**Lesson 1:**

**What Spring Security can do**

User name/ password authentication

SSO/ Okta / LDAP

App level Authorization

Intra App Authrorization like OAuth

Microservice security (using tokens, JWT)

Method level security

**Lesson 2:**

**5 Core Concepts in Spring Security**

Authentication

Authorization

Principal (currently logged in user)

Granted Authority (group of permission allowed for user)

Roles (Group of authority)

**Lesson 3:**

**Adding Spring Security to new Spring Boot Project**

@RestController  
**public class** HomeResource {  
  
 @GetMapping(**"/"**)  
 **public** String home(){  
 **return** (**"<h1> Welcome </h1>"**);  
 }  
}

<http://localhost:8080/>

Add spring boot starter security dependency to integrate spring security.

<dependency>

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

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

</dependency>

After adding this dependency spring security will work.

Browser url: <http://localhost:8080/>

Sign in form will appear.

How this happens?

Filters

Spring Security default behavior

1. Adds mandatory authentication for urls
2. Adds login form
3. Handles login error
4. Creates a user and sets a default password (password is provided in console and default user name = user)

For changing username and password

Application.properties

**spring.security.user.name**=**foo  
spring.security.user.password**=**foo**

**Lesson 4:**

**How to configure Spring Security Authentication**

Authentication Manager manages authentication

authenticate()

return successful authentication or throw errors.

We don’t create our own authentication manger but we configuration authentication manger using builder pattern.

we don’t directly interact with Authentication Manger.

Steps:

1. Get hold of AuthenticationMangerBuilder.
2. Set the configuration on it.

Authentication Manager Builder ask?

What type of auth>

In memory auth, please.

Ok tell me the user name, pwd, and role.

After that new Authentication Manger is created.

Extend the class and override the method

Configure(AuthenticationMangerBuilder authenticationMangerBuilder)

Application.properties

Add comment

*#spring.security.user.name=foo  
#spring.security.user.password=foo*

@EnableWebSecurity  
**public class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {  
 @Override  
 **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.inMemoryAuthentication()  
 .withUser(**"blah"**).password(**"blah"**).roles(**"USER"**)  
 .and()  
 .withUser(**"foo"**).password(**"foo"**).roles(**"ADMIN"**);  
 }  
  
 @Bean  
 **public** PasswordEncoder getPasswordEncoder() {  
 **return** NoOpPasswordEncoder.*getInstance*();  
 }  
}

@EnableWebSecurity  
**public class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {  
 @Override  
 **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.inMemoryAuthentication()  
 .withUser(**"blah"**).password(**"blah"**).roles(**"USER"**)  
 .and()  
 .withUser(**"foo"**).password(**"foo"**).roles(**"ADMIN"**);  
 }  
  
 @Bean  
 **public** PasswordEncoder getPasswordEncoder() {  
 **return** NoOpPasswordEncoder.*getInstance*();  
 }  
}

**Lesson 5:**

**How to configure Spring Security Authorization**

**API Roles allows to access it**

/ All(unauthenticated)

/user USER AND ADMIN roles

/admin ADMIN role

@RestController  
**public class** HomeResource1 {  
  
 @GetMapping(**"/"**)  
 **public** String home1(){  
 **return** (**"<h1> Welcome </h1>"**);  
 }  
  
  
 @GetMapping(**"/user"**)  
 **public** String user(){  
 **return** (**"<h1> Welcome User </h1>"**);  
 }  
  
  
 @GetMapping(**"/admin"**)  
 **public** String admin(){  
 **return** (**"<h1> Welcome Admin</h1>"**);  
 }  
}

HttpSecurity

Allows to configure path and role

Extends WebSecurityConfigurereAdapter

Configure(HttpSecurity httpSecurity)

Htpp.authorizeRequest()

<path> => <allowed-roles>

<path> => <allowed-roles>

/\*\* => matches all path (\*\* => indicate all path in current. / => indicates below or nested)

.antMatchers(**"/\*\*"**).hasRole(**"USER"**) This path is accessible if user has USER role.

.antMatchers(**"/\*\*"**).hasAnyRole(**"USER"**, **"ADMIN"**) This path is accessible if user has USER role or admin role.

Can also be used to configure login type.

.and().formLogin(); => login type is form based.

Logout url: <http://localhost:8080/logout>

Add first more restrictive and then least restrictive while configuring role for path.

@EnableWebSecurity  
**public class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {  
 @Override  
 **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.inMemoryAuthentication()  
 .withUser(**"blah"**).password(**"blah"**).roles(**"USER"**)  
 .and()  
 .withUser(**"foo"**).password(**"foo"**).roles(**"ADMIN"**);  
 }  
  
 @Bean  
 **public** PasswordEncoder getPasswordEncoder() {  
 **return** NoOpPasswordEncoder.*getInstance*();  
 }  
  
 @Override  
 **protected void** configure(HttpSecurity http) **throws** Exception {  
 http.authorizeRequests()  
 .antMatchers(**"/admin"**).hasRole( **"ADMIN"**)  
 .antMatchers(**"/user"**).hasAnyRole(**"USER"**, **"ADMIN"**)  
 .antMatchers(**"/"**, **"static/css"**, **"static/js"**).permitAll()  
 .and().formLogin();  
 }  
}

If you put / at first it will match all url

**Lesson 6**

**How Spring Security Authentication works – Java Brains**

**Filters**

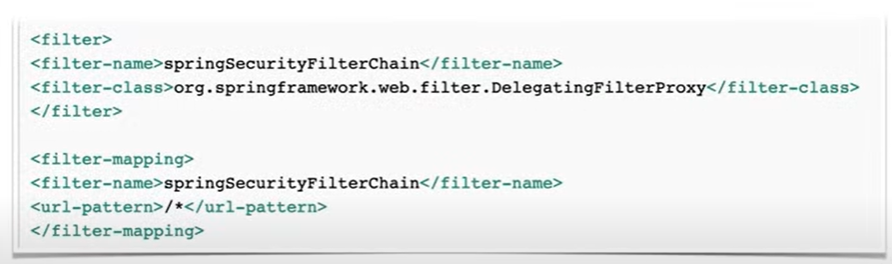
Filters can be applied to all or certain url.

Eg: applied filter for url starts with /admin

Spring Security filter

When we add spring dependency in spring application, it intercept all request /\* and maps to spring security filter DelegatingFilterProxy.

If you are not working with spring boot application, you need to manually add this filter.

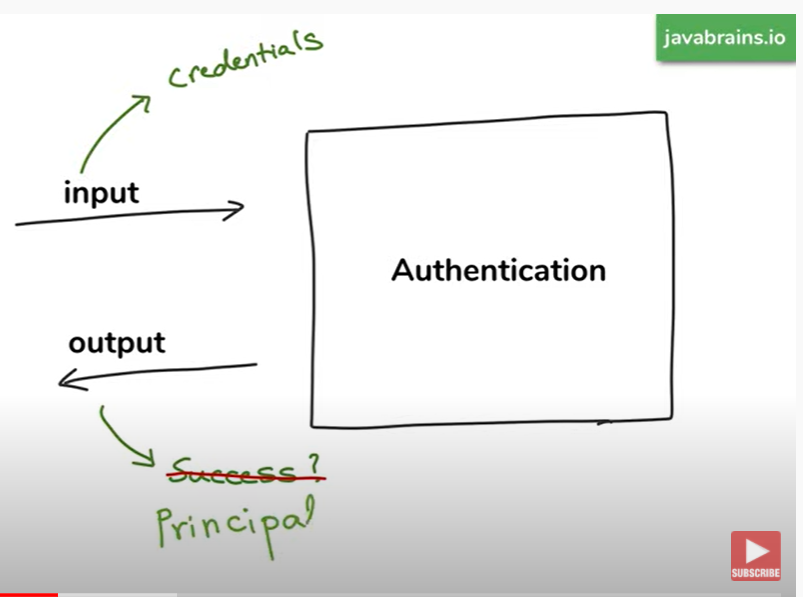


There are around 5 filter in spring security.

One of them is authenticationfilter which intercept all authentication request and initiate the authenticate process.

There is also authorization filter.

**Bootstrap authentication**

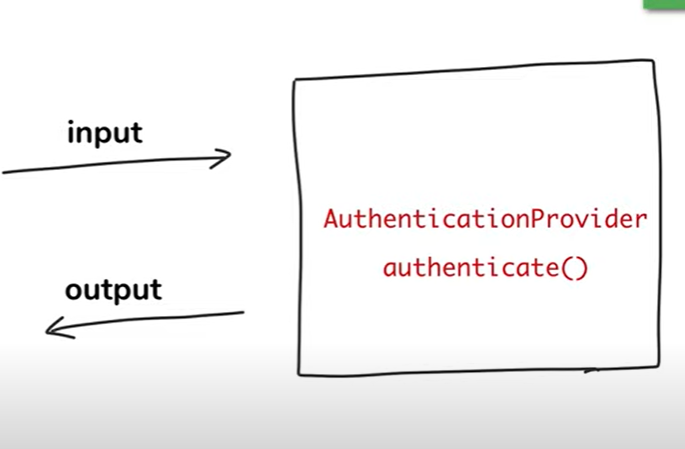


Authentication

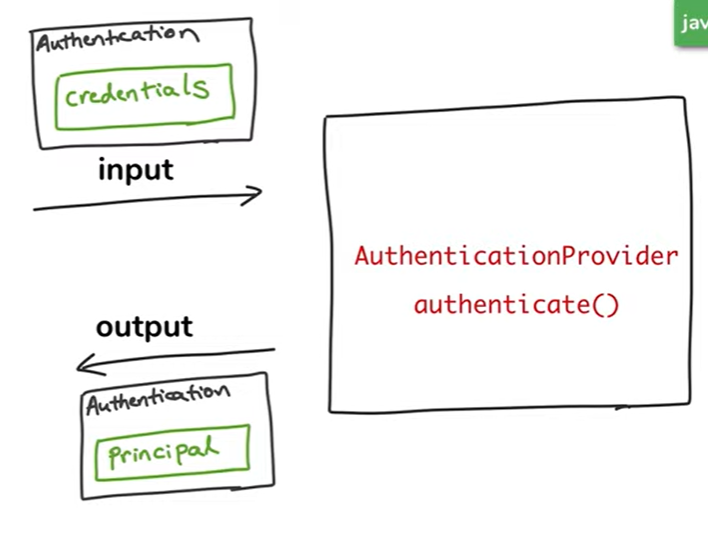
When spring security perform authentication it keeps tracks of both input and output using authentication object. Authentication is a spring security internal interface. It holds credential before authentication and once user is authenticate it holds principal.

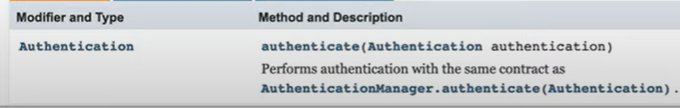
**What does Authentication?**

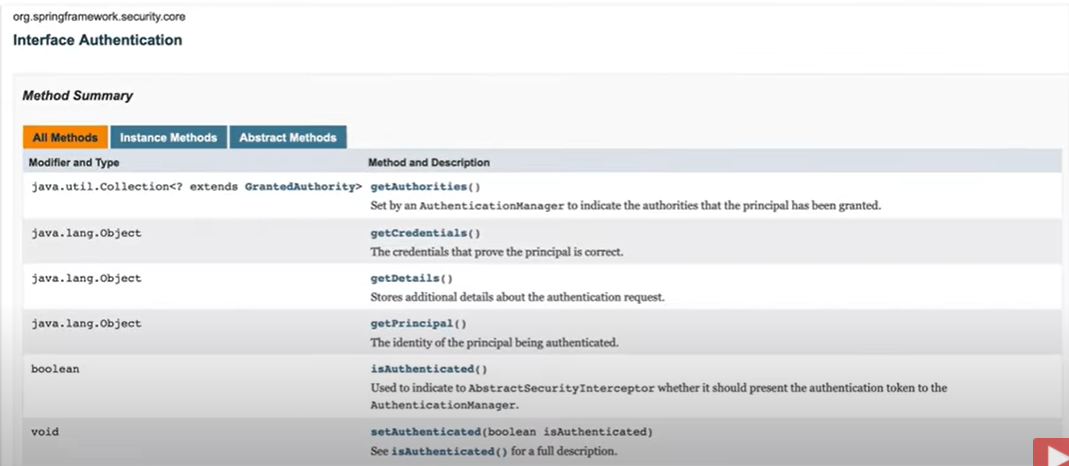
The most common pattern is **Provider**.



AuthenticationProvider does actual authentication. You need to implement authenticate() method in any application and tells spring security and spring security will call this method.







Before authentication: getCredentials() it has credential.

After authentication: getPrincipal() it has principal.

isAuthenticated() : is user is authenticate?

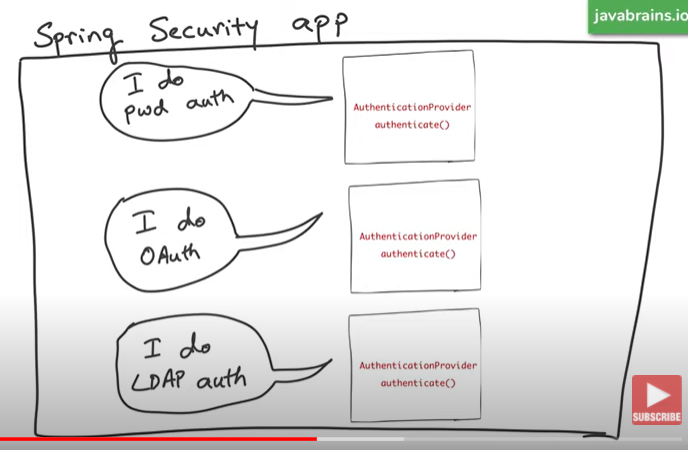
Typical App may have(multiple authentication)

username password authentication

oauth based authentication

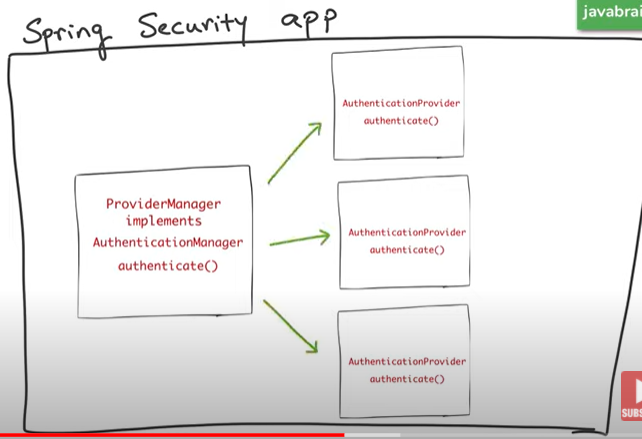
sso/ldap authentication

Hence Thera may be multiple authentication Provider.



How does these authentication provider coordinate with each other?

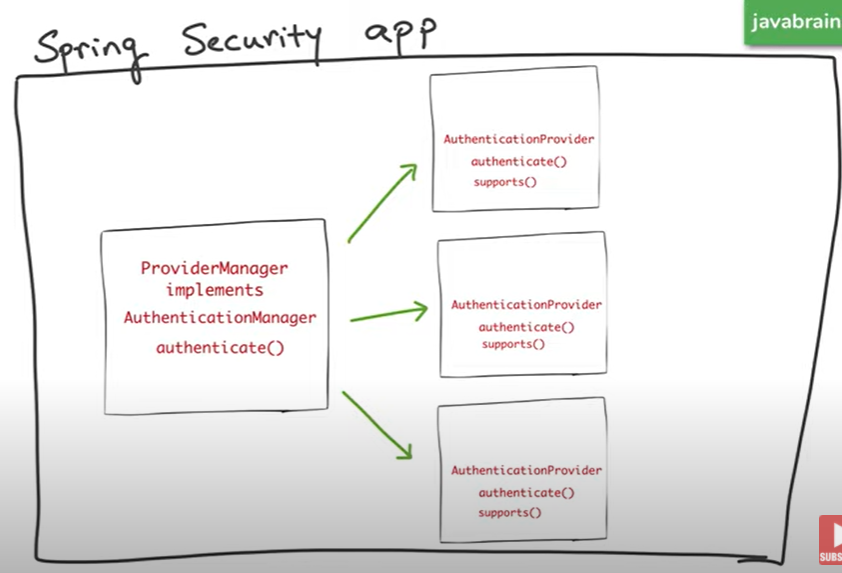
* AuthenticationManger

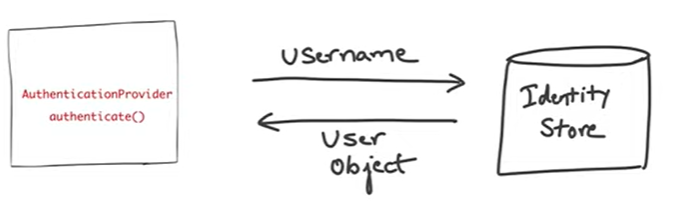


ProviderManager coordinate all authenticationprovider.

For ldap authentication it asks for all authentication provider. Do you support this authentication?

To this each authentication provider also has supports method.

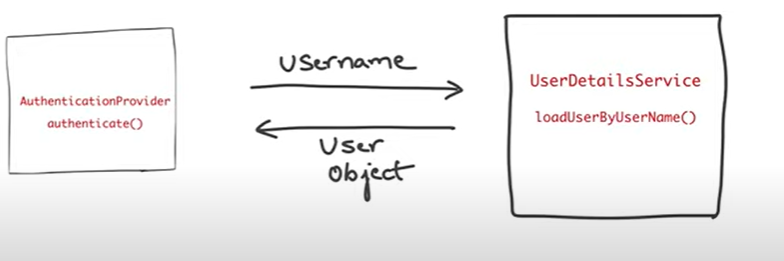




This is same for all authentication provider.

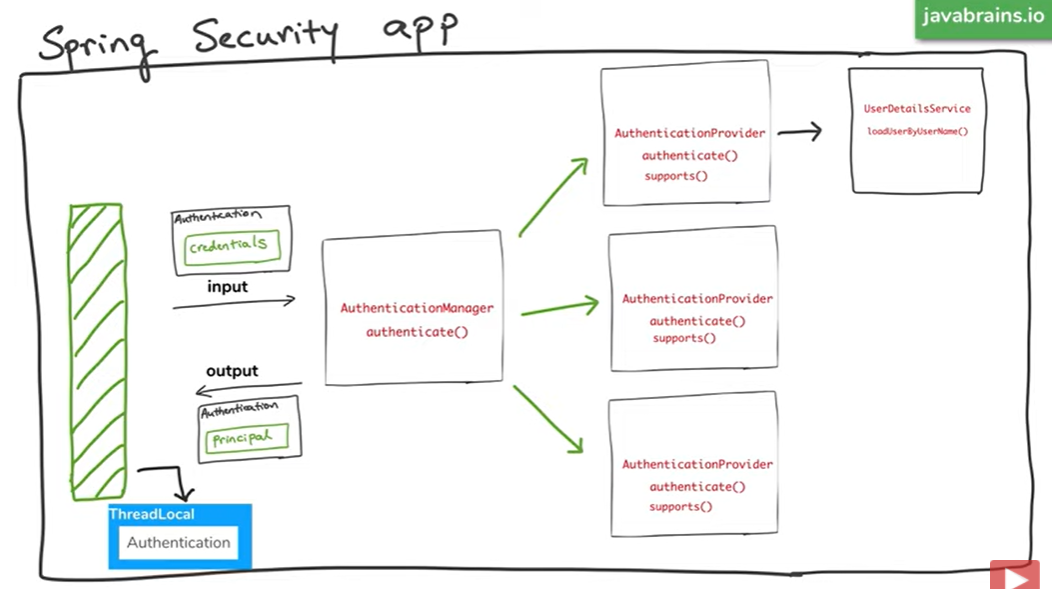
Givern username, all authentication provider should retrieve user info from database and then verify valid user, password not expire, account not lock.

Once user is retrieve from database using ldap or database check we make is same(verification). Spring security has extract that part out into userDetailService. UserDetailService take username and and return userDetails Object.



Either user is valid, password expire is in user detail object.

Once user detail object is return, authenticationprovider authenticate successful and mostly return same userDetails object.



If authentication fail, authentication exception.

Affter authentication filter get principal, it saves authentication in ThreadLocal ins SecurityContext.

It can aslo be saved in session that can be done by another filter. It gets authenticated principal and save in session.

**Lesson 7**

**How to setup JDBC authentication with Spring Security from scratch - Java Brains (Spring-Security-2 Project)**

@RestController  
**public class** HomeResource {  
 @GetMapping(**"/"**)  
 **public** String home() {  
 **return** (**"</h1>Welcome</h1>"**);  
 }  
  
 @GetMapping(**"/user"**)  
 **public** String user() {  
 **return** (**"</h1>Welcome User</h1>"**);  
 }  
  
 @GetMapping(**"/admin"**)  
 **public** String admin() {  
 **return** (**"</h1>Welcome Admin</h1>"**);  
 }  
  
  
}

@EnableWebSecurity  
**public class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {  
  
 @Autowired  
 DataSource **dataSource**;  
  
 @Override  
 **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.jdbcAuthentication()  
 .dataSource(**dataSource**);  
 }  
  
 @Override  
 **protected void** configure(HttpSecurity http) **throws** Exception {  
 http.authorizeRequests()  
 .antMatchers(**"/admin"**).hasRole( **"ADMIN"**)  
 .antMatchers(**"/user"**).hasAnyRole(**"USER"**, **"ADMIN"**)  
 .antMatchers(**"/"**, **"static/css"**, **"static/js"**).permitAll()  
 .and().formLogin();  
  
 }

@Bean  
**public** PasswordEncoder getPasswordEncoder() {  
 **return** NoOpPasswordEncoder.*getInstance*();  
}

}

When you add embedded database(H2) in spring application, spring boot automatically create datasource for you. In above example @Autowired Databasource will point to H2 database without any bean configuration.

The next step is to create schema for table. spring security can create default table for us.

@Override  
**protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.jdbcAuthentication()  
 .dataSource(**dataSource**)  
 .withDefaultSchema();  
}

We can also populate data to table.

@Override  
**protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.jdbcAuthentication()  
 .dataSource(**dataSource**)  
 .withDefaultSchema()  
 .withUser(User.*withUsername*(**"user"**).password(**"pass"**).roles(**"USER"**))  
 .withUser(User.*withUsername*(**"admin"**).password(**"pass"**).roles(**"ADMIN"**));  
  
}

In real world application we don’t have defautlschema with data populated. Uncomment the code like below.

@Override  
**protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.jdbcAuthentication()  
 .dataSource(**dataSource**);  
 */\*.withDefaultSchema()  
 .withUser(User.withUsername("user").password("pass").roles("USER"))  
 .withUser(User.withUsername("admin").password("pass").roles("ADMIN"));\*/*}

This is the ideal state.

Spring Security default Schema location

https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/appendix-schema.html

resources/schema.sql

**create table** users(  
 **username** varchar\_ignorecase(50) **not null primary key**,  
 **password** varchar\_ignorecase(50) **not null**,  
 **enabled** boolean **not null**);  
  
**create table** authorities (  
 **username** varchar\_ignorecase(50) **not null**,  
 **authority** varchar\_ignorecase(50) **not null**,  
 **constraint** fk\_authorities\_users **foreign key**(**username**) **references** users(**username**)  
);  
**create unique** index ix\_auth\_username **on** authorities (username,authority);

schema.sql is a file that spring boot use to set up the schema.

When spring boot run it run ddl statemtnt in schema.sql.

For populating data use data.sql

**INSERT INTO** users(username, password, enabled)  
**values** (**'user'**, **'pass'**, **true**);  
  
**INSERT INTO** users(username, password, enabled)  
**values** (**'admin'**, **'pass'**, **true**);  
  
**INSERT INTO** authorities(username, authority)  
**values** (**'user'**, **'ROLE\_USER'**);  
  
  
**INSERT INTO** authorities(username, authority)  
**values** (**'admin'**, **'ROLE\_ADMIN'**);

What if your database store data in different format?

@Override  
**protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.jdbcAuthentication()  
 .dataSource(**dataSource**)  
 .usersByUsernameQuery(**"select username,password,enabled from users where username=?"**)  
 .authoritiesByUsernameQuery(**"select username,authority from authorities where username=?"**);  
 */\*.withDefaultSchema()  
 .withUser(User.withUsername("user").password("pass").roles("USER"))  
 .withUser(User.withUsername("admin").password("pass").roles("ADMIN"));\*/*}

What if your datasource is different.

application.properties

**spring.datasource.url**=  
**spring.datasource.username**=  
**spring.datasource.password**=

**Lesson 10**

**What is JWT authorization really about**

Authorization strategies

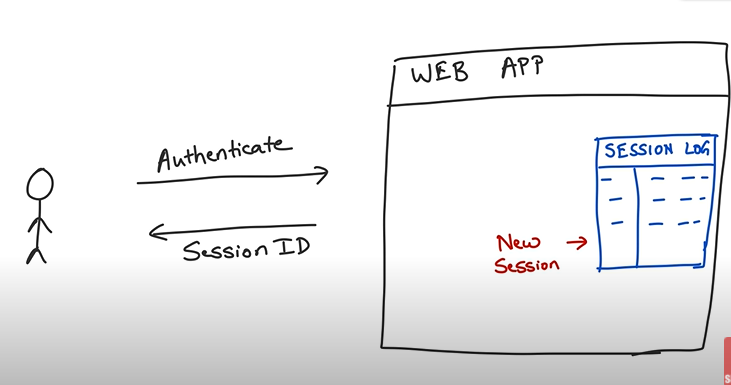
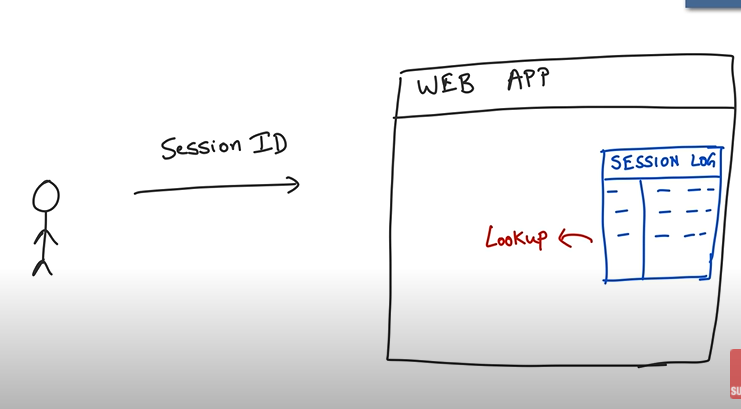
Session token

JSON web token

HTTP

Stateless protocol

**Session token**

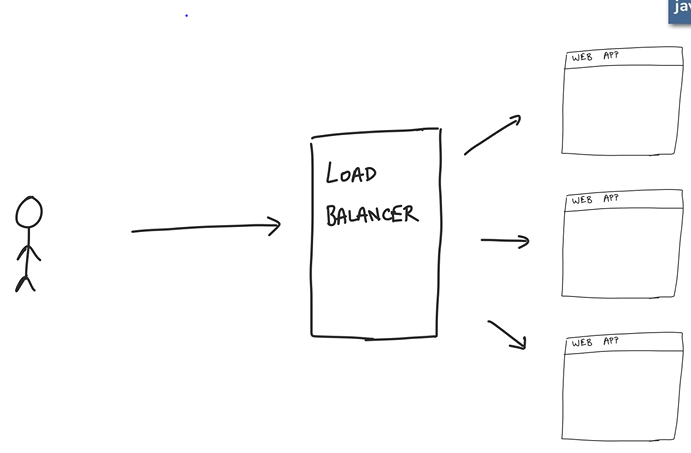
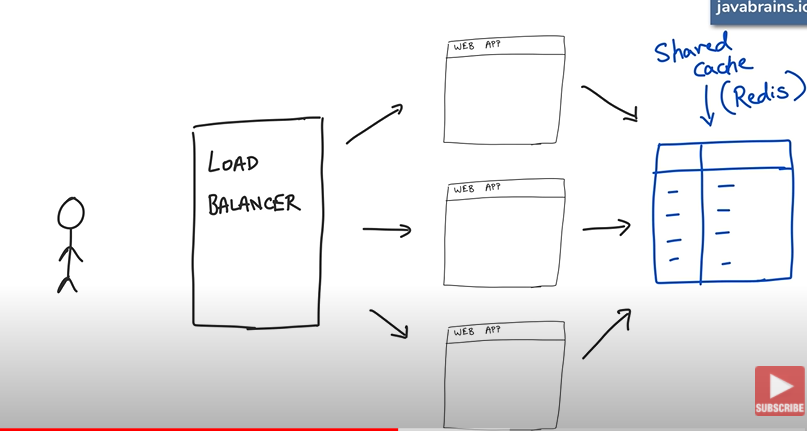
 

Session id is saved in cookie so that session is automatically added at request header upon each request.

Session ID + Cookies => Most popular mechanism for authorization

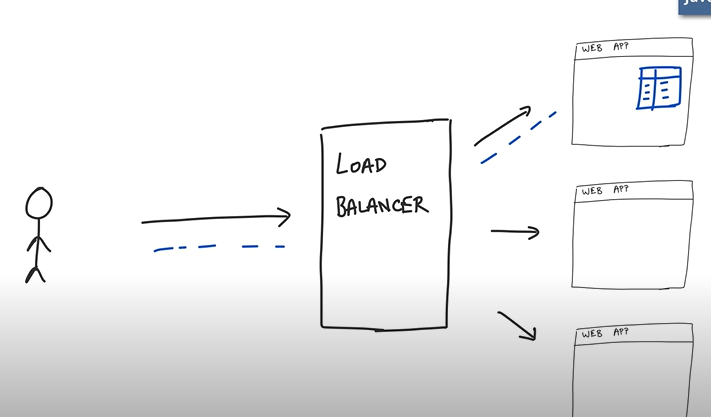
DrawBack:

It assumes there is only one monolothic server.

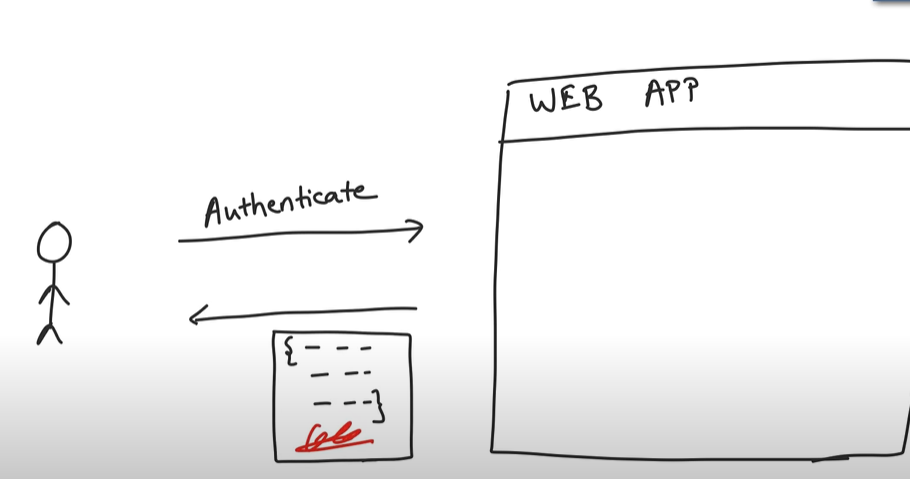
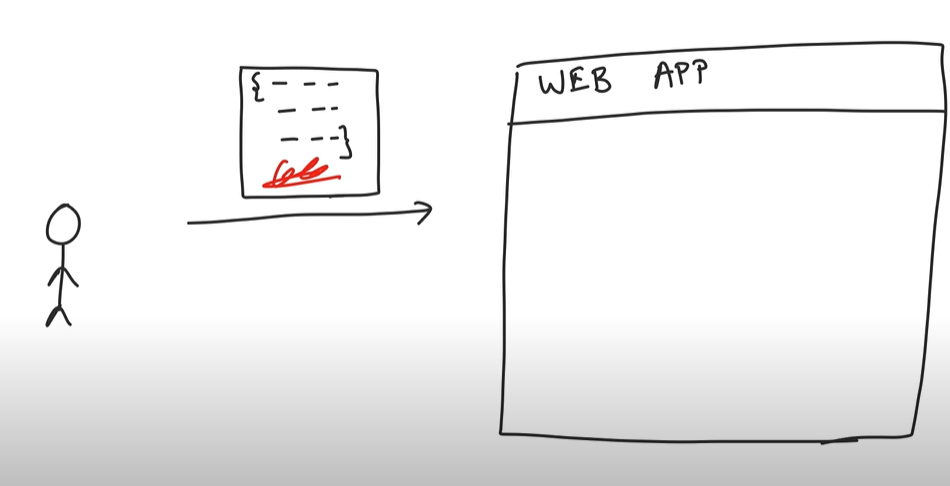
 

Single point of failure. If redis fail.

Instead it uses stick session. Load balancer will know which server has session



**JSON token**

JWT can also be send like session.

**LESSON 11**

**What is the structure of a JWT - Java Brains**

Three parts to a JWT

Header

Payload

Signature

Jwt.io

**Lesson 12**

**Spring Boot + Spring Security + JWT from scratch - Java Brains (Spring-Security-3 Project)**

@RestController  
**public class** HelloResource {  
  
 @RequestMapping(**"/hello"**)  
 **public** String hello() {  
 **return "Hello World"**;  
 }  
}

Overriding **default** user behaviour service()

@EnableWebSecurity  
**public class** SecurityConfigurer **extends** WebSecurityConfigurerAdapter {  
  
 @Autowired  
 **private** MyUserDetailsService **myUserDetailsService**;  
  
 @Override  
 **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.userDetailsService(**myUserDetailsService**);  
 }  
  
 @Bean  
 **public** PasswordEncoder passwordEncoder() {  
 **return** NoOpPasswordEncoder.*getInstance*();  
 }  
}

@Service  
**public class** MyUserDetailsService **implements** UserDetailsService {  
 @Override  
 **public** UserDetails loadUserByUsername(String userName) **throws** UsernameNotFoundException {  
 **return new** User(**"foo"**, **"foo"**, **new** ArrayList<>());  
 }  
}

Add dependency:

*<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt -->*<**dependency**>  
 <**groupId**>io.jsonwebtoken</**groupId**>  
 <**artifactId**>jjwt</**artifactId**>  
 <**version**>0.9.1</**version**>  
</**dependency**>

**public class** JwtUtil {  
 **private** String **SECRET\_KEY** = **"secret"**;  
  
 **public** String extractUsername(String token) {  
 **return** extractClaim(token, Claims::getSubject);  
 }  
  
 **public** Date extractExpiration(String token) {  
 **return** extractClaim(token, Claims::getExpiration);  
 }  
  
  
 **public** <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {  
 **final** Claims claims = extractAllClaims(token);  
 **return** claimsResolver.apply(claims);  
 }  
  
 **public** Claims extractAllClaims(String token) {  
 **return** Jwts.*parser*().setSigningKey(**SECRET\_KEY**).parseClaimsJws(token).getBody();  
 }  
  
 **private** Boolean isTokenExpired(String token) {  
 **return** extractExpiration(token).before(**new** Date());  
 }  
  
 **public** String generateToken(UserDetails userDetails) {  
 Map<String, Object> claims = **new** HashMap<>();  
 **return** createToken(claims, userDetails.getUsername());  
 }  
  
 **private** String createToken(Map<String, Object> claims, String subject) {  
 **return** Jwts.*builder*().setClaims(claims).setSubject(subject)  
 .setIssuedAt(**new** Date(System.*currentTimeMillis*()))  
 .setExpiration(**new** Date(System.*currentTimeMillis*() + 1000 \* 60 \* 60 \* 10))  
 .signWith(SignatureAlgorithm.***HS256***, **SECRET\_KEY**).compact();  
 }  
  
 **public** Boolean validateToken(String token, UserDetails userDetails) {  
 **final** String username = extractUsername(token);  
 **return** (username.equals(userDetails.getUsername()) && !isTokenExpired(token));  
 }  
}

Step 1

Create authenticate API endpoint

Accepts user Id and password

Returns JWT as response

**public class** AuthenticationRequest {  
 **private** String **username**;  
 **private** String **password**;  
  
 **public** AuthenticationRequest() {  
 }  
  
 **public** AuthenticationRequest(String username, String password) {  
 **this**.**username** = username;  
 **this**.**password** = password;  
 }  
  
 **public** String getUsername() {  
 **return username**;  
 }  
  
 **public void** setUsername(String username) {  
 **this**.**username** = username;  
 }  
  
 **public** String getPassword() {  
 **return password**;  
 }  
  
 **public void** setPassword(String password) {  
 **this**.**password** = password;  
 }  
}

**public class** AuthenticationResponse {  
 **private final** String **jwt**;  
  
 **public** AuthenticationResponse(String jwt) {  
 **this**.**jwt** = jwt;  
 }  
  
 **public** String getJwt() {  
 **return jwt**;  
 }  
}

@RestController  
**public class** HelloResource {  
 @Autowired  
 **private** AuthenticationManager **authenticationManager**;  
  
 @Autowired  
 **private** MyUserDetailsService **userDetailsService**;  
  
 @Autowired  
 **private** JwtUtil **jwtUtil**;  
  
 @RequestMapping(**"/hello"**)  
 **public** String hello() {  
 **return "Hello World"**;  
 }  
  
 @RequestMapping(value = **"/authenticate"**, method = RequestMethod.***POST***)  
 **public** ResponseEntity<?> createAuthenticationToken(@RequestBody AuthenticationRequest authenticationRequest) **throws** Exception {  
 **try** {  
 **authenticationManager**.authenticate(**new** UsernamePasswordAuthenticationToken(authenticationRequest.getUsername(), authenticationRequest.getPassword()));  
 }**catch** (BadCredentialsException e){  
 **throw new** Exception(**"Incorrect username and password"**, e);  
 }  
  
 **final** UserDetails userDetails = **userDetailsService**.loadUserByUsername(authenticationRequest.getUsername());  
 **final** String jwt = **jwtUtil**.generateToken(userDetails);  
 **return** ResponseEntity.*ok*(**new** AuthenticationResponse(jwt));  
 }  
}

POST REQUEST: <http://localhost:8080/authenticate>

HEADER:

Content-Type: application/json

BODY:

{

    "username": "foo",

    "password": "foo"

}

GET REQUEST: <http://localhost:8080/hello>

HEADER:

Content-Type: application/json

Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJmb28iLCJleHAiOjE2MDUxMjMxODMsImlhdCI6MTYwNTA4NzE4M30.akF3UF34s3CJMTnhN19dMoklvBzNtu9yk69B8NEsud4

This get request will work once we suppor jwt token.

Step 2

Intercept all incoming request

-Extract JWT from the header

-Validate and set in execution context

To do this is to create filter.

OncePerRequestFilter

Run once per request

@Component  
**public class** JwtRequestFilter **extends** OncePerRequestFilter {  
  
 @Autowired  
 **private** MyUserDetailsService **userDetailsService**;  
  
 @Autowired  
 **private** JwtUtil **jwtUtil**;  
  
 @Override  
 **protected void** doFilterInternal(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, FilterChain filterChain) **throws** ServletException, IOException {  
 **final** String authorizationHeaer = httpServletRequest.getHeader(**"Authorization"**);  
 String username = **null**;  
 String jwt = **null**;  
 **if**(authorizationHeaer != **null** && authorizationHeaer.startsWith(**"Bearer "**)){  
 jwt = authorizationHeaer.substring(7);  
 username = **jwtUtil**.extractUsername(jwt);  
 }  
  
 **if**(username != **null** && SecurityContextHolder.*getContext*().getAuthentication() == **null**) {  
 UserDetails userDetails = **this**.**userDetailsService**.loadUserByUsername(username);  
 **if**(**jwtUtil**.validateToken(jwt, userDetails)) {  
 UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = **new** UsernamePasswordAuthenticationToken(userDetails, **null**, userDetails.getAuthorities());  
 usernamePasswordAuthenticationToken.setDetails(**new** WebAuthenticationDetailsSource().buildDetails(httpServletRequest));  
 SecurityContextHolder.*getContext*().setAuthentication(usernamePasswordAuthenticationToken);  
 }  
 }  
 filterChain.doFilter(httpServletRequest, httpServletResponse);  
 }  
}

// Add this in order to work for above filter.

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

// Spring don’t manage session

http.addFilterBefore(**jwtRequestFilter**, UsernamePasswordAuthenticationFilter.**class**);

@EnableWebSecurity  
**public class** SecurityConfigurer **extends** WebSecurityConfigurerAdapter {  
  
 @Autowired  
 **private** MyUserDetailsService **myUserDetailsService**;  
  
 @Autowired  
 **private** JwtRequestFilter **jwtRequestFilter**;  
 @Override  
 **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.userDetailsService(**myUserDetailsService**);  
 }  
  
 @Bean  
 **public** PasswordEncoder passwordEncoder() {  
 **return** NoOpPasswordEncoder.*getInstance*();  
 }  
  
 @Override  
 @Bean  
 **public** AuthenticationManager authenticationManagerBean() **throws** Exception {  
 **return super**.authenticationManagerBean();  
 }  
  
 @Override  
 **protected void** configure(HttpSecurity http) **throws** Exception {  
 http.csrf().disable()  
 .authorizeRequests().antMatchers(**"/authenticate"**).permitAll()  
 .anyRequest().authenticated()  
 .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***STATELESS***);  
 http.addFilterBefore(**jwtRequestFilter**, UsernamePasswordAuthenticationFilter.**class**);  
 }  
}

**Lesson 13**

**OAUTH**

* Mean for authorization
* Originally created authorization between services

Eg: Photo Printing Service

1. Upload photo
2. Pay
3. Order prints

Feature request (Photo in cloud)

Please add google driver and import feature (without user download)

1. Import photo
2. Pay
3. Order prints

Photo print app => google drive service

Google driver need your authentication and how photo print app can access.

1. Photo print app can ask your username and password(no you don’t trust)

You give access to certain photo. Not whole account.

1. Share your photo(may be you don’t want)

To solve this OAuth come into it.

Valet key example: (parking)

Rich car driver and hand over key to valet. Valet take key and park. Car is expensive. What if valley take car key and long driver and open trunk.

Master car key & valet key.

Valet key can only open the car.

Service can authorize each other.

OAuth flow:

Look video.

**Lesson 14:**

**OAuth terminologies and flows explained - OAuth tutorial -**

Resource => Protected Resource (photos on google drive)

Resource Owner => User

Resource Server => Google drive

Client => Photo Printing Service

Who has the burden of security?

Google drive

Authorization server=> take care of authorization

The server issuing access tokens to the client

**OAuth Flow 1**

**Authorization Code Flow**

1. User logged in. Resource Owner asked client to access google drive photo
2. Photo printing service know google drive has authorization server. It sends request to authorization server.Authorizatin server: resource server might ask me but cannot trust client. Authroization only trust user.
3. Authrozation server ask user that phtoto printing service wants to accesss photot.

Which include permission.

1. Resource owner give permission
2. Authroization server provide auth token (short lived)to photo printing server.
3. Photo printing service use authorization token and contact authorization server to get access token.
4. Authroization server provide access token to photo printing service.
5. Call api call to google drive with access token for file access
6. Google drive verify token and send file.

Best and safe flow.

**OAuth Flow 2**

**Implicit flow**

Same as flow 1 but simplified.

1. Same as above
2. Same as above
3. Same as above
4. Same as above
5. Authorization server send access token directly
6. Same as step 8
7. Same as step 9

Drawbacks:

Someone get access token then they can access resource. (less secure)

Used with JavasScript apps

**OAuth Flow 3**

**OAuth for authorization between microservices**

**Client Credentials Flow**

When the client is well trusted (confidential clients)

Micros service1 => call=> mircorservice2 => database

1. Mircorservice1 call Auth Server with client1
2. Auth Server provide access token
3. Microservice 1 send api call to microservice2 with access token
4. Microservice2 provide resource to microservice1

This is used when microservice has access to certain api

**LESSON 15**

**Implementing login with Facebook and Github from scratch - Java Brains**

Oauth for authentication