Security

REF:

- https://docs.spring.io/spring-security/site/docs/current/reference/html5/#introduction
- https://www.marcobehler.com/guides/spring-security
- https://medium.com/javarevisited/springboot-security-with-jwt-fca1446790ba
- https://www.youtube.com/watch?v=her_7pa0vrg

- Secure the application using Basic Authentication; consume secured endpoints; define user and roles; store hashed credentials; describe different types of hashing algorithms
- Provide method based authorization
- Describe the concepts of OAuth and how JWT tokens are used
- Describe the concepts of Single Sign On and provide use cases
- Secure the Spring Boot micro service using Oauth
- Generate SSL certificates and secure the embedded servlet container
- Provide a practical example and hands on laboratory

Sooner or later everyone needs to add security to his project and in the Spring ecosystem you do that with the help of the Spring Security library.

So you go along, add Spring Security to your Spring Boot (or plain Spring) project and suddenly…

- …you have auto-generated login-pages.
- …you cannot execute POST requests anymore.
- …your whole application is on lockdown and prompts you to enter a username and password.

Spring Security

At its core, Spring Security is really just a bunch of servlet filters that help you add authentication and authorization to your web application.

It also integrates well with frameworks like Spring Web MVC (or Spring Boot), as well as with standards like OAuth2 or SAML. And it auto-generates login/logout pages and protects against common exploits like CSRF.

Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.

It is a sub-project of Spring framework which was started in 2003 by Ben Alex. Later on, in 2004, It was released under the Apache License as Spring Security 2.0.0.

It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application.

This framework targets two major areas of application are authentication and authorization.

Authentication is the process of knowing and identifying the user that wants to access.

Authorization is the process to allow authority to perform actions in the application.

We can apply authorization to authorize web request, methods and access to individual domain.

Technologies that support Spring Security Integration

Spring Security framework supports wide range of authentication models. These models either provided by third parties or framework itself. Spring Security supports integration with all of these technologies.

- HTTP BASIC authentication headers
- HTTP Digest authentication headers
- HTTP X.509 client certificate exchange
- LDAP (Lighweight Directory Access Protocol)
- Form-based authentication
- OpenID authentication
- Automatic remember-me authentication
- Kerberos
- JOSSO (Java Open Source Single Sign-On)
- AppFuse
- AndroMDA
- Mule ESB
- DWR(Direct Web Request)

The beauty of this framework is its flexible authentication nature to integrate with any software solution. Sometimes, developers want to integrate it with a legacy system that does not follow any security standard, there Spring Security works nicely.

Advantages

Spring Security has numerous advantages:

- Comprehensive support for authentication and authorization.
- Protection against common tasks
- Servlet API integration
- Integration with Spring MVC
- Portability
- CSRF protection
- Java Configuration support

Spring Security History

In late 2003, a project **Acegi Security System for Spring** started with the intention to develop a Spring-based security system. So, a simple security system was implemented but not released officially. Developers used that code internally for their solutions and by 2004 about 20 developers were using that.

Initially, authentication module was not part of the project, around a year after, module was added and complete project was reconfigure to support more technologies.

After some time this project became a subproject of Spring framework and released as 1.0.0 in 2006.

in 2007, project is renamed to Spring Security and widely accepted. Currently, it is recognized and supported by developers open community world wide.

Spring Security Features

- LDAP (Lightweight Directory Access Protocol)
- Single sign-on
- JAAS (Java Authentication and Authorization Service) LoginModule
- Basic Access Authentication
- Digest Access Authentication
- Remember-me
- Web Form Authentication
- Authorization
- Software Localization
- HTTP Authorization

LDAP (Lightweight Directory Access Protocol)

It is an open application protocol for maintaining and accessing distributed directory information services over an Internet Protocol.

Single sign-on

This feature allows a user to access multiple applications with the help of single account(user name and password).

JAAS (Java Authentication and Authorization Service) LoginModule

It is a Pluggable Authentication Module implemented in Java. Spring Security supports it for its authentication process.

Basic Access Authentication

Spring Security supports Basic Access Authentication that is used to provide user name and password while making request over the network.

Digest Access Authentication

This feature allows us to make authentication process more secure than Basic Access Authentication. It asks to the browser to confirm the identity of the user before sending sensitive data over the network.

Remember-me

Spring Security supports this feature with the help of HTTP Cookies. It remember to the user and avoid login again from the same machine until the user logout.

Web Form Authentication

In this process, web form collect and authenticate user credentials from the web browser. Spring Security supports it while we want to implement web form authentication.

Authorization

Spring Security provides the this feature to authorize the user before accessing resources. It allows developers to define access policies against the resources.

Software Localization

This feature allows us to make application user interface in any language.

HTTP Authorization

Spring provides this feature for HTTP authorization of web request URLs using Apache Ant paths or regular expressions.

Hashing Algorithms

SHA-256 hash – With cryptographic hashing algorithms, similar inputs produce vastly different outputs. Using the SHA-256 hash generator creates an entirely different hashed output even if only one character is changed. This makes it much more difficult for hackers to reverse engineer the input values from the output values. As a result, SHA-256 is the hashing algorithm with Bitcoin cryptocurrency.

MD5 (Message Digest Algorithm) – MD5 is a cryptographic algorithm that will always produce an output of 128 bits (typically expressed as a 32 digit hexadecimal number) no matter the length of the input. It was one of the most widely used hashing algorithms but is now no longer recommended. MD5 is not collision resistant, meaning it's possible to produce the same hash with different inputs, which makes it a poor cryptographic hashing function.

MD5's downfall when it comes to passwords was that it was too fast and too popular. As a result, brute force attacks are more likely to be successful due to the thousands of inputs tested, and the popularity of the function makes it attractive to hackers. Today you can find the input to a MD5 hash in seconds by Googling it. Since many businesses already use MD5, they have taken to adding salt to it, creating a salted MD5 output.

MD5Crypt – MD5Crypt added extra functionality to MD5 to make it more resistant to brute force attacks. However, in 2012, the author of MD5Crypt, Poul-Hennin Kamp, declared it as insecure due to the speed of modern hardware.

SHA-1 – SHA-1 suffers from many of the same problems as MD5; it's very fast, it's also experienced collision attacks, and is now considered unsafe. Faster computations result in faster brute force attacks, making SHA-1 inherently insecure for storing passwords.

BCrypt – Unlike SHA-1 and MD5, Bcrypt is intentionally slow, which is a good thing when it comes to password security as it limits the attacker's ability to perform successful brute force attacks. A key aspect of hashing is that it should be a one-way form of encryption. It should be easy to go from the input to the output, but infeasible to find the input from the output. This slowed down hashing function makes cracking the hashes more impervious because it is time-consuming and uses a lot of computing power.

Web Application Security

Before you become a Spring Security master, you need to understand three important concepts:

- Authentication
- Authorization
- Servlet Filters

1. Authentication

First off, if you are running a typical (web) application, you need your users to *authenticate*. That means your application needs to verify if the user is *who* he claims to be, typically done with a username and password check.

2. Authorization

In simpler applications, authentication might be enough: As soon as a user authenticates, she can access every part of an application.

But most applications have the concept of permissions (or roles). Imagine: customers who have access to the public-facing frontend of your webshop, and administrators who have access to a separate admin area.

Both type of users need to login, but the mere fact of authentication doesn't say anything about what they are allowed to do in your system. Hence, you also need to check the permissions of an authenticated user, i.e. you need to *authorize* the user.

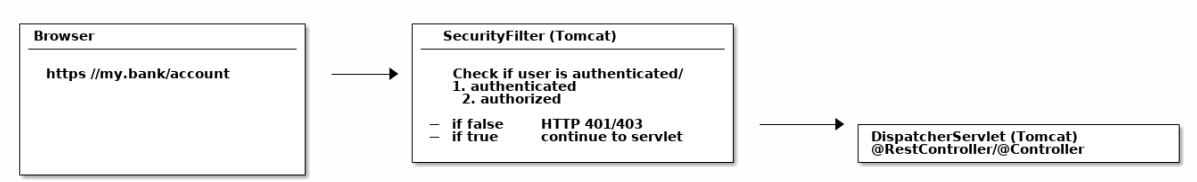
3. Servlet Filters

Spring web application is *just* one servlet: Spring's good old DispatcherServlet, that redirects incoming HTTP requests (e.g. from a browser) to your @Controllers or @RestControllers.

The thing is: There is no security hardcoded into that DispatcherServlet and you also very likely don't want to fumble around with a raw HTTP Basic Auth header in your @Controllers. Optimally, the authentication and authorization

should be done *before* a request hits your @Controllers.

Luckily, there's a way to do exactly this in the Java web world: you can put filters *in front* of servlets, which means you could think about writing a SecurityFilter and configure it in your Tomcat (servlet container/application server) to filter every incoming HTTP request before it hits your servlet.



A naive SecurityFilter

A SecurityFilter has roughly 4 tasks and a naive and overly-simplified implementation could look like this:

```
import javax.servlet.*;
import javax.servlet.http.HttpFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class SecurityServletFilter extends HttpFilter {

    @Override
    protected void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws IOException, Se

        UsernamePasswordToken token = extractUsernameAndPasswordFrom(request);  // (1)

        if (notAuthenticated(token)) {  // (2)
            // either no or wrong username/password
            // unfortunately the HTTP status code is called "unauthorized", instead of "unauthenticated"
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // HTTP 401.
            return;
        }

        if (notAuthorized(token, request)) { // (3)
            // you are logged in, but don't have the proper rights
            response.setStatus(HttpServletResponse.SC_FORBIDDEN); // HTTP 403
            return;
        }

        // allow the HttpRequest to go to Spring's DispatcherServlet
        // and @RestControllers/@Controllers.
        chain.doFilter(request, response); // (4)
    }

    private UsernamePasswordToken extractUsernameAndPasswordFrom(HttpServletRequest request) {
        // Either try and read in a Basic Auth HTTP Header, which comes in the form of user:password
        // Or try and find form login request parameters or POST bodies, i.e. "username=me" & "password="myPass"
        return checkVariousLoginOptions(request);
    }


    private boolean notAuthenticated(UsernamePasswordToken token) {
        // compare the token with what you have in your database...or in-memory...or in LDAP...
        return false;
    }

    private boolean notAuthorized(UsernamePasswordToken token, HttpServletRequest request) {
       // check if currently authenticated user has the permission/role to access this request's /URI
       // e.g. /admin needs a ROLE_ADMIN , /callcenter needs ROLE_CALLCENTER, etc.
       return false;
    }
}
```

1. First, the filter needs to extract a username/password from the request. It could be via a Basic Auth HTTP Header, or form fields, or a cookie, etc.
2. Then the filter needs to validate that username/password combination against *something*, like a database.
3. The filter needs to check, after successful authentication, that the user is authorized to access the requested URI.
4. If the request *survives* all these checks, then the filter can let the request go through to your DispatcherServlet, i.e. your @Controllers.

NOTE: basic access authentication is a method for an HTTP user agent (e.g. a web browser) to provide a user name and password when making a request. In basic HTTP authentication, a request contains a header field in the form of **Authorization: Basic <credentials>**, where credentials is the **Base64** encoding of ID and password joined by a single colon :.

FilterChains

Reality Check: While the above code compiles, it would sooner or later lead to one monster filter with a ton of code for various authentication and authorization mechanisms.

In the real-world, however, you would split this one filter up into *multiple* filters, that you then *chain* together.

For example, an incoming HTTP request would...

First, go through a LoginMethodFilter...Then, go through an AuthenticationFilter...Then, go through an AuthorizationFilter...Finally, hit your servlet.

This concept is called *FilterChain* and the last method call in your filter above is actually delegating to that very chain: `chain.doFilter(request, response);`

With such a filter (chain) you can basically handle every authentication or authorization problem there is in your application, without needing to change your actual application implementation (think: your @RestControllers / @Controllers).

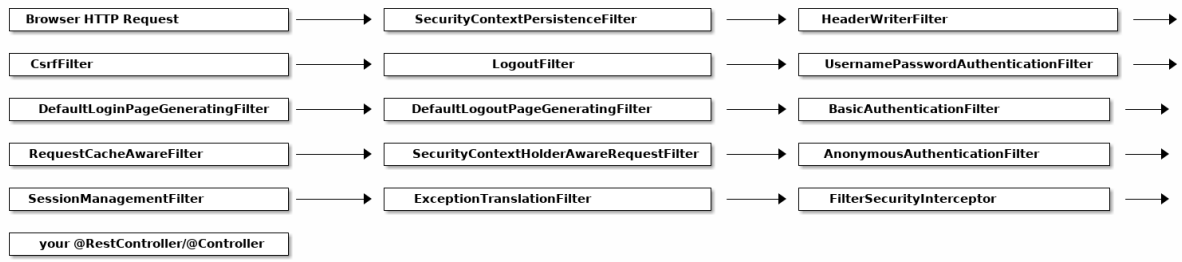
Spring's DefaultSecurityFilterChain

Let's assume you set up Spring Security correctly and then boot up your web application. You'll see the following log message:

```
2020-09-25 10:24:27.875  INFO 11116 --- [          main] o.s.s.web.DefaultSecurityFilterChain     : Creating filter chain: any
```

It looks like Spring Security does not just install *one* filter, instead it installs a whole filter chain consisting of 15 (!) different filters.

So, when an HTTPRequest comes in, it will go through *all* these 15 filters, before your request finally hits your @RestControllers. The order is important, too, starting at the top of that list and going down to the bottom.



Analyzing Spring's FilterChain

It would go too far to have a detailed look at every filter of this chain, but here's the explanations for a few of those filters. Feel free to look at Spring Security's source code to understand the other filters.

- **BasicAuthenticationFilter**: Tries to find a Basic Auth HTTP Header on the request and if found, tries to authenticate the user with the header's username and password.
- **UsernamePasswordAuthenticationFilter**: Tries to find a username/password request parameter/POST body and if found, tries to authenticate the user with those values.
- **DefaultLoginPageGeneratingFilter**: Generates a login page for you, if you don't explicitly disable that feature. THIS filter is why you get a default login page when enabling Spring Security.
- **DefaultLogoutPageGeneratingFilter**: Generates a logout page for you, if you don't explicitly disable that feature.
- **FilterSecurityInterceptor**: Does your authorization.

So with these couple of filters, Spring Security provides you a login/logout page, as well as the ability to login with Basic Auth or Form Logins, as well as a couple of additional goodies like the CsrfFilter, that we are going to have a look at later.

Those filters, for a large part, *are* Spring Security. Not more, not less. They do all the work. What's left for you is to *configure* how they do their work, i.e. which URLs to protect, which to ignore and what database tables to use for authentication.

Hence, we need to have a look at how to configure Spring Security, next.

How to configure Spring Security: WebSecurityConfigurerAdapter

With the latest Spring Security and/or Spring Boot versions, the way to configure Spring Security is by having a class that:

Is annotated with @EnableWebSecurity.Extends WebSecurityConfigurer, which basically offers you a configuration DSL/methods. With those methods, you can specify what URIs in your application to protect or what exploit protections to enable/disable.

Here's what a typical WebSecurityConfigurerAdapter looks like:

```
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter { // (1)

  @Override
  protected void configure(HttpSecurity http) throws Exception {  // (2)
      http
        .authorizeRequests()
          .antMatchers("/", "/home").permitAll() // (3)
          .anyRequest().authenticated() // (4)
          .and()
       .formLogin() // (5)
         .loginPage("/login") // (5)
         .permitAll()
         .and()
      .logout() // (6)
        .permitAll()
        .and()
      .httpBasic(); // (7)
  }
}
```

1. A normal Spring @Configuration with the @EnableWebSecurity annotation, extending from WebSecurityConfigurerAdapter.
2. By overriding the adapter's configure(HttpSecurity) method, you get a nice little DSL with which you can configure your FilterChain.
3. All requests going to `/` and `/home` are allowed (permitted) - the user does *not* have to authenticate. You are using an antMatcher, which means you could have also used wildcards (*, **, ?) in the string.
4. Any other request needs the user to be authenticated *first*, i.e. the user needs to login.

5. You are allowing form login (username/password in a form), with a custom loginPage (`/login`, i.e. not Spring Security's auto-generated one). Anyone should be able to access the login page, without having to log in first (permitAll; otherwise we would have a Catch-22!).
6. The same goes for the logout page
7. On top of that, you are also allowing Basic Auth, i.e. sending in an HTTP Basic Auth Header to authenticate.

What is important for now, is that *THIS* `configure` method is where you specify:

What URLs to protect (authenticated()) and which ones are allowed (permitAll()).Which authentication methods are allowed (formLogin(), httpBasic()) and how they are configured.In short: your application's complete security configuration.

Note: You wouldn't have needed to immediately override the adapter's configure method, because it comes with a pretty reasonable implementation - by default. This is what it looks like:

```
public abstract class WebSecurityConfigurerAdapter implements
                WebSecurityConfigurer<WebSecurity> {

    protected void configure(HttpSecurity http) throws Exception {
            http
                .authorizeRequests()
                    .anyRequest().authenticated()  // (1)
                    .and()
                .formLogin().and()   // (2)
                .httpBasic();  // (3)
        }
}
```

To access *any* URI (`anyRequest()`) on your application, you need to authenticate (authenticated()).Form Login (`formLogin()`) with default settings is enabled.As is HTTP Basic authentication (`httpBasic()`).

This default configuration is why your application is on lock-down, as soon as you add Spring Security to it.

Authentication with Spring Security

When it comes to authentication and Spring Security you have roughly three scenarios:

1. The **default**: You *can* access the (hashed) password of the user, because you have his details (username, password) saved in e.g. a database table.
2. **Less common**: You *cannot* access the (hashed) password of the user. This is the case if your users and passwords are stored *somewhere* else, like in a 3rd party identity management product offering REST services for authentication. Think: Atlassian Crowd.
3. **Also popular**: You want to use OAuth2 or "Login with Google/Twitter/etc." (OpenID), likely in combination with JWT.

Note: Depending on your scenario, you need to specify different @Beans to get Spring Security working, otherwise you'll end up getting pretty confusing exceptions (like a NullPointerException if you forgot to specify the PasswordEncoder).

Secenario 1: UserDetailsService: Having access to the user's password

Imagine you have a database table where you store your users. It has a couple of columns, but most importantly it has a username and password column, where you store the user's hashed(!) password.

```
create table users (id int auto_increment primary key, username varchar(255), password varchar(255));
```

In this case Spring Security needs you to define two beans to get authentication up and running.

1. A UserDetailsService.
2. A PasswordEncoder.

Specifying a UserDetailsService is as simple as this:

```
@Bean
public UserDetailsService userDetailsService() {
    return new MyDatabaseUserDetailsService(); // (1)
}
```

1. MyDatabaseUserDetailsService implements UserDetailsService, a very simple interface, which consists of one method returning a UserDetails object:

```
public class MyDatabaseUserDetailsService implements UserDetailsService {

        UserDetails loadUserByUsername(String username) throws UsernameNotFoundException { // (1)
         // 1. Load the user from the users table by username. If not found, throw UsernameNotFoundException.
         // 2. Convert/wrap the user to a UserDetails object and return it.
        return someUserDetails;
    }
}

public interface UserDetails extends Serializable { // (2)

    String getUsername();

    String getPassword();

    // <3> more methods:
    // isAccountNonExpired,isAccountNonLocked,
    // isCredentialsNonExpired,isEnabled
}
```

1. A UserDetailsService loads UserDetails via the user's username. Note that the method takes **only** one parameter: username (not the password).
2. The UserDetails interface has methods to get the (hashed!) password and one to get the username.
3. UserDetails has even more methods, like is the account active or blocked, have the credentials expired or what permissions the user has - but we won't cover them here.

Off-The-Shelf Implementations

Just a quick note: You can always implement the UserDetailsService and UserDetails interfaces yourself.

But, you'll also find off-the-shelf implementations by Spring Security that you can use/configure/extend/override instead.

1. **JdbcUserDetailsManager**, which is a JDBC(database)-based UserDetailsService. You can configure it to match your *user* table/column structure.
2. **InMemoryUserDetailsManager**, which keeps all userdetails in-memory and is great for testing.
3. **org.springframework.security.core.userdetail.User**, which is a sensible, default UserDetails implementation that you could use. That would mean potentially mapping/copying between your entities/database tables and this user class. Alternatively, you could simply make your entities implement the UserDetails interface.

Full UserDetails Workflow: HTTP Basic Authentication

Now think back to your HTTP Basic Authentication, that means you are securing your application with Spring Security and Basic Auth. This is what happens when you specify a UserDetailsService and try to login:

1. Extract the username/password combination from the HTTP Basic Auth header in a filter. You don't have to do anything for that, it will happen under the hood.

2. Call *your* MyDatabaseUserDetailsService to load the corresponding user from the database, wrapped as a UserDetails object, which exposes the user's hashed password.
3. Take the extracted password from the HTTP Basic Auth header, hash it *automatically* and compare it with the hashed password from your UserDetails object. If both match, the user is successfully authenticated.

That's all there is to it. But hold on, *how* does Spring Security hash the password from the client (step 3)? With what algorithm?

PasswordEncoders

Spring Security cannot magically guess your preferred password hashing algorithm. That's why you need to specify another @Bean, a *PasswordEncoder*. If you want to, say, use the BCrypt password hashing function (Spring Security's default) for *all your passwords*, you would specify this @Bean in your SecurityConfig.

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

OR What if you have multiple password hashing algorithms, because you have some legacy users whose passwords were stored with MD5 (don't do this), and newer ones with Bcrypt or even a third algorithm like SHA-256? Then you would use the following encoder:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

Summary: Having access to the user's password

The takeaway for this section is: if you are using Spring Security and have access to the user's password, then:

1. Specify a UserDetailsService. Either a custom implementation or use and configure one that Spring Security offers.
2. Specify a PasswordEncoder.

That is Spring Security authentication in a nutshell.

Secenario 2: AuthenticationProvider: Not having access to the user's password

Now, imagine that you are using Atlassian Crowd for centralized identity management. That means all your users and passwords for all your applications are stored in Atlassian Crowd and not in your database table anymore.

This has two implications:

1. You do *not have* the user passwords anymore in your application, as you cannot ask Crowd to just give you those passwords.
2. You do, however, have a REST API that you can login against, with your username and password. (A POST request to the `/rest/usermanagement/1/authentication` REST endpoint).

If that is the case, you cannot use a UserDetailsService anymore, instead you need to implement and provide an **AuthenticationProvider** @Bean.

```
@Bean
public AuthenticationProvider authenticationProvider() {
```

```
    return new AtlassianCrowdAuthenticationProvider();
}
```

An AuthenticationProvider consists primarily of one method and a naive implementation could look like this:

```
public class AtlassianCrowdAuthenticationProvider implements AuthenticationProvider {

        Authentication authenticate(Authentication authentication)  // (1)
                throws AuthenticationException {
            String username = authentication.getPrincipal().toString(); // (1)
            String password = authentication.getCredentials().toString(); // (1)

            User user = callAtlassianCrowdRestService(username, password); // (2)
            if (user == null) {                                    // (3)
                throw new AuthenticationException("could not login");
            }
            return new UserNamePasswordAuthenticationToken(user.getUsername(), user.getPassword(), user.getAuthorities()); // (
        }
            // other method ignored
}
```

1. Compared to the UserDetails load() method, where you only had access to the username, you now have access to the complete authentication attempt, *usually* containing a username and password.
2. You can do whatever you want to authenticate the user, e.g. call a REST-service.
3. If authentication failed, you need to throw an exception.
4. If authentication succeeded, you need to return a fully initialized UsernamePasswordAuthenticationToken. It is an implementation of the Authentication interface and needs to have the field authenticated be set to true (which the constructor used above will automatically set). We'll cover authorities in the next chapter.

Full AuthenticationProvider Workflow: HTTP Basic Authentication

Now think back to your HTTP Basic Authentication, that means you are securing your application with Spring Security and Basic Auth. This is what happens when you specify an AuthenticationProvider and try to login:

1. Extract the username/password combination from the HTTP Basic Auth header in a filter. You don't have to do anything for that, it will happen under the hood.
2. Call *your* AuthenticationProvider (e.g. AtlassianCrowdAuthenticationProvider) with that username and password for you to do the authentication (e.g. REST call) yourself.

There is no password hashing or similar going on, as you are essentially delegating to a third-party to do the actual username/password check. That's AuthenticationProvider authentication in a nutshell!

Summary: AuthenticationProvider

The takeaway for this section is: if you are using Spring Security and *do not* have access to the user's password, then *implement and provide an AuthenticationProvider @Bean*.

Authorization with Spring Security

What is Authorization?

Take your typical e-commerce web-shop. It likely consists of the following pieces:

- The web-shop itself. Let's assume its URL is `www.youramazinshop.com`.
- Maybe an area for callcenter agents, where they can login and see what a customer recently bought or where their parcel is. Its URL could be `www.youramazinshop.com/callcenter`.

- A separate admin area, where administrators can login and manage callcenter agents or other technical aspects (like themes, performance, etc.) of the web-shop. Its URL could be `www.youramazinshop.com/admin`.

This has the following implications, as simply authenticating users is not enough anymore:

- A customer obviously shouldn't be able to access the callcenter *or* admin area. He is only allowed to shop in the website.
- A callcenter agent shouldn't be able to access the admin area.
- Whereas an admin can access the web-shop, the callcenter area and the admin area.

Simply put, you want to allow different access for different users, depending on their *authorities* or *roles*.

What are Authorities? What are Roles?

Simple:

- An authority (in its simplest form) is just a string, it can be anything like: user, ADMIN, ROLE_ADMIN or 53cr37_r0l3.
- A role is an authority with a `ROLE_` prefix. So a role called `ADMIN` is the same as an authority called `ROLE_ADMIN`.

The distinction between roles and authorities is purely conceptual and something that often bewilders people new to Spring Security.

Why is there a distinction between roles and authorities?

Honestly, I've read the Spring Security documentation as well as a couple of related StackOverflow threads on this very question and I can't give you a definitive, *good* answer.

What are GrantedAuthorities? What are SimpleGrantedAuthorities?

Of course, Spring Security doesn't let you get away with *just* using Strings. There's a Java class representing your authority String, a popular one being SimpleGrantedAuthority.

```
public final class SimpleGrantedAuthority implements GrantedAuthority {

      private final String role;

      @Override
      public String getAuthority() {
              return role;
      }
}
```

1. UserDetailsService: Where to store and get authorities?

Assuming you are storing the users in your own application (think: UserDetailsService), you are going to have a Users table.

Now, you would simply add a column called "authorities" to it. I chose a simple string column here, though it could contain multiple, comma-separated values. Alternatively I could also have a completely separate table AUTHORITIES, but for the scope of this article this will do.

Note: Referring back to What are Authorities? What are Roles?: You save authorities, i.e. Strings, to the database. It so happens that these authorities start with the ROLE_ prefix, so, in terms of Spring Security these authorities are also roles.

Users

username	password	authorities
john@doe.com	{bcrypt}$2y$12$6t86Rpr3llMANhCUt26oUen2WhvXr/A89Xo9zJion8W7gWgZ/zA0C	ROLE_ADMIN
my@user.com	{sha256}5ffa39f5757a0dad5dfada519d02c6b71b61ab1df51b4ed1f3beed6abe0ff5f6	ROLE_USER

The only thing that's left to do is to adjust your UserDetailsService to read in that authorities column.

```
public class MyDatabaseUserDetailsService implements UserDetailsService {

  UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
     User user = userDao.findByUsername(username);
     List<SimpleGrantedAuthority> grantedAuthorities = user.getAuthorities()
          .map(authority -> new SimpleGrantedAuthority(authority)).collect(Collectors.toList()); // (1)
     return new org.springframework.security.core.userdetails
          .User(user.getUsername(), user.getPassword(), grantedAuthorities); // (2)
  }
}
```

1. You simply map whatever is inside your database column to a list of SimpleGrantedAuthorities. Done.
2. Again, we're using Spring Security's base implementation of UserDetails here. You could also use your own class implementing UserDetails here and might not even have to map then.

2. AuthenticationManager: Where to store and get authorities?

When the users comes from a third-party application, like Atlassian Cloud, you'll need to find out what concept they are using to support authorities. Atlassian Crowd had the concepts of "roles", but deprecated it in favour of "groups".

So, depending on the actual product you are using, you need to map this to a Spring Security authority, in your AuthenticationProvider.

```
public class AtlassianCrowdAuthenticationProvider implements AuthenticationProvider {

    Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
        String username = authentication.getPrincipal().toString();
        String password = authentication.getCredentials().toString();

        atlassian.crowd.User user = callAtlassianCrowdRestService(username, password); // (1)
        if (user == null) {
            throw new AuthenticationException("could not login");
        }
        return new UserNamePasswordAuthenticationToken(user.getUsername(), user.getPassword(),
            mapToAuthorities(user.getGroups())); // (2)
    }
            // other method ignored
}
```

1. Note: This is not *actual* Atlassian Crowd code, but serves its purpose. You authenticate against a REST service and get back a JSON User object, which then gets converted to an atlassian.crowd.User object.
2. That user can be a member of one or more groups, which are assumed to be just strings here. You can then simply map these groups to Spring's "SimpleGrantedAuthority".

Revisiting WebSecurityConfigurerAdapter for Authorities

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

        @Override
        protected void configure(HttpSecurity http) throws Exception {
        http
          .authorizeRequests()
            .antMatchers("/admin").hasAuthority("ROLE_ADMIN") // (1)
            .antMatchers("/callcenter").hasAnyAuthority("ROLE_ADMIN", "ROLE_CALLCENTER") // (2)
            .anyRequest().authenticated() // (3)
            .and()
         .formLogin()
           .and()
         .httpBasic();
        }


// SAME AS
/**
http
    .authorizeRequests()
      .antMatchers("/admin").hasRole("ADMIN") // (1)
      .antMatchers("/callcenter").hasAnyRole("ADMIN", "CALLCENTER") // (2)
*/
}
```

1. To access the `/admin` area you (i.e. the user) need to be authenticated *AND* have the authority (a simple string) ROLE_ADMIN.
2. To access the `/callcenter` area you need to be authenticated *AND* have either the authority ROLE_ADMIN *OR* ROLE_CALLCENTER.
3. For any other request, you do not need a specific role but still need to be authenticated.

hasAccess and SpEL

Last, but not least, the most powerful way to configure authorizations, is with the access method. It lets you specify pretty much any valid SpEL expressions.

```
http
    .authorizeRequests()
      .antMatchers("/admin").access("hasRole('admin') and hasIpAddress('192.168.1.0/24') and @myCustomBean.checkAccess(authenti
```

Checking that the user has ROLE_ADMIN, with a specific IP address as well as a custom bean check.

To get a full overview of what's possible with Spring's Expression-Based Access Control, have a look at the official documentation.

Common Exploit Protections

There is a variety of common attacks that Spring Security helps you to protect against. It starts with timing attacks (i.e. Spring Security will always hash the supplied password on login, even if the user does not exist) and ends up with protections against cache control attacks, content sniffing, click jacking, cross-site scripting and more.

Spring Security & Spring Framework

Until now, you only specified security configurations on the *web tier* of your application. You protected certain URLs with antMatcher or regexMatchers with the WebSecurityConfigurerAdapter's DSL. That is a perfectly fine and standard

approach to security.

In addition to protecting your web tier, there's also the idea of "defense in depth". That means in addition to protecting URLs, you might want to protect your business logic itself. Think: your @Controllers, @Components, @Services or even @Repositories. In short, your Spring beans.

Method Security

That approach is called `method security` and works through annotations that you can basically put on any public method of your Spring beans. You also need to explicitly enable method security by putting the @EnableGlobalMethodSecurity annotation on your ApplicationContextConfiguration.

```
@Configuration
@EnableGlobalMethodSecurity(
  prePostEnabled = true, // (1)
  securedEnabled = true, // (2)
  jsr250Enabled = true) // (3)
public class YourSecurityConfig extends WebSecurityConfigurerAdapter{
}
```

1. The prePostEnabled property enables support for Spring's `@PreAuthorize` and `@PostAuthorize` annotations. Support means, that Spring will ignore this annotation unless you set the flag to true.
2. The securedEnabled property enables support for the `@Secured` annotation. Support means, that Spring will ignore this annotation unless you set the flag to true.
3. The jsr250Enabled property enables support for the `@RolesAllowed` annotation. Support means, that Spring will ignore this annotation unless you set the flag to true.

What is the difference between @PreAuthorize, @Secured and @RolesAllowed?

@Secured and **@RolesAllowed** are basically the same, though @Secured is a Spring-specific annotation coming with the spring-security-core dependency and @RolesAllowed is a standardised annotation, living in the javax.annotation-api dependency. Both annotations take in an authority/role string as value.

@PreAuthorize/**@PostAuthorize** are also (newer) Spring specific annotations and more powerful than the above annotations, as they can contain not only authorities/roles, but also *any* valid SpEL expression.

```
+-----------------------------------------------+-----------------------------------------------------------+-------------------
|                                               |                         @Secured                          |
+-----------------------------------------------+-----------------------------------------------------------+-------------------
| Spring EL expressions                         | Does'nt supports.                                         | Supports
+-----------------------------------------------+-----------------------------------------------------------+-------------------
| Multiple roles conjunctions with AND operator | Does'nt supports.(If there are multiple roles defined     | Supports
|                                               |they will be automatically combined with OR operator)      |
+-----------------------------------------------+-----------------------------------------------------------+-------------------
| To enable annotation                          | Add following line to spring-security.xml                 | Add following line
|                                               | <global-method-security secured-annotations="enabled" />  | <global-method-sec
+-----------------------------------------------+-----------------------------------------------------------+-------------------
| Example                                       | @Secured({ROLE_ADMIN , ROLE_USER})                        | @PreAuthorize("has
|                                               | public void addUser(UserInfo user){...}                   | public void addUse
+-----------------------------------------------+-----------------------------------------------------------+-------------------
```

Lastly, all these annotations will raise an `AccessDeniedException` if you try and access a protected method with an insufficient authority/role.

```
@Service
public class SomeService {

    @Secured("ROLE_CALLCENTER") // (1)
```

```
    // == @RolesAllowed("ADMIN")
    public BankAccountInfo get(...) {

    }

    @PreAuthorize("isAnonymous()") // (2)
    // @PreAuthorize("#contact.name == principal.name")
    // @PreAuthorize("ROLE_ADMIN")
    public void trackVisit(Long id);

    }
}
```

1. As mentioned, @Secured takes an authority/role as parameter. @RolesAllowed, likewise. **Note**: Remember that `@RolesAllowed("ADMIN")` will check for a granted authority `ROLE_ADMIN`.
2. As mentioned, @PreAuthorize takes in authorities, but also any valid SpEL expression. For a list of common built-in security expressions like `isAnonymous()` above, as opposed to writing your own SpEL expressions, check out the official documentation.

Spring Security & Spring Web MVC

As for the integration with Spring WebMVC, Spring Security allows you to do a couple of things:

1. In addition to antMatchers and regexMatchers, you can also use mvcMatchers. The difference is, that while antMatchers and regexMatchers basically match URI strings with wildcards, mvcMatchers behave *exactly* like @RequestMappings.
2. Injection of your currently authenticated principal into a @Controller/@RestController method.
3. Injection of your current session CSRFToken into a @Controller/@RestController method.
4. Correct handling of security for async request processing.

```
@Controller
public class MyController {

    @RequestMapping("/messages/inbox")
    public ModelAndView findMessagesForUser(@AuthenticationPrincipal CustomUser customUser, CsrfToken token) {  // (1) (2)

    // .. find messages for this user and return them ...
    }
}
```

1. @AuthenticationPrincipal will inject a principal if a user is authenticated, or null if no user is authenticated. This principal is the object coming from your UserDetailsService/AuthenticationManager!
2. Or you could inject the current session CSRFToken into each method.

If you are not using the @AuthenticationPrincipal annotation, you would have to fetch the principal yourself, through the SecurityContextHolder. A technique often seen in legacy Spring Security applications.

```
@Controller
public class MyController {

    @RequestMapping("/messages/inbox")
    public ModelAndView findMessagesForUser(CsrfToken token) {
         SecurityContext context = SecurityContextHolder.getContext();
         Authentication authentication = context.getAuthentication();

         if (authentication != null && authentication.getPrincipal() instanceof UserDetails) {
             CustomUser customUser = (CustomUser) authentication.getPrincipal();
```

```
            // .. find messages for this user and return them ...
        }

        // todo
    }
}
```

Spring Security & Thymeleaf

Spring Security integrates well with Thymeleaf. It offers a special Spring Security Thymeleaf dialect, which allows you to put security expressions directly into your Thymeleaf HTML templates.

```
<div sec:authorize="isAuthenticated()">
  This content is only shown to authenticated users.
</div>
<div sec:authorize="hasRole('ROLE_ADMIN')">
  This content is only shown to administrators.
</div>
<div sec:authorize="hasRole('ROLE_USER')">
  This content is only shown to users.
</div>
```

OAuth concepts

- **Resource owner** An entity capable of authorizing access to a protected resource. When the resource owner is a person, it is called an user.
- **OAuth client** A third-party application that wants access to the private resources of the resource owner. The OAuth client can make protected resource requests on behalf of the resource owner after the resource owner grants it authorization. OAuth 2.0 introduces two types of clients: confidential and public. Confidential clients are registered with a client secret, while public clients are not.
- **OAuth server** Known as the **Authorization server** in OAuth 2.0. The server that gives OAuth clients scoped access to a protected resource on behalf of the resource owner. The server issues an access token to the OAuth client after it successfully does the following actions:
  - Authenticates the resource owner.
  - Validates a request or an authorization grant.
  - Obtains resource owner authorization.

  An authorization server can also be the resource server.
- **Scope** A property requested by the OAuth client, to specify the scope of the access request. The scope is used by the caller to tag the intended use of the token. The authorization server can use the scope response parameter to tell the client the scope of the access token that was issued. Scopes are usually shown on the consent page, so that a user can understand the client's intended use of the token. Common scopes include profile and email.
- **Access token** A string that represents authorization granted to the OAuth client by the resource owner. This string represents specific scopes and durations of access. It is granted by the resource owner and enforced by the OAuth server.
- **Bearer token** Token issued from the token endpoint. This includes an access token and potentially a refresh token. See http://tools.ietf.org/html/rfc6750 for more information on bearer tokens.
- **Protected resource** A restricted resource that can be accessed from the OAuth server using authenticated requests.
- **Resource server** The server that hosts the protected resources. It can use access tokens to accept and respond to protected resource requests. The resource server might be the same server as the authorization server.
- **Authorization grant** A grant that represents the resource owner authorization to access its protected resources. OAuth clients use an authorization grant to obtain an access token. There are four authorization grant types:

authorization code, implicit, resource owner password credentials, and client credentials.

- **Authorization code** A code that the Authorization server generates when the resource owner authorizes a request.
- **Refresh token** A string that is used to obtain a new access token. A refresh token is optionally issued by the authorization server to the OAuth client together with an access token. The OAuth client can use the refresh token to request another access token that is based on the same authorization, without involving the resource owner again.

JWT(Json Web Token) is a token format. It is digitally-signed, self-contained, and compact. It provides a convenient mechanism for transferring data. JWT is not inherently secure, but the use of JWT can ensure the authenticity of the message so long as the signature is verified and the integrity of the payload can be guaranteed. JWT is often used for stateless authentication in simple use cases involving non-complex systems.

JWT and OAuth2 are entirely different and serve different purposes, but they are compatible and can be used together. The OAuth2 protocol does not specify the format of the tokens, therefore JWTs can be incorporated into the usage of OAuth2.

For example, the access_token returned from the OAuth2 Authorization Server could be a JWT carrying additional information in the payload. This could potentially increase performance by reducing round trips for the required information between the Resource Server and the Authorization Server. This is a good use case for incorporating JWT into OAuth2 implementations when transparent tokens are acceptable - there are scenarios requiring token opacity where this is not optimal.

Another common way to use JWT in conjunction with OAuth2 is to issue two tokens: a reference token as access_token, and a JWT containing identity information in addition to that access token. In use cases where this implementation seems necessary, it is probably worth looking into OpenID Connect - an extension built upon OAuth2 and provides additional standardizations, including having an access_token and an id_token.

What is JSON Web Token?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on *signed* tokens. Signed tokens can verify the *integrity* of the claims contained within it, while encrypted tokens *hide* those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

- **Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
- **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

What is the JSON Web Token structure?

In its compact form, JSON Web Tokens consist of three parts separated by dots (`.`), which are:

- Header
- Payload

- Signature

Therefore, a JWT typically looks like the following.

`xxxxx.yyyyy.zzzzz`

Let's break down the different parts.

Header

The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.

- **Registered claims**: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), and others.

  > Notice that the claim names are only three characters long as JWT is meant to be compact.

- **Public claims**: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.
- **Private claims**: These are the custom claims created to share information between parties that agree on using them and are neither *registered* or *public* claims.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

> Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
```

```
  base64UrlEncode(payload),
  secret)
```

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

Putting all together

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2
```

Spring Security JWT implementation

How to use JWT: username + password + JSON map+ Base64 + key + expiration date

How it works: When user login with username + password for the first time, the system will exchange back the access token, which this token represents a JSON map containing all user information, such as user profiles, roles, and privileges, encoded with Base64 and signed with a private key. After receiving the token, the user can keep using this until it is expired by itself.

Benefits: The JWT token is stateless. This means that the server can decrypt the token into the user information state and there is no need for additional looking up from the database with this token. This is a huge benefit in reducing the load on the server. This approach is widely used all over the world.

```
# Technologies

- Spring boot 2
- Spring Secruity
- JWT jjwt
- Mysql
- Database JPA
```

```
springboot2-jwt
|-- README.md
|-- pom.xml
|-- src
|   `-- main
|       |-- java
|       |   `-- com
|       |       `-- sma
|       |           `-- security
|       |               |-- Application.java
|       |               |-- config
|       |               |   |-- JwtAuthenticationEntryPoint.java
|       |               |   |-- JwtRequestFilter.java
|       |               |   |-- JwtTokenService.java
|       |               |   `-- WebSecurityConfig.java
|       |               |-- controller
|       |               |   |-- JwtTokenController.java
|       |               |   `-- TestController.java
|       |               |-- dao
|       |               |   `-- UserRepository.java
|       |               |-- exception
|       |               |   `-- GlobalExceptionHandler.java
|       |               |-- json
```

```
|       |               |   |-- JwtResponse.java
|       |               |   `-- UserDTO.java
|       |               |-- model
|       |               |   `-- UserEntity.java
|       |               `-- service
|       |                   `-- JwtUserDetailsService.java
|       `-- resources
|           `-- application.properties
```

Single Sign On

Single sign-on (**SSO**) is an authentication scheme that allows a user to log in with a single ID and password to any of several related, yet independent, software systems. It is often accomplished by using the Lightweight Directory Access Protocol (LDAP) and stored LDAP databases on (directory) servers.[1] A simple version of single sign-on can be achieved over IP networks using cookies but only if the sites share a common DNS parent domain.[2]

For clarity, a distinction should be made between Directory Server Authentication and single sign-on: Directory Server Authentication refers to systems requiring authentication for each application but using the same credentials from a directory server, whereas single sign-on refers to systems where a single authentication provides access to multiple applications by passing the authentication token seamlessly to configured applications.

Conversely, **single sign-off** or **single log-out** (**SLO**) is the property whereby a single action of signing out terminates access to multiple software systems.

As different applications and resources support different authentication mechanisms, single sign-on must internally store the credentials used for initial authentication and translate them to the credentials required for the different mechanisms.

Other shared authentication schemes, such as OpenID and OpenID Connect, offer other services that may require users to make choices during a sign-on to a resource, but can be configured for single sign-on if those other services (such as user consent) are disabled.[3] An increasing number of federated social logons, like Facebook Connect, do require the user to enter consent choices upon first registration with a new resource, and so are not always single sign-on in the strictest sense.

Benefits

Benefits of using single sign-on include:

- Mitigate risk for access to 3rd-party sites (user passwords not stored or managed externally)
- Reduce password fatigue from different username and password combinations
- Reduce time spent re-entering passwords for the same identity
- Reduce IT costs due to lower number of IT help desk calls about passwords

SSO shares centralized authentication servers that all other applications and systems use for authentication purposes and combines this with techniques to ensure that users do not have to actively enter their credentials more than once.

Criticism

The term reduced sign-on (RSO) has been used by some to reflect the fact that single sign-on is impractical in addressing the need for different levels of secure access in the enterprise, and as such more than one authentication server may be necessary.

As single sign-on provides access to many resources once the user is initially authenticated ("keys to the castle"), it increases the negative impact in case the credentials are available to other people and misused. Therefore, single sign-on requires an increased focus on the protection of the user credentials, and should ideally be combined with strong authentication methods like smart cards and one-time password tokens.

Single sign-on also makes the authentication systems highly critical; a loss of their availability can result in denial of access to all systems unified under the SSO. SSO can be configured with session failover capabilities in order to maintain the system operation. Nonetheless, the risk of system failure may make single sign-on undesirable for systems to which access must be guaranteed at all times, such as security or plant-floor systems.

Furthermore, the use of single-sign-on techniques utilizing social networking services such as Facebook may render third party websites unusable within libraries, schools, or workplaces that block social media sites for productivity reasons. It can also cause difficulties in countries with active censorship regimes, such as China and its "Golden Shield Project," where the third party website may not be actively censored, but is effectively blocked if a user's social login is blocked.

Privacy

As originally implemented in Kerberos and SAML, single sign-on did not give users any choices about releasing their personal information to each new resource that the user visited. This worked well enough within a single enterprise, like MIT where Kerberos was invented, or major corporations where all of the resources were internal sites. However, as federated services like Active Directory Federation Services proliferated, the user's private information was sent out to affiliated sites not under control of the enterprise that collected the data from the user. Since privacy regulations are now tightening with legislation like the GDPR, the newer methods like OpenID Connect have started to become more attractive; for example MIT, the originator of Kerberos, now supports OpenID Connect.

Email address

Single sign-on in theory can work without revealing identifying information like email address to the relying party (credential consumer), but many credential providers do not allow users to configure what information is passed on to the credential consumer. As of 2019, Google and Facebook sign-in do not require users to share email address with the credential consumer. 'Sign in with Apple' introduced in iOS 13 allows user to request a unique relay email each time the user signs up for a new service, thus reducing the likelihood of account linking by the credential consumer.

Common configurations

Kerberos-based

- Initial sign-on prompts the user for credentials, and gets a Kerberos ticket-granting ticket (TGT).
- Additional software applications requiring authentication, such as email clients, wikis, and revision-control systems, use the ticket-granting ticket to acquire service tickets, proving the user's identity to the mailserver / wiki server / etc. without prompting the user to re-enter credentials.

Windows environment - Windows login fetches TGT. Active Directoryaware applications fetch service tickets, so the user is not prompted to re-authenticate.

Unix/Linux environment - Log in via Kerberos PAM modules fetches TGT. Kerberized client applications such as Evolution, Firefox, and SVN use service tickets, so the user is not prompted to re-authenticate.

Smart-card-based

Initial sign-on prompts the user for the smart card. Additional software applications also use the smart card, without prompting the user to re-enter credentials. Smart-card-based single sign-on can either use certificates or passwords stored on the smart card.

Integrated Windows Authentication

Integrated Windows Authentication is a term associated with Microsoft products and refers to the SPNEGO, Kerberos, and NTLMSSP authentication protocols with respect to SSPI functionality introduced with Microsoft Windows 2000 and included with later Windows NTbased operating systems. The term is most commonly used to refer to the automatically authenticated connections between Microsoft Internet Information Services and Internet Explorer. Cross-platform Active Directory integration vendors have extended the Integrated Windows Authentication paradigm to Unix (including Mac) and GNU/Linux systems.

Security Assertion Markup Language

Security Assertion Markup Language (SAML) is an XMLbased method for exchanging user security information between an SAML identity provider and a SAML service provider. SAML 2.0 supports W3C XML encryption and service-provider–initiated web browser single sign-on exchanges. A user wielding a user agent (usually a web browser) is called the subject in SAML-based single sign-on. The user requests a web resource protected by a SAML service provider. The service provider, wishing to know the identity of the user, issues an authentication request to a SAML identity provider through the user agent. The identity provider is the one that provides the user credentials. The service provider trusts the user information from the identity provider to provide access to its services or resources.

Generate SSL certificates and secure the embedded servlet container

SSL certificates are what enable websites to move from **HTTP** to **HTTPS**, which is more secure. An SSL certificate is a data file hosted in a website's **origin server**. SSL certificates make **SSL/TLS encryption** possible, and they contain the website's public key and the website's identity, along with related information. Devices attempting to communicate with the origin server will reference this file to obtain the public key and verify the server's identity. The private key is kept secret and secure.

What is SSL?

SSL, more commonly called TLS, is a protocol for encrypting Internet traffic and verifying server identity. Any website with an HTTPS web address uses SSL/TLS. See **What is SSL**? and **What is TLS**? to learn more.

What information does an SSL certificate contain?

SSL certificates include:

- The **domain name** that the certificate was issued for
- Which person, organization, or device it was issued to
- Which certificate authority issued it
- The certificate authority's digital signature
- Associated subdomains
- Issue date of the certificate
- Expiration date of the certificate
- The public key (the private key is kept secret)

The public and private keys used for SSL are essentially long strings of characters used for encrypting and decrypting data. Data encrypted with the public key can only be decrypted with the private key, and vice versa.

Why do websites need an SSL certificate?

A website needs an SSL certificate in order to keep user data secure, verify ownership of the website, prevent attackers from creating a fake version of the site, and gain user trust.

Encryption: SSL/TLS encryption is possible because of the public-private key pairing that SSL certificates facilitate. Clients (such as web browsers) get the public key necessary to open a TLS connection from a server's SSL certificate.

Authentication: SSL certificates verify that a client is talking to the correct server that actually owns the domain. This helps prevent domain **spoofing** and other kinds of attacks.

HTTPS: Most crucially for businesses, an SSL certificate is necessary for an HTTPS web address. HTTPS is the secure form of HTTP, and HTTPS websites are websites that have their traffic encrypted by SSL/TLS.

In addition to securing user data in transit, HTTPS makes sites more trustworthy from a user's perspective. Many users won't notice the difference between an http:// and an https:// web address, but most browsers have **started tagging HTTP sites as "not secure"** in more noticeable ways, attempting to provide incentive for switching to HTTPS and increasing security.

How does a website obtain an SSL certificate?

For an SSL certificate to be valid, domains need to obtain it from a certificate authority (CA). A CA is an outside organization, a trusted third party, that generates and gives out SSL certificates. The CA will also digitally sign the certificate with their own private key, allowing client devices to verify it. Most, but not all, CAs will charge a fee for issuing an SSL certificate.

Once the certificate is issued, it needs to be installed and activated on the website's origin server. Web hosting services can usually handle this for website operators. Once it's activated on the origin server, the website will be able to load over HTTPS and all traffic to and from the website will be encrypted and secure.

What is a self-signed SSL certificate?

Technically, anyone can create their own SSL certificate by generating a public-private key pairing and including all the information mentioned above. Such certificates are called self-signed certificates because the digital signature used, instead of being from a CA, would be the website's own private key.

But with self-signed certificates, there's no outside authority to verify that the origin server is who it claims to be. Browsers don't consider self-signed certificates trustworthy and may still mark sites with one as "not secure," despite the https:// URL. They may also terminate the connection altogether, blocking the website from loading.

Implementation:

To enable SSL or HTTPS for Spring Boot web application, puts the certificate file .p12 or .jks in the resources folder, and declares the server.ssl.* values in the application.properties

```
#application.properties
# SSL
server.port=8443
server.ssl.key-store=classpath:cert.p12
server.ssl.key-store-password=123456

# JKS or PKCS12
server.ssl.keyStoreType=PKCS12

# Spring Security
# security.require-ssl=true
```

We will use the JDK's keytool to generate a self-sign certificate in PKCS12 format.

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore cert.p12 -validity 365

Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  spring
What is the name of your organizational unit?
  [Unknown]:  training
What is the name of your organization?
  [Unknown]:  spring
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=spring, OU=training, O=spring, L=Unknown, ST=Unknown, C=Unknown correct?
  [no]:  yes
```

Redirect all traffic from port 8080 to 8443.

```
// spring boot 2.x
    @Bean
    public ServletWebServerFactory servletContainer() {
        TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory() {
            @Override
            protected void postProcessContext(Context context) {
                SecurityConstraint securityConstraint = new SecurityConstraint();
                securityConstraint.setUserConstraint("CONFIDENTIAL");
                SecurityCollection collection = new SecurityCollection();
                collection.addPattern("/*");
                securityConstraint.addCollection(collection);
                context.addConstraint(securityConstraint);
            }
        };
        tomcat.addAdditionalTomcatConnectors(redirectConnector());
        return tomcat;
    }

    private Connector redirectConnector() {
        Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
        connector.setScheme("http");
        connector.setPort(8080);
        connector.setSecure(false);
        connector.setRedirectPort(8443);
        return connector;
    }
```