Лабораторная работа №5

Custom OAuth2Authorization

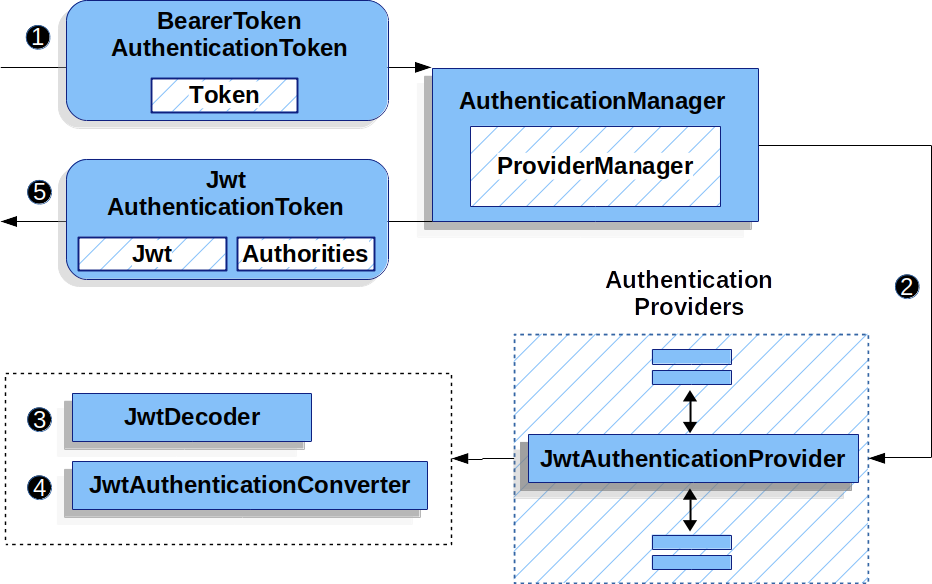


Схема OAuth2 Resource Server Jwt Authentication в Spring Security

1. BearerTokenAuthenticationFilter - фильтр, который перехватывает запросы, проверяет на наличие токена и пробует его конвертировать в аутентификацию.
2. JwtAuthenticationProvider - провайдер, задача которого аутентифицировать токен при помощи JwtDecoder и JwtAuthenticationProvider
3. JwtDecoder - объект в JwtAuthenticationProvider, ответственен за парсинг строкового токена в Jwt объект + валидацию токена
4. JwtAuthenticationConverter - перегоняет Jwt токен в авторизацию с разбором всех прав доступа, которые находятся в payload у Jwt
5. JwtAuthenticationToken - конечная успешная аутентификация, наследующая от AbstractAuthenticationToken

Работа выполнялась вместе со следующими зависимостями:

* spring-boot-starter-security: Spring фреймворк, обеспечивающий безопасность приложения. ОДНАКО он используется ТОЛЬКО для обеспечения доступа к эндпоинтам по ролям + содержит большое количество классов, благодаря которым удобно добавлять кастомную защиту в приложение (Писать с нуля свой аналог SpringSecurity тянет на Магистерскую)
* nimbus-jose-jwt:9.36: Библиотека, позволяющая работать с Jwt токенами, а именно: валидировать, верифицировать, подписывать и шифровать их.

Описание структуры приложения:

**Security** - директория, содержащая настройку безопасности приложения

В ней:

**filter** - директория с фильтрами

**model** - директория с сущностями для security

**jwt** - директория с классами, относящимися к Jwt

**provider** - директория с провайдерами

**configurer** - директория с конфигурациями

Краткое описание классов:

**JWTProcessor** - Nimbus класс, который при помощи публичных (иногда даже приватных) ключей проводит валидацию токена (верификация + проверка на истечения срока) чтобы распарсить payload токена

**JWT** - Nimbus реализация Jwt токена

**filter/BearerTokenAuthenticationFilter** - фильтр, перехватывающий запросы и проводящий валидацию токена в заголовке Authorization

**filter/StandardBearerTokenResolver** - класс, который извлекает токен из заголовка и проверяет его на корректность + проверяет, чтобы токен был один, а не множество.

**model/BearerTokenAuthentication** - кастомный класс аутентификации, наследуемый от AbstractAuthenticationToken, который хранит JWT токен и права доступа

**jwt/JwtDecoder** - интерфейс, который переводит строковый токен в JWT токен

**jwt/AccessTokenDecoder** - реализация JwtDecoder. Проводит как конвертацию, так и валидацию токена при помощи JWTProcessor

**jwt/DefaultJWTProcessorSupplier** - lazy поставщик JWTProcessor’a. Когда понадобится проверить первый токен, тогда мы его и получим. (Для этого есть свои причины)

**jwt/JwtAuthenticationConverter** - конвертер JWT токена в BearerTokenAuthentication с извлечением прав доступа из JWT.

**provider/JwtTokenAuthenticationProvider** - провайдер, который и конвертирует токен строку в авторизацию при помощи JwtDecoder и JwtAuthenticationCoverter

**provider/JWtTokenCacheAuthenticationProvider** - провайдер, который ищет в кэше ранее провалидированный и пока ещё не истекший JWT токен, после чего при помощи JwtAuthenticationCoverter конвертирует его в авторизацию. (Мог бы и авторизацию хранить, но это уже попахивает изобретение кастомной сессии, которая была отключена в приложении)

**configurer/CustomJwtConfigurer** - OAuth2Jwt конфигуратор для более удобной настройки Jwt аутентификации. Собирает в себя все выше описанные классы в одну конфигурацию, где их инициализирует + настраивает. Базовая настройка из коробки

**configurer/OAuth2JwtResourceServerProperties** - класс с пропертями, которые относятся к OAuth2 конфигурации, такими как: issuer-uri, jwk-uri и т.д. Может по запросу через метод сходить за openID конфигурацией к SSO и обновить ее

**configurer/SecurityConfiguration** - основная конфигурация безопасности приложения.

Всё, что не относится к директории security:

**repository/OAuth2TokenRepository** - интерфейс описывающий способы взаимодействия с JWT кэшом.

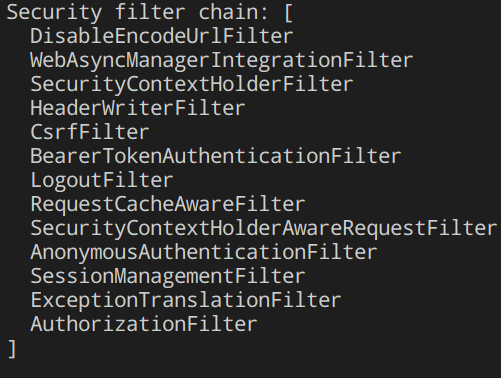
**repository/InMemoryCacheTokenRepository** - кэш, хранящий провалидированные JWT токены в ConcurrentHashMap. Ключом для кэширование выступает значение строки, полученный при помощи перегонки payload’a у JWT через SHA-256, значением - сам JWT токен

Почему существует DefaultJWTSupplier?

Всё просто - для JWTProcessor нужны ключи. Но их можно либо захардкодить (но keycloak меняет их с течением времени), либо получать по jwk\_uri ссылке через openID конфиг. Поэтому, чтобы отложить этот процесс и был придуман такой костыль. (Подсмотрено у Spring реализации).

В данном случае сначала делаю запрос по issuer-uri для получения конфига, затем ищу там jwks\_uri и потом делаю запрос туда для получения публичных ключей. (**OAuth2JwtResourceServerProperties в методе updateConfiguration и DefaultJWTProcessorSupplier в методе getJWKSetFromUri**)

Используемая цепочка фильтров:



**BearerTokenAuthenticationFilter** - кастомный фильтр

**CsrfFilter** - да, он включен. Однако в CustomJwtConfigurer (метод RegisterDefaultCsrfOverride) был подменён ignoreRequestMatcher, отчего при отсутствии csrf токена, но при наличии bearer токена, он будет пропускать запрос, а не разворачивать его

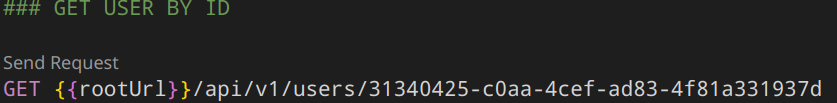
**AuthorizationFilter** - фильтр, который решает судьбу запроса. В нём и проходит голосование на пропуск запроса, и если хотя бы один голосующий скажет да - то you my friend now. Нормально описано здесь: [Путь запроса по внутренностям Spring Security / Хабр](https://habr.com/ru/articles/346628/)

Сами запросы:

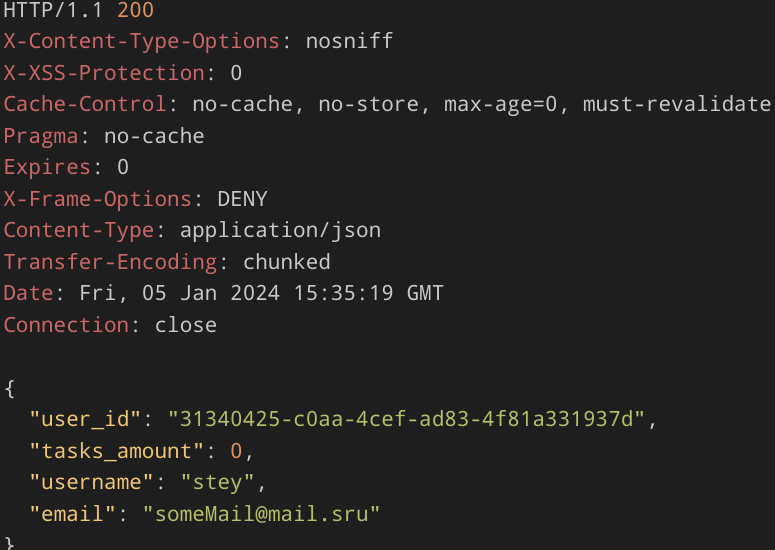
Все Get методы по пути api/v1/users/\*\* и api/v1/todo - возможны и без аутентификации, остальные только с ней. Причём POST /api/v1/user на создание пользователя - только для роли ROLE\_MODERATOR

Пока ещё не описан swagger файл + не пробовал через SATS, но нужно что-то показать уже сейчас (+ написать тесты и прокомментировать код, но пока сейчас времени нет)

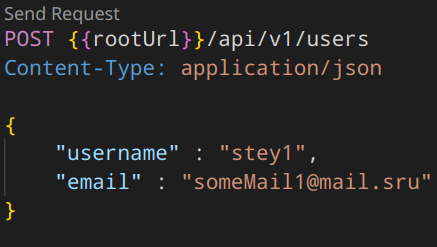
Запрос на получение пользователя без аутентификации (да и с ней прокатит)



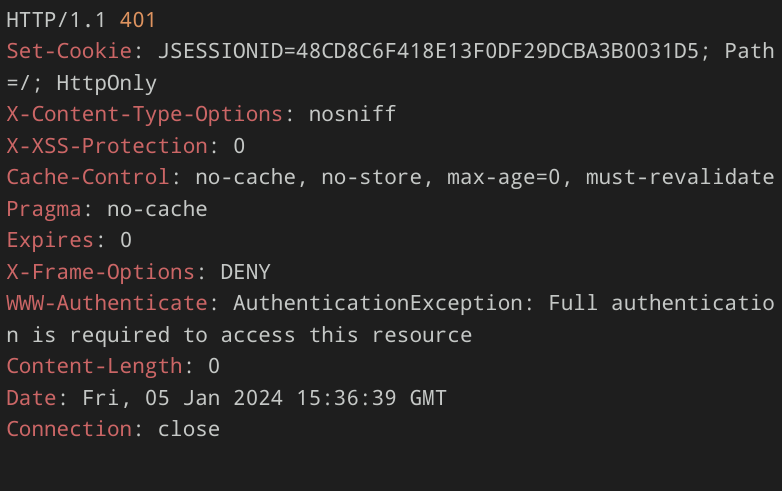
Ответ:



Запрос на создание пользователя без токена

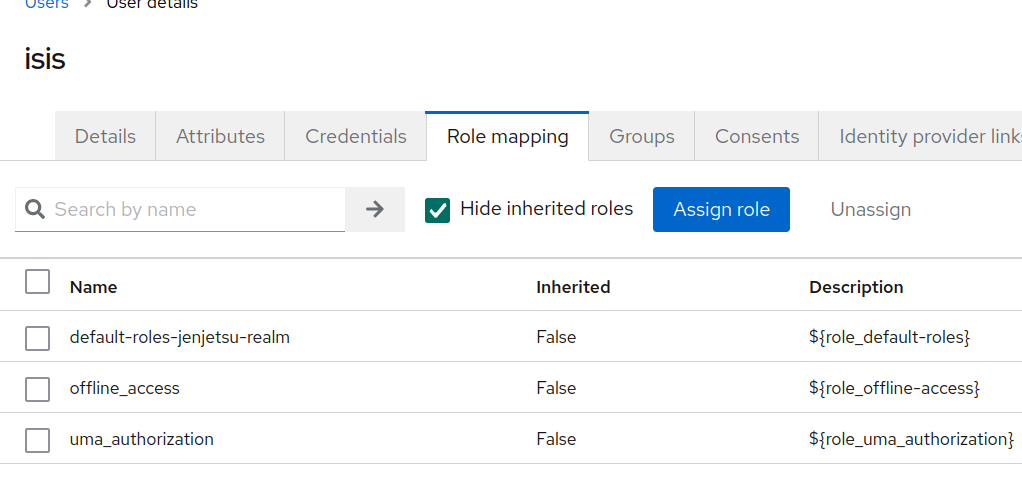


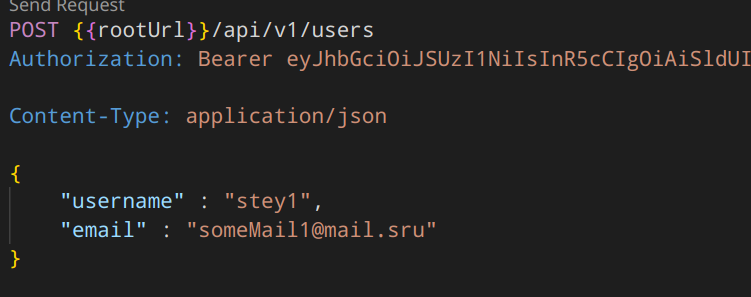
Ответ:



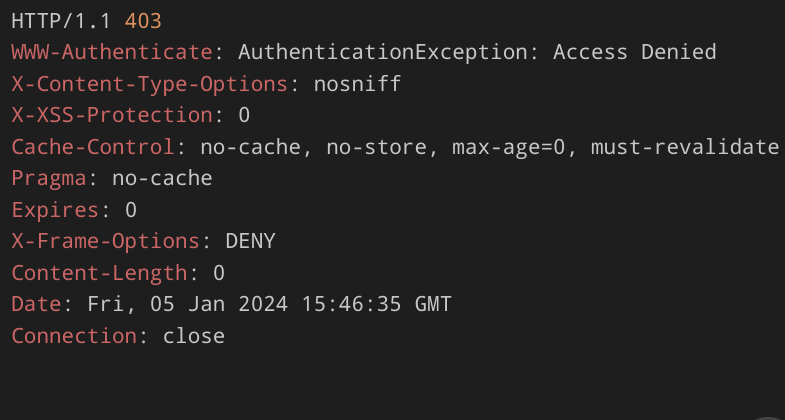
В заголовке WWW-Authenticate описана причина 401 кода

Запрос на добавление пользователя без роли ROLE\_MODERATOR

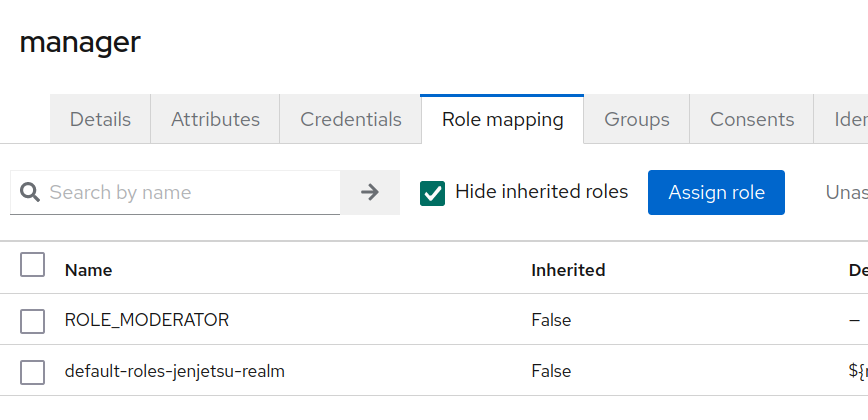


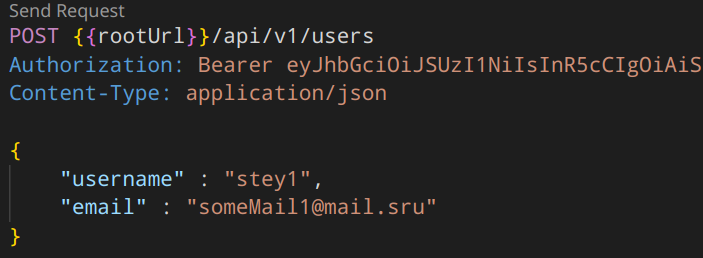


Ответ:

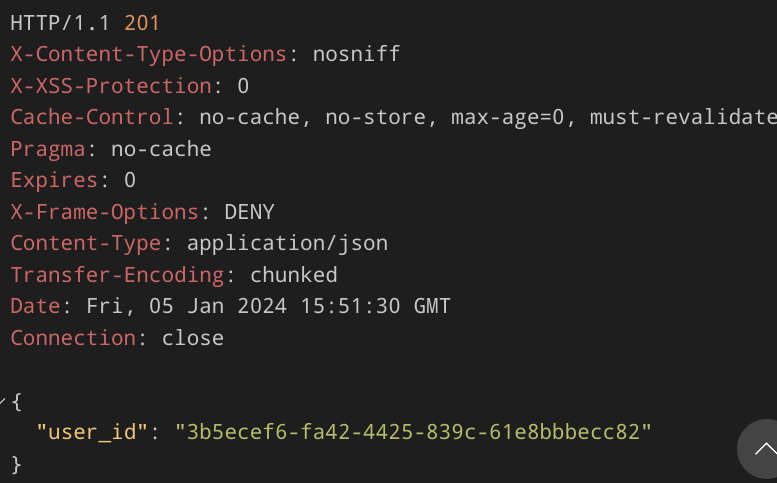


Запрос на добавление пользователя с ролью ROLE\_MODERATOR





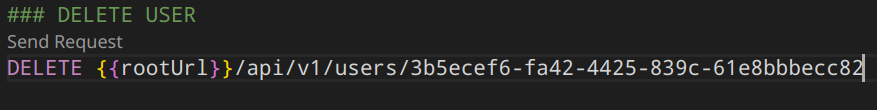
Ответ:



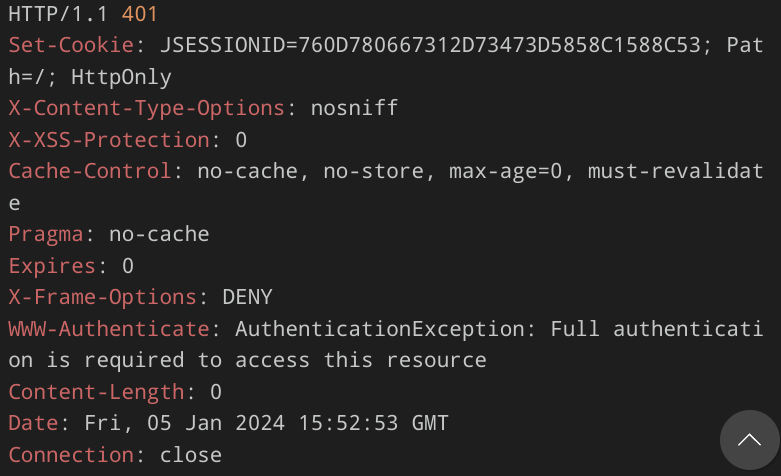
Токен:

eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI3QWE5dks0aUpqcG9FRzZ1dDBtblRsN1J4QnNzMGV4OVNFTjR0LXU3TTZVIn0..G51VRkFRdHYJVFc3Z97OYaO7i8x-gthdK1bFJBKOQwnxrRuAGX7hxAmbjuCpKuIp0EOGH\_M0E-UOLNoQgbNKtWGmmLboGvVjxxDeLhmJYJbkMFk3OHNInZUes4yAIhGTQFssIrtwpUER9lLVPQ5Vkz\_B9pjSPwbnPgZZHu5L1uJrMe5SUXP6PPmgdZjMTxAHp03j9APg7muhNZYwKVDAiteUmBLZHR8T1bztgENoZl-rAg0y19Od8JNnSIxz1Cxk8e2I7R9js-U8AjE10ZrMjwtZ\_YmbTI09nQ2vWpwUp2AQg0B4heZkPG1wzsbBPcV2ghVbc9gaWkECrUFtEYs86A

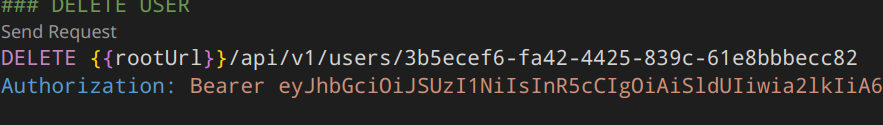
Запрос на удаление пользователя без токена



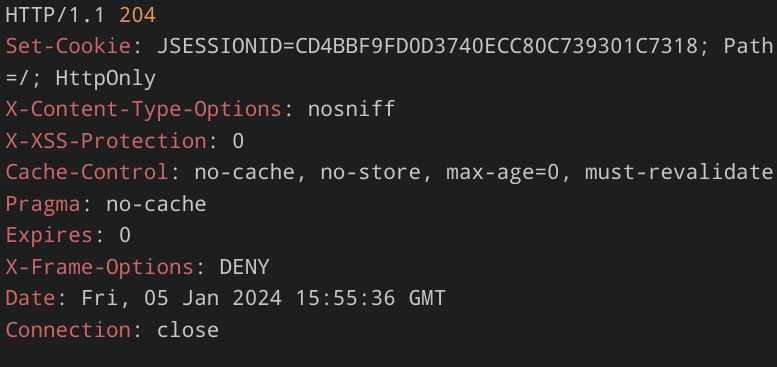
Ответ:



Запрос на удаление пользователя с токеном, но без роли ROLE\_MODERATOR



Ответ:

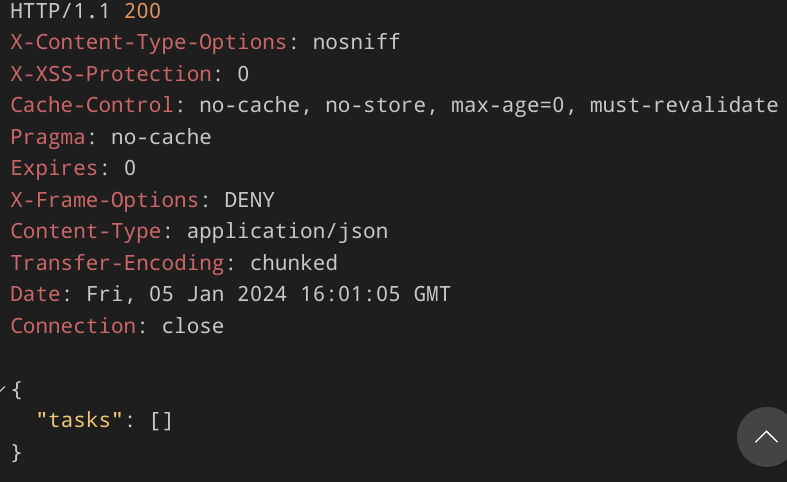


Токен: eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI3QWE5dks0aUpqcG9FRzZ1dDBtblRsN1J4QnNzMGV4OVNFTjR0LXU3TTZVIn0..bNcSKw2zKgI4iYI39nwbKLIOHdcPS3miI6Hd3lkJhhtPs\_Jj2CPoRLakDQPWmHHfPHjJ6XBr3q87d9F4akQTvzPACa9t4jkPdx\_UxM3-tYfqMkwhmy0eM6D7OeJnybgEcZLjnhdL8dururInO8ARs7J7w8cKWEuCN2N8TyQhB6K7\_scxe3HTAyxMj3jbs19tnf5Ig9Vvj7IVkzhIcPYev9uc411CxBk1UAQ96fAEwhjZUPB-tyXvGfFVRkNsbJzd0EASCbDPYhwSBW3GFaDS8OD0XWtSnLUngw7ZZkea0u3yFALS4IWrKHxW0-kvNMh94jIK2xugcP8iVwC3-btcYA

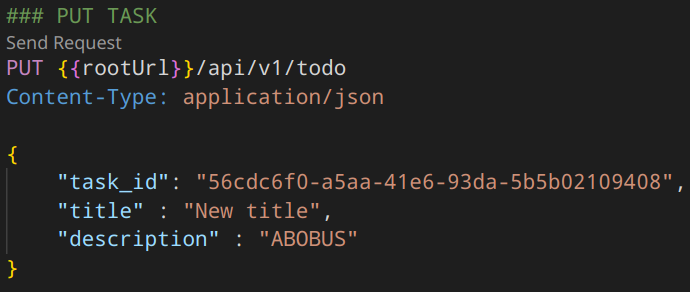
Запрос на получение задач пользователя без токена



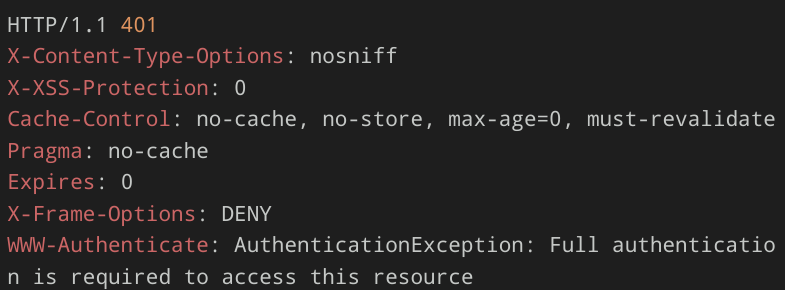
Ответ:



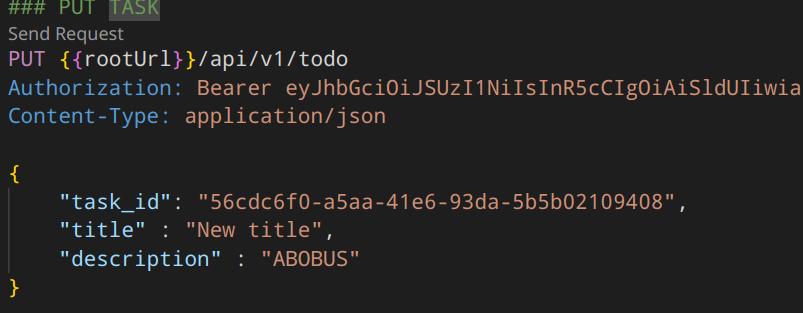
Запрос на обновление задачи без токена:



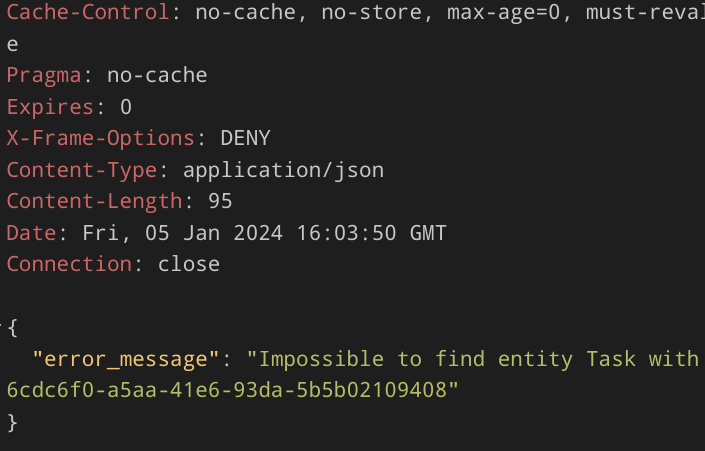
Ответ:



Запрос на обновление задачи с токеном и ролью ROLE\_MODERATOR



Ответ:



404 - т.к. нет задачи с таким if