**Содержание**

[Ключевые слова и определения 2](#_Toc103818936)

[Аннотация 3](#_Toc103818937)

[Введение 4](#_Toc103818938)

[JSON Web Token 5](#_Toc103818939)

[Общая информация 5](#_Toc103818940)

[Типы токенов 6](#_Toc103818941)

[Схема взаимодействия клиента и сервера 6](#_Toc103818942)

[Реализация аутентификации с помощью JSON Web Token 8](#_Toc103818943)

[Используемый инструментарий 8](#_Toc103818944)

[Определение эндпоинтов приложения 8](#_Toc103818945)

[Модель базы данных 9](#_Toc103818946)

[Определение обработчиков 9](#_Toc103818947)

[Механизм генерации и валидации токенов 16](#_Toc103818948)

[Взаимодействие клиента и сервера 21](#_Toc103818949)

[Заключение 23](#_Toc103818950)

[Библиографический список 24](#_Toc103818951)

Ключевые слова и определения

**Аутентифика́ция** (англ. authentication < греч. αὐθεντικός [authentikos] «реальный, подлинный» < αὐτός [autos] «сам; он самый») — процедура проверки подлинности, например:

1. проверка подлинности пользователя путём сравнения введённого им пароля (для указанного логина) с паролем, сохранённым в базе данных пользовательских логинов;
2. подтверждение подлинности электронного письма путём проверки цифровой подписи письма по открытому ключу отправителя;
3. проверка контрольной суммы файла на соответствие сумме, заявленной автором этого файла.

**DoS** (аббр. англ. Denial of Service «отказ в обслуживании») — хакерская атака на вычислительную систему с целью довести её до отказа, то есть создание таких условий, при которых добросовестные пользователи системы не смогут получить доступ к предоставляемым системным ресурсам (серверам), либо этот доступ будет затруднён.

Аннотация

В данном докладе рассматривается современный подход к построению системы аутентификации на сервере с помощью JSON Web Token стандарт, программная реализация данного стандарта на языке программирования Golang [2], а также определение проблем, возникающих при данном способе аутентификации и возможных доработках в отношении авторизации на множестве устройствах.

Введение

В современном мире использование онлайн-сервисов стало неотъемлимой частью жизни современного человека: онлайн магазины, онлайн заказы, сервисы по доставке еды, сервисы такси и так далее.

Социальные сети, мессенджеры, платформы для ведения личных блогов и микроблогов безостановочно набирают популярность.

В настоящее время на уровне государства разрабатываются различные сервисы для облегчения бюрократической волокиты и регулирования очередей, чтобы пользователи могли спокойно пройти интересующую их процедуру или получить определённые государственные услуги.

Однако на ряду с возрастающей популярностью интернет сервисов различной направленности неизбежно встаёт вопрос об улучшении существующих методов аутентификации для обеспечения безопасности пользовательских данных. С развитием программных средств, сервисов, интернет-технологий возрастают и риски хакерских атак, взломов, несанкционированного получения пользовательских данных.

В настоящей теме данного доклада рассматривается аутентификация как способ обеспечения безопасности пользовательских данных и применение стандарта JSON Web Token (JWT) для обеспечения безопасного доступа к онлайн сервисам после авторизации пользователя.

JSON Web Token (JWT) [3] — это открытый стандарт (RFC 7519) для создания токенов доступа, основанный на формате JSON. Используется для передачи данных на сервер с автоматической аутентификацией конкретного пользователя. Токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в дальнейшем использует данный токен для подтверждения своей личности, а также обновления срока жизни токенов (access token).

Безусловно, аутентификация не решает все проблемы с безопасностью онлайн сервисов, однако она способствует существенному ограничению разнообразия DoS-атак на сервера, поскольку средствами middleware на сервере имеется возможность проверять токен доступа и отключать канал связи, однако свободные формы в таком случае (например, форма регистрации и авторизации) будут доступны для атаки на сервер (http-флуд). Помимо ограничения вариантов атак на сервер аутентификация также решает проблему взлома пользователя: достаточно сложно получить доступ к аккаунту пользователя, имеющего только зашифрованный токен доступа на клиенте, и даже в случае получения токена доступа злоумышленником имеется возможность лишить его доступа, поскольку время жизни токена может быть завершена, а токена обновления у злоумышленика может не оказаться, однако рано или поздно и время жизни токена обновления также закончится.

JSON Web Token

Общая информация

Структура JWT-токена представляет из себя закодированную строку, которая состоит из трёх частей [1] (см. рис. 1):

1. header – заголовок токена, содержащий общую информацию
2. payload – закодированные полезные данные токена (роль пользователя, id, и так далее)
3. signature – сигнатура токена (подпись)

C:\Users\DNS\Downloads\Untitled Diagram.drawio (1).png

Рисунок 1 – Структура JWT-токена

Header JWT-токена содержит информацию о том, каким образом должна вычисляться JWT подпись. Header представляет собой JSON объект, который выглядит следующим образом:

header = { "alg": "HS256", "typ": "JWT"}

Поле **typ** обозначает тип заголовка, в данном случае – заголовок JWT-токена.

Поле **alg** определяет алгоритм хэширования. Данный алгоритм используется при создании подписи. HS256 – это HMAC-SHA256, для его вычисления потребуется один секретный ключ. Помимо HS256 может быть использован алгоритм RS256, однако в отличии от HS256 он создаёт два ключа – публичный и приватный. С помощью приватного ключа создаётся подпись, а с помощью публичного ключа проверяется подлинность подписи.

Payload — это полезные данные, которые хранятся внутри JWT. Эти данные также называют JWT-claims (заявки). В примере, который рассматриваем мы, сервер аутентификации создает JWT с информацией об id пользователя — userId.

Payload – это полезные данные, которые хранятся внутри JWT-токена. Такие данные называют также JWT-claims (заявки).

Пример содержимого payload может быть представлен следующим образом:

payload = {

"exp": 1652887832,

"iat": 1652884232,

"users\_id": "b3375eb2-e11b-4a11-ae05-b75dd29e9a35",

"roles\_id": "34a875c8-d2d6-4df5-9478-ab2aaf9617cd"

}

Существует список Standart JWT-claims, в которые могут входить следующие данные:

1. exp – поле, определяющее время жизни токена в формате Unix Time
2. iat – поле, определяющее время создания токена в формате Unix Time

Для получения сигнатуры JWT-токена необходимо задание секретного ключа в приложении, которое осуществляет шифрование токена.

Секретный ключ должен знать только серверное приложение, работающее по схеме JWT-аутентификации.

Секретный ключ используется при кодировании заголовка и полезных данных, в результате чего получается сигнатура JWT-токена (сигнатура шифрует header и payload JWT-токена).

Сигнатура необходима для обеспечения безопасности токена, так как с помощью сигнатуры токена можно убедиться в том, что он не был подделан.

Типы токенов

Каждый JWT-токен имеет срок своей жизни и через какое-то время токен становится не валидным и использовать его для подтверждения своей личности не представляется возможным. Такой подход необходим для обеспечения безопасности пользователя. Например, если злоумышленник украл JWT-токен, а у токена срок жизни 5-10 минут, то сервисом злоумышленник пользоваться сможет не более отведённого времени, а дальше – необходимо ввести авторизационные данные.

Существует два типа токенов:

1. Access token – токен доступа, имеет малое время жизни (от 5 минут до 1-го часа). Время жизни данного токена зависит от степени защищённости данных, расположенных на серверах и может варьироваться. Используется для доступа к сервису. В веб-приложениях обычно хранится в localstorage.
2. Refresh token – токен обновления, имеет по сравнению с access-токеном большее время жизни (от 2 недель до 1-2 месяца). С помощью данного токена перезаписывается токен доступа (обновляется). В веб-приложениях обычно хранится в httpOnly cookie.

Схема взаимодействия клиента и сервера

Общая схема взаимодействия сервера и клиента при использовании JWT-токена в описательной форме содержит следующие пункты:

1. Пользователь заходит в клиентское приложение и проходит процедуру авторизации при которой вводит свой email-адрес и пароль.
2. В теле запроса на сервер будут указаны данный email и пароль, которые отправляются на сервер аутентификации.
3. Сервер аутентификации проверяет пользовательские данные, обрабатывает ошибки и в случае успешной процедуры проверки авторизационных данных генерирует пару токенов (access и refresh).
4. Acess-токен сохраняется в localstorage, а refresh-токен сервер должен установить в httpOnly cookie.
5. В приложении пользователь авторизовался, и он может пользоваться услугами сервиса, однако для отправки каждого запроса сервису необходимо встраивать токен доступа в каждое тело запроса в заголовок Authorization.
6. На сервере при обработке любого запроса авторизованного пользователя происходит проверка токена доступа, который был получен из заголовка Authorization.
7. Если после проверки токена сервер подтвердил его валидность, то сервер возвращает статус код 200 и необходимую пользователю информацию.
8. В случае, если токен не валиден, то сервер сразу возвращает статус код 401 (пользователь не атворизован). В таком случае необходимо предусмотреть на клиенте перехватчик (интерцептор) 401 кода и отправлять запрос серверу аутентификации на обновление токена доступа с помощью токена обновления, после получения статус кода 401.
9. При получении токена обновления сервером, токен доступа проходит процедуру обновления, после которой сервер возвращает новую пару access и refresh-токенов
10. После получения access и refresh-токенов в интерцепторе происходит дублирование предыдущего не успешного запроса со статус-кодом 401 с новым токеном доступа.

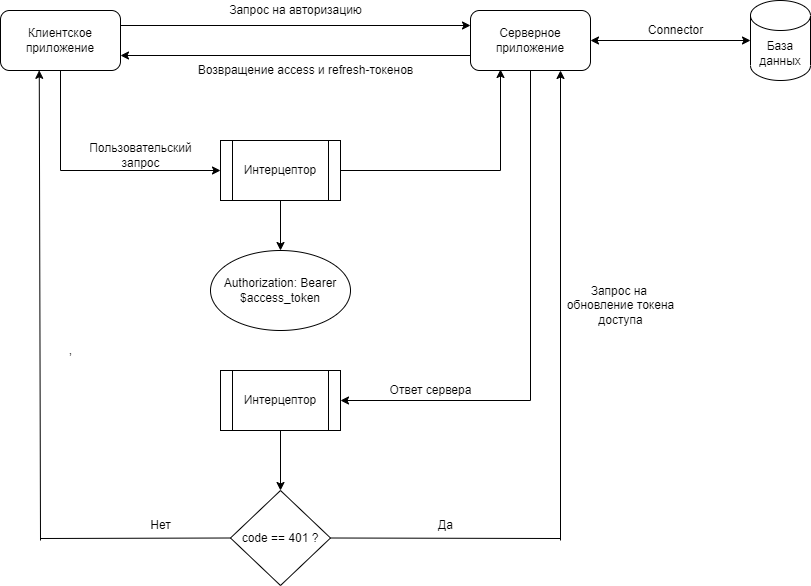


Рисунок 2 – Общая схема взаимодействия сервера и клиента

Реализация аутентификации с помощью JSON Web Token

Далее будет представлена краткая информация по реализации аутентификации на сервере с помощью JSON Web Token с помощью языка программирования Golang (Go).

Используемый инструментарий

Для реализации серверного приложения будет использован редактор кода Visual Studio Code с надстройками для работы с Go.

Для взаимодействия с внешним API сервера будет использована программа Postman, позволяющая отправлять запросы и получать ответ в любом доступном формате (в данном примере будет использован формат JSON).

Для создания серверного приложения использовался фреймворк Gin-Gonic. Соответственно реализация эндпоинтов приложения будет представлена в соответствии с концепциями данного фреймворка.

Определение эндпоинтов приложения

На рисунке 3 представлено дерево эндпоинтов серверного приложения, которые будут рассмотрены в рамках реализации аутентификации через JSON Web Token.

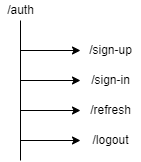


Рисунок 3 – Дерево маршрутов, исследуемое в рамках текущей реализации

Код определения маршрутов выглядит следующим образом:

package <name package>

import (

<imports>

)

type Handler struct {

services \*service.Service

}

func NewHandler(services \*service.Service) \*Handler {

return &Handler{services: services}

}

// Init routes

func (h \*Handler) InitRoutes() \*gin.Engine {

router := gin.New()

auth := router.Group("/auth")

{

auth.POST("/sign-up", h.signUp)

auth.POST("/sign-in", h.signIn)

auth.POST("/refresh", h.refresh)

auth.POST("/logout", h.logout)

}

return router

}

Для каждого маршрута из группы “/auth” определены обработчики, которые выполняют обработку пользовательских запросов.

Модель базы данных

Определим модель базы данных, которая будет рассмотрена в рамках текущей реализации. Модель базы данных представлена на рисунке 4.

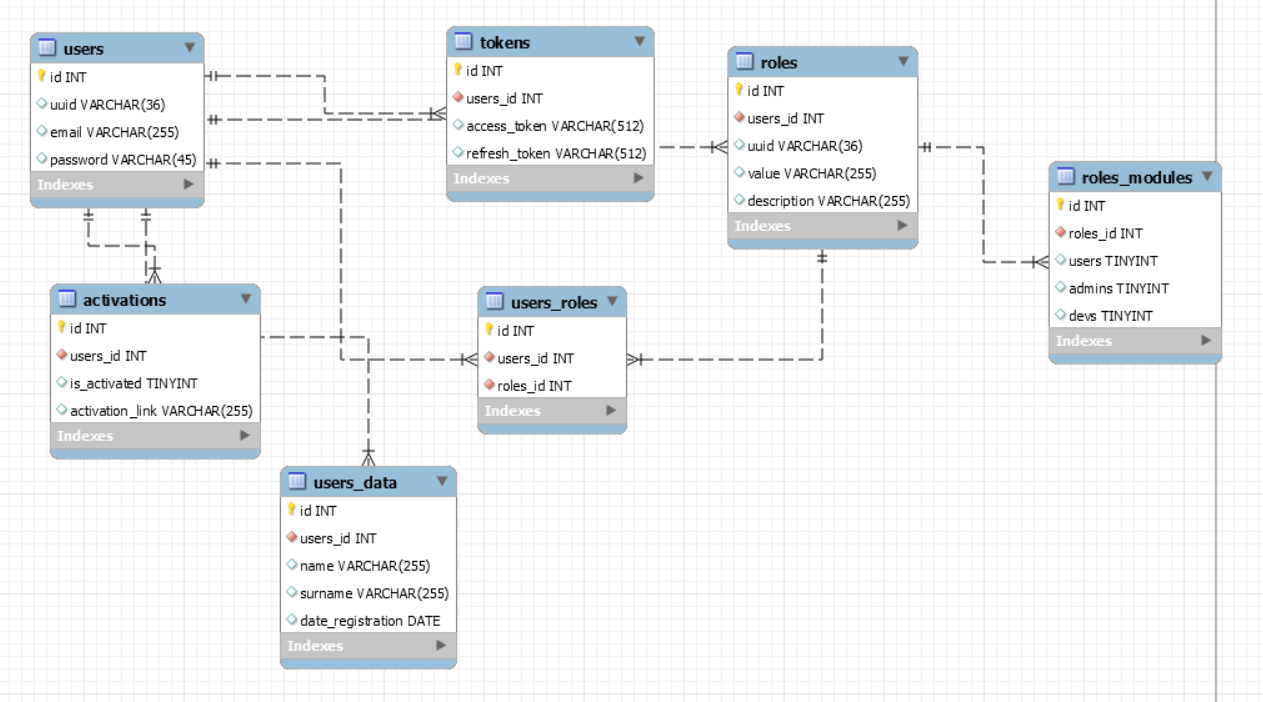


Рисунок 4 – Модель базы данных

В модели базы данных основное внимание будет уделено таблице tokens, которая хранит токены доступа и обновления, а также таблице users и roles, которые хранят информацию о пользователях и ролях соответственно.

Определение обработчиков

Для регистрации пользователя используется маршрут “/auth/sign-up”, на вход данный запрос принимает следующие данные:

1. email
2. Пароль
3. Имя
4. Фамилию

Код обработчика по маршруту “/auth/sign-up” выглядит следующим образом:

func (h \*Handler) signUp(c \*gin.Context) {

var input model.UserRegisterModel

if err := c.BindJSON(&input); err != nil {

newErrorResponse(c, http.StatusBadRequest, "invalid input body")

return

}

data, err := h.services.Authorization.CreateUser(input)

if err != nil {

newErrorResponse(c, http.StatusInternalServerError, err.Error())

return

}

c.JSON(http.StatusOK, data)

}

**Пояснения к коду:**

Определена функция signUp, которая доступна только из того пакета, в котором была определена (по нижнему регистру начального символа начала функции). В обработчике происходит получение данных из JSON-объекта, валидация входных данных и вызов функции создания пользователя, который будет описан чуть ниже. Возвращает обработчик пару токенов доступа и обновления соответственно.

Код создания нового пользователя:

/\*

\* Функция регистрации пользователя

\*/

func (r \*AuthPostgres) CreateUser(user model.UserRegisterModel) (model.UserAuthDataModel, error) {

// Validate check

<code>

// Хэширование пароля

// user.Password = generatePasswordHash(user.Password)

hashedPassword, err := bcrypt.GenerateFromPassword([]byte(user.Password), viper.GetInt("crypt.cost"))

if err != nil {

return model.UserAuthDataModel{}, err

}

user.Password = string(hashedPassword)

var id int

var userUuid string

query := fmt.Sprintf("INSERT INTO %s (email, password, uuid) values ($1, $2, $3) RETURNING id, uuid", usersTable)

// Generate UUID

<code>

// Добавление пользовательских данных

<code>

// Добавление роли пользователю (по-умолчанию данная роль - USER)

<code>

// Генерация токенов доступа и обновления

accessToken, err := GenerateToken(userUuid, role.Uuid, tokenTTL\_access, viper.GetString("token.signing\_key\_access"))

if err != nil {

tx.Rollback()

return model.UserAuthDataModel{}, err

}

refreshToken, err := GenerateToken(userUuid, role.Uuid, tokenTTL\_refresh, viper.GetString("token.signing\_key\_refresh"))

if err != nil {

tx.Rollback()

return model.UserAuthDataModel{}, err

}

// Установка токенов пользователю

query = fmt.Sprintf("INSERT INTO %s (users\_id, access\_token, refresh\_token) values ($1, $2, $3)", tokensTable)

\_, err = tx.Exec(query, id, accessToken, refreshToken)

if err != nil {

tx.Rollback()

return model.UserAuthDataModel{}, err

}

return model.UserAuthDataModel{

AccessToken: accessToken,

RefreshToken: refreshToken,

}, tx.Commit()

}

**Пояснения к коду:**

Функция CreateUser создаёт нового пользователя. С начала функция производит проверку на входные данные и в случае не корректных данных возвращает ошибку. Затем происходит хэширование пароля, генерация UUID и добавление нового пользователя в базу данных. Затем начинается генерация две пары токенов: access и refresh, которые имеют разное время жизни. Токены добавляются в таблицу tokens согласно модели базы данных.

Маршрут, по которому можно авторизоваться следующий: “/auth/sign-in”.

Код обработчика маршрута для авторизации пользователя:

func (h \*Handler) signIn(c \*gin.Context) {

var input model.UserLoginModel

if err := c.BindJSON(&input); err != nil {

newErrorResponse(c, http.StatusBadRequest, "invalid input body")

return

}

data, err := h.services.Authorization.LoginUser(input)

if err != nil {

newErrorResponse(c, http.StatusInternalServerError, err.Error())

return

}

c.JSON(http.StatusOK, data)

}

**Пояснения к коду:**

Код аналогичен предыдущему коду, на вход поступают авторизационные данные (email, пароль), происходит валидация тела запроса и вызов функции LoginUser, в которой происходит проверка пользовательских данных и генерация токенов обновления и доступа.

Код функции LoginUser:

/\*

\* Функция авторизации пользователя

\*/

func (r \*AuthPostgres) LoginUser(user model.UserLoginModel) (model.UserAuthDataModel, error) {

// Validate check

<code>

if err := bcrypt.CompareHashAndPassword([]byte(findUser.Password), []byte(user.Password)); err != nil {

return model.UserAuthDataModel{}, errors.New("Не правильный пароль! Повторите попытку")

}

tx, err := r.db.Begin()

if err != nil {

return model.UserAuthDataModel{}, err

}

// Get role id

<code>

// Генерация токенов доступа и обновления

accessToken, err := GenerateToken(findUser.Uuid, role.Uuid, tokenTTL\_access, viper.GetString("token.signing\_key\_access"))

if err != nil {

tx.Rollback()

return model.UserAuthDataModel{}, err

}

refreshToken, err := GenerateToken(findUser.Uuid, role.Uuid, tokenTTL\_refresh, viper.GetString("token.signing\_key\_refresh"))

if err != nil {

tx.Rollback()

return model.UserAuthDataModel{}, err

}

// Установка токенов пользователю

query = fmt.Sprintf("INSERT INTO %s (users\_id, access\_token, refresh\_token) values ($1, $2, $3)", tokensTable)

\_, err = tx.Exec(query, findUser.Id, accessToken, refreshToken)

if err != nil {

tx.Rollback()

return model.UserAuthDataModel{}, err

}

return model.UserAuthDataModel{

AccessToken: accessToken,

RefreshToken: refreshToken,

}, tx.Commit()

}

**Пояснения к коду:**

Сначала происходит валидация данных и в случае ошибки функция прерывается. Также используется механизм транзакций, который обеспечивает обратный откат базы данных в случае если при работе функции возникают ошибки. В завершении в функции происходит генерация пары токенов и их возвращение, в которые передаются UUID пользователя и роли, являющиеся публичными ID, которые могут быть сохранены во внутреннем хранилище клиентского приложения.

Код обработки запроса “/auth/refresh”:

func (h \*Handler) refresh(c \*gin.Context) {

var input model.TokenRefreshModel

if err := c.BindJSON(&input); err != nil {

newErrorResponse(c, http.StatusBadRequest, "invalid input body")

return

}

data, err := h.services.Authorization.Refresh(input)

if err != nil {

newErrorResponse(c, http.StatusInternalServerError, err.Error())

return

}

c.JSON(http.StatusOK, data)

}

Код функции Refresh:

/\*

\* Функция обновления токена доступа

\*/

func (r \*AuthPostgres) Refresh(token model.TokenRefreshModel) (model.UserAuthDataModel, error) {

userData, err := ParseTokenWithoutValid(token.RefreshToken, viper.GetString("token.signing\_key\_refresh"))

if err != nil {

return model.UserAuthDataModel{}, err

}

user, err := r.GetUser("uuid", userData.UsersId)

if err != nil {

return model.UserAuthDataModel{}, err

}

// Validate token

<code>

// Get role user

<code>

isValid := ValidToken(token.RefreshToken, viper.GetString("token.signing\_key\_refresh"))

setValues := make([]string, 0)

args := make([]interface{}, 0)

argId := 1

var refreshToken string

logrus.Info(isValid)

if !isValid {

refreshToken, err = GenerateToken(user.Uuid, role.Uuid, tokenTTL\_refresh, viper.GetString("token.signing\_key\_refresh"))

if err != nil {

return model.UserAuthDataModel{}, err

}

setValues = append(setValues, fmt.Sprintf("refresh\_token=$%d", argId))

args = append(args, refreshToken)

argId++

} else {

refreshToken = token.RefreshToken

}

accessToken, err := GenerateToken(user.Uuid, role.Uuid, tokenTTL\_access, viper.GetString("token.signing\_key\_access"))

if err != nil {

return model.UserAuthDataModel{}, err

}

// Set values

<code>

// Обновление данных о токене пользователя

\_, err = r.db.Exec(query, args...)

if err != nil {

return model.UserAuthDataModel{}, err

}

// Возвращение авторизационных данных пользователя

return model.UserAuthDataModel{

AccessToken: accessToken,

RefreshToken: refreshToken,

}, nil

}

**Пояснения к коду:**

Сначала происходит проверка полученного токена обновления, и если он не валидный, то обновляется как токен обновления, так и токен доступа соответственно.

Обработчик маршрута “/auth/logout”:

func (h \*Handler) logout(c \*gin.Context) {

var input model.TokenDataModel

if err := c.BindJSON(&input); err != nil {

newErrorResponse(c, http.StatusBadRequest, "invalid input body")

return

}

data, err := h.services.Authorization.Logout(input)

if err != nil {

newErrorResponse(c, http.StatusInternalServerError, err.Error())

return

}

c.JSON(http.StatusOK, LogoutOutputModel{

IsLogout: data,

})

}

Код функции Logout:

/\*

\* Функция разлогирования пользователя

\*/

func (r \*AuthPostgres) Logout(tokens model.TokenDataModel) (bool, error) {

query := fmt.Sprintf("DELETE FROM %s tl WHERE tl.access\_token=$1 AND tl.refresh\_token=$2 RETURNING id", tokensTable)

row := r.db.QueryRow(query, tokens.AccessToken, tokens.RefreshToken)

var id int

if err := row.Scan(&id); err != nil {

return false, err

}

// logrus.Info(string(rune(id)))

return true, nil

}

**Пояснения к коду:**

Функция выполняет удаление записи в таблице tokens по условию совпадения токенов обновления и токена доступа пользователя, что означает что пользователь прекращает использование услуг сервиса.

Механизм генерации и валидации токенов

Функция, которая используется для генерации токена определена следующим образом:

// Структура определяющая данные токена

type tokenClaims struct {

jwt.StandardClaims

<attributes>

}

/\*

\* Функция генерации токена

\*/

func GenerateToken(uuid, rolesUuid string, tokenTTL time.Duration, signingKey string) (string, error) {

token := jwt.NewWithClaims(jwt.SigningMethodHS256, &tokenClaims{

jwt.StandardClaims{

ExpiresAt: time.Now().Add(tokenTTL).Unix(),

IssuedAt: time.Now().Unix(),

},

uuid,

rolesUuid,

})

return token.SignedString([]byte(signingKey))

}

**Пояснения к коду:**

Структура tokenClaims расширяет стандартную заявку (JWT claims), что позволяет разнообразить payload JWT-токена.

В функции GenerateToken происходит генерация нового токена и его последующая подпись секретным ключом, в результате чего мы получаем строку, характеризующую JWT-токен.

Код функции декодирования токена выглядит следующим образом:

/\*

\* Функция получения данных из токена без проверки на валидацию

\*/

func ParseTokenWithoutValid(pToken, signingKey string) (model.TokenOutputParseString, error) {

token, \_ := jwt.ParseWithClaims(pToken, &tokenClaims{}, func(token \*jwt.Token) (interface{}, error) {

if \_, ok := token.Method.(\*jwt.SigningMethodHMAC); !ok {

return nil, errors.New("invalid signing method")

}

return []byte(signingKey), nil

})

// Получение данных из токена (с преобразованием к указателю на tokenClaims)

claims, ok := token.Claims.(\*tokenClaims)

if !ok {

return model.TokenOutputParseString{}, errors.New("token claims are not of type")

}

return model.TokenOutputParseString{

UsersId: claims.UsersId,

RolesId: claims.RolesId,

}, nil

}

**Пояснения к коду:**

В функции ParseTokenWithoutValid происходит декодирование токена, даже в случае завершения его срока жизни. Такой подход использован с целью определить данные в токене обновления пользователя, чтобы не проходить процедуру авторизации повторно.

Функция проверки валидности токена выглядит следующим образом:

/\*

\* Функция проверки валидности токена

\*/

func ValidToken(pToken, signingKey string) bool {

\_, err := jwt.ParseWithClaims(pToken, &tokenClaims{}, func(token \*jwt.Token) (interface{}, error) {

if \_, ok := token.Method.(\*jwt.SigningMethodHMAC); !ok {

return nil, errors.New("invalid signing method")

}

return []byte(signingKey), nil

})

if err != nil {

return false

}

return true

}

**Пояснения к коду:**

В функции происходит проверка токена на валидность (с учётом времени жизни токена).

Функция парсинга токена с валидацией выглядит следующим образом:

// Функция парсинга токена

func (s \*AuthService) ParseToken(pToken, signingKey string) (model.TokenOutputParse, error) {

token, err := jwt.ParseWithClaims(pToken, &tokenClaims{}, func(token \*jwt.Token) (interface{}, error) {

if \_, ok := token.Method.(\*jwt.SigningMethodHMAC); !ok {

return nil, errors.New("invalid signing method")

}

return []byte(signingKey), nil

})

if !token.Valid {

return model.TokenOutputParse{}, errors.New("token is not valid")

}

if err != nil {

return model.TokenOutputParse{}, err

}

// Получение данных из токена (с преобразованием к указателю на tokenClaims)

claims, ok := token.Claims.(\*tokenClaims)

if !ok {

return model.TokenOutputParse{}, errors.New("token claims are not of type")

}

user, err := s.repo.GetUser("uuid", claims.UsersId)

if err != nil {

return model.TokenOutputParse{}, err

}

role, err := s.repo.GetRole("uuid", claims.RolesId)

if err != nil {

return model.TokenOutputParse{}, err

}

return model.TokenOutputParse{

UsersId: user.Id,

RolesId: role.Id,

}, nil

}

**Пояснения к коду:**

Данная функция производит парсинг токена с получением всех необходимых данных для дальнейшего использования в теле функции.

Для быстрой проверки JWT-токена с помощью заголовка запроса Authorization необходимо добавить middleware на все группы запросов, которые работают с самим сервисом.

Код middleware выглядит следующим образом:

func (h \*Handler) userIdentity(c \*gin.Context) {

header := c.GetHeader(authorizationHeader)

if header == "" {

newErrorResponse(c, http.StatusUnauthorized, "Пустой заголовок авторизации!")

return

}

headerParts := strings.Split(header, " ")

if len(headerParts) != 2 {

newErrorResponse(c, http.StatusUnauthorized, "Не корректный авторизационный заголовок!")

return

}

// Парсинг токена доступа

data, err := h.services.Authorization.ParseToken(headerParts[1], viper.GetString("token.signing\_key\_access"))

if err != nil {

newErrorResponse(c, http.StatusUnauthorized, err.Error())

return

}

c.Set(usersCtx, data.UsersId)

c.Set(rolesCtx, data.RolesId)

}

**Пояснения к коду:**

Данный код определяет middleware для проверки токена доступа перед основной обработкой запроса.

Взаимодействие клиента и сервера

Продемонстрируем работу системы аутентификации с помощью JWT-токенов посредством приложения Postman.

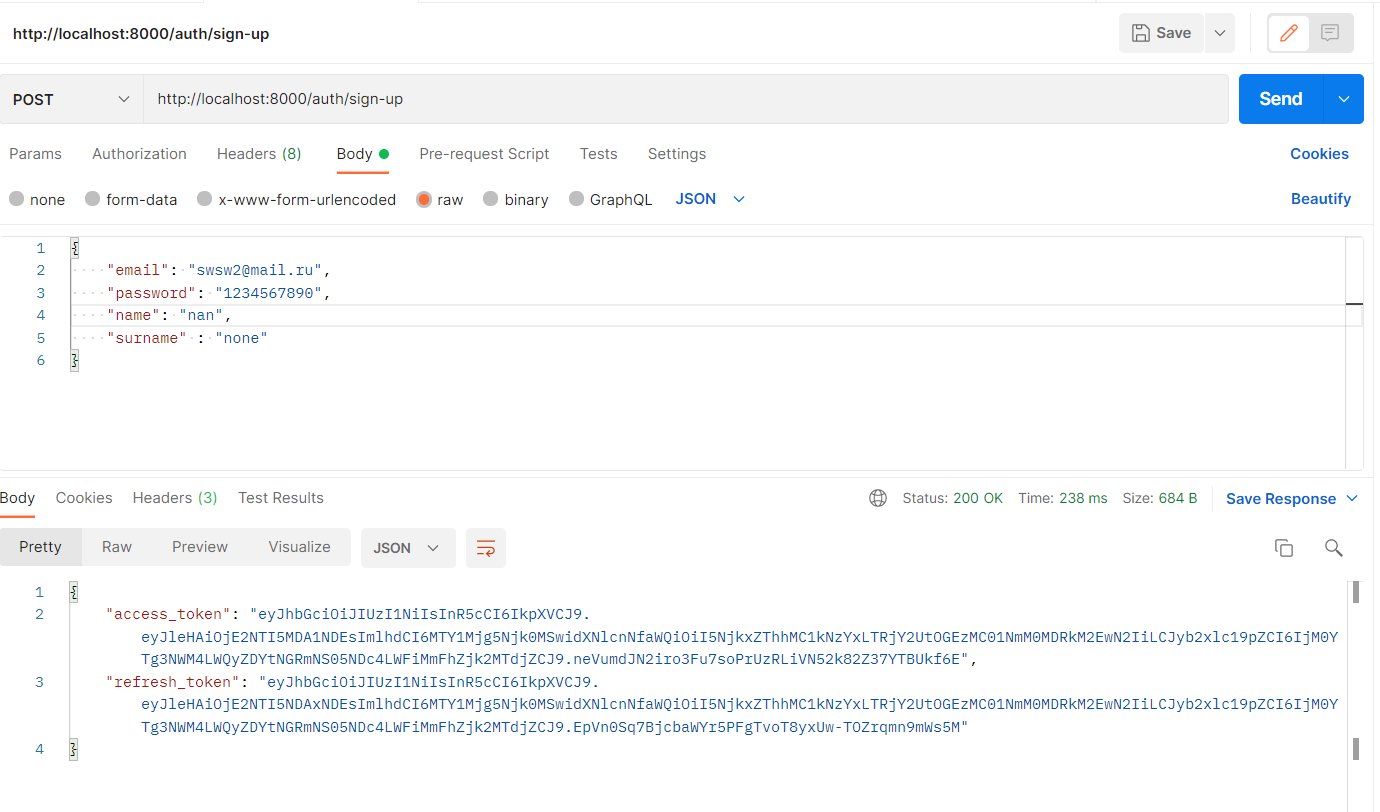


Рисунок 5 – Регистрация пользователя в системе

На рисунке 5 продемонстрирована регистрация пользователя в системе. Как видно из рисунка данные, которые отправляются на сервер, соответствуют требуемым данным для регистрации. После завершения процедуры регистрации (в данном случае успешной) происходит возвращение пары токенов (access / refresh) для работы с сервисом.

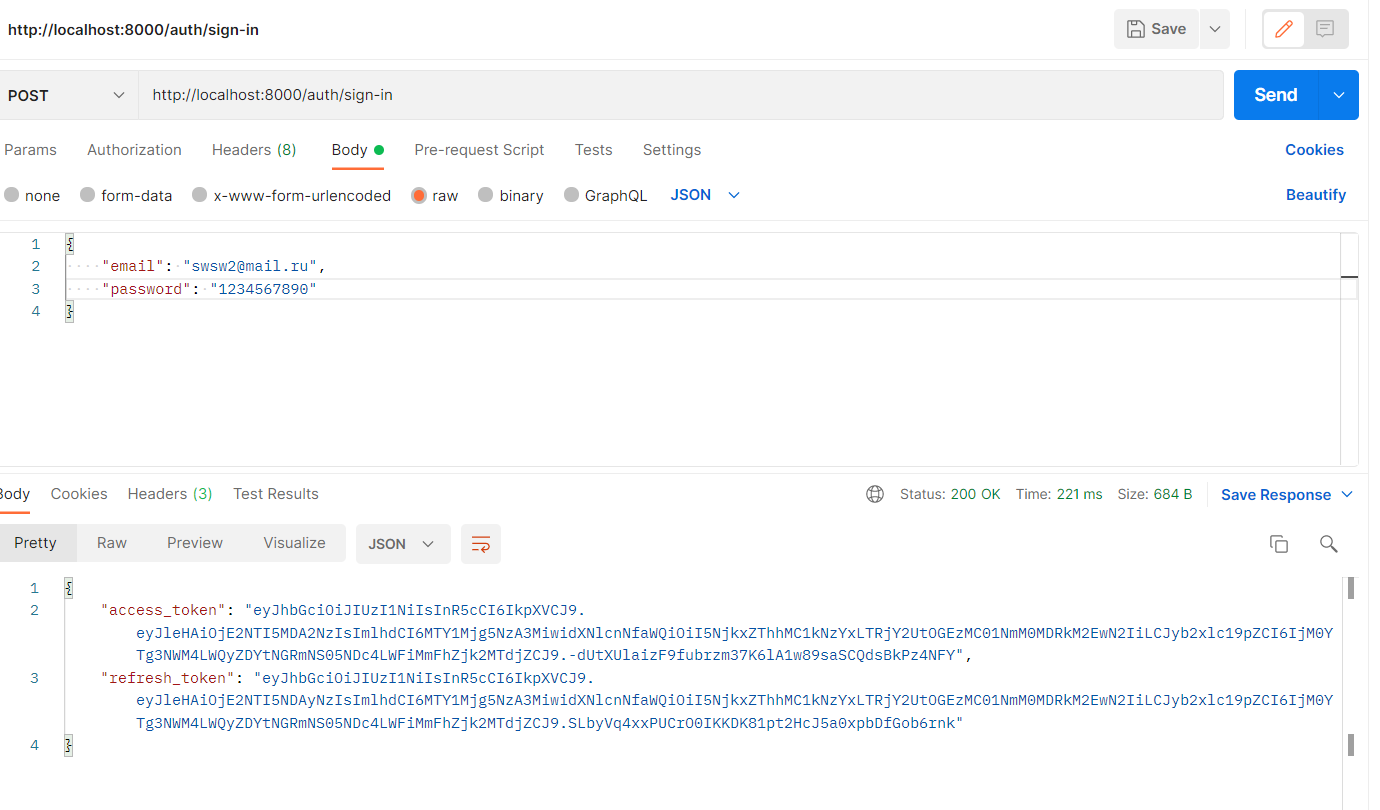


Рисунок 6 – Авторизация пользователя

На рисунке 6 продемонстрирована авторизация пользователя в системе. Так как пароль и email совпали, то возвращается пара токенов для работы в системе (что видно на рисунке).

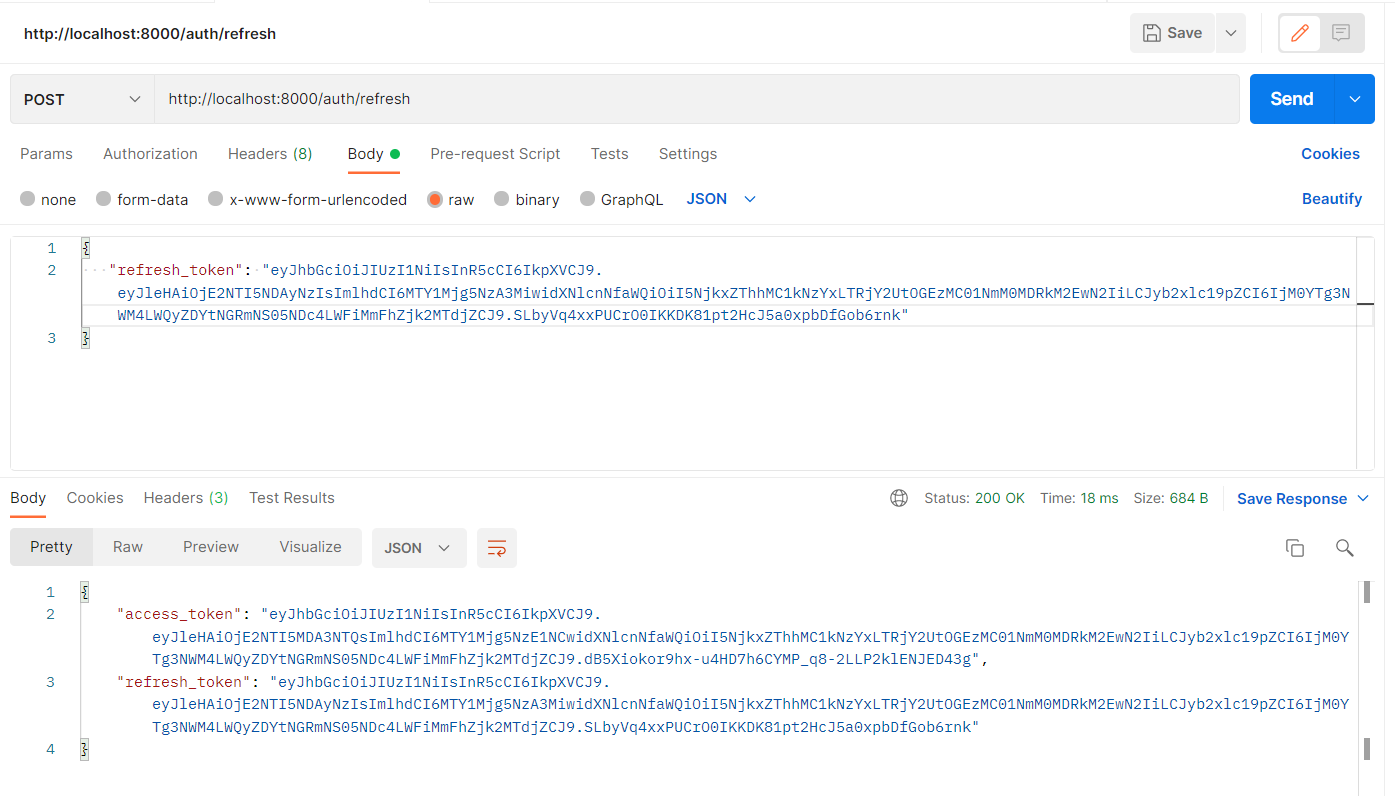


Рисунок 7 – Обновление access-токена

На рисунке 7 продемонстрировано обновление токена доступа с помощью токена обновления.

Сравнивая рисунки 6 и 7 можно заметить, что токен доступа изменился, однако токен обновления остался таким, каким был. Это связано с тем, что токен обновления оставался и остаётся валидным до сих пор.

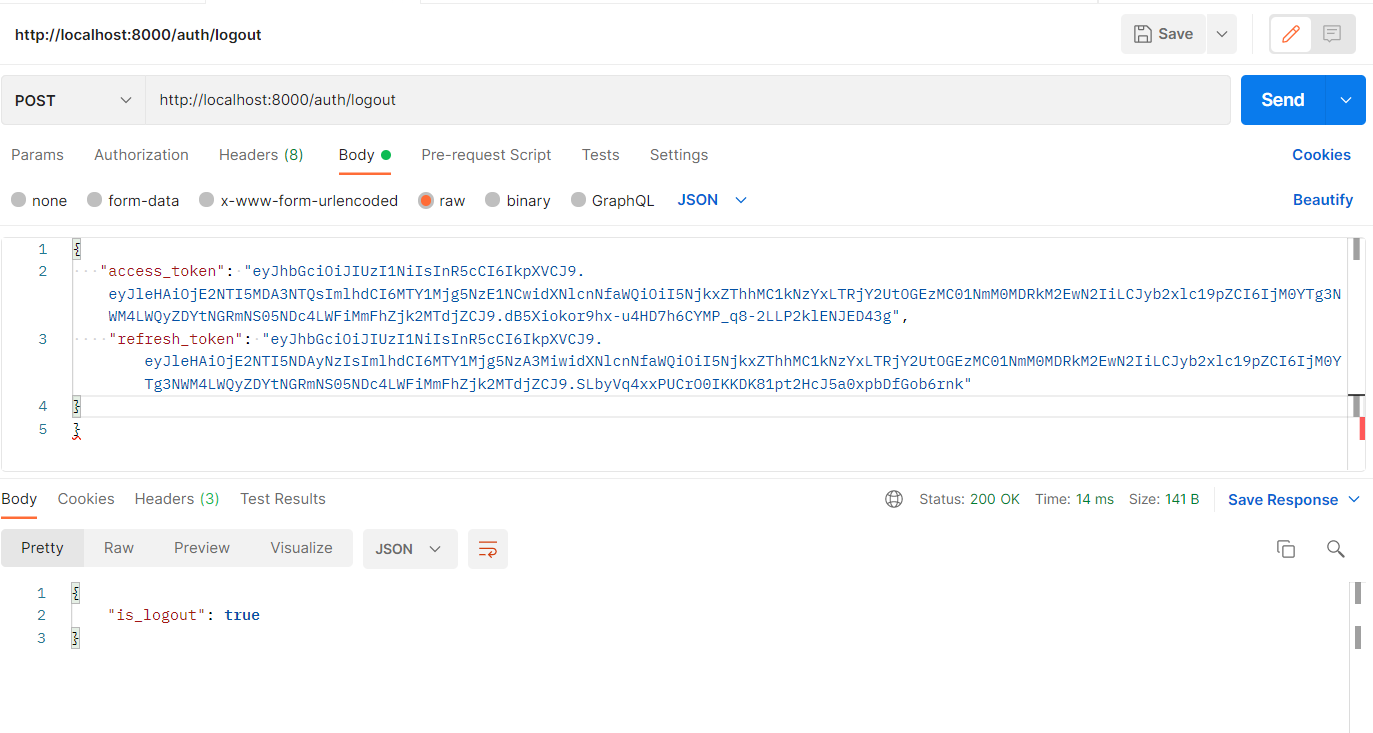


Рисунок 8 – Выход пользователя из аккаунта

Заключение

JWT-аутентификация достаточно легка в освоении и реализации, что способствует её большому распространению и популярности как стандарт.

Была проанализирована тема с JSON Web Token стандартом и определены ключевые понятия, структура и схема взаимодействия сервера и пользователя. Были обозначены особенности данного стандарта как обеспечивающего безопасность посредством различных алгоритмов шифрования (HS256) и методикой шифрования на сервере посредством секретного ключа.

Была реализована схема JWT-аутентификации на сервере с использованием языка программирования Golang, реализованы эндпоинты приложения, даны краткие пояснения программным инструкциям и выявлены их особенности.

В заключении была продемонстрирована работоспособность серверного приложения в соответствии с общей схемой клиент-серверного взаимодействия с JWT-аутентификацией. В результате было получено приложение, которое предоставляет возможность аутентификации пользователей и обеспечивает надёжный уровень защиты от несанкционированного доступа, а также ограничение возможностей для осуществления DDoS-атак.

Библиографический список

1. borisyuzhakov. Пять простых шагов для понимания JSON Web Tokens (JWT) [Электронный ресурс]. URL: https://habr.com/ru/post/340146/? (дата обращения: 18.05.2022).

2. Go [Электронный ресурс]. URL: 18.05.2022 (дата обращения: https://ru.wikipedia.org/wiki/Go).

3. Павел Кочетов. JWT — как безопасный способ аутентификации и передачи данных [Электронный ресурс]. URL: https://vc.ru/dev/106534-jwt-kak-bezopasnyy-sposob-autentifikacii-i-peredachi-dannyh?ysclid=l3bmaevnnv (дата обращения: 18.05.2022).