

Лекция №9 Spring Security

Spring Security Framework. Request Security. Servlet filters. Security setting. Authentication and authorization. Interception of requests. Security support in Spring Security at the method level. Configure Spring Security for OAuth 2.0 Login and Resource Server

Введение

Spring Security это framework, предоставляющий механизмы построения систем аутентификации и авторизации, а также другие возможности обеспечения безопасности для корпоративных приложений, созданных с помощью Spring Framework.

<https://docs.spring.io/autorepo/docs/spring-security/5.3.0.RELEASE/api/>

Он также содержит проекты, специально предназначенные для работы с рядом механизмов аутентификации, таких как **LDAP**, **OAuth** и **SAML**. Spring Security предоставляет вам достаточно механизмов для борьбы с распространенными атаками безопасности, такими как фиксация сеансов (Session Fixation), перехват кликов (Clickjacking) и подделка межсайтовых запросов (Cross-Site RequestForgery). Имеет хорошую интеграцию с другими проектами - Spring MVC, Spring WebFlux, Spring Data, Spring Integration и Spring Boot.

Терминология

Principal.

Authentication

Credentials

Authorization.

Secured item/resource.

GrantedAuthority

SecurityContext

- **SecurityContextHolder.**
- **UserDetails.**
- **UserDetailsService,**

```
UserDetails      loadUserByUsername (String      username)      throws
UsernameNotFoundException;
```

Spring Security предоставляет несколько подходов:

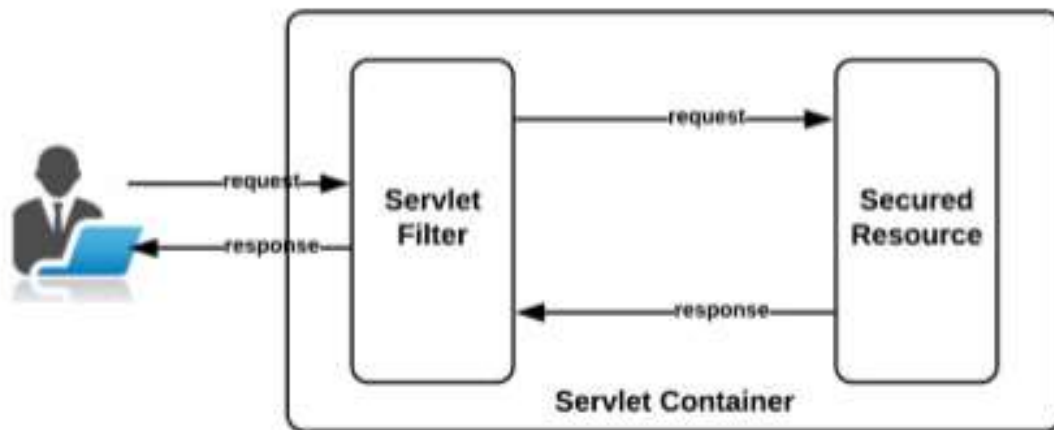
Web-URL: Основываясь на URL-адресе или шаблоне URL-адреса, вы можете контролировать доступ.

Method invocation: Даже для метода в JavaBean можно управлять доступом,

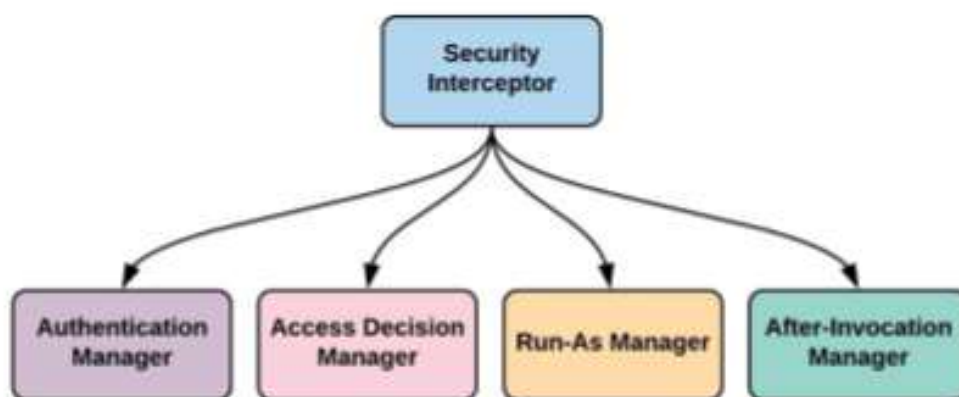
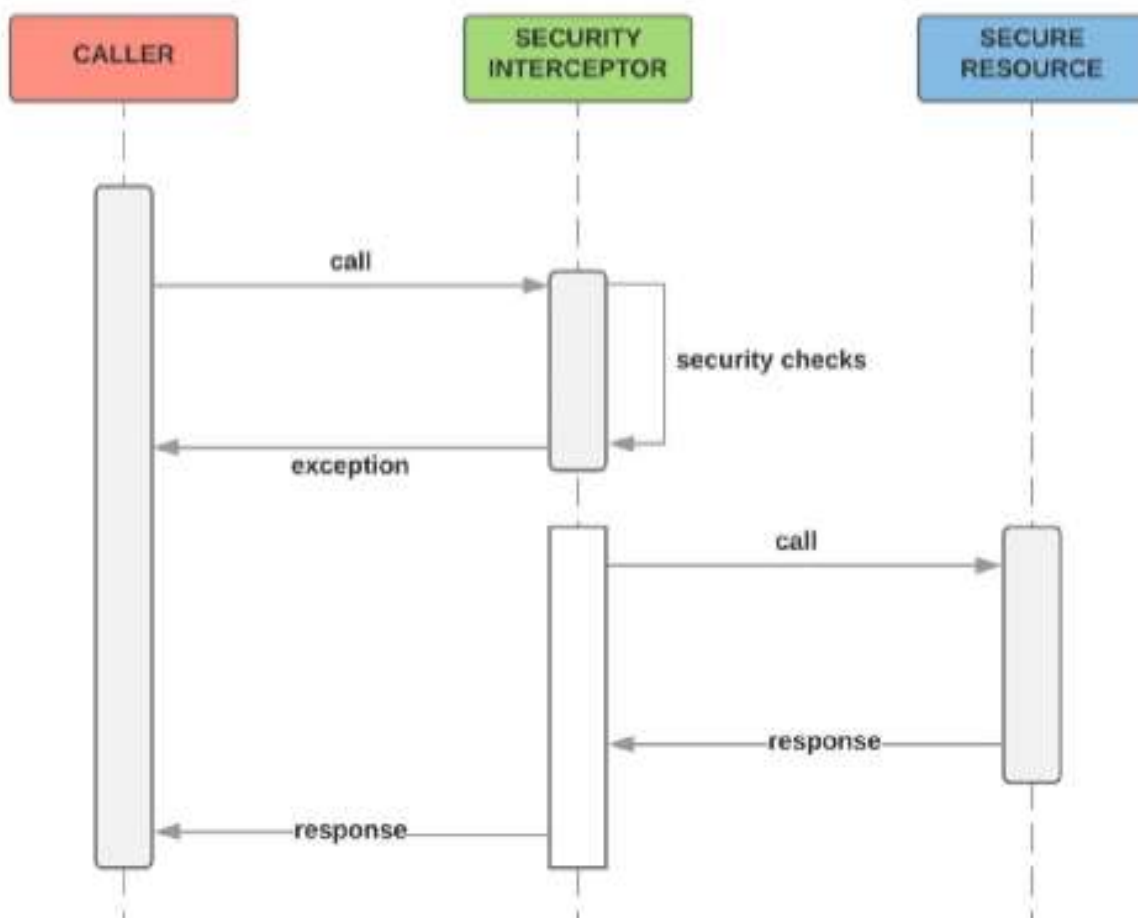
Domain instance: Одна из очень полезных функций - управление доступом к определенным данным с помощью контроля доступа к объектам домена.

Web service: позволяет защитить открытые веб-службы

Servlet Filter



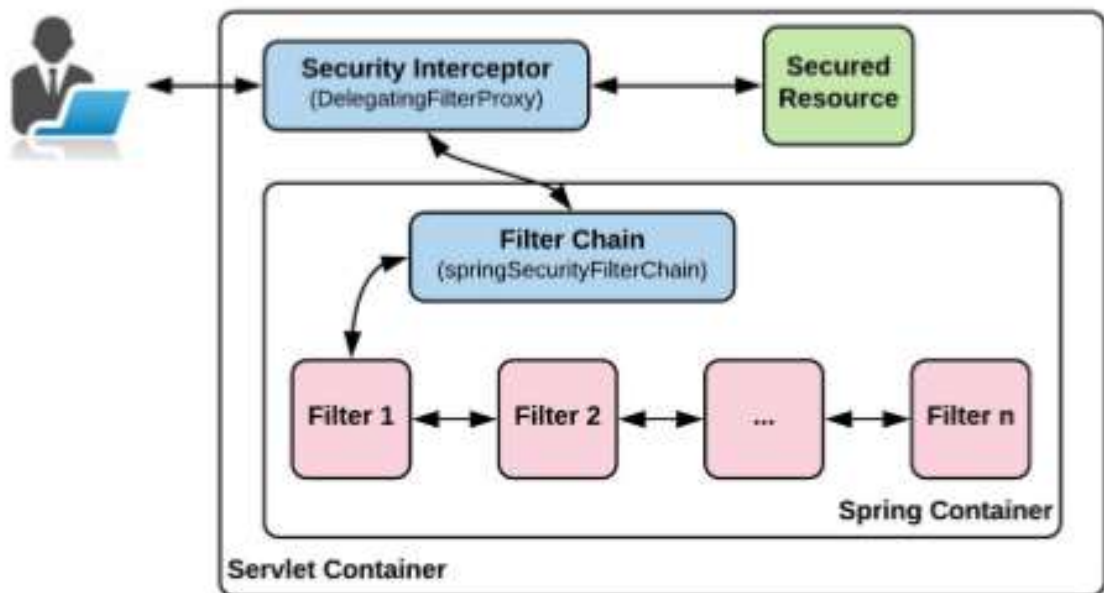
Security Interceptor (DelegatingFilterProxy)



Interceptor Security реализует **DelegatingFilterProxy**

В следующем примере кода показано, как настроить DelegatingProxyFilter в файле web.xml:

```
<web-app>
  <filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>
      springSecurityFilterChain
    </filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

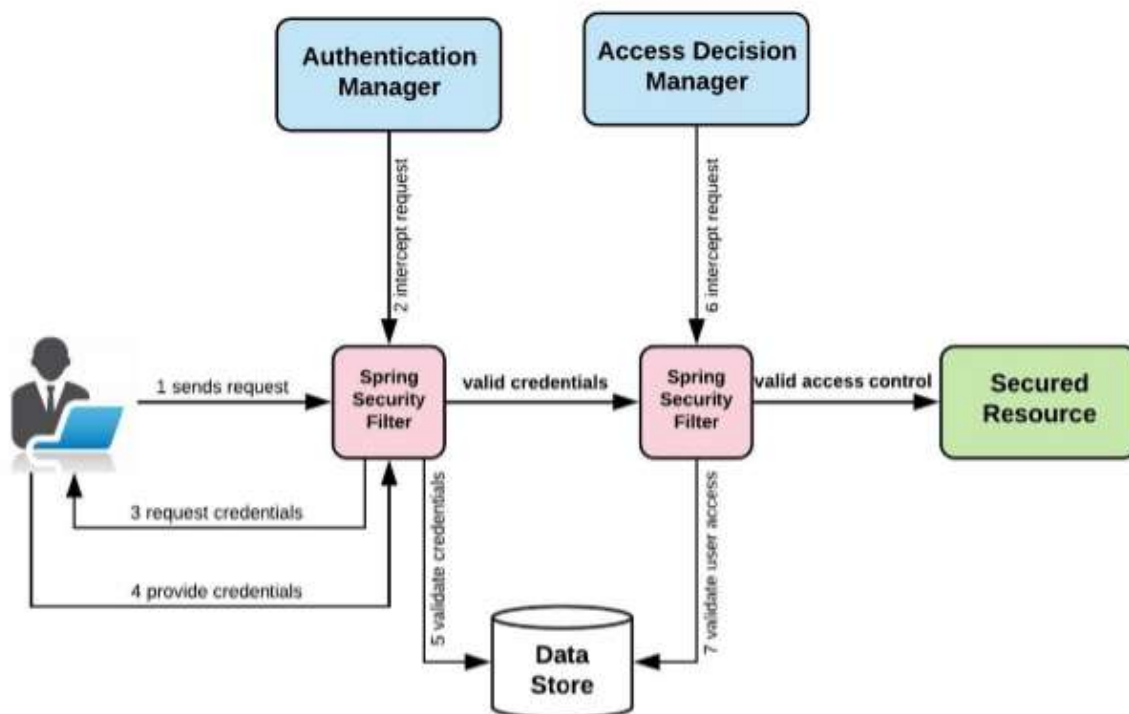


<https://docs.spring.io/spring-security/site/docs/current/reference/html/security-filter-chain.html>

Чтобы получить полный список фильтров сервлетов и их функций, перейдите по ссылке на Spring Security

<https://docs.spring.io/spring-security/site/docs/4.2.1.RELEASE/reference/htmlsingle/#filter-ordering>

Общая схема аутентификации и авторизации работает таким образом:



Spring Security структура модулей

Spring Security является проектом верхнего уровня. В рамках проекта Spring Security (<https://github.com/spring-projects/spring-security>) есть несколько модулей:

Core (spring-security-core): Spring security's базовые классы аутентификация и контроль доступа.

Remoting (spring-security-remoting).

Aspect (spring-security-aspects): Aspect-Oriented Programming (AOP)поддержка.

Config (spring-security-config): XML и Java конфигурация.

Crypto (spring-security-crypto).

Data (spring-security-data): Интеграция с Spring Data.

Messaging (spring-security-messaging)

OAuth2: Поодержка OAuth 2.x

Core (spring-security-oauth2-core)

Client (spring-security-oauth2-client)

JOSE (spring-security-oauth2-jose)

OpenID (spring-security-openid).

CAS (spring-security-cas):Central AuthenticationService

TagLib(spring-security-taglibs).

Test (spring-security-test).

Web (spring-security-web): Содержит фильтры , Servlet API

и и.д.

+ несколько проектов

Security Assertion Markup Language (SAML)

Lightweight Directory Access Protocol (LDAP)

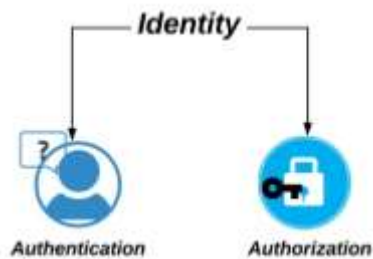
Kerberos.

Authentication и Authorization

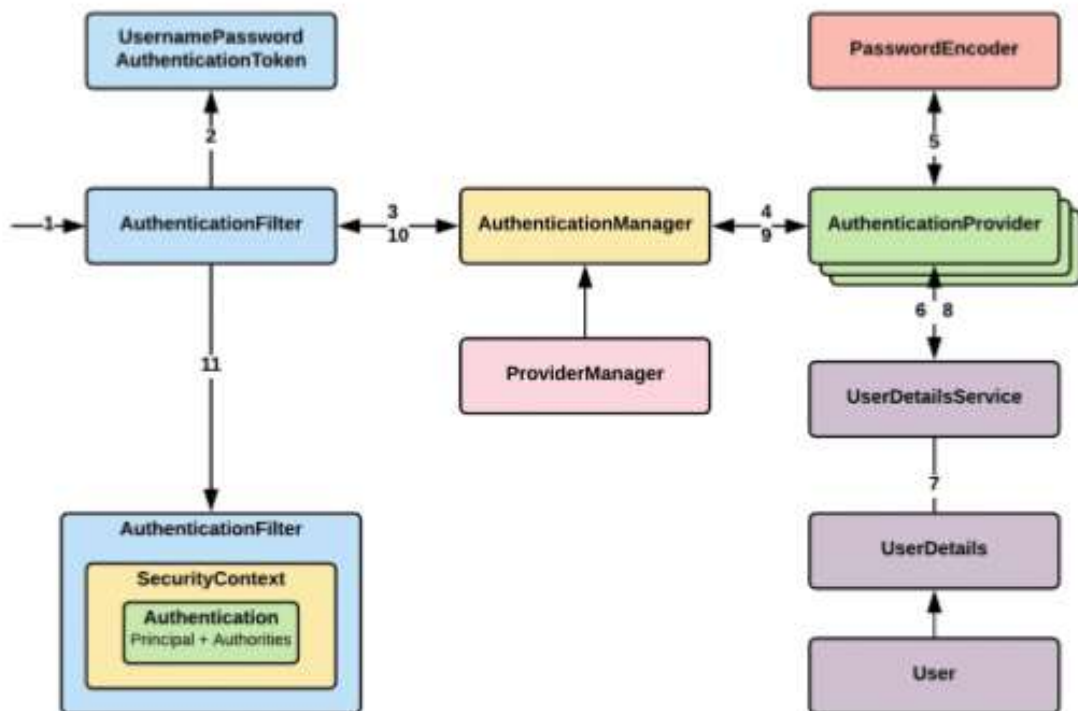
Два наиболее важных аспекта безопасности:

- Найти пользователя.
- Найти ресурсы, к которым у этого пользователя есть доступ

Аутентификация - это механизм, с помощью которого вы узнаете, кто такой пользователь, а **авторизация** - это механизм, который позволяет приложению выяснить, что пользователь может делать с приложением



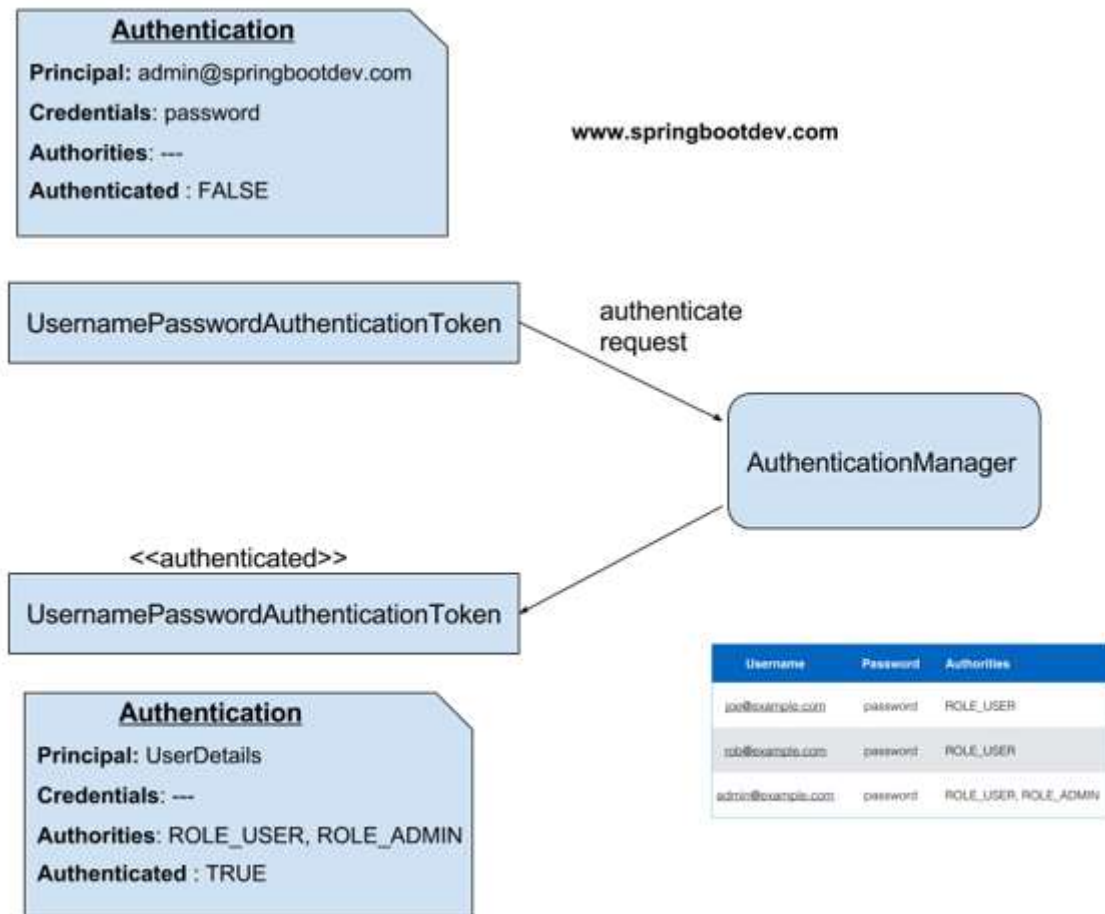
Authentication



```
public interface AuthenticationManager {  
    Authentication authenticate(Authentication authentication) throws  
    AuthenticationException; }  
throws
```

Метод *authenticate* в **AuthenticationManager** может возвращать следующее:

- объект **Authentication** с `authenticated = true`, если Spring Security может проверить предоставленные учетные данные пользователя
- **AuthenticationException**, если Spring Security обнаружит, что предоставленные учетные данные пользователя не действительны,
- **null** если Spring Security не может определить, является ли оно истинным или ложным (состояние с ошибками)



```
public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication) throws
    AuthenticationException;
    boolean supports(Class<?> authentication); }
```

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException; }
```

Настройка AuthenticationManager

AuthenticationManagerBuilder.

WebSecurityConfigurerAdapter - это класс, который расширяется файлом конфигурации Spring, что упрощает внедрение Spring Security в приложение Spring.

Например, настроим глобальный AuthenticationManager с помощью аннотации @Autowired

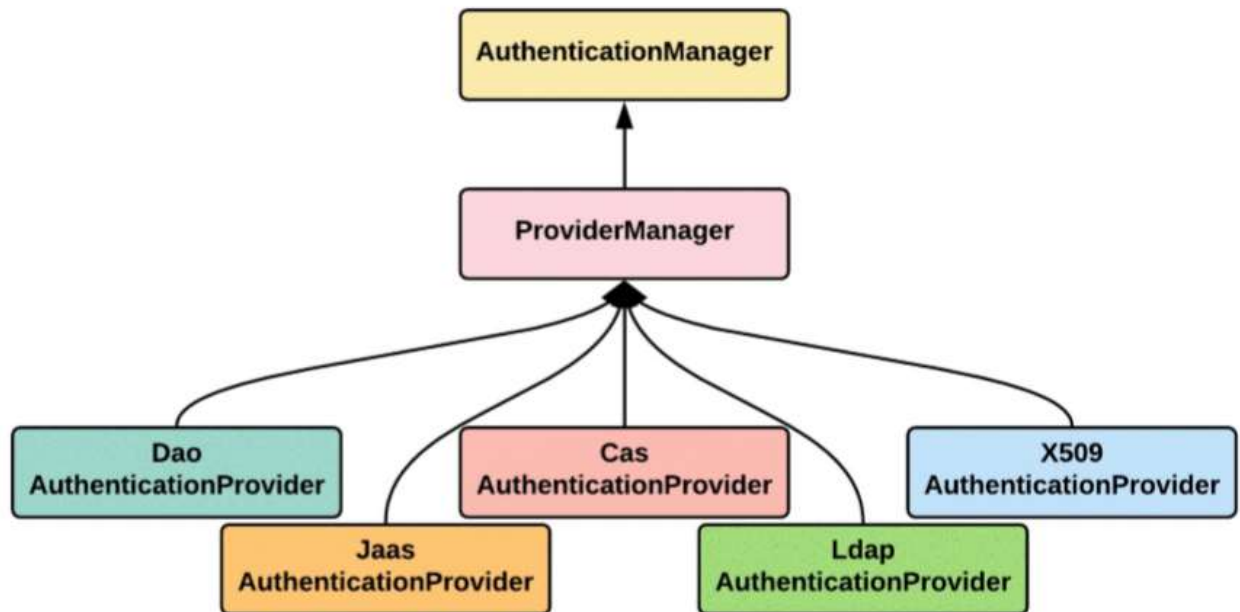
```
@Configuration
@EnableWebSecurity
class SpringSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    public void confGlobalAuthManager(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()
            .withUser("admin")
            .password("admin")
            .roles("ADMIN");
    }
}
```

Вы также можете создать локальный AuthenticationManager, который доступен только для этого конкретного WebSecurityConfigurerAdapter, переопределив метод configure, как показано в следующем коде:

```
@Configuration
@EnableWebSecurity
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
            .withUser("admin").
            password("admin").
            roles("ADMIN");
    }
}
```

AuthenticationProvider



```
@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {
    @Override
    public Authentication authenticate(Authentication authentication) throws
    AuthenticationException {
        String username = authentication.getName();
        String password = authentication.getCredentials().toString();
        if ("user".equals(username) && "password".equals(password)) {
            return new UsernamePasswordAuthenticationToken(username, password,
            Collections.emptyList());
        } else {
            throw new BadCredentialsException("Authentication failed");
        }
    }

    @Override
    public boolean supports(Class<?> aClass) {
        return aClass.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

Multiple AuthenticationProvider

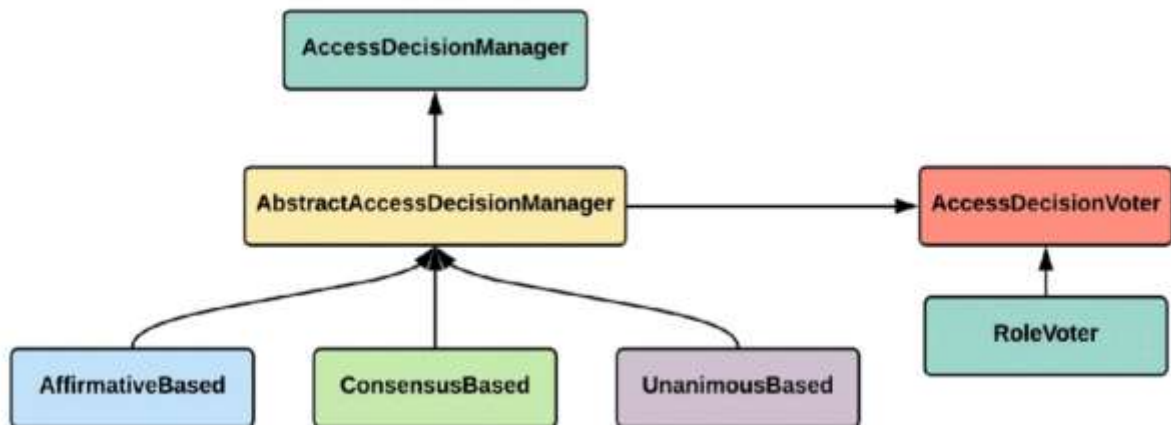
```
@EnableWebSecurity
@ComponentScan(basePackageClasses = CustomAuthenticationProvider.class)
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    CustomAuthenticationProvider customAuthenticationProvider;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic().and().authorizeRequests().antMatchers("/**").authenticated();    //
        Use Basic authentication
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        // Custom authentication provider - Order 1
        auth.authenticationProvider(customAuthenticationProvider);    // Built-
        in authentication provider - Order 2

        auth.inMemoryAuthentication().withUser("admin").password("{noop}admin@password")
            .roles("ADMIN")    // Role of the user
            .and()
            .withUser("user")
            .password("{noop}user@password")
            .credentialsExpired(true)
            .accountExpired(true)
            .accountLocked(true).roles("USER");
    }
}
```

Authorization



Аторизация для защищенного ресурса предоставляется путем вызова избирателей и последующего подсчета полученных голосов. Три встроенные реализации подсчитывают голоса, полученные по-разному:

AffirtivesBased: если хотя бы один голосующий дает голос, пользователю предоставляется доступ к защищенному ресурсу.

ConsensusBased: если между избирателями и их голосами достигнут четкий консенсус, то пользователь получают доступ к ресурсу

UnanimousBased: если все избиратели голосуют, то пользователю предоставляется доступ к защищенному ресурсу.

Spring Security предоставляет два подхода к авторизации:

Web-URL: входящий URL-адрес (определенный URL-адрес или регулярное выражение).

Method: Контроль доступа идет по сигнатуре метода.

Web URL

Здесь идет авторизация сопоставления с образцом

```
http.antMatcher("/rest/**").  
    httpBasic()  
    .disable()  
    .authorizeRequests()  
        .antMatchers("/rest/movie/**", "/rest/ticket/**", "/index")  
            .hasRole("ROLE_USER");
```

MvcRequestMatcher: использует Spring MVC для сопоставления пути и затем извлекает переменные.

RegexRequestMatcher: использует регулярное выражение для соответствия URL. При необходимости он также может использоваться для соответствия методу HTTP. Соответствие является CaseSensitive и принимает форму (servletPath + PathInfo + QueryString):

```
http  
    .authorizeRequests()  
    .regexMatchers("^((?!(/rest|advSearch)).)*$")  
        .hasRole("ADMIN")  
    .regexMatchers("^((?!(/rest|basicSearch)).)*$")  
        .access("hasRole(USER)")  
        .anyRequest()  
    .authenticated()  
    .and()  
    .httpBasic();
```

Method invocation

Если вы хотите включить защиту методов в приложении, сначала аннотируйте класс с помощью **@EnableMethodSecurity**.

Существует три типа аннотаций, с помощью которых вы можете аннотировать методы и авторизовать их:

- *Аннотации на основе голосования:* наиболее часто используемые аннотации в Spring Security **@Secured**. Их нужно включить:

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    ... }
```

После этого использование аннотации разрешено и можно использовать **@Secured**:

```
@DeleteMapping(value = "/delete/{id}")
@Secured(ROLE_ADMIN)
@ResponseStatus(HttpStatus.OK)
public void deletePerson(@PathVariable("id") Long id) throws
ResourceNotFoundException {
    personService.deletePerson(personService.getById(id));
}
```

- *Аннотации JSR-250:* аннотации безопасности Enterprise JavaBeans 3.0 (EJB 3). Перед использованием этих аннотаций их необходимо включить с помощью

@EnableGlobalMethodSecurity (jsr250Enabled = true)

В следующем фрагменте показана аннотация безопасности JSR-250 в действии:

```
@GetMapping(value = {"/all"})
@PermitAll
public CollectionModel<EntityModel<PersonDto>> personList() {
    // return Mapper.mapAll(personService.getAllPerson(), PersonDto.class);

    CollectionModel<EntityModel<PersonDto>> resource = CollectionModel.wrap(
        Mapper.mapAll(personService.getAllPerson(), PersonDto.class));

    for (final EntityModel<PersonDto> res : resource) {
        Link selfLink = LinkTo(PersonController.class)
            .slash(res.getContent().getPersonId()).withSelfRel();
        res.add(selfLink);
    }
    resource.add(LinkTo(methodOn(PersonController.class)
        .personList()).withRel("all"));
    return resource;
}
```

```

    }
    @DeleteMapping(value = "/delete/{id}")
    @RolesAllowed({"ROLE_ADMIN"})
    @ResponseStatus(HttpStatus.OK)
    public void deletePerson(@PathVariable("id") Long id) throws
ResourceNotFoundException {
        personService.deletePerson(personService.getById(id));
    }
}

```

- *Expression-based annotation* .Аннотации на основе @Pre и @Post попадают в эту категорию. Они включены с помощью

```
@EnableGlobalMethodSecurity (prePostEnabled = true)
```

```

@GetMapping(value = {"/all"})
@PreAuthorize("permitAll()")
public CollectionModel<EntityModel<PersonDto>> personList() {
    ...
}

```

```

@DeleteMapping(value = "/delete/{id}")
@PreAuthorize("hasAnyAuthority('ROLE_ADMIN')")
@ResponseStatus(HttpStatus.OK)
public void deletePerson(@PathVariable("id") Long id) throws
ResourceNotFoundException {
    personService.deletePerson(personService.getById(id));
}

```

Spring Expression Language (SpEL)

Domain instance

. Spring Security Access Control List (ACL)

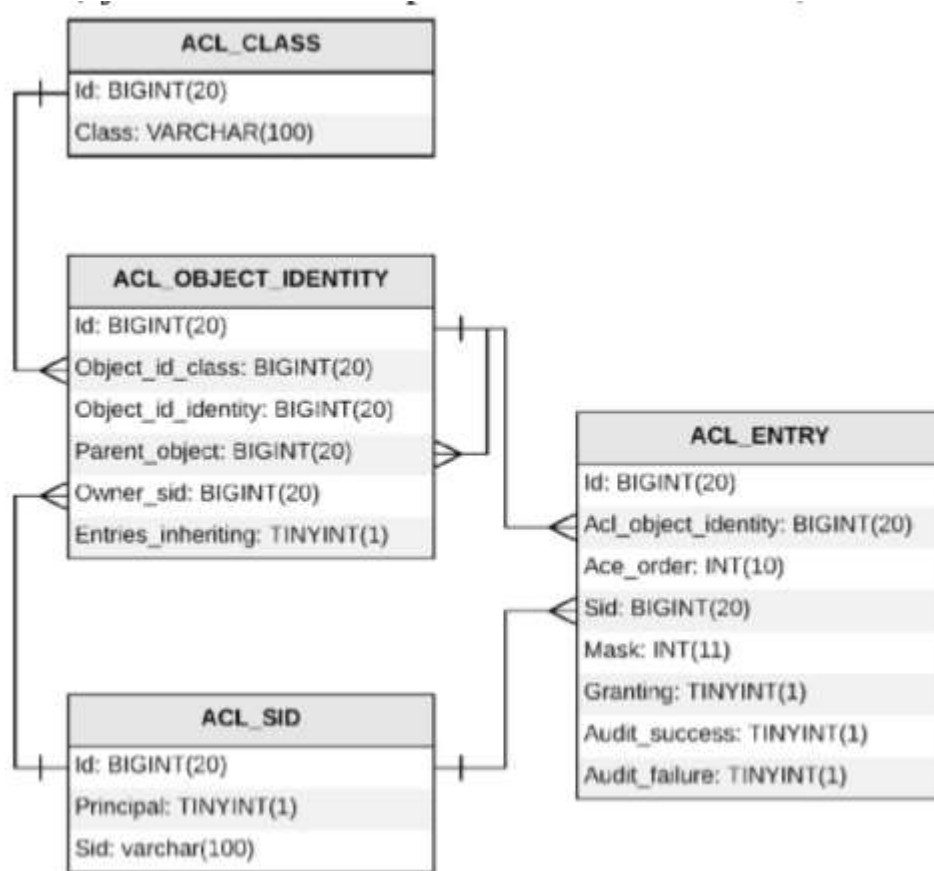


Таблица **ACL_CLASS**: в ней хранится имя класса объекта домена.

Таблица **ACL_SID**: Security Identity (SID) хранит либо имя пользователя (testuser), либо имя роли (ROLE_ADMIN). Столбец **PRINCIPAL** хранит 0 или 1, 0, если SID является именем пользователя, и 1, если это имя роли.

Таблица **ACL_OBJECT_IDENTITY**: ей поручено хранить информацию об объектах и связывать другие таблицы.

Таблица **ACL_ENTRY**: хранит разрешение, предоставленное каждому идентификатору безопасности для каждого **OBJECT_IDENTITY**.

Spring Security ACL также требует кэш. EhCache - одна из самых простых для интеграции с Spring.

Поддерживаются следующие разрешения:

READ

WRITE

CREATE

DELETE

ADMINISTRATION

Включить его можно:

```
@EnableGlobalMethodSecurity (prePostEnabled = true, securedEnabled = true)
```

Теперь - аннотации для начала контроля доступа к объектам домена

```
@GetMapping(value = {"/all"})
@PostFilter("hasPermission (filterObject, 'READ')")
public CollectionModel<EntityModel<PersonDto>> personList() { ..
```

После запроса записей (постфильтрация) анализируется результат (список), и выполняется фильтрация, чтобы вернуть только объект, на который у пользователя есть разрешение READ.

Можно использовать @PostAuthorize следующим образом:

```
@GetMapping(value = {"/all"})
@PostAuthorize("hasPermission (returnObject, 'READ')")
public CollectionModel<EntityModel<PersonDto>> personList() {
```

После выполнения метода (@Post), если пользователь имеет доступ READ к объекту, он возвращается. В противном случае генерируется исключение **AccessDeniedException**:

```
@PostMapping("/add")
@ResponseStatus(HttpStatus.CREATED)
@PreAuthorize("hasPermission(#person, 'WRITE')")
```

Можно использовать:

```
hasRole([role_name])
hasAnyRole([role_name1, role_name2])
hasAuthority([authority])
hasAnyAuthority([authority1, authority2])
permitAll
denyAll
isAnonymous()
isRememberMe()
isAuthenticated()
isFullyAuthenticated()
hasPermission(Object target, Object permission)
```

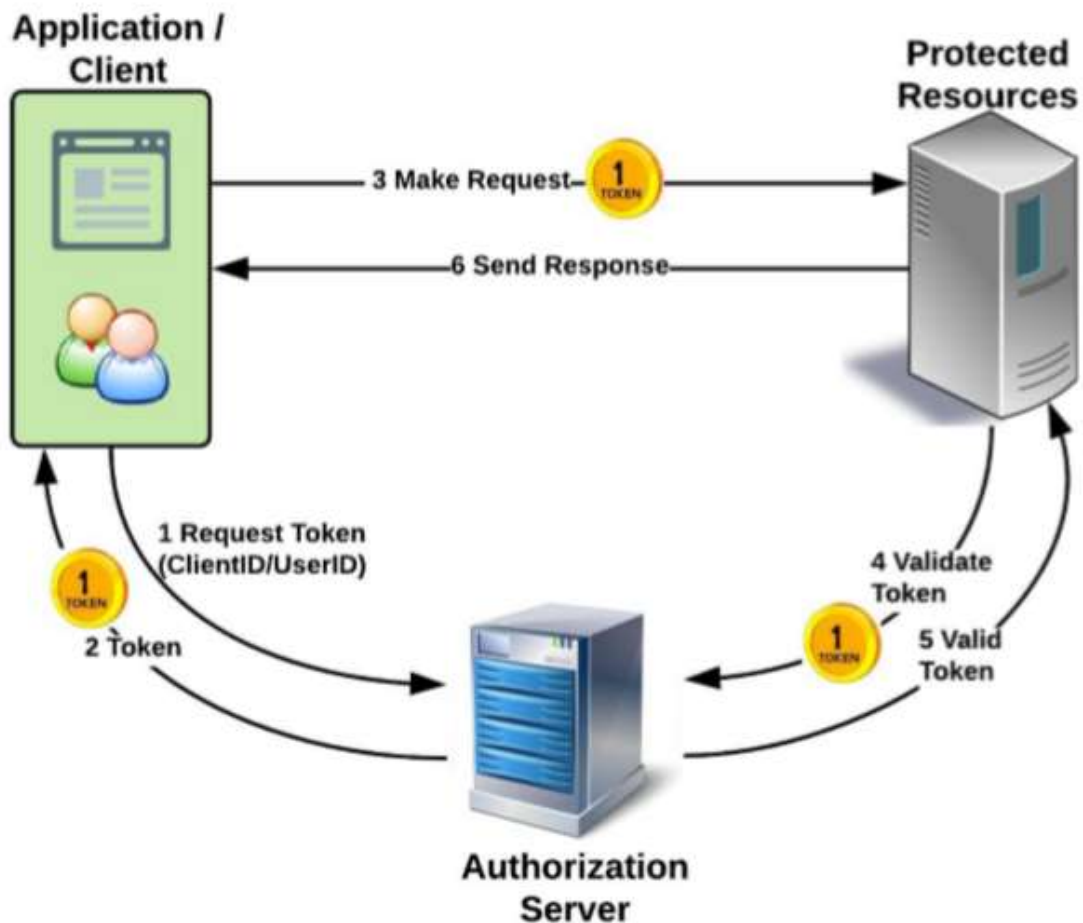
OAuth2 и OpenID

OAuth является открытым стандартом / спецификация для авторизации. Он работает через HTTPS.

Есть две версии

OAuth 1.0 (<https://tools.ietf.org/html/rfc5849>)

OAuth 2.0 (<https://tools.ietf.org/html/rfc6749>)



Open ID Connect (OIDC)

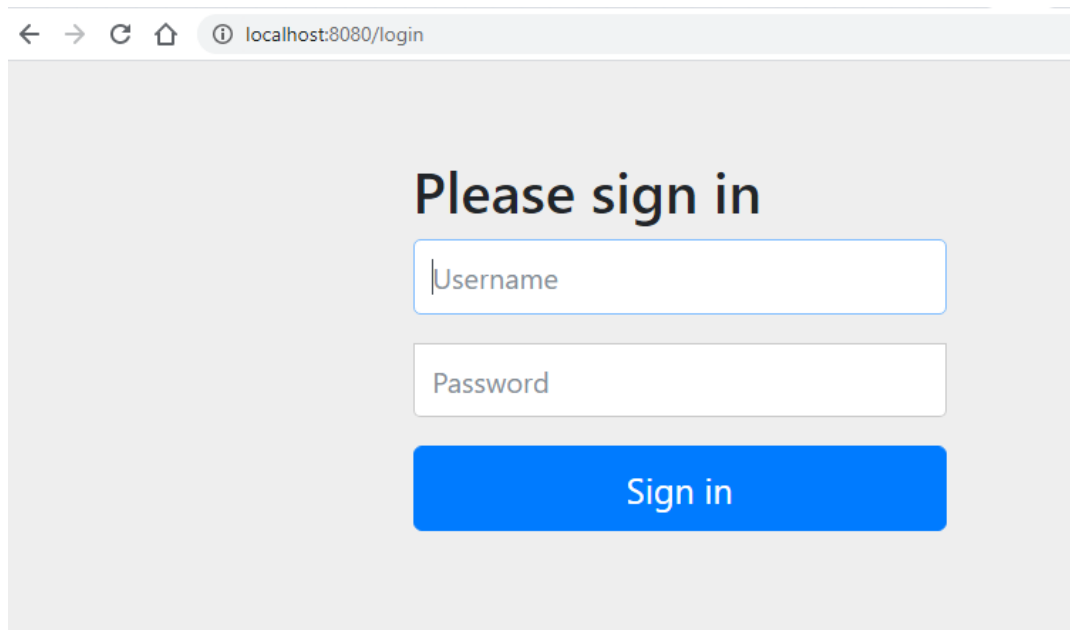
OIDC ввел новый токен ID (`id_token`), а также конечную точку **UserInfo**, которая обеспечит минимальные пользовательские атрибуты.

Пример

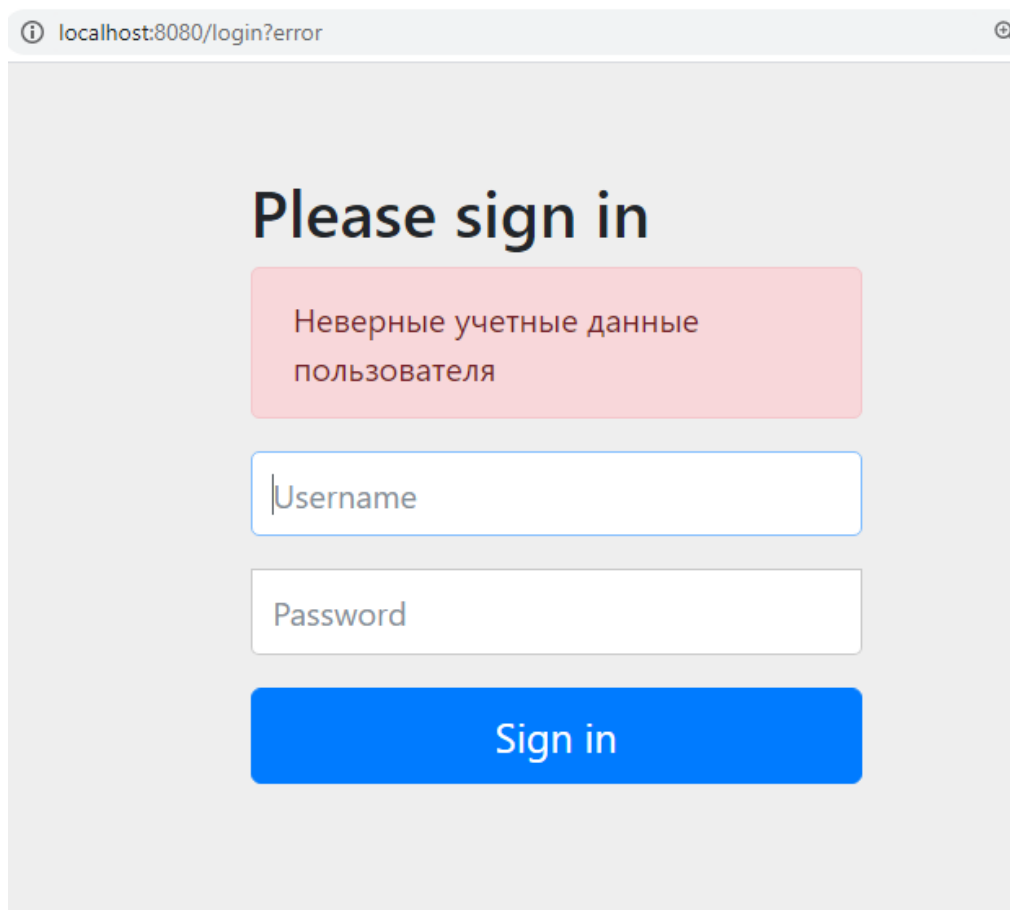
Необходимо подключение. spring-boot-oauth-oidc-authentication.zip
Можно со Spring Boot или без

Добавим application.properties

```
spring.security.user.name=name  
spring.security.user.password=password  
spring.security.basic.authorize-mode=authenticated  
spring.basic.path = /**
```



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/login'. The page content is centered and features the heading 'Please sign in'. Below the heading are two text input fields: the first is labeled 'Username' and the second is labeled 'Password'. At the bottom of the form is a prominent blue button with the text 'Sign in'.



Далее в файле свойств объявляем два свойства для каждого поставщика.

Например, будем внедрять поставщика Google, но вы можете добавить любое количество поставщиков. Простое добавление этих свойств изменит страницу входа :

```
spring.security.oauth2.client.registration.<provider_name>.client-id=<client id>
spring.security.oauth2.client.registration.<provider_name>.client-secret=<client secret>
```

Например так

#Google app details

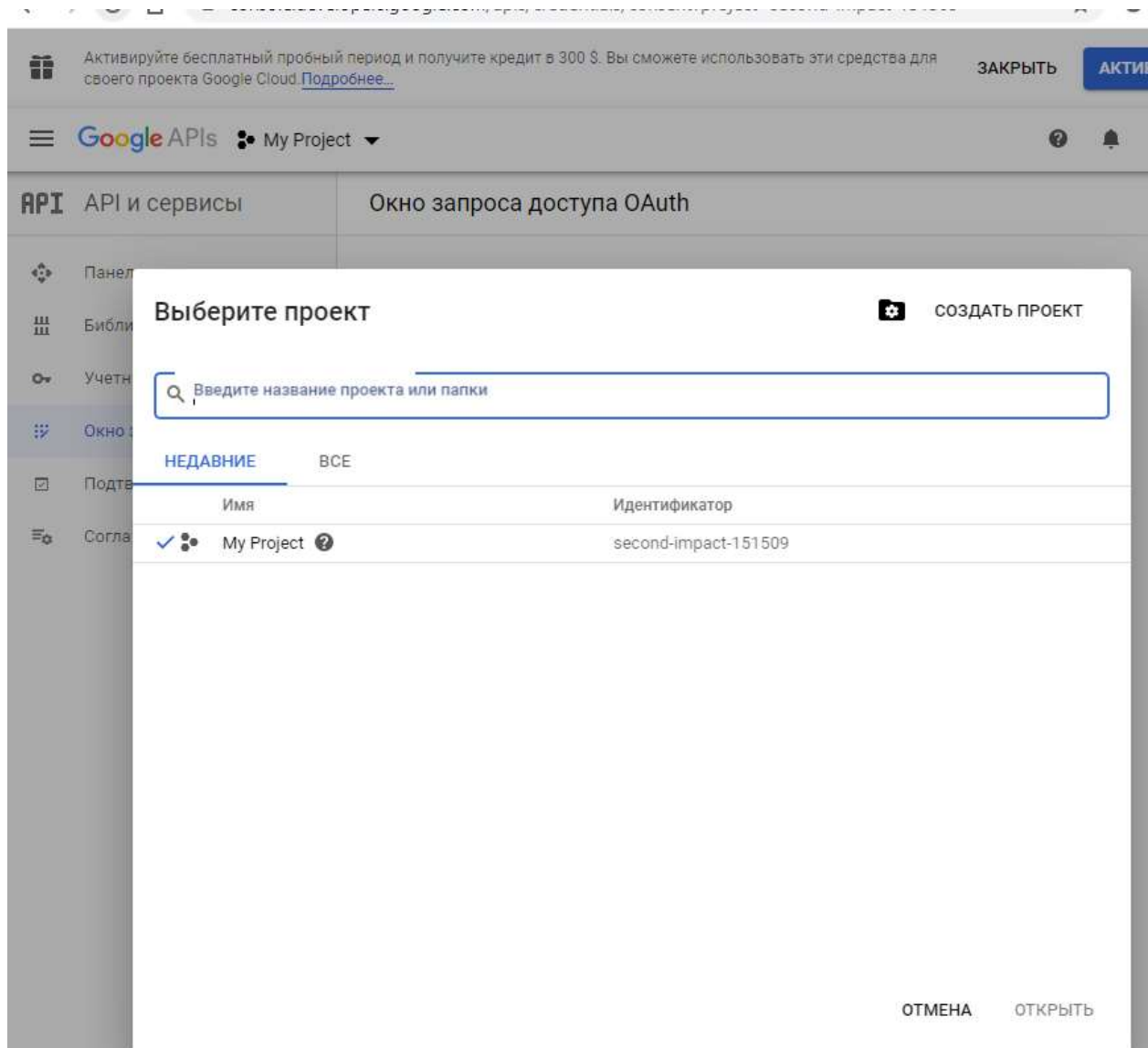
```
spring.security.oauth2.client.registration.google.client-id=1085570125bbh18j2r88b5i5gb73vkht1f8j7u3hvd78.apps.googleusercontent.com
spring.security.oauth2.client.registration.google.client-secret=MdtcKpArG51Feffggp
```

Настройка провайдера

1.Перейдите по ссылке

<https://console.developers.google.com/>

2. Создайте проект

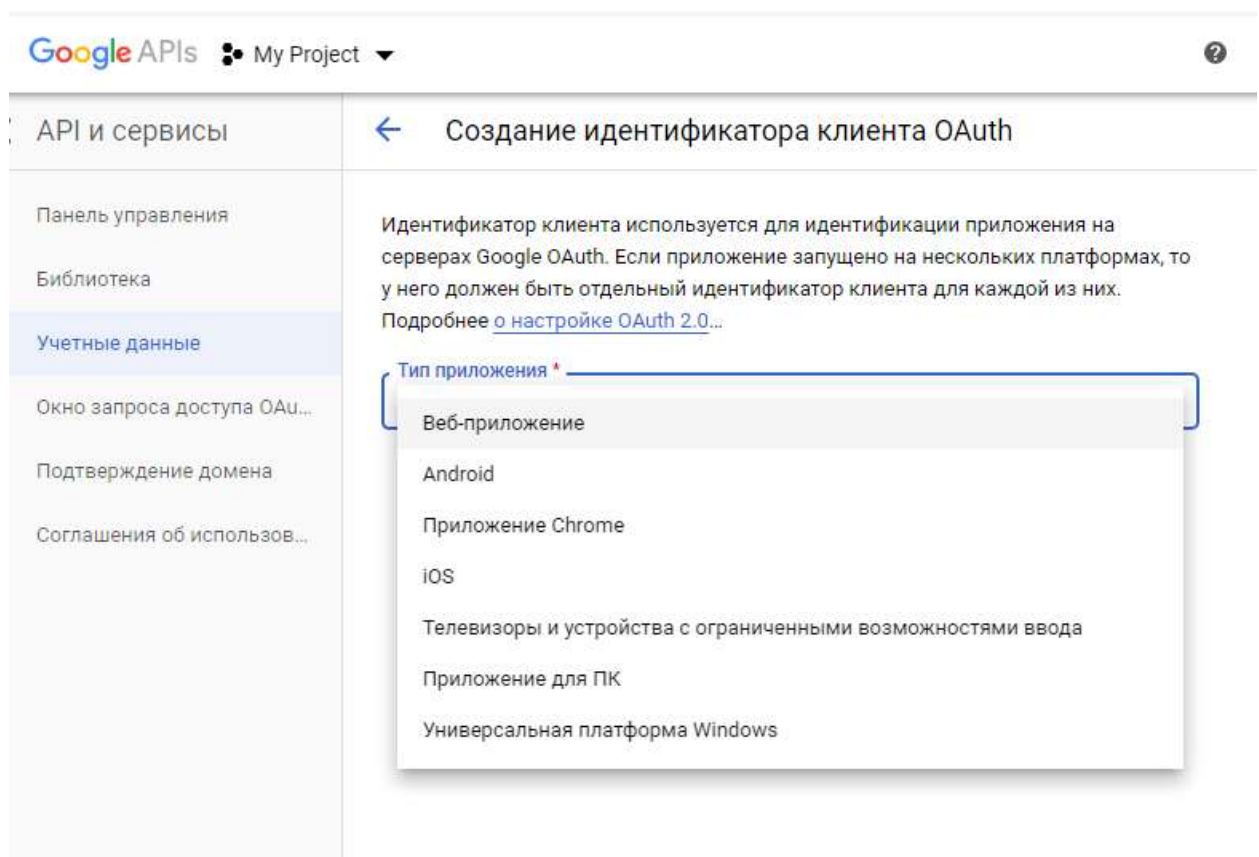
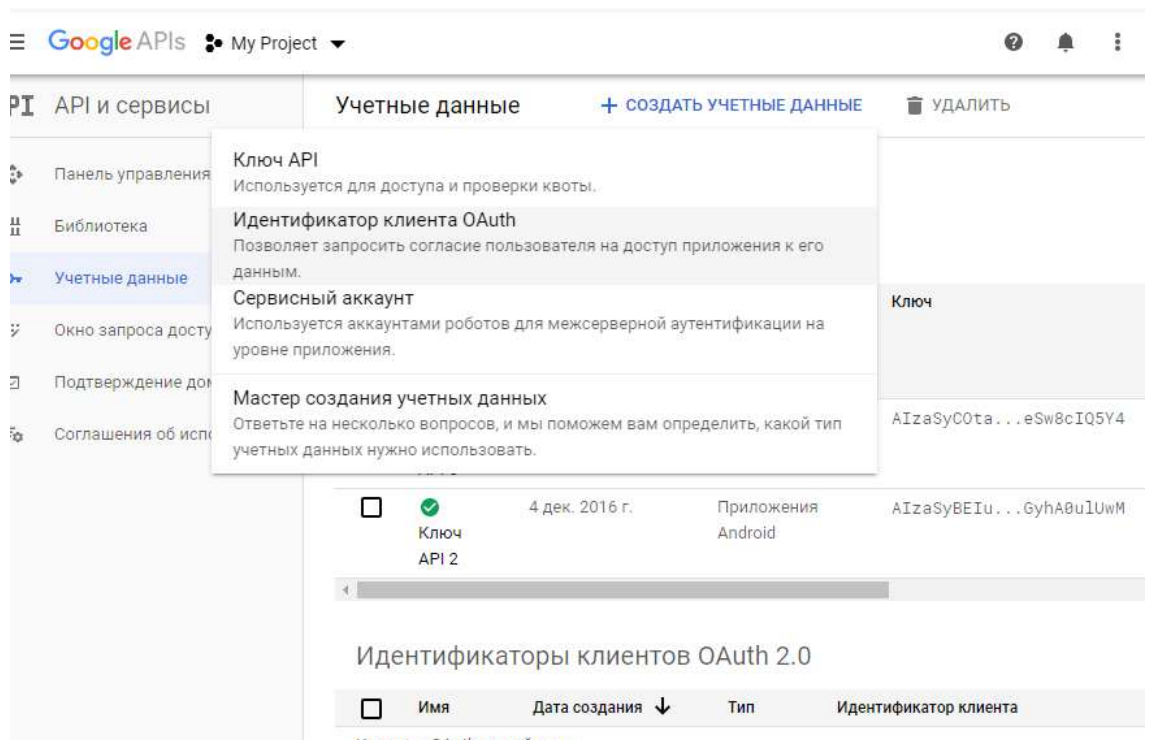


3. Создайте credentials.

Выберите только что созданный проект (на следующем снимке экрана он показан рядом с логотипом API Google) и нажмите ссылку Credentials (учетные данные) в боковом меню:

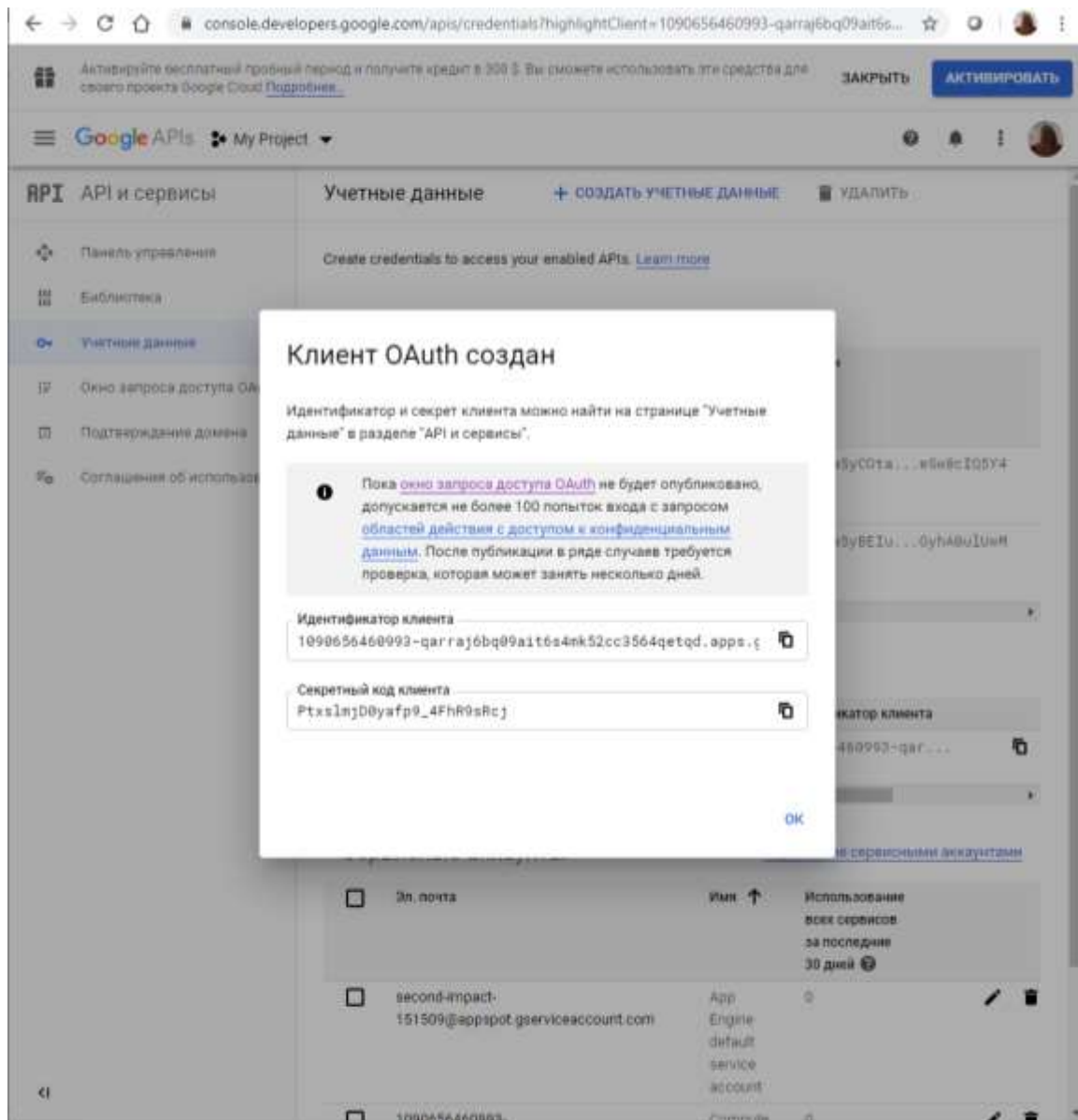
4. Нажмите Создать

5. Из выпадающего меню выберите OAuth идентификатор клиента. Это приведет вас к странице, показанной на следующем снимке экрана.



Выполните конфигурацию. Введите соответствующие данные (оставьте необязательные поля во время заполнения формы) и нажмите кнопку Сохранить.

Выберите тип приложения в качестве веб-приложения и введите соответствующие данные. Нажмите на кнопку Создать, и вам будет показано следующее всплывающее окно:



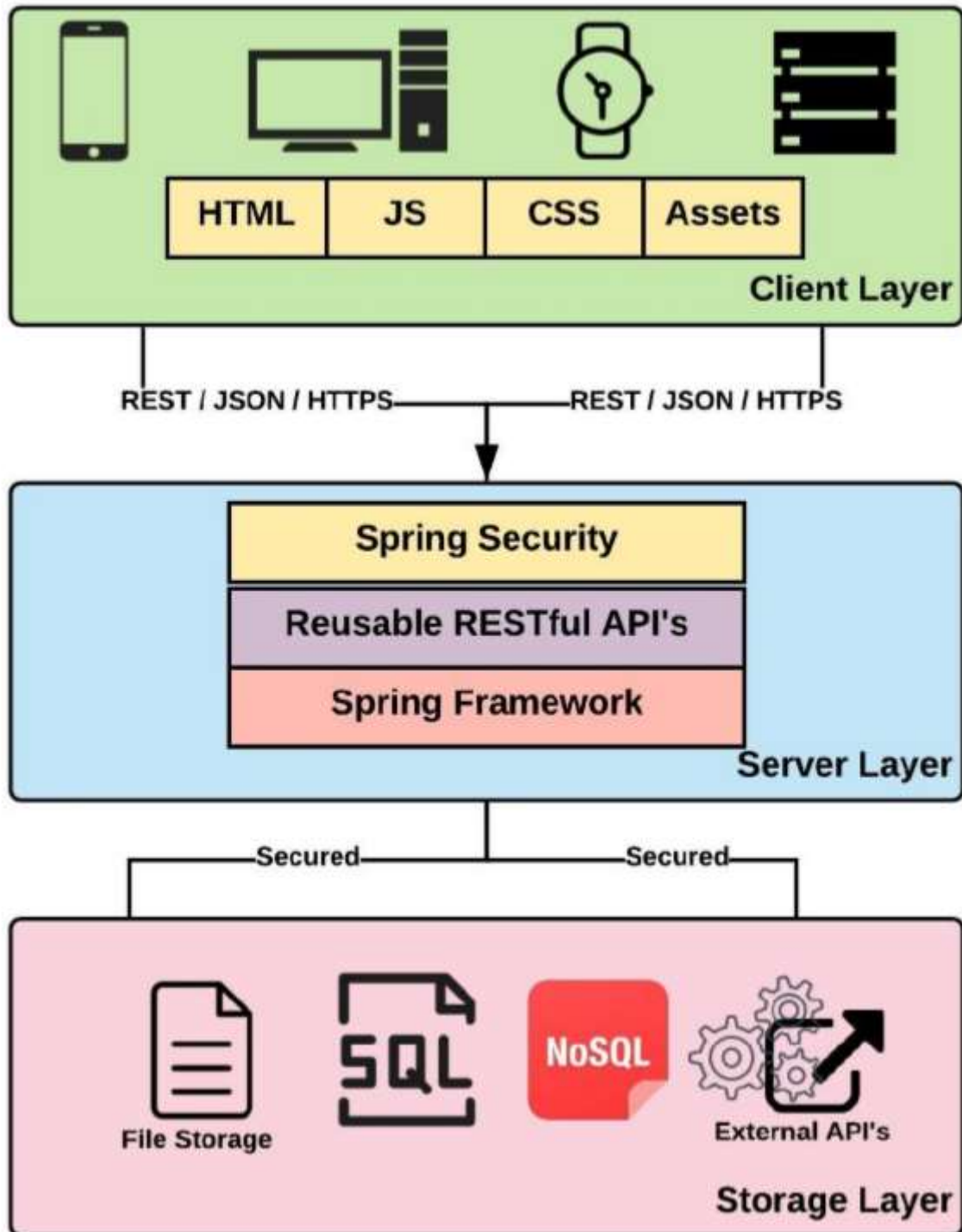
Теперь у вас есть идентификатор клиента и секретный код клиента от Google. Скопируйте и вставьте эти значения в файл свойств `.properties`.

← → ↻ 🏠 ⓘ localhost:8080/login

Login with OAuth 2.0

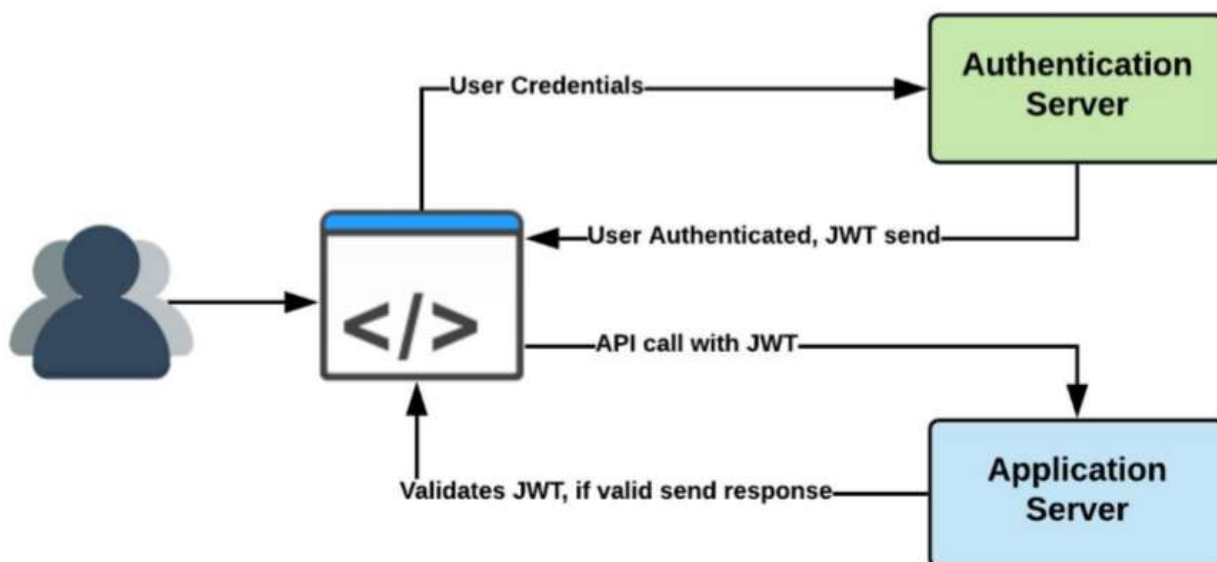
Google

При возникновении проблем читайте тут
<https://developers.google.com/identity/protocols/oauth2/openid-connect#setredirecturi>



JSON Web Token (JWT)

JSON Web Tokens - это открытый промышленный стандарт RFC 7519 для безопасного представления заявок между двумя сторонами
<https://jwt.io/>



преимущества использования JWT:

- *Повышение производительности*
- *Простота*