МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ ВЕБ-ПРИЛОЖЕНИЯ И ЕГО ИНТЕГРАЦИЯ С OAUTH2 НА ЯЗЫКЕ JAVA С ИСПОЛЬЗОВАНИЕМ SPRING FRAMEWORK

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

Студентки 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Бебешко Алёны Викторовны

Научный руководитель
доцент, к. ф.-м. н. ______________ А. С. Иванова

Заведующий кафедрой
к. ф.-м. н., доцент ______________ С. В. Миронов

Саратов 2023

ВВЕДЕНИЕ

В настоящее время, когда растет количество крупных многопользовательских сервисов, у заказчиков программных продуктов возникает потребность в надежном и проверенном решении вопроса о том, как обеспечить безопасность данных пользователей. В том числе на этапе авторизации. Одним из возможных решений может выступать использование Google oAuth2 авторизационного протокола.

В теоретической части данной работы описываются принципы работы протокола, а так же jwt-токенов, использующихся в oAuth2 для передачи информации, их преимущества и недостатки с точки зрения разработки и использования. В практической части данной работы описывается процесс реализации API веб-приложения, подробно рассматривается реализация интеграции этого приложения с oAuth2 и проводится сравнение данного метода авторизации с альтернативным.

Актуальность работы может быть обусловлена с одной стороны возрастанием популярности oAuth2 протокола, и с другой стороны недостаточной, на мой взгляд, изученностью его уязвимостей и недостатков.

Таким образом, целью настоящей работы является проектирование и разработка серверной части виртуального интернет-магазина и интеграция приложения с Google oAuth2 протоколом.

Поставленная цель определила следующие задачи:

- исследовать принципы работы авторизационного протокола Google oAuth2;
- разработать restful API виртуального магазина;
- использовать Google oAuth2 протокол для авторизации пользователей;
- описать практические преимущества и недостатки использования oAuth2 для авторизации пользователей.

В ходе работы использовались следующие программные средства и технологии. Со стороны сервера — язык Java, фреймворк Spring. Со стороны клиента — язык JavaScript, фреймворк ReactJS. Для хранения данных — PostgreSQL. Для развертывания базы данных — Docker. Для тестирования разработанного функционала — Postgres. И в ходе интеграции с oAuth2 — Google Cloud Platform.

1 Теоретические основы

Спецификация oAuth2 определяет протокол делегирования, который предоставляет клиентам безопасный доступ к ресурсам пользователя на сервисе-провайдере. Такой подход избавляет пользователя от необходимости вводить пароль за пределами сервиса-провайдера: весь процесс сводится к нажатию кнопки «Согласен предоставить доступ к ..». Идея в том, что имея один хорошо защищенный аккаунт, пользователь может использовать его для аутентификации на других сервисах, не раскрывая при этом своего пароля. [1], [5], [6], [7]

Общая схема выполнения oauth2 authentication flow может быть описана следующим образом.

1. Клиент запрашивает авторизацию у владельца ресурса;
2. Клиент получает грант авторизации;
3. Клиент запрашивает токен доступа путем аутентификации с помощью сервера авторизации и предоставление гранта авторизации;
4. Сервер авторизации аутентифицирует клиента, проверяя грант авторизации и, если он действителен, выдает токен доступа (access token) и рефреш токен (refresh token);
5. Клиент запрашивает защищенный ресурс у провайдера и аутентифицируется, представляя токен доступа;
6. Провайдер проверяет токен доступа и, если он действителен, обслуживает запрос.

Дальше клиент общается с провайдером данных, пока у access токена не кончится срок годности. Затем, чтобы получить доступ к данным провайдера снова, клиенту нужно воспользоваться refresh токеном и отправить запрос на генерацию новых refresh token и access token.

2 Имплементация функционала

2.1 Реализация oAuth2 flow

В ходе реализации oAuth2-авторизации с использованием jwt-токена был разработан ряд классов, отвечающих за данный функционал. Кратко разберем их методы и значения.

oAuth2 client **настройка**

Алгоритм создания и настройки проекта в Google Cloud Consol

1. Для начала создадим новый проект в Google Cloud Consol и создадим Client ID для веб-приложения;
2. После этого справа можно будет увидеть Client ID и Client Secret. Эти значения нужно будет прописать в properties-приложения;
3. В раздел Authorized Redirect URIs добавим адреса страниц, на которые будут происходить перенаправления в ходе аутентификации;
4. Выберем необходимые Scopes и определим таким образом, какие именно данные о гугл-аккаунте пользователя может получить наше приложение. Соответствующие Client ID в дальнейшем укажем в запросе к Google API;
5. Добавим ограниченный круг тестовых пользователей, которые смогут аутентифицироваться через Google, пока приложение не функционирует в production-среде.

Класс SecurityConfig. Здесь мы добавляем corsFilter, говорим Spring Security, что не нужно использовать сессии (так как мы собираемся использовать cookies с токенами), задаем базовый uri для запросов на аутентификацию и uri, по которому будут обращаться успешно авторизованные запросы.

С помощью аннотации @Bean мы инжектим класс HttpCookieOAuth2AuthorizationRequestRepository — репозиторий, который будет использоваться для хранения авторизованных запросов. И аналогичным образом инжектим TokenAuthenticationFilter — фильтр, который будет отвечать за получение jwt из хедеров каждого запроса к серверу и за дальнейшую логику по его обработке. Про этот и другие способы инжекта бинов можно прочитать в источнике [8]

Класс AppConfig. Это класс, в котором мы определяем secretKey, на основе которого будет шифровать токены, tokenExpirationMsec — время, через которое просрочится токен и authorizedRedirectUris — uri,на которой пользователь будет перенаправляться после успешной авторизации. Задание обозна-

ченных значений происходит в файле application.properties.

Класс WebMvcConfig. В нем глобально включаем CORS. (Подробнее с ними можно ознакомиться здесь [9])
.allowedMethods() разрешает принимать на сервер запросы определенных типов.
.allowedHeaders("*") разрешает использовать во входящих запросах любые заголовки.
.allowCredentials(true) определяет, должен ли браузер отправлять учетные данные (например cookie) вместе с междоменными запросами в аннотрованную конечную точку. То есть веб-сайт в другом домене может отправить учетные данные пользователя, выполнившего вход в приложение, от имени пользователя без ведома пользователя.

Класс TokenAuthenticationFilter**.** Этот класс является нашим собственным пользовательским фильтром. Причем это будет такой фильтр, который будет выполняться до выполнения сервлета и только один раз. Даже в случае если в цепочке фильтров этот фильтр вызовется дважды. Подробнее о принципах работы фильтров можно прочитать в источниках [10], [11], [12].

В этом классе мы переопределяем метод doFilterInternal. Он будет вызываться при обращении по адресу, где слушает сервер при каждом входе в систему. Сначала мы получаем jwt-токен из HttpSrvletRequest и получаем из этого токена userId с помощью функции getUserIdFrpmToken.

Затем с помощью userService.loadUserById(userId) смотрим, есть ли уже в базе пользователь с таким id. Если есть, то с помощью UserPrincipal.create получаем из базы данных информацию о пользователе (id, email, authorities). Если же пользователя с таким id нет, то loadUserById кидает Exception.

Если userService.loadUserById вернула данные о пользователе, то создаем объект класса Authentiction (вернее его подкласса UsernamePasswordAuthenticationToken) с использованием полученных UserDetails.

Объект класса Authentication представляет пользователя с точки зрения Spring Security. То есть, создавая новый объект класса Authentication, мы создаем нового пользователя в Spring Security.

Созданный объект добавляем в Security Context, который как раз и предназначен для хранения объектов Authentication. Затем вызываем

filterChain.doFilter, который вызывает переход к выполнению следующего фильтра в цепочке или к сервлету, если больше фильтров в цепочке нет.

Класс TokenProvider**.**

Здесь реализован метод getUserIdFromToken, в котором мы из jwt получаем claims, то есть части информации зашифрованной в токене. Кроме того в это классе определены методы createToken и validateToken для создания и валидации токенов.

В ходе валидации токена обрабатываются и логгируются все возможные случаи, при которых полученный токен может быть некорректным.

Класс CustomOAuth2UserService**.**

В классе CustomOAuth2UserService функция loadUser загружает информацию о пользователе из запроса к endpoint и возвращает processOauth2User от двух аргументов.

processOauth2User из запроса к endpoint получает идентификатор регистрации, из информации о юзере получает его атрибуты (атрибуты токена) и эти значения передает в функцию getOAuth2UserInfo, которая возвращает сущность, из которой можно получить id, name и email для сохранения в базу. После этого в зависимости от наличия пользователя в базе происходит или save нового пользователя в базу или update уже существующего.

Класс HttpCookieOAuth2AuthorizationRequestRepository**.** Данный класс имплементирует AuthorizationRequestRepository<OAuth2AuthorizationRequest>, который является cookie-based репозиторием для хранения аутентентификационных запросов. Данный класс использует класс CookieUtils — класс для получения, добавления, удаления, а также сериализации и десериализации cookie.

Метод getCookie считывает cookie используя класс HttpServletRequest. В материале [13] представлены этот и другие способы получения cookie. Метод getCookie извлекает cookie из запроса, проверяя его на null на нулевую длину. В классе HttpCookieOAuth2AuthorizationRequestRepository вызывается методом loadAuthorizationRequest.

addCookie принимает ответ, который сервер собирается отослать клиенту, а также имя cookie, её значение и время, в течении которого будет храниться cookie. В addCookie мы делаем следующее:

1. Создаем новую cookie;
2. Задаем то, какие страницы смогут её видеть. В нашем случае это все страницы, являющиеся поддиректориями директории "\";
3. задаем setHtttOnly(true), чтобы наша cookie оставалась невидимой для js-скриптов;
4. Задаем время, через которое cookie просрочится;
5. Добавляем cookie к ответу.

addCookie используется методом saveAuthorizationRequest.

Процедура удаления cookie производится через изменение аналогичных параметров.

Процедура serialize кодирует cookie с помощью Base64, процедура deserialize декодирует их.

Теперь перейдем к методам класса.

HttpCookieOAuth2AuthorizationRequestRepository используется для хранения authorization requests в cookies в течении некоторого времени после входа пользователя. Это нужно для того, чтобы не проводить выдачу токена повторно при каждом запросе. С помощью класса HttpCookieOAuth2AuthorizationRequestRepository мы говорим спрингу, что OAuth2 flow уже был выполнен для этого пользователя и результат находится в cookie.

В используемом нами пакете spring-boot-starter-oauth2-client есть свой HttpSessionOAuth2AuthorizationRequestRepository для сохранения authorization requests, наше приложение stateless. Следовательно не будут использоваться сессии и сессионные cookies. Поэтому мы описываем свой класс для работы с cookies. Прочитать о работе с сессия и о разнице между stateless и statefull приложениями возможно в источниках [14], [15], [16].

Процедура loadAuthorizationRequest возвращает набор десериализованных cookies.

Если authorizationRequest == null, то удаляем наши cookies, если — нет, то добавляем в responce сериализованный authorizationRequest и redirectUriAfterLogin и задаем время действия cookies 3 минуты. redirectUriAfterLogin получаем их request.

В классе HttpCookieOAuth2AuthorizationRequestRepository есть также процедура removeAuthorizationRequest, которая воспроизводит дефолтный ме-

тод, описанный в имплементируемом интерфейсе, и процедура removeAuthorizationRequestCookies, которая опустошает cookies.

Класс OAuth2AuthenticationSuccessHandler**.**

Внутри класса определен метод isAuthorizedRedirectUri, который проверяет, совпадает ли redirectUri из cookies запроса с одним из значений списка isAuthorizedRedirectUris, значения которого заданы в конфигах приложения.

В процедуре determineTargetUrl мы получаем redirectUri из cookies и с помощью isAuthorizedRedirectUri. Присваиваем переменной targetUri значение redirectUri, если оно корректно извлекается из Optional, а если нет, то присваиваем ему defaultTargetUri (то есть "/"). Таким образом в случае если нам поступит запрос без валидного redirectUri в cookie или такого cookie вообще не будет, то пользователь будет перенаправлен в корень сайта. В переменную token кладем jwt-токен, который создаем на основе данных из authentication.

Метод onAuthenticationSuccess сработает сразу после успешной авторизации. Он отвечает за формирование uri с использованием двух описанных выше функций, на который пользователь будет перенаправлен, и за редирект пользователя.

Класс OAuth2AuthenticationFailureHandler**.**

Класс OAuth2AuthenticationFailureHandler содержит метод onAuthenticationFailure

Здесь все происходит так же как и в onAuthenticationSuccess. Разница только в том, что вместо переменной token в targetUri добавляетя переменная error, содержащая сообщение об ошибке.

Код реализованной функциональности приведен в приложении Г

2.2 Конфигурация и подключение базы данных

Вместо локальной базы данных будем использовать образ PostgreSQL, который поднимем в контейнере докера. В ходе разработки учебного тренировочного сервиса данное решение может казаться излишне сложным с точки зрения настройки и подключения, однако навык работы докер-контейнерами и иными инструментами devops разработчика чрезвычайно важен на продакшн-проекте.

Создадим файл dockerCompose.yml где пропишем конфигурацию будущего контейнера, в которой будем делать следующее. Здесь мы подтягиваем из Docker Hub образы PostgreSQL и PgAdmin. Первое — база данных, второе —

сервис предоставляющий интерфейс и средства манипуляции этими данными. Все логины и пароли, используемые в коде являются тестовыми и не являются конфиденциальной информацией.

Выполняя команды:

— docker login
— docker -t build -t store-api-new
— docker compose up

Таким образом мы:

— логинимся на Docker Hub
— собираем докер-файл и скачиваем нужные образы
— поднимаем сервисы на указанных в коде портах

Файл application при этом заполняем так, чтобы password, username и порт и домен в url обязательно совпадали с данными из dockerCompose.

Далее остается только залогиниться в PdAdmin.

2.3 Реализация API

В ходе проектирования и разработки API были созданы следующие сущности

— User: информация о пользователях;
— Product: доступные для покупки товары;
— Order: заказы пользователей;
— Cart: хранится в сессии пользователя.

Тестирование проводилось с помощью Postman с опорой на методы из источника [17]. Краткий обзор API представлен на рисунке 1.

cart-controller : Cart Controller — Show/Hide | List Operations | Expand Operations

GET /cart get

POST /cart addItem

PUT /cart updateItem

DELETE /cart/{id} deleteItem

order-controller : Order Controller — Show/Hide | List Operations | Expand Operations

GET /orders getAll

POST /orders checkout

DELETE /orders/{id} cancel

product-controller : Product Controller — Show/Hide | List Operations | Expand Operations

GET /products getAll

user-controller : User Controller — Show/Hide | List Operations | Expand Operations

POST /register registerUser

Рисунок 1 – API

API было разработано в соответствии с RESTful принципами. Это архитектурные принципы, подразумевающие согласно источникам [18], [19] следующее:

— Клиент-серверная архитектура;
— Связь сервера и клиента без сохранения состояния;
— Единый интерфейс для доступа к ресурсам и управления ими;
— Ориентированность на ресурсы;
— Поддержка операций без сохранения состояния;
— Кэшируемость ответов.

2.4 Реализация кастомной аутентификации

Для сравнения с oAuth2 была реализована кастомная аутентификация средствами Spring Security с использованием источников [20] и [21].

Рассмотрим и разберем типичный сценарий авторизации пользователя в системе средствами Spring Security. Он схематично представлен на рисунке 2.

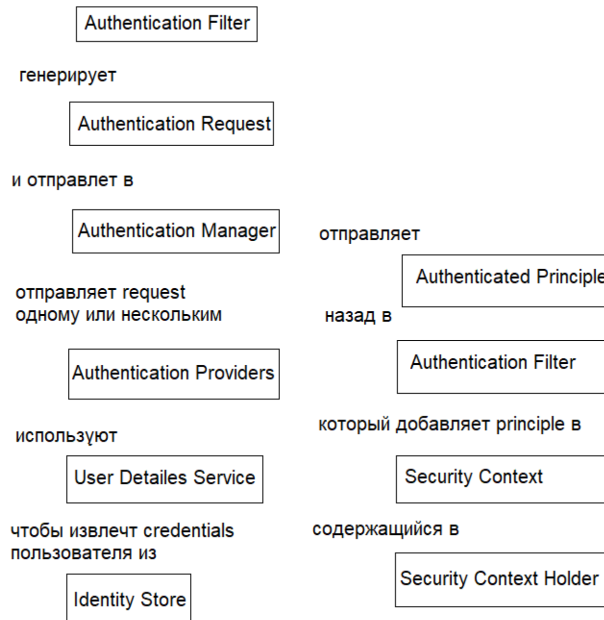

Рисунок 2 – Типичный сценарий авторизации

Чтобы использовать определенный тип аутентификации Spring Security, надо настроить соответствующий AuthenticationFilter (Например, BasicAuthenticationFilter, OpenIDAuthenticationFilter и так далее). В нашем случае будем использовать UsernamePasswordAuthenticationFilter. В зависимости от типа AuthenticationFilter будут формироваться разные типы AuthenticationRequestToken внутри AuthenticationRequest. Можно написать свой собственный фильтр и внедрить его Spring Security Filter Chain. А если, например, необходимо авторизовывать админа и обычного пользователя разным образом, то можно создать два кастомных фильтра и разграничить их области действия, прописав в методе matches соответствующие URL.

Есть также разные AuthenticationProvider. Например, OpenIDAuthenticationProvider, DaoAuthenticationProvider. В нашем случае используем DaoAuthenticationProvider.

Для UserDetailsService также есть имплементации такие как InMemoryUserDetailsManager, LdbcUserDetailsManager и так далее. Однако чаще прописывается собственный переопределенный сервис.

Важно отметить, что Security Context Holder не является thread-local, а значит AuthenticationPrinciple доступны только в текущем потоке.

SecurityContextPersistentFilter срабатываем первым из всей цепочки. Он пытается извлечь SecurityContext из SecurityContextRepository. Если пользователь уже авторизован, то извлечение SecurityContext, проходит успешно. А из контекста извлекается уже AuthenticationPrinciple. Если пользователь не авторизован, то в SecurityContextHolder добавится пустой SecurityContext. Затем в ходе работы других фильтров в SecurityContext добавляется AuthenticationPrinciple с данными о пользователе. Этот механизм позволяет пользователю не входить в систему повторно до тех пор, пока сессия не закончится или он не сделает log out.

Для реализации аутентификации переопределим ряд классов.

— Во-первых, добавим наш кастомный AuthenticationFilter в очередь фильтров спринга. Код приведен в приложении А;
— Затем переопределим UserDetailsService, где пропишем логику работы с IdentityStore, в качестве которого будет выступать база данных. Код приведен в приложении Б;
— И переопределим UserPrincipal, чтобы хранить там дополнительную информацию. Код приведен в приложении В.

2.5 Выводы об эффективности технического решения

Было реализовано два различных метода аутентификации для одного и то же веб-приложения. Наблюдения сделанные в ходе реализации и тестирования обоих методов позволяют выделить следующие преимущества и недостатки протокола oAuth2.

Преимущества oAuth2 состоят в следующем.

1. Повышенная безопасность. oAuth2 позволяет пользователям предоставлять доступ к своим ресурсам mintinlinejavaGoogle, не передавая свои фактические учетные данные сторонним приложениям;
2. Ограниченный доступ. Приложение получает доступ только к ресурсам, явно разрешенным пользователем, что снижает риск несанкционированного раскрытия данных;
3. Согласие пользователя. oAuth2 требует явного согласия пользователя перед предоставлением доступа к его ресурсам;
4. Интеграция с API Google.

С другой стороны oAuth2 имеет некоторые недостатки.

1. Сложность имплементации;

2. Многоэтапное взаимодействие с пользователем;
3. Зависимость от сторонних поставщиков;
4. Ограниченная настройка;
5. Необходимость доверия к сторонним приложениям.

Таким образом, важно отметить, что, хотя $\mathrm{oAuth2}$ отвечает многим общим требованиям, он может быть не оптимальным решением для каждого сценария. При определении того, является ли $\mathrm{oAuth2}$ наиболее подходящим протоколом аутентификации и авторизации для конкретного приложения, следует учитывать такие факторы, как уровень контроля, сложность и совместимость с существующими системами.

Код реализованной функциональности приведен в приложении Г

ЗАКЛЮЧЕНИЕ

В настоящей работе спроектирована и разработана серверная часть виртуального интернет-магазина и выполнена его интеграция с Google oAuth2 протоколом.

Поставленная цель определила следующие задачи:

— исследованы принципы работы авторизационным протоколом Google oAuth2;
— разработана restful API виртуального магазина;
— успешно использован Google oAuth2 протокол для авторизации пользователей;
— на практике выявлены и описаны преимущества и недостатки использования oAuth2 для авторизации пользователей.