* Four important features/concepts of security are Authentication, Authorization, Confidentiality and Integrity.

**Authentication and authorization:**

* **Authentication is the processes of letting the application know who we are.**
* Authentication is how we verify the identity of who is trying to access a particular resource. A common way to authenticate users is by requiring the user to enter a username and password. Once authentication is performed we know the identity and can perform authorization.
* Typically, when we access any application, for example, banking application, the first step is to

Log in by providing our user ID and password. The banking application will validate that user id and password. This process is called authentication. If the authentication is successful, then we can access other pages in the application. If not, we cannot.

* Next is authorization, **once a user logs in or authenticate, how does the application know how much access the user or another application has?** That is where authorization jumps in and it uses roles to do authorization. Each role is mapped to certain URLs or methods in the application, and the user with certain roles will have access to certain functionality within the application and certain users will not have access to certain functionality in the application.
* **An authorization is the process of the application determining how much access you have depending on the role that is assigned to your user.**

**Confidentiality:**

* **This is where applications ensure that the data they are sharing is not vulnerable to hackers.**
* For example, you are logging in providing authentication details. What if a friendly neighborhood hacker hacks your username and password over the network? That is where encryption and decryption comes in. The simplest way of encrypting and decrypting is by using his HTTP. Once the communication is encrypted, the data that is exchanged, even if the hacker captures it, he will not be able to make sense out of it because these application the sender application will use a certain key public key and sends that user details. And then the application will use a private key which will decrypt those details. Even if a hacker gets the details in between, he will not be able to do anything with those details unless he has the private key.
* That is nothing but maintaining the confidentiality of the data being exchanged across applications or from a user to the application.

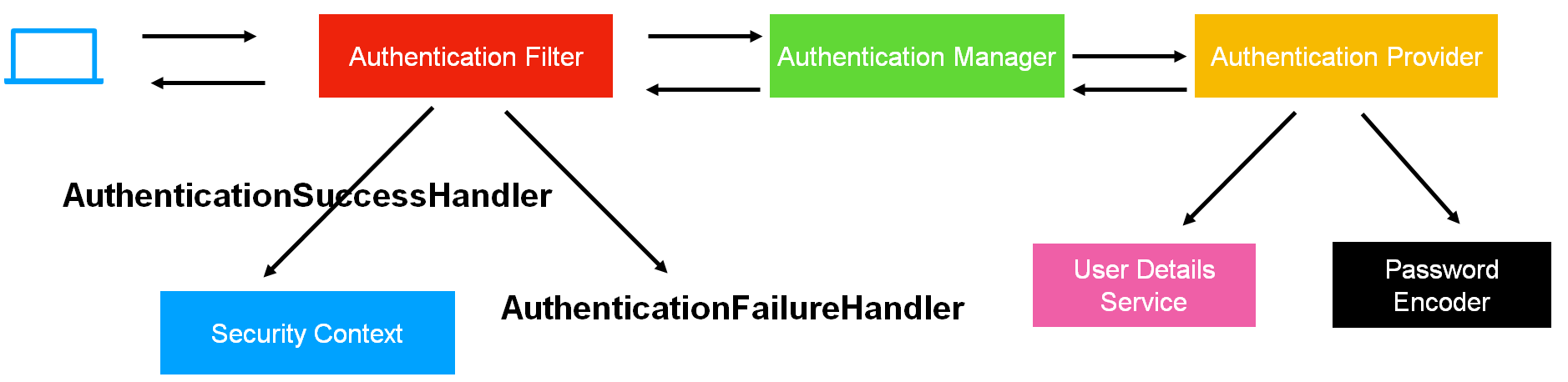
**Integrity:**

* **Integrity is where applications can ensure that whatever data is being exchanged is really coming from the same user that is expecting or the application it is expecting and it is not changed in the process.**
* For example, if this hacker captures the authentication details or any data, if he put something else in that data and sends it to the banking application, the banking application should know that the data was not tweaked/corrupted and the integrity is still maintained.

**CSRF & CORS:**

* **CSRF- Cross Site Request Forgery & CORS- Cross Origin Resource Sharing**
* **Cross site request forgery will prevent another website to submit data on our behalf.**
* **Cross origin Resource sharing will allow applications running, especially if you are developing your front end using Angular or React back and using Java or Node for these applications which are running on different domains need to communicate with each other without any issues.** That is where cors comes in. Browsers will not allow cross origin resource sharing by default.
* you should enable cors within your server side applications.

**Spring Security Architecture:**



1. **AuthenticationFilter:**

* This is the filter that intercepts requests and attempts to authenticate it. In Spring Security, it converts the request to an Authentication Object and delegates the authentication to the AuthenticationManager.

1. **AuthenticationManager:**

* It is the main strategy interface for authentication. It uses the lone method authenticate() to authenticate the request. The authenticate() method performs the authentication and returns an Authentication Object on successful authentication or throw an AuthenticationException in case of authentication failure. If the method can’t decide, it will return null. The process of authentication in this process is delegated to the AuthenticationProvider which we will discuss next.

1. **AuthenticationProvider:**

* The AuthenticationManager is implemented by the ProviderManager which delegates the process to one or more AuthenticationProvider instances. Any class implementing the AuthenticationProvider interface must implement the two methods – authenticate() and supports(). First, let us talk about the supports() method. It is used to check if the particular authentication type is supported by our AuthenticationProvider implementation class. If it is supported it returns true or else false. Next, the authenticate() method. Here is where the authentication occurs. If the authentication type is supported, the process of authentication is started. Here is this class can use the loadUserByUsername() method of the **UserDetailsService** implementation. If the user is not found, it can throw a UsernameNotFoundException.
* On the other hand, if the user is found, then the authentication details of the user are used to authenticate the user. For example, in the basic authentication scenario, the password provided by the user may be checked with the password in the database. If they are found to match with each other, it is a success scenario. Then we can return an Authentication object from this method which will be stored in the Security Context.

1. **UserDetailsService:**

* It is one of the core interfaces of Spring Security. The authentication of any request mostly depends on the implementation of the UserDetailsService interface. It is most commonly used in database backed authentication to retrieve user data. The data is retrieved with the implementation of the lone loadUserByUsername() method where we can provide our logic to fetch the user details for a user. The method will throw a UsernameNotFoundException if the user is not found.

1. **PasswordEncoder:**

* Until Spring Security 4, the use of PasswordEncoder was optional. The user could store plain text passwords using in-memory authentication. But Spring Security 5 has mandated the use of PasswordEncoder to store passwords. This encodes the user’s password using one its many implementations. The most common of its implementations is the BCryptPasswordEncoder. Also, we can use an instance of the NoOpPasswordEncoder for our development purposes. It will allow passwords to be stored in plain text. But it is not supposed to be used for production or real-world applications.

1. **Spring Security Context:**

* This is where the details of the currently authenticated user are stored on successful authentication. The authentication object is then available throughout the application for the session. So, if we need the username or any other user details, we need to get the SecurityContext first. This is done with the SecurityContextHolder, a helper class, which provides access to the security context. We can use the setAuthentication () and get Authentication () methods for storing and retrieving the user details respectively.

**Spring Authentication Types:** They are Basic Authentication,

1. **Basic Authentication:**

* Basic Authentication is enabled when we add Spring security maven dependency to pom.xml.
* There is a default implementation of User Details Service and Password Encoder that is used by the Authentication Provider.
* The job of User Details Service default implementation is to create a default user with username being “user” and password being generated which is uuid and shown in console when application starts.
* These default user and password are stored in memory and gone once the application terminates.
* Every time the application is run default user is created with new password each time being generated and gets stored in memory
* The password being generated here is plain text not encoded.
* **To access any REST API secured under basic auth, go to postman enter URL for API and in authorization tab choose basic auth and provide username as “user” with generated password.**

**Resending Basic Auth Details:**

* When we hit the url using basic auth first time, it goes through all the process and if the authentication is successful, login success info and user details are stored inside security context and then a cookie name **JsessionId** is generated.
* That cookie is sent back to client (postman or web) and now client sends this cookie in header each request.
* Now, The Authentication Filter uses that cookie and checks in security context. If it find cookie, it will not ask prompt for authentication as long as cookie is available.

Creating Custom Security Configuration:

* Till now, we have used default security configuration provided by Spring Security.
* Let us now write our custom security configuration.

1. Create a class named MySecurityConfig.java class annotated with @Configuration
2. Create Security filter chain bean
3. Create custom password encoder bean
4. Create custom user details service bean

@Configuration  
public class MySecurityConfig {  
 @Bean  
 public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {  
 httpSecurity.httpBasic();  
 httpSecurity.authorizeHttpRequests().anyRequest().authenticated();  
 return httpSecurity.build();  
 }  
   
 @Bean  
 public BCryptPasswordEncoder bCryptPasswordEncoder()  
 {  
 return new BCryptPasswordEncoder();  
 }  
 @Bean  
 public UserDetailsService userDetailsService()  
 {  
 InMemoryUserDetailsManager userDetailsManager=new InMemoryUserDetailsManager();  
 UserDetails user= User.*withUsername*("tom").  
 password(bCryptPasswordEncoder().encode("cruise"))  
 .authorities("read").build();  
 userDetailsManager.createUser(user);  
 return userDetailsManager;  
 }  
}

Now, you can hit rest API again with username being “tom” and password being “cruise”.

Here, username and password are stored in memory.

**Custom Authentication Provider:**

* Let us now write our custom authentication provider to provide username and password authentication in authenticate method.
* At runtime, Authentication manager provides Authentication object and supports method Returns true if this AuthenticationProvider supports the indicated Authentication object. Accordingly, Authentication Manager chooses Authentication provider that matches authentication type.

@Component  
public class MyAuthenticationProvider implements AuthenticationProvider {  
 @Override  
 public Authentication authenticate(Authentication authentication) throws AuthenticationException {  
 String userName= authentication.getName();  
 String passWord=authentication.getCredentials().toString();  
 if("tom".equals(userName) && "cruise".equals(passWord))  
 return new UsernamePasswordAuthenticationToken(userName,passWord, Arrays.*asList*());  
 else throw new BadCredentialsException("Invalid Username or Password");  
 }  
  
 @Override  
 public boolean supports(Class<?> authentication) {  
 return authentication.equals(UsernamePasswordAuthenticationToken.class);  
 }  
}

* You can comment UserDetailsService Bean and re run the application again.

1. **Form login:**

* To use form login, change configuration in filter chain bean.

@Bean  
public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {  
 httpSecurity.formLogin();  
 httpSecurity.authorizeHttpRequests().anyRequest().authenticated();  
 return httpSecurity.build();  
}

* Re run application and check in web browser.

Configuring user to access only /hello URL and deny request to remaining all API URL’s

@Bean  
public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {  
 httpSecurity.formLogin();  
 httpSecurity.authorizeHttpRequests().

requestMatchers("/hello").authenticated()  
 .anyRequest().denyAll();  
 return httpSecurity.build();  
}

**Creating custom Authentication Filter:**

MySecurityFilter.java

public class MySecurityFilter implements Filter {  
  
 @Override  
 public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) throws IOException, ServletException {  
 System.*out*.println("Before");  
 filterChain.doFilter(request,response);  
 System.*out*.println("After");  
 }  
}

Now, configure MySecurityFilter in SecurityFilterChain in MySecurityConfig class

@Bean  
public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {  
 httpSecurity.httpBasic();  
 httpSecurity.authorizeHttpRequests().requestMatchers("/hello").authenticated();  
 **httpSecurity.addFilterBefore(new MySecurityFilter(), BasicAuthenticationFilter.class);**  
 return httpSecurity.build();  
}

Other filter classes are:

GenericFilterBean & OncePerRequestFilter

**Securing REST API’s using Role base authorization:**

**Example:**

Create a coupon service with two API’s one to get coupon based on code and another to create coupon.

Any user can call get API, but only admin can create coupon.

1. Create user, role, user\_role tables in mysql db with many to many relationship
2. create user entity

@Entity  
public class User {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 @Column(name = "first\_name")  
 private String firstName;  
 @Column(name="last\_name")  
 private String lastName;  
 private String email;  
 private String password;  
 @ManyToMany(fetch = FetchType.*EAGER*)  
 @JoinTable(name = "user\_role", joinColumns = @JoinColumn(name = "user\_id"), inverseJoinColumns = @JoinColumn(name = "role\_id"))  
 private Set<Role> roles;

//getters & setters

}

1. configure roles entity to implement GrantedAuthority

@Entity  
public class Role implements GrantedAuthority {  
  
 private static final long *serialVersionUID* = 1L;  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 private String name;  
 @ManyToMany(mappedBy = "roles")  
 private Set<User> users;  
  
 @Override  
 public String getAuthority() {  
 return name;  
 }

//getters & setters

}

1. Implement user details service. Here we are using email as username for user to login.

@Service  
public class UserDetailsServiceImpl implements UserDetailsService {  
  
 @Autowired  
 private UserRepo userRepo;  
  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 //here user name is email id, user logs in with email id as user name  
 User user= userRepo.findByEmail(username);  
 if(user==null)  
 throw new UsernameNotFoundException("user not found with email:"+username);  
 return new org.springframework.security.core.userdetails.User(user.getEmail(),user.getPassword(),user.getRoles());  
  
 }  
}

1. Configure security with role based authorization

@Configuration  
public class WebSecurityConfig {  
 @Bean  
 public BCryptPasswordEncoder bCryptPasswordEncoder()  
 {  
 return new BCryptPasswordEncoder();  
 }  
  
 @Bean  
 public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
 http.httpBasic();  
 http.authorizeHttpRequests().

requestMatchers(HttpMethod.*GET*,"/couponapi/coupons/{code:^[AZ]\*$}").hasAnyRole("ADMIN","USER").  
 requestMatchers(HttpMethod.*POST*,"/couponapi/coupons").hasRole("ADMIN")  
 .and().csrf().disable();  
 return http.build();  
 }  
}

**Securing Web APP:**

**Customizing Login:**

1. **Create Security Service interface& its Implementation class:**

public interface SecurityService {  
 boolean login(String userName, String password, HttpServletRequest request, HttpServletResponse response);  
}

@Service  
public class SecurityServiceImpl implements SecurityService{  
  
 @Autowired  
 UserDetailsService userDetailsService;  
  
 @Autowired  
 AuthenticationManager authenticationManager;  
  
 @Autowired  
 SecurityContextRepository securityContextRepository;  
  
 @Override  
 public boolean login(String userName, String password, HttpServletRequest request, HttpServletResponse response) {  
 UserDetails userDetails= userDetailsService.loadUserByUsername(userName);  
 UsernamePasswordAuthenticationToken token=new UsernamePasswordAuthenticationToken(userDetails,password,userDetails.getAuthorities());  
 authenticationManager.authenticate(token);  
 boolean result= token.isAuthenticated();  
 if(result)  
 {  
 SecurityContext context= SecurityContextHolder.*getContext*();  
 context.setAuthentication(token);  
 securityContextRepository.saveContext(context,request,response);  
 }  
 return result;  
 }  
}

Here, We have configured authentication manager to authenticate user and save to security context if authentication is successful.

1. **Configure SecurityFilter, Authentication Manager and Security Context**

@Configuration  
public class WebSecurityConfig {  
  
 @Autowired  
 UserDetailsService userDetailsService;  
  
 @Bean  
 public SecurityContextRepository securityContextRepository()  
 {  
 return new DelegatingSecurityContextRepository(new RequestAttributeSecurityContextRepository(),  
 new HttpSessionSecurityContextRepository());  
 }  
  
 @Bean  
 public AuthenticationManager authManager()  
 {  
 DaoAuthenticationProvider provider=new DaoAuthenticationProvider();  
 provider.setUserDetailsService(userDetailsService);  
 provider.setPasswordEncoder(bCryptPasswordEncoder());  
 return new ProviderManager(provider);  
 }  
  
 @Bean  
 public BCryptPasswordEncoder bCryptPasswordEncoder()  
 {  
 return new BCryptPasswordEncoder();  
 }  
  
 @Bean  
 public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
  
 http.authorizeHttpRequests().  
requestMatchers(HttpMethod.*GET*,"/couponapi/coupons/{code:^[A-Z]\*$}","/showGetCoupon","/getCoupon").hasAnyRole("ADMIN","USER").  
 requestMatchers(HttpMethod.*GET*,"/showCreateCoupon","/createCoupon",

"/createResponse").hasRole("ADMIN").  
 requestMatchers(HttpMethod.*POST*,"/couponapi/coupons","/saveCoupon").hasRole("ADMIN").  
 requestMatchers(HttpMethod.*POST*,"/getCoupon").hasAnyRole("ADMIN","USER").  
 requestMatchers("/","/login","/showReg","/registerUser").permitAll()  
 .and().logout().logoutSuccessUrl("/")  
 .and().csrf().disable();  
 http.securityContext((context)-> context.requireExplicitSave(true));  
 return http.build();  
 }  
}

Here, we have configured Authentication manager to use UserDetails interface (inbuilt) and bcypt password encoder(configured as bean). We have created bean for Security Context Repository. Configured Security Filter Chain to make security context to explicitly save the context.

**CSRF (Cross Site Request Forgery):**

The best way to understand a CSRF attack is by taking a look at a concrete example.

Assume that your bank’s website provides a form that allows transferring money from the currently

logged in user to another bank account.

For example, the transfer form might look like:

**Transfer form**

<form method="post"

    action="/transfer">

<input type="text"

    name="amount"/>

<input type="text"

    name="routingNumber"/>

<input type="text"

    name="account"/>

<input type="submit"

    value="Transfer"/>

</form>

The corresponding HTTP request might look like:

**Transfer HTTP request**

POST /transfer HTTP/1.1

Host: bank.example.com

Cookie: JSESSIONID=randomid

Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876

Now suppose if you authenticate to your bank’s website and then, without logging out, you have visited an evil website. The evil website contains an HTML page with the following form:

**Evil transfer form**

<form method="post"

    action="https://bank.example.com/transfer">

<input type="hidden"

    name="amount"

    value="100.00"/>

<input type="hidden"

    name="routingNumber"

    value="evilsRoutingNumber"/>

<input type="hidden"

    name="account"

    value="evilsAccountNumber"/>

<input type="submit"

    value="Win Money!"/>

</form>

You like to win money, so you click on the submit button. In the process, you have unintentionally transferred $100 to a malicious user. **This happens because, while the evil website cannot see your cookies, the cookies associated with your bank are still sent along with the request**.

Worse yet, this whole process could have been automated by using JavaScript. This means you did not even need to click on the button. Furthermore, it could just as easily happen when visiting an honest site that is a victim of a [XSS attack](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)).

**CSRF is nothing but hacker or malicious website using the information that you submitted in the form of cookies (stored in browser) along with your request.**

The predominant and most comprehensive way to protect against CSRF attacks is to use the [**Synchronizer Token Pattern**](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#synchronizer-token-pattern). **This solution is to ensure that each HTTP request requires, in addition to our session cookie, a secure random generated value called a CSRF token be present in the HTTP request.**  This token must be submitted to the server on every HTTP request that modifies state (PATCH, POST, PUT and DELETE — not GET).

When an HTTP request is submitted, the server must look up the expected CSRF token and compare it against the actual CSRF token in the HTTP request. If the values do not match, the HTTP request should be rejected.

The key to this working is that **the actual CSRF token should be in a part of the HTTP request that is not automatically included by the browser**. For example, **requiring the actual CSRF token in an HTTP parameter or an HTTP header will protect against CSRF attacks**. **Requiring the actual CSRF token in a cookie does not work because cookies are automatically included in the HTTP request by the browser**.

**Note: CSRF is enable by default in Spring Project by adding spring security dependency.**

**Configuring CSRF :**

**CORS:**

In any modern browser, [Cross-Origin Resource Sharing (CORS)](https://www.baeldung.com/cs/cors-preflight-requests) is a relevant specification with the emergence of HTML5 and JS clients that consume data via REST APIs.

Often, the host that serves the JS (e.g. example.com) is different from the host that serves the data (e.g. api.example.com). In such a case, CORS enables cross-domain communication.

**Cross-Origin Resource Sharing (CORS) is an HTTP-header-based mechanism** that allows servers to **explicitly allowlist certain origins** and helps **bypass the same-origin policy**.

This is required since **browsers by default apply the same-origin policy for security**. By implementing CORS in a web application, a webpage could request additional resources and load into the browser from other domains.

**Enabling CORS is straightforward — just add the annotation @CrossOrigin.**

**Using CORS Config, we can allow Origins, HTTP Headers and Methods.**

| **Attributes** | **Description** |
| --- | --- |
| origins | Allows you to specify a list of allowed origins. By default, it allows all origins. The attribute value will be set in the Access-Control-Allow-Origin header of both the preflight response and the actual response. **Example Usage:** @CrossOrigin(origins = "http://localhost:8080") @CrossOrigin(origins = {"http://localhost:8080", "http://testserver:8087"}) |
| allowedHeaders | Allows you to specify a list of headers that will be accepted when the browser makes the request. By default, any headers will be allowed. The value specified in this attribute is used in Access-Control-Allow-Headers in the preflight response. **Example Usage:** @CrossOrigin(allowedHeaders = {"Authorization", "Origin"}) |
| exposedHeaders | List of headers that are set in the actual response header. If not specified, only the [safelisted headers](https://developer.mozilla.org/en-US/docs/Glossary/CORS-safelisted_response_header) will be considered safe to be exposed by the client script. **Example Usage:** @CrossOrigin(exposedHeaders = {"Access-Control-Allow-Origin","Access-Control-Allow-Credentials"}) |
| allowCredentials | When credentials are required to invoke the API, set Access-Control-Allow-Credentials header value to true. In case no credentials are required, omit the header. **Example Usage:** @CrossOrigin(allowCredentials = true) |
| maxAge | Default maxAge is set to 1800 seconds (30 minutes). Indicates how long the preflight responses can be cached. **Example Usage:** @CrossOrigin(maxAge = 300) |