Spring Security 5

Implementing Spring Security 5 in an existing Spring Boot 2 application

gfi.be - gfi.lu - gfi.world

Who am I?

Jesus Perez Franco

https://www.linkedin.com/in/jpefranco/

Application Architect

- **Agile** DevOps Enthusiast
- I have been an **all-purpose Software Engineer** for over 17 years' experience.
- I worked at **Gfi Spain** for 12 years
- I arrived in **GfiBelux in January** 2018
- I'm currently working for **Toyota Motors Europe** in the CAST Team

Besides work…

- Enjoying a lot with **my family**
  - Two children ☺
- Doing **sport**
  - Running, Fitness, Footboll...
- **Planting** vegetables
  - Ecological Orchard
- Drinking Belgium **beer**
  - With colleagues better ;)

Summary

Session 1

I. **Introduction**: Spring Framework vs. Spring Boot vs. Spring Security
II. What is Spring Security?
III. Spring Security **Fundamentals** I
IV. Spring Security **Workflow**
V. Spring Security **Architecture**
VI. Spring Security **Configuration**

Authentication methods

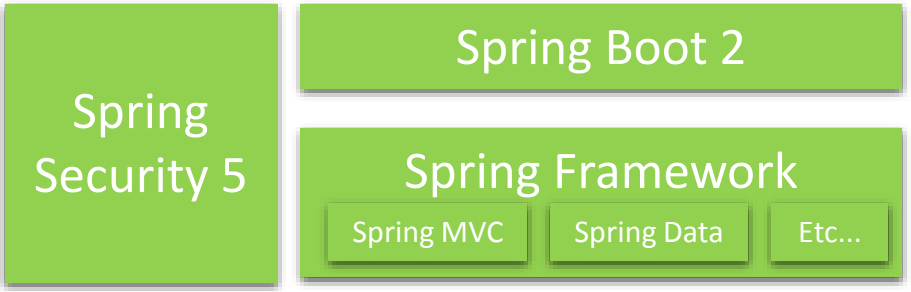
- In-Memory Authentication
- JDBC Authentication
- LDAP Authentication
- UserDetailsService
- AuthenticationProvider

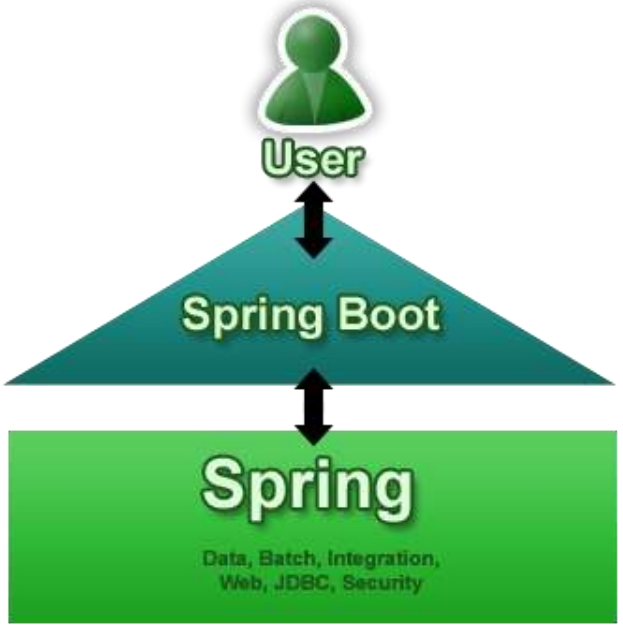
Session 2

VII. Spring Security **Fundamentals** II
VIII. Spring Security with **REST API** or RESTful Web Services
IX. Spring Security **OAuth2**
X. JSON Web Token (**JWT**) with REST API
XI. Practice: Impl Security

Introduction:
Spring Framework vs. Spring Boot vs. Spring Security

- **Spring Framework** is a Java platform that provides comprehensive infrastructure support for developing Java applications.
- **Spring Boot** is based on the **Spring Framework**, providing auto-configuration features to your Spring applications and is designed to get you up and running as quickly as possible.
- **Spring Security** provides comprehensive security services for Java EE-based software applications. There is a particular emphasis on supporting projects built using the **Spring Framework**.





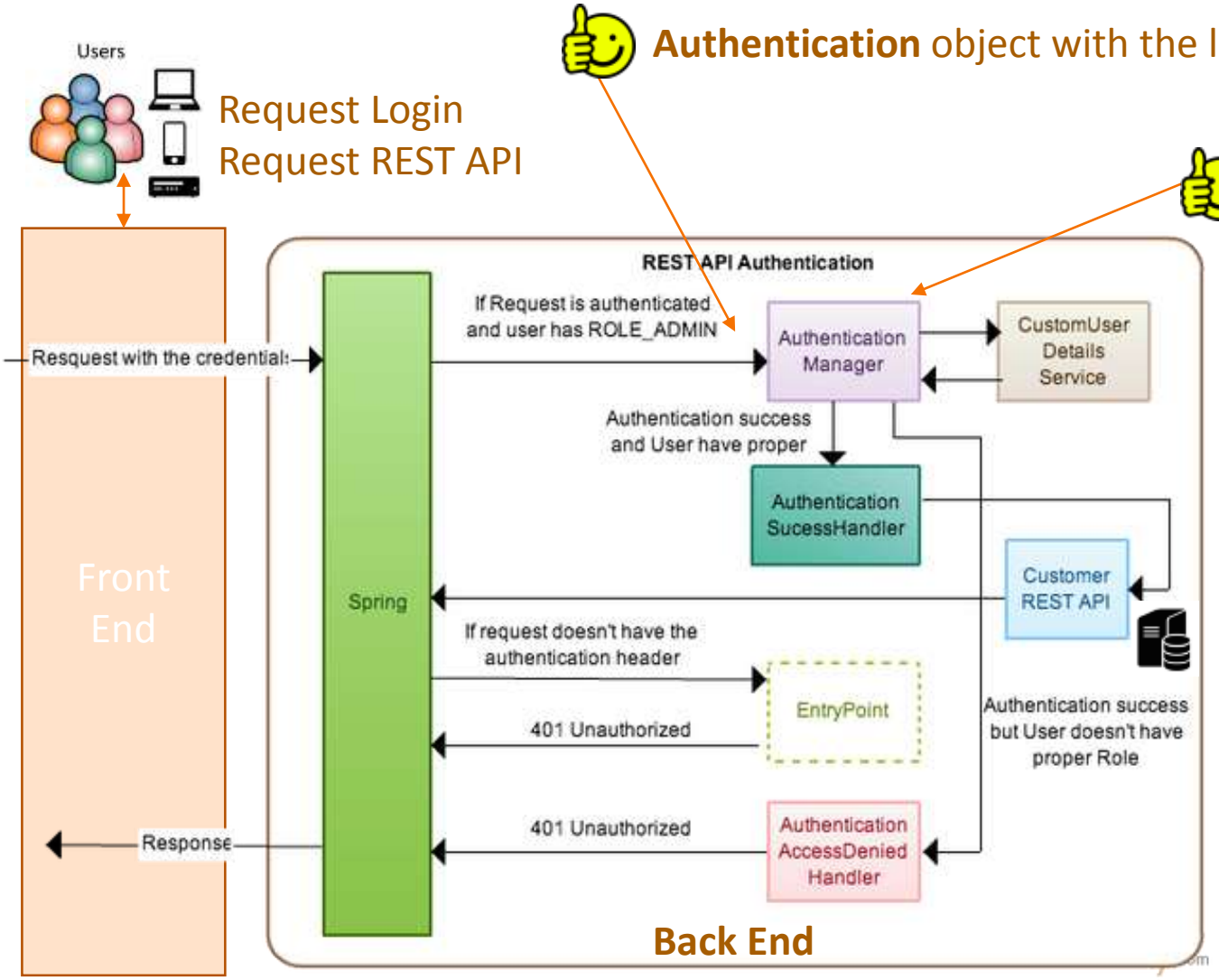
What is Spring Security?

- Spring Security is a **framework** that focuses on **providing both authentication and authorization** (or “access-control”) to Java web application and SOAP/RESTful web services
- Spring Security currently **supports integration** with all of the following **technologies**:
  - HTTP basic access authentication
  - LDAP system
  - OpenID identity providers
  - JAAS API
  - CAS Server
  - ESB Platform
  - .....
  - Your own authentication systems
- It is built on top of Spring Framework

Spring Security Fundamentals I

- Principal
  - User that performs the action
- Authentication
  - Confirming truth of credentials
- Authorization
  - Define access policy for principal
- GrantedAuthority
  - Application permission granted to a principal
- SecurityContext
  - Hold the authentication and other security information
- SecurityContextHolder
  - Provides access to SecurityContext
- AuthenticationManager
  - Controller in the authentication process
- AuthenticationProvider
  - Interface that maps to a data store which stores your user data.
- Authentication Object
  - Object is created upon authentication, which holds the login credentials.
- UserDetails
  - Data object which contains the user credentials, but also the Roles of the user.
- UserDetailsService
  - Collects the user credentials, authorities(roles) and build an UserDetails object.

Spring Security Common Workflow



Authentication object with the login credentials is created

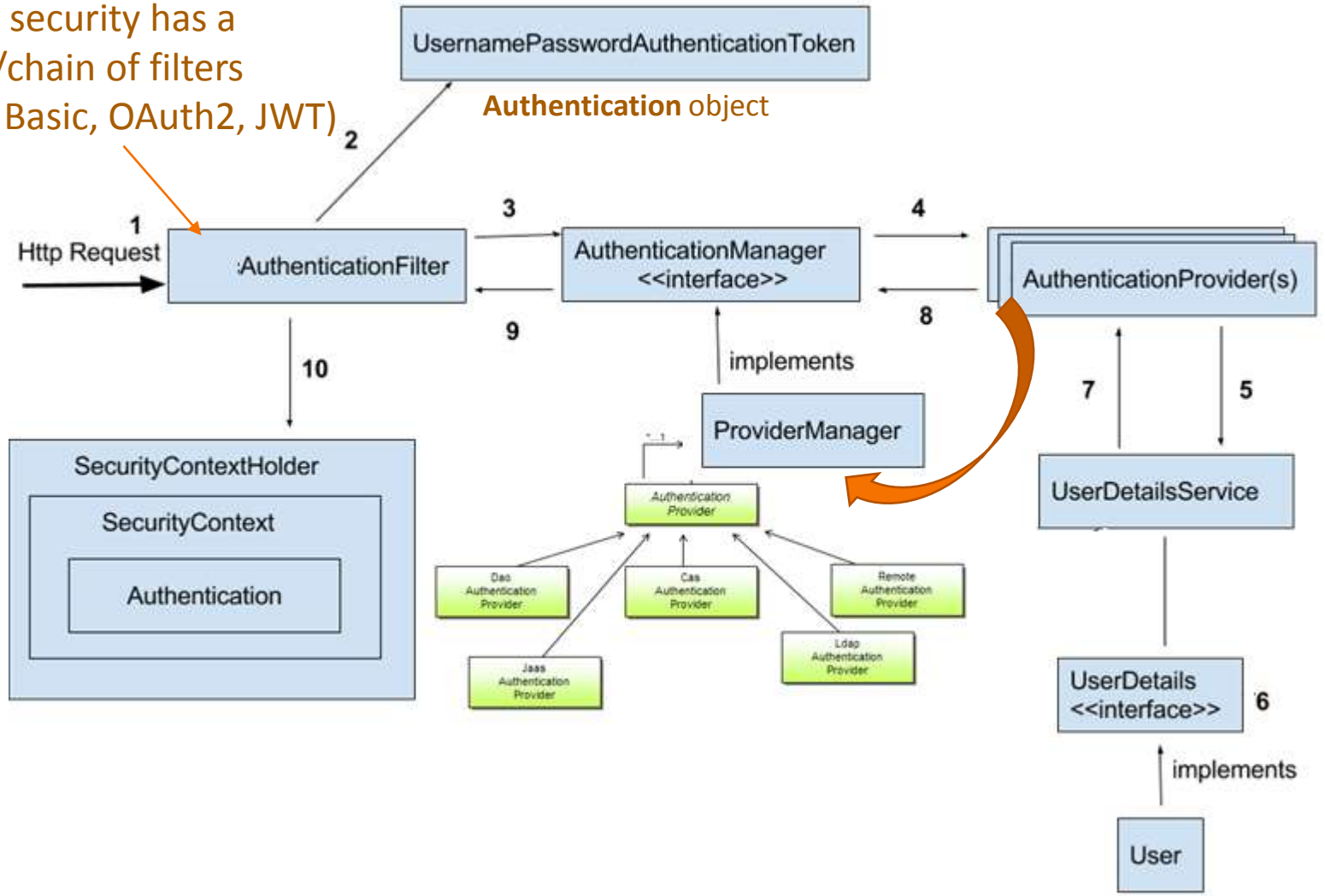
Controller in the process that it just delegates to a collection of **AuthenticationProvider** instances.

Call an object that implements the **UserDetailsService** interface, which it looks up the user data and returns a **UserDetails** object.

Will check that the password matches the password the user entered.

Spring Security Architecture

Spring security has a series/chain of filters (HTTP Basic, OAuth2, JWT)



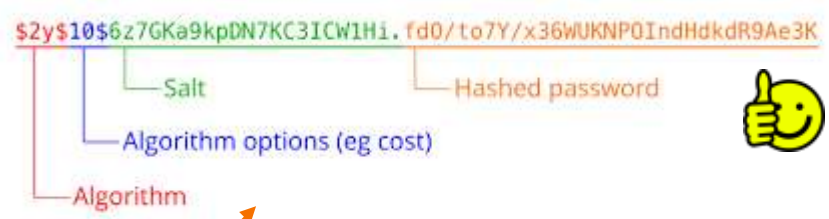
Spring Security Configuration

- Step 1. Project structure
- Step 2. Maven or Gradle dependencies
- Step 3. Spring Security configuration
  - Web Security Authentication (**@EnableWebSecurity**)
    - In-Memory Authentication
    - JDBC Authentication
    - LDAP Authentication
    - UserDetailsService
    - AuthenticationProvider
  - REST API
  - OAuth 2.0 SSO Authenticatic
  - JSON Web Token (JWT)

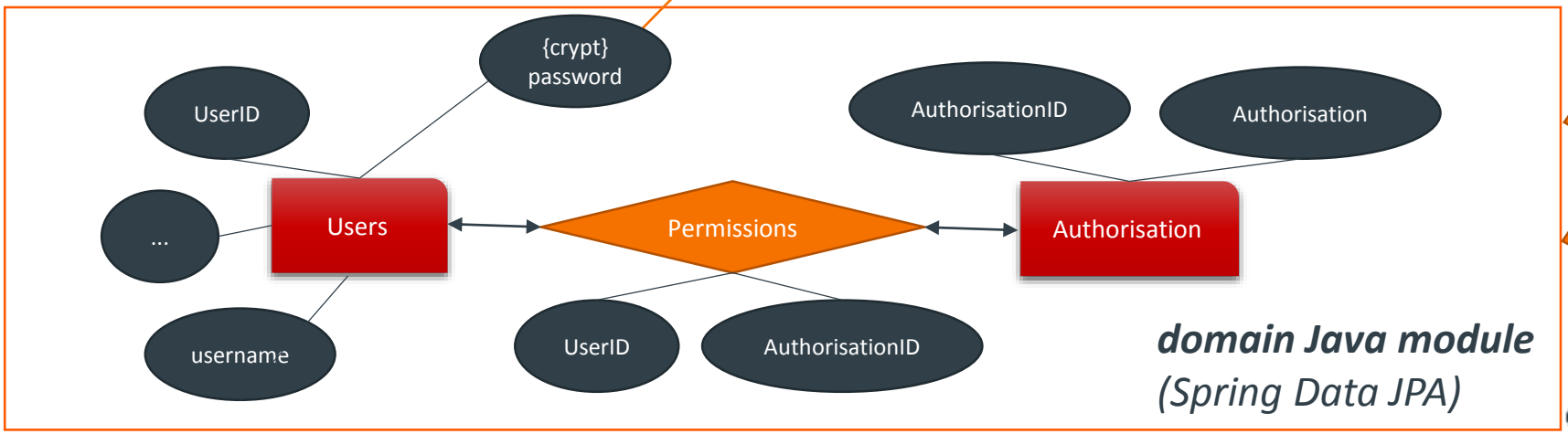
WebSecurityConfigurerAdapter

business Java module
(Spring Data REST)

security Java module
(Spring Security)



Use **BCrypt**, as it's usually the best solution available.

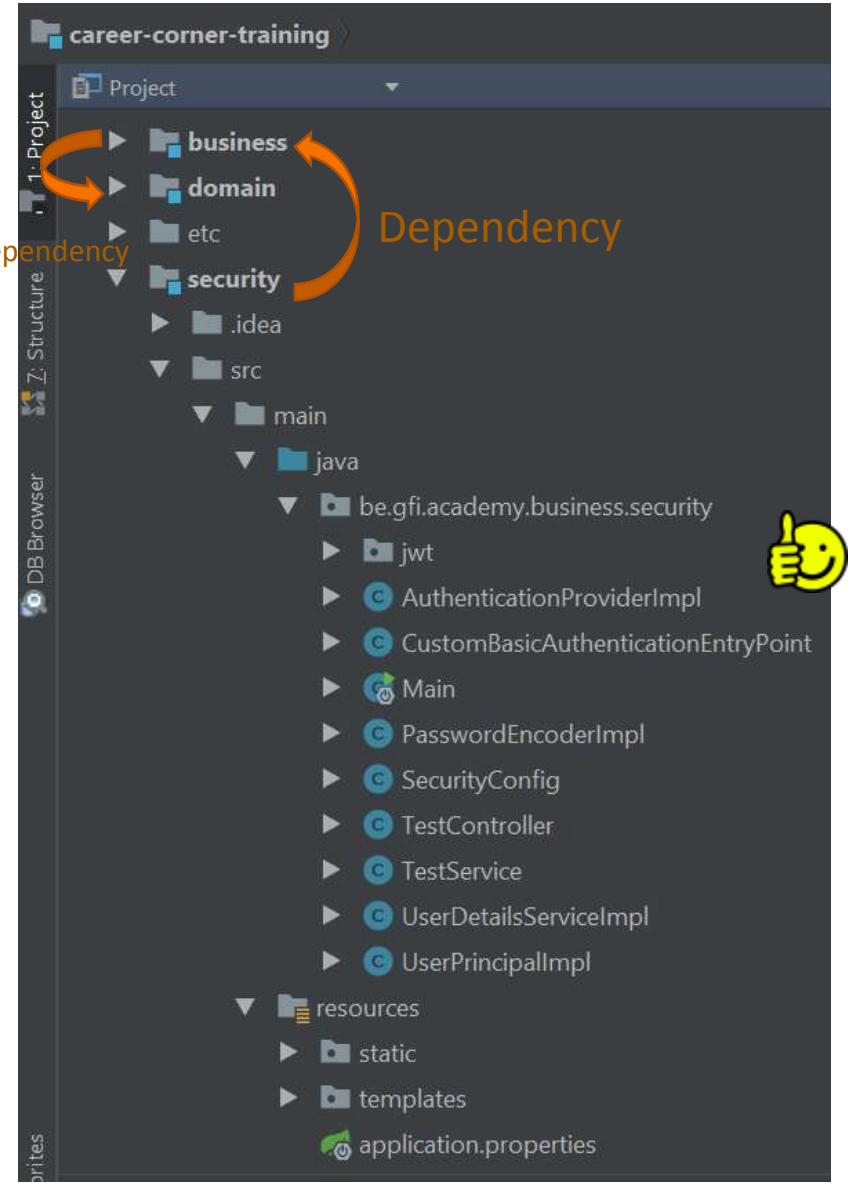


domain Java module
(Spring Data JPA)

import

Spring Security Configuration:

Project structure



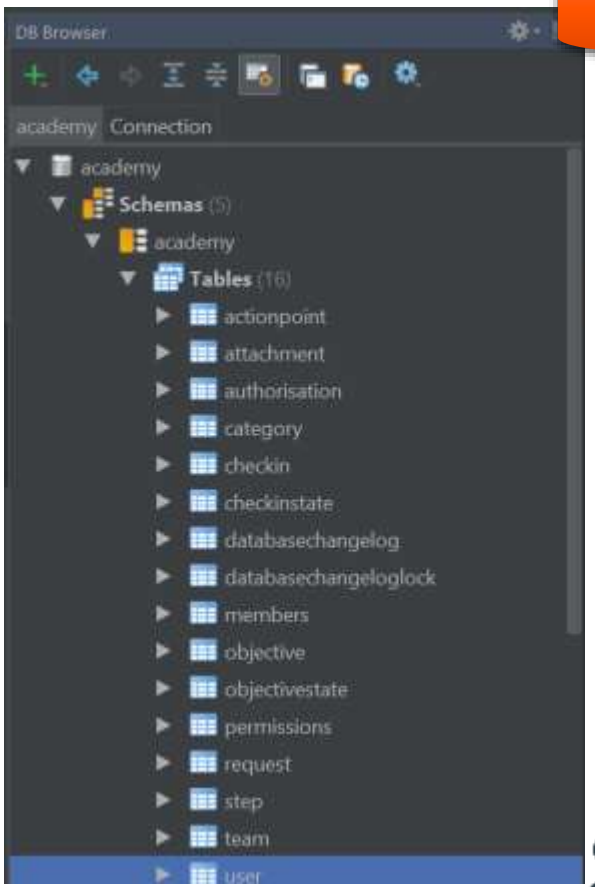
You can use start.spring.io to generate a basic project.

You can create the project web+spring security from IDE

Spring Security provides a variety of options for performing authentication.

We are going to see the most frequently used.

Database



Spring Security Configuration:
Maven dependencies

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>5.0.7.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>5.0.7.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>5.0.7.RELEASE</version>
</dependency>
```

You can get hold of Spring Security in several ways. You can also use Gradle and to set the build.gradle file.

- pom.xml
  - Each release uses a standard triplet of integers: MAJOR.MINOR.PATCH
  - All GA releases (i.e. versions ending in .RELEASE) are deployed to Maven Central.
  - If you are using a SNAPSHOT version, you will need to have the Spring Snapshot repository:

```
<repositories>
  <repository>
    <id>spring-snapshot</id>
    <name>Spring Snapshot Repository</name>
    <url>http://repo.spring.io/snapshot</url>
  </repository>
</repositories>
```

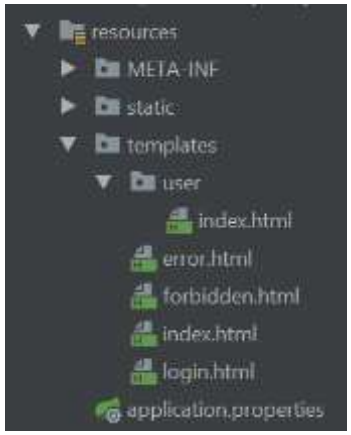
  - If you are using a release candidate or milestone version:

```
<repositories>
  <repository>
    <id>spring-milestone</id>
    <name>Spring Milestone Repository</name>
    <url>http://repo.spring.io/milestone</url>
  </repository>
</repositories>
```

Spring Security Configuration:
@EnableWebSecurity and HttpSecurity

```
@RequestMapping("/")
public String root() {
    return "redirect:/index";
}

@RequestMapping("/index")
public String index() {
    return "index";
}

@RequestMapping("/user/index")
public String userIndex() { return "user/index"; }

@RequestMapping(value = "/login")
public String login() {
    return "login";
}

@RequestMapping(value = "/forbidden")
public String forbidden() {
    return "forbidden";
}
```

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws   Exception {

        http
                .authorizeRequests()
                .antMatchers("/", "/index").permitAll()
                .anyRequest()
                .hasAnyRole("Consultant“, “Teamlead”, “Hr”)
                .and()
                .formLogin()
                .loginPage("/login").failureUrl("/login-error")
                .and()
                .exceptionHandling()
                .accessDeniedPage("/forbidden");

...
```



You have to adapt it for your own application

- Adding more configuration options:
  - Form Login
  - Authorize request
  - Handling logout, exception, custom filters...

Spring Security Configuration:

In-Memory Authentication - AuthenticationManagerBuilder


```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception
{

    auth
        .inMemoryAuthentication()
        .withUser("user").password("{noop}password")
        .roles("Consultant", "Teamlead", "Hr");
}
...
```

Configure the authentication method

*In-memory authentication is usually used in **development phase***

Spring Security Configuration:

JDBC Authentication - AuthenticationManagerBuilder

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception

DataSource dataSource = new JpaConfiguration().dataSource();

auth
        .jdbcAuthentication()
        .dataSource(dataSource)
        .usersByUsernameQuery("select username, password, 'true' as enabled"
                + " from user where username=?")
        .authoritiesByUsernameQuery("select username, authorisation as authority from"
                + " authorisation a, permissions p, user u"
                + " where a.AuthorisationID=p.AuthorisationID and"
                + " u.UserID=p.UserID and username=?")
        .passwordEncoder(passwordEncoder());
...
```

It's defined in Domain module

It's possible to use the default database scheme provided in spring security documentation or define the SQL query

Create my encryption mechanisms

With JDBC-based authentication the user's authentication and authorization information are **stored in the database.**

The standard JDBC implementation of the UserDetailsService requires tables to load the password, account status (enabled or disabled) and a list of authorities (roles) for the user. **You have to use the schema defined**.

Spring Security Configuration:

JDBC Authentication – PasswordEconderImpl

```
@Service
public class PasswordEncoderImpl implements PasswordEncoder {

    @Override
    public String encode(CharSequence rawPassword) {
        String hashed = BCrypt.hashpw(rawPassword.toString(), BCrypt.gensalt(12));
        return hashed;
    }
    @Override
    public boolean matches(CharSequence rawPassword, String encodedPassword) {
        return BCrypt.checkpw(rawPassword.toString(), encodedPassword);
    }
}
```

Default value is 10

Spring Security 5 recommends the use of **BCrypt,**

a **salt** that helps prevent pre-computed dictionary attacks.

Most of the other encryption mechanisms, such as the **MD5PasswordEncoder** and **ShaPasswordEncoder** use weaker algorithms and are now deprecated.

Spring Security Configuration:

JDBC Authentication - Encrypted properties with Spring

```
<properties>
   <database.host>localhost</database.host>
   <database.name>academy</database.name>
   <database.port>3306</database.port>
   <database.driver>com.mysql.jdbc.Driver</database.driver>
   <database.username>root</database.username>
   <database.password>{cipher}1a5f8bbf59c7290ec5b99fe91e05dd02d692cc07bb1d7bd931f7b4c27679a391
   </database.password>
   <database.url>jdbc:mysql://${database.host}:${database.port}/${database.name}?useSSL=false</database.url>
</properties>
```

Database configuration both pom.xml and application.properties file

```
be.gfi.academy.password=${database.password}
```

You must store encrypted credentials, so would **NOT allow** everybody with access to the repository to use these credentials.

*I use **Spring Boot CLI** to encrypt it:*

```
# spring encrypt secret --key root
1a5f8bbf59c7290ec5b99fe91e05dd02d692cc07bb1d7bd931f7b4c27679a391
```

```
# set ENCRYPT_KEY=secret
# mvn clean install –DskiptTests=true
# cd security
# mvn spring-boot:run
```

You have to provide the key via the property encrypt.key or **ENCRYPT_KEY** variable environment

To automate this process, you can add the encryption key to your deployment infrastructure like Jenkins, Bamboo, etc.

Spring recommends to use BCrypt BCryptPasswordEncoder, a stronger hashing algorithm with randomly generated salt.

Spring Security Configuration:

LDAP Authentication - AuthenticationManagerBuilder


```
<dependency>
  <groupId>org.springframework.ldap</groupId>
  <artifactId>spring-ldap-core</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-ldap</artifactId>
</dependency>
```

Use the configurations files, either application.properties or application.yml

```
public void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .ldapAuthentication()
        .userDnPatterns("uid={0},ou=people")
        .groupSearchBase("ou=groups")
        .contextSource()
        .url("ldap://localhost:389/dc=springframework,dc=org")
        .and()
        .passwordCompare()
        .passwordEncoder(passwordEncoder())
        .passwordAttribute("userPassword");
}
```

You must store encrypted credentials, so would **NOT allow** everybody with access to the repository to use these credentials.

encrypted.property={cipher}71144.....

To automate this process, you can add the encryption key to your deployment infrastructure like Jenkins, Bamboo and so on.

With LDAP-based authentication the user's authentication and authorization information are stored in a directory LDAP.

Spring Security Configuration:

UserDetailsService I - AuthenticationManagerBuilder


```
@Autowired
private UserDetailsServiceImpl userDetailsService;
@Autowired
private PasswordEncoderImpl passwordEncoder;

@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {

    auth
        .authenticationProvider(userDetailsService);
        .passwordEncoder(passwordEncoder);
}
```

The standard scenario uses a simple UserDetailsService where I usually store the password in database and spring security framework performs a check on the password. **What happens when you are using a different authentication system and the password is not provided in your own database/data model?**

Spring Security Configuration:

UserDetailsService II - AuthenticationManagerBuilder

```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private IUserDao userDAO;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        UserVO user = userDAO.findByUsername(username);

        if (user == null) {
            throw new UsernameNotFoundException(username);
        }

        return new UserPrincipalImpl(user);
    }
}
```

It's defined in Domain module

This User Details Service **only has access to the username** in order to retrieve the full user entity – and in a large number of scenarios, this is enough.

Spring Security Configuration:

UserPrincipal - AuthenticationManagerBuilder

In-Memory Authentication
JDBC Authentication
LDAP Authentication
UserDetailsService
AuthenticationProvider

```
public class UserPrincipalImpl implements UserDetails {
    String ROLE_PREFIX = "ROLE_";

    private UserVO user;

    public UserPrincipalImpl(UserVO user) {
        this.user = user;
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
    public String getUsername() {
        return user.getUsername();
    }

    @Override
    public Collection<GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> grantedAuthorities = new ArrayList<>();
        List<AuthorisationVO> roles = user.getRoles();

        for(AuthorisationVO item : roles) {
            grantedAuthorities.add(new SimpleGrantedAuthority(ROLE_PREFIX+item.getAuthorisation()));
        }
        return grantedAuthorities;
    }
...
...
}
```

```
@RestController
@RequestMapping("/api")
public class TestService {

    @RequestMapping("/user-information")
    @ResponseBody
    public Principal information(Principal principal) {
        return principal;
    }
}
```

API Test

http://localhost:8080/api/user-information

Spring Security Configuration:

Authentication Provider I - AuthenticationManagerBuilder

```
@Autowired
private AuthenticationProviderImpl authProvider;

@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {

    auth
        .authenticationProvider(authProvider);
}
```

More custom scenarios will still need to access the full *Authentication* request to be able to perform the authentication process – for example when authenticating against some external, third party service (e.g. CAS/IS - Centralized Authentication System/Identity Server, so my system have no idea about the password) **– both the *username* and the *password* from the authentication request will be necessary**.

For these more advanced scenarios we'll need to **define a custom Authentication Provider.**

Spring Security Configuration:

Authentication Provider II - AuthenticationManagerBuilder

```
public class AuthenticationProviderImpl implements AuthenticationProvider {

    @Autowired
    private IUserDao userDAO;

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {

        String name = authentication.getName();
        String password = authentication.getCredentials().toString();

        UserVO user = userDAO.findByUsername(name);
        if (user == null) {
            throw new BadCredentialsException("Authentication failed for " + name);
        }
        List<GrantedAuthority> grantedAuthorities = new ArrayList<>();
        if (new PasswordEncoderImpl().matches(password, user.getPassword())) {
            List<AuthorisationVO> roles = user.getRoles();

            for(AuthorisationVO item : roles) {
                grantedAuthorities.add(new SimpleGrantedAuthority(ROLE_PREFIX+item.getAuthorisation()));
            }
        }
...
```

Use the credentials

Authentication against a third-party system like a database in our use case

Must be “ROLE_”

Summary

Session 1

I. **Introduction**: Spring Framework vs. Spring Boot vs. Spring Security
II. What is Spring Security?
III. Spring Security **Fundamentals** I
IV. Spring Security **Workflow**
V. Spring Security **Architecture**
VI. Spring Security **Configuration**

Authentication methods

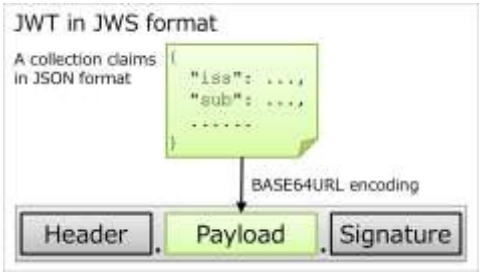
- In-Memory Authentication
- JDBC Authentication
- LDAP Authentication
- UserDetailsService
- AuthenticationProvider

Session 2

VII. Spring Security **Fundamentals** II
VIII. Spring Security with **REST API** or RESTful Web Services
IX. Spring Security **OAuth2**
X. JSON Web Token (**JWT**) with REST API
XI. Practice: Impl Security

Spring Security Fundamentals II

- **JWT (JSON Web Tokens)**
  - **It is just a token format**. JWT tokens are JSON encoded data structures containing information about issuer, subject (claims), expiration time, etc. JWT is simpler than SAML 1.1/2.0, is supported by all devices and it is more powerful than SWT (Simple Web Token). See https://jwt.io/



- **OAuth2**
  - **OAuth2 is a framework** that solves a problem when a user wants to access the data using a client software like browser-based web apps, native mobile apps or desktop apps. **OAuth2 is just for authorization**, client software can be authorized to access the resources on behalf of the end user by using an access token.

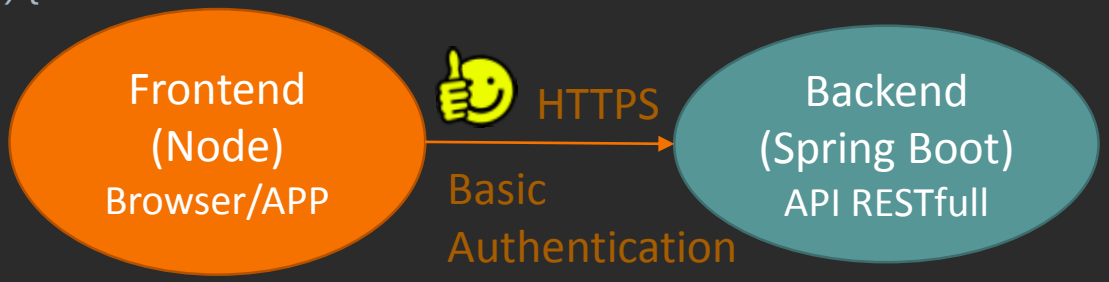
- **OpenID Connect**
  - OpenID Connect builds on **top of OAuth2** and adds authentication. OpenID Connect adds some constraints to OAuth2 like UserInfo Endpoint, ID Token, discovery and dynamic registration of OpenID Connect providers and session management. **JWT is the mandatory format for the token.**

Spring Security with REST API

Authorization with annotations on controller I

```
@RestController
@RequestMapping("/api")
public class TestService {
    @Secured("ROLE_Consultant")
    @RequestMapping("/user-information")
    public Principal information(Principal principal) {
        return principal;
    }
    @PreAuthorize("hasRole('ROLE_Consultant') and hasRole('ROLE_Teamlead') and hasRole('ROLE_Hr')")
    @RequestMapping("/user-details")
    @ResponseBody
    public Object details(Authentication auth) {
        return auth.getDetails();
    }
    @Secured("ROLE_Consultant")
    @RequestMapping("/thanks")
    @ResponseBody
    public String getThanks() {
        return "{\"message\":\"Thanks!\"}";
    }
}
```

API with access control



Note: Base64 encoding does not mean encryption or hashing! This method is equally secure as sending the credentials in clear text (base64 is a reversible encoding). Prefer to use HTTPS in conjunction with Basic Authentication.

Spring Security with REST API

Authorization with annotations on controller II

Basic Authentication with REST API (stateless). **Very basic mechanism**. Should be used only in HTTPS environmen

```
C:\Users\academy\career-corner-training>curl -i -u jpefranco:jperanco http://localhost:8080/index/user
HTTP/1.1 401
WWW-Authenticate: Basic realm="Realm"
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Set-Cookie: JSESSIONID=2D8DA44DF55156A957924B58496938F5; Path=/; HttpOnly
WWW-Authenticate: Basic realm="Realm"
Content-Length: 0
Date: Sat, 15 Sep 2018 17:13:39 GMT
```

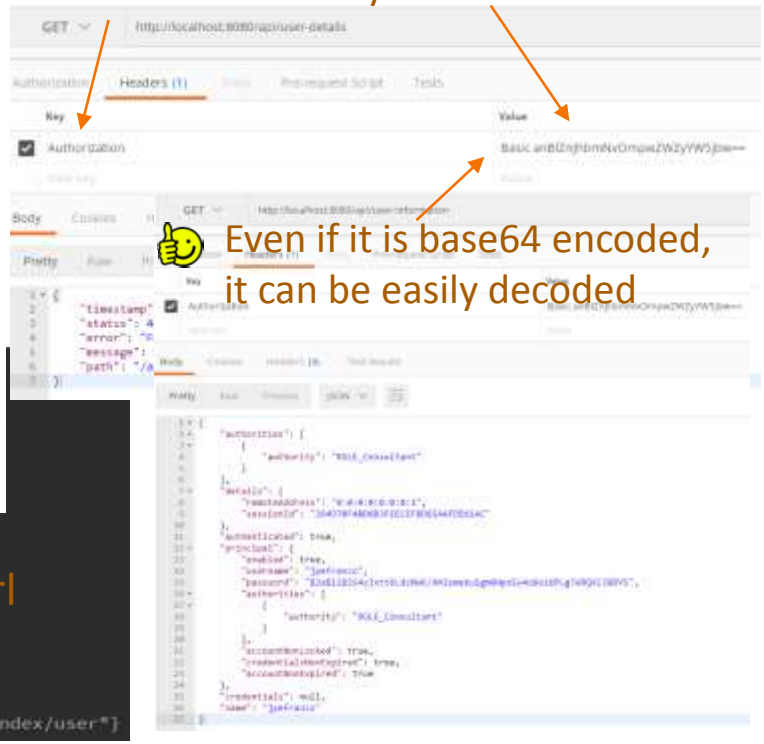
Traditional form based authentication
Cookie/Session Based Authentication (stateful)

Login page

Example user: jpefranco / jpefranco

Example with Google account

Username:
Password:
Log in
Back to Home Page



Even if it is base64 encoded, it can be easily decoded

```
C:\Users\academy\career-corner-training>curl -i http://localhost:8080/index/user
HTTP/1.1 401
Set-Cookie: JSESSIONID=ECFCC2F75567DA112FE3085849170B0F; Path=/; HttpOnly
WWW-Authenticate: Basic realm="Realm"
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Sat, 15 Sep 2018 17:11:45 GMT

{"timestamp":"2018-09-15T17:11:45.878+0000","status":401,"error":"Unauthorized","message":"Unauthorized","path":"/index/user"}
```

Use the Postman tool or the curl command testing your API Rest

By default, the **BasicAuthenticationEntryPoint** provisioned by Spring Security returns a full page for a 401 Unauthorized response back to the client. This HTML representation of the error renders well in a browser, but it not well suited for other scenarios, such as a REST API where a json representation may be preferred.

Spring Security with REST API

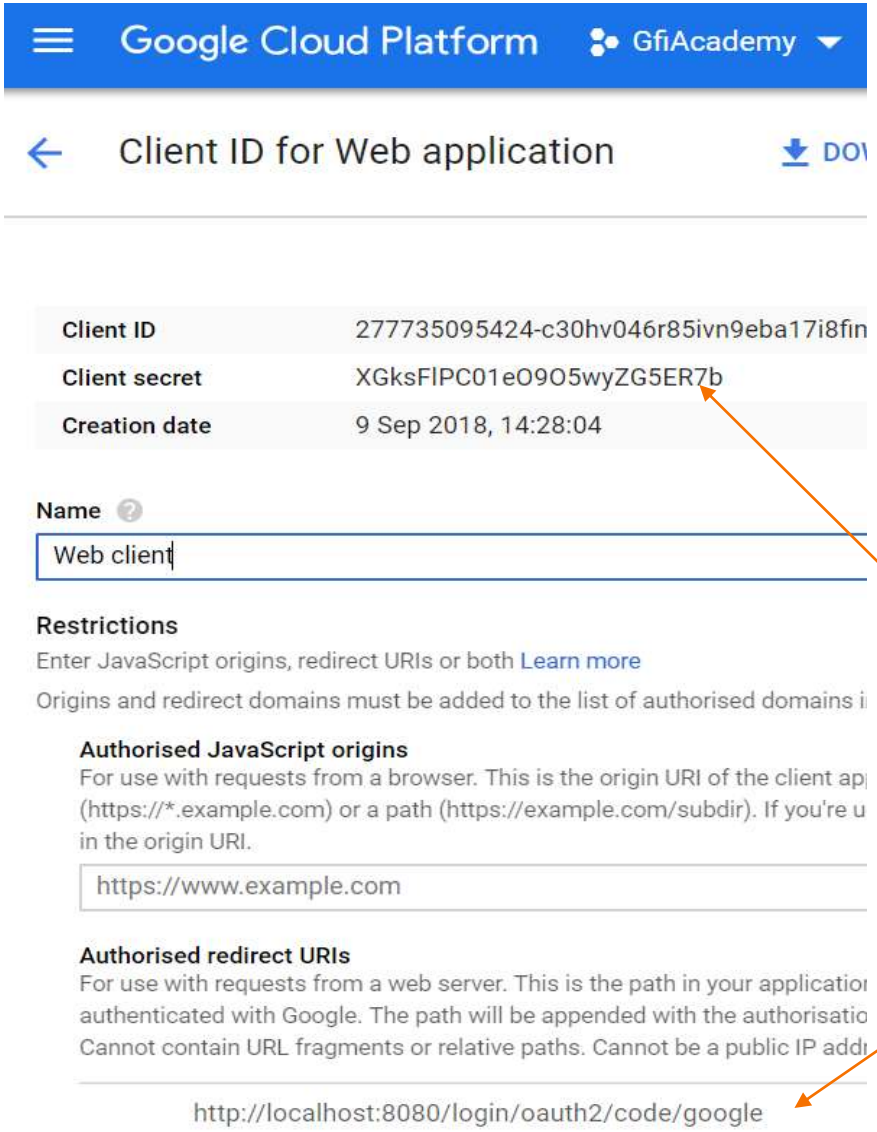
Authorization with annotations on controller III

```
public class CustomBasicAuthenticationEntryPoint extends BasicAuthenticationEntryPoint {
    @Override
    public void commence (HttpServletRequest request, HttpServletResponse response, AuthenticationException
authEx throws IOException {
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        response.setHeader("WWW-Authenticate", "Basic realm=" + getRealmName());
        response.setContentType(MediaType.APPLICATION_JSON_UTF8_VALUE);

        PrintWriter writer = response.getWriter();
        writer.println("HTTP Status 401 : " + authEx.getMessage());
    }
    @Override
    public void afterPropertiesSet() throws Exception {
        setRealmName(SecurityConfig.REALM_NAME);
        super.afterPropertiesSet();
    }
}
```

```
http
        .authorizeRequests().anyRequest().authenticated()
        .and()
        .httpBasic()
        .authenticationEntryPoint(authenticationEntryPoint);
```

Implementation of a custom BasicAuthenticationEntryPoint

Spring Security OAuth2
Single Sign-On



To begin, obtain OAuth 2.0 client credentials from the Google API Console.

Google's OAuth 2.0 APIs can be used for both authentication and authorization, which conforms to the OpenID Connect specification.

Copying the client-id and client-secret to application.properties or application.yml

Setting the redirect URI

Spring Security OAuth2
Single Sign-On

Application.properties file

```
spring.security.oauth2.client.registration.google.client-id=277735095424.....
spring.security.oauth2.client.registration.google.client-secret=XGksESFC01e58.....
```

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-client</artifactId>
  <version>${spring-security-version}</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-jose</artifactId>
  <version>${spring-security-version}</version>
</dependency>
```

```
@Override
 protected void configure(HttpSecurity http) {
http
        .authorizeRequests()
        .antMatchers("/", "/index")
        .permitAll()
        .antMatchers("/user/**")
        .authenticated()
        .and()
        .oauth2Login()
        .loginPage("/login");
```

Use the Google's OAuth 2.0 endpoints to authorize access to Google APIs.

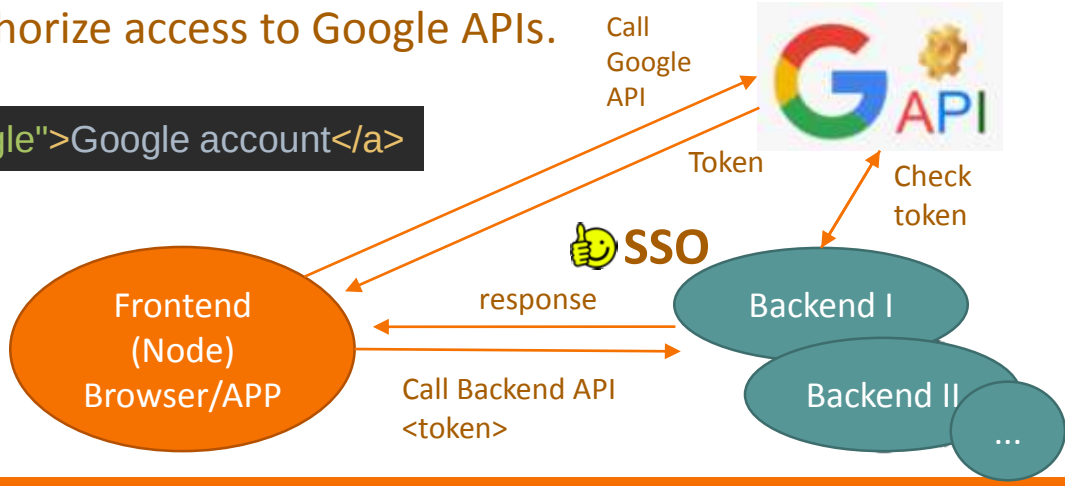
Login page

Example user: jpefranco / jpefranco

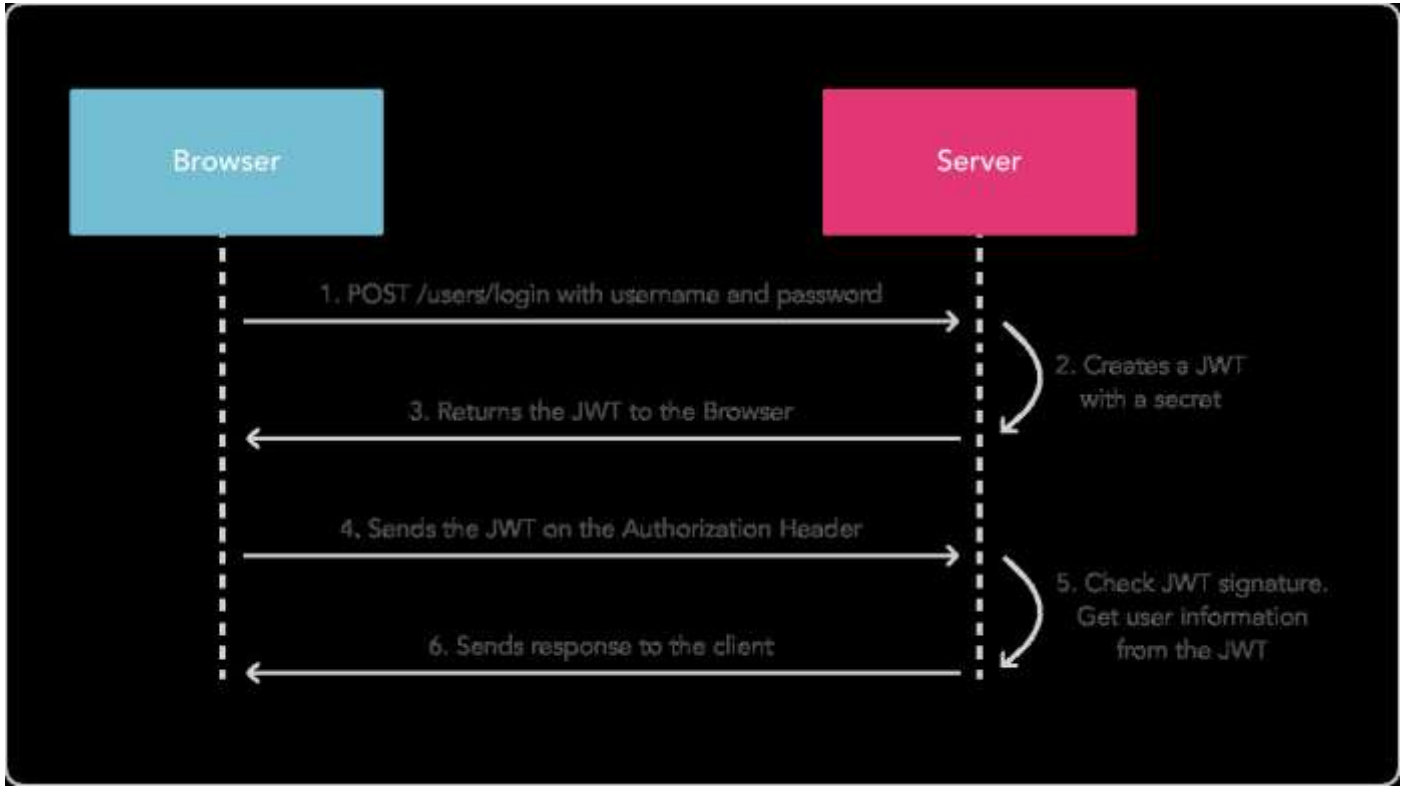
Example with Google account

Username:
Password:
Log in
Back to Home Page

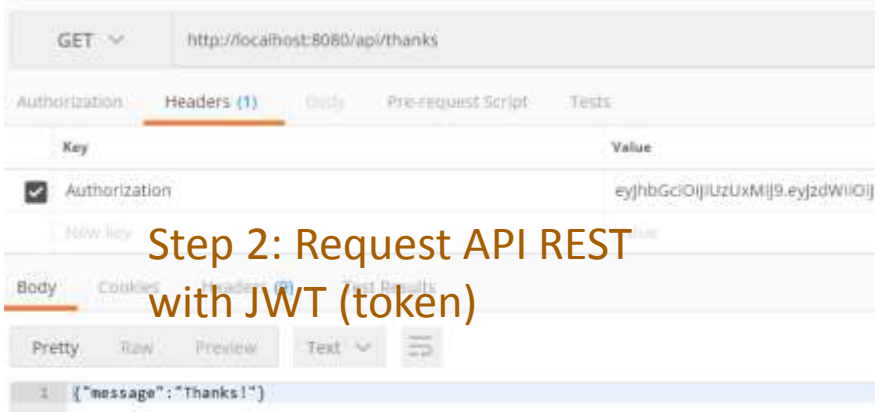
```
<a href="/oauth2/authorization/google">Google account</a>
```



JSON Web Token (JWT) with REST API



Step 1: Request /api/login and get the JWT

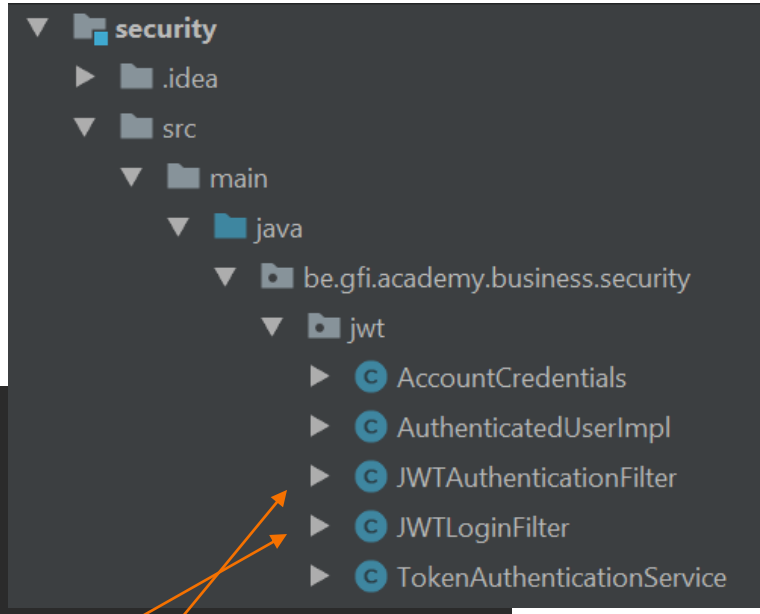


Step 2: Request API REST with JWT (token)

JSON Web Token (JWT) with REST API

Configuration

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.0</version>
</dependency>
```




```
@Override
 protected void configure(HttpSecurity http) throws   Exception {

http
        .authorizeRequests()
        .antMatchers(HttpMethod.POST,"/api/login")
        .permitAll()
        .anyRequest()
        .authenticated()
        .and()
        // We filter the api/login requests
        .addFilterBefore(new JWTLoginFilter("/api/login", authenticationManager()),
                UsernamePasswordAuthenticationFilter.class)
        // And filter other requests to check the presence of JWT in header
        .addFilterBefore(new JWTAuthenticationFilter(),
                UsernamePasswordAuthenticationFilter.class);
```

JSON Web Token (JWT) with REST API

JWTLoginFilter

```
public class JWTLoginFilter extends AbstractAuthenticationProcessingFilter{

    private TokenAuthenticationService tokenAuthenticationService;

    public JWTLoginFilter(String url, AuthenticationManager authenticationManager)
    {
        super(new AntPathRequestMatcher(url));
        setAuthenticationManager(authenticationManager);
        tokenAuthenticationService = new TokenAuthenticationService();
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse)
            throws AuthenticationException, IOException, ServletException {
        AccountCredentials credentials = new ObjectMapper().readValue(httpServletRequest.getInputStream(),AccountCredentials.class);
        UsernamePasswordAuthenticationToken token = new UsernamePasswordAuthenticationToken(credentials.getUsername(),
credentials.getPassword());
        return getAuthenticationManager().authenticate(token);
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication
authentication)
            throws IOException, ServletException{
        String name = authentication.getName();
        tokenAuthenticationService.addAuthentication(response,name);
    }
}
```

JSON Web Token (JWT) with REST API

TokenAuthenticationService

```
public class TokenAuthenticationService {

    private long EXPIRATIONTIME = 1000 * 60 * 60 * 24 * 10; // 10 days
    private String secret = "secret";    // Must be an environment variable into OS
    private String tokenPrefix = "";   // "Bearer "
    private String headerString = "Authorization";

    public void addAuthentication(HttpServletResponse response, String username)
    {
        String JWT = Jwts.builder()
                .setSubject(username)
                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATIONTIME))
                .signWith(SignatureAlgorithm.HS512, secret).compact();
        response.addHeader(headerString,tokenPrefix + JWT);
    }

    public Authentication getAuthentication(HttpServletRequest request)
    {
        String token = request.getHeader(headerString);
        if(token != null)
        {
            String username = Jwts.parser()
                    .setSigningKey(secret)
                    .parseClaimsJws(token)
                    .getBody()
                    .getSubject();
            if(username != null) // we managed to retrieve a user
            {
                return new AuthenticatedUserImpl(username);
            }
        }
        return null;
    }
}
```

JSON Web Token (JWT) with REST API

AuthenticatedUserImpl

```
public class AuthenticatedUserImpl implements Authentication {

    String ROLE_PREFIX = "ROLE_";

    private boolean authenticated = true;
    private String name;

    @Autowired
    private IUserDao userDAO;

    AuthenticatedUserImpl(String name){
        this.name = name;
    }

    @Override
    public Collection<GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> grantedAuthorities = new ArrayList<>();

        UserVO user = userDAO.findByUsername(name);
        List<AuthorisationVO> roles = user.getRoles();

        for(AuthorisationVO item : roles) {
            grantedAuthorities.add(new SimpleGrantedAuthority(ROLE_PREFIX+item.getAuthorisation()));
        }
        grantedAuthorities.add(new SimpleGrantedAuthority(ROLE_PREFIX+"Consultant"));

        return grantedAuthorities;
    }
....
....
```

JSON Web Token (JWT) with REST API

JWTLoginFilter and AccountCredentials

```
public class JWTAuthenticationFilter extends GenericFilterBean{

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) throws IOException, ServletException {
        Authentication authentication = new TokenAuthenticationService().getAuthentication((HttpServletRequest)request);

        SecurityContextHolder.getContext().setAuthentication(authentication);
        filterChain.doFilter(request,response);
    }
}
```

```
public class AccountCredentials {

    private String username;
    private String password;

    String getUsername() { return username; }
    String getPassword() { return password; }

    public void setUsername(String _username) { this.username = _username; }
    public void setPassword(String _password) { this.password = _password; }
}
```

Jesus Perez Franco

Application Architect
Digital Business
Gfi

Jesus.perez.franco@gfi.be
https://www.linkedin.com/in/jpefranco/

Registered office
Square de Meeûs 28/40 – 1000 Brussels
Tel: +32(0)16381111
Fax: +32(0)16381100
info@gfi.be