**What is Single sign-on?**

Single sign-on (or SSO) allow users to use a single set of credentials to login into multiple related yet independent web applications. SSO also includes not asking users to login again into application B if they have already logged into application A given that A and B use SSO. SSO is achieved by implementing a centralised login system that handles authentication of users and share that information with applications that need that data.

The most common example of SSO that most of us use is by Google. When you use login to any Google service, you are redirected to [https://accounts.google.com](https://accounts.google.com/) for authentication. For example, if you go to gmail.com you will be redirected to [https://accounts.google.com](https://accounts.google.com/) for login. Same is the case when you to try to sign in to Youtube. After successful authentication, users are redirected to the application.

The requirement for SSO is common in enterprises where different applications come into the system at different times . Some applications are developed by different business units in isolation or some come through acquisitions. Each system come packaged with their own identity systems. Having different identity systems not only make it difficult for end users to use the applications but it also makes it difficult for an enterprise to link multiple identities to a user so that they can view a user as a single customer. Having a centralised login system is the first step to gain better understanding of the end user.

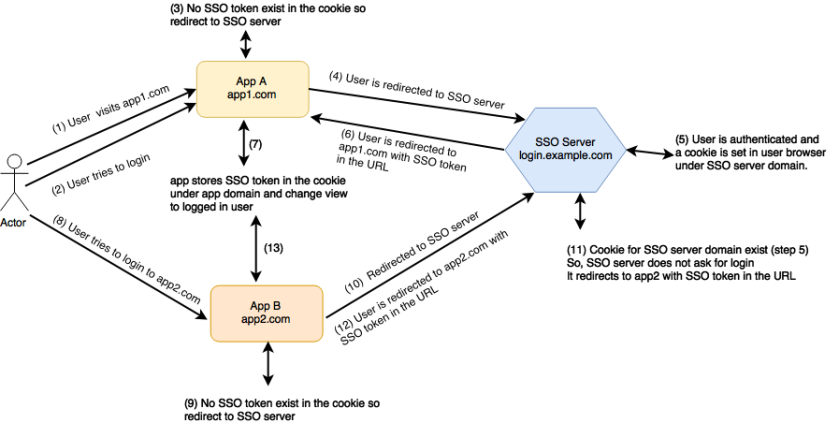
**Benefits of Single sign-on**

Many enterprises are adopting SSO because of the benefits it offers. Some of the benefits are mentioned below:

1. **Reduced IT support cost**: Gartner has reported that 20%-50% of the support tickets are password resets. In a report, Forester said it costs $70 to solve a password reset request. This cost is too high. Even if we make it 10% i.e. $7 it is still a high number for such a simple task. SSO simplifies it for the user by making them use one credential for set of applications. This can significantly reduce help desk calls. Hence, reducing overall cost as well.
2. **Improved User experience**: How frustrating it is to keep entering username and password to access different services? Not only this is a bad experience for the user but it also kills their productivity. Some people call it [password fatigue](https://en.wikipedia.org/wiki/Password_fatigue). SSO stitches together different services so that user can navigate between services seamlessly.
3. **Better understanding of the customer**: An enterprise can better understand a customer if they can track his/her usage across different products in their suite. This can only be achieved if they have a single view of the customer. SSO can do that for an enterprise.
4. **Better security**: With SSO, you have a single centralised server that manages user identity. It is the responsibility of the SSO users to securely store credentials. Individual services do not need to manage the credentials. Hence, reduce the attack surface area.

**How does SSO authentication works?**

This section is not talking about a specific SSO server implementation. We will cover that when we will look at Spring Security OAuth. In this section, we will understand the basic idea behind most SSO systems. Below is a diagram that depicts the SSO flow. We have two applications app1.com and app2.com. There is a centralised SSO server login.example.com.



This is what happens in the SSO flow:

1. A user goes to the app1.com for the first time. As user is not logged in, a login button is available to the user. User clicks the login button and user is redirected to the SSO server.
2. User enters credentials on the login page rendered by the SSO server. SSO server validates the credentials and generates a SSO token. SSO server sets the SSO token in the cookie for future login attempts by the user.
3. SSO server redirects user to the app1.com. In the redirect URL, it also appends SSO token as the query parameter.
4. app1.com saves the token in its cookie and change view to the logged in user. app1.com can get information about the user either by querying the SSO server or if token is a JWT token then it can get basic user information from the token itself.
5. Now, the same user tries to access app2.com. As an application can only access cookie for the same origin it has no knowledge that user is logged in to app1.com. So, user will still be shown login button on app2.
6. User clicks the login button and then user is redirected to the SSO server. SSO server sees that it already has a cookie set so it will immediately redirect the user to app2.com with SSO token appended in the URL as query parameter.
7. app2.com stores the token in the cookie and change its view to logged in user.

At the end of this process there will be three cookies set in the user browser each for app1.com, app2.com, and login.example.com domains.

**SSO implementations**

There are variety of SSO implementations that exist today. Some of the popular one include Facebook Connect, Open Id Connect, CAS, Kerbos, SAML, etc. You can find [list of Single sign-on implementations on Wikipedia](https://en.wikipedia.org/wiki/List_of_single_sign-on_implementations).

**Setting your own SSO server with Spring Boot and Spring Security OAuth**

Spring Boot along with Spring Security OAuth makes it easy to set up your own SSO server. We will use the setup that we discussed while explaining SSO flow.

1. app1 and aap2 will be the two applications using SSO
2. sso-server will be the centeralized login system

When user will try to login into app1 or app2 they will be redirected to the sso-server

**Create sso-server application**

We will start by creating sso-server application. The easiest way to create a Spring Boot application is to Spring Initialzr project available at [https://start.spring.io](https://start.spring.io/). You can either use the web interface or tool like cURL to create the project. Below is the cURL command, to create the sso-server project.

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ curl <https://start.spring.io/starter.zip> \  -d dependencies=web,cloud-oauth2 \  -d groupId=com.shekhargulati -d artifactId=sso-server -d name=sso-server \  -d description="SSO Server" -d baseDir=sso-server \  -o sso-server.zip |

Now, unzip the source code zip and import it in your favourite IDE.

**Enable authorization server**

To make a Spring Boot application act as an authorization server you have to mark it with @EnableAuthorizationServer annotation as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;    @SpringBootApplication  @EnableAuthorizationServer  public class SsoServerApplication {    public static void main(String[] args) {  SpringApplication.run(SsoServerApplication.class, args);  }    } |

This is what happens when you add @EnableAuthorizationServer annotation.

1. This annotation imports two configuration classes — AuthorizationServerEndpointsConfiguration and AuthorizationServerSecurityConfiguration.
2. The AuthorizationServerEndpointsConfiguration configuration class create beans for three REST controllers — AuthorizationEndpoint, TokenEndpoint, and CheckTokenEndpoint. These three controllers provide implementations for endpoints specified in OAuth2 specification.
3. The AuthorizationServerSecurityConfiguration configuration class configures Spring Security for OAuth endpoints.

When you will start the application, you will see in the logs various oauth specific URL mappings as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 018-01-27 16:49:27.391 INFO 46210 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/authorize]}"  2018-01-27 16:49:27.394 INFO 46210 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/authorize],methods=[POST],params=[user\_oauth\_approval]}"  2018-01-27 16:49:27.396 INFO 46210 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/token],methods=[GET]}"  2018-01-27 16:49:27.397 INFO 46210 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/token],methods=[POST]}"  2018-01-27 16:49:27.421 INFO 46210 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/check\_token]}"  2018-01-27 16:49:27.423 INFO 46210 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/confirm\_access]}"  2018-01-27 16:49:27.423 INFO 46210 --- [ main] .s.o.p.e.FrameworkEndpointHandlerMapping : Mapped "{[/oauth/error]}" |

**Configure Spring Security for login**

We will define our own Spring Security configuration class that will use form based login instead of the basic authentication mechanism used with default configuration. To do that, update the SsoServerApplication as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.context.annotation.Configuration;  import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  import org.springframework.security.config.annotation.web.builders.HttpSecurity;  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  import org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;    @SpringBootApplication  @EnableAuthorizationServer  public class SsoServerApplication {    public static void main(String[] args) {  SpringApplication.run(SsoServerApplication.class, args);  }    @Configuration  protected static class LoginConfig extends WebSecurityConfigurerAdapter {    @Override  protected void configure(HttpSecurity http) throws Exception {  http.requestMatchers()  .antMatchers("/login", "/oauth/authorize")  .and()  .authorizeRequests()  .anyRequest().authenticated()  .and()  .formLogin().permitAll();  }    @Override  protected void configure(AuthenticationManagerBuilder auth) throws Exception {  auth.inMemoryAuthentication()  .withUser("user")  .password("password")  .roles("USER");  }  }  } |

In the LoginConfig shown above, we are explicitly saying that we want to use this security configuration for only two URLs /login and /oauth.authorize. All other urls will remain unaffected. This is very important to keep in mind.

Restart the application now you will be asked to login using a default form created by Spring Security. You can login using user/password credentials.

**Create OAuth2Config**

So far we have not explicitly specified OAuth configuration. Create a new static inner class OAuth2Config to specify OAuth2 configuration as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.context.annotation.Configuration;  import org.springframework.security.authentication.AuthenticationManager;  import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  import org.springframework.security.config.annotation.web.builders.HttpSecurity;  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  import org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsServiceConfigurer;  import org.springframework.security.oauth2.config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;  import org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;  import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;  import org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerEndpointsConfigurer;  import org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerSecurityConfigurer;    @SpringBootApplication  @EnableResourceServer  public class SsoServerApplication {    public static void main(String[] args) {  SpringApplication.run(SsoServerApplication.class, args);  }    @Configuration  protected static class LoginConfig extends WebSecurityConfigurerAdapter {    @Override  protected void configure(HttpSecurity http) throws Exception {  http.requestMatchers()  .antMatchers("/login", "/oauth/authorize")  .and()  .authorizeRequests()  .anyRequest().authenticated()  .and()  .formLogin().permitAll();  }    @Override  protected void configure(AuthenticationManagerBuilder auth) throws Exception {  auth.inMemoryAuthentication()  .withUser("user")  .password("password")  .roles("USER");  }  }    @Configuration  @EnableAuthorizationServer  protected static class OAuth2Config extends AuthorizationServerConfigurerAdapter {  @Autowired  private AuthenticationManager authenticationManager;    @Override  public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  clients.inMemory()  .withClient("foo")  .secret("bar")  .authorizedGrantTypes("authorization\_code", "refresh\_token", "password")  .scopes("user\_info")  .autoApprove(true);  }    @Override  public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws Exception {  oauthServer  .tokenKeyAccess("permitAll()")  .checkTokenAccess("isAuthenticated()");  }    @Override  public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {  endpoints.authenticationManager(authenticationManager);  }  }  } |

**Set a different context root**

In sso-server project application.properties set the context root of the application as shown below.

|  |  |
| --- | --- |
| 1 | server.context-path=/sso-server |

This is important to make sure cookie are for /sso-server domain not localhost.

**User information**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import java.security.Principal;    import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.RestController;    @RestController  public class UserController {    @GetMapping("/user/me")  public Principal user(Principal principal) {  return principal;  }  } |

**Create app1 client application**

In this section, we will create our client app app1 which will be using the SSO server for login.

Let’s create the application by running the following cURL command.

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ curl <https://start.spring.io/starter.zip> \  -d dependencies=web,cloud-oauth2,thymeleaf \  -d groupId=com.shekhargulati -d artifactId=app1 -d name=app1 \  -d description="App1" -d baseDir=app1 \  -o app1.zip |

Open the project in your favourite IDE. Update the pom.xml with one more dependency.

|  |  |
| --- | --- |
| 1  2  3  4 | <dependency>    <groupId>org.thymeleaf.extras</groupId>    <artifactId>thymeleaf-extras-springsecurity4</artifactId>  </dependency> |

**Enabling SSO**

To enable SSO, we have to annotate our application class with EnableOAuth2Sso as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.autoconfigure.security.oauth2.client.EnableOAuth2Sso;    @SpringBootApplication  @EnableOAuth2Sso  public class App1Application {    public static void main(String[] args) {  SpringApplication.run(App1Application.class, args);  }  } |

Next, we need to tell where to find SSO server. Create new file application.yml and define following configuration.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | server:      port: 8082      context-path: /app1  security:    basic:      enabled: false    oauth2:      client:        clientId: foo        clientSecret: bar        accessTokenUri: <http://localhost:8080/sso-server/oauth/token>        userAuthorizationUri: <http://localhost:8080/sso-server/oauth/authorize>      resource:        userInfoUri: <http://localhost:8080/sso-server/user/me> |

**Create index.html**

Create index.html in the templates directory.

|  |  |
| --- | --- |
| 1 | <h1 th:text="'Welcome to App1, ' + ${#authentication.name}"></h1> |

We have to define its mapping. Make App1Application extend WebMvcConfigurerAdapter and override addViewControllers as shown here.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.autoconfigure.security.oauth2.client.EnableOAuth2Sso;  import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;  import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;    @SpringBootApplication  @EnableOAuth2Sso  public class App1Application extends WebMvcConfigurerAdapter {    @Override  public void addViewControllers(ViewControllerRegistry registry) {  registry.addViewController("/").setViewName("index");  }    public static void main(String[] args) {  SpringApplication.run(App1Application.class, args);  }  } |

**SSO in action**

Start both the applications. Go to <http://localhost:8082/app1>. You will be redirected to <http://localhost:8088/sso-server/login>.

Next, you will enter user/password credentials. As these are valid credentials, you will be redirected back to <http://localhost:8082/app1>. Here you will see the index page as shown below.



This works the same when you have more than one applications as well.