Стажировка

«Веб-приложение для публикации постов – авторизация через JWT»

Выполнил: Ноздряков Богдан Валериевич

Проверил: Пармузин Александр Игоревич

Санкт-Петербург

2024

Условие:

Переписать текущую реализацию авторизации (сессионные куки) на JWT.

Анализ:

1) **Сессионные куки:**

   a. **Принцип:**

   -Сессия – это механизм, который позволяет серверу сохранить информацию о состоянии пользователя между различными запросами, тем самым во время нескольких запросов сервер будет понимать кто именно делает запрос.

   -При авторизации через сессионную куку – пользователь отправляет на сервер свои данные, через которые он аутентифицируется, например, почту и пароль. Сервер обработает эти данные. Если данные пройдут валидацию сервера, то сервер создаст сессию. Эта сессия будет хранить данные авторизированного пользователя, а также сессия будет хранить свой id, который будет создан и зашифрован с помощью секретного ключа. После создания сессии сервер передает клиенту куки файл, который будет содержать зашифрованный id созданной сессии. Браузер автоматически сохранит этот куки файл, и каждый раз при отправке пользователем запроса на сервер – браузер автоматически будет передавать этот куки с id сессии. Сервер же получит этот куки, возьмет из него id сессии, расшифрует этот id с помощью того же секретного ключа и найдет нужную сессию по расшифрованному id. Дальше сервер просто восстанавливает информацию о состоянии пользователя по сессии

   b. **Хранение:**

   -Сами сессии хранятся на сервере. Однако на клиенте хранятся куки файлы, которые содержат соответствующий id сессии. На сервере же сессии можно хранить:

   - В памяти сервера (в нашей текущей реализации авторизации через сессионные куки мы использовали именно такой подход) – информация о сессии доступна только во время выполнения текущего процесса Node.js и исчезает, если процесс завершается (при перезапуске сервера текущие сессии теряют свое состояние и пользователю придется заново авторизироваться)

   - В базе данных – позволяет сохранить информацию о всех сессиях даже после перезапуска сервера. И в таком случае – пользователю не придется заново авторизироваться

Из этих двух подходов, наиболее предпочтительным является хранение сессий в базе данных, так как в таком случае все сессии будут устойчивы к сбоям на сервере (все состояния сохраняются после перезапуска сервера).

Также стоит отметить, что для наибольшей безопасности, на клиенте id сессий стоит хранить в куки-файлах httpOnly

c. Преимущества:

- Сессии позволяют сохранять состояние пользователя на сервере. Это один большой плюс для использования сессий. Благодаря хранению состояния пользователя – будет сокращено сокращается количество данных, передаваемых между клиентом и сервером. Также если служба безопасности будет подозревать, что аккаунт взломан, то они немедленно могут аннулировать id сессии, чтобы пользователь вышел из системы

- Такой подход считается достаточно безопасным, и он активно используется для веб-приложений. Злоумышленнику будет непросто что-то сделать, так как ему нужно будет получить id сессии для входа в систему, что сложно реализовать. К тому же применяются различные механизмы, помогающие понять, что сессией завладел злоумышленник. В таком случае можно будет без проблем аннулировать сессию, так как она хранится на сервере

d. Недостатки:

- Сервер хранит и синхронизирует информацию о каждой активной сессии. Это требует значительные ресурсы, а также нагружает сервер

- Данный метод может быть уязвимым к определенным атакам, например, подделка межсайтовых запросов

2) JWT:

a. Принцип:

-Идея авторизации на JWT-токенах заключается в том, чтобы предоставить злоумышленнику как можно меньше времени на пользование контентом пользователя, если злоумышленник смог каким-то образом получить доступ к аккаунту пользователя.

-JWT (JSON WEB TOKEN)– открытый стандарт, определяющий способ передачи информации между двумя сторонами в виде JSON объекта. Способ основан на создании токенов доступа через формат JSON. Обычно используется для передачи данных для аутентификации пользователей в клиент-серверных приложениях. Токены создаются сервером, подписываются через секретный

ключ и передаются пользователю, которые потом будут использоваться для подтверждения личности пользователя.

-В приложении, написанном на JWT, чтобы злоумышленнику получить доступ к контенту пользователя, ему нужно узнать либо токен доступа, либо токен обновления.

-JWT-токен состоит из трех частей:

- header – JSON, закодированный с помощью base64. Представляет из себя заголовок токена. В расшифрованном виде выглядит так:

  {
  “alg”: “HS256”,
  “typ”: “JWT”
  }

  здесь alg – это алгоритм, с помощью которого создается подпись токена. Обычно используются криптографические алгоритмы HS256 или RS256. Но можно использовать и другие алгоритмы шифрования.

  В зашифрованном виде будет выглядеть примерно так:
  `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`

- payload - JSON, закодированный с помощью base64. Представляет из себя полезную нагрузку токена. Здесь хранится нужная информация о пользователе и токене, например, имя пользователя, его id и т.д. В расшифрованном виде будет выглядеть как-то так:

  {
  “name”: “Bogdan”,
  “email”: “voyagerbvb@gmail.com”
  }

  В зашифрованном виде будет выглядеть примерно так:

  `ewxhbGkiOiIxMjM0NTY3ODkwIiwibmFtZSI6ICJtYWlsIiwiYWRtaW4iOnRydWV9`

  Важно: эти данные шифруются и расшифровать их может кто угодно. Поэтому в полезной нагрузке никогда не должны храниться приватные данные, только публичные.

- signature – подпись токена. Создается по принципу:

  signature = HMAC_SHA256(secretKey, base64urlEncoding(header) + ‘.’ + base64urlEncoding(payload))

  В зашифрованном виде будет выглядеть примерно так:
  `eyJuYW1lIjoiSm9obiBEb2UiLCJzdWIiOiIxMjM0NTY3ODkwIiwgImFnZSI6dHJ1ZXQh`

То есть хедер и полезная нагрузка кодируются с помощью base64urlEncoding, конкатинируются в одну строку, разделенную точкой, а после получившуюся строку кодируют с помощью секретного ключа secretKey и выбранного алгоритма, который был указан в хедере.

secretKey – это ключ для шифровки и проверки подписи. Он генерируется и хранится на сервере и используется для подписи токена при генерации. Также он нужен для проверки токена при получении. Важно обеспечивать недоступность данного ключа. Если к злоумышленнику попадет данный ключ, то он сможет самостоятельно создавать валидные токены к приложению. Обычно данный ключ генерируется в формате hex

-Конечный JWT-токен представляет из себя строку, состоящую из трех ранее описанных частей, соединенных точками. Согласно примеру выше, в итоге мы получим следующий JWT-токен:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
ewxhbGkiOiIxMjM0NTY3ODkwIiwibmFtZSI6ICJtYWlsIiwiYWRtaW4iOnRydWV9.
eyJuYW1lIjoiSm9obiBEb2UiLCJzdWIiOiIxMjM0NTY3ODkwIiwgImFnZSI6dHJ1ZXQh
```

То есть мы получили строку – header.payload.signature

-Данный формат гарантирует нам сохранность данных и невозможность подмены, так как для проверки подлинности токена достаточно взять из него его header и payload, и с их помощью, а также с помощью нашего секретного ключа, создать signature по описанному принципу и сравнить его с signature, который реально присутствует в переданном токене.

Теперь если злоумышленник как-то изменит данные в украденном токене, или попытается подделать токен по полезной нагрузке через свой секретный ключ (не наш), то токен будет признан недействительным – запрос будет отклонен сервером.

b. Токены доступа и обновления:

-Важной частью авторизации на основе JWT является наличие токенов доступа и обновления. Токен доступа – токен, который отвечает за доступ к информации. Именно за счет этого токена сервер проверяет – авторизирован ли пользователь, отправляющий запрос или нет. Токен доступа в целях безопасности должен жить недолго (время жизни измеряется минутами). Токен обновления – токен, который отвечает за создание нового токена доступа, если старый токен доступа истек. Токен обновления живет долго (время жизни измеряется днями).

Авторизация на JWT работает на основе токенов:

Пользователь отправляет на сервер свои данные, например, почту и пароль. Сервер проводит валидацию этих данных. Если валидация пройдет успешно, то сервер создаст токен доступа и токен обновления на основе данных пользователя и секретного ключа. Также сервер установит срок жизни для обоих токенов – для токена доступа где-то минут 15, а для токена обновления где-то пару дней. Токен обновления в целях безопасности будет записан в базу данных. После пользователю отправляются два этих токена на клиент. На клиенте эти токены сохраняются. Дальше при каждом запросе пользователя – будут передаваться эти токены. Сервер будет проводить валидацию токена доступа. Если валидация пройдет успешно, то сервер получит из расшифрованного токена полезную нагрузку с данными пользователя, и на основе этих данных, сервер поймет кто делает запрос. Когда токен доступа истечет, то сервер просто воспользуется переданным токеном обновления. Сначала сервер будет искать этот токен в базе данных. Если он там есть, то сервер затем проведет валидацию токена обновления. Если валидация пройдет успешно, то сервер создаст на основе данного токена обновления новый токен доступа и вернет его пользователю. Когда истечет и токен обновления – пользователю придется заново авторизироваться. Таким образом, пользователь авторизируется только один раз в системе, а дальше он в течении долгого времени может не заполнять форму авторизации, а просто пользоваться функционалом приложения, так как приложение будет понимать, что это за пользователь.

-Также возможно использовать только один токен доступа без токена обновления. Однако такой подход не является практичным. Если делать авторизацию на основе одного токена доступа, то придется сделать так, чтобы он мало жил – максимум несколько часов. Это делается ради безопасности. Напомню авторизация JWT говорит, что если злоумышленник получит доступ к аккаунту пользователя, то у него должно быть как можно меньше времени для того, чтобы воспользоваться функционалом. А если единственный токен доступа будет мало жить, то пользователю придется очень часто заново заполнять форму авторизации, что может сказаться плохо как на пользователе, так и на нагрузке на сервер. Именно поэтому важно создавать токен доступа и токен обновления, где токен доступа будет очень мало жить, а токен обновления будет жить много. Кроме того, использование токена обновления повышает безопасность – злоумышленнику будет тяжелее получить доступ к аккаунту пользователя, а также в случае, если злоумышленник завладеет токеном, то это будет проще отследить, и будет проще закрыть доступ злоумышленнику к аккаунту пользователя.

c. Хранение:

-Токены доступа и обновления хранят на клиенте, однако токен обновления хранят, помимо клиента, в базе данных. Все способы хранить токены на клиенте:

- Local Storage
- Session Storage
- Cookie
- Память клиентского приложения

Самым безопасным способом считается хранение токенов на клиенте в Cookie-файлах, установленных в httpOnly. Однако даже такой способ не гарантирует того, что злоумышленник не сможет выкрасть токены. Из-за этого – нужно включать дополнительные меры безопасности при работе с токенами на сервере:

- Не использовать только токен доступа. Помимо токена доступа – также использовать токен обновления
- Токены доступа должны отвечать за доступ к функционалу, а токен обновления только за обновление токенов доступа
- Токены доступа не должны долго жить (жизнь считается минутами), а токены обновления должны жить долго (жизнь считается днями). Токены доступа должны регулярно обновляться
- Хранить токен обновления в базе данных и каждый раз, когда токен доступа обновляется через переданный токен обновления, проводить не только валидацию переданного токена обновления, но и искать его в базе данных
- Использовать дополнительные меры безопасности. Например, анализ приходящих запросов на подозрительность их источника и т.д.

d. Преимущества:

- Пользователю нужно будет один раз отправить свои учетные данные на сервер, а после ему не нужно будет это выполнять в течение длительного периода времени
- JWT никак не оперирует состояниями, поэтому серверу не нужно хранить записи с пользовательскими токенами или сессиями. Каждый токен самодостаточен, содержит все необходимые для проверки данные, а также передаёт затребованную пользовательскую информацию (однако это спорное преимущество, поскольку оно не всегда реализуется, так как для лучшей безопасности токен обновления стоит хранить в базе данных, и искать в базе данных токен обновления при каждом запросе, где используется токен обновления для обновления токена доступа)
- Данный метод просто реализовать на любых платформах: веб, мобильные платформы, приложения интернета вещей
- Данный метод авторизации считается безопасным и довольно часто применяется в различных приложениях. Злоумышленнику будет трудно причинить вред, так как для этого ему потребуется получить либо токен доступа, либо токен обновления, либо сразу оба. При соблюдении качественных мер безопасности (использовать https, хранить оба токена в куки httpOnly и т.д.) – злоумышленнику будет сделать это очень

сложно. Даже если он получит токен доступа – токен доступа живет очень мало, поэтому время будет сильно ограничено. Если же он получит токен обновления, то будут проблемы. Однако их можно решить с помощью дополнительных мер безопасности (хранение токена обновления в базе данных, анализу входящих запросов и т.д.). При правильной реализации – можно будет узнать, что злоумышленник завладел токеном обновления и закрыть ему доступ

e. Недостатки:

- Нелегко отозвать все JWT-токены, так как это механизм аутентификации без состояния
- Данный метод может быть уязвимым к определенным атакам, например, “Человек посередине”
- Непросто понять, что какой-то из токенов был получен злоумышленником, а также непросто закрыть доступ злоумышленнику
- Сложно управлять жизненным циклом токенов
- JWT может содержать большой объем данных, из-за чего будут проблемы с производительностью

Сравнение сессионной куки с JWT:

-Оба метода направлены на то, чтобы пользователь отправил свои учетные данные один раз, и ему в течение длительного периода не требовалось повторно отправлять свои учетные данные. Оба метода имеют как ряд преимуществ, так и ряд недостатков. Оба метода считаются достаточно безопасными и часто используются в различных приложениях. Оба метода не идеальны и могут пострадать от атак злоумышленников, из-за чего приходится внедрять и туда, и туда дополнительные меры безопасности, а также использовать наиболее безопасные подходы в хранении и управлении.

-В чем-то JWT превосходит сессионные куки, например, JWT можно применить на многих платформах, а с сессионными куками будут определенные проблемы на мобильных платформах, их нельзя применить в приложениях интернета вещей и т.д. Также JWT не требуют постоянного хранения состояния сессии на сервере, благодаря чему сервер может обрабатывать запросы без необходимости обращаться к базе данных или другому хранилищу (однако тут не все просто, так как для повышения безопасности, рекомендуется хранить токен обновления в базе данных, и искать токен обновления каждый раз, когда происходит обновление токена доступа через переданный токен обновления)

-В чем-то сессионные куки превосходят JWT, например, при таком подходе легче всем управлять, к примеру, выходом пользователя из системы и т.д.

Решение:

Наше приложение будет реализовывать авторизацию через JWT следующим образом:

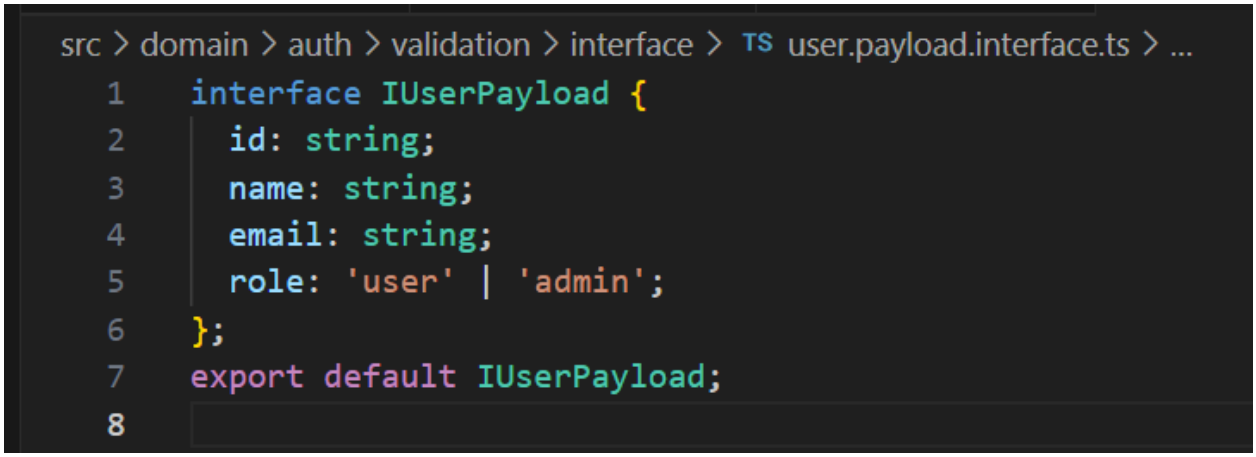
-Авторизация работает на двух JWT-токенах: токен доступа и токен обновления, что является наиболее практичным и безопасным способом, чем использовать один токен доступа. Оба токена будут храниться на клиенте в куки файлах httpOnly: токен доступа в куки access-token, токен обновления в куки refresh-token. Хранить данные токены на клиенте в куки httpOnly является наиболее безопасным способом хранения.

-Токен обновления будет храниться не только на клиенте. Токен обновления будет также храниться на сервере в базе данных в колонке refreshToken соотвествующего пользователя. Это необходимо для обеспечения большей безопасности (каждый раз при использовании токена обновления, мы будем искать его в базе данных перед тем, как переходить к валидации). Также это нужно, чтобы правильно реализовать метод logout и иметь возможность обнулить токен обновления пользователю в случае, когда это необходимо.

-Когда пользователь авторизируется в системе, в клиенте создадутся два куки httpOnly – accesss-token под токен доступа и refresh-token под токен обновления. Токен доступа будет жить 15 минут, а токен обновления 3 дня. Куки файлы будут жить в соответствии их токенам: access-token – 15 минут, refresh-token – 3 дня. Токен доступа будет отвечать за то, чтобы дать понять серверу, что пользователь авторизирован. Токен обновления будет обновлять пользователю токен доступа каждый раз, когда токен доступа истечет. Таким образом, когда пользователь авторизируется в системе – он сможет 3 дня не заполнять форму авторизации и просто заходить на страницу и сразу пользоваться функционалом, требующим авторизации. Так происходит, потому что сервер проверяет полученный токен доступа пользователя. Если токен доступа пройдет проверку – сервер разрешит пользователю пользоваться функционалом приложения. А когда текущий токен доступа истечет – сразу после того, как клиент выполнит любой запрос, требующий авторизации – сервер возьмет токен обновления, проведет его проверку, и, если проверка прошла успешно – сервер создаст новый токен доступа, а также новый куки httpOnly, хранящий данный токен доступа. Если же и токен доступа истечет, то пользователю заново придется заполнять форму авторизации и входить в систему. Благодаря такому подходу – пользователь может один раз авторизоваться, а после ему долго не потребуется заново выполнять это процедуру – до тех пор, пока не истечет токен обновления.

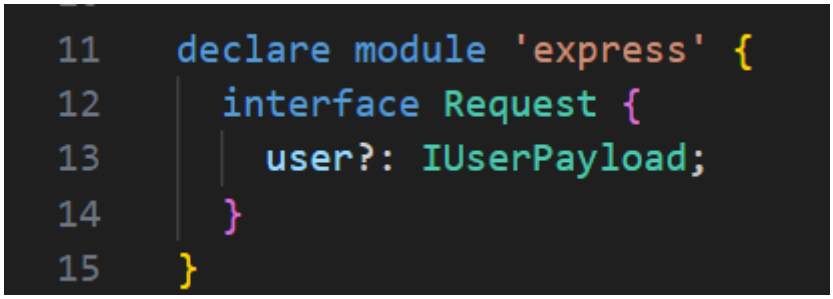
-Поскольку на клиенте и токен доступа, и токен обновления хранятся в куках httpOnly, то на клиенте нельзя получить оба этих токена. Это значит, что из клиента нельзя отправить в заголовках запроса данные токены. Из-за этого сам сервер будет брать данные токены из куки и проводить с ними все необходимые процедуры. Такой подход – повышает безопасность.

Как было сказано ранее, для создания JWT, нам понадобится создать специальный объект, представляющий из себя полезную нагрузку. Также во многих методах API для их корректной работы – нужно будет получать из запроса специальный объект типа User. Для удобства – создадим один интерфейс IUserPayload для объекта полезной нагрузки и объекта, который требуется в API (./src/domain/auth/validation/interface/user.payload.ts):



Данный интерфейс содержит id пользователя, его имя, почту и роль. Это минимальный набор, который требуется для работы API. Также все эти данные являются безопасными, так как если ими завладеет злоумышленник, то они ему мало что дадут.

Также нужно переопределить интерфейс Request из express, чтобы можно было передавать в запросе эти данные типа IUserPayload. Раньше мы переопределяли интерфейс SessionData из express-session, чтобы можно было в сессии записывать пользователя. Однако теперь мы не используем сессии. Переопределим Request (./src/app.config.ts):



Теперь мы сможем в запросах устанавливать данные IUserPayload, благодаря чему в API можно будет получать эти данные.

Для внедрения JWT создадим специальный класс JWTStrategy (./src/utils/lib/jwt/jwt.strategy.ts):

Данный класс является провайдером для различных методов из библиотеки jsonwebtoken, которая позволяет реализовать авторизацию через JWT. Класс содержит:

1) secretKey – поле, которое содержит секретный ключ, необходимый для создания JWT-токена и его декодирования. Значение секретного ключа берется из env-файла

2) createAccessToken – метод, который создает токен доступа JWT на основе переданной полезной нагрузки IUserPayload и секретного ключа secretKey. Данный токен истекает через 15 минут после его создания

3) createRefreshToken - метод, который создает токен обновления JWT на основе переданной полезной нагрузки IUserPayload и секретного ключа secretKey. Данный токен истекает через 3 дня после его создания

4) validateToken – метод, который проводит валидацию переданного токена. Данный метод пытается декодировать полученный токен с помощью секретного ключа secretKey. Если вдруг переданный токен был создан с помощью другого ключа или его содержимое было как-то изменено, то декодировать токен не получится, и в таком случае вернется ошибка. Это будет означать, что токен не прошел валидацию, то есть полученный токен – от злоумышленника. Если же полученный токен был создан с помощью ключа secretKey, а также его содержимое не было изменено, то его получится декодировать. В таком случае метод вернет данные токена, и будет считаться, что токен прошел валидацию

Также не забудем создать класс JWTModule (./src/utils/lib/jwt/jwt.module.ts), который будет создавать синглтон JWTStrategy, делая его инъектируемым, а также создать сам экземпляр модуля JWTModule в AppModule (./src/app.module.ts).

Перед переходом на JWT потребуется создать новую колонку в таблицу User – refreshToken. Даная колонка будет хранить токен обновления для данного пользователя. Когда пользователь авторизуется, созданный для него токен обновления будет не только записан в куки на клиенте, но еще и в базу данных, как-раз в колонку refreshToken данного пользователя. В дальнейшем, при обновлении токена доступа через токен обновления, мы будем искать переданный токен обновления в базе данных. Этим мы значительно повышаем безопасность.

Создадим данную колонку в таблице User (./src/database/models/user/user.model.ts):

```
60    @Column({
61      type: DataType.TEXT,
62      allowNull: true
63    })
64    refreshToken!: string | null;
```

Также нам необходимо написать миграцию, которая будет добавлять данную колонку в базу данных (./src/database/migrations/20240708094818-add_refresh_token_column_to_user_table.js):

src > database > migrations > JS 20240708094818-add_refresh_token_column_to_user_table.js > ...

```
1   const { Sequelize } = require('sequelize')
2
3   async function up({ context: queryInterface }) {
4     const tableInfo = await queryInterface.describeTable('Users');
5     if (!tableInfo.refreshToken) {
6       await queryInterface.addColumn('Users', 'refreshToken', {
7         type: Sequelize.TEXT,
8         allowNull: true
9       });
10    }
11  }
12
13  async function down({ context: queryInterface }) {
14    const tableInfo = await queryInterface.describeTable('Users');
15    if (tableInfo.refreshToken) {
16      await queryInterface.removeColumn('Users', 'refreshToken');
17    }
18  }
19
20  module.exports = { up, down }
21
```

После реализации всех данных шагов, можно переписывать логику авторизации на JWT.

Изменим AuthContoller (./src/domain/auth/auth.controller.ts):

1) В данном контроллере нам потребуется использовать функционал из созданного ранее класса JWTStrategy, поэтому инъектируем в AuthContoller – JWTStrategy:

```
12  class AuthController extends DomainController {
13    constructor(
14      private readonly mailerTransporter: MailerTransporter,
15      private readonly cryptoProvider: CryptoProvider,
16      private readonly jwtStrategy: JWTStrategy,
17      private readonly userService: UserService
18    ) {
19      super();
20    }
```

Не забудем в AuthModule (./src/domain/auth/auth.module.ts) при создании синглтона AuthContoller – передать в конструктор JWTStrategy

2) Нужно реализовать метод, через который на клиенте можно будет получать данные авторизованного пользователя. Для этого изменим метод getAuthData на getUserPayload:

```
22  public getUserPayload(req: Request, res: Response, next: NextFunction): Response {
23    const user = req.user;
24
25    if (user) {
26      return res.status(200).json({
27        status: 200,
28        data: {
29          name: user.name,
30          role: user.role,
31          email: user.email
32        },
33        message: "User details"
34      });
35    }
36
37    return res.status(204).end();
38  }
```

В данном методе – мы из запроса получаем данные пользователя (возможно за счет переопределения интерфейса Request модуля express в app.config.ts). Дальше если эти данные были заданы, то возвращается ответ, содержащий имя пользователя, его роль, его почта. Если же данные не были заданы, то возвращается ответ 204

3) Изменим метод signin:

```
40  public async signin(req: Request, res: Response, next: NextFunction): Promise<Response | void> {
41    try {
42      const userDataAuth = req.body;
43      const { error } = UserAuthSchema.validate(userDataAuth);
44
45      if (error) {
46        return res.status(422).send(`Validation error: ${error.details[0].message}`);
47      }
48
49      const user = await this.userService.getUserByEmail(req.body.email);
50      const userPassword = req.body.password;
51      const hashUserPassword = await this.cryptoProvider.hashStringBySHA256(userPassword);
52
53      if (user.dataValues.password !== hashUserPassword) {
54        return res.status(401).send("Unauthorized - incorrect email or password");
55      }
56
57      const userPayload: IUserPayload = {
58        id: user.dataValues.id,
59        name: user.dataValues.name,
60        email: user.dataValues.email,
61        role: user.dataValues.role
62      };
63
64      const accessToken = this.jwtStrategy.createAccessToken(userPayload);
65      const refreshToken = this.jwtStrategy.createRefreshToken(userPayload);
```

```
64        const accessToken = this.jwtStrategy.createAccessToken(userPayload);
65        const refreshToken = this.jwtStrategy.createRefreshToken(userPayload);
66
67        await this.userService.updateUserAuthData(user, {
68          refreshToken: refreshToken
69        });
70
71        return res.status(200)
72          .cookie('access-token', accessToken, {
73            httpOnly: true,
74            secure: false,
75            maxAge: 900000
76          })
77          .cookie('refresh-token', refreshToken, {
78            httpOnly: true,
79            secure: false,
80            maxAge: 259200000
81          })
82          .json({
83            status: 200,
84            data: {
85              name: user.name,
86              email: user.email
87            },
88            message: "Authorization was successful"
89          });
90      }
```

```
    catch (err) {
      if (err instanceof HttpError && err.status === 404) {
        return res.status(401).send("Unauthorized - incorrect email or password");
      }

      next(err);
    }
  }
```

Отличие заключается в том, что теперь после того, как метод определил, что пользователь ввел верные почту и пароль – создаются токены доступа и обновления для данного пользователя с помощью инъектированного JWTStrategy. Токен обновления записывается в базу данных в колонку refreshToken данного пользователя. В конце – отправляется ответ, содержащий информацию, что пользователь успешно авторизовался, а также создаются два кука httpOnly – access-token, который содержит созданный токен доступа и refresh-token, который содержит созданный токен обновления. Живут данные куки столько же, сколько и токены, которые в них записаны - access-token живет 15 минут, а refresh-token 3 дня.

4) Изменим метод logout:

```
public async logout(req: Request, res: Response, next: NextFunction): Promise<Response | void> {
  try {
    const userId = req.user?.id as string;
    const user = await this.userService.getUserById(userId, false);
    await this.userService.updateUserAuthData(user, {
      refreshToken: null
    });

    if (req.cookies['refresh-token']) {
      res.clearCookie('refresh-token');
    }

    res.clearCookie('access-token');
  }
  catch (err) {
    next(err);
  }

  return res.status(200).json({ status: 200, message: "Logged out successfully" });
}
```

Теперь мы получаем из запроса данные пользователя типа IUserPayload и с помощью этих данных у соответствующего пользователя обнуляем токен обновления в базе данных. После мы удаляем на клиенте пользователя куки, который хранит токен доступа, а также удаляем на клиенте пользователя куки, который хранит токен обновления, если он еще живет. Если все прошло успешно, то возвращается ответ 200 о том, что пользователь успешно вышел из аккаунта.

Такой подход позволит обычному пользователю выйти из системы. Он не сможет пользоваться функционалом приложения, пока не пройдет заново авторизацию, так как мы удаляем его файлы куки. А гарды проверяют наличие токенов именно по данным куки. Если же пользователь выйдет из системы, но ранее злоумышленник украдет об его токена, то злоумышленник сможет что-то сделать только пока у

него будет существовать токен доступа. А токен доступа живет очень мало – всего 15 минут. Поэтому по истечению этого времени – злоумышленник ничего не сможет. Хоть у него и будет токен обновления – это ему ничего не даст, поскольку мы удалили токен обновления из базы данных, а сервер при обновлении токена доступа через токен обновления, проверяет наличие токена обновления в базе данных. Таким образом, имя только токен обновления – злоумышленник ничего не получит. Таким образом, мы придерживаемся главной концепции авторизации на JWT – максимально ограничить время злоумышленнику, в течение которого он может пользоваться контентом пользователя, в случае кражи данных пользователя.

Все остальные методы контроллера AuthController не изменяются.

Также нам необходимо изменить два гарда модуля auth. Для начала внедрим новое промежуточное ПО refreshToken (./src/domain/auth/middleware/auth.middleware.ts):

src > domain > auth > middleware > TS auth.middleware.ts > ...

```ts
import { Request, Response } from 'express';
import dependencyContainer from '../../../utils/lib/dependencyInjection/dependency.container';
import UserService from '../../user/service/user.service';
import JWTStrategy from '../../../utils/lib/jwt/jwt.strategy';

const refreshToken = async (req: Request, res: Response) => {
  const userService = dependencyContainer.getInstance<UserService>('userService');
  await userService.getUserByRefreshToken(req.cookies['refresh-token']);

  const jwtStrategy = dependencyContainer.getInstance<JWTStrategy>('jwtStrategy');
  const userPayload = await jwtStrategy.validateToken(req.cookies['refresh-token']);
  const accessToken = jwtStrategy.createAccessToken({
    id: userPayload.id,
    name: userPayload.name,
    email: userPayload.email,
    role: userPayload.role
  });

  res.cookie('access-token', accessToken, {
    httpOnly: true,
    secure: false,
    maxAge: 900000
  });

  return accessToken;
};
export default refreshToken;

```

Данный метод будет отвечать за то, чтобы обновить токен доступа пользователя через его токен обновления. Сначала метод проверяет, что в базе данных есть переданный токен обновления. Если такой токен обновления в базе данных есть, то проводится валидация данного токена через метод validateToken класса JWTStrategy. Если валидация прошла успешно, то создается новый токен доступа, а также создается куки httpOnly содержащий данный токен доступа. В конце функция возвращает созданный токен доступа.

Данное промежуточное ПО позволит самому серверу создавать новый токен доступа за счет токена обновления.

Теперь изменим два гарда:

1) authGuard (./src/domain/auth/middleware/guard/auth.guard.ts):

Изменение данного гарда заключается в том, что он теперь не только защищает маршруты, но и следит за обновлением токенов доступа. Гард берет из куки httpOnly access-token токен доступа. При этом происходит проверка – если полученный токен доступа не определен, а также токен обновления в куки httpOnly refresh-token определен, то тогда токен доступа обновляется за счет созданного ранее промежуточного ПО refreshToken. Если же токен доступа не определен, а также токен обновления не определен, то это означает, что срок действия токена доступа закончился и пользователю нужно заново авторизироваться. В таком случае гард возвращает ответ о том, что пользователю нужно авторизироваться.

Также данный гард по-другому проверяет авторизирован ли пользователь. После успешного получения токена доступа, гард проводит валидация полученного токена доступа с помощью метода validateToken класса JWTStrategy. Если валидация пройдет успешно, то в запросе создается user, который будет содержать данные IUserPayload, полученные после декодирования токена доступа методом validateToken. Таким образом, все API смогут получать эти данные и использовать их. Если что-то пойдет не так, то гард вернет сообщение о том, что пользователь не авторизирован.

2) authAdminGuard (./src/domain/auth/middleware/guard/auth.admin.guard.ts):

```
src > domain > auth > middleware > guard > TS auth.admin.guard.ts > ...
1   import { NextFunction, Request, Response } from 'express';
2   import authGuard from './auth.guard';
3
4   const authAdminGuard = async (req: Request, res: Response, next: NextFunction) => {
5     authGuard(req, res, () => {
6       if (req.user?.role !== 'admin') {
7         return res.status(401).send("Unauthorized - you are not signed in as admin");
8       }
9     });
10
11    next();
12  };
13  export default authAdminGuard;
14
```

Изменение данного гарда заключается в том, что теперь он сначала вызывает работу authGuard. Если authGuard отработает успешно, то authAdminGuard просто проверит через переданные данные пользователя в запрос, что у пользователя роль admin. Если роль действительно админ, то гард просто передаст запрос дальше на обработку. Если же у пользователя нет роли админа, то гард вернет ответ о том, что пользователь не является админом.

Также нужно изменить роутер AuthRouter (./src/domain/auth/auth.routes.ts):

```
22    protected override setupRouter(): void {
23      this.authRouter.post('/signin', (...args) => this.authController.signin(...args));
24      this.authRouter.post('/signup', (...args) => this.authController.signup(...args));
25      this.authRouter.get('/signup/verifyUserEmail', (...args) => this.authController.verifyUserEmail(...args));
26      this.authRouter.get('/signup/verifyAdminRole', (...args) => this.authController.verifyAdminRole(...args));
27      this.authRouter.use(authGuard);
28      this.authRouter.get('/payload', (...args) => this.authController.getUserPayload(...args));
29      this.authRouter.get('/logout', (...args) => this.authController.logout(...args));
30    }
31  }
32  export default AuthRouter;
33
```

Изменение будет заключаться только в том, что мы добавим новый маршрут получения данных авторизированного пользователя /payload с вызовом метода getUserPayload контроллера AuthController, привязав к нему гард authGuard, а также удалим старый маршрут для получения данных авторизированного пользователя.

Теперь остается немного изменить каждый контроллер в domain (./src/domain). Изменение будет незначительным – нужно просто везде, где мы получали пользователя типа User через сессию, заменить на получения пользователя типа IUserPayload из запроса. Пример:

```
const user = req.session.user as User;
```

Нужно заменить на следующее:

```
const user = req.user as IUserPayload;
```

Также нужно добавить в UserService (./src/domain/user/service/user.service.ts) метод, который позволит получать пользователя по его токену обновления. Это нужно, чтобы мы могли использовать этот метод для проверки наличия токена обновления в базе данных в созданном ранее промежуточном ПО refreshToken. Добавим метод getUserByRefreshToken:

```
154  public async getUserByRefreshToken(refreshToken: string): Promise<User> {
155    const user = await User.findOne({
156      where: {
157        refreshToken
158      }
159    });
160
161    if (!user) {
162      throw new NotFound(`User with refresh token: ${refreshToken} - is not found`);
163    }
164
165    return user;
166  }
```

Данный метод ищет переданный токен обновления в базе данных. Если такого токена нет, то вернется ошибка 404.

Также необходимо переопределить интерфейс обновления данных авторизации пользователя IUserUpdateAuth (./src/domain/auth/validation/interface/user.auth.update.interface.ts):

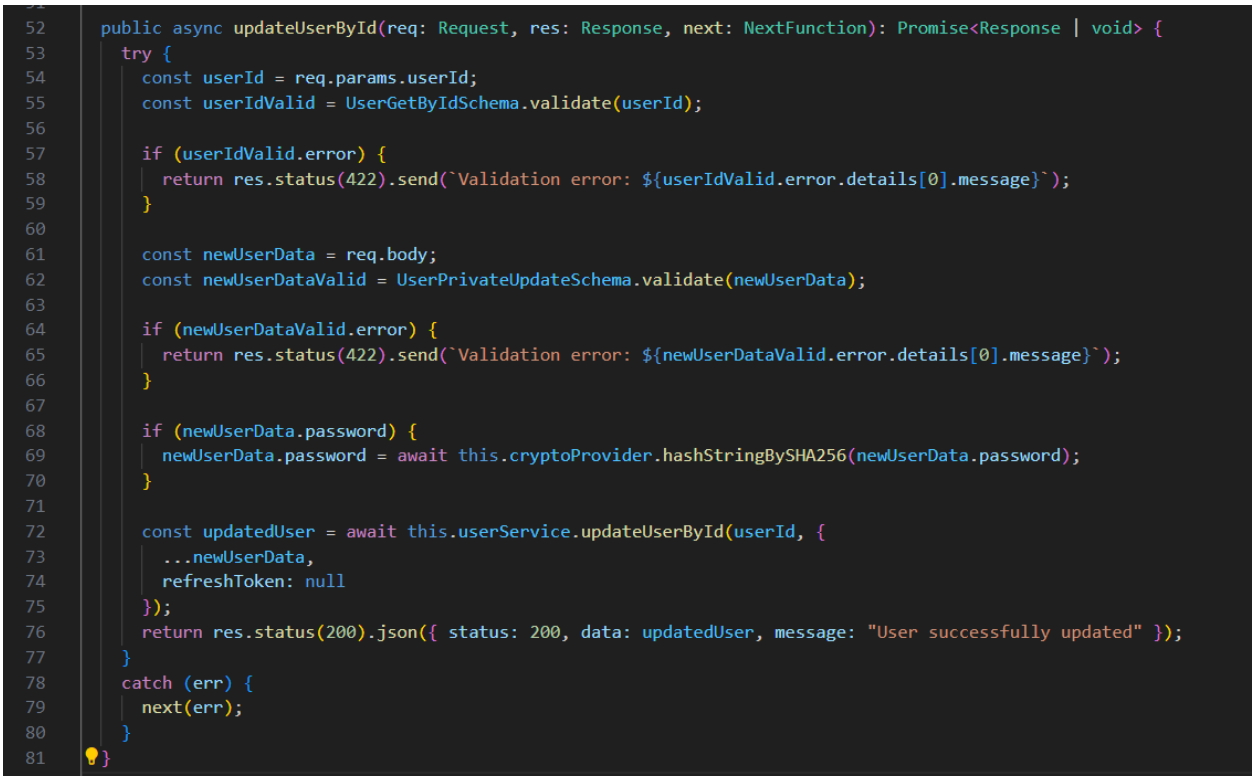
src > domain > auth > validation > interface > TS user.auth.update.interface.ts > ...

```
1  interface IUserUpdateAuth {
2    email?: string;
3    isActivated?: boolean;
4    refreshToken?: string | null;
5    verifToken?: string | null;
6  };
7  export default IUserUpdateAuth;
8
```

Мы добавили сюда дополнительное поле refreshToken. Это нужно, чтобы в методе signin мы могли записывать созданный токен доступа в базу данных.

Также нужно изменить методы updateUserById и updateUser в UserController (./src/domain/user/controller/user.controller.ts). В обоих методах можно изменить данные пользователя, которые содержатся в полезной нагрузке токенов. Соответственно, при изменении таких данных – нужно и изменить данные в полезной нагрузке, чтобы API работал с корректными данными. Сделаем это:

1) updateUserById:



```
public async updateUserById(req: Request, res: Response, next: NextFunction): Promise<Response | void> {
  try {
    const userId = req.params.userId;
    const userIdValid = UserGetByIdSchema.validate(userId);

    if (userIdValid.error) {
      return res.status(422).send(`Validation error: ${userIdValid.error.details[0].message}`);
    }

    const newUserData = req.body;
    const newUserDataValid = UserPrivateUpdateSchema.validate(newUserData);

    if (newUserDataValid.error) {
      return res.status(422).send(`Validation error: ${newUserDataValid.error.details[0].message}`);
    }

    if (newUserData.password) {
      newUserData.password = await this.cryptoProvider.hashStringBySHA256(newUserData.password);
    }

    const updatedUser = await this.userService.updateUserById(userId, {
      ...newUserData,
      refreshToken: null
    });
    return res.status(200).json({ status: 200, data: updatedUser, message: "User successfully updated" });
  }
  catch (err) {
    next(err);
  }
}
```

-Этот метод используют админы, чтобы изменять пользователей по их id. Соответственно, если админ изменит какие-либо данные пользователя, которые содержатся в полезной нагрузке, то в таком случае – в токенах пользователя будут содержать старые данные. Именно поэтому здесь при обновлении пользователя – мы обнуляем его refreshToken. Если refreshToken обнулится в базе данных, то пользователь не сможет пользоваться функционалом приложения. В таком случае, пользователю придется заново авторизироваться в системе, чтобы были созданы новые токены на основе обновленных данных пользователя

2) updateUser:

-Этот метод используют пользователи, чтобы изменять свои данные. В этом методе пользователи могут изменить свое имя, свою почту и свой пароль. При этом в полезной нагрузке хранятся имя и почта. Из-за этого необходимо доработать данный метод.

Данный метод был изменен так, чтобы проверялось – хочет ли пользователь изменить свое имя. Если хочет, то в таком случае создается новая полезная нагрузка, в которой все данные сохраняются, но меняется имя пользователя на новое. Дальше создаются новые токены доступа и обновления на основе

обновленной полезной нагрузке. После создаются новые куки httpOnly для новых токенов обновления и доступа. Также у пользователя обновляется в базе данных поле refreshToken на обновленное значение токена обновления. Тем самым, мы перезаписываем все токены пользователя на новые, которые содержат обновленные данные пользователя. И при этом пользователю не нужно заново заполнять форму авторизации при обновлении своего имени.

Однако здесь добавлено обновление только для имени. Но пользователь также может поменять свой email, который тоже содержится в полезной нагрузке. Данный метод не будет обновлять email в токенах, так как в этом методе email никогда не обновляется, поскольку если пользователь захотел изменить свой email – ему сначала нужно его подтвердить. Поэтому email будет обновляться в методе verifyUserEmail

3) verifyUserEmail:

-Этот метод срабатывает, когда пользователь подтверждает свою новую почту, которую он хотел себе установить в updateUser.

Метод был изменен так, чтобы создавалась новая полезная нагрузка, содержащая обновленную почту, а также создавались новые токены доступа и обновления на основе обновленной полезной нагрузки. Дальше изменяется сам пользователь – помимо изменения почты и токена верификации, также изменяется его токен обновления на новый. После создаются новые куки httpOnly на основе новых токенов доступа и обновления. Таким образом, мы изменяем почту пользователя, при этом обновляя все его токены так, что пользователю не придется при обновлении своей почты заново заполнять форму авторизации.

Тестирование:

Перед тем, как переходить к тестам – сначала нужно изменить соответствующим образом Swagger-документацию (./src/middleware/swagger/swagger.yaml):

-Просто удалим ненужные маршруты и добавим новые. Также нужно изменить схему авторизации с сессионной куки на JWT:

```
 999  components:
1000    securitySchemes:
1001      BearerAuth:
1002        type: http
1003        scheme: bearer
1004        bearerFormat: JWT
```

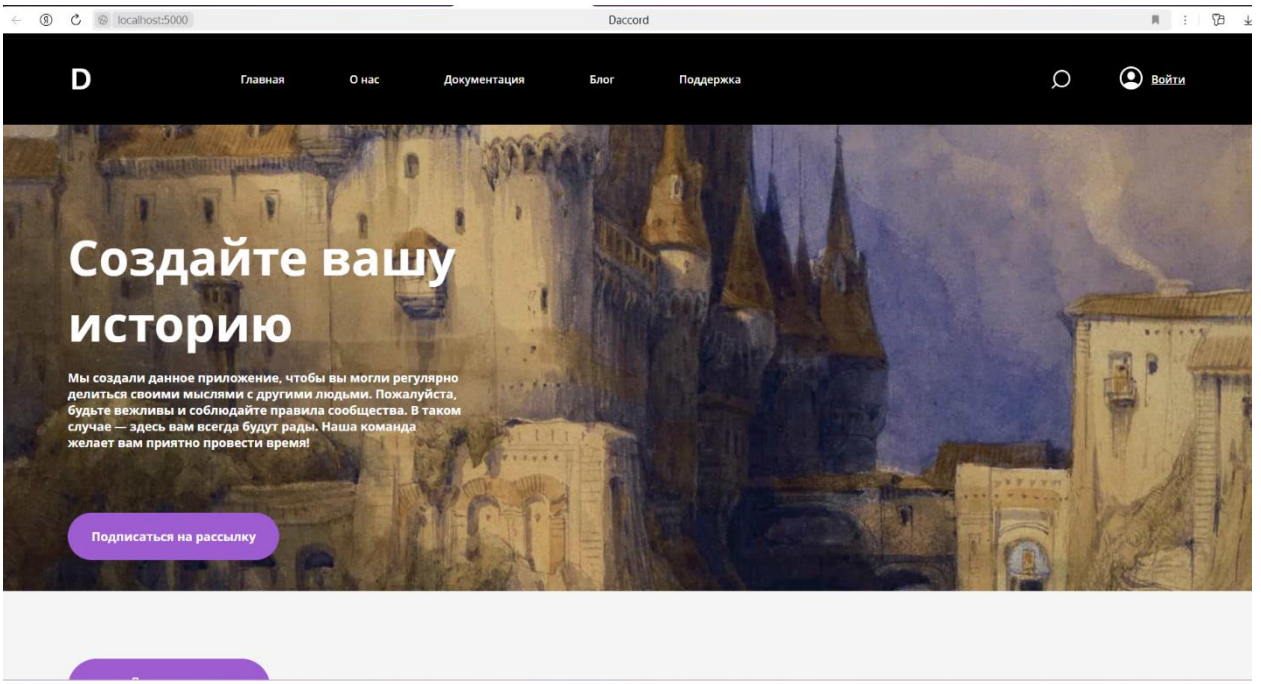
Также нужно везде, где требуется авторизация, указать на новую схему авторизации JWT:

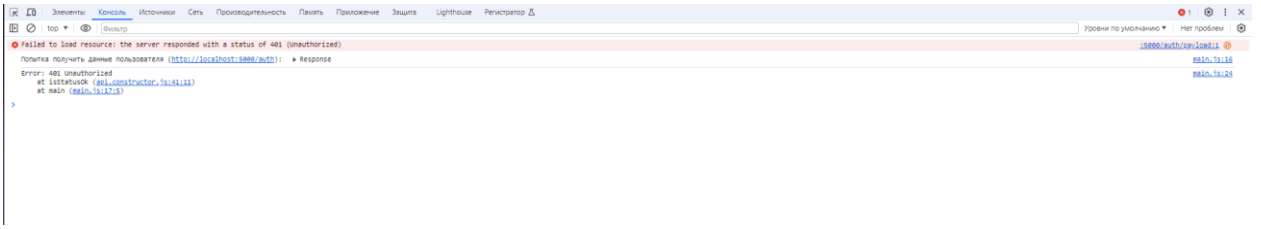
```
security:
  - BearerAuth: []
```

После изменения Swagger-документации, можно переходить к тестам. Выполним тесты как на клиенте, так и в Swagger:

1) **Тесты на клиенте:**

-Сначала просто зайдем на страницу:

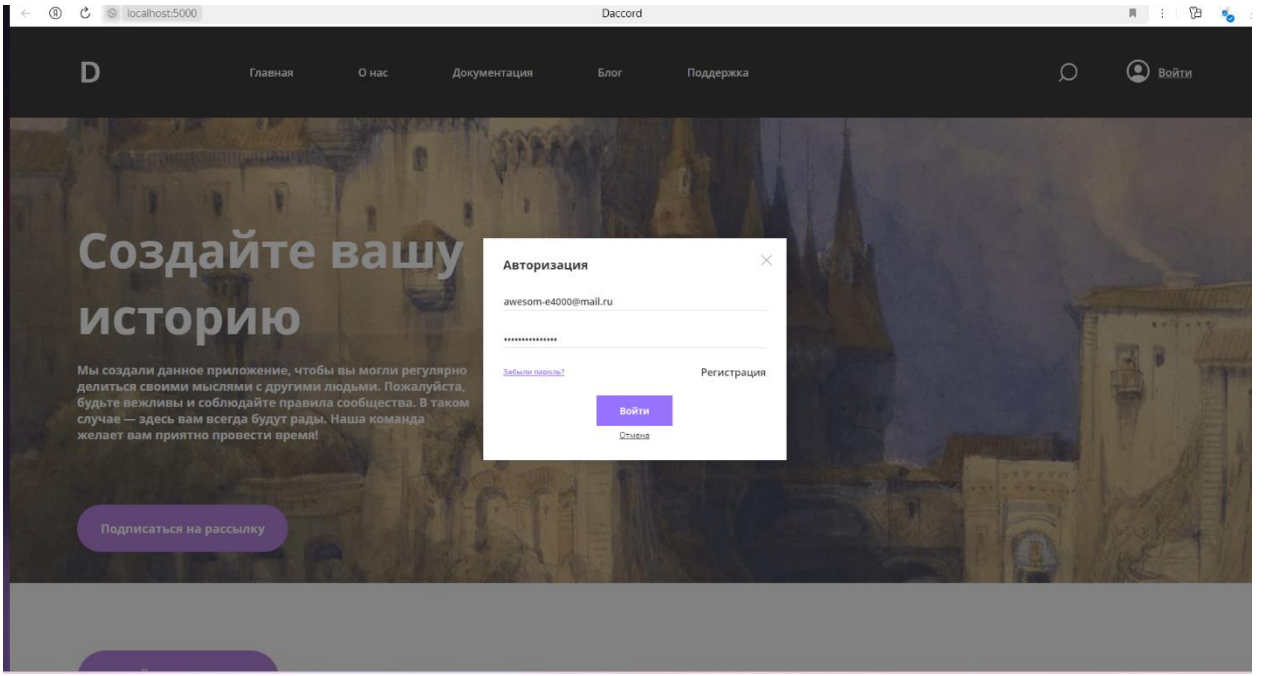


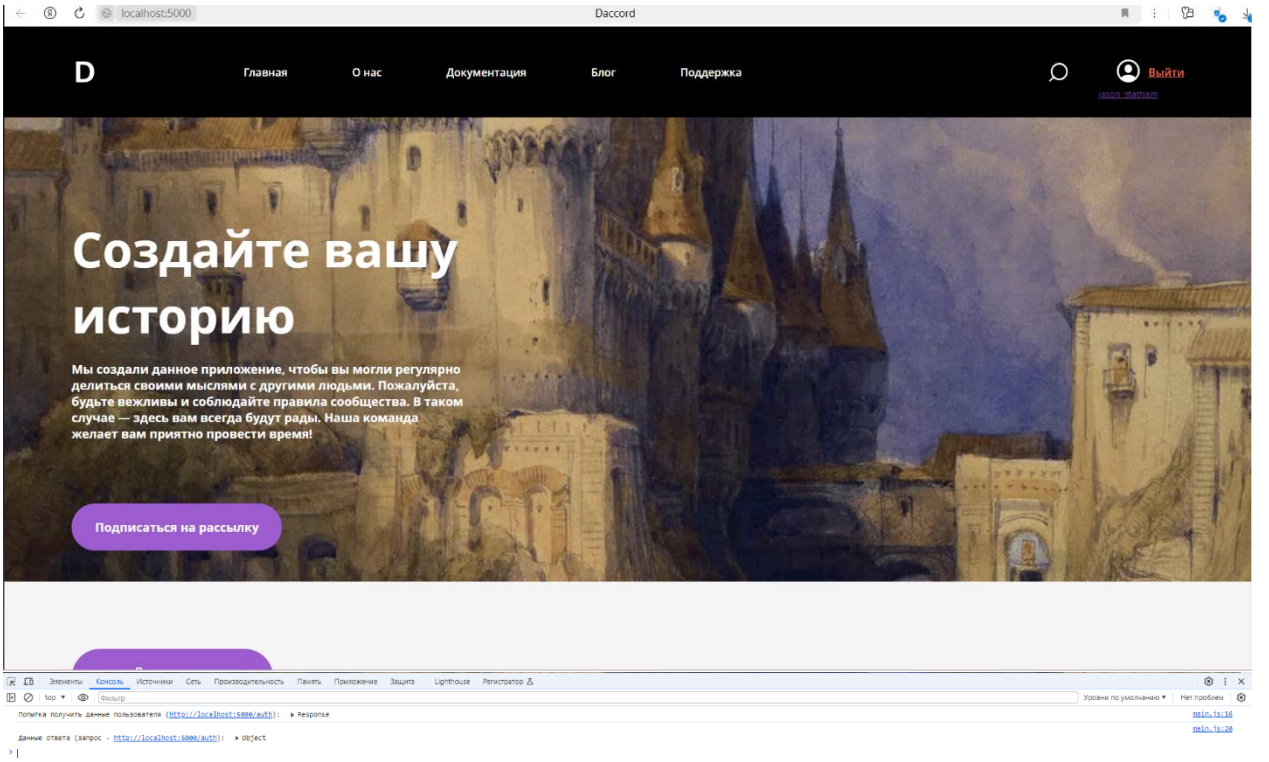


Как можно заметить – приложение попыталась получить данные пользователя, но не смогло, так как пользователь еще не авторизировался. Попытаемся получить всех пользователей, а также свои посты через соответствующие кнопки:

Вернулся ответ 401 – мы ничего не смогли получить, так как мы еще не авторизировались.

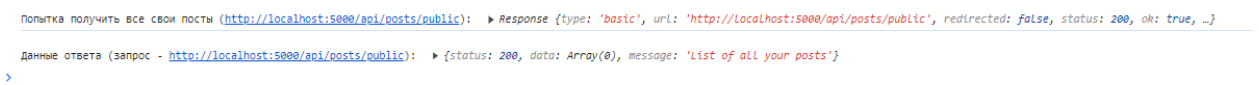
Авторизируемся в системе как пользователь:





Теперь мы авторизовались в системе. После авторизации – страница перезагрузилась и теперь клиент смог получить данные авторизированного пользователя.

Попробуем получить все свои посты (через кнопку “Получить все свои посты”):

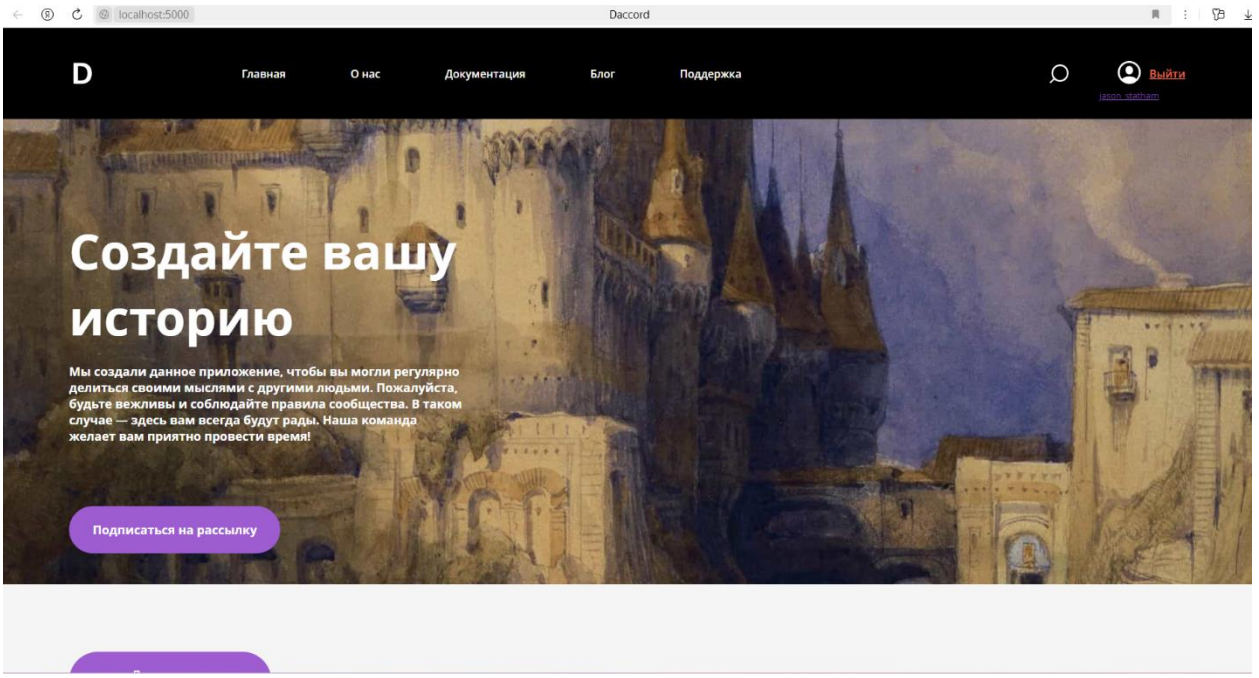


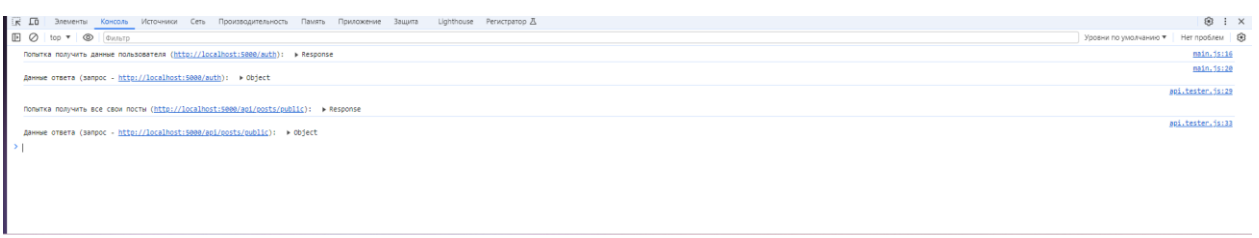
Посты мы успешно получили, так как авторизовались.

Теперь убедимся, что мы не можем получить всех пользователей (кнопка “Получить всех пользователей”), так как вошли в систему как обычный пользователь, а не админ:

Мы получили ответ 401, а это значит, что все верно.

Теперь закроем страницу и опять зайдем на нее, чтобы убедиться, что мы по-прежнему авторизованы (даже после закрытия страницы или браузера):

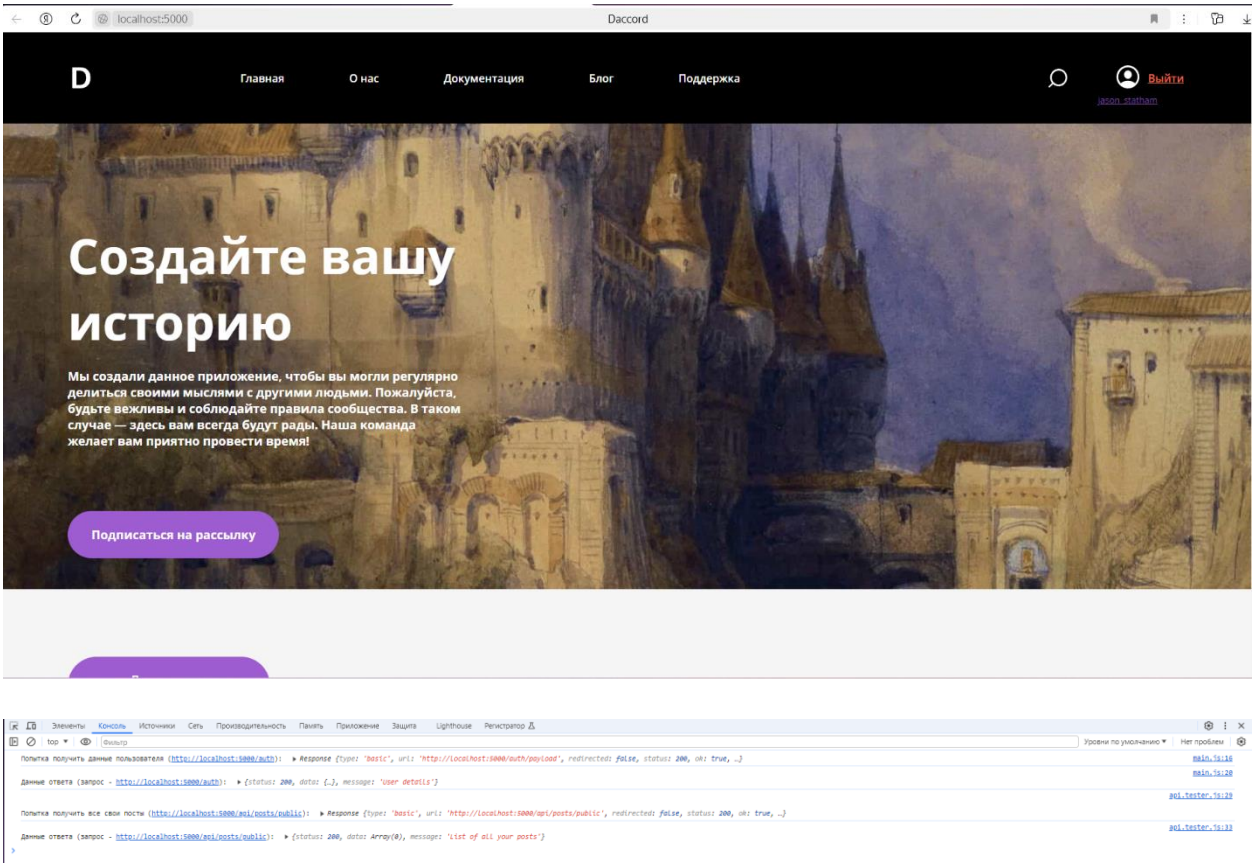




Как можно заметить – все запросы выполнились успешно. Мы по-прежнему авторизованы несмотря на то, что закрыли страницу и еще раз зашли.

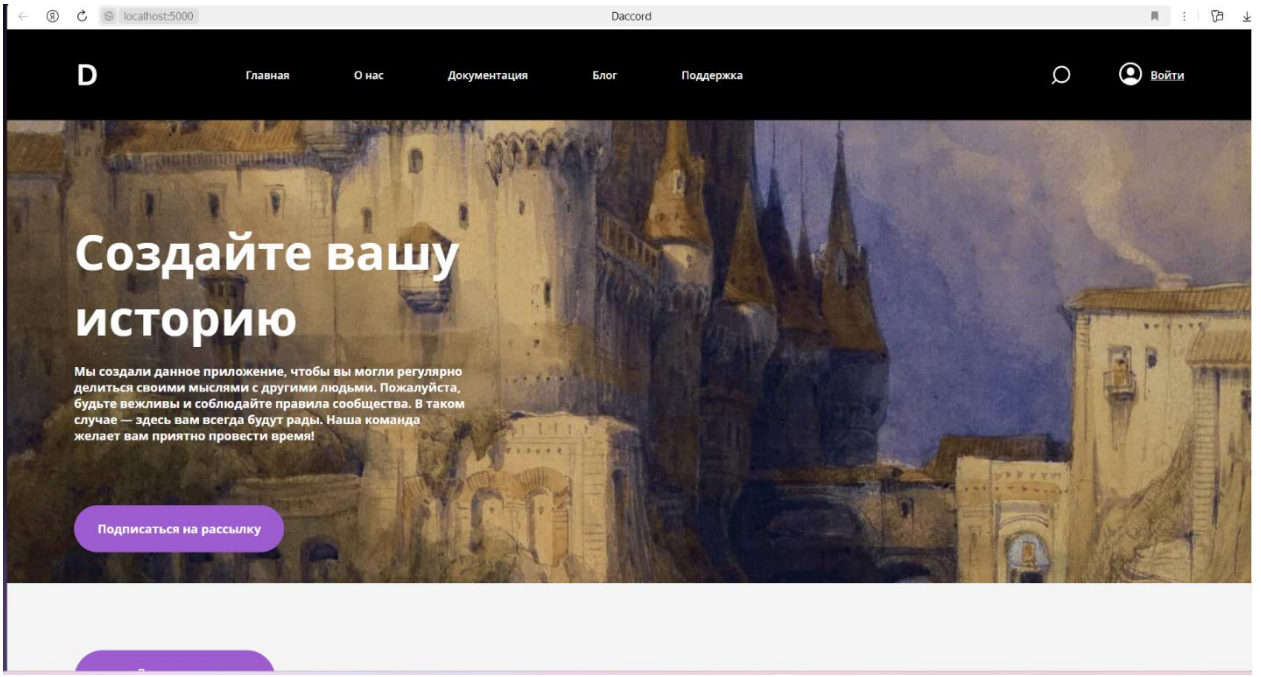
Теперь убедимся, что сервер обновляет токен доступа, если он истек. Для этого закроем страницу, подождем 15 минут (столько живет токен доступа), а после опять зайдем на страницу. Несмотря на это – мы по-прежнему должны будем остаться авторизованными – все запросы должны выполняться без ошибок.

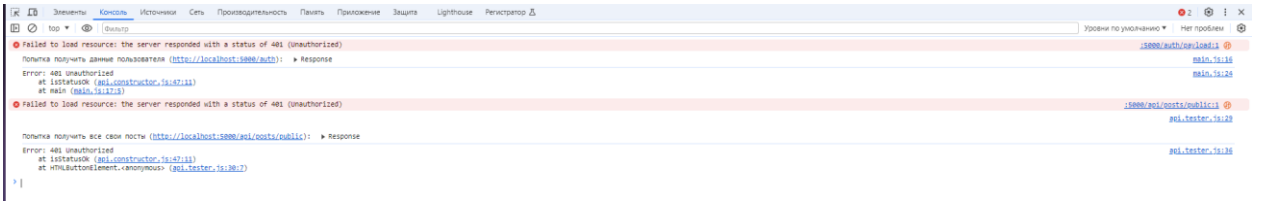
Заходим на страницу по истечению 15-ти минут:



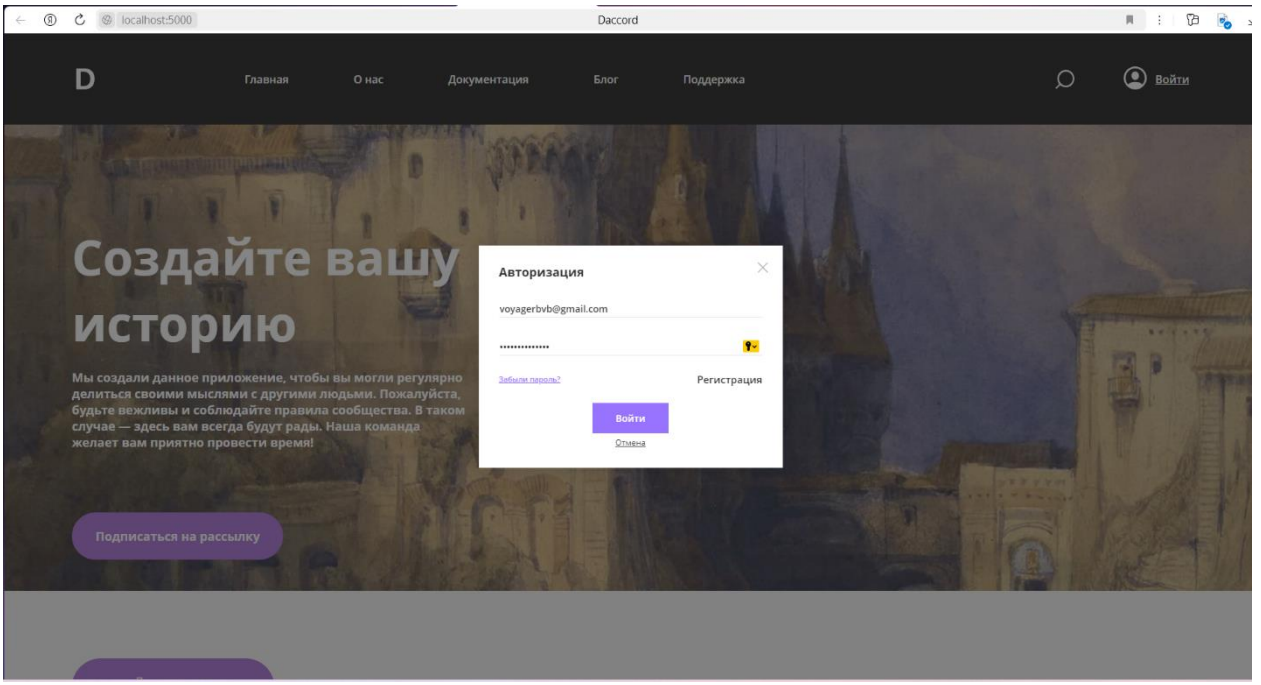
Как можно заметить – все запросы выполнились без ошибок. Это означает, что сервер правильно обновляет токены доступа.

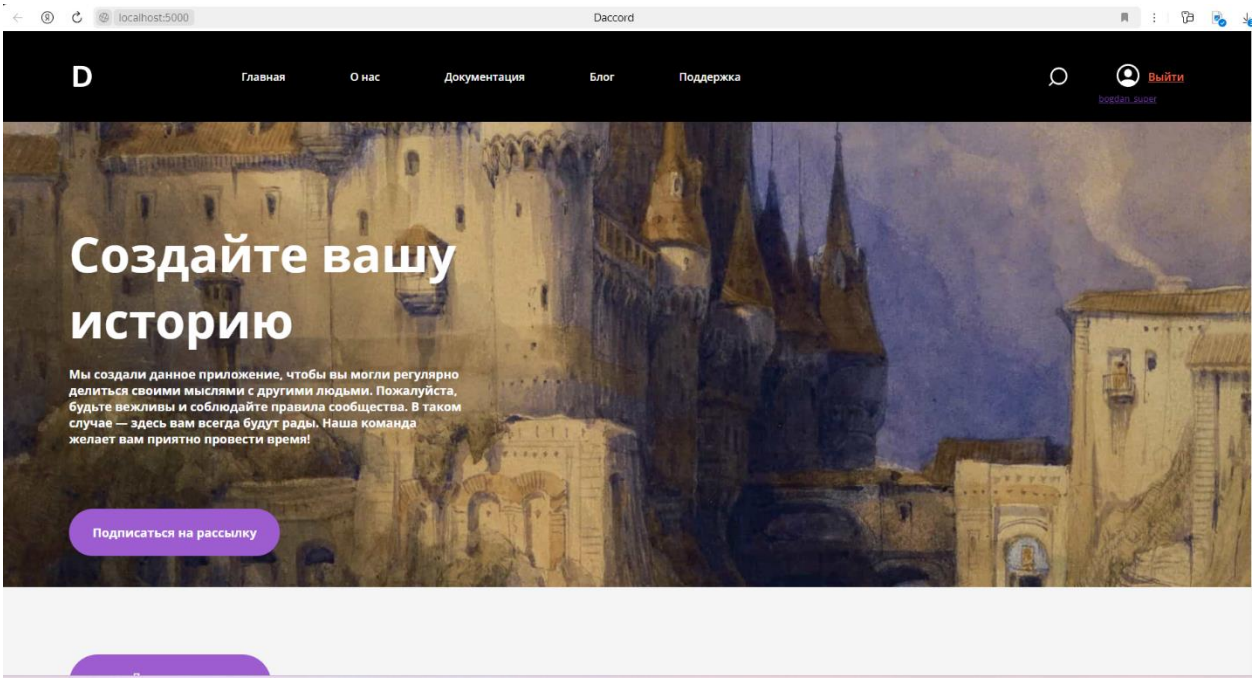
Теперь войдем в систему как админ и убедимся, что мы можем в таком случае получать всех пользователей. Для этого – выйдем из аккаунта, нажав кнопку выйти:

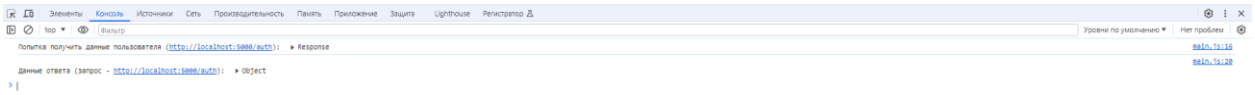




Мы вышли из аккаунта. Теперь авторизируемся как admin:





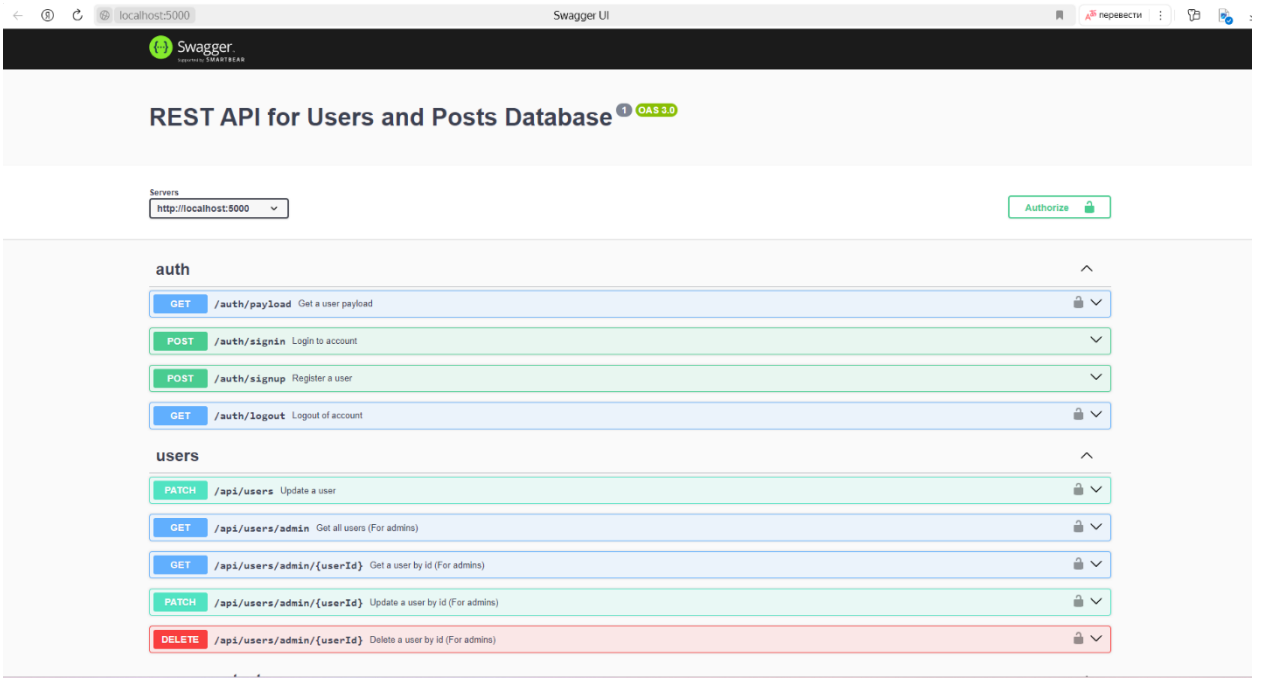


Мы вошли как админ. Теперь попробуем получить всех пользователей, а также все посты:

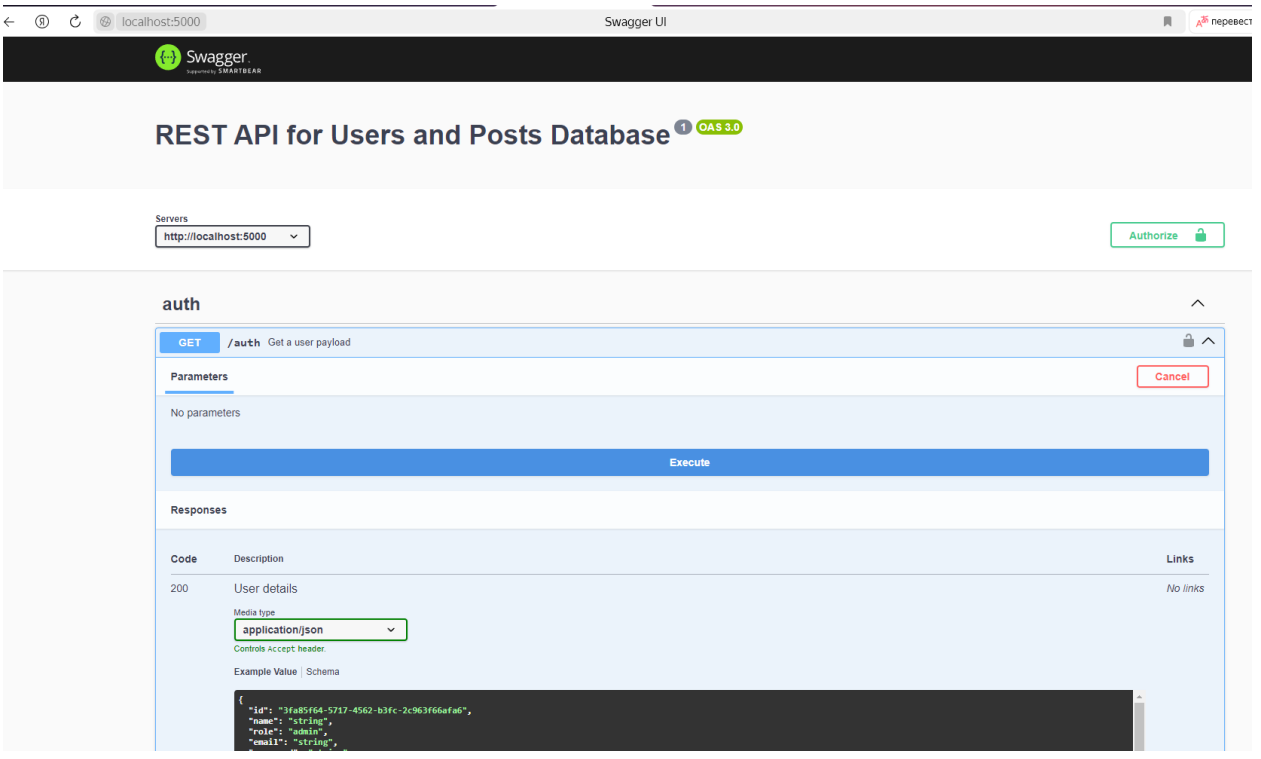
Все запросы успешно выполняются. То есть мы можем воспользоваться методом для админов (получить всех пользователей), если вошли как админ.

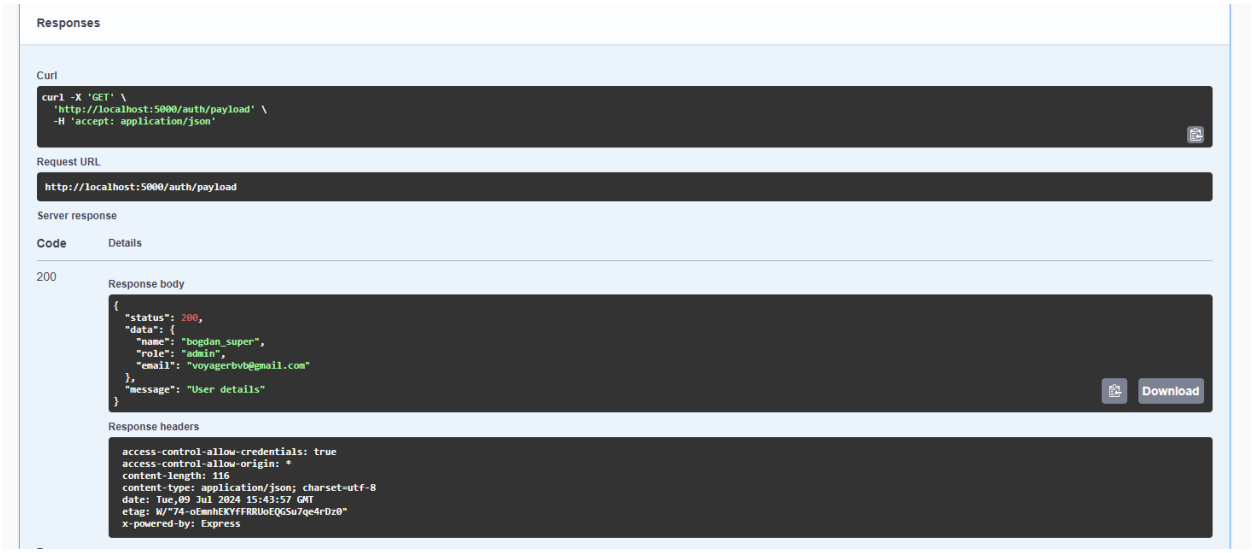
2) **Тесты в Swagger:**

Для начала просто зайдем на Swagger (http://localhost:5000/api) и попытаемся выполнить любой метод, где требуется авторизация. Так как мы сейчас тестировали на клиенте, то мы уже авторизированы, а значит не должно быть никакой ошибки. То есть мы спокойно без авторизации должны выполнять любой метод, требующий авторизации:



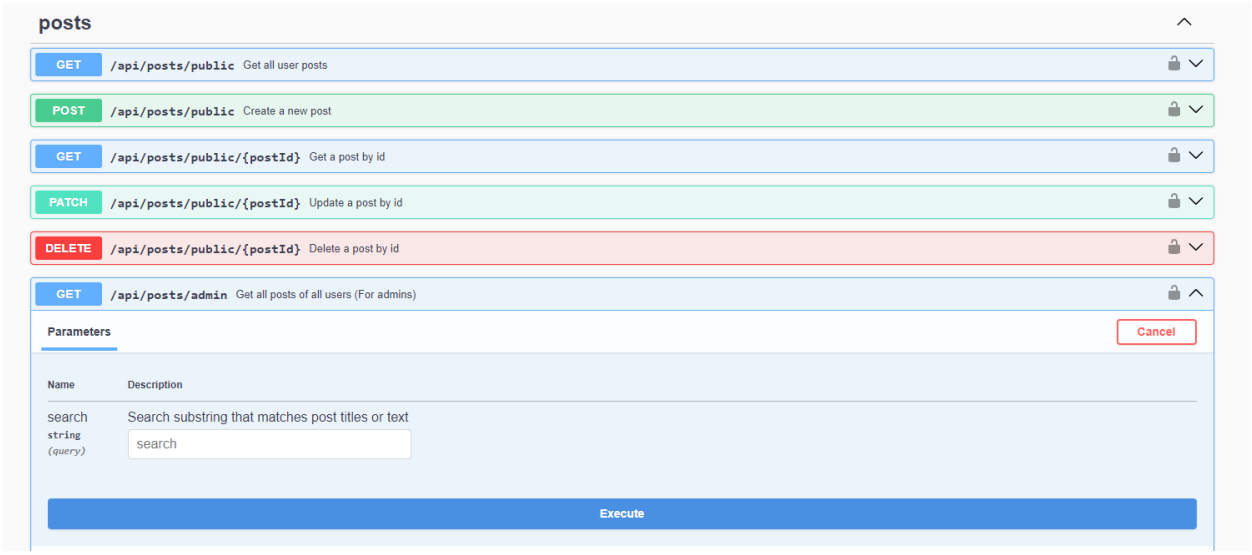
Мы зашли на страницу. Выполним GET-метод для получения данных пользователя (GET /auth/payload – Get a user payload). Метод должен вернуть нам наши данные, так как мы по-прежнему авторизированы из-за тестов на клиенте:





Как можно заметить – мы действительно получили наши данные, так как мы по-прежнему авторизированы.

Выполним ради тестов еще один метод. Например, Get-метод получения всех постов всех пользователей (GET /api/posts/admin – Get all posts of all users):



Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts/admin' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/admin
```

Server response

Code	Details
200	Response body


```
{
  "status": 200,
  "data": [
    {
      "id": "1edfc759-927a-4bba-8337-efe74706531a",
      "title": "Властелин колец - книги против фильмов",
      "access": "public",
      "createdAt": "2024-07-02T10:17:25.753Z",
      "updatedAt": "2024-07-02T10:17:56.970Z",
      "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",
      "rating": 0,
      "tags": [
        "фильмы",
        "книги",
        "Властелин Колец"
      ],
      "authorId": "75db0cf8-32ba-4f04-8095-7dca55052ea0",
      "author": {
        "id": "75db0cf8-32ba-4f04-8095-7dca55052ea0",
        "name": "ingaale",
        "role": "user",
        "email": "nozdryakovbogdanm3112@mail.ru",
        "password": "dbd10d4c4cf81de9c54ce60b9b0f98a3b84a07d1aa4c0bbaf6c703072ecbd800",
        "isActivated": true,
        "refreshToken": null,
        "verifToken": null,
        "createdAt": "2024-07-02T09:35:03.550Z",
        "updatedAt": "2024-07-02T09:36:52.684Z",
```


Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 1865
content-type: application/json; charset=utf-8
date: Tue,09 Jul 2024 15:47:07 GMT
etag: W/"749-9nU3380p0FFkw4dzbnD2AF6pkFs"
keep-alive: timeout=5
x-powered-by: Express
```


Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts/admin' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/admin
```

Server response

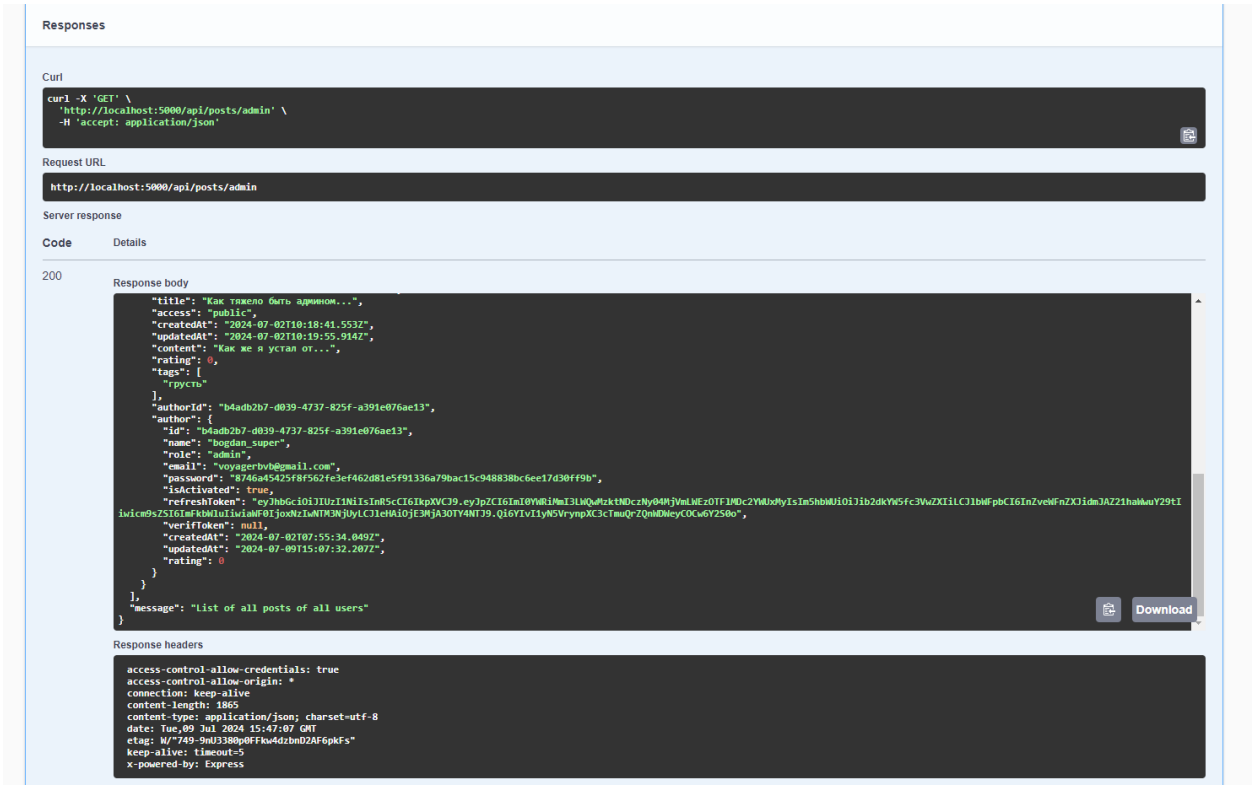
Code	Details
200	Response body


```
        "verifToken": null,
        "createdAt": "2024-07-02T09:35:03.550Z",
        "updatedAt": "2024-07-02T09:36:52.684Z",
        "rating": 0
      }
    },
    {
      "id": "7423227b-7b33-411a-b0de-0aabc9e75859",
      "title": "Как тяжело быть админом...",
      "access": "public",
      "createdAt": "2024-07-02T10:18:41.553Z",
      "updatedAt": "2024-07-02T10:19:55.914Z",
      "content": "Как же я устал от...",
      "rating": 0,
      "tags": [
        "грусть"
      ],
      "authorId": "b4adb2b7-d039-4737-825f-a391e076ae13",
      "author": {
        "id": "b4adb2b7-d039-4737-825f-a391e076ae13",
        "name": "bogdan_super",
        "role": "admin",
        "email": "voyagerbvb@gmail.com",
        "password": "8746a45425f8f562fe3ef462d81e5f91336a79bac15c948838bc6ee17d30ff9b",
        "isActivated": true,
        "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6ImI0YWRiMmI3LWQwMzktNDczNy04MjVmLWEzOTFlMDc2YWUxMyIsIm5hbWUiOiJib2dkYW5fc3VwZXIiLCJlbWFpbCI6InZveWFnZXJidmJAZ21haWwuY29tIiwicm9sZSI6ImFkbWluIiwiaWF0IjoxNzIwNTM3NjUyLCJleHAiOjE3MjA3OTY4NTJ9.Qi6YIvI1yN5VrynpXC3cTmuQrZQnWDWeyCOCw6Y2S0o",
        "verifToken": null,
```


Download

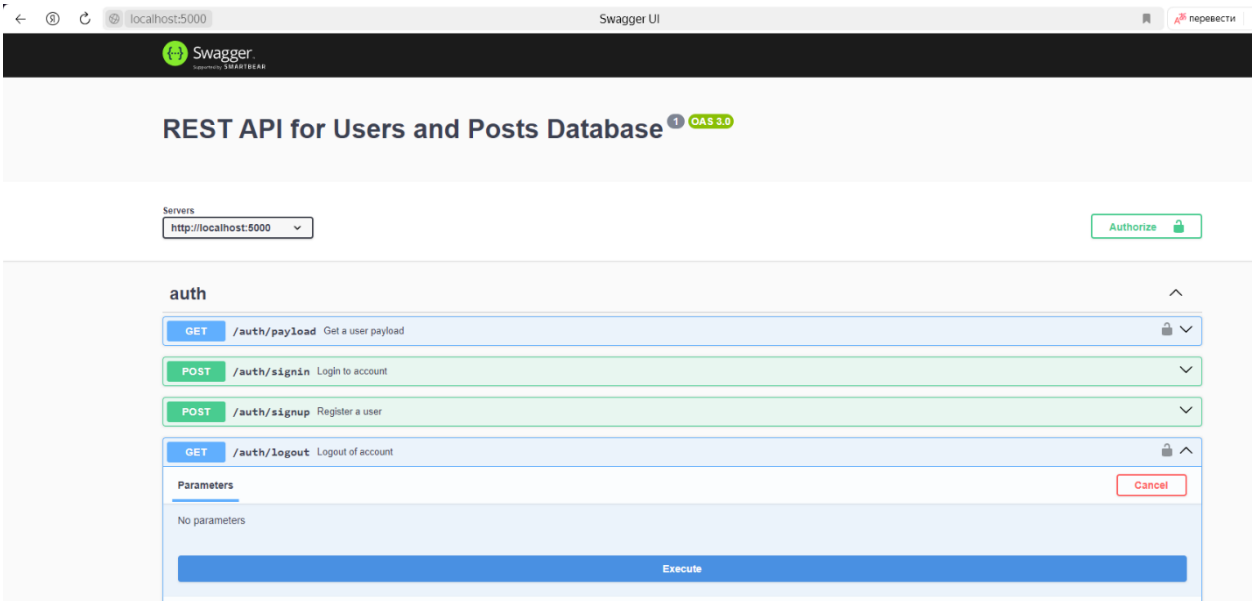
Response headers

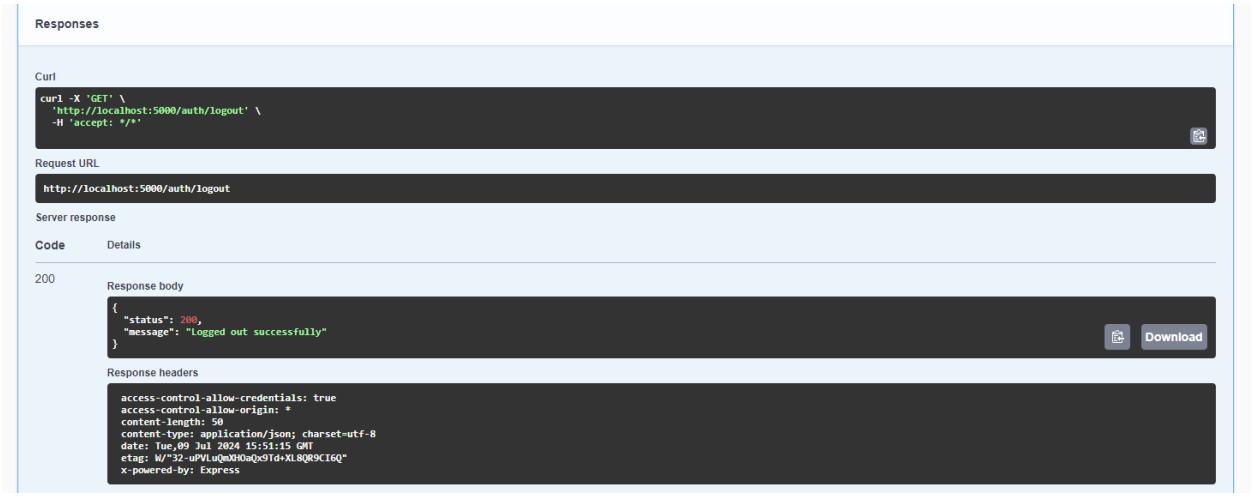

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 1865
content-type: application/json; charset=utf-8
date: Tue,09 Jul 2024 15:47:07 GMT
etag: W/"749-9nU3380p0FFkw4dzbnD2AF6pkFs"
keep-alive: timeout=5
x-powered-by: Express
```



Мы успешно выполнили метод.

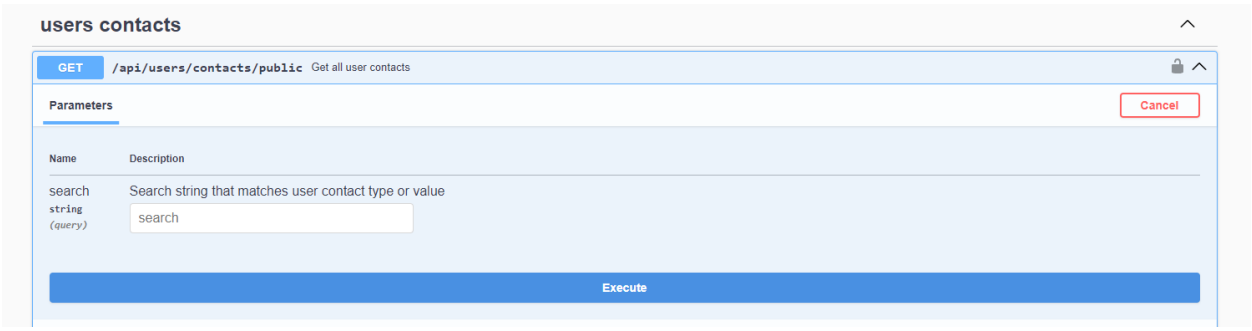
Теперь выйдем из аккаунта, выполнив GET-метод для выхода из аккаунта (GET /auth/logout – Logout of account):





Метод выполнился успешно, значит мы должны были выйти из аккаунта.

Попробуем выполнить любой метод, требующий авторизации, например, GET-метод для получения всех контактов пользователя (GET /api/users/contacts/public – Get all user contacts):



Нам вернули сообщение, что мы не авторизированы. Значит все правильно – метод logout отработал верно.