# About JWT Wokring

# 

# 

# 

# Refresh Token

# 

# Spring Boot Security and JWT

[Last modified: June 16, 2023](https://www.bezkoder.com/spring-boot-security-jwt/)  [bezkoder](https://www.bezkoder.com/author/bezkoder/)  [Security](https://www.bezkoder.com/category/security/), [Spring](https://www.bezkoder.com/category/spring/)

In this tutorial, we’re gonna build a Spring Boot, Spring Security that supports JWT working with H2 embedded Database. You’ll know:

* Flow for User Login, Registration, Authorization with JWT and HttpOnly Cookie
* Spring Boot Rest Api Architecture with Spring Security and JWT
* How to configure Spring Security to work with JWT
* How to define Data Models and association for Authentication and Authorization
* Way to use Spring Data JPA to interact with H2 Database

– Related Posts:

* [Using Token in HTTP Authorization Headers](https://www.bezkoder.com/spring-boot-jwt-authentication/)
* [Spring Boot JPA + H2 example: CRUD Rest APIs](https://www.bezkoder.com/spring-boot-jpa-h2-example/)
* [Spring Boot File upload example](https://bezkoder.com/spring-boot-file-upload/)
* [@RestControllerAdvice example in Spring Boot](https://bezkoder.com/spring-boot-restcontrolleradvice/)
* [Spring Boot @ControllerAdvice & @ExceptionHandler example](https://bezkoder.com/spring-boot-controlleradvice-exceptionhandler/)
* [@DataJpaTest example for Spring Data Repository Unit Test](https://bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/)
* Documentation: [Spring Boot + Swagger 3 example (with OpenAPI 3)](https://www.bezkoder.com/spring-boot-swagger-3/)
* [Spring Boot Redis Cache example](https://www.bezkoder.com/spring-boot-redis-cache-example/)
* [Spring Boot custom Validation example](https://www.bezkoder.com/spring-boot-custom-validation/)

Other Databases:  
– [Spring Boot, Spring Security example with JWT and MySQL](https://www.bezkoder.com/spring-boot-login-example-mysql/)  
– [Spring Boot, Spring Security example with JWT and MongoDB](https://www.bezkoder.com/spring-boot-mongodb-login-example/)

**Contents**[[hide](https://www.bezkoder.com/spring-boot-security-jwt/)]

* [Overview](https://www.bezkoder.com/spring-boot-security-jwt/#Overview)
* [Flow](https://www.bezkoder.com/spring-boot-security-jwt/#Flow)
* [Architecture](https://www.bezkoder.com/spring-boot-security-jwt/#Architecture)
* [Technology](https://www.bezkoder.com/spring-boot-security-jwt/#Technology)
* [Project Structure](https://www.bezkoder.com/spring-boot-security-jwt/#Project_Structure)
* [Setup Project](https://www.bezkoder.com/spring-boot-security-jwt/#Setup)
* [Configuration](https://www.bezkoder.com/spring-boot-security-jwt/#Configure)
* [Create the models](https://www.bezkoder.com/spring-boot-security-jwt/#Models)
* [Implement Repositories](https://www.bezkoder.com/spring-boot-security-jwt/#Repositories)
* [Configure Spring Security](https://www.bezkoder.com/spring-boot-security-jwt/#Configure_Spring_Security)
* [Implement UserDetails & UserDetailsService](https://www.bezkoder.com/spring-boot-security-jwt/#UserDetails_UserDetailsService)
* [Filter the Requests](https://www.bezkoder.com/spring-boot-security-jwt/#Filter_the_Requests)
* [Create JWT Utility class](https://www.bezkoder.com/spring-boot-security-jwt/#Create_JWT_Utility_class)
* [Handle Exception](https://www.bezkoder.com/spring-boot-security-jwt/#Handle_Authentication_Exception)
* [Define payloads](https://www.bezkoder.com/spring-boot-security-jwt/#Define_payloads)
* [Create Rest Controllers](https://www.bezkoder.com/spring-boot-security-jwt/#Create_Controllers)
* [Run & Check](https://www.bezkoder.com/spring-boot-security-jwt/#Run)
* [Conclusion](https://www.bezkoder.com/spring-boot-security-jwt/#Conclusion)
* [Source Code](https://www.bezkoder.com/spring-boot-security-jwt/#Source_Code)
* [Further Reading](https://www.bezkoder.com/spring-boot-security-jwt/#Further_Reading)

## Overview of Spring Boot Security JWT example

We will build a Spring Boot + Spring Security application with JWT in that:

* User can signup new account (registration), or login with username & password.
* By User’s role (admin, moderator, user), we authorize the User to access resources.

These are APIs that we need to provide:

| **Methods** | **Urls** | **Actions** |
| --- | --- | --- |
| POST | /api/auth/signup | signup new account |
| POST | /api/auth/signin | login an account |
| POST | /api/auth/signout | logout the account |
| GET | /api/test/all | retrieve public content |
| GET | /api/test/user | access User’s content |
| GET | /api/test/mod | access Moderator’s content |
| GET | /api/test/admin | access Admin’s content |

The database we will use is H2 by configuring project dependency & datasource.

## Flow of Spring Boot Security JWT example

The diagram shows flow of how we implement User Registration, User Login/Logout and Authorization process.

A legal JWT will be stored in **HttpOnly Cookie** if Client accesses protected resources.

You may need to implement Refresh Token:

More details at: [Spring Boot Refresh Token with JWT example](https://www.bezkoder.com/spring-security-refresh-token/)

## Spring Boot Architecture with Spring Security

You can have an overview of our Spring Boot Security JWT example with the diagram below:

Now I will explain it briefly.

**Spring Security**

– WebSecurityConfig is the crux of our security implementation. It configures cors, csrf, session management, rules for protected resources. We can also extend and customize the default configuration that contains the elements below.  
(WebSecurityConfigurerAdapter is deprecated from Spring 2.7.0, you can check the source code for update. More details at:  
[WebSecurityConfigurerAdapter Deprecated in Spring Boot](https://www.bezkoder.com/websecurityconfigureradapter-deprecated-spring-boot/))

– [UserDetailsService](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/" \l "tech-userdetailsservice) interface has a method to load User by username and returns a UserDetails object that Spring Security can use for authentication and validation.

– UserDetails contains necessary information (such as: username, password, authorities) to build an Authentication object.

– [UsernamePasswordAuthenticationToken](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/UsernamePasswordAuthenticationToken.html) gets {username, password} from login Request, AuthenticationManager will use it to authenticate a login account.

– [AuthenticationManager](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/" \l "core-services-authentication-manager) has a DaoAuthenticationProvider (with help of UserDetailsService & PasswordEncoder) to validate UsernamePasswordAuthenticationToken object. If successful, AuthenticationManager returns a fully populated Authentication object (including granted authorities).

– [OncePerRequestFilter](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/filter/OncePerRequestFilter.html) makes a single execution for each request to our API. It provides a doFilterInternal() method that we will implement parsing & validating JWT, loading User details (using UserDetailsService), checking Authorizaion (using UsernamePasswordAuthenticationToken).

– [AuthenticationEntryPoint](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/web/AuthenticationEntryPoint.html) will catch authentication error.

**Repository** contains UserRepository & RoleRepository to work with Database, will be imported into **Controller**.

**Controller** receives and handles request after it was filtered by OncePerRequestFilter.

– AuthController handles signup/login requests

– TestController has accessing protected resource methods with role based validations.

Understand the architecture deeply and grasp the overview more easier:  
[Spring Boot Architecture for JWT with Spring Security](https://bezkoder.com/spring-boot-jwt-mysql-spring-security-architecture/)

## Technology

* Java 17 / 11 / 8
* Spring Boot 3 / 2 (with Spring Security, Spring Web, Spring Data JPA)
* jjwt 0.11.5
* H2 – embedded database
* Maven

## Project Structure

This is folders & files structure for our Spring Boot Security JWT example:

**security**: we configure Spring Security & implement Security Objects here.

* WebSecurityConfig

(WebSecurityConfigurerAdapter is deprecated from Spring 2.7.0, you can check the source code for update. More details at:  
[WebSecurityConfigurerAdapter Deprecated in Spring Boot](https://www.bezkoder.com/websecurityconfigureradapter-deprecated-spring-boot/))

* UserDetailsServiceImpl implements UserDetailsService
* UserDetailsImpl implements UserDetails
* AuthEntryPointJwt implements AuthenticationEntryPoint
* AuthTokenFilter extends OncePerRequestFilter
* JwtUtils provides methods for generating, parsing, validating JWT

**controllers** handle signup/login requests & authorized requests.

* AuthController: @PostMapping(‘/signup’), @PostMapping(‘/signin’), @PostMapping(‘/signout’)
* TestController: @GetMapping(‘/api/test/all’), @GetMapping(‘/api/test/[role]’)

**repository** has interfaces that extend Spring Data JPA JpaRepository to interact with Database.

* UserRepository extends JpaRepository<User, Long>
* RoleRepository extends JpaRepository<Role, Long>

**models** defines two main models for Authentication (User) & Authorization (Role). They have many-to-many relationship.

* User: id, username, email, password, roles
* Role: id, name

**payload** defines classes for Request and Response objects

We also have **application.properties** for configuring Spring Datasource, Spring Data JPA and App properties (such as JWT Secret string or Token expiration time).

## Setup new Spring Boot Security with JWT project

Use [Spring web tool](http://start.spring.io/) or your development tool ([Spring Tool Suite](https://spring.io/tools), Eclipse, [Intellij](https://www.jetbrains.com/idea/download/)) to create a Spring Boot project.

Then open **pom.xml** and add these dependencies:

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-api</artifactId>

<version>0.11.5</version>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-impl</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt-jackson</artifactId>

<version>0.11.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

## Configure Spring Datasource, JPA, App properties

Under src/main/resources folder, open application.properties, add some new lines.

spring.h2.console.enabled=true

# default path: h2-console

spring.h2.console.path=/h2-ui

spring.datasource.url=jdbc:h2:file:./testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=update

# App Properties

bezkoder.app.jwtCookieName=bezkoder

bezkoder.app.jwtSecret= ======================BezKoder=Spring===========================

bezkoder.app.jwtExpirationMs=86400000

* spring.datasource.url: jdbc:h2:mem:[database-name] for In-memory database and jdbc:h2:file:[path/database-name] for disk-based database.
* spring.datasource.username & spring.datasource.password properties are the same as your database installation.
* Spring Boot uses Hibernate for JPA implementation, we configure H2Dialect for H2 Database
* spring.jpa.hibernate.ddl-auto is used for database initialization. We set the value to update value so that a table will be created in the database automatically corresponding to defined data model. Any change to the model will also trigger an update to the table. For production, this property should be validate.
* spring.h2.console.enabled=true tells the Spring to start H2 Database administration tool and you can access this tool on the browser: http://localhost:8080/h2-console.
* spring.h2.console.path=/h2-ui is for H2 console’s url, so the default url http://localhost:8080/h2-console will change to http://localhost:8080/h2-ui.

## Create the models

We’re gonna have 3 tables in database: **users**, **roles** and **user\_roles** for many-to-many relationship.

Let’s define these models.  
In models package, create 3 files:

**ERole enum** in ERole.java.  
In this example, we have 3 roles corresponding to 3 enum.

package com.bezkoder.spring.security.login.models;

public enum ERole {

ROLE\_USER,

ROLE\_MODERATOR,

ROLE\_ADMIN

}

**Role model** in Role.java

package com.bezkoder.spring.security.login.models;

import jakarta.persistence.\*;

@Entity

@Table(name = "roles")

public class Role {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

@Enumerated(EnumType.STRING)

@Column(length = 20)

private ERole name;

public Role() {

}

public Role(ERole name) {

this.name = name;

}

// getters and setters

}

**User model** in User.java.  
It has 5 fields: id, username, email, password, roles.

package com.bezkoder.spring.security.login.models;

import java.util.HashSet;

import java.util.Set;

import jakarta.persistence.\*;

import jakarta.validation.constraints.Email;

import jakarta.validation.constraints.NotBlank;

import jakarta.validation.constraints.Size;

@Entity

@Table(name = "users",

uniqueConstraints = {

@UniqueConstraint(columnNames = "username"),

@UniqueConstraint(columnNames = "email")

})

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank

@Size(max = 20)

private String username;

@NotBlank

@Size(max = 50)

@Email

private String email;

@NotBlank

@Size(max = 120)

private String password;

@ManyToMany(fetch = FetchType.LAZY)

@JoinTable(name = "user\_roles",

joinColumns = @JoinColumn(name = "user\_id"),

inverseJoinColumns = @JoinColumn(name = "role\_id"))

private Set<Role> roles = new HashSet<>();

public User() {

}

public User(String username, String email, String password) {

this.username = username;

this.email = email;

this.password = password;

}

// getters and setters

}

## Implement Repositories

Now, each model above needs a repository for persisting and accessing data. In repository package, let’s create 2 repositories.

**UserRepository**

There are 3 necessary methods that JpaRepository supports.

package com.bezkoder.spring.security.login.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.bezkoder.spring.security.login.models.User;

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

Optional<User> findByUsername(String username);

Boolean existsByUsername(String username);

Boolean existsByEmail(String email);

}

**RoleRepository**

This repository also extends JpaRepository and provides a finder method.

package com.bezkoder.spring.security.login.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.bezkoder.spring.security.login.models.ERole;

import com.bezkoder.spring.security.login.models.Role;

@Repository

public interface RoleRepository extends JpaRepository<Role, Long> {

Optional<Role> findByName(ERole name);

}

## Configure Spring Security

### Without WebSecurityConfigurerAdapter

WebSecurityConfigurerAdapter is deprecated from Spring 2.7.0. More details at:  
[WebSecurityConfigurerAdapter Deprecated in Spring Boot](https://www.bezkoder.com/websecurityconfigureradapter-deprecated-spring-boot/).

In security package, create WebSecurityConfig class.

WebSecurityConfig.java

package com.bezkoder.spring.security.login.security;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.dao.DaoAuthenticationProvider;

import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;

import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import com.bezkoder.spring.security.login.security.jwt.AuthEntryPointJwt;

import com.bezkoder.spring.security.login.security.jwt.AuthTokenFilter;

import com.bezkoder.spring.security.login.security.services.UserDetailsServiceImpl;

@Configuration

@EnableMethodSecurity

//(securedEnabled = true,

//jsr250Enabled = true,

//prePostEnabled = true) // by default

public class WebSecurityConfig {

@Value("${spring.h2.console.path}")

private String h2ConsolePath;

@Autowired

UserDetailsServiceImpl userDetailsService;

@Autowired

private AuthEntryPointJwt unauthorizedHandler;

@Bean

public AuthTokenFilter authenticationJwtTokenFilter() {

return new AuthTokenFilter();

}

@Bean

public DaoAuthenticationProvider authenticationProvider() {

DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();

authProvider.setUserDetailsService(userDetailsService);

authProvider.setPasswordEncoder(passwordEncoder());

return authProvider;

}

@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration authConfig) throws Exception {

return authConfig.getAuthenticationManager();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http.csrf(csrf -> csrf.disable())

.exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))

.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

.authorizeHttpRequests(auth ->

auth.requestMatchers("/api/auth/\*\*").permitAll()

.requestMatchers("/api/test/\*\*").permitAll()

.anyRequest().authenticated()

);

// fix H2 database console: Refused to display ' in a frame because it set 'X-Frame-Options' to 'deny'

http.headers(headers -> headers.frameOptions(frameOption -> frameOption.sameOrigin()));

http.authenticationProvider(authenticationProvider());

http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);

return http.build();

}

}

### With WebSecurityConfigurerAdapter

In security package, create WebSecurityConfig class that extends WebSecurityConfigurerAdapter.

WebSecurityConfig.java

package com.bezkoder.spring.security.login.security;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import com.bezkoder.spring.security.login.security.jwt.AuthEntryPointJwt;

import com.bezkoder.spring.security.login.security.jwt.AuthTokenFilter;

import com.bezkoder.spring.security.login.security.services.UserDetailsServiceImpl;

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(

// securedEnabled = true,

// jsr250Enabled = true,

prePostEnabled = true)

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Value("${spring.h2.console.path}")

private String h2ConsolePath;

@Autowired

UserDetailsServiceImpl userDetailsService;

@Autowired

private AuthEntryPointJwt unauthorizedHandler;

@Bean

public AuthTokenFilter authenticationJwtTokenFilter() {

return new AuthTokenFilter();

}

@Override

public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {

authenticationManagerBuilder.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());

}

@Bean

@Override

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http.cors().and().csrf().disable()

.exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()

.authorizeRequests().antMatchers("/api/auth/\*\*").permitAll()

.antMatchers("/api/test/\*\*").permitAll()

.antMatchers(h2ConsolePath + "/\*\*").permitAll()

.anyRequest().authenticated();

// fix H2 database console: Refused to display ' in a frame because it set 'X-Frame-Options' to 'deny'

http.headers().frameOptions().sameOrigin();

http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);

}

}

Let me explain the code above.

– @EnableWebSecurity allows Spring to find and automatically apply the class to the global Web Security.

\*For **Spring Boot 2**: @EnableGlobalMethodSecurity provides AOP security on methods. It enables @PreAuthorize, @PostAuthorize, it also supports [JSR-250](https://en.wikipedia.org/wiki/JSR_250). You can find more parameters in configuration in [Method Security Expressions](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#method-security-expressions).

– @EnableGlobalMethodSecurity is deprecated in **Spring Boot 3**. You can use @EnableMethodSecurity instead. For more details, please visit [Method Security](https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html).

– We override the configure(HttpSecurity http) method from WebSecurityConfigurerAdapter interface. It tells Spring Security how we configure CORS and CSRF, when we want to require all users to be authenticated or not, which filter (AuthTokenFilter) and when we want it to work (filter before UsernamePasswordAuthenticationFilter), which Exception Handler is chosen (AuthEntryPointJwt).

– Spring Security will load User details to perform authentication & authorization. So it has UserDetailsService interface that we need to implement.

– The implementation of UserDetailsService will be used for configuring DaoAuthenticationProvider by AuthenticationManagerBuilder.userDetailsService() method.

– We also need a PasswordEncoder for the DaoAuthenticationProvider. If we don’t specify, it will use plain text.

## Implement UserDetails & UserDetailsService

If the authentication process is successful, we can get User’s information such as username, password, authorities from an Authentication object.

Authentication authentication =

authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(username, password)

);

UserDetails userDetails = (UserDetails) authentication.getPrincipal();

// userDetails.getUsername()

// userDetails.getPassword()

// userDetails.getAuthorities()

If we want to get more data (id, email…), we can create an implementation of this UserDetails interface.

security/services/UserDetailsImpl.java

package com.bezkoder.spring.security.login.security.services;

import java.util.Collection;

import java.util.List;

import java.util.Objects;

import java.util.stream.Collectors;

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.core.userdetails.UserDetails;

import com.bezkoder.spring.security.login.models.User;

import com.fasterxml.jackson.annotation.JsonIgnore;

public class UserDetailsImpl implements UserDetails {

private static final long serialVersionUID = 1L;

private Long id;

private String username;

private String email;

@JsonIgnore

private String password;

private Collection<? extends GrantedAuthority> authorities;

public UserDetailsImpl(Long id, String username, String email, String password,

Collection<? extends GrantedAuthority> authorities) {

this.id = id;

this.username = username;

this.email = email;

this.password = password;

this.authorities = authorities;

}

public static UserDetailsImpl build(User user) {

List<GrantedAuthority> authorities = user.getRoles().stream()

.map(role -> new SimpleGrantedAuthority(role.getName().name()))

.collect(Collectors.toList());

return new UserDetailsImpl(

user.getId(),

user.getUsername(),

user.getEmail(),

user.getPassword(),

authorities);

}

@Override

public Collection<? extends GrantedAuthority> getAuthorities() {

return authorities;

}

public Long getId() {

return id;

}

public String getEmail() {

return email;

}

@Override

public String getPassword() {

return password;

}

@Override

public String getUsername() {

return username;

}

@Override

public boolean isAccountNonExpired() {

return true;

}

@Override

public boolean isAccountNonLocked() {

return true;

}

@Override

public boolean isCredentialsNonExpired() {

return true;

}

@Override

public boolean isEnabled() {

return true;

}

@Override

public boolean equals(Object o) {

if (this == o)

return true;

if (o == null || getClass() != o.getClass())

return false;

UserDetailsImpl user = (UserDetailsImpl) o;

return Objects.equals(id, user.id);

}

}

Look at the code above, you can notice that we convert Set<Role> into List<GrantedAuthority>. It is important to work with Spring Security and Authentication object later.

As I have said before, we need UserDetailsService for getting UserDetails object. You can look at UserDetailsService interface that has only one method:

public interface UserDetailsService {

UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

}

So we implement it and override loadUserByUsername() method.

security/services/UserDetailsServiceImpl.java

package com.bezkoder.spring.security.login.security.services;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.bezkoder.spring.security.login.models.User;

import com.bezkoder.spring.security.login.repository.UserRepository;

@Service

public class UserDetailsServiceImpl implements UserDetailsService {

@Autowired

UserRepository userRepository;

@Override

@Transactional

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

User user = userRepository.findByUsername(username)

.orElseThrow(() -> new UsernameNotFoundException("User Not Found with username: " + username));

return UserDetailsImpl.build(user);

}

}

In the code above, we get full custom User object using UserRepository, then we build a UserDetails object using static build() method.

## Filter the Requests

Let’s define a filter that executes once per request. So we create AuthTokenFilter class that extends OncePerRequestFilter and override doFilterInternal() method.

security/jwt/AuthTokenFilter.java

package com.bezkoder.spring.security.login.security.jwt;

import java.io.IOException;

import jakarta.servlet.FilterChain;

import jakarta.servlet.ServletException;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

import org.springframework.web.filter.OncePerRequestFilter;

import com.bezkoder.spring.security.login.security.services.UserDetailsServiceImpl;

public class AuthTokenFilter extends OncePerRequestFilter {

@Autowired

private JwtUtils jwtUtils;

@Autowired

private UserDetailsServiceImpl userDetailsService;

private static final Logger logger = LoggerFactory.getLogger(AuthTokenFilter.class);

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

throws ServletException, IOException {

try {

String jwt = parseJwt(request);

if (jwt != null && jwtUtils.validateJwtToken(jwt)) {

String username = jwtUtils.getUserNameFromJwtToken(jwt);

UserDetails userDetails = userDetailsService.loadUserByUsername(username);

UsernamePasswordAuthenticationToken authentication =

new UsernamePasswordAuthenticationToken(userDetails,

null,

userDetails.getAuthorities());

authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authentication);

}

} catch (Exception e) {

logger.error("Cannot set user authentication: {}", e);

}

filterChain.doFilter(request, response);

}

private String parseJwt(HttpServletRequest request) {

String jwt = jwtUtils.getJwtFromCookies(request);

return jwt;

}

}

What we do inside doFilterInternal():  
– get JWT from the HTTP Cookies  
– if the request has JWT, validate it, parse username from it  
– from username, get UserDetails to create an Authentication object  
– set the current UserDetails in [SecurityContext](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#securitycontextholder-securitycontext-and-authentication-objects) using setAuthentication(authentication) method.

After this, everytime you want to get UserDetails, just use SecurityContext like this:

UserDetails userDetails =

(UserDetails) SecurityContextHolder.getContext().getAuthentication().getPrincipal();

// userDetails.getUsername()

// userDetails.getPassword()

// userDetails.getAuthorities()

## Create JWT Utility class

This class has 3 main funtions:

* getJwtFromCookies: get JWT from Cookies by Cookie name
* generateJwtCookie: generate a Cookie containing JWT from username, date, expiration, secret
* getCleanJwtCookie: return Cookie with null value (used for clean Cookie)
* getUserNameFromJwtToken: get username from JWT
* validateJwtToken: validate a JWT with a secret

security/jwt/JwtUtils.java

package com.bezkoder.spring.security.login.security.jwt;

import java.security.Key;

import java.util.Date;

import jakarta.servlet.http.Cookie;

import jakarta.servlet.http.HttpServletRequest;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.http.ResponseCookie;

import org.springframework.stereotype.Component;

import org.springframework.web.util.WebUtils;

import com.bezkoder.spring.security.login.security.services.UserDetailsImpl;

import io.jsonwebtoken.\*;

import io.jsonwebtoken.io.Decoders;

import io.jsonwebtoken.security.Keys;

@Component

public class JwtUtils {

private static final Logger logger = LoggerFactory.getLogger(JwtUtils.class);

@Value("${bezkoder.app.jwtSecret}")

private String jwtSecret;

@Value("${bezkoder.app.jwtExpirationMs}")

private int jwtExpirationMs;

@Value("${bezkoder.app.jwtCookieName}")

private String jwtCookie;

public String getJwtFromCookies(HttpServletRequest request) {

Cookie cookie = WebUtils.getCookie(request, jwtCookie);

if (cookie != null) {

return cookie.getValue();

} else {

return null;

}

}

public ResponseCookie generateJwtCookie(UserDetailsImpl userPrincipal) {

String jwt = generateTokenFromUsername(userPrincipal.getUsername());

ResponseCookie cookie = ResponseCookie.from(jwtCookie, jwt).path("/api").maxAge(24 \* 60 \* 60).httpOnly(true).build();

return cookie;

}

public ResponseCookie getCleanJwtCookie() {

ResponseCookie cookie = ResponseCookie.from(jwtCookie, null).path("/api").build();

return cookie;

}

public String getUserNameFromJwtToken(String token) {

return Jwts.parserBuilder().setSigningKey(key()).build()

.parseClaimsJws(token).getBody().getSubject();

}

private Key key() {

return Keys.hmacShaKeyFor(Decoders.BASE64.decode(jwtSecret));

}

public boolean validateJwtToken(String authToken) {

try {

Jwts.parserBuilder().setSigningKey(key()).build().parse(authToken);

return true;

} catch (MalformedJwtException e) {

logger.error("Invalid JWT token: {}", e.getMessage());

} catch (ExpiredJwtException e) {

logger.error("JWT token is expired: {}", e.getMessage());

} catch (UnsupportedJwtException e) {

logger.error("JWT token is unsupported: {}", e.getMessage());

} catch (IllegalArgumentException e) {

logger.error("JWT claims string is empty: {}", e.getMessage());

}

return false;

}

public String generateTokenFromUsername(String username) {

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

.setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))

.signWith(key(), SignatureAlgorithm.HS256)

.compact();

}

}

Remember that we’ve added bezkoder.app.jwtSecret, bezkoder.app.jwtExpirationMs and bezkoder.app.jwtCookieName properties in application.properties file, and jwtSecret has **64** characters.

## Handle Authentication Exception

Now we create AuthEntryPointJwt class that implements AuthenticationEntryPoint interface. Then we override the commence() method. This method will be triggerd anytime unauthenticated User requests a secured HTTP resource and an AuthenticationException is thrown.

security/jwt/AuthEntryPointJwt.java

package com.bezkoder.spring.security.login.security.jwt;

import java.io.IOException;

import java.util.HashMap;

import java.util.Map;

import jakarta.servlet.ServletException;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.http.MediaType;

import org.springframework.security.core.AuthenticationException;

import org.springframework.security.web.AuthenticationEntryPoint;

import org.springframework.stereotype.Component;

import com.fasterxml.jackson.databind.ObjectMapper;

@Component

public class AuthEntryPointJwt implements AuthenticationEntryPoint {

private static final Logger logger = LoggerFactory.getLogger(AuthEntryPointJwt.class);

@Override

public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)

throws IOException, ServletException {

logger.error("Unauthorized error: {}", authException.getMessage());

response.sendError(HttpServletResponse.SC\_UNAUTHORIZED, "Error: Unauthorized");

}

}

HttpServletResponse.SC\_UNAUTHORIZED is the **401** Status code. It indicates that the request requires HTTP authentication.

If you want to customize the response data, just use an ObjectMapper like following code:

@Override

public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)

throws IOException, ServletException {

logger.error("Unauthorized error: {}", authException.getMessage());

response.setContentType(MediaType.APPLICATION\_JSON\_VALUE);

response.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);

final Map<String, Object> body = new HashMap<>();

body.put("status", HttpServletResponse.SC\_UNAUTHORIZED);

body.put("error", "Unauthorized");

body.put("message", authException.getMessage());

body.put("path", request.getServletPath());

final ObjectMapper mapper = new ObjectMapper();

mapper.writeValue(response.getOutputStream(), body);

}

We’ve already built all things for Spring Security. The next sections of this tutorial will show you how to implement Controllers for our Rest APIs.

## Define payloads for Authentication Controller

Let me summarize the payloads for our RestAPIs:  
– Requests:

* LoginRequest: { username, password }
* SignupRequest: { username, email, password }

– Responses:

* UserInfoResponse: { id, username, email, roles }
* MessageResponse: { message }

To keep the tutorial not so long, I don’t show these POJOs here.  
You can find details for payload classes in source code of the project on [Github](https://github.com/bezkoder/spring-boot-security-login).

## Create Spring Rest Controllers

**Controller for Authentication**

This controller provides APIs for register and login, logout actions.

– /api/auth/signup

* check existing username/email
* create new User (with ROLE\_USER if not specifying role)
* save User to database using UserRepository

– /api/auth/signin

* authenticate { username, pasword }
* update SecurityContext using Authentication object
* generate JWT
* get UserDetails from Authentication object
* response contains JWT and UserDetails data

– /api/auth/signout: clear the Cookie.

controllers/AuthController.java

package com.bezkoder.spring.security.login.controllers;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

import java.util.stream.Collectors;

import jakarta.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpHeaders;

import org.springframework.http.ResponseCookie;

import org.springframework.http.ResponseEntity;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.web.bind.annotation.CrossOrigin;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import com.bezkoder.spring.security.login.models.ERole;

import com.bezkoder.spring.security.login.models.Role;

import com.bezkoder.spring.security.login.models.User;

import com.bezkoder.spring.security.login.payload.request.LoginRequest;

import com.bezkoder.spring.security.login.payload.request.SignupRequest;

import com.bezkoder.spring.security.login.payload.response.UserInfoResponse;

import com.bezkoder.spring.security.login.payload.response.MessageResponse;

import com.bezkoder.spring.security.login.repository.RoleRepository;

import com.bezkoder.spring.security.login.repository.UserRepository;

import com.bezkoder.spring.security.login.security.jwt.JwtUtils;

import com.bezkoder.spring.security.login.security.services.UserDetailsImpl;

@CrossOrigin(origins = "\*", maxAge = 3600)

@RestController

@RequestMapping("/api/auth")

public class AuthController {

@Autowired

AuthenticationManager authenticationManager;

@Autowired

UserRepository userRepository;

@Autowired

RoleRepository roleRepository;

@Autowired

PasswordEncoder encoder;

@Autowired

JwtUtils jwtUtils;

@PostMapping("/signin")

public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginRequest loginRequest) {

Authentication authentication = authenticationManager

.authenticate(new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword()));

SecurityContextHolder.getContext().setAuthentication(authentication);

UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();

ResponseCookie jwtCookie = jwtUtils.generateJwtCookie(userDetails);

List<String> roles = userDetails.getAuthorities().stream()

.map(item -> item.getAuthority())

.collect(Collectors.toList());

return ResponseEntity.ok().header(HttpHeaders.SET\_COOKIE, jwtCookie.toString())

.body(new UserInfoResponse(userDetails.getId(),

userDetails.getUsername(),

userDetails.getEmail(),

roles));

}

@PostMapping("/signup")

public ResponseEntity<?> registerUser(@Valid @RequestBody SignupRequest signUpRequest) {

if (userRepository.existsByUsername(signUpRequest.getUsername())) {

return ResponseEntity.badRequest().body(new MessageResponse("Error: Username is already taken!"));

}

if (userRepository.existsByEmail(signUpRequest.getEmail())) {

return ResponseEntity.badRequest().body(new MessageResponse("Error: Email is already in use!"));

}

// Create new user's account

User user = new User(signUpRequest.getUsername(),

signUpRequest.getEmail(),

encoder.encode(signUpRequest.getPassword()));

Set<String> strRoles = signUpRequest.getRole();

Set<Role> roles = new HashSet<>();

if (strRoles == null) {

Role userRole = roleRepository.findByName(ERole.ROLE\_USER)

.orElseThrow(() -> new RuntimeException("Error: Role is not found."));

roles.add(userRole);

} else {

strRoles.forEach(role -> {

switch (role) {

case "admin":

Role adminRole = roleRepository.findByName(ERole.ROLE\_ADMIN)

.orElseThrow(() -> new RuntimeException("Error: Role is not found."));

roles.add(adminRole);

break;

case "mod":

Role modRole = roleRepository.findByName(ERole.ROLE\_MODERATOR)

.orElseThrow(() -> new RuntimeException("Error: Role is not found."));

roles.add(modRole);

break;

default:

Role userRole = roleRepository.findByName(ERole.ROLE\_USER)

.orElseThrow(() -> new RuntimeException("Error: Role is not found."));

roles.add(userRole);

}

});

}

user.setRoles(roles);

userRepository.save(user);

return ResponseEntity.ok(new MessageResponse("User registered successfully!"));

}

@PostMapping("/signout")

public ResponseEntity<?> logoutUser() {

ResponseCookie cookie = jwtUtils.getCleanJwtCookie();

return ResponseEntity.ok().header(HttpHeaders.SET\_COOKIE, cookie.toString())

.body(new MessageResponse("You've been signed out!"));

}

}

**Controller for testing Authorization**

There are 4 APIs:  
– /api/test/all for public access  
– /api/test/user for users has ROLE\_USER or ROLE\_MODERATOR or ROLE\_ADMIN  
– /api/test/mod for users has ROLE\_MODERATOR  
– /api/test/admin for users has ROLE\_ADMIN

Do you remember that we used @EnableGlobalMethodSecurity(prePostEnabled = true) (or @EnableMethodSecurity for Spring Boot 3) in WebSecurityConfig class?

@Configuration

// @EnableWebSecurity

// @EnableGlobalMethodSecurity(prePostEnabled = true)

@EnableMethodSecurity // Spring Boot 3

public class WebSecurityConfig extends WebSecurityConfigurerAdapter { ... }

Now we can secure methods in our Apis with @PreAuthorize annotation easily.

controllers/TestController.java

package com.bezkoder.spring.security.login.controllers;

import org.springframework.security.access.prepost.PreAuthorize;

import org.springframework.web.bind.annotation.CrossOrigin;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@CrossOrigin(origins = "\*", maxAge = 3600)

@RestController

@RequestMapping("/api/test")

public class TestController {

@GetMapping("/all")

public String allAccess() {

return "Public Content.";

}

@GetMapping("/user")

@PreAuthorize("hasRole('USER') or hasRole('MODERATOR') or hasRole('ADMIN')")

public String userAccess() {

return "User Content.";

}

@GetMapping("/mod")

@PreAuthorize("hasRole('MODERATOR')")

public String moderatorAccess() {

return "Moderator Board.";

}

@GetMapping("/admin")

@PreAuthorize("hasRole('ADMIN')")

public String adminAccess() {

return "Admin Board.";

}

}

## Run & Check

Run Spring Boot Security JWT application with command: mvn spring-boot:run

Let’s check H2 database connection with url: http://localhost:8080/h2-ui:

Click on **Connect** button, tables that we define in models package will be automatically generated in Database.

We also need to add some rows into **roles** table before assigning any role to User.  
Run following SQL insert statements:

INSERT INTO roles(name) VALUES('ROLE\_USER');

INSERT INTO roles(name) VALUES('ROLE\_MODERATOR');

INSERT INTO roles(name) VALUES('ROLE\_ADMIN');

Then check the **roles** table:

Register some users with /signup API:

* **admin** with ROLE\_ADMIN
* **mod** with ROLE\_MODERATOR and ROLE\_USER
* **zkoder** with ROLE\_USER

Check **users** and **user\_roles** tables:

**Access public resource:** GET /api/test/all

**Access protected resource without Login:** GET /api/test/user

**Login an account:** POST /api/auth/signin

Check the Cookies:

**Access ROLE\_USER and ROLE\_MODERATOR resource:  
–**GET /api/test/user  
– GET /api/test/mod

**Access ROLE\_ADMIN resource:** GET /api/test/admin, reponse will be **403 Forbidden**:

**Logout the Account:**POST /api/auth/signout

## Solve Problem: javax.validation cannot be resolved

For Spring Boot **2.3** and later, you can see the compile error:  
The import javax.validation cannot be resolved

It is because Validation Starter no longer included in web starters. So you need to add the starter yourself.  
– For Maven:

<dependency>

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

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

– For Gradle:

dependencies {

...

implementation 'org.springframework.boot:spring-boot-starter-validation'

}

## Solve Problem with JDK 14

If you run this Spring Boot App with JDK 14 and get following error when trying to authenticate:

FilterChain java.lang.NoClassDefFoundError: javax/xml/bind/DatatypeConverter

Just add following dependency to pom.xml:

<dependency>

<groupId>jakarta.xml.bind</groupId>

<artifactId>jakarta.xml.bind-api</artifactId>

<version>2.3.2</version>

</dependency>

Everything’s gonna work fine.

## Conclusion

Today we’ve learned so many interesting things about Spring Boot Security example with JWT and H2 database using HttpOnly Cookie.

For understanding the architecture deeply and grasp the overview more easier:  
[Spring Boot Architecture for JWT with Spring Security](https://bezkoder.com/spring-boot-jwt-mysql-spring-security-architecture/)

You should continue to know how to implement Refresh Token:  
[Spring Boot Refresh Token with JWT example](https://www.bezkoder.com/spring-security-refresh-token/)

Validate the signup request (password, confirm password):  
[Spring Boot custom Validation example](https://www.bezkoder.com/spring-boot-custom-validation/)

You can also know how to deploy Spring Boot App on AWS (for free) with [this tutorial](https://bezkoder.com/deploy-spring-boot-aws-eb/).

Happy learning! See you again.

# Core Configuration

## Spring Boot 2.x Sample

Spring Boot 2.x brings full auto-configuration capabilities for OAuth 2.0 Login.

This section shows how to configure the [**OAuth 2.0 Login sample**](https://github.com/spring-projects/spring-security-samples/tree/main/servlet/spring-boot/java/oauth2/login) by using Google as the Authentication Provider and covers the following topics:

* [Initial Setup](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-sample-initial-setup)
* [Setting the Redirect URI](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-sample-redirect-uri)
* [Configure application.yml](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-sample-application-config)
* [Boot up the Application](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-sample-boot-application)

### Initial Setup

To use Google’s OAuth 2.0 authentication system for login, you must set up a project in the Google API Console to obtain OAuth 2.0 credentials.

|  |  |
| --- | --- |
|  | [Google’s OAuth 2.0 implementation](https://developers.google.com/identity/protocols/OpenIDConnect) for authentication conforms to the [OpenID Connect 1.0](https://openid.net/connect/) specification and is [OpenID certified](https://openid.net/certification/). |

Follow the instructions on the [OpenID Connect](https://developers.google.com/identity/protocols/OpenIDConnect) page, starting in the “Setting up OAuth 2.0” section.

After completing the “Obtain OAuth 2.0 credentials” instructions, you should have new OAuth Client with credentials consisting of a Client ID and a Client Secret.

### Setting the Redirect URI

The redirect URI is the path in the application that the end-user’s user-agent is redirected back to after they have authenticated with Google and have granted access to the OAuth Client ([created in the previous step](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-sample-initial-setup)) on the Consent page.

In the “Set a redirect URI” subsection, ensure that the **Authorized redirect URIs** field is set to [localhost:8080/login/oauth2/code/google](http://localhost:8080/login/oauth2/code/google).

|  |  |
| --- | --- |
|  | The default redirect URI template is {baseUrl}/login/oauth2/code/{registrationId}. The registrationId is a unique identifier for the [ClientRegistration](https://docs.spring.io/spring-security/reference/servlet/oauth2/client/index.html" \l "oauth2Client-client-registration). |
|  | If the OAuth Client runs behind a proxy server, you should check the [Proxy Server Configuration](https://docs.spring.io/spring-security/reference/features/exploits/http.html#http-proxy-server) to ensure the application is correctly configured. Also, see the supported [URI template variables](https://docs.spring.io/spring-security/reference/servlet/oauth2/client/authorization-grants.html#oauth2Client-auth-code-redirect-uri) for redirect-uri. |

### Configure application.yml

Now that you have a new OAuth Client with Google, you need to configure the application to use the OAuth Client for the authentication flow. To do so:

1. Go to application.yml and set the following configuration:
2. spring:
3. security:
4. oauth2:
5. client:
6. registration:
7. google:
8. client-id: google-client-id

client-secret: google-client-secret

Copied!

OAuth Client properties

|  |  |
| --- | --- |
|  | spring.security.oauth2.client.registration is the base property prefix for OAuth Client properties. |
|  | Following the base property prefix is the ID for the [ClientRegistration](https://docs.spring.io/spring-security/reference/servlet/oauth2/client/index.html" \l "oauth2Client-client-registration), such as Google. |

1. Replace the values in the client-id and client-secret property with the OAuth 2.0 credentials you created earlier.

### Boot up the Application

Launch the Spring Boot 2.x sample and go to [localhost:8080](http://localhost:8080/). You are then redirected to the default auto-generated login page, which displays a link for Google.

Click on the Google link, and you are then redirected to Google for authentication.

After authenticating with your Google account credentials, you see the Consent screen. The Consent screen asks you to either allow or deny access to the OAuth Client you created earlier. Click **Allow** to authorize the OAuth Client to access your email address and basic profile information.

At this point, the OAuth Client retrieves your email address and basic profile information from the [UserInfo Endpoint](https://openid.net/specs/openid-connect-core-1_0.html" \l "UserInfo) and establishes an authenticated session.

## Spring Boot 2.x Property Mappings

The following table outlines the mapping of the Spring Boot 2.x OAuth Client properties to the [ClientRegistration](https://docs.spring.io/spring-security/reference/servlet/oauth2/client/index.html" \l "oauth2Client-client-registration) properties.

| **Spring Boot 2.x** | **ClientRegistration** |
| --- | --- |
| spring.security.oauth2.client.registration.[registrationId] | registrationId |
| spring.security.oauth2.client.registration.[registrationId].client-id | clientId |
| spring.security.oauth2.client.registration.[registrationId].client-secret | clientSecret |
| spring.security.oauth2.client.registration.[registrationId].client-authentication-method | clientAuthenticationMethod |
| spring.security.oauth2.client.registration.[registrationId].authorization-grant-type | authorizationGrantType |
| spring.security.oauth2.client.registration.[registrationId].redirect-uri | redirectUri |
| spring.security.oauth2.client.registration.[registrationId].scope | scopes |
| spring.security.oauth2.client.registration.[registrationId].client-name | clientName |
| spring.security.oauth2.client.provider.[providerId].authorization-uri | providerDetails.authorizationUri |
| spring.security.oauth2.client.provider.[providerId].token-uri | providerDetails.tokenUri |
| spring.security.oauth2.client.provider.[providerId].jwk-set-uri | providerDetails.jwkSetUri |
| spring.security.oauth2.client.provider.[providerId].issuer-uri | providerDetails.issuerUri |
| spring.security.oauth2.client.provider.[providerId].user-info-uri | providerDetails.userInfoEndpoint.uri |
| spring.security.oauth2.client.provider.[providerId].user-info-authentication-method | providerDetails.userInfoEndpoint.authenticationMethod |
| spring.security.oauth2.client.provider.[providerId].user-name-attribute | providerDetails.userInfoEndpoint.userNameAttributeName |

|  |  |
| --- | --- |
|  | You can initially configure a ClientRegistration by using discovery of an OpenID Connect Provider’s [Configuration endpoint](https://openid.net/specs/openid-connect-discovery-1_0.html#ProviderConfig) or an Authorization Server’s [Metadata endpoint](https://tools.ietf.org/html/rfc8414#section-3), by specifying the spring.security.oauth2.client.provider.[providerId].issuer-uri property. |

## CommonOAuth2Provider

CommonOAuth2Provider pre-defines a set of default client properties for a number of well known providers: Google, GitHub, Facebook, and Okta.

For example, the authorization-uri, token-uri, and user-info-uri do not change often for a provider. Therefore, it makes sense to provide default values, to reduce the required configuration.

As demonstrated previously, when we [configured a Google client](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-sample-application-config), only the client-id and client-secret properties are required.

The following listing shows an example:

spring:

security:

oauth2:

client:

registration:

google:

client-id: google-client-id

client-secret: google-client-secret

Copied!

|  |  |
| --- | --- |
|  | The auto-defaulting of client properties works seamlessly here because the registrationId (google) matches the GOOGLE enum (case-insensitive) in CommonOAuth2Provider. |

For cases where you may want to specify a different registrationId, such as google-login, you can still leverage auto-defaulting of client properties by configuring the provider property.

The following listing shows an example:

spring:

security:

oauth2:

client:

registration:

google-login:

provider: google

client-id: google-client-id

client-secret: google-client-secret

Copied!

|  |  |
| --- | --- |
|  | The registrationId is set to google-login. |
|  | The provider property is set to google, which will leverage the auto-defaulting of client properties set in CommonOAuth2Provider.GOOGLE.getBuilder(). |

## Configuring Custom Provider Properties

There are some OAuth 2.0 Providers that support multi-tenancy, which results in different protocol endpoints for each tenant (or sub-domain).

For example, an OAuth Client registered with Okta is assigned to a specific sub-domain and have their own protocol endpoints.

For these cases, Spring Boot 2.x provides the following base property for configuring custom provider properties: spring.security.oauth2.client.provider.[providerId].

The following listing shows an example:

spring:

security:

oauth2:

client:

registration:

okta:

client-id: okta-client-id

client-secret: okta-client-secret

provider:

okta:

authorization-uri: https://your-subdomain.oktapreview.com/oauth2/v1/authorize

token-uri: https://your-subdomain.oktapreview.com/oauth2/v1/token

user-info-uri: https://your-subdomain.oktapreview.com/oauth2/v1/userinfo

user-name-attribute: sub

jwk-set-uri: https://your-subdomain.oktapreview.com/oauth2/v1/keys

Copied!

|  |  |
| --- | --- |
|  | The base property (spring.security.oauth2.client.provider.okta) allows for custom configuration of protocol endpoint locations. |

## Overriding Spring Boot 2.x Auto-configuration

The Spring Boot 2.x auto-configuration class for OAuth Client support is OAuth2ClientAutoConfiguration.

It performs the following tasks:

* Registers a ClientRegistrationRepository @Bean composed of ClientRegistration(s) from the configured OAuth Client properties.
* Registers a SecurityFilterChain @Bean and enables OAuth 2.0 Login through httpSecurity.oauth2Login().

If you need to override the auto-configuration based on your specific requirements, you may do so in the following ways:

* [Register a ClientRegistrationRepository @Bean](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-register-clientregistrationrepository-bean)
* [Register a SecurityFilterChain @Bean](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-provide-securityfilterchain-bean)
* [Completely Override the Auto-configuration](https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-completely-override-autoconfiguration)

### Register a ClientRegistrationRepository @Bean

The following example shows how to register a ClientRegistrationRepository @Bean:

* **Java**
* **Kotlin**

@Configuration

public class OAuth2LoginConfig {

@Bean

public ClientRegistrationRepository clientRegistrationRepository() {

return new InMemoryClientRegistrationRepository(this.googleClientRegistration());

}

private ClientRegistration googleClientRegistration() {

return ClientRegistration.withRegistrationId("google")

.clientId("google-client-id")

.clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT\_SECRET\_BASIC)

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION\_CODE)

.redirectUri("{baseUrl}/login/oauth2/code/{registrationId}")

.scope("openid", "profile", "email", "address", "phone")

.authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")

.tokenUri("https://www.googleapis.com/oauth2/v4/token")

.userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")

.userNameAttributeName(IdTokenClaimNames.SUB)

.jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")

.clientName("Google")

.build();

}

}

Copied!

### Register a SecurityFilterChain @Bean

The following example shows how to register a SecurityFilterChain @Bean with @EnableWebSecurity and enable OAuth 2.0 login through httpSecurity.oauth2Login():

*OAuth2 Login Configuration*

* **Java**
* **Kotlin**

@Configuration

@EnableWebSecurity

public class OAuth2LoginSecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(authorize -> authorize

.anyRequest().authenticated()

)

.oauth2Login(withDefaults());

return http.build();

}

}

Copied!

### Completely Override the Auto-configuration

The following example shows how to completely override the auto-configuration by registering a ClientRegistrationRepository @Bean and a SecurityFilterChain @Bean.

*Overriding the auto-configuration*

* **Java**
* **Kotlin**

@Configuration

public class OAuth2LoginConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(authorize -> authorize

.anyRequest().authenticated()

)

.oauth2Login(withDefaults());

return http.build();

}

@Bean

public ClientRegistrationRepository clientRegistrationRepository() {

return new InMemoryClientRegistrationRepository(this.googleClientRegistration());

}

private ClientRegistration googleClientRegistration() {

return ClientRegistration.withRegistrationId("google")

.clientId("google-client-id")

.clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT\_SECRET\_BASIC)

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION\_CODE)

.redirectUri("{baseUrl}/login/oauth2/code/{registrationId}")

.scope("openid", "profile", "email", "address", "phone")

.authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")

.tokenUri("https://www.googleapis.com/oauth2/v4/token")

.userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")

.userNameAttributeName(IdTokenClaimNames.SUB)

.jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")

.clientName("Google")

.build();

}

}

Copied!

## Java Configuration without Spring Boot 2.x

If you are not able to use Spring Boot 2.x and would like to configure one of the pre-defined providers in CommonOAuth2Provider (for example, Google), apply the following configuration:

*OAuth2 Login Configuration*

* **Java**
* **Kotlin**
* **Xml**

@Configuration

@EnableWebSecurity

public class OAuth2LoginConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(authorize -> authorize

.anyRequest().authenticated()

)

.oauth2Login(withDefaults());

return http.build();

}

@Bean

public ClientRegistrationRepository clientRegistrationRepository() {

return new InMemoryClientRegistrationRepository(this.googleClientRegistration());

}

@Bean

public OAuth2AuthorizedClientService authorizedClientService(

ClientRegistrationRepository clientRegistrationRepository) {

return new InMemoryOAuth2AuthorizedClientService(clientRegistrationRepository);

}

@Bean

public OAuth2AuthorizedClientRepository authorizedClientRepository(

OAuth2AuthorizedClientService authorizedClientService) {

return new AuthenticatedPrincipalOAuth2AuthorizedClientRepository(authorizedClientService);

}

private ClientRegistration googleClientRegistration() {

return CommonOAuth2Provider.GOOGLE.getBuilder("google")

.clientId("google-client-id")

.clientSecret("google-client-secret")

.build();

}

}

Copied!