessary knowledge to use sessions with confidence.

Together we will create a very simple **REST API using session-based authentication in NestJs.**
As a bonus, we will also see how to prepare our API for scale using **Redis** :)

Why do we need sessions anyway?

Before we jump into the code, let's have a look at the big picture!
If I've learned one thing in my career, it is that overlooking the fundamentals and the "why of things" often leads to trouble.

So, why do we need sessions?
We need them because of how the Hypertext Transfer Protocol (HTTP) works.

There are a lot of protocols out there, one of the most known is the HTTP protocol. It is used for web traffic, in other words, it can be used for getting data from the server, or sending data to the server. You use HTTP every day when you browse the web.

It allows you to view content like images and text but also send that content to the web server.

However, there is a hitch. The HTTP protocol is stateless. There is no shared memory, or state, between requests. In other words, the server does not remember you.

If I were to illustrate it with an analogy, It would be something like that:

- **Me:** Hi!
- **Server:** Hi, what's your name?
- **Me:** My, name is Vlad
- **Server:** Hi Vlad!
- **Me:** Hi!
- **Server:** Hi, what's your name?
- ...

And that's a problem!
Websites need to know who their users are **throughout the user journey**.

Sometimes they need to know it for tracking purposes like analytics, but very often applications can't function without the ability to remember users <u>between requests</u>.

Imagine if Facebook or Instagram forced you to enter your password every time you want to click on an image or a link.

Or if Amazon could not remember what you put in the basket?

Useless!

We needed something to solve this problem and make the server remember its users.
Hence, the sessions were born!

What are sessions

Because HTTP is stateless, in order to associate a request to any other request, we need a way to **store user data between requests**.

It can't be done in the browser since the user could hijack that data. So it is stored on the server.

Sessions are mostly JSON objects that are stored **on the server**. They can contain all kinds of user information, but very often they will store **an identifier** that links the user to its data in a relational or NoSQL database.

One of the most common use cases for sessions is authentication, this is the use case that we are going to learn in this book. **Session authentication**!

During authentication, when the user signs in, a session with his personal data is created on the server. This session itself (JSON object) is linked to an id that is sent back to the user, **the session-id**. (This is different from the identifier that links the user to a database though)

Bear in mind that a session is just a technology. A tool. And as a tool, it can have different purposes.

One of the purposes can be tracking (like in the case of Google analytics). Google Analytics doesn't really authenticate users. But when a new user comes in, a session is created.
When that same user returns to the website, Google Analytics knows that it's a returning user, because of the **session-id saved in the browser**.

However, most of the time sessions are used for authentication and authorization in web applications.

We refer to **session-based authentication or session authentication** when sessions are used to authenticate users. The other alternative is token-based authentication used by JSON web tokens or JWT, but we are not going to cover it in this book.

Session-based authentication is everywhere! It is the most common way to log users into a website. All tech giants use them, and so will we.
We need to explain another concept before jumping into the code: Authentication, and its counterpart: Authorization

Authentication vs Authorization

Developers use those concepts interchangeably but it’s rather incorrect. What they usually mean is that users can access the website and view private information only if they have created an account.

But authentication and authorization refer to different processes at different times. Authentication comes first!

Authentication is the process that verifies user identity. When signing in, we often want to verify that the provided email and password match our authentication policy.

Is the password secure enough?
Does the user email provided exist in our database?
Do passwords match?

Only then, can we sign the user in by creating a session on the server and send the session-id back to the client. This session-id will be stored in the browser and then automatically re-sent upon every subsequent request.

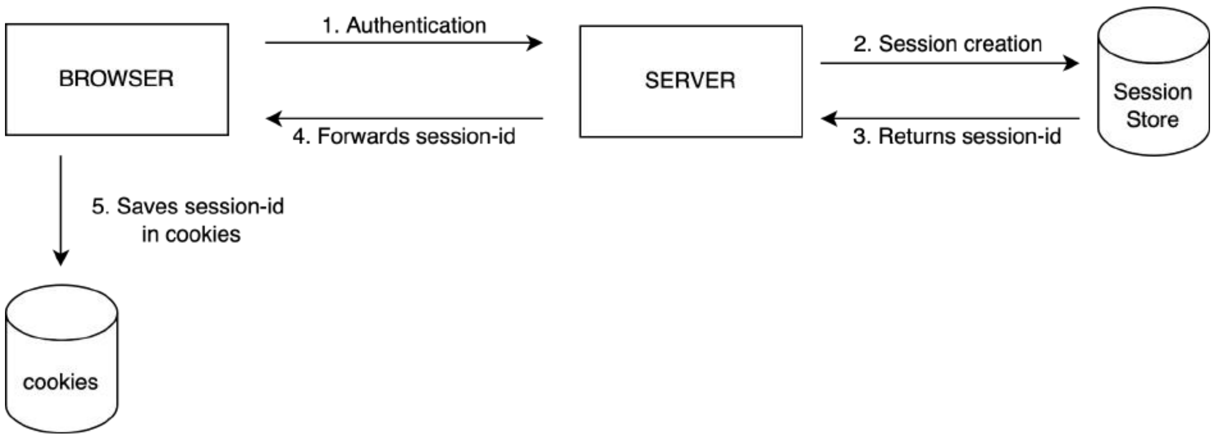
Authorization on the other hand is the process that allows or denies the already authenticated user to perform certain actions. Viewing other users’ personal information for example, or downloading a file.

The confusion between authentication and authorization usually comes from the fact that a lot of apps don’t necessarily separate authentication from authorization. If you create a simple API and don’t restrict the endpoints that users can hit based on arbitrary factors, you are basically adopting a system with no authorization.

Or to put it in other words, a system where everything is authorised.

So what does it all mean in practice regarding sessions?

Take a look at the schema below 👇



Once the user has authenticated. His data will be saved in a session as a JSON object and **a session-id** linking to this JSON object will be generated by the server.

The user then receives this generated **session-id**.

Every time the user does something on the website, he sends the session-id back to the server. This way the server knows who the user is as long **as the session has not been destroyed by the server** (which happens if the user logs out, or is inactive for a long time).

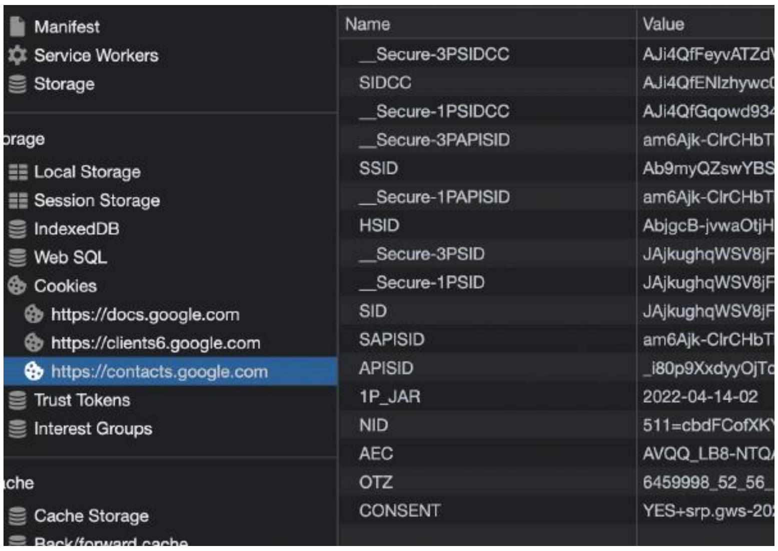
Thanks to the session-id, the server knows which user is making the request. It will then authorise the request if the session exists in the session store.

This protection is performed by the authorization in place. This authorization can be as simple as "the user should be signed in". Or as complex as "the user should be a subscriber of the pro+ plan to access this feature".

So, in a nutshell, authorization gives access or denies access based on the results of the authentication.

Oh, and a very important point about the session-id.
It's stored in a special place in the user's browser memory.
It's called a cookie.

No, not these cookies.
Those cookies!



Name	Value
__Secure-3PSIDCC	AJi4QfFeyvATZd
SIDCC	AJi4QfENIzhywc
__Secure-1PSIDCC	AJi4QfGqowd93
__Secure-3PAPISID	am6Ajk-ClrCHbT
SSID	Ab9myQZswYBS
__Secure-1PAPISID	am6Ajk-ClrCHbT
HSID	AbjgcB-jvwaOtjH
__Secure-3PSID	JAjkughqWSV8jF
__Secure-1PSID	JAjkughqWSV8jF
SID	JAjkughqWSV8jF
SAPISID	am6Ajk-ClrCHbT
APISID	_i80p9XxdyyOjTc
1P_JAR	2022-04-14-02
NID	511=cbdFCofXK
AEC	AVQQ_LB8-NTQ
OTZ	6459998_52_56_
CONSENT	YES+srp.gws-20

Since I've introduced cookies to the mix, let's briefly go through cookies vs sessions as well! I promise it won't be long :)

Cookies vs Sessions

Those two concepts are also often used together but again, they refer to different things. A session is a backend technology. It's the information that the server stores related to a user. (linked by the session-id)

A cookie on the other hand is a frontend technology, implemented by your browser. Cookies can store various information, not only session-ids.

So, a session is like an intermediary backend database storing user information **based on a session-id**. A cookie is like a frontend database that stores information based on the **website domain name**.

Cookies are great for authentication because:

- They can store a session id securely, preventing javascript from accessing it and stealing it (if used as an HTTP-only cookie)
- They are segmented by the domain name. The website www.facebook.com won't be able to access the cookies from www.amazon.com and vice versa.
- The session-id can be automatically sent with every request to the server, making the authorization straightforward. You don't need to pass the cookies to the headers like with JWT-based authorization, where you pass the access token as a Bearer token.

Congratulations if you've read so far!
You know everything you need to know to start coding.

I know you've been waiting for this moment so let's jump straight into the code..

Adding sessions to NestJs

I will try my best to make this tutorial easy to follow, but if you feel that you're stuck somewhere here is the GitHub repository to the completed project:

Create a new project

Let's start by creating a new project with nest. Make sure that you have the following dependencies installed:

- Node v16

To use nest, it's better to install the command-line interface as a global npm package. You can do it the following way:

```
npm i -g @nestjs/cli
nest new nestjs-sessions-example
```

Clean unused files

Let's also delete the files that we don't need (in red). Our application will be very simple so we won't need the *app.service.ts* and the *app.controller.spec.ts*.

```
├── README.md
├── nest-cli.json
├── package-lock.json
├── package.json
├── src
│   ├── app.controller.spec.ts
│   ├── app.controller.ts
│   ├── app.module.ts
│   ├── app.service.ts
│   └── main.ts
├── test
│   ├── app.e2e-spec.ts
│   └── jest-e2e.json
├── tsconfig.build.json
└── tsconfig.json
```

After the deletion, this is what you should have.

```
├── README.md
├── nest-cli.json
├── package-lock.json
├── package.json
├── src
│   ├── app.controller.ts
│   ├── app.module.ts
│   └── main.ts
├── test
│   ├── app.e2e-spec.ts
│   └── jest-e2e.json
├── tsconfig.build.json
└── tsconfig.json
```

Also, **don't forget to delete the AppService class import from the app.module.ts**, otherwise NestJs will complain 🙁

Install session dependency

Since nest is using express.js under the hood, we can enable session support using middlewares. There is a library called express-session that we can use.

```
npm i express-session
npm i -D @types/express-session
```

express-session is an express js middleware. We need to inject it into our nest application with app.use(). We'll use the default options for now.

main.js

```
import { NestFactory } from '@nestjs/core';
import * as session from 'express-session';

import { AppModule } from './app.module';

async function bootstrap() {
 const app = await NestFactory.create(AppModule);

 // add sessions middleware to NestJs
 app.use(
   session({
     secret: 'super-secret',
     resave: false,
     saveUninitialized: false,
   }),
 );

 // change port to 3333 for convenience
 await app.listen(3333);
}
bootstrap();
```

That’s it, your application is now configured to work with sessions.

Bear in mind that if you are using sessions in production the **secret** property should be **a randomly generated number** that you don’t share with anyone as it’s used to sign sessions and generate session-ids.
So super secret!

Also, in production you would use a session store like Redis, otherwise your app will have a **memory leak**. But more on that later…

Add session logic to AppController

Now that we have configured the app to work with sessions, we can write the logic that will allow us to:

- Create sessions
- Delete sessions
- Read from sessions

All that can be done in the AppController class!

app.controller.ts

```
import { Controller, Get, Post } from '@nestjs/common';

@Controller()
export class AppController {
 @Get('me')
 getMe() {}

 @Post('auth')
 auth() {}

 @Post('logout')
 logout() {}
}
```

The routes are:

- GET /me: to get current user from session
- POST /auth: authenticate the user and create a session
- POST /logout: to delete user session

This is in my opinion, the minimum example that you need to work with sessions. No need to overcomplicate things.

If you feel like you need more help ping me on my youtube channel and I'll create more content on the subject :)

Let's now add the session decorator to our code!

app.controller.ts

```
import { Controller, Get, Post, Session as GetSession } from '@nestjs/common';

@Controller()
export class AppController {
 @Get('me')
 getMe(@GetSession() session) {}

 @Post('auth')
 auth(@GetSession() session) {}

 @Post('logout')
 logout(@GetSession() session) {}
}
```

You might wonder why I renamed the @Session decorator to @GetSession.
I've renamed it because the session decorator is conflicting with the session type imported from the express package (which we are going to import in a second).

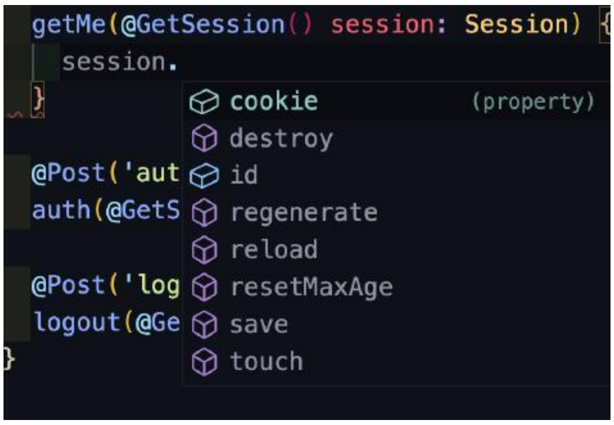
Right now, the session variable has a type of "any". Which is not very typescript friendly. To make it more typescript friendly let's give it a type.

app.controller.ts

```
import { Controller, Get, Post, Session as GetSession } from '@nestjs/common';
import { Session } from 'express-session';

@Controller()
export class AppController {
 @Get('me')
 getMe(@GetSession() session: Session) {}

 @Post('auth')
 auth(@GetSession() session: Session) {}

 @Post('logout')
 logout(@GetSession() session: Session) {}
}
```

Note that "Session" type comes from express-session :)

This way, when we use the session object, we get some help from typescript.



```
getMe(@GetSession() session: Session) {
  session.
}

@Post('aut
auth(@GetS

@Post('log
logout(@Ge
}
```

Magic! 💥

This is great! But we need to account for one more hiccup. Usually, the convention is to save user data on the session object, as a property called “user”.

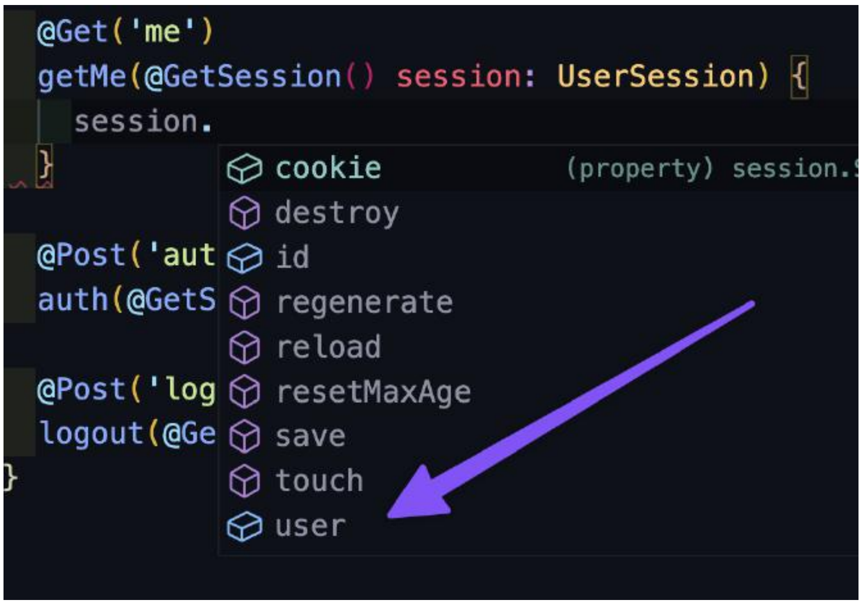
So “**session.user**” should be the JSON object storing user information. But we don’t see it in the picture above. Let’s add it to our Session type!

Typescript has a very handy way of representing objects with Record<key, value>, where key and value are types or values of the object’s key and value.

app.controller.ts

```
import { Controller, Get, Post, Session as GetSession } from '@nestjs/common';
import { Session } from 'express-session';

type UserSession = Session & Record<'user', any>;

@Controller()
export class AppController {
 @Get('me')
 getMe(@GetSession() session: UserSession) {}

 @Post('auth')
 auth(@GetSession() session: UserSession) {}

 @Post('logout')
 logout(@GetSession() session: UserSession) {}
}
```

Note that ‘user’ is in quotes. It’s a **value** and not a type!



```
@Get('me')
getMe(@GetSession() session: UserSession) {
  session.
}
              cookie          (property) session.S
              destroy
@Post('aut    id
auth(@GetS    regenerate
              reload
@Post('log    resetMaxAge
logout(@Ge    save
}             touch
              user
```

This is more like it!
We have created a UserSession type that will be a union between the Session type (imported from express-session) and an object that is named "user".

Now we have an object called user **that we can write into**! Any operation to that object will be saved into the session store.

Saving data to Sessions

Add POST /auth

We know that nest is an abstraction layer on top of express.js. So the @GetSession decorator actually gets the session property from the request object (created by express).

When we will call POST /auth, to authenticate the user, we will be able to add properties to the user object in the session. That will automatically update the session store by creating a session.

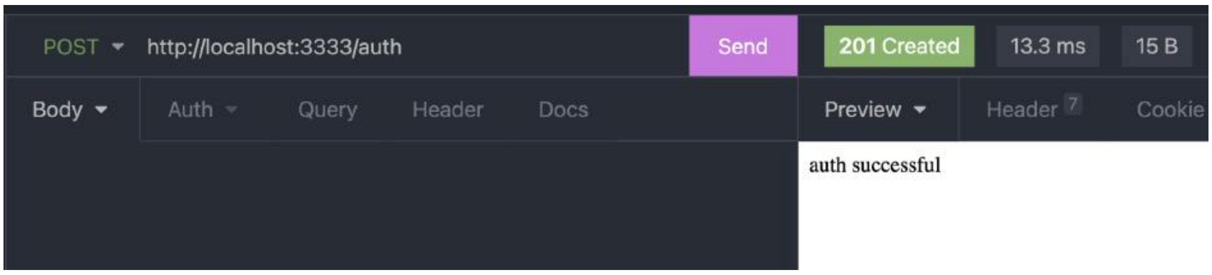
Let’s add that logic and jump into insomnia to test our API!

```
import { Controller, Get, Post, Session as GetSession } from '@nestjs/common';
import { Session } from 'express-session';

type UserSession = Session & Record<'user', any>;

@Controller()
export class AppController {
 @Get('me')
 getMe(@GetSession() session: UserSession) {}

 @Post('auth')
 auth(@GetSession() session: UserSession) {
   session.user = {
     email: 'vlad@codewithvlad.com',
   };
   return 'auth successful';
 }

 @Post('logout')
 logout(@GetSession() session: UserSession) {}
}
```

Don’t forget to start the server with

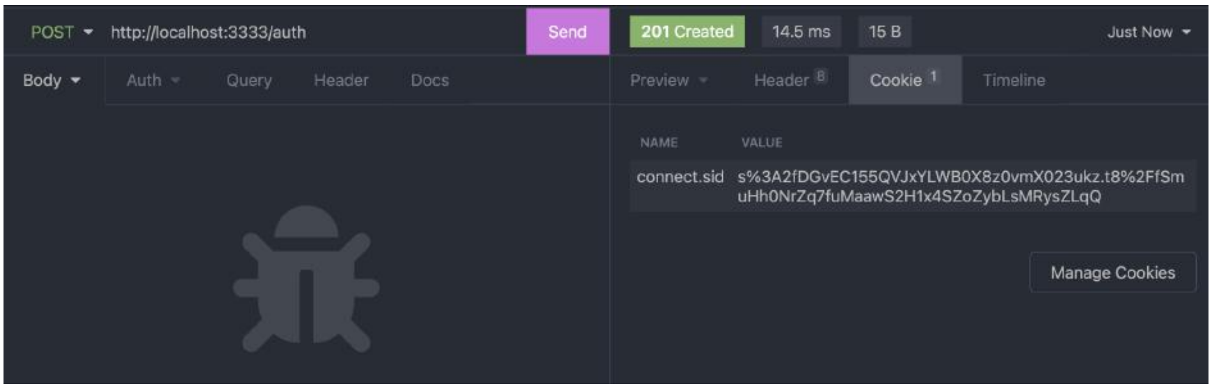
```
npm run start:dev
```

The server should be running on port 3333. The request returned ‘auth successful’ as we’ve programmed it above.



```
auth successful
```

If you now click on the cookie tab, you will see that Nest sent us back the session-id **as a cookie**. Insomnia automatically saves cookies and resends them on subsequent requests.



NAME	VALUE
connect.sid	s%3A2fDGvEC155QVJxYLWB0X8z0vmX023ukz.t8%2FfSmuHh0NrZq7fuMaawS2H1x4SZoZybLsMRysZLqQ

Add GET /me

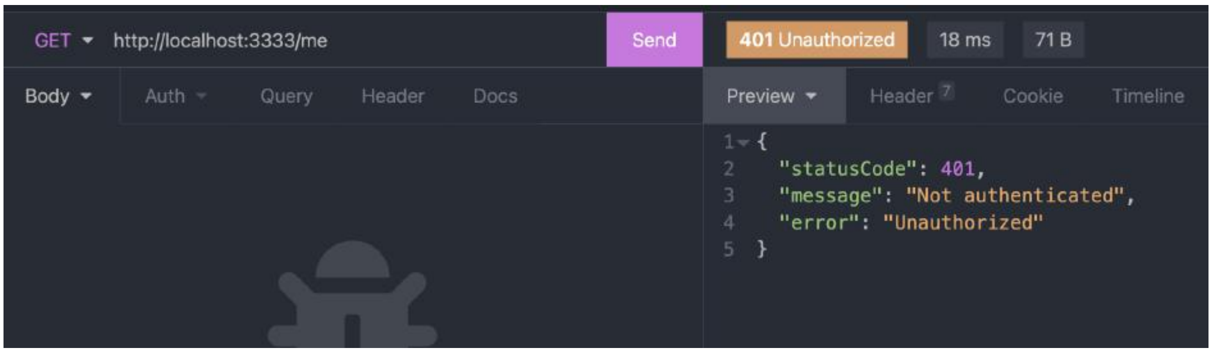
Let's now turn GET /me into a private route.

app.controller.ts

```
import {
 Controller,
 Get,
 Post,
 Session as GetSession,
 UnauthorizedException,
} from '@nestjs/common';
import { Session } from 'express-session';

type UserSession = Session & Record<'user', any>;

@Controller()
export class AppController {
 @Get('me')
 getMe(@GetSession() session: UserSession) {
   if (!session.user) throw new UnauthorizedException('Not authenticated');
   return session.user;
 }

 @Post('auth')
 auth(@GetSession() session: UserSession) {
   session.user = {
     email: 'vlad@codewithvlad.com',
   };
   return 'auth successful';
 }

 @Post('logout')
 logout(@GetSession() session: UserSession) {}
}
```

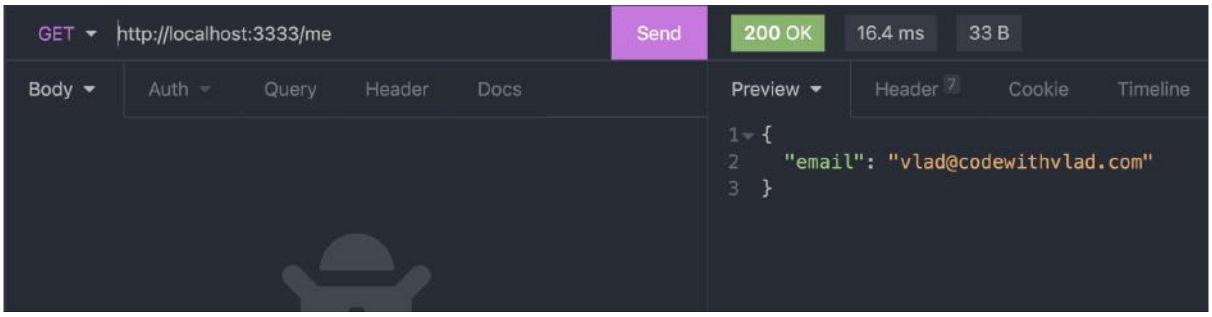
All that code does is that it throws a 401 Error if the session object does not have a user object. In other words, it throws an error if the user is not authenticated. And if the user is authenticated, it will return the user object.

Let’s try this out!



```
{
  "statusCode": 401,
  "message": "Not authenticated",
  "error": "Unauthorized"
}
```

Now let’s try it out after we hit POST /auth:



```
{
  "email": "vlad@codewithvlad.com"
}
```

Perfect! We have in fact created some **authorization** here. Very basic because it requires the user to be authenticated, but authorization nevertheless.

Add POST /logout

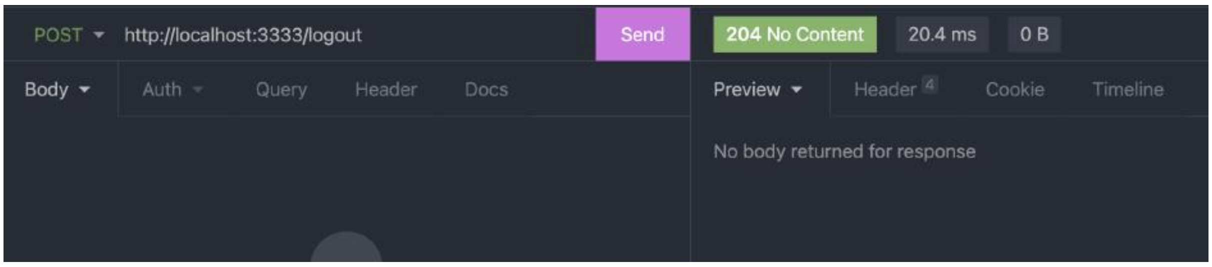
Let’s add the “logging out” logic now!

```
import {
 Controller,
 Get,
 HttpCode,
 HttpStatus,
 Post,
 Session as GetSession,
 UnauthorizedException,
} from '@nestjs/common';
import { Session } from 'express-session';

type UserSession = Session & Record<'user', any>;

@Controller()
export class AppController {
 @Get('me')
 getMe(@GetSession() session: UserSession) {
   if (!session.user) throw new UnauthorizedException('Not authenticated');
   return session.user;
 }

 @Post('auth')
 auth(@GetSession() session: UserSession) {
   session.user = {
     email: 'vlad@codewithvlad.com',
   };
   return 'auth successful';
 }

 @HttpCode(HttpStatus.NO_CONTENT)
 @Post('logout')
 logout(@GetSession() session: UserSession) {
   return new Promise((resolve, reject) => {
     session.destroy((err) => {
       if (err) reject(err);
       resolve(undefined);
     });
   });
 }
}
```

Note the `@HttpCode(HttpStatus.NO_CONTENT)` this is because we want the logout route to return a 204 status code. Which is usually the standard when we deal with the “logout” route.

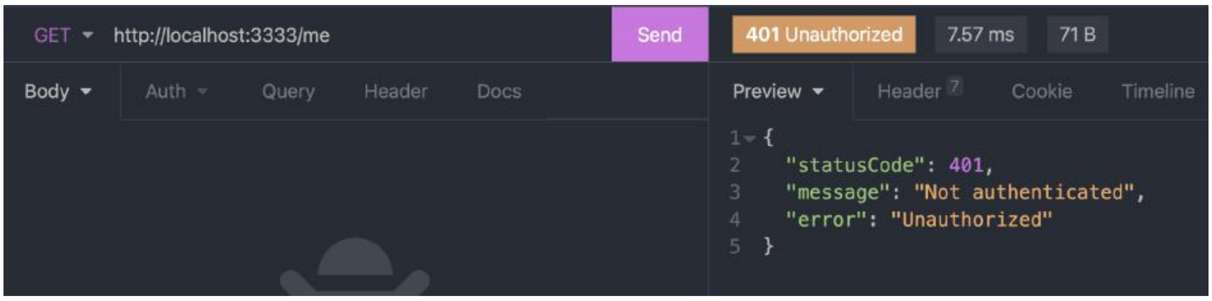
The deletion of the session is an asynchronous operation, it needs a callback function and does not support promises. We wrap that callback function into a promise. This promise will return “undefined” when the deletion of the session is complete (204 means no content, thus undefined is a good value to return imo.)

If we test it, the session should be deleted.



```
No body returned for response
```

Since the session is deleted a 401 is returned when we try to get user information.



```
{
  "statusCode": 401,
  "message": "Not authenticated",
  "error": "Unauthorized"
}
```

Scaling session store with Redis

Why Redis

Up until now, our session storage (the actual place where the key values are stored) was the **server's memory, server's RAM**. It is called MemoryStore. This is okay for testing purposes and even small apps, but is a problem if you run a production server.

Storing sessions in memory is a bad idea because:

- If your server restarts, all sessions will be deleted as they only live in the server's memory (all the users will be logged out)
- You won't be able to scale your app with more web servers because web servers won't be able to **share session data**
- Your server can run out of memory and crash if there are too many sessions created. (all the users will be logged out)
- No analytics on session usage.
- No ability to delete sessions manually, to log out a specific user manually.

Redis is a popular in-memory database that is **very fast** and can be used for a lot of things, like caching.
Sessions are in fact a particular use-case of caching because data is saved as a JSON object and for a specific time (session max-age).

Let’s set up our application to use sessions! For that, you will have to install docker, I recommend installing “docker for desktop”. Which should come along with docker-compose, a tool to spawn and manage docker containers.

Add docker-compose

Once you have installed docker and docker-compose, we can create the docker-compose.yml file that will define which containers we want to spawn and how.
In the root directory, create a docker-compose.yml file.

```
.
├── README.md
├── dist
├── docker-compose.yml
├── nest-cli.json
├── package-lock.json
├── package.json
├── src
├── test
├── tsconfig.build.json
└── tsconfig.json
```

docker-compose.yml

```yaml
version: '3.8'
services:
 redis:
   image: redis:6.0
   ports:
     - 6379:6379
 redis-commander:
   container_name: redis-commander
   hostname: redis-commander
   image: ghcr.io/joeferner/redis-commander:latest
   environment:
     - REDIS_HOSTS=local:redis:6379
   ports:
     - "8081:8081"
```

We won’t go too much into details but this docker-compose file spawns two containers. A Redis container and a redis-commander container.

The Redis container will use Redis version 6, and will be exposed on ports 6379 (standard ports for Redis).
The Redis commander is a web client for Redis. It is useful to inspect Redis (a bit like pg admin) and can be accessible in the browser on http://localhost:8081

After the docker-compose file is created, you can start the containers with:

```
docker compose up -d
```

That will start our two containers in the background. That's it, Redis is ready to use!

Add Redis to NestJs

Our Redis database is live at **redis://localhost:6379**.
To access it we need a Redis client, let's go with "**ioredis**" which is a modern and performant Redis client. We also need a Redis adapter for the express-session package, called **connect-redis**. The adapter will know how to plug express-sessions into Redis through the ioredis client.

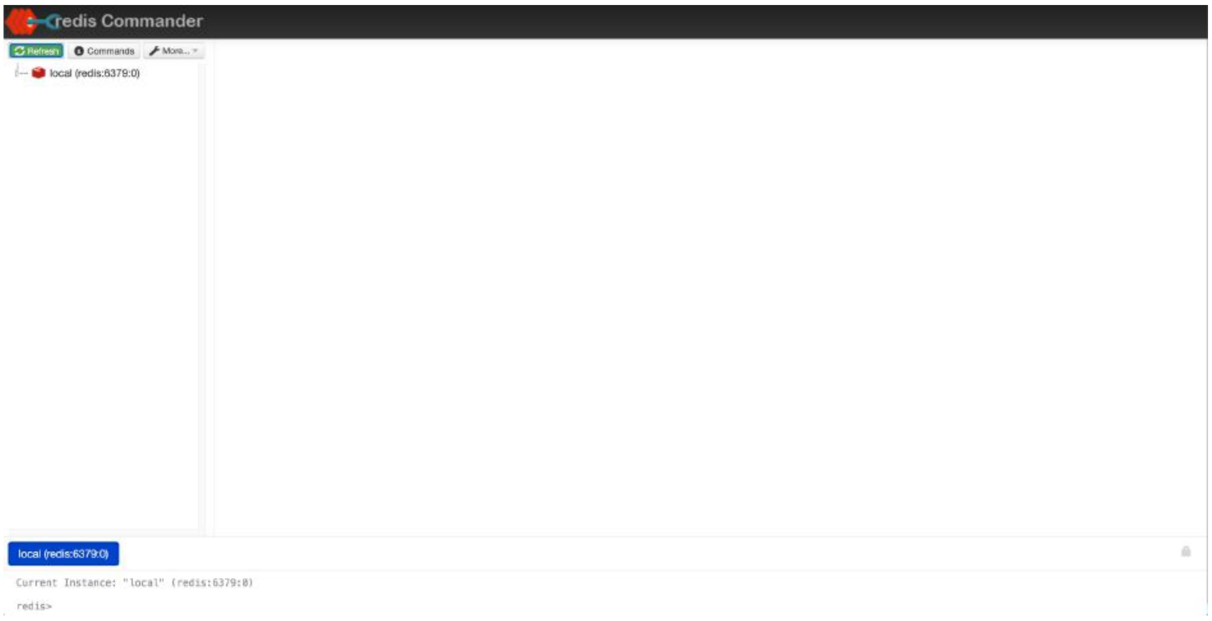
```
npm i ioredis connect-redis
```

```
import { NestFactory } from '@nestjs/core';
import * as session from 'express-session';
import * as connectRedis from 'connect-redis'; // new code
import IoRedis from 'ioredis'; // new code

import { AppModule } from './app.module';

// new code
const RedisStore = connectRedis(session);
// new code
const redisClient = new IoRedis('redis://localhost:6379');

async function bootstrap() {
 const app = await NestFactory.create(AppModule);

 // we add sessions middleware
 app.use(
   session({
     // new code
     store: new RedisStore({ client: redisClient }),
     secret: 'super-secret',
     resave: false,
     saveUninitialized: false,
   }),
 );

 await app.listen(3333);
}
bootstrap();
```

I find that the process of connecting Redis to the session middleware a bit clunky, especially with this pattern `const RedisStore = connectRedis(session)` but it is what it is.

We are initialising an ioredis client and connecting it to the session store via connect-redis library.

If you restart your nest server, your session store should be now connected to Redis and the Redis commander should be accessible on http://localhost:8081

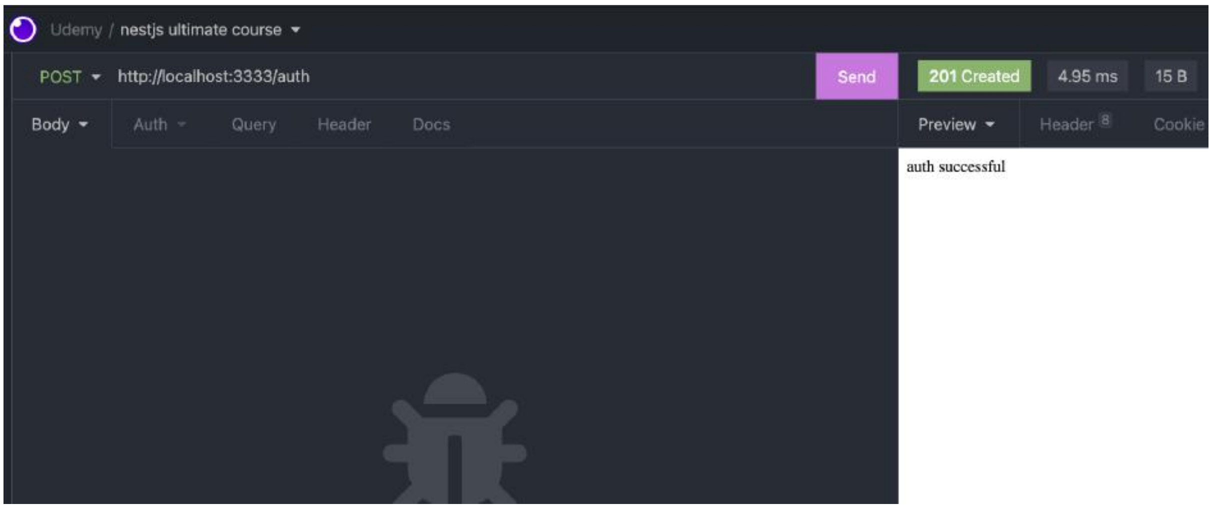


```
Current Instance: "local" (redis:6379:0)
redis>
```

Save sessions to Redis

By connecting our session store to Redis we tell NestJs to not store the session objects inside the server's memory but inside the Redis database. In production, Redis will have its own server, completely independent of your NodeJs application.

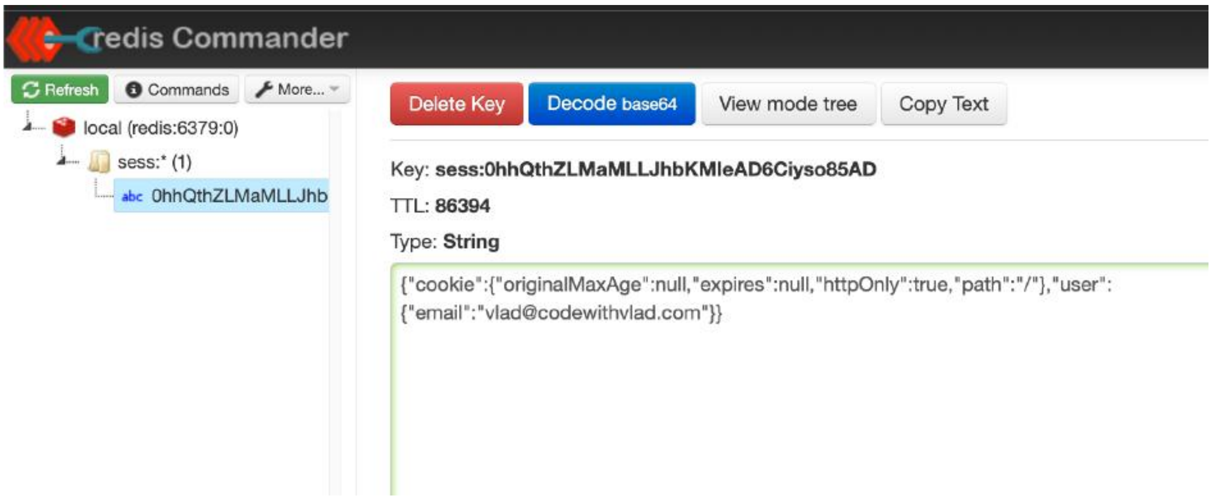
And if we trigger the POST /auth endpoint again



```
POST http://localhost:3333/auth
```

auth successful

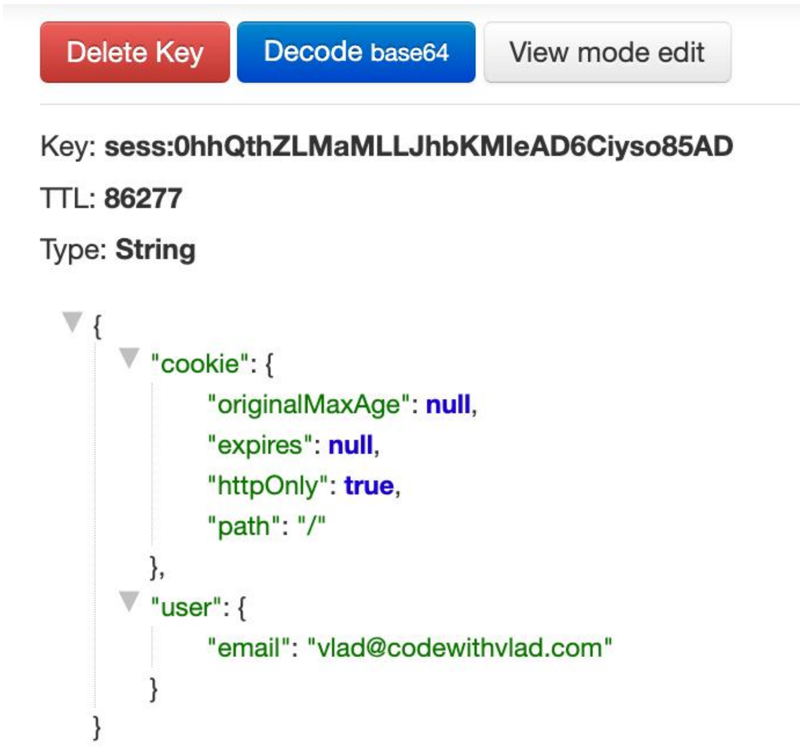
We will now be able to observe how sessions are being populated in Redis through the redis-commander.
You can see here the time to live (TTL) in seconds slowly decreasing. Once the TTL drops to 0, the session is deleted. Effectively logging the user out of the application.



Key: **sess:0hhQthZLMaMLLJhbKMleAD6Ciyso85AD**
TTL: **86394**
Type: **String**

```
{"cookie":{"originalMaxAge":null,"expires":null,"httpOnly":true,"path":"/"},"user":
{"email":"vlad@codewithvlad.com"}}
```

You can also click on “view mode tree” to get a more JSON friendly rendering of the data saved in the session.



Key: **sess:0hhQthZLMaMLLJhbKMleAD6Ciyso85AD**

TTL: **86277**

Type: **String**

```
{
    "cookie": {
        "originalMaxAge": null,
        "expires": null,
        "httpOnly": true,
        "path": "/"
    },
    "user": {
        "email": "vlad@codewithvlad.com"
    }
}
```

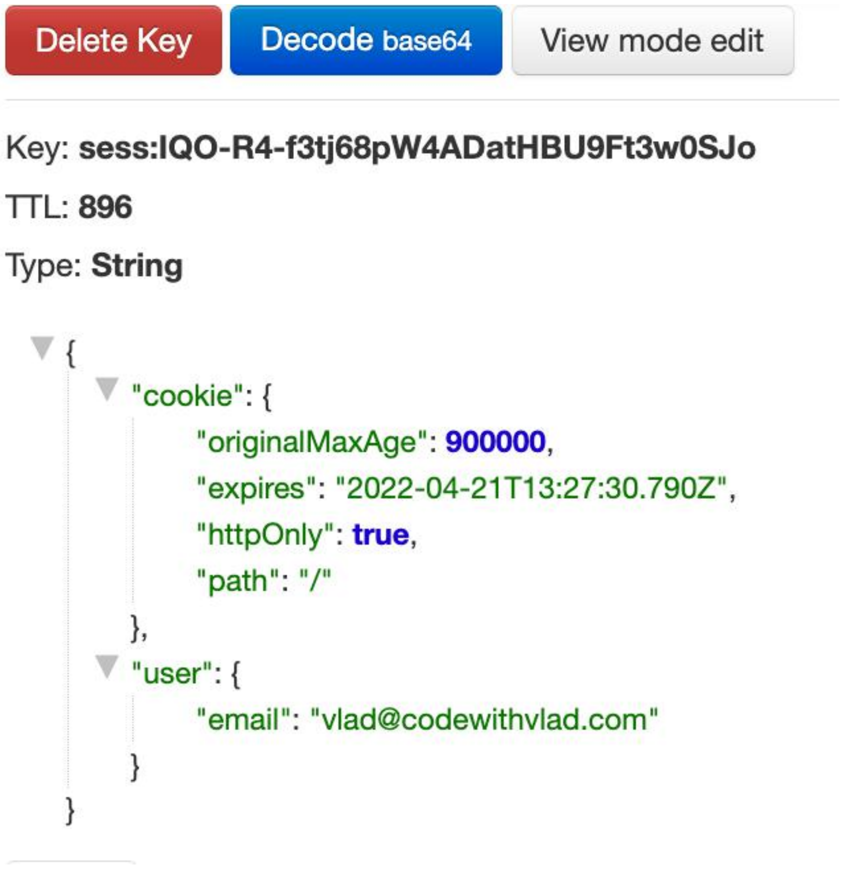
Update session expiration

You might notice that the cookie does not have an expiration date. Let’s add one.

main.ts

```
app.use(
  session({
    store: new RedisStore({ client: redisClient }),
    secret: 'super-secret',
    resave: false,
    saveUninitialized: false,
    cookie: {
      httpOnly: true, // prevents javascript from accessing the cookies
      maxAge: 1000 * 60 * 15, // cookie expiration date = 15 minutes
    },
  }),
);
```

The TTL has now be changed to 15 minutes (in seconds)



Key: **sess:lQO-R4-f3tj68pW4ADatHBU9Ft3w0SJo**

TTL: **896**

Type: **String**

```
{
    "cookie": {
        "originalMaxAge": 900000,
        "expires": "2022-04-21T13:27:30.790Z",
        "httpOnly": true,
        "path": "/"
    },
    "user": {
        "email": "vlad@codewithvlad.com"
    }
}
```

Conclusion

Congratulations if you've read so far. You have learned what sessions are and how to create them in your applications and scale them with Redis.

If you've enjoyed reading this eBook I would appreciate it if you can leave feedback.

If you want to learn more do not hesitate to write me at vlad@codewithvlad.com, leave a comment on my Twitter https://twitter.com/vladimir_agaev, or on my youtube channel https://www.youtube.com/channel/UCjmouj0JizYt0qTI53TAtFg

Any constructive criticism is, of course, welcome and I will make sure to include the changes in further versions of this book.

Thank you very much and good luck in your learning!

PS: As for the next steps, to master the subject even further I recommend you add a database to the mix and some authorization with roles (like admin, user).

Some of the routes will be accessible only to admins and the verification would be done based on sessions.