**Автоматическая настройка Spring Boot OAuth2 (с использованием Устаревшего стека)**

Spring Boot имеет специальную поддержку автоконфигурации для OAuth2.

Поддержка [OAuth Spring Security,](https://docs.spring.io/spring-security-oauth2-boot/docs/current/reference/html5/) которая входила в Spring Boot 1.x, была удалена в более поздних версиях загрузки вместо первоклассной поддержки OAuth, которая поставляется вместе с [Spring Security 5](https://docs.spring.io/spring-security/site/docs/current/reference/html5/" \l "oauth2) . Мы увидим, как это использовать, в следующем разделе.

Для устаревшего стека (с использованием Spring Security OAuth) сначала нам нужно добавить зависимость Maven, чтобы начать настройку нашего приложения:

<**dependency**>

<**groupId**>org.springframework.security.oauth</**groupId**>

<**artifactId**>spring-security-oauth2</**artifactId**>

</**dependency**>

Эта зависимость включает набор классов, способных запускать механизм *автоконфигурации,* определенный в классе *OAuth2AutoConfiguration*.

Теперь у нас есть несколько вариантов продолжения, в зависимости от области применения нашего приложения.

**5.1. Автоконфигурация сервера авторизации OAuth2**

Если мы хотим, чтобы наше приложение было поставщиком OAuth2, мы можем использовать *@EnableAuthorizationServer* .

При запуске мы заметим в журналах, что классы автоконфигурации будут генерировать идентификатор клиента и секрет клиента для нашего сервера авторизации и, конечно же, случайный пароль для базовой аутентификации.

Using default security password: a81cb256-f243-40c0-a585-81ce1b952a98

security.oauth2.client.client-id = 39d2835b-1f87-4a77-9798-e2975f36972e

security.oauth2.client.client-secret = f1463f8b-0791-46fe-9269-521b86c55b71

Эти учетные данные можно использовать для получения токена доступа:

curl -X POST -u 39d2835b-1f87-4a77-9798-e2975f36972e:f1463f8b-0791-46fe-9269-521b86c55b71 \

-d grant\_type=client\_credentials

-d username=user

-d password=a81cb256-f243-40c0-a585-81ce1b952a98 \

-d scope=write http://localhost:8080/oauth/token

В нашей [другой статье вы](https://www.baeldung.com/rest-api-spring-oauth2-angular-legacy#server) найдете более подробную информацию по этому поводу.

**5.2. Другие параметры автоконфигурации Spring Boot OAuth2**

Есть и другие варианты использования Spring Boot OAuth2, например:

1. [Сервер ресурсов](https://www.baeldung.com/rest-api-spring-oauth2-angular-legacy#resource) - *@EnableResourceServer*
2. [Клиентское приложение](https://www.baeldung.com/sso-spring-security-oauth2-legacy) - *@ EnableOAuth2Sso* или *@ EnableOAuth2Client*

Если нам нужно, чтобы наше приложение было одним из перечисленных выше типов, нам просто нужно добавить некоторую конфигурацию в свойства приложения, как подробно описано по ссылкам, указанным выше.

Все специфические свойства OAuth2 можно найти в [общих свойствах приложения Spring Boot](https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html) .

**Автоконфигурация Spring Boot OAuth2 (с использованием Нового стека)**

Чтобы использовать новый стек, нам нужно добавить зависимости в зависимости от того, что мы хотим настроить - сервер авторизации, сервер ресурсов или клиентское приложение.

Давайте посмотрим на них по очереди.

**6.1. Поддержка сервера авторизации OAuth2**

Как мы видели в предыдущем разделе, стек Spring Security OAuth предлагает возможность настройки сервера Авторизации в качестве приложения Spring. Но проект устарел, и Spring на данный момент не поддерживает собственный сервер Авторизации. Вместо этого рекомендуется использовать существующих, хорошо зарекомендовавших себя поставщиков, такого как Okta для которого Spring Security содержит конфигурации по умолчанию.

Однако Spring Boot действительно упрощает настройку таких провайдеров.

**6.2. Поддержка сервера ресурсов OAuth2**

Чтобы включить поддержку сервера ресурсов, нам нужно добавить эту зависимость:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-oauth2-resource-server</**artifactId**>

</**dependency**>

Для получения информации о последней версии [посетите Maven Central](https://search.maven.org/search?q=a:spring-boot-starter-oauth2-resource-server) .

Кроме того, в нашей конфигурации безопасности нам необходимо включить DSL [*oauth2ResourceServer ()*](https://docs.spring.io/spring-security/site/docs/current/reference/html5/#webflux-oauth2resourceserver-jwt-sansboot) :

@Configuration

**public** **class** JWTSecurityConfig **extends** WebSecurityConfigurerAdapter {

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

...

.oauth2ResourceServer(oauth2 -> oauth2.jwt());

...

}

}

Наш [сервер ресурсов OAuth 2.0 с Spring Security 5](https://www.baeldung.com/spring-security-oauth-resource-server) дает подробный обзор этой темы.

**6.3. Поддержка клиентов OAuth2**

Подобно тому, как мы настроили сервер ресурсов, клиентскому приложению также нужны свои собственные зависимости и DSL.

Вот конкретная зависимость для поддержки клиента OAuth2:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-oauth2-client</**artifactId**>

</**dependency**>

Последнюю версию можно найти на [Maven Central](https://search.maven.org/search?q=a:spring-boot-starter-oauth2-client) .

Spring Security 5 также обеспечивает первоклассную поддержку входа в систему через DSL [*oath2Login ()*](https://www.baeldung.com/spring-security-5-oauth2-login) .

Для получения подробной информации о поддержке единого входа в новом стеке, пожалуйста, обратитесь к нашему [Простому единому входу с Spring Security OAuth2](https://www.baeldung.com/sso-spring-security-oauth2) .

Сервер ресурсов OAuth 2.0 с Spring Security 5

**1. Обзор**

В этом руководстве мы узнаем, **как настроить сервер ресурсов OAuth 2.0 с помощью Spring Security 5** .

Мы сделаем это с помощью JWT, а также непрозрачных токенов, двух типов токенов-носителей, поддерживаемых Spring Security.

Прежде чем перейти к реализации и примерам кода, мы установим некоторую предысторию.

**2. Немного предыстории**

**2.1. Что такое JWT и непрозрачные токены?**

JWT или [JSON Web Token](https://tools.ietf.org/html/rfc7519) - это способ безопасной передачи конфиденциальной информации в широко распространенном формате JSON. Содержащаяся информация может быть о пользователе или о самом токене, например о сроке действия и эмитенте.

С другой стороны, непрозрачный токен, как следует из названия, непрозрачен с точки зрения информации, которую он несет. Токен - это просто идентификатор, который указывает на информацию, хранящуюся на сервере авторизации, - он проверяется посредством интроспекции на стороне сервера.

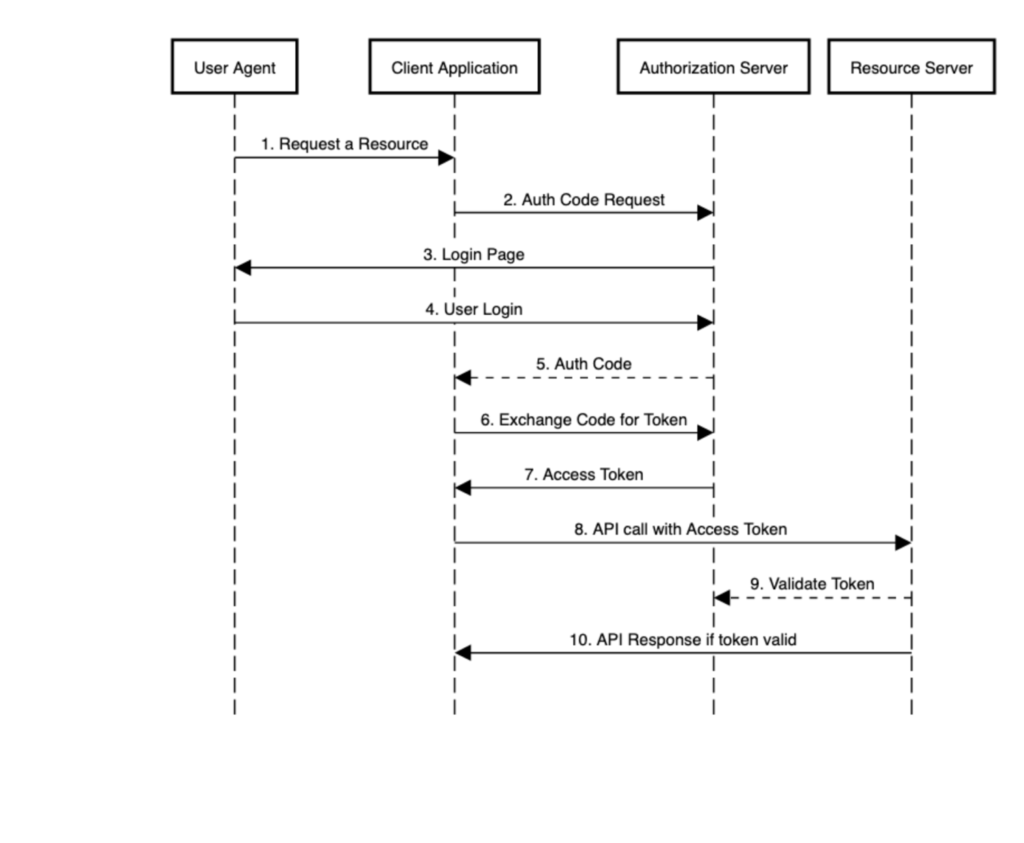
**2.2. Что такое сервер ресурсов?**

В контексте OAuth 2.0 **сервер ресурсов - это приложение, которое защищает ресурсы с помощью токенов OAuth** . Эти токены выдаются сервером авторизации, обычно клиентскому приложению. Задача сервера ресурсов - проверить токен перед тем, как предоставить ресурс клиенту.

Срок действия токена определяется несколькими вещами:

* Этот токен пришел с настроенного сервера авторизации?
* Это еще не истекло?
* Является ли этот ресурсный сервер его целевой аудиторией?
* Имеет ли токен необходимые права доступа для доступа к запрошенному ресурсу?

Для наглядности давайте посмотрим на диаграмму последовательности для [потока кода авторизации](https://tools.ietf.org/html/rfc6749#section-1.3.1) и увидим всех действующих лиц в действии:

[](https://www.baeldung.com/wp-content/uploads/2020/08/AuthCodeFlowSequenceDiagram-1.png)Как мы видим на шаге 8, когда клиентское приложение вызывает API сервера ресурсов для доступа к защищенному ресурсу, оно сначала обращается к серверу авторизации для проверки токена, содержащегося в заголовке запроса *Authorization: Bearer* , а затем отвечает клиенту.

**Шаг 9 - это то, на чем мы сосредоточены в этом руководстве.**

Хорошо, теперь перейдем к части кода. Мы настроим сервер авторизации с помощью Keycloak, сервер ресурсов, проверяющий токены JWT, другой сервер ресурсов, проверяющий непрозрачные токены, и пару тестов JUnit для имитации клиентских приложений и проверки ответов.

**3. Сервер авторизации**

Сначала мы настроим сервер авторизации или то, что выдает токены.

**Для этого мы будем использовать [Keycloak, встроенный в приложение Spring Boot](https://www.baeldung.com/keycloak-embedded-in-spring-boot-app)** . Keycloak - это решение с открытым исходным кодом для управления идентификацией и доступом. Поскольку в этом руководстве мы сосредоточены на сервере ресурсов, мы не будем углубляться в него.

В нашем встроенном сервере Keycloak определены два клиента - *fooClient* и *barClient -* соответствующие двум нашим приложениям сервера ресурсов.

**4. Сервер ресурсов - Использование JWT**

Наш ресурсный сервер будет состоять из четырех основных компонентов:

* *Модель* - ресурс, который нужно защищать
* *API* - REST-контроллер для предоставления ресурса
* *Конфигурация безопасности* - класс для определения контроля доступа к защищенному ресурсу, который предоставляет API.
* *application.yml* - файл конфигурации для объявления свойств, включая информацию о сервере авторизации

Давайте посмотрим на них один за другим для нашего сервера ресурсов, обрабатывающего токены JWT, после того, как взглянем на зависимости.

**4.1. Зависимости Maven**

В основном нам понадобится [*spring-boot-starter-oauth2-resource-server*](https://search.maven.org/search?q=spring-boot-starter-oauth2-resource-server) , стартер Spring Boot для поддержки сервера ресурсов. Этот стартер по умолчанию включает Spring Security, поэтому нам не нужно добавлять его явно:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-web</**artifactId**>

<**version**>2.2.6.RELEASE</**version**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-oauth2-resource-server</**artifactId**>

<**version**>2.2.6.RELEASE</**version**>

</**dependency**>

<**dependency**>

<**groupId**>org.apache.commons</**groupId**>

<**artifactId**>commons-lang3</**artifactId**>

<**version**>3.9</**version**>

</**dependency**>

Помимо этого, мы также добавили веб-поддержку.

В наших демонстрационных целях мы будем генерировать ресурсы случайным образом, а не получать их из базы данных, с некоторой помощью библиотеки Apache [*commons-lang3*](https://search.maven.org/artifact/org.apache.commons/commons-lang3) .

**4.2. Модель**

Для простоты мы будем использовать *Foo* , POJO, в качестве нашего защищенного ресурса:

**public** **class** Foo {

**private** **long** id;

**private** String name;

// constructor, getters and setters

}

**4.3. API**

Вот наш контроллер отдыха, чтобы сделать *Foo* доступным для манипуляций:

@RestController

@RequestMapping(value = "/foos")

**public** **class** FooController {

@GetMapping(value = "/{id}")

**public** Foo findOne(@PathVariable Long id) {

**return** **new** Foo(Long.parseLong(randomNumeric(2)), randomAlphabetic(4));

}

@GetMapping

**public** List findAll() {

List fooList = **new** ArrayList();

fooList.add(**new** Foo(Long.parseLong(randomNumeric(2)), randomAlphabetic(4)));

fooList.add(**new** Foo(Long.parseLong(randomNumeric(2)), randomAlphabetic(4)));

fooList.add(**new** Foo(Long.parseLong(randomNumeric(2)), randomAlphabetic(4)));

**return** fooList;

}

@ResponseStatus(HttpStatus.CREATED)

@PostMapping

**public** **void** create(@RequestBody Foo newFoo) {

logger.info(**"Foo created"**);

}

}

Как очевидно, у нас есть возможность ПОЛУЧИТЬ все *Foo* , ПОЛУЧИТЬ *Foo* по идентификатору и POST a *Foo* .

**4.4. Конфигурация безопасности**

В этом классе конфигурации мы определяем уровни доступа для нашего ресурса:

@Configuration

**public** **class** JWTSecurityConfig **extends** WebSecurityConfigurerAdapter {

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests(authz -> authz

.antMatchers(HttpMethod.GET, **"/foos/\*\*"**).hasAuthority(**"SCOPE\_read"**)

.antMatchers(HttpMethod.POST, **"/foos"**).hasAuthority(**"SCOPE\_write"**)

.anyRequest().authenticated())

.oauth2ResourceServer(oauth2 -> oauth2.jwt());

}

}

Любой, у кого есть токен доступа с областью *чтения,* может получить *Foo* s. Чтобы отправить новый *Foo* на POST , их токен должен иметь область *записи* .

Кроме того, **мы добавили вызов *jwt ()* с использованием**[**DSL**](https://docs.spring.io/spring-integration/docs/5.1.0.M1/reference/html/java-dsl.html)***oauth2ResourceServer (),* чтобы указать здесь тип токенов, поддерживаемых нашим сервером** .

**4.5. *application.yml***

В свойствах приложения, помимо обычного номера порта и пути контекста, **нам нужно определить путь к URI эмитента нашего сервера авторизации, чтобы сервер ресурсов мог обнаружить**[**конфигурацию**](https://openid.net/specs/openid-connect-discovery-1_0.html#ProviderConfig)**своего**[**поставщика**](https://openid.net/specs/openid-connect-discovery-1_0.html#ProviderConfig) :

server:

port: 8081

servlet:

context-path: /resource-server-jwt

spring:

security:

oauth2:

resourceserver:

jwt:

issuer-uri: http://localhost:8083/auth/realms/baeldung

Сервер ресурсов использует эту информацию для проверки токенов JWT, поступающих из клиентского приложения, в соответствии с шагом 9 нашей диаграммы последовательности.

Чтобы эта проверка работала с использованием свойства *эмитента-uri* , сервер авторизации должен быть запущен и работать. В противном случае сервер ресурсов не запустится.

Если нам нужно запустить его независимо, тогда мы можем предоставить *свойство jwk-set-uri* вместо того, чтобы указывать на конечную точку сервера авторизации, предоставляющую открытые ключи:

jwk-set-uri: http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/certs

И это все, что нам нужно, чтобы наш сервер проверял токены JWT.

**4.6. Тестирование**

Для тестирования мы настроим JUnit. Чтобы выполнить этот тест, нам нужно, чтобы сервер авторизации, а также сервер ресурсов были запущены и работали.

Давайте проверим, что в нашем тесте мы можем получить *Foo* s из *resource-server-jw* t с токеном с ограниченной областью *чтения* :

@Test

**public** **void** givenUserWithReadScope\_whenGetFooResource\_thenSuccess() {

String accessToken = obtainAccessToken(**"read"**);

Response response = RestAssured.given()

.header(HttpHeaders.AUTHORIZATION, **"Bearer "** + accessToken)

.get(**"http://localhost:8081/resource-server-jwt/foos"**);

assertThat(response.as(List.class)).hasSizeGreaterThan(0);

}

В приведенном выше коде в строке № 3 мы получаем токен доступа с областью *чтения* с сервера авторизации, охватывающий шаги с 1 по 7 нашей диаграммы последовательности.

Шаг 8 выполняется *RestAssured* «s *Get ()* вызова. Шаг 9 выполняется сервером ресурсов с конфигурациями, которые мы видели, и прозрачен для нас как пользователей.

**5. Сервер ресурсов - Использование непрозрачных токенов**

Затем давайте посмотрим, как те же компоненты для нашего сервера ресурсов обрабатывают непрозрачные токены.

**5.1. Зависимости Maven**

Для поддержки непрозрачных токенов нам дополнительно понадобится [*зависимость oauth2-oidc-sdk*](https://search.maven.org/search?q=oauth2-oidc-sdk):

<**dependency**>

<**groupId**>com.nimbusds</**groupId**>

<**artifactId**>oauth2-oidc-sdk</**artifactId**>

<**version**>8.19</**version**>

<**scope**>runtime</**scope**>

</**dependency**>

**5.2. Модель и контроллер**

Для этого мы добавим ресурс *Bar* :

**public** **class** Bar {

**private** **long** id;

**private** String name;

// constructor, getters and setters

}

У нас также будет *BarController* с конечными точками, подобными нашему *FooController* ранее, чтобы выдавать *Bar* .

**5.3. *application.yml***

Здесь в *application.yml* нам нужно добавить *introspection-uri,* соответствующий конечной точке интроспекции нашего сервера авторизации. Как упоминалось ранее, непрозрачный токен проверяется следующим образом:

server:

port: 8082

servlet:

context-path: /resource-server-opaque

spring:

security:

oauth2:

resourceserver:

opaque:

introspection-uri: http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/token/introspect

introspection-client-id: barClient

introspection-client-secret: barClientSecret

**5.4. Конфигурация безопасности**

Сохраняя уровни доступа, аналогичные уровню доступа *Foo* для ресурса *Bar* , **этот класс конфигурации также вызывает *opaqueToken (),* используя DSL *oauth2ResourceServer (),* чтобы указать использование непрозрачного типа токена** :

@Configuration

**public** **class** OpaqueSecurityConfig **extends** WebSecurityConfigurerAdapter {

@Value("${spring.security.oauth2.resourceserver.opaque.introspection-uri}")

String introspectionUri;

@Value("${spring.security.oauth2.resourceserver.opaque.introspection-client-id}")

String clientId;

@Value("${spring.security.oauth2.resourceserver.opaque.introspection-client-secret}")

String clientSecret;

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests(authz -> authz

.antMatchers(HttpMethod.GET, **"/bars/\*\*"**).hasAuthority(**"SCOPE\_read"**)

.antMatchers(HttpMethod.POST, **"/bars"**).hasAuthority(**"SCOPE\_write"**)

.anyRequest().authenticated())

.oauth2ResourceServer(oauth2 -> oauth2

.opaqueToken(token -> token.introspectionUri(**this**.introspectionUri)

.introspectionClientCredentials(**this**.clientId, **this**.clientSecret)));

}

}

Здесь мы также указываем учетные данные клиента, соответствующие клиенту сервера авторизации, который мы будем использовать. Мы определили их ранее в нашем *application.yml* .

**5.5. Тестирование**

Мы настроим JUnit для нашего непрозрачного сервера ресурсов на основе токенов, аналогично тому, как мы сделали это для JWT.

В этом случае давайте проверим, может ли токен доступа с ограниченным доступом для *записи* выполнить POST-запрос *Bar* до состояния *непрозрачного для сервера-ресурса* :

@Test

**public** **void** givenUserWithWriteScope\_whenPostNewBarResource\_thenCreated() {

String accessToken = obtainAccessToken(**"read write"**);

Bar newBar = **new** Bar(Long.parseLong(randomNumeric(2)), randomAlphabetic(4));

Response response = RestAssured.given()

.contentType(ContentType.JSON)

.header(HttpHeaders.AUTHORIZATION, **"Bearer "** + accessToken)

.body(newBar)

.log()

.all()

.post(**"http://localhost:8082/resource-server-opaque/bars"**);

assertThat(response.getStatusCode()).isEqualTo(HttpStatus.CREATED.value());

}

Если мы получим статус СОЗДАЛИСЬ назад, это означает , что сервер ресурса успешно проверен непрозрачный маркер и создал *панель* для нас.

Простой единый вход с Spring Security OAuth2

**1. Обзор**

В этом руководстве мы обсудим, как реализовать **SSO - Single Sign On (**технология, при использовании которой [пользователь](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C) переходит из одного раздела портала в другой, либо из одной системы в другую, не связанную с первой системой, без повторной [аутентификации](https://ru.wikipedia.org/wiki/%D0%90%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F). Например, если на веб-портале существует несколько обширных независимых разделов ([форум](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%84%D0%BE%D1%80%D1%83%D0%BC), [чат](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0)), [блог](https://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%B3) и т. д.) то, пройдя процедуру аутентификации в одном из сервисов, пользователь автоматически получает доступ ко всем остальным, что избавляет его от многократного ввода данных своей учётной записи.**) - с использованием Spring Security OAuth и Spring Boot, используя Keycloak** в качестве сервера авторизации.

Мы будем использовать 4 отдельных приложения:

* Сервер авторизации - это центральный механизм аутентификации.
* Сервер ресурсов - провайдер *Foo* s
* Два клиентских приложения - приложения, использующие SSO

Проще говоря, когда пользователь пытается получить доступ к ресурсу через одно клиентское приложение, он сначала будет перенаправлен на аутентификацию через сервер авторизации. Keycloak выполнит вход пользователя в систему, и, пока он все еще находится в первом приложении, если доступ ко второму клиентскому приложению осуществляется с помощью того же браузера, пользователю не нужно будет снова вводить свои учетные данные.

Мы собираемся использовать тип предоставления *кода авторизации* из OAuth2, чтобы управлять делегированием аутентификации.

**Мы будем использовать стек OAuth в Spring Security 5.** Если вы хотите использовать устаревший стек Spring Security OAuth, [прочтите](https://www.baeldung.com/sso-spring-security-oauth2-legacy) эту предыдущую статью:  [Простой единый вход с Spring Security OAuth2 (устаревший стек)](https://www.baeldung.com/sso-spring-security-oauth2-legacy)

Согласно [руководству](https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Migration-Guide#changes-in-approach-1) по [миграции](https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Migration-Guide#changes-in-approach-1) :

Spring Security называет эту функцию OAuth 2.0 Login, а Spring Security OAuth называет ее SSO.

**2. Сервер авторизации**

Ранее стек Spring Security OAuth предлагал возможность настройки сервера авторизации как приложения Spring.

Однако стек OAuth устарел, и теперь мы будем использовать Keycloak в качестве нашего сервера авторизации.

**Итак, на этот раз мы**[**настроим**](https://www.baeldung.com/keycloak-embedded-in-a-spring-boot-application/)**наш сервер авторизации как**[**встроенный сервер Keycloak в приложении Spring Boot**](https://www.baeldung.com/keycloak-embedded-in-a-spring-boot-application/) .

В нашей [предварительной конфигурации](https://www.baeldung.com/keycloak-embedded-in-spring-boot-app#keycloak-config) , **мы определим два клиента, *ssoClient-1* и *ssoClient-2****,* по одному для каждого клиентского приложения.

**3. Сервер ресурсов**

Затем нам понадобится сервер ресурсов или REST API, который предоставит нам *Foo,* которое будет использовать наше клиентское приложение.

По сути, это то [же самое, что мы использовали ранее для наших клиентских приложений Angular](https://www.baeldung.com/rest-api-spring-oauth2-angular#resource-server) .

**4. Клиентские приложения**

Теперь давайте посмотрим на наше клиентское приложение Thymeleaf; мы, конечно, будем использовать Spring Boot, чтобы минимизировать конфигурацию.

Имейте в виду, что **нам понадобятся два из них, чтобы продемонстрировать функциональность единого входа** .

**4.1. Зависимости Maven**

Во-первых, нам потребуются следующие зависимости в нашем *pom.xml* :

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-web</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-oauth2-client</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-thymeleaf</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.thymeleaf.extras</**groupId**>

<**artifactId**>thymeleaf-extras-springsecurity5</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework</**groupId**>

<**artifactId**>spring-webflux</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>io.projectreactor.netty</**groupId**>

<**artifactId**>reactor-netty</**artifactId**>

</**dependency**>

**Чтобы включить всю необходимую поддержку клиентов, включая безопасность, нам просто нужно добавить *spring-boot-starter-oauth2-client*** . Кроме того, поскольку старый *RestTemplate* будет устаревшим, мы собираемся использовать *WebClient* , и поэтому мы добавили *spring-webflux* и *Reaction -netty* .

**4.2. Конфигурация безопасности**

Далее, самая важная часть, конфигурация безопасности нашего первого клиентского приложения:

@EnableWebSecurity

**public** **class** UiSecurityConfig **extends** WebSecurityConfigurerAdapter {

@Override

**public** **void** configure(HttpSecurity http) **throws** Exception {

http.antMatcher(**"/\*\*"**)

.authorizeRequests()

.antMatchers(**"/"**)

.permitAll()

.anyRequest()

.authenticated()

.and()

.**oauth2Login()**;

}

@Bean

WebClient webClient(ClientRegistrationRepository clientRegistrationRepository,

OAuth2AuthorizedClientRepository authorizedClientRepository) {

ServletOAuth2AuthorizedClientExchangeFilterFunction oauth2 =

**new** ServletOAuth2AuthorizedClientExchangeFilterFunction(clientRegistrationRepository,

authorizedClientRepository);

oauth2.setDefaultOAuth2AuthorizedClient(**true**);

**return** WebClient.builder().apply(oauth2.oauth2Configuration()).build();

}

}

**Ядром этой конфигурации является метод *oauth2Login ()* , который используется для включения поддержки входа в систему OAuth 2.0 в Spring Security.**Поскольку мы используем Keycloak, который по умолчанию является решением единой регистрации для веб-приложений и веб-служб RESTful, нам не нужно добавлять дополнительную конфигурацию для SSO.

Наконец, мы также определили bean-компонент *WebClient, который* будет действовать как простой HTTP-клиент для обработки запросов, отправляемых на наш сервер ресурсов.

А вот и *application.yml* :

spring:

security:

oauth2:

client:

registration:

custom:

client-id: ssoClient-1

client-secret: ssoClientSecret-1

scope: read,write

authorization-grant-type: authorization\_code

redirect-uri: http://localhost:8082/ui-one/login/oauth2/code/custom

provider:

custom:

authorization-uri: http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/auth

token-uri: http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/token

user-info-uri: http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/userinfo

user-name-attribute: preferred\_username

thymeleaf:

cache: false

server:

port: 8082

servlet:

context-path: /ui-one

resourceserver:

api:

project:

url: http://localhost:8081/sso-resource-server/api/foos/

Здесь *spring.security.oauth2.client.registration* - это корневое пространство имен для регистрации клиента. Мы определили клиента с *пользовательским* идентификатором регистрации . Затем мы определили его *client-id* , *client-secret* , *scope* , *authorization-grant-type* и *redirect-uri* , которые, конечно, должны быть такими же, как те, что определены для нашего сервера авторизации.

После этого мы определили нашего поставщика услуг или сервер авторизации, снова с тем же идентификатором *custom* , и перечислили его различные URI для использования Spring Security. Это все, что нам нужно определить, и **фреймворк без проблем выполняет весь процесс входа в систему, включая перенаправление на Keycloak** .

Также обратите внимание, что в нашем примере здесь мы развернули наш сервер авторизации, но, конечно, мы также можем использовать других сторонних поставщиков, таких как [Facebook или GitHub](https://www.baeldung.com/spring-security-5-oauth2-login) .

**4.3. Контроллер**

Давайте теперь реализуем наш контроллер в клиентском приложении, чтобы запрашивать *Foo* с нашего сервера ресурсов:

@Controller

**public** **class** FooClientController {

@Value("${resourceserver.api.url}")

**private** String fooApiUrl;

@Autowired

**private** WebClient webClient;

@GetMapping("/foos")

**public** String getFoos(Model model) {

List<FooModel> foos = **this**

.webClient

.get()

.uri(fooApiUrl)

.retrieve()

.bodyToMono(**new**

ParameterizedTypeReference<List<FooModel>>() {})

.block();

model.addAttribute(**"foos"**, foos);

**return** **"foos"**;

}

}

Как мы видим, у нас есть только один метод, который распределяет ресурсы в шаблон *foos* . **Нам не нужно добавлять код для входа в систему.**

**4.4. Внешний интерфейс**

Теперь давайте посмотрим на конфигурацию внешнего интерфейса нашего клиентского приложения. Мы не собираемся здесь заострять внимание на этом, в основном потому, что мы [уже рассказали об этом на сайте](https://www.baeldung.com/spring-thymeleaf-3) .

У нашего клиентского приложения очень простой интерфейс; вот ***index.html*** :

<**a** class=**"navbar-brand"** th:href=**"@{/foos/}"**>Spring OAuth Client Thymeleaf - 1</**a**>

<**label**>Welcome !</**label**> <**br** /> <**a** th:href=**"@{/foos/}"**>Login</**a**>

И файл***foos.html*** :

<**a** class=**"navbar-brand"** th:href=**"@{/foos/}"**>Spring OAuth Client Thymeleaf -1</**a**>

Hi, <**span** sec:authentication=**"name"**>preferred\_username</**span**>

<**h1**>All Foos:</**h1**>

<**table**>

<**thead**>

<**tr**>

<**td**>ID</**td**>

<**td**>Name</**td**>

</**tr**>

</**thead**>

<**tbody**>

<**tr** th:if=**"${foos.empty}"**>

<**td** colspan=**"4"**>No foos</**td**>

</**tr**>

<**tr** th:each=**"foo : ${foos}"**>

<**td**><**span** th:text=**"${foo.id}"**> ID </**span**></**td**>

<**td**><**span** th:text=**"${foo.name}"**> Name </**span**></**td**>

</**tr**>

</**tbody**>

</**table**>

Страница ***foos.html*** требует, чтобы пользователи *прошли* аутентификацию. **Если неаутентифицированный пользователь попытается получить доступ к  *foos.html*, он *сначала* будет перенаправлен на страницу входа в Keycloak** .

**4.5. Второе клиентское приложение**

Мы настроим второе приложение, *Spring OAuth Client Thymeleaf -2,* используя другой *client\_id ssoClient-2* .

В основном это будет то же самое, что и первое приложение, которое мы только что описали.

***Application.yml* будет отличаться включать различные *client\_id* , *client\_secret* и *redirect\_uri* в его *spring.security.oauth2.client.registration:***

spring:

security:

oauth2:

client:

registration:

custom:

client-id: ssoClient-2

client-secret: ssoClientSecret-2

scope: read,write

authorization-grant-type: authorization\_code

redirect-uri: http://localhost:8084/ui-two/login/oauth2/code/custom

И, конечно же, нам также нужен другой порт сервера, чтобы мы могли запускать их параллельно:

server:

port: 8084

servlet:

context-path: /ui-two

Наконец, мы настроим интерфейсные HTML-файлы так, чтобы они имели заголовок *Spring OAuth Client Thymeleaf - 2* вместо *- 1,* чтобы мы могли различать их.

**5. Проверка работы системы единого входа**

Чтобы проверить поведение SSO, запустим наши приложения.

Для этого нам понадобятся все наши 4 загрузочных приложения - сервер авторизации, сервер ресурсов и оба клиентских приложения.

Теперь давайте откроем браузер, скажем Chrome, и *войдем* в [*Client-1,*](http://localhost:8082/ui-one) используя учетные данные *john@test.com / 123* . Затем в другом окне или на вкладке нажмите URL-адрес для [*Client-2*](http://localhost:8084/ui-two) . При нажатии кнопки входа в систему мы сразу же будем перенаправлены на страницу *Foos* , минуя этап аутентификации.

Точно так же, если пользователь сначала входит в *Client-2* , ему не нужно вводить свое имя пользователя / пароль для *Client-1* .

**6. Заключение**

В этом руководстве мы сосредоточились на реализации единого входа с использованием Spring Security OAuth2 и Spring Boot с использованием Keycloak в качестве поставщика удостоверений.

Аутентификация пользователей с помощью Facebook, Google или других учетных данных с помощью OAuth2 в Spring Security 5

Spring Security 5 представляет новый класс OAuth2LoginConfigurer, который мы можем использовать для настройки внешнего сервера авторизации.

В этой статье **мы рассмотрим некоторые из различных вариантов конфигурации, доступных для элемента oauth2Login ()** .

**2. Зависимости Maven**

**В проекте Spring Boot все, что нам нужно, это добавить стартовый**[***spring-boot-starter-oauth2-client***](https://search.maven.org/search?q=a:spring-boot-starter-oauth2-client)**:**

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-oauth2-client</**artifactId**>

<**version**>2.3.3.RELEASE</**version**>

</**dependency**>

В проекте, [*отличном от*](https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-security-oauth2-client%22%20AND%20g%3A%22org.springframework.security%22) Boot, помимо стандартных зависимостей Spring и Spring Security, нам также необходимо явно добавить зависимости [*spring-security-oauth2-client*](https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-security-oauth2-client%22%20AND%20g%3A%22org.springframework.security%22) и [*spring-security-oauth2-jose*](https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22spring-security-oauth2-jose%22) :

<**dependency**>

<**groupId**>org.springframework.security</**groupId**>

<**artifactId**>spring-security-oauth2-client</**artifactId**>

<**version**>5.3.4.RELEASE</**version**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.security</**groupId**>

<**artifactId**>spring-security-oauth2-jose</**artifactId**>

<**version**>5.3.4.RELEASE</**version**>

</**dependency**>

**3. Настройка клиентов**

В проекте Spring Boot все, что нам нужно сделать, это добавить несколько стандартных свойств для каждого клиента, который мы хотим настроить.

**Давайте настроим наш проект для входа в систему с клиентами, зарегистрированными в Google и Facebook в качестве поставщиков аутентификации.**

**3.1. Получение учетных данных клиента**

Чтобы получить учетные данные клиента для аутентификации Google OAuth2, перейдите в [консоль Google API](https://console.developers.google.com/) - раздел «Учетные данные».

Здесь мы создадим учетные данные типа «OAuth2 Client ID» для нашего веб-приложения. Это приводит к тому, что Google устанавливает для нас идентификатор клиента и секрет.

Нам также необходимо настроить авторизованный URI перенаправления в консоли Google, то есть путь, по которому пользователи будут перенаправлены после успешного входа в систему с помощью Google.

По умолчанию Spring Boot настраивает этот URI перенаправления как */ login / oauth2 / code / {registrationId}.*Поэтому для Google мы добавим URI:

http://localhost:8081/login/oauth2/code/google

Чтобы получить учетные данные клиента для аутентификации в Facebook, нам необходимо зарегистрировать приложение на веб-сайте [Facebook для разработчиков](https://developers.facebook.com/docs/facebook-login" \t "_blank) и настроить соответствующий URI как «Действительный URI перенаправления OAuth»:

http://localhost:8081/login/oauth2/code/facebook

**3.3. Конфигурация безопасности**

Затем нам нужно добавить учетные данные клиента в файл *application.properties* . **Свойства Spring Security имеют префикс *«spring.security.oauth2.client.registration»,* за которым следует имя клиента, затем имя свойства клиента:**

spring.security.oauth2.client.registration.google.client-id=<your client id>

spring.security.oauth2.client.registration.google.client-secret=<your client secret>

spring.security.oauth2.client.registration.facebook.client-id=<your client id>

spring.security.oauth2.client.registration.facebook.client-secret=<your client secret>

**Добавление этих свойств по крайней мере для одного клиента *включит* класс *Oauth2ClientAutoConfiguration,*** который устанавливает все необходимые компоненты.

**Автоматическая конфигурация веб-безопасности эквивалентна определению простого *элемента oauth2Login ()* :**

@Configuration

**public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.authorizeRequests()

.anyRequest().authenticated()

.and()

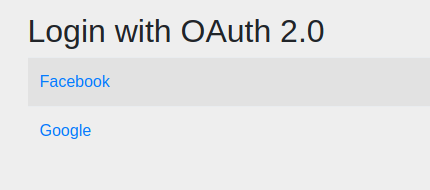
.oauth2Login();

}

}

Здесь мы видим, что *элемент oauth2Login ()* используется аналогично уже известным *элементам httpBasic ()* и *formLogin ()* .

**Теперь, когда мы пытаемся получить доступ к защищенному URL-адресу, приложение отображает автоматически созданную страницу входа с двумя клиентами:**

[](https://www.baeldung.com/wp-content/uploads/2018/01/oauth-login-default.png)

**3.4. Другие клиенты**

**Обратите внимание, что помимо Google и Facebook проект Spring Security также содержит конфигурации по умолчанию для GitHub и Okta.**Эти конфигурации по умолчанию предоставляют всю необходимую информацию для аутентификации, что позволяет нам вводить только учетные данные клиента.

Если мы хотим использовать другого поставщика аутентификации, не настроенного в Spring Security, нам нужно определить полную конфигурацию с такой информацией, как URI авторизации и URI токена. [Вот](https://github.com/spring-projects/spring-security/blob/master/config/src/main/java/org/springframework/security/config/oauth2/client/CommonOAuth2Provider.java) посмотрите на конфигурации по умолчанию в Spring Security, чтобы иметь представление о необходимых свойствах.

**4. Настройка в не загрузочном проекте**

**4.1. Создание *ClientRegistrationRepository* Bean**

**Если мы не работаем с приложением Spring Boot, нам нужно будет определить bean-компонент *ClientRegistrationRepository,*** который содержит внутреннее представление информации о клиенте, принадлежащей серверу авторизации:

@Configuration

@EnableWebSecurity

@PropertySource("classpath:application.properties")

**public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {

**private** **static** List<String> clients = Arrays.asList(**"google"**, **"facebook"**);

@Bean

**public** ClientRegistrationRepository clientRegistrationRepository() {

List<ClientRegistration> registrations = clients.stream()

.map(c -> getRegistration(c))

.filter(registration -> registration != **null**)

.collect(Collectors.toList());

**return** **new** InMemoryClientRegistrationRepository(registrations);

}

}

Здесь мы создаем *InMemoryClientRegistrationRepository* со списком *ClientRegistration* объектов.

**4.2. Создание объектов регистрации *клиентов***

Давайте посмотрим на *метод getRegistration (),* который строит эти объекты:

**private** **static** String CLIENT\_PROPERTY\_KEY = **"spring.security.oauth2.client.registration."**;

@Autowired

**private** Environment env;

**private** ClientRegistration getRegistration(String client) {

String clientId = env.getProperty(CLIENT\_PROPERTY\_KEY + client + **".client-id"**);

**if** (clientId == **null**) {

**return** **null**;

}

String clientSecret = env.getProperty(CLIENT\_PROPERTY\_KEY + client + **".client-secret"**);

**if** (client.equals(**"google"**)) {

**return** CommonOAuth2Provider.GOOGLE.getBuilder(client)

.clientId(clientId).clientSecret(clientSecret).build();

}

**if** (client.equals(**"facebook"**)) {

**return** CommonOAuth2Provider.FACEBOOK.getBuilder(client)

.clientId(clientId).clientSecret(clientSecret).build();

}

**return** **null**;

}

Здесь мы читаем учетные данные клиента из аналогичного файла *application.properties*, а затем используем перечисление *CommonOauth2Provider,* уже определенное в Spring Security для остальных свойств клиента для клиентов Google и Facebook.

Каждый экземпляр *ClientRegistration* соответствует клиенту.

**4.3. Регистрация *КлиентаРегистрацияРепозитория***

Наконец, мы должны создать bean *OAuth2AuthorizedClientService* на основе *ClientRegistrationRepository* фасоли и зарегистрировать как с *oauth2Login ()* элемент:

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.authorizeRequests().anyRequest().authenticated()

.and()

.oauth2Login()

.clientRegistrationRepository(clientRegistrationRepository())

.authorizedClientService(authorizedClientService());

}

@Bean

**public** OAuth2AuthorizedClientService authorizedClientService() {

**return** **new** InMemoryOAuth2AuthorizedClientService(

clientRegistrationRepository());

}

Как показано здесь, **мы можем использовать метод *clientRegistrationRepository ()* функции *oauth2Login ()* для регистрации репозитория пользовательской регистрации.**

Нам также нужно будет определить настраиваемую страницу входа в систему, поскольку она больше не будет создаваться автоматически. Мы увидим больше информации об этом в следующем разделе.

Давайте продолжим дальнейшую настройку нашего процесса входа в систему

**5. Настройка *oauth2Login ()***

Есть несколько элементов, которые использует процесс OAuth 2 и которые мы можем настроить с *помощью* методов *oauth2Login ()* .

**Обратите внимание, что все эти элементы имеют конфигурации по умолчанию в Spring Boot, и явная конфигурация не требуется.**

Но если мы хотим свою конфигурацию, то:

**5.1. Пользовательская страница входа вместо страницы “по умолчанию”**

Несмотря на то, что Spring Boot генерирует для нас страницу входа по умолчанию, мы обычно хотим определить нашу собственную настраиваемую страницу.

**Начнем с настройки нового URL-адреса входа в систему для элемента *oauth2Login ()* с помощью *метода loginPage ()* :**

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.authorizeRequests()

.antMatchers(**"/oauth\_login"**)

.permitAll()

.anyRequest()

.authenticated()

.and()

.oauth2Login()

.loginPage(**"/oauth\_login"**);

}

Здесь мы установили наш URL-адрес для входа в */ oauth\_login.*

Затем давайте определим *LoginController* с методом, который сопоставляется с этим URL:

@Controller

**public** **class** LoginController {

**private** **static** String authorizationRequestBaseUri

= **"oauth2/authorization"**;

Map<String, String> oauth2AuthenticationUrls

= **new** HashMap<>();

@Autowired

**private** ClientRegistrationRepository clientRegistrationRepository;

@GetMapping("/oauth\_login")

**public** String getLoginPage(Model model) {

// ...

**return** **"oauth\_login"**;

}

}

**Этот метод должен отправить карту доступных клиентов и их конечных точек авторизации в представление** , которое мы получим из bean-компонента *ClientRegistrationRepository* :

**public** String getLoginPage(Model model) {

Iterable<ClientRegistration> clientRegistrations = **null**;

ResolvableType type = ResolvableType.forInstance(clientRegistrationRepository)

.as(Iterable.class);

**if** (type != ResolvableType.NONE &&

ClientRegistration.class.isAssignableFrom(type.resolveGenerics()[0])) {

clientRegistrations = (Iterable<ClientRegistration>) clientRegistrationRepository;

}

clientRegistrations.forEach(registration ->

oauth2AuthenticationUrls.put(registration.getClientName(),

authorizationRequestBaseUri + **"/"** + registration.getRegistrationId()));

model.addAttribute(**"urls"**, oauth2AuthenticationUrls);

**return** **"oauth\_login"**;

}

Наконец, нам нужно определить нашу страницу *oauth\_login.html* :

<**h3**>Login with:</**h3**>

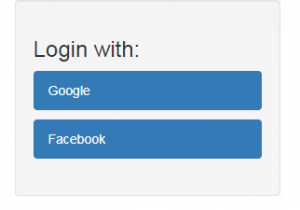
<**p** th:each=**"url : ${urls}"**>

<**a** th:text=**"${url.key}"** th:href=**"${url.value}"**>Client</**a**>

</**p**>

**Это простая HTML-страница, на которой отображаются ссылки для аутентификации каждого клиента.**

После добавления к нему стиля, мы можем изменить внешний вид страницы входа:



**5.2. Пользовательское поведение при успешной и неудачной аутентификации**

Мы можем контролировать поведение после аутентификации, используя разные методы:

* *defaultSuccessUrl ()* и *failureUrl ()* - для перенаправления пользователя на заданный URL
* *successHandler ()* и *failureHandler ()* - для выполнения пользовательской логики после процесса аутентификации

Давайте посмотрим, как мы можем настроить пользовательский URL-адрес для перенаправления пользователя на:

.oauth2Login()

.defaultSuccessUrl(**"/loginSuccess"**)

.failureUrl(**"/loginFailure"**);

Если пользователь посетил защищенную страницу до аутентификации, он будет перенаправлен на эту страницу после входа в систему; в противном случае они будут перенаправлены в */ loginSuccess.*

Если мы хотим, чтобы пользователь всегда отправлялся на URL-адрес */ loginSuccess* независимо от того, были ли они на защищенной странице раньше или нет, мы можем использовать метод *defaultSuccessUrl («/ loginSuccess», true)* .

Чтобы использовать настраиваемый обработчик, нам нужно будет создать класс, реализующий интерфейсы *AuthenticationSuccessHandler* или *AuthenticationFailureHandler* , переопределить унаследованные методы, а затем установить компоненты с помощью *методов successHandler ()* и failureHandler ().

**5.3. Пользовательская конечная точка авторизации**

Конечная точка авторизации - это конечная точка, которую Spring Security использует для запуска запроса авторизации на внешний сервер.

Во-первых, **давайте установим новые свойства для конечной точки авторизации** :

.oauth2Login()

.authorizationEndpoint()

.baseUri(**"/oauth2/authorize-client"**)

.authorizationRequestRepository(authorizationRequestRepository());

Здесь мы изменили *baseUri* на */ oauth2 / authorize-client* вместо значения по умолчанию */ oauth2 / authorization.*Мы также явно устанавливаем bean-компонент *authorizationRequestRepository (),* который мы должны определить:

@Bean

**public** AuthorizationRequestRepository<OAuth2AuthorizationRequest>

authorizationRequestRepository() {

**return** **new** HttpSessionOAuth2AuthorizationRequestRepository();

}

В нашем примере мы использовали реализацию нашего bean-компонента, предоставленную Spring, но мы также можем предоставить собственную реализацию.

**5.4. Пользовательская конечная точка токена**

*Конечная точка* токена обрабатывает токены доступа.

**Давайте явно настроим *tokenEndpoint ()***с реализацией клиента ответа по умолчанию:

.oauth2Login()

.tokenEndpoint()

.accessTokenResponseClient(accessTokenResponseClient());

А вот клиентский компонент ответа:

@Bean

**public** OAuth2AccessTokenResponseClient<OAuth2AuthorizationCodeGrantRequest>

accessTokenResponseClient() {

**return** **new** NimbusAuthorizationCodeTokenResponseClient();

}

Эта конфигурация идентична конфигурации по умолчанию и использует реализацию Spring, основанную на обмене кодом авторизации с поставщиком.

Конечно, мы могли бы также заменить клиента настраиваемого ответа.

**5.5. Пользовательская конечная точка перенаправления**

Это конечная точка для перенаправления после аутентификации с внешним поставщиком.

**Давайте посмотрим, как мы можем изменить *baseUri* для конечной точки перенаправления:**

.oauth2Login()

.redirectionEndpoint()

.baseUri(**"/oauth2/redirect"**)

URI по умолчанию - *login / oauth2 / code.*

Обратите внимание, что если мы его изменим, нам также придется обновить свойство *redirectUriTemplate* каждого *ClientRegistration* и добавить новый URI в качестве авторизованного URI перенаправления для каждого клиента.

**5.6. Пользовательская конечная точка информации о пользователе**

Конечная точка информации о пользователе - это место, которое мы можем использовать для получения информации о пользователе.

**Мы можем настроить эту конечную точку с помощью метода**  ***userInfoEndpoint()***. Для этого мы можем использовать такие методы, как *userService ()* и *customUserType (),* чтобы изменить способ получения информации о пользователе.

**6. Доступ к информации о пользователе**

Обычная задача, которую мы можем захотеть решить, - это поиск информации о вошедшем в систему пользователе. Для этого **мы можем сделать запрос к конечной точке информации о пользователе.**

Во-первых, нам нужно получить клиента, соответствующего токену текущего пользователя:

@Autowired

**private** OAuth2AuthorizedClientService authorizedClientService;

@GetMapping("/loginSuccess")

**public** String getLoginInfo(Model model, OAuth2AuthenticationToken authentication) {

OAuth2AuthorizedClient client = authorizedClientService

.loadAuthorizedClient(

authentication.getAuthorizedClientRegistrationId(),

authentication.getName());

//...

**return** **"loginSuccess"**;

}

Затем мы отправим запрос в конечную точку информации о пользователе клиента и *получим карту userAttributes:*

String userInfoEndpointUri = client.getClientRegistration()

.getProviderDetails().getUserInfoEndpoint().getUri();

**if** (!StringUtils.isEmpty(userInfoEndpointUri)) {

RestTemplate restTemplate = **new** RestTemplate();

HttpHeaders headers = **new** HttpHeaders();

headers.add(HttpHeaders.AUTHORIZATION, **"Bearer "** + client.getAccessToken()

.getTokenValue());

HttpEntity entity = **new** HttpEntity(**""**, headers);

ResponseEntity <map>response = restTemplate

.exchange(userInfoEndpointUri, HttpMethod.GET, entity, Map.class);

Map userAttributes = response.getBody();

model.addAttribute("name", userAttributes.get("name"));

}

Добавив свойство *name* в качестве атрибута *Model* , мы можем отобразить его в представлении *loginSuccess* как приветственное сообщение для пользователя:

https://www.baeldung.com/wp-content/uploads/2018/01/welcome-300x59.png

Помимо *названия, userAttributes Карта* также содержит свойства , такие как *электронная почта, FAMILY\_NAME, изображения, языка.*