Spring Security

https://www.toptal.com/spring/spring-security-tutorial
More to learn

We can use external authentication techniques like ldap and outh2.

Spring security supports all features like ldap, oauth2, bcrypt, etc.
It includes all the libraries we have to directly configure them for us.
So it provides authentication and authorisation.

We have to include the dependency of spring boot security starter.

It also provides the login and logout page for us. Just put the dependency and we can see that changes.

By default the username is user and password is given by spring security in console from where we can pick it up.

After signing it will redirect to home page.

To have our own security features we can make a class like AppSecurityConfig extending WebSecurityConfigurerAdapter.
Add @Configuration over it.

To enable web security for our application we have to add:
@EnableWebSecurity

We need to override userDetailsService method and give it @Bean annotation so that we have it in our application.
We can now give our username and password here.

```
List<UserDetails> users = new ArrayList<>();
// we are using in memory user name and password
// UserDetails is an inbuilt class
users.add(User.defaultPasswordEncoder().username(“Pratik”).roles(“USER”).password(“1234”).build());
return InMemoryUserDetailsManager(users);
// as we are not using any database.
```

Now to use username and password from database:

In order to talk to database and verify we have to use AuthenticationProvider which is implemented by class DaoAuthenticationProvider.

Comment the userDetailsService method.

```
public AuthenticationProvider authProvider() {
	DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
	provider.setUserDetails(userDetailsService); // UserDetailsService
userDetailsService is injected
	provider.setPasswordEncoder(NoOpPasswordEncoder().getInstance()); //
used plain password I.e. no encryption
	return provider;
}
```

Now we need to implement our own UserDetailsService interface.

```
@Component
public class MyUserDetailsService implements UserDetailsService {
	@Autowired
	UserRepository repo;
	public UserDetails loadUserByUserName(String username) throws
UsernameNotFoundException {
		User user = repo.findByUsername(username); // we have used entity
User
		if (user == null) {
			throw new UsernameNotFoundException(“User 404”);
		}
		return new UserPrincipal(user);
	}
}
```

Now we have to implement our own UserDetails through UserPrincipal

```
public UserPrincipal implements UserDetails {
	private User user;
	public UserPrincipal(User user) {
		super();
		this.user = user;
	}

	@Override
	public Collection<? extends GrantedAuthority> getAuthorities() {
		return Collections.singleton(new SimpleGrantedAuthority(“USER”));
	}

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

	@Override
```

```
	public String userName() {
		user.getUsername();
	}
	// more methods to override.

}
```

In b-crypt password we get password in form $2a$<number of rounds of salt>$<rest of password>
$2a stands for bycrpt.

We can add in the line
provider.setPasswordEncoder(new BCryptPasswordEncoder());

To customise our own login page, in AppSecurityConfig we have to override configure(HttpSecurity httpSecurity) method

We have to write the code
```
// csrf -> cross site reference which we are disabling for spring
http.csrf.disable().authorizeRequests().antMatchers("/login").permitAll() // we need to make the /login to handle valid http requests
                        .formlogin().loginPage("/login").permitAll()
                        .anyRequest().authenticated() // we have to first authenticate and then request
                        .and().logout().invalidateHttpSession(true)
                        .clearAuthentication(true)
                        .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
                        .logoutSuccessUrl("/login-success").permitAll(); // we can also redirect to home
```

In login.jsp we can use ${SPRING_SECURITY_LAST_EXCEPTION.message} to have the error message while login.

To have google login in our app:

Before spring 4 or 5 we had the oauth2 inside spring security but now we have to include the dependency spring security oauth2 auto configure.

On AppSecurityConfig configuration class write:
@EnableOAuth2Sso (Single sign on)

In the class we have to only have method configure(HttpSecurity httpSecurity)

```
http.csrf.disable().authorizeRequests().antMatchers("/login").permitAll()
```

.anyRequest().authenticated();

This configuration is typically used in Spring Security to define access rules for different endpoints in a web application. Let's break down what each part does:

1. `http.csrf.disable()`: This disables CSRF (Cross-Site Request Forgery) protection. CSRF protection helps prevent attackers from making unauthorized requests on behalf of a user. Disabling CSRF protection is sometimes necessary for certain types of applications, but it should be done cautiously, as it can introduce security vulnerabilities.

2. `authorizeRequests()`: This indicates that authorization rules for requests will be configured next.

3. `antMatchers("/login").permitAll()`: This specifies that requests to the "/login" endpoint should be permitted without authentication. In other words, users can access the login page without being logged in.

4. `anyRequest().authenticated()`: This specifies that any request other than those explicitly permitted should require authentication. In other words, any request to endpoints not matched by previous antMatchers should only be allowed for authenticated users.

In summary, this configuration ensures that the "/login" endpoint is accessible to all users (permitAll), while all other endpoints require authentication (authenticated). The CSRF protection is disabled, which may be necessary depending on the specific requirements of the application.

Now we have to configure some properties like
clientId, clientSecret, accessTokenUri=https://www.googleapis.com/oauth2/v3/token, userAuthorizationUri=https://accounts.google.com/o/oauth2/auth, tokenName=oauth_token, authenticationScheme=query, clientAuthenticationScheme=form, scope = profile email (what I want from google), userInfoUri=https://www.googleapis.com/userInfo/v2/me, preferTokenInfo=false, clientId=<provide own id>,
clientSecret= <provide own Secret key>

To set up client id and secret
Go to google console -> api and services (find from left sidebar) -> oauth consent screen

To get the user we can mention it in controller:

@RequestMapping(“/user”)
@ResponseBody

```
public Principal user(Principal principal) {
        return principal;
}
//Principal is present in java.security.
```

https://medium.com/geekculture/springboot-api-authentication-using-oauth2-with-google-655b8759f0ac

This is also one of the way.

The `@EnableGlobalMethodSecurity(prePostEnabled = true)` annotation is used in Spring Security to enable method-level security for Spring beans. It enables the use of Pre and Post Annotations for Spring Security. Let's break down its use:

1. **Pre and Post Annotations**: Spring Security provides several annotations that you can use to secure individual methods or classes. Two commonly used annotations are `@PreAuthorize` and `@PostAuthorize`.
   - `@PreAuthorize`: Specifies that the method can only be invoked if the expression specified in `@PreAuthorize` evaluates to true before the method is invoked.
   - `@PostAuthorize`: Specifies that the method should be invoked first, and then the expression specified in `@PostAuthorize` evaluates the result after the method returns. If the expression evaluates to false, an AccessDeniedException is thrown.

2. **Enabling Method-Level Security**: By default, method-level security annotations such as `@PreAuthorize` and `@PostAuthorize` are not enabled in Spring Security. To enable them, you need to use `@EnableGlobalMethodSecurity(prePostEnabled = true)`.

Here's an example of how you can use `@PreAuthorize` and `@PostAuthorize` with `@EnableGlobalMethodSecurity(prePostEnabled = true)`:

```java
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig {
    // Method-level security configuration
```

```
}
```

```java
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    public void doSomethingAdminOnly() {
        // Method implementation
    }

    @PostAuthorize("returnObject.owner == authentication.name")
    public SomeObject findObjectById(Long id) {
        // Method implementation
    }
}
```

In this example:
- `MethodSecurityConfig` is a configuration class annotated with `@EnableGlobalMethodSecurity(prePostEnabled = true)`, enabling method-level security annotations.
- `MyService` is a service class with methods annotated with `@PreAuthorize` and `@PostAuthorize` to specify security constraints.
- The expressions provided in the annotations define the access control rules for invoking or returning from the methods.

By enabling method-level security with `@EnableGlobalMethodSecurity(prePostEnabled = true)`, you can apply fine-grained access control to individual methods in your Spring beans based on various conditions and expressions.

In Spring Security's method security annotations, such as `@PostAuthorize`, `returnObject` refers to the return value of the method being annotated.

So, `returnObject.owner` refers to a property named `owner` of the object returned by the annotated method.

Let's illustrate this with an example:

```java
import org.springframework.security.access.prepost.PostAuthorize;
```

```
import org.springframework.stereotype.Service;

@Service
public class MyService {

    @PostAuthorize("returnObject.owner == authentication.name")
    public SomeObject findObjectById(Long id) {
        // Method implementation that returns SomeObject
    }
}
```

In this example:

- `findObjectById` is a method in `MyService` that returns an object of type `SomeObject`.
- The `@PostAuthorize` annotation is used to specify that the method should be invoked first, and then the expression `returnObject.owner == authentication.name` is evaluated after the method returns.
- `returnObject` refers to the object returned by the `findObjectById` method.
- `returnObject.owner` would be accessing a property named `owner` of the returned `SomeObject`.
- `authentication.name` refers to the name of the currently authenticated principal (user).

So, the expression `returnObject.owner == authentication.name` is checking if the owner of the returned object matches the name of the currently authenticated user. If the expression evaluates to true, the method call is allowed; otherwise, an `AccessDeniedException` is thrown.

This allows you to apply fine-grained access control based on the properties of the returned objects and the authentication context.