

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
УФИМСКИЙ УНИВЕРСИТЕТ НАУКИ И ТЕХНОЛОГИЙ

РЕФЕРАТ

по дисциплине: **«Администрирование информационных систем»**
на тему **«Аутентификация и авторизация в RESTful API: сравнение
подходов (JWT, OAuth 2.0, сессии)»**

Выполнил: студент гр. ПРО-436Б

Хомутов Д. А.

Проверил:

доцент каф. ВМиК

Чернышев Е. С.

Уфа 2025 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
1 Концептуальные основы: аутентификация и авторизация	4
1.1 Определение и разграничение понятий	4
1.2 Особенности RESTful API и требования к управлению доступом	4
2 Традиционный подход: аутентификация на основе сессий	5
2.1 Механизм работы (Cookie, Server-Side Storage)	5
2.2 Преимущества и недостатки подхода на основе сессий	6
3 Аутентификация без состояния с использованием JWT	7
3.1 Структура и принцип работы JWT (Header, Payload, Signature)	7
3.2 Алгоритм обмена: от выдачи токена до его валидации на защищённых маршрутах	8
4 Протокол делегированного доступа: OAuth 2.0 и OIDC	9
4.1 Роли и гранты	9
4.2 OpenID Connect как надстройка над OAuth 2.0	11
5 Сравнительный анализ подходов и выбор стратегии	11
5.1 Сравнительный анализ	11
5.2 Рекомендации по применению	13
ЗАКЛЮЧЕНИЕ	15
СПИСОК ЛИТЕРАТУРЫ	16

ВВЕДЕНИЕ

В современной архитектуре информационных систем, особенно построенных на принципах микросервисов и ориентированных на веб- и мобильные клиенты, RESTful API стал стандартом для взаимодействия между компонентами. Однако предоставление доступа к функционалу и данным через API создает критически важную задачу обеспечения безопасности, центральными элементами которой являются аутентификация и авторизация. Неверно выбранный или некорректно реализованный механизм управления доступом может стать причиной утечек данных, несанкционированных действий и компрометации всей системы.

Актуальность темы обусловлена повсеместным распространением API-ориентированной разработки и возрастающими требованиями к безопасности, отраженными в таких стандартах, как OWASP API Security Top 10. На рынке сосуществуют несколько принципиально разных подходов к решению задачи: от традиционных сессий на стороне сервера до современных статистических токенов (JWT) и протоколов делегированного доступа (OAuth 2.0). Каждый из них имеет свою область оптимального применения, преимущества и уязвимости.

Цель данного реферата – провести сравнительный анализ основных подходов к аутентификации и авторизации в RESTful API: JWT, OAuth 2.0 и сессионной аутентификации. В задачи работы входит: раскрыть принцип работы каждого механизма, оценить их с точки зрения безопасности, масштабируемости, производительности и сложности внедрения, а также сформулировать практические рекомендации по выбору стратегии в зависимости от архитектурного контекста информационной системы.

1 Концептуальные основы: аутентификация и авторизация

1.1 Определение и разграничение понятий

Аутентификация (Authentication) – это процесс проверки подлинности субъекта (пользователя, сервиса или устройства), устанавливающий, что он действительно является тем, за кого себя выдает. Результатом успешной аутентификации является установление идентификатора субъекта (user identity). Основные факторы аутентификации: знание (пароль), обладание (токен, сертификат), свойство (биометрия).

Авторизация (Authorization) – это процесс определения, какие действия, ресурсы или данные разрешены уже аутентифицированному субъекту. Авторизация отвечает на вопрос «Что ты можешь делать?» после того, как аутентификация ответила на вопрос «Кто ты?». Механизмы авторизации включают ролевой доступ (RBAC), доступ на основе атрибутов (ABAC) и политики разрешений.

В контексте RESTful API оба процесса реализуются на уровне каждого HTTP-запроса. Запрос к защищенному ресурсу должен содержать учетные данные, позволяющие пройти аутентификацию, после чего система проверяет права доступа к запрашиваемым данным.

1.2 Особенности RESTful API и требования к управлению доступом

REST (Representational State Transfer) – архитектурный стиль, который основывается на принципах отсутствия состояния (statelessness), единообразия интерфейса и разделения клиента и сервера. Принцип отсутствия состояния является ключевым для понимания выбора механизма аутентификации. Он означает, что каждый запрос от клиента к серверу должен содержать всю

информацию, необходимую для его понимания и обработки. Сервер не хранит состояние сессии клиента между запросами.

Это накладывает специфические требования на механизмы безопасности:

1) Переносимость учетных данных: Учетные данные или токен доступа должны легко передаваться в каждом HTTP-запросе, обычно через заголовки (например, Authorization: Bearer <token>).

2) Масштабируемость: Отсутствие привязки к сессии на конкретном сервере позволяет легко горизонтально масштабировать пул серверов приложений.

3) Совместимость с разнородными клиентами: Механизм должен одинаково хорошо работать с веб-браузерами, мобильными приложениями, серверными сервисами и устройствами Интернета вещей (IoT).

4) Стандартизация: Предпочтение отдается открытым, широко принятым стандартам (RFC), которые поддерживаются большинством библиотек и фреймворков.

2 Традиционный подход: аутентификация на основе сессий

2.1 Механизм работы (Cookie, Server-Side Storage)

Этот подход является классическим для веб-приложений. Его алгоритм применительно к API выглядит следующим образом:

1) Клиент отправляет учетные данные (логин/пароль) на эндпоинт аутентификации. Эндпоинт аутентификации – это адрес, обычно представленный в виде URL, через который клиентское приложение взаимодействует с сервером или сервисом для выполнения операций или получения данных.

2) Сервер проверяет учетные данные. В случае успеха создает уникальный идентификатор сессии (Session ID), который сохраняет в хранилище данных на стороне сервера (база данных, Redis, Memcached). В сессию также записываются данные пользователя (его ID, роли).

3) Сервер отправляет Session ID клиенту, обычно устанавливая его в cookie HTTP-ответа (Set-Cookie: sessionId=abc123; HttpOnly; Secure).

4) При последующих запросах к API клиент автоматически отправляет cookie с Session ID (благодаря браузеру) или явно добавляет его в заголовок.

5) Сервер приложения, получив запрос, извлекает Session ID, по нему находит данные сессии в хранилище, восстанавливает контекст пользователя и проверяет его права доступа.

2.2 Преимущества и недостатки подхода на основе сессий

Преимущества:

1) Простота инвалидации: Поскольку сессия хранится на сервере, ее можно мгновенно удалить, что обеспечивает немедленный выход (logout) и возможность отзыва доступа.

2) Защита от кражи: Использование флагов HttpOnly и Secure для cookies затрудняет кражу сессионного токена через XSS-атаки.

3) Хранение большого объема контекста: На сервере можно хранить произвольные данные пользователя, не передавая их по сети.

Недостатки:

1) Нарушение принципа stateless: Сервер вынужден хранить состояние, что противоречит одному из ключевых принципов REST.

2) Проблемы с масштабируемостью: необходимо настраивать общее хранилище сессий для всех состояний приложения (sticky sessions или внешнее хранилище), что добавляет сложность и создает единую точку отказа.

3) Проблемы с CORS и мобильными клиентами: Использование cookies может быть нетривиальным для нативных мобильных приложений или при взаимодействии с API из домена, отличного от основного (требует корректной настройки CORS).

4) Производительность: Каждый запрос требует дополнительного обращения к хранилищу данных для проверки сессии, что создает нагрузку и увеличивает задержку.

Таким образом, сессионная аутентификация менее пригодна для высоконагруженных микросервисных архитектур, но может оставаться валидным выбором для монолитных веб-приложений с преимущественно браузерными клиентами.

3 Аутентификация без состояния с использованием JWT

3.1 Структура и принцип работы JWT (Header, Payload, Signature)

JWT (JSON Web Tokens) – это компактный, URL-безопасный способ представления утверждений, которые будут передаваться между двумя сторонами. Токен самодостаточен и содержит всю необходимую информацию о пользователе, что устраняет необходимость хранения состояния на сервере.

Структура JWT: Header.Payload.Signature.

1) Header содержит метаданные: тип токена (type: "JWT") и алгоритм подписи (alg: "HS256" или RS256).

2) Payload содержит утверждения (claims) – набор пар «ключ-значение» о субъекте. Стандартные claims: iss (issuer), sub (subject), exp (expiration time). Также здесь размещаются кастомные данные (например, userId, roles).

3) Signature создается путем кодирования header и payload с использованием секретного ключа (при HMAC) или приватного ключа (при

RSA) по указанному алгоритму. Подпись гарантирует целостность токена – любое изменение содержимого сделает подпись невалидной.

Токен кодируется в формат Base64Url и передается в заголовке запроса: `Authorization: Bearer <JWT>`.

3.2 Алгоритм обмена: от выдачи токена до его валидации на защищённых маршрутах

Алгоритм обмена:

Аутентификация и выдача: Клиент отправляет учетные данные. Сервер аутентификации (Auth Server) проверяет их и, в случае успеха, генерирует JWT, подписывает его и возвращает клиенту.

Использование: Клиент сохраняет JWT (обычно в localStorage или memory) и добавляет его в заголовок каждого последующего запроса к защищенным ресурсам.

Валидация: Ресурсный сервер (Resource Server), получив запрос с токеном, выполняет:

- 1) Проверку структуры и корректности подписи, используя секретный/публичный ключ.
- 2) Проверку стандартных утверждений: срок действия (exp), издателя (iss), аудиторию (aud).
- 3) Извлечение данных пользователя (например, roles из payload) для проведения авторизации.

Преимущества JWT:

- 1) Полная stateless-архитектура: Серверу не нужно хранить состояние сессии, что идеально подходит для горизонтального масштабирования микросервисов.
- 2) Универсальность: Токен может нести в себе данные для авторизации (роли), сокращая количество запросов к БД.

3) Поддержка разнородных клиентов: Легко используется любым клиентом, умеющим отправлять HTTP-заголовки.

Недостатки и риски:

1) Сложность отзыва (Logout): Токен валиден до истечения его срока жизни (exp). Для реализации немедленного отзыва необходимо внедрять дополнительные механизмы (черные списки токенов – Token Blacklist), что частично возвращает состояние.

2) Безопасность хранения на клиенте: В браузере уязвим к XSS-атакам, если хранится в localStorage.

3) Некорректная реализация: Риск использования слабых алгоритмов подписи, отсутствия проверки aud, хранения чувствительных данных в payload (который легко декодируется).

4) Увеличение размера запроса: Токен может быть объемным, особенно если в него помещено много данных.

JWT является отличным выбором для stateless API, микросервисов и сценариев, где необходима возможность делегирования проверки подлинности между разными сервисами без обращения к центральному хранилищу.

4 Протокол делегированного доступа: OAuth 2.0 и OIDC

4.1 Роли и гранты

OAuth 2.0 – это протокол авторизации. Основная цель – позволить приложению (Client) получить ограниченный доступ к ресурсам пользователя (Resource Owner) на другом сервисе (Resource Server), не раскрывая пароли пользователя.

Ключевые роли (Roles):

- 1) Resource Owner: Пользователь, который владеет защищенными данными.
- 2) Client: Приложение (веб-сайт, мобильное приложение), которое запрашивает доступ.
- 3) Resource Server: Сервер, на котором хранятся защищенные ресурсы (например, API Google Drive).
- 4) Authorization Server: Сервер, который аутентифицирует Resource Owner, получает его согласие и выдает токены доступа (Access Token) Client.

Типы разрешений (Grants):

- 1) Authorization Code Flow: Наиболее безопасный поток для веб-серверных приложений. Пользователь перенаправляется на Authorization Server для аутентификации и согласия, после чего Client получает код, который обменивает на Access Token и Refresh Token.
- 2) Client Credentials Flow: Используется для взаимодействия между сервисами (M2M), где нет пользователя. Приложение аутентифицируется с помощью своего client_id и client_secret, получая токен для доступа к собственным ресурсам.
- 3) Resource Owner Password Credentials Flow: Пользователь передает свои логин/пароль напрямую Client, который использует их для получения токена. Применяется только для доверенных приложений (например, официальные мобильные клиенты).

Access Token в OAuth 2.0 – это обычно непрозрачный токен (opaque token), который клиент использует для доступа к API. Сам по себе он не содержит данных; Resource Server должен интроспектировать его (сделать запрос к Authorization Server) для проверки валидности и получения информации о субъекте.

4.2 OpenID Connect как надстройка над OAuth 2.0

OpenID Connect (OIDC) – это тонкий слой поверх OAuth 2.0, который добавляет функционал аутентификации. Он предоставляет стандартный способ получения информации о пользователе.

Ключевое дополнение – ID Token. Это JWT, который содержит утверждения о аутентификации пользователя (sub – идентификатор пользователя, name, email и др.). ID Token предназначен для клиента, чтобы тот мог узнать, кто аутентифицирован. Access Token используется для доступа к API.

Таким образом, связка OAuth 2.0 + OIDC решает обе задачи: аутентификацию пользователя (через ID Token) и авторизацию для доступа к API (через Access Token). Это промышленный стандарт для реализации единого входа (Single Sign-On, SSO), социального логина (через Google, Facebook) и централизованного управления доступом в корпоративных системах.

5 Сравнительный анализ подходов и выбор стратегии

5.1 Сравнительный анализ

Сравнительный анализ трех основных подходов позволяет выявить их принципиальные отличия и определить области оптимального применения.

Безопасность:

1) Сессионный подход демонстрирует высокую устойчивость за счет возможности мгновенного отзыва доступа и защиты токена флагами HttpOnly и Secure, что эффективно против многих XSS-атак. Однако он уязвим к CSRF, для защиты от которого требуются дополнительные меры.

2) JWT создает существенные риски при некорректной реализации. Главный недостаток — сложность немедленного отзыва (требуемая внедрения черных списков), а хранение в localStorage делает его мишенью для XSS. Также существует риск утечки данных, если разработчик ошибочно помещает чувствительную информацию в payload.

3) OAuth 2.0 с OIDC представляет наиболее комплексный и безопасный фреймворк. Он изначально спроектирован с разделением ролей, использованием короткоживущих токенов и протокола PKCE, защищающего от перехвата кода авторизации.

Масштабируемость:

1) Сессионная аутентификация является наименее предпочтительной. Необходимость в общем хранилище сессий создает узкое место и точку отказа, а использование липких сессий (sticky sessions) усложняет балансировку нагрузки.

2) JWT, будучи полностью без состояния, является идеальным решением для горизонтального масштабирования. Каждый сервис может независимо проверить токен, что идеально подходит для микросервисной архитектуры.

3) Масштабируемость OAuth 2.0 зависит от типа токена. Использование JWT в качестве Access Token обеспечивает stateless-масштабируемость. Классическая схема с непрозрачными токенами требует запросов на интроспекцию к центральному Authorization Server, что создает централизованную нагрузку.

Производительность:

1) Сессионный подход требует дополнительного сетевого запроса к хранилищу для валидации каждого запроса, что увеличивает задержку.

2) JWT обеспечивает максимальную производительность: валидация сводится к локальной криптографической операции и проверке утверждений.

3) Производительность OAuth 2.0 вариативна: использование JWT дает высокую скорость, а валидация непрозрачного токена через интроспекцию добавляет существенную сетевую задержку.

Сложность реализации:

1) Сессионный механизм наиболее прост для понимания и внедрения в рамках монолитного приложения.

2) Реализация JWT требует глубокого понимания криптографии, правильного выбора алгоритмов и управления ключами.

3) OAuth 2.0 и OIDC являются наиболее сложными для правильной реализации. Они требуют развертывания нескольких компонентов, понимания различных типов грантов и обеспечения безопасного хранения секретов.

5.2 Рекомендации по применению

Одностраничное приложение (SPA) + REST API Backend:

Наиболее современным и безопасным выбором является Authorization Code Flow with PKCE (OAuth 2.1 / OIDC). Устаревший Implicit Flow использовать не следует. Access Token (предпочтительно в формате JWT) передается в заголовках API-запросов. Для хранения Refresh Token следует рассмотреть использование защищенного, HttpOnly cookie (если frontend и backend находятся на одном домене) в комбинации с короткоживущими Access Tokens для минимизации рисков.

Нативное мобильное приложение:

Здесь также стандартом выступает Authorization Code Flow with PKCE. Этот поток специально разработан для публичных клиентов (мобильных приложений), где невозможно безопасно хранить секрет. Токены сохраняются в защищенных хранилищах операционной системы (Keychain для iOS, Keystore для Android).

Микросервисная архитектура:

В современных облачных средах часто применяется гибридный подход. OIDC используется на входе (например, через API-шлюз) для аутентификации пользователя. После успешной аутентификации шлюз генерирует внутренний JWT, содержащий утверждения о пользователе, и передает его во внутренние микросервисы. Это сочетает преимущества централизованного управления доступом (OIDC) и высокой скорости межсервисной коммуникации (JWT). Для фонового взаимодействия сервисов между собой (service-to-service) оптимальным является Client Credentials Flow (OAuth 2.0).

Традиционное веб-приложение с серверным рендерингом:

Для данного сценария сессионная аутентификация остается валидным, простым и эффективным выбором. Она отлично интегрируется с механизмами работы веб-фреймворков, обеспечивает простой немедленный выход и не требует передачи сложных токенов в каждом AJAX-запросе.

ЗАКЛЮЧЕНИЕ

В ходе написания реферата были проанализированы три принципиально разных подхода к обеспечению безопасности RESTful API: традиционная сессионная аутентификация, статистическая аутентификация с использованием JWT и протокол делегированного доступа OAuth 2.0 с расширением OpenID Connect.

Каждый механизм решает задачу аутентификации и авторизации в своем архитектурном контексте. Сессии, нарушая принцип stateless, остаются простым и эффективным решением для классических веб-приложений. JWT, являясь самодостаточным токеном, стал стандартом для построения масштабируемых, независимых микросервисных экосистем, где приоритетом являются производительность и простота валидации. OAuth 2.0 и OIDC представляют собой наиболее комплексное и безопасное решение для сценариев, требующих централизованного управления доступом, единого входа, интеграции со сторонними сервисами и разделения ответственности между компонентами системы.

Выбор оптимальной стратегии не является универсальным и должен основываться на тщательном анализе требований конкретной информационной системы: типа клиентов, необходимого уровня безопасности, требуемой масштабируемости и доступных ресурсов для разработки и поддержки. Зачастую в рамках одной системы комбинируются несколько подходов (например, OIDC для входа пользователя и последующая выдача JWT для доступа к внутренним API), что позволяет использовать сильные стороны каждой технологии. Грамотная реализация выбранного механизма, следование лучшим практикам безопасности (валидация токенов, защита от CSRF/XSS, использование HTTPS) и постоянный мониторинг являются не менее важными факторами, чем сам выбор технологии.

СПИСОК ЛИТЕРАТУРЫ

1. IETF. The OAuth 2.0 Authorization Framework [Электронный ресурс] / D. Hardt. – 2012. – URL: <https://datatracker.ietf.org/doc/html/rfc6749> (дата обращения: 22.12.2025).
2. IETF. JSON Web Token (JWT) [Электронный ресурс] / M. Jones. – 2015. – URL: <https://datatracker.ietf.org/doc/html/rfc7519> (дата обращения: 22.12.2025)
3. OpenID Foundation. OpenID Connect Core 1.0 [Электронный ресурс] / N. Sakimura. – 2023. – URL: https://openid.net/specs/openid-connect-core-1_0.html (дата обращения: 22.12.2025)
4. OWASP Foundation. OWASP API Security Top-10 2023 [Электронный ресурс]. – URL: <https://owasp.org/API-Security/editions/2023/en/0x00-toc/> (дата обращения: 22.12.2025)
5. OWASP Foundation. OWASP: Authentication [Электронный ресурс]. – URL: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html (дата обращения: 22.12.2025)
6. Microsoft Identity Platform. Microsoft identity platform and OAuth 2.0 authorization code flow [Электронный ресурс]. – URL: <https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-auth-code-flow> (дата обращения: 22.12.2025)
7. Okta Developer. OAuth 2.0 and OpenID Connect Overview [Электронный ресурс]. – URL: <https://developer.okta.com/docs/concepts/oauth-openid/> (дата обращения: 22.12.2025)
8. Postman. What is OAuth 2.0? [Электронный ресурс]. – URL: <https://learning.postman.com/docs/sending-requests/authorization/oauth-20/> (дата обращения: 22.12.2025)

Реферат размещён по ссылке:

[https://github.com/AIS-436/Khomutov Denis Aleksandrovich 18](https://github.com/AIS-436/Khomutov_Denis_Aleksandrovich_18)