### ****ДИПЛОМНАЯ РАБОТА****

**На тему:**  
**«Анализ и сравнение различных методов аутентификации и авторизации в веб-приложениях: OAuth, JWT и session-based authentication»**

#### ****Выполнил:**** Вавилин Сергей Александрович

Github –репозиторий с проектом  
<https://github.com/sergeivavilin/Fastapi_authentications>

**2024**

**Введение**

**Актуальность темы**

С развитием веб-приложений и увеличением объема обрабатываемых данных вопрос обеспечения безопасности стал одним из ключевых. Веб-приложения требуют надежных механизмов аутентификации и авторизации, чтобы защитить данные пользователей и предотвратить несанкционированный доступ. Различные методы, такие как OAuth, JWT и session-based authentication, активно применяются в современных системах, однако каждый из них имеет свои особенности, преимущества и недостатки.

Выбор подходящего метода аутентификации и авторизации зависит от специфики приложения, требований к безопасности и удобству использования. Например, JWT стал популярным благодаря легкости интеграции в распределенные системы, в то время как OAuth является стандартом для интеграции с внешними сервисами. Session-based authentication, несмотря на свою долгую историю, остается актуальной благодаря своей простоте и широкому применению.

Актуальность исследования обусловлена необходимостью глубокого анализа этих методов для выбора наиболее подходящего решения в зависимости от требований приложения.

#### ****Цели и задачи работы****

Целью работы является анализ и сравнение методов аутентификации и авторизации в веб-приложениях на основе их реализации, безопасности и удобства использования.

Для достижения цели необходимо решить следующие задачи:

1. Провести теоретический обзор методов аутентификации и авторизации, включая OAuth, JWT и session-based authentication.
2. Реализовать каждый из методов в веб-приложениях на Python.
3. Провести анализ безопасности с учетом защиты от CSRF, XSS и утечек токенов.
4. Сравнить удобство использования каждого метода для разработчиков и конечных пользователей.
5. Сформулировать выводы и рекомендации на основе проведенного анализа.

**Методы исследования**

Для реализации целей будут использованы:

* Разработка веб-приложений на Python с использованием фреймворка FastAPI.
* Анализ посвященный вопросам аутентификации и авторизации.
* Сравнительный анализ на основе заданных критериев безопасности и удобства.

**Структура работы**

Работа состоит из следующих разделов:

1. Введение: описание целей, задач и структуры работы.
2. Обзор методов аутентификации и авторизации.
3. Теоретический анализ.
4. Практическая реализация методов с использованием фреймворка FastAPI.
5. Результаты и выводы, включая рекомендации для разработчиков.

### ****Обзор методов аутентификации и авторизации****

#### ****Основные принципы аутентификации и авторизации****

Аутентификация и авторизация — это два ключевых процесса, необходимых для обеспечения безопасности веб-приложений:

1. **Аутентификация** (Authentication) — процесс проверки личности пользователя. Например, ввод логина и пароля.
2. **Авторизация** (Authorization) — процесс проверки прав пользователя на доступ к определенным ресурсам или действиям после прохождения аутентификации.

Современные подходы к аутентификации и авторизации нацелены на обеспечение:

* Безопасного хранения данных пользователя.
* Удобства использования для конечного пользователя.
* Масштабируемости системы.

Аутентификация и авторизация являются ключевыми процессами в современных веб-приложениях, обеспечивая доступ к защищенным ресурсам и данным. Рассмотрим три наиболее распространенных метода: OAuth, JWT и session-based authentication.

**OAuth 2.0**

**Описание**:  
OAuth 2.0 – это протокол авторизации, который позволяет предоставлять доступ третьим сторонам к ресурсам пользователя без передачи учетных данных. Это делается с использованием токенов доступа, выдаваемых сервером авторизации.

**Ключевые особенности**:

* Поддерживает несколько типов токенов (токены доступа и обновления).
* Работает с различными сценариями (веб-приложения, мобильные приложения, API).

**Преимущества**:

* Широкая поддержка в экосистеме веб-приложений.
* Возможность делегирования прав доступа третьим сторонам.
* Удобство использования для пользователей за счет интеграции с популярными платформами (например, Google, Facebook).

**Недостатки**:

* Сложность реализации, особенно для серверов авторизации.
* Потенциальные угрозы безопасности, такие как атаки через утечку токенов или неправильную конфигурацию.

**JSON Web Token (JWT)**

**Описание**:  
JWT – это компактный, самодостаточный токен, который используется для передачи данных между клиентом и сервером. Токен состоит из трех частей: заголовка, полезной нагрузки (payload) и подписи, что делает его безопасным и легко проверяемым.

**Ключевые особенности**:

* Полезная нагрузка может содержать любую информацию, включая данные о пользователе и права доступа.
* Токен подписывается секретным ключом или с использованием асимметричного шифрования, что гарантирует его подлинность.

**Преимущества**:

* Подходит для распределенных систем и микросервисов, так как серверу не нужно хранить состояние.
* Гибкость: токены можно использовать как для аутентификации, так и для авторизации.

**Недостатки**:

* Риск компрометации токена: если токен украден, его можно использовать до истечения срока действия.
* Удаление токена до его истечения требует дополнительной реализации (например, черных списков).

**Session-Based Authentication**

**Описание**:  
Session-based authentication – это традиционный подход, в котором после успешной аутентификации на сервере создается уникальная сессия для пользователя. Эта сессия хранится на сервере, а клиент получает идентификатор сессии (обычно в виде cookie), который отправляется с каждым запросом.

**Ключевые особенности**:

* Сессии хранятся на сервере, обеспечивая централизованный контроль.
* Клиент хранит только идентификатор сессии, что минимизирует риск утечки данных.
* Сессии могут быть настроены на автоматическое истечение после определенного времени.

**Преимущества**:

* Легкость реализации в серверно-ориентированных приложениях.
* Высокий уровень безопасности при правильной настройке (например, использование защищенных cookie с атрибутами HttpOnly и Secure).

**Недостатки**:

* Неэффективность при масштабировании: увеличение числа пользователей требует больше ресурсов для хранения сессий.
* Зависимость от сервера делает метод менее удобным для распределенных систем.

#### ****Сравнительная таблица методов****

| **Характеристика** | **OAuth** | **JWT** | **Session-based Authentication** |
| --- | --- | --- | --- |
| **Масштабируемость** | Высокая | Высокая | Низкая (зависит от сервера) |
| **Простота реализации** | Низкая | Средняя | Высокая |
| **Безопасность** | Высокая (при правильной настройке) | Средняя (в случае утечки токенов) | Высокая |
| **Удобство для разработчика** | Среднее | Высокое | Высокое |
| **Подходит для** | Авторизации сторонних приложений | Микросервисов и SPA | Серверно-ориентированных систем |

### ****Теоретический анализ****

#### ****Исторический обзор и эволюция методов аутентификации и авторизации****

История методов аутентификации и авторизации тесно связана с развитием интернета и программного обеспечения. На каждом этапе эволюции появлялись новые технологии, отвечающие на вызовы времени.

1. **1960–1980-е годы: Ранние системы**
   * В первые десятилетия вычислительных систем аутентификация осуществлялась с помощью **логина и пароля**, хранимых локально или в централизованных базах данных.
   * Авторизация представляла собой простой процесс сопоставления прав пользователя с ресурсами.
2. **1990-е годы: Веб-технологии и сессии**
   * С развитием интернета (WWW) появились первые веб-приложения.
   * Аутентификация стала основана на **сессионных cookie**, которые сохраняли информацию о пользователе на стороне клиента.
   * Недостатки ранних подходов, такие как низкая защищенность от кражи cookie, привели к появлению механизма HTTPS для шифрования передаваемых данных.
3. **2000-е годы: OAuth и распределенные системы**
   * В 2006 году появился стандарт **OAuth 1.0**, который обеспечивал безопасный доступ к ресурсам без необходимости передачи паролей.
   * В 2012 году был представлен OAuth 2.0 — усовершенствованная версия, более гибкая и удобная в интеграции.
   * В это же время начало развиваться API-first программирование, где аутентификация через токены стала основным стандартом.
4. **2010-е годы: JWT и микросервисы**
   * В 2015 году был стандартизирован **JWT** (RFC 7519) — удобный способ передачи данных между участниками распределенной системы.
   * JWT стал популярным благодаря простоте работы в микросервисной архитектуре, где серверное хранение сессий было затруднено.
5. **2020-е годы: Zero Trust и усиление безопасности**
   * Современные подходы, такие как **Zero Trust Architecture**, ставят безопасность на первое место. Это привело к широкому использованию мультифакторной аутентификации (MFA), шифрования токенов и динамической авторизации.

**Современные подходы**:

* Появление токенов (например, JWT) стало революцией, так как они упростили работу с распределенными системами.
* OAuth 2.0 предложил решение проблемы делегирования прав доступа между сервисами.

Каждый из методов адаптировался к вызовам времени, включая более строгие требования к безопасности и повышенные ожидания пользователей в плане удобства работы.

**Критерии анализа методов**

Для детального сравнения методов аутентификации и авторизации используются следующие критерии:

1. **Уровень безопасности**:
   * Уязвимость к основным угрозам (CSRF, XSS, утечка данных).
   * Надежность хранения учетных данных и токенов.
2. **Масштабируемость**:
   * Способность метода эффективно работать при увеличении количества пользователей.
   * Загрузка серверов при большом числе запросов.
3. **Производительность**:
   * Время обработки запроса.
   * Влияние на скорость работы веб-приложения.
4. **Удобство использования**:
   * Простота реализации для разработчиков (наличие библиотек, документации).
   * Удобство для конечных пользователей (например, минимальное число действий для входа в систему).

**Проблемы безопасности**

Каждый метод аутентификации и авторизации сталкивается с уникальными угрозами:

* **Session-Based Authentication**:
  + Угрозы перехвата сессионных cookie (например, через XSS).
  + Недостаточная защита от CSRF-атак.
* **JWT**:
  + Риски утечки токенов из-за их хранения на клиентской стороне.
  + Уязвимости в алгоритмах подписи (например, использование слабого алгоритма none).
* **OAuth 2.0**:
  + Атаки через утечку токенов доступа или обновления.
  + Ошибки в конфигурации сервера авторизации.

Для каждой из угроз разрабатываются методы их предотвращения, такие как использование HTTPS, внедрение защиты от CSRF, шифрование токенов и аудит конфигураций.

**Интеграция с современными архитектурами**

Современные веб-приложения часто используют архитектуры на основе микросервисов и облачные решения, что влияет на выбор метода аутентификации и авторизации:

* **Session-Based Authentication**:
  + Ограничена в масштабируемых системах из-за необходимости синхронизации состояния между серверами.
  + Более удобна для монолитных приложений.
* **JWT**:
  + Хорошо подходит для микросервисов, так как каждый сервис может проверить токен локально.
  + Уменьшает нагрузку на серверы аутентификации, так как они не участвуют в каждом запросе.
* **OAuth 2.0**:
  + Подходит для сложных распределенных систем с делегированием прав.
  + Часто используется в связке с провайдерами идентификации (например, Google).

**Подходы к улучшению методов**

Для повышения надежности и удобства работы методов аутентификации и авторизации применяются следующие подходы:

* Использование многофакторной аутентификации (например, комбинации пароля и одноразового кода).
* Разработка централизованных серверов авторизации для упрощения управления токенами.
* Постоянный мониторинг и аудит конфигурации систем для предотвращения атак.

#### ****Выводы теоретического анализа****

Анализ показал, что выбор метода аутентификации и авторизации зависит от специфики приложения. OAuth и JWT подходят для распределенных систем и API, тогда как session-based authentication более удобен для традиционных серверных приложений. Важно учитывать риски, такие как утечка токенов и атаки XSS, и применять дополнительные меры защиты.

### ****Практическая реализация методов с использованием фреймворка FastAPI.****

#### ****Описание этапов реализации****

Практическая часть дипломной работы заключается в реализации трех методов аутентификации и авторизации — OAuth, JWT, и session-based authentication — с использованием фреймворка **FastAPI**. Каждый метод будет продемонстрирован с учетом современных подходов к безопасности, включая защиту от CSRF, XSS и утечек токенов.

#### ****Архитектура веб-приложения****

Приложение реализовано на основе **модульной архитектуры**, где каждая реализация метода аутентификации и авторизации (Session-Based Authentication, JWT, OAuth2) изолирована в своем собственном модуле. Это позволяет гибко тестировать, дорабатывать и внедрять разные методы аутентификации независимо друг от друга.

**Основные уровни архитектуры:**

1. **Уровень маршрутов (Routing Layer)**:
   * В каждом модуле определены маршруты для взаимодействия пользователя с системой: регистрация, вход в систему, доступ к защищенным ресурсам.
   * Реализованы эндпоинты для работы с токенами (JWT, OAuth2) или сессиями.
2. **Уровень логики (Business Logic Layer)**:
   * Логика проверки токенов, сессий и прав доступа.
   * Обработка данных пользователей (хэширование паролей, валидация данных).
   * В Session-Based Authentication реализована работа с базой данных для хранения пользовательских данных и сессий.
3. **Уровень данных (Data Layer)**:
   * Используется база данных SQLite для хранения пользователей и сессий (Session-Based Authentication).
   * В JWT и OAuth2 авторизация статусы пользователей и токены хранятся на стороне клиента или в памяти.
4. **Шаблонный уровень (Presentation Layer)**:
   * HTML-шаблоны Jinja2 предоставляют визуальный интерфейс для взаимодействия с системой.
   * Страницы для входа, регистрации, домашней страницы и профиля пользователя.
5. **Уровень безопасности (Security Layer)**:
   * Включает хэширование паролей (hashlib), защиту от CSRF-атак (Session-Based Authentication), а также проверку токенов (JWT и OAuth2).

**Структура приложения**

****

**Использованные технологии**

**1. Python**

* Основной язык программирования для реализации бизнес-логики и работы с веб-фреймворком.

**2. FastAPI**

* Фреймворк для разработки RESTful API, который обеспечивает:
  + Простую интеграцию с Jinja2 для рендеринга HTML-шаблонов.
  + Удобную маршрутизацию и асинхронность.
  + Генерацию интерактивной документации через Swagger и Redoc.
  + Встроенные методы безопасности (модуль security)

**3. SQLite**

* Легковесная реляционная база данных для хранения пользовательских данных и сессий (Session-Based Authentication).

**4. Alembic**

* Используется для управления миграциями базы данных (Session-Based Authentication).

**5. Jinja2**

* Шаблонизатор для создания HTML-страниц. Позволяет реализовать:
  + Общую структуру через базовый шаблон (base.html).
  + Динамические элементы, такие как отображение профиля пользователя или списка пользователей.

**6. JSON Web Token (JWT)**

* Используется для аутентификации пользователей. Токены содержат полезную нагрузку (payload) и подписаны секретным ключом.
* Особенности реализации:
  + Генерация и проверка токенов через библиотеку PyJWT.
  + Обеспечение безопасности данных с помощью HMAC SHA256.

**7. OAuth 2.0**

* Протокол авторизации, позволяющий пользователям делегировать права доступа третьим сторонам без передачи учетных данных.
* Особенности реализации:
  + Конфигурация клиентов через .env файл.
  + Генерация и проверка токенов доступа.
  + Создание OAuth сервиса авторизации с помощью Google APIs & Services

**8. hashlib, secrets**

* **Hashlib** используется для хэширования пользовательских паролей, что предотвращает их хранение в открытом виде.
* **Secrets** используется для генерации пользовательских сессий

**9. Утилиты**

* Библиотеки для обеспечения функциональности:
  + python-dotenv для работы с конфигурационным файлом .env.
  + uvicorn для запуска сервера.
  + fastapi.security для настройки OAuth2 и работы с токенами.

**10. Bootstrap (CSS)**

* Использован для стилизации интерфейса. Обеспечивает:
  + Адаптивный дизайн HTML-страниц.
  + Быстрое создание пользовательского интерфейса без необходимости написания кастомного CSS.

**Потоки данных**

1. **Session-Based Authentication**:
   * Пользователь отправляет учетные данные (логин/пароль) через форму.
   * Сервер проверяет данные, создает запись сессии в базе данных и возвращает cookie с идентификатором сессии.
   * При последующих запросах сервер сверяет идентификатор сессии с базой данных для предоставления доступа.
2. **JWT**:
   * Пользователь вводит логин/пароль, и сервер генерирует JWT, включающий полезную нагрузку (payload) с данными пользователя.
   * Токен отправляется клиенту, с помощью cookie
   * При каждом запросе клиент передает токен, а сервер проверяет его подлинность.
3. **OAuth 2.0**:
   * Пользователь перенаправляется на сервер авторизации.
   * После успешной аутентификации сервер авторизации возвращает токен доступа.
   * Токен используется для доступа к защищенным ресурсам.

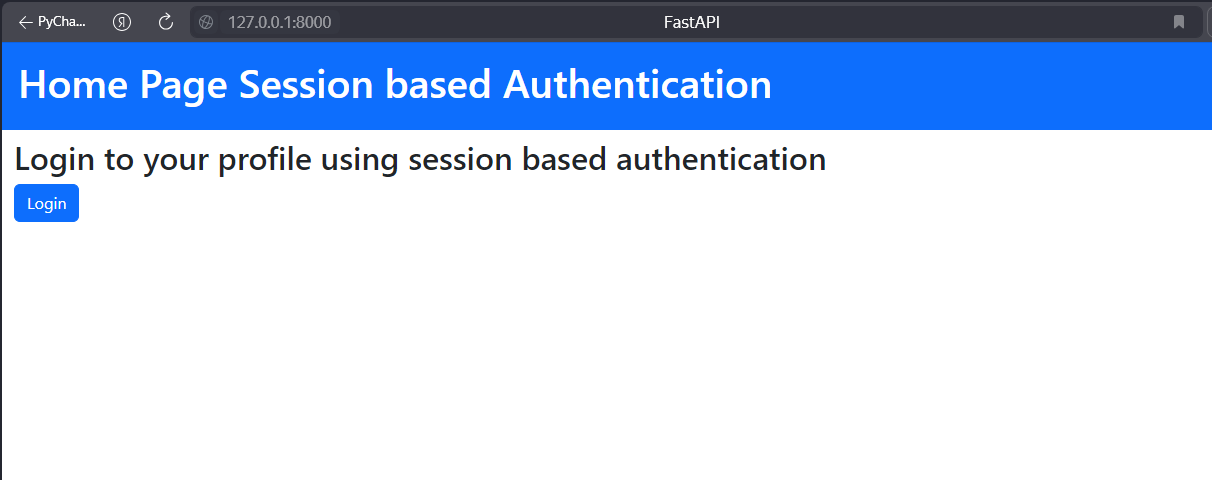
**Особенности безопасности**

1. **Хэширование паролей**: Все пароли пользователей шифруются с помощью hashlib.
2. **CSRF защита**: Используются cookie с атрибутами HttpOnly и Secure для предотвращения атак.
3. **Шифрование токенов**: JWT подписываются с использованием алгоритма HMAC SHA256, что предотвращает их подделку.
4. **Защита данных**: При деплое приложения рекомендуется использовать HTTPS для передачи конфиденциальной информации.

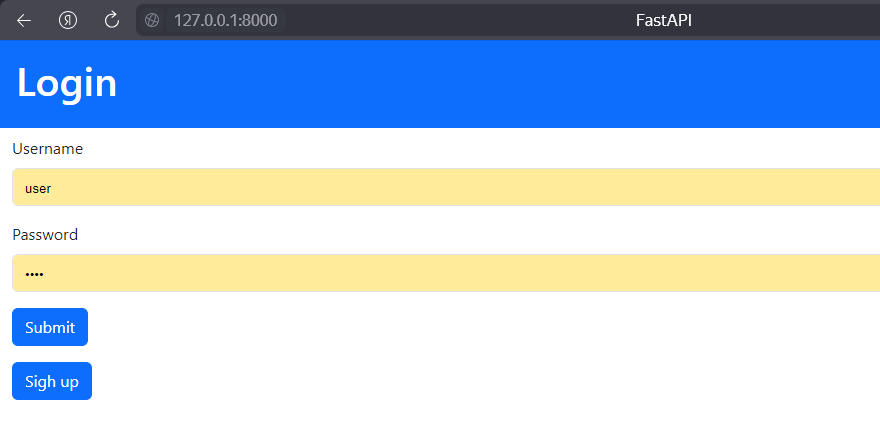
**Особенности запуска**

**Запуск приложений необходимо осуществлять по отдельности через main.py в корне проекта либо через консоль командой “python main.py”. Для запуска конкретного приложения раскомментируйте блок. В дальнейшем можно будет запускать через Docker образы и docker-compose**

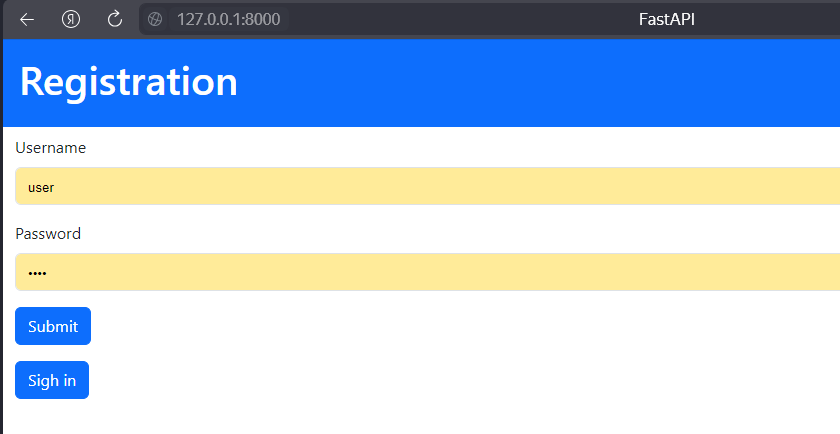
**Домашняя страница Session-based authentication**

****

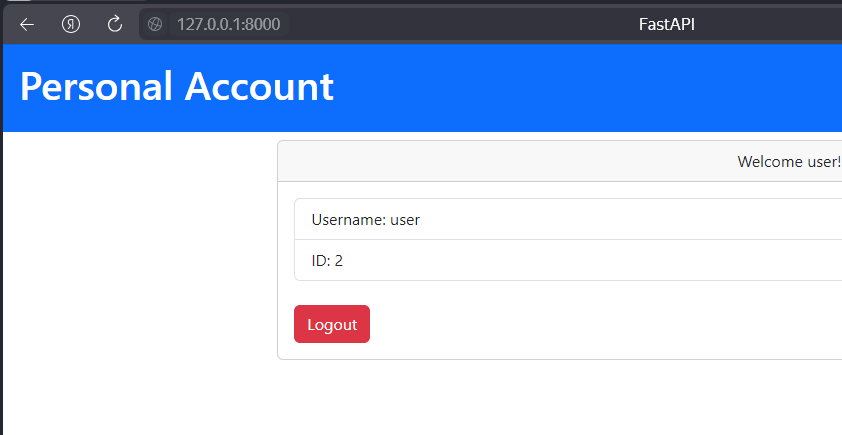
**Страница входа Session-based authentication**

****

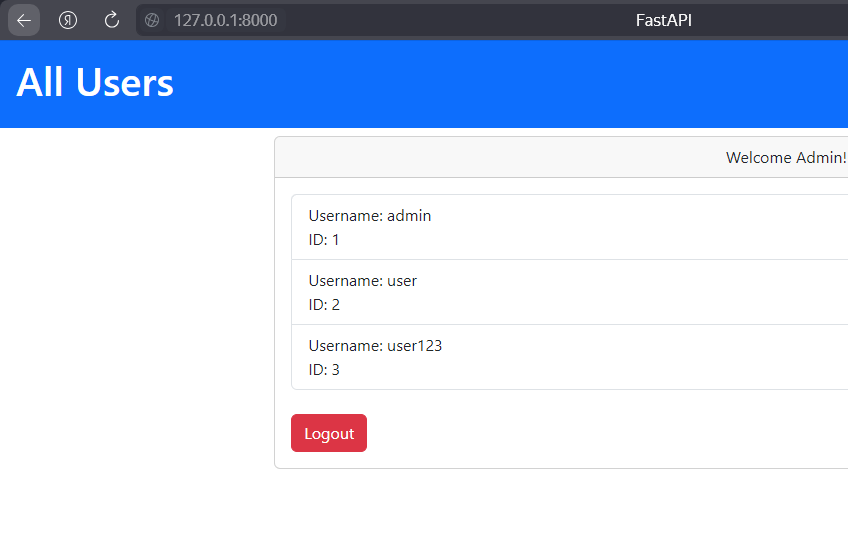
**Страница регистрации Session-based authentication**

****

**Страница профиля Session-based authentication**

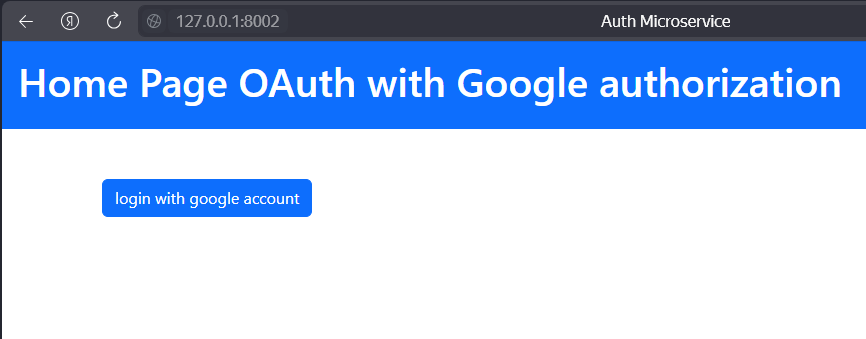
****

**Страница all\_users Session-based authentication**

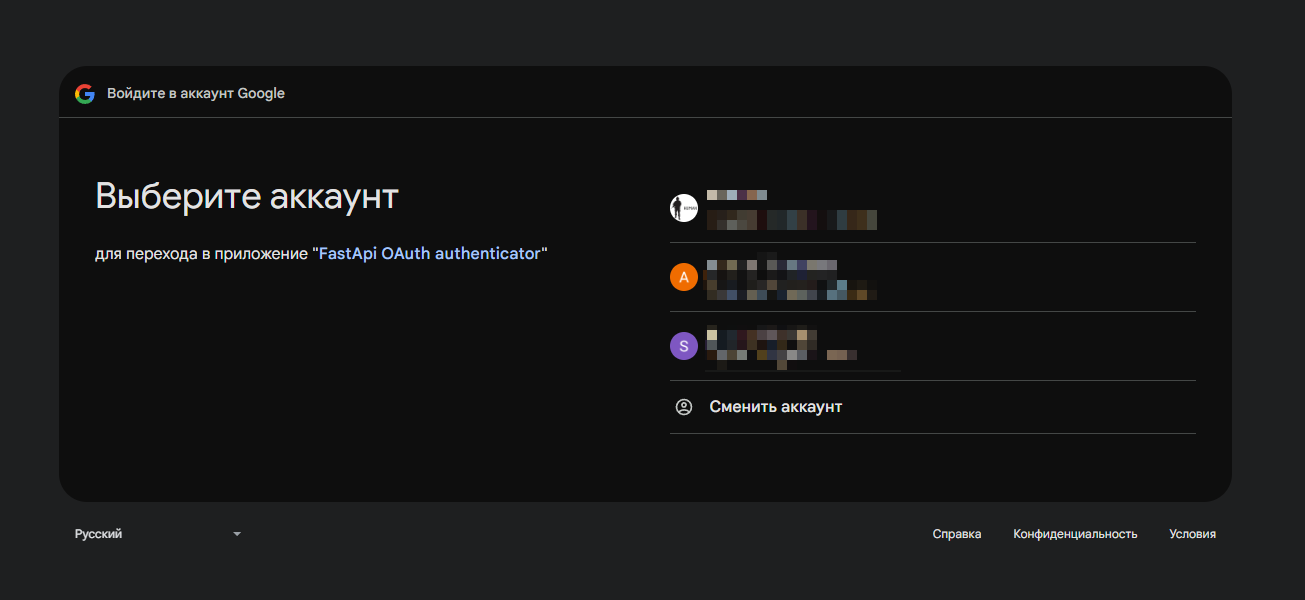
****

**Страницы JWT authentication выполнены по тем же шаблонам, разница видна только в реализации на бекенде**

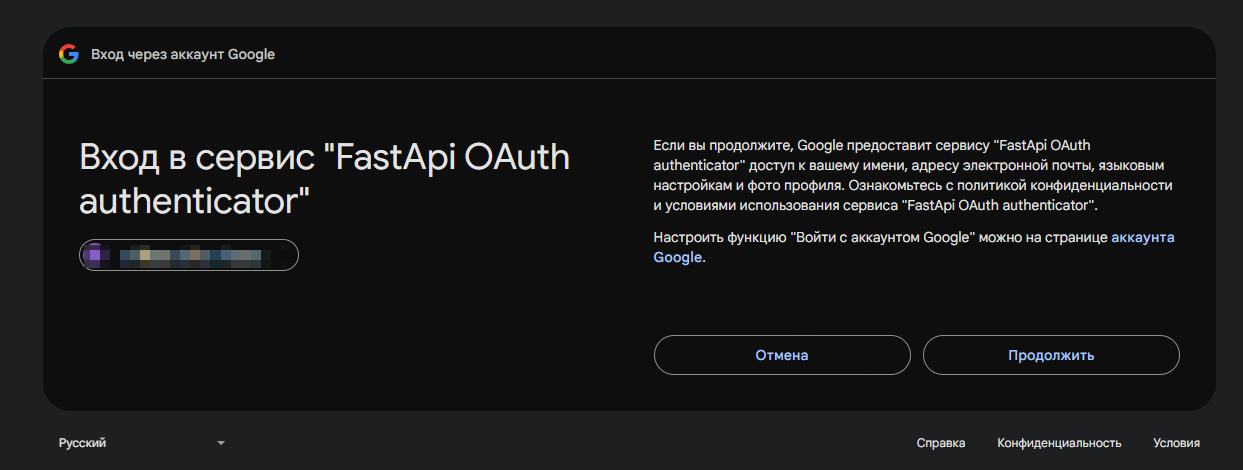
**Домашняя страница OAuth authentication**



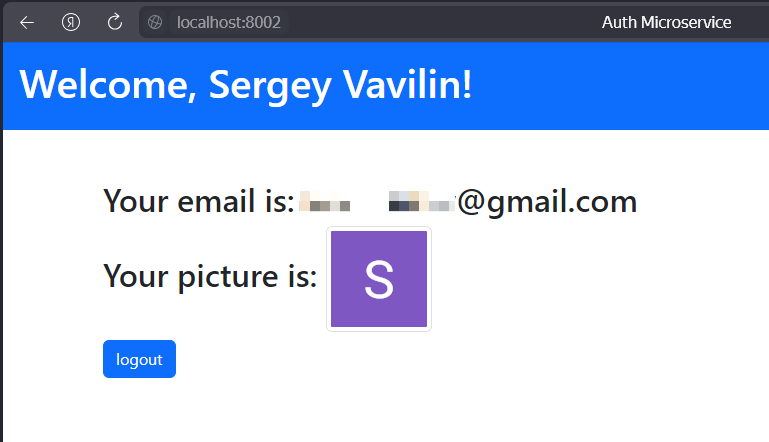
**Страница входа OAuth authentication**



**Страница подтверждения OAuth authentication**

****

**Страница профиля OAuth authentication**

****

### ****Результаты и выводы, включая рекомендации для разработчиков.****

На основе проведенного анализа, практической реализации и сравнительного изучения методов аутентификации и авторизации в веб-приложениях были получены следующие результаты:

**1. Теоретические выводы**

* **Session-Based Authentication**:  
  Традиционный и хорошо изученный метод, подходящий для серверно-ориентированных монолитных приложений. Обеспечивает высокий уровень безопасности при использовании современных механизмов защиты (например, HttpOnly cookie). Однако ограничивает масштабируемость и требует значительных серверных ресурсов для хранения сессий.
* **JWT**:  
  Эффективный способ передачи аутентификационных данных без необходимости хранения состояния на сервере. Идеален для распределенных систем и микросервисов. Однако требует тщательного управления токенами для предотвращения утечек и обеспечения их безопасности.
* **OAuth 2.0**:  
  Оптимален для делегирования прав доступа сторонним приложениям. Его сложность компенсируется возможностями интеграции с крупными платформами. Требует строгой настройки безопасности для предотвращения утечек токенов и атак через сервер авторизации.

**2. Результаты практической реализации**

Реализация всех трех методов в веб-приложениях с использованием FastAPI подтвердила их работоспособность и удобство интеграции в Python-экосистему.

* **Session-Based Authentication** показала простоту в реализации, однако потребовала использования дополнительных инструментов для хранения сессий
* **JWT** обеспечил высокую производительность за счет отсутствия необходимости постоянного взаимодействия с базой данных для проверки токенов.
* **OAuth 2.0** потребовал большего объема настройки, но обеспечил возможность подключения стороннего сервиса для управления аутентификацией.

**Рекомендации для разработчиков**

1. **Выбор метода аутентификации и авторизации**:
   * Для **монолитных приложений**, где серверное состояние легко контролируется, рекомендуется использовать **Session-Based Authentication**. Это обеспечит высокий уровень безопасности и простоту реализации.
   * Для **микросервисных архитектур** и **SPA-приложений** на клиентской стороне (Single Page Applications) предпочтителен **JWT**, так как он упрощает взаимодействие между сервисами.
   * Для приложений, которые предоставляют доступ сторонним сервисам, оптимален **OAuth 2.0**, особенно при необходимости интеграции с платформами, поддерживающими этот протокол.
2. **Усиление безопасности**:
   * Для **сессий** используйте безопасные cookie с атрибутами **HttpOnly, Secure,** а также внедрите защиту от CSRF.
   * Для **JWT**:
     + Храните токены в **HttpOnly** cookie или в защищенных хранилищах.
     + Регулярно обновляйте токены с помощью механизмов рефрешинга.
   * Для **OAuth 2.0**:
     + Настраивайте минимальные права доступа (scopes) для токенов.
     + Используйте шифрование токенов и безопасные каналы связи (HTTPS).
3. **Удобство разработки и поддержки**:
   * Для небольших команд или приложений с ограниченными ресурсами выбирайте простые в реализации методы (например, Session-Based Authentication).
   * Автоматизируйте проверку конфигурации безопасности и настройку среды с помощью инструментов CI/CD.
4. **Масштабируемость**:
   * Используйте распределенные хранилища для сессий (например, Redis) при большом числе пользователей.
   * Для масштабируемых приложений отдавайте предпочтение безсессионным методам (JWT или OAuth 2.0).
5. **Интеграция с внешними сервисами**:
   * Для интеграции с платформами (Google, GitHub) используйте **OAuth 2.0** или его надстройки (например, OpenID Connect).
   * Разрабатывайте серверы авторизации с учетом стандартов безопасности и тестируйте их на уязвимости.

**Общие выводы**

Каждый метод имеет свою область применения, и универсального решения для всех задач не существует. Однако сочетание нескольких подходов (например, использование **JWT** для API и **Session-Based Authentication** для веб-интерфейса) может быть наиболее эффективным в сложных системах. Выбор подходящего метода должен быть основан на специфике проекта, уровне ожидаемой нагрузки, требованиях к безопасности и удобству использования как для разработчиков, так и для конечных пользователей.