



Building Layers of Defense for Your Application Using Spring Security Framework

Neha Sardana

About Me

- Software Developer
- Fellow JUG Leader at Garden State Java User Group, NJ and New York Java SIG, NY
- Building Spring based applications for almost 10 years
- Love to travel, practice yoga and watch cricket
- Github : nsardana-bny
- [linkedin.com/in/nehasardana9786/](https://www.linkedin.com/in/nehasardana9786/)
- Twitter: @nehasardana09, Medium: @neha-nsit





Agenda

- Authentication and Authorization
- Layers of defense for a web app
- Spring Security - The 5Ws
- Common Security Threats
- Protection against threats
- Basic Authentication
- JWT, OAuth and OpenID Connect
- Code!
- Principles of Application Security



Pre-requisites

- Java
- Spring Framework
- Maven

What is Security?



Authentication

- Authentication is about validating your credentials like User Name/User ID and password to verify your identity.
- The system determines whether you are what you say you are using your credentials.



Authorization

- Authorization is the process to determine whether the authenticated user has access to particular resources.
- It verifies your rights to grant you access to resources such as information, databases, files, etc.
- Authorization usually comes after authentication which confirms your privileges to perform.





Layers of defense for a web app

- Network Firewall
- Login Page
- Security Questions
- Authentication or Multi-factor Authentication
- Hashing and/or Salting
- Authorization

Don't give very specific error message to the user!!! EVER!

Introducing Spring Security Framework

SECURITY





What is Spring Security?

- Spring Security is a powerful and highly customizable authentication and access-control framework for Java based applications.
- It is the de-facto standard for securing Spring-based applications.
- Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements



Why Use Spring Security?

- Easy Integration with Spring Boot
- Supported by large Open Source Community
- Authentication/Authorization (Out of the box)
- Application layer framework (not tied to a particular web server)
- Loosely Coupled
- Filter Entry point/Servlet
- Multiple authentication models
- Protects from Common Security Threats!



Why use Spring Security?

- Due to increased adoption of frameworks like Spring Security, many of the previously common exploits such as: Cross Site Request Forgery and Clickjacking and many others are no longer on OWASP (Open Web Application Security Project) top 10.

How Spring Security integrates



Web



Database



Central Authentication Service



OAuth




LDAP.com
Lightweight Directory Access Protocol

LDAP



Web Services

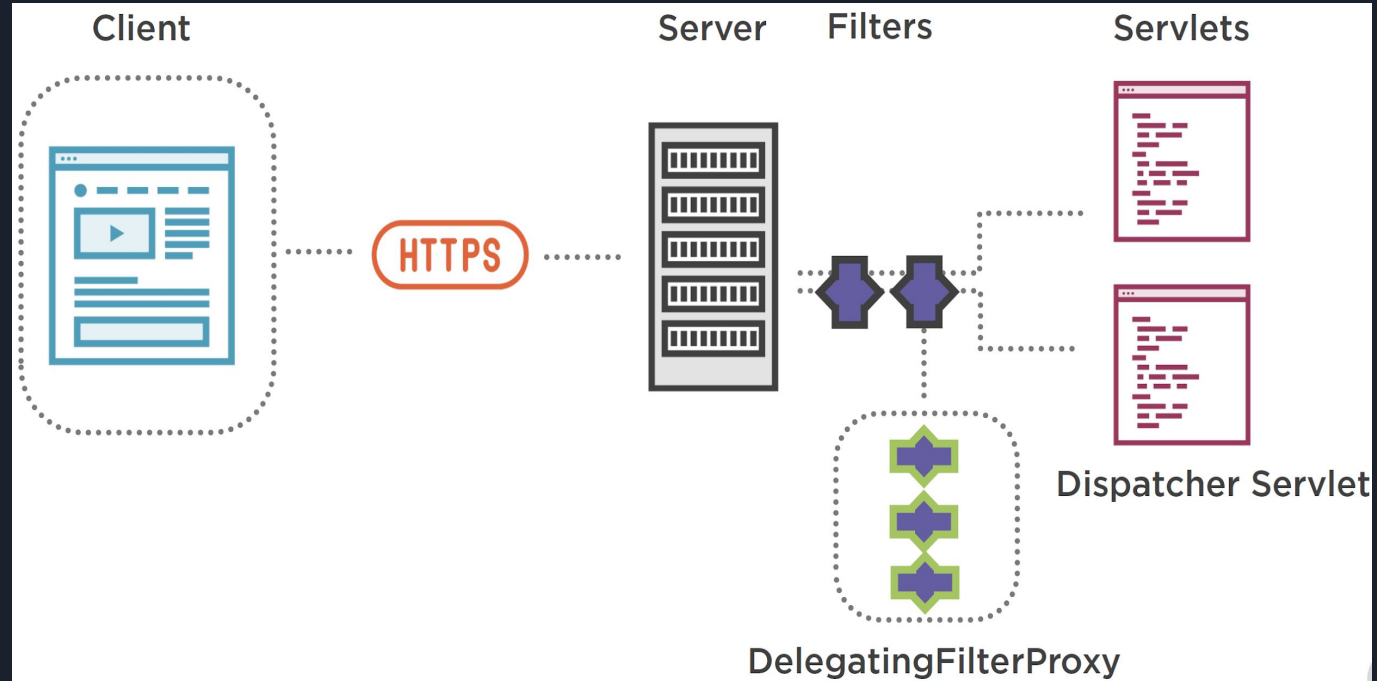
How to use Spring Security in your application



Add below code to your pom.xml in Spring Boot Project

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

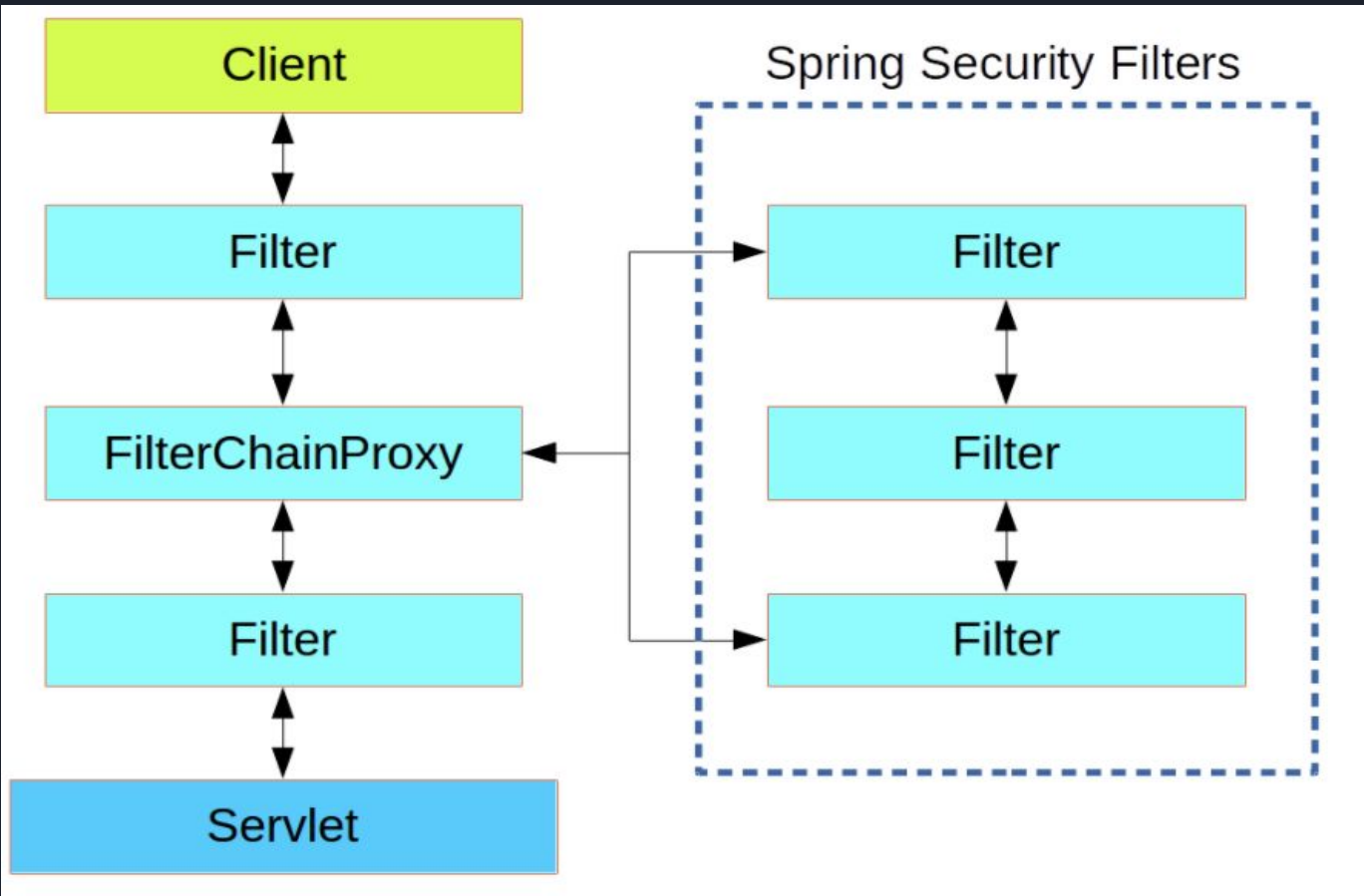
How Spring Security Works





How Spring Security Works

- To enable Spring Security, you have to register Spring Security Filter, also known as, DelegatingFilterProxy, with your servlet container.
- DelegatingFilterProxy will forward all requests to other Spring Security Filters to perform Authentication or Authorization checks.
- Two important things to kick start your security framework: Dispatcher Servlet & Delegating Proxy Filter.



Filter Interface

```
public interface Filter {  
  
    /** <p>Called by the web container ...</p>*/  
    default public void init(FilterConfig filterConfig) throws ServletException {}  
  
    /** The <code>doFilter</code> method of the Filter is called by the ...</p>*/  
    public void doFilter(ServletRequest request, ServletResponse response,  
                        FilterChain chain)  
        throws IOException, ServletException;  
  
    /** <p>Called by the web container ...</p>*/  
    default public void destroy() {}  
}
```



Filter Interface

- **Init()** - Called by the web container to indicate to a filter that it is being placed into service.
- **destroy()** - Called by the web container to indicate to a filter that it is being taken out of service.
- **doFilter()** - The doFilter method of the Filter is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain.
- The FilterChain passed in to this method allows the Filter to pass on the request and response to the next entity in the chain.



FilterChainProxy & Security Filter Chain

- DelegatingFilterProxy delegates the request to FilterChainProxy (FCP) which in turn delegates to another object SecurityFilterChain (SFC).
- SFC is a wrapper around collection of Spring Filters that performs actual security tasks.
- When a request comes, FCP iterates through SFC in order to find the one which matches with the request url.
- After the above step, the appropriate Filter is invoked to perform security tasks.
- The Order of SFC matters a lot!
- There are options to have multiple SFC in one application to have multiple types of authentication for different types of URLs.



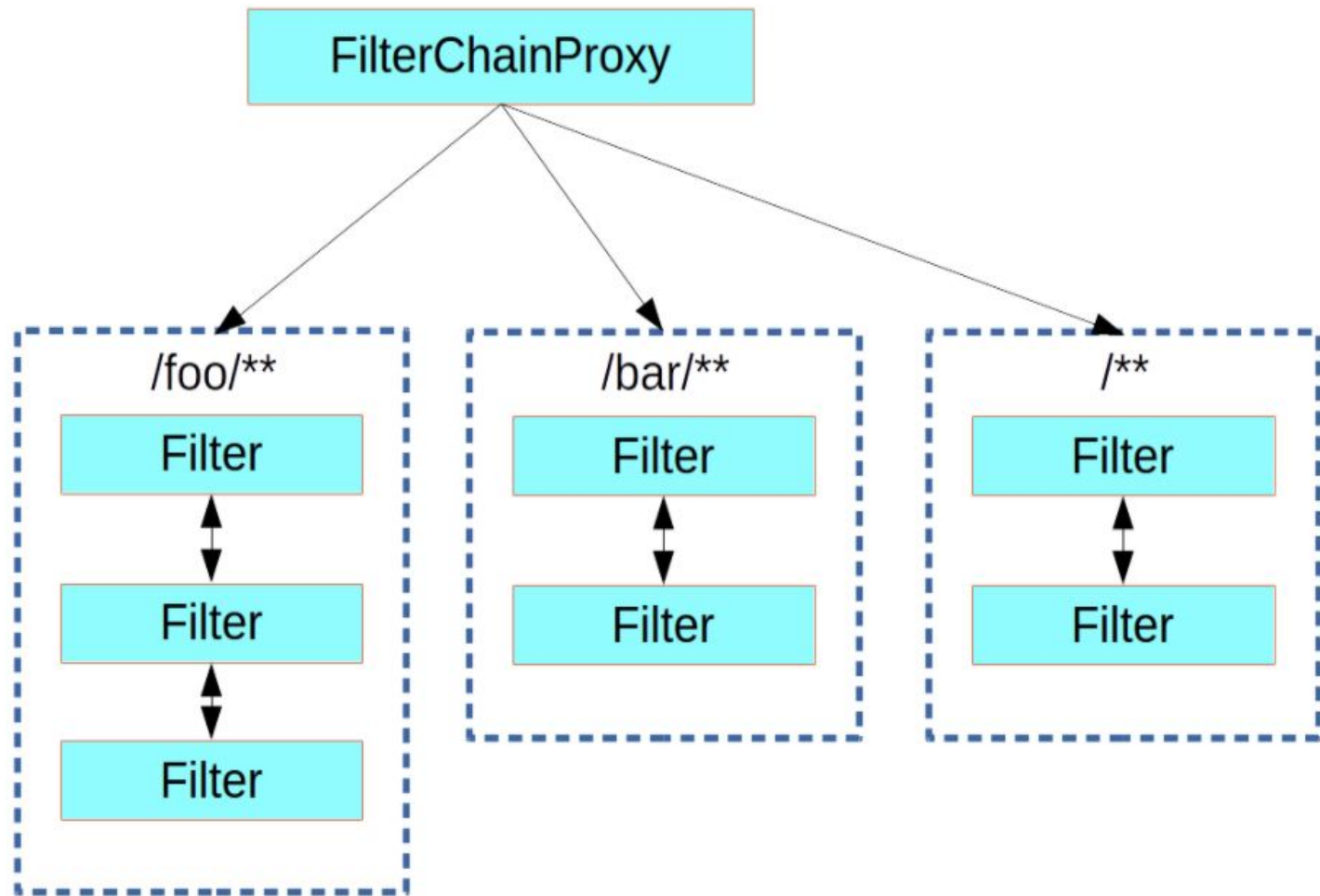
SecurityFilterChain Interface

```
public interface SecurityFilterChain {  
  
    boolean matches(HttpServletRequest request);  
  
    List<Filter> getFilters();  
  
}
```



SecurityFilterChain Interface

- **matches()** - returns boolean to checks if the request matches the filter chain
- **getFilters()** - returns a list of security filters for the matched request

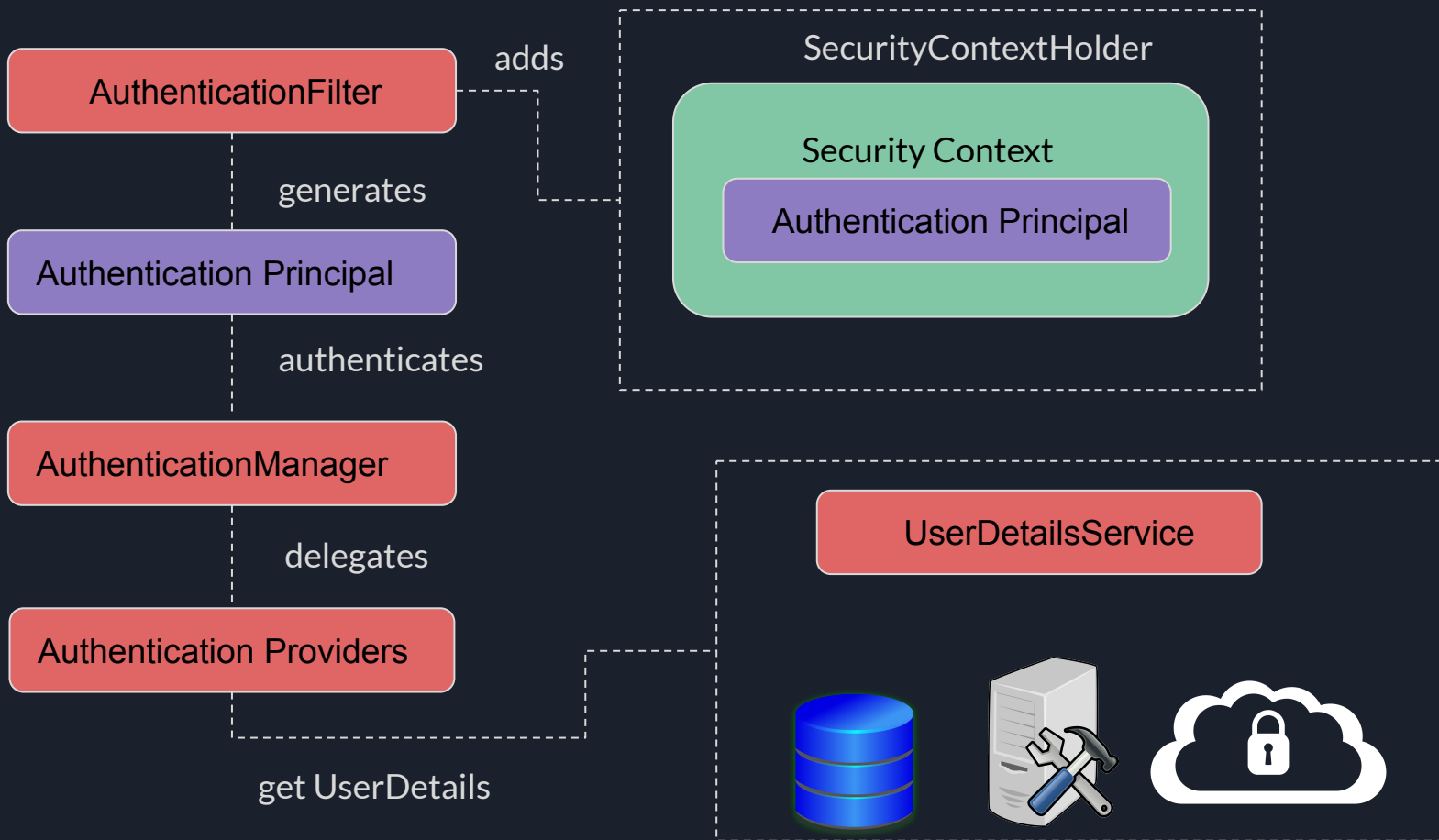


Authentication Flow



Authentication Flow

- Authentication Filter creates an “Authentication Request” and passes it to the Authentication Manager
- Authentication Manager delegates to the Authentication Provider
- Authentication Provider uses UserDetailsService to load the UserDetails and returns an “Authenticated Principal”
- Authentication Filter sets the Authentication in the SecurityContext





Authentication Object

- **Principal** - identifies the user. When authenticating with a username/password this is often an instance of [UserDetails](#).
- **Credentials** - Often a password. In many cases this will be cleared after the user is authenticated to ensure it is not leaked.
- **Authorities** - the [GrantedAuthorities](#) are high level permissions the user is granted. A few examples are roles or scopes.

Authentication Object


```
public interface Authentication extends Principal, Serializable {  
  
    /** Set by an <code>AuthenticationManager</code> to indicate the authorities that the ...*/  
    Collection<? extends GrantedAuthority> getAuthorities();  
  
    /** The credentials that prove the principal is correct. This is usually a password, ...*/  
    Object getCredentials();  
  
    /** Stores additional details about the authentication request. These might be an IP ...*/  
    Object getDetails();  
  
    /** The identity of the principal being authenticated. In the case of an authentication ...*/  
    Object getPrincipal();  
  
    /** Used to indicate to {@code AbstractSecurityInterceptor} whether it should present ...*/  
    boolean isAuthenticated();  
  
    /** See {@link #isAuthenticated()} for a full description. ...*/  
    void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException;  
  
}
```


Authorization Flow



Authorization Flow

- FilterSecurityInterceptor obtains the “Security Metadata” by matching the current request
- FilterSecurityInterceptor gets the current Authentication
- The Authentication, Security Metadata and Request is passed to the AccessDecisionManager
- AccessDecisionManager delegates it to the AccessDecisionVoter(s) for decisioning



What happens when there is Exception?

- When “Access Denied” for current Authentication, ExceptionTranslationFilter delegates to the AccessDeniedHandler and returns a Http 403 status code
- When current Authentication is “Anonymous”, ExceptionTranslationFilter delegates to the AuthenticationEntryPoint to start the Authentication Process



Filter Chain Ordering

1. ChannelProcessingFilter
2. WebAsyncManagerIntegrationFilter
3. SecurityContextPersistenceFilter
4. HeaderWriterFilter
5. CorsFilter
6. CsrfFilter
7. LogoutFilter
8. OAuth2AuthorizationRequestRedirectFilter
9. Saml2WebSsoAuthenticationRequestFilter
10. X509AuthenticationFilter
11. AbstractPreAuthenticatedProcessingFilter
12. CasAuthenticationFilter
13. OAuth2LoginAuthenticationFilter
14. Saml2WebSsoAuthenticationFilter
15. UsernamePasswordAuthenticationFilter
16. OpenIDAuthenticationFilter
17. DefaultLoginPageGeneratingFilter



Filter Chain Ordering (Contd.)

- | | | | |
|-----|---|-----|------------------------------------|
| 18. | DefaultLogoutPageGeneratingFilter | 26. | RememberMeAuthenticationFilter |
| 19. | ConcurrentSessionFilter | 27. | AnonymousAuthenticationFilter |
| 20. | DigestAuthenticationFilter | 28. | OAuth2AuthorizationCodeGrantFilter |
| 21. | BearerTokenAuthenticationFilter | 29. | SessionManagementFilter |
| 22. | BasicAuthenticationFilter | 30. | ExceptionTranslationFilter |
| 23. | RequestCacheAwareFilter | 31. | FilterSecurityInterceptor |
| 24. | SecurityContextHolderAwareRequestFilter | 32. | SwitchUserFilter |
| 25. | JaasApiIntegrationFilter | | |



Common Security Threats

- Injection
- Broken Authentication
- Sensitive Data Exposure
- XML External Entities
- Broken Access Control
- Security Misconfiguration
- Cross-Site Scripting
- Insecure Deserialization
- Using Components with Known Vulnerabilities
- Insufficient logging and monitoring

How Spring Security protects against Threats



Spring Security Headers

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate  
Pragma: no-cache  
Expires: 0  
X-Content-Type-Options: nosniff  
Strict-Transport-Security: max-age=31536000 ; includeSubDomains  
X-Frame-Options: DENY  
X-XSS-Protection: 1; mode=block
```




Spring Security Headers

- **Disables Browser Cache**
 - If your application provides its own cache control headers Spring Security will back out of the way
- **Disables Content Sniffing**
 - Content Sniffing- when the browser is trying to guess the content type of a request
- **Disables Frame (prevents Clickjacking)**
 - By default Spring Security disables rendering pages within an iframe
- **HTTP Strict Transport Security (HSTS)**
 - Instructs Browser to treat this domain and subdomains as an HSTS host for a year by default



Spring Security Headers

- **Reflective XSS**
 - Prevents Browser from rendering a page if it suspects a reflective XSS attack



CSRF Protection

- Cross-Site Request Forgery is an attack that forces a user to execute unwanted actions in an application they're currently logged into.
- If the user is a normal user, a successful attack can involve state-changing requests like transferring funds or changing their email address.
- If the user has elevated permissions, a CSRF attack can compromise the entire application.
- Spring Security uses Synchronizer Token Pattern where along with creating the session cookie on authentication, it also creates a csrf token which a malicious site can not access or use.
- On the backend side, CSRF Filter expects this token along with state changing requests to allow the request to pass through.



Example of CSRF for SPA

```
@EnableWebSecurity  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http  
            .csrf()  
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());  
    }  
}
```



HTTP Verb Tampering

- HTTP Verb Tampering is an attack that exploits vulnerabilities in HTTP verb (also known as HTTP method) authentication and access control mechanisms.
- Many authentication mechanisms only limit access to the most common HTTP methods, thus allowing unauthorized access to restricted resources by other HTTP methods.



Example of HTTP Verb Tampering

```
@Bean
public StrictHttpFirewall httpFirewall() {
    StrictHttpFirewall firewall = new StrictHttpFirewall();
    firewall.setAllowedHttpMethods(Arrays.asList("GET", "POST"));
    return firewall;
}
```



Enable Content Security Policy to Prevent XSS Attacks

- [Content Security Policy](#) (CSP) is an added layer of security that helps mitigate [XSS](#) ([cross-site scripting](#)) and data injection attacks.
- To enable it, you need to configure your app to return a Content-Security-Policy header.
- You can also use a `<meta http-equiv="Content-Security-Policy">` tag in your HTML page.



Example for Content Security Policy prevention

```
@EnableWebSecurity
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.headers().contentSecurityPolicy( policyDirectives: "script-src "+
            "'self' https://trustedscripts.example.com; object-src "+
            "https://trustedplugins.example.com; report-uri /cspreport-endpoint/");
    }
}
```




Session Fixation

- Session fixation attacks are a potential risk where it is possible for a malicious attacker to create a session by accessing a site, then persuade another user to log in with the same session (by sending them a link containing the session identifier as a parameter, for example).
- Spring Security protects against this automatically by creating a new session or otherwise changing the session ID when a user logs in.



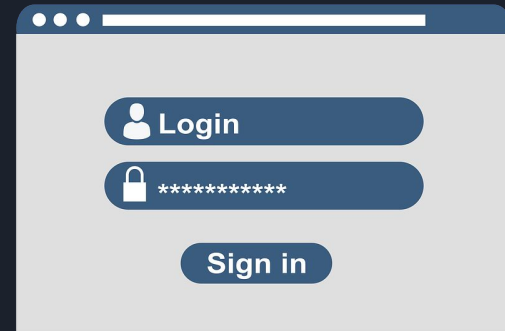
HTTPS in Production and NOT HTTP!

```
@Configuration
public class WebSecurityConfig extends
    WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.requiresChannel().anyRequest().requiresSecure();
    }
}
```

Basic Authentication

Basic Authentication

- Credentials are transmitted through header
- Header name: Authentication
- Header value: Basic + Base64(username:password)
- Eg: Authorization: Basic YMDDdnskslkdllsklddlk

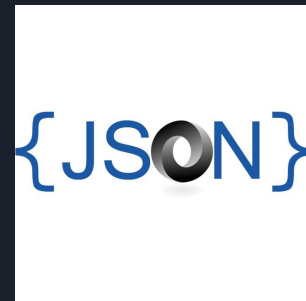




Common Challenges with Basic Authentication

- Base64 is easy to decode!
- Managing password hash with application data is a bottleneck.
- Sharing username and password for integrating with third party vendor can cause a lot of problems.
- In case of a distributed application, while scaling up, you have to scale up the identity and authentication mechanism as well.
- If starting from a monolith application and going further down into Microservices, you will require Authentication and Authorization for each of the service which creating a problem later.

Introducing OAuth2, OIDC and JWT





Goal of OAuth2

- This was driven mainly because the applications realised that user password sharing with multiple unknown apps was a big security threat.
- To allow applications to access data from third party apps without the users sharing their password.
- Problems while implementing single sign on because of the need to share passwords between applications



OAuth2

- OAuth 2.0 is the industry-standard protocol for authorization.
- OAuth 1.0 was published in Dec 2007.
- It uses scopes to define permissions about what actions an authorized user can perform.
- However, OAuth 2.0 is not an authentication protocol and provides no information about the authenticated user.
- <https://auth0.com/>



OpenID Connect

- OpenID Connect (OIDC) is an OAuth 2.0 extension that provides user information.
- It adds an ID token in addition to an access token, as well as a /userinfo endpoint that you can get additional information from.
- It also adds an endpoint discovery feature and dynamic client registration.



JSON Web Token (JWT)

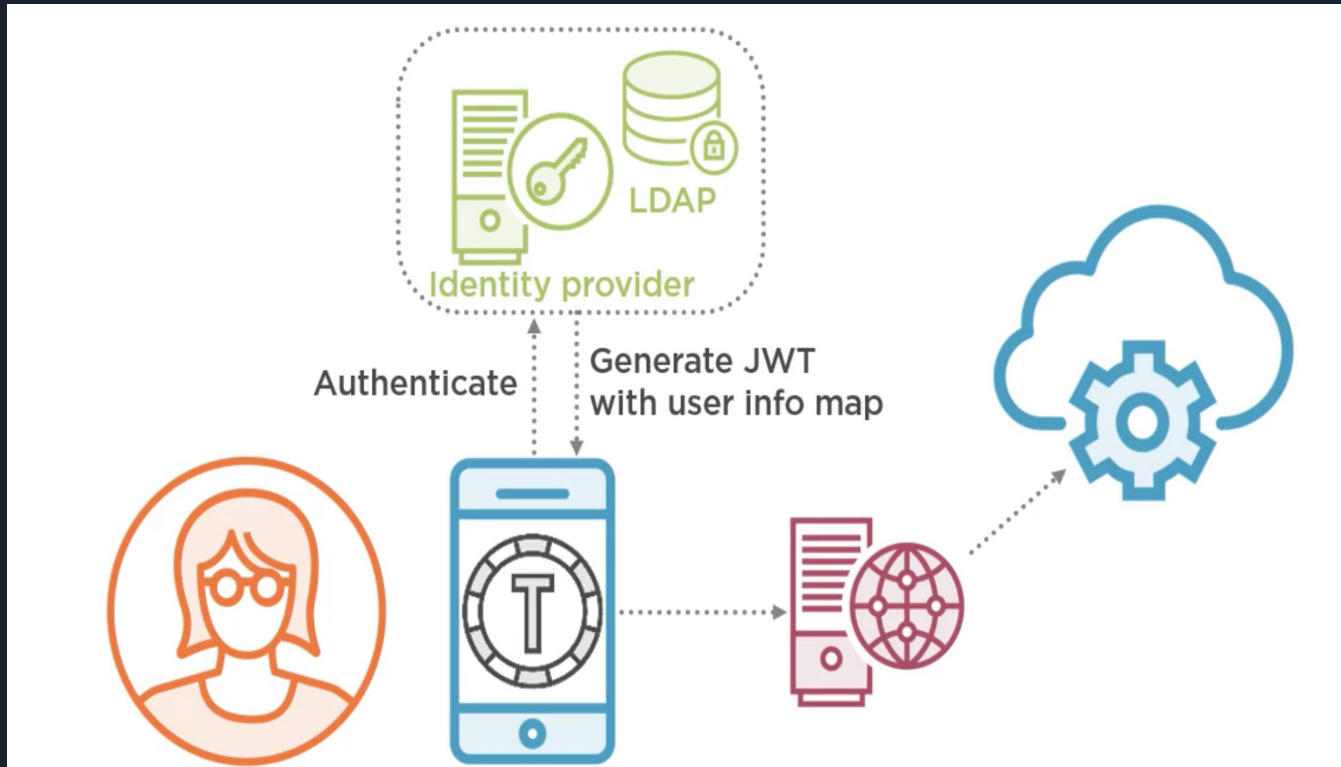
- Light weight
 - JSON Map Information
- Verifiable
 - Digitally signed and Base64 URL encoded
- Protocol Agnostic
 - Can be used standalone
- Expiration Time



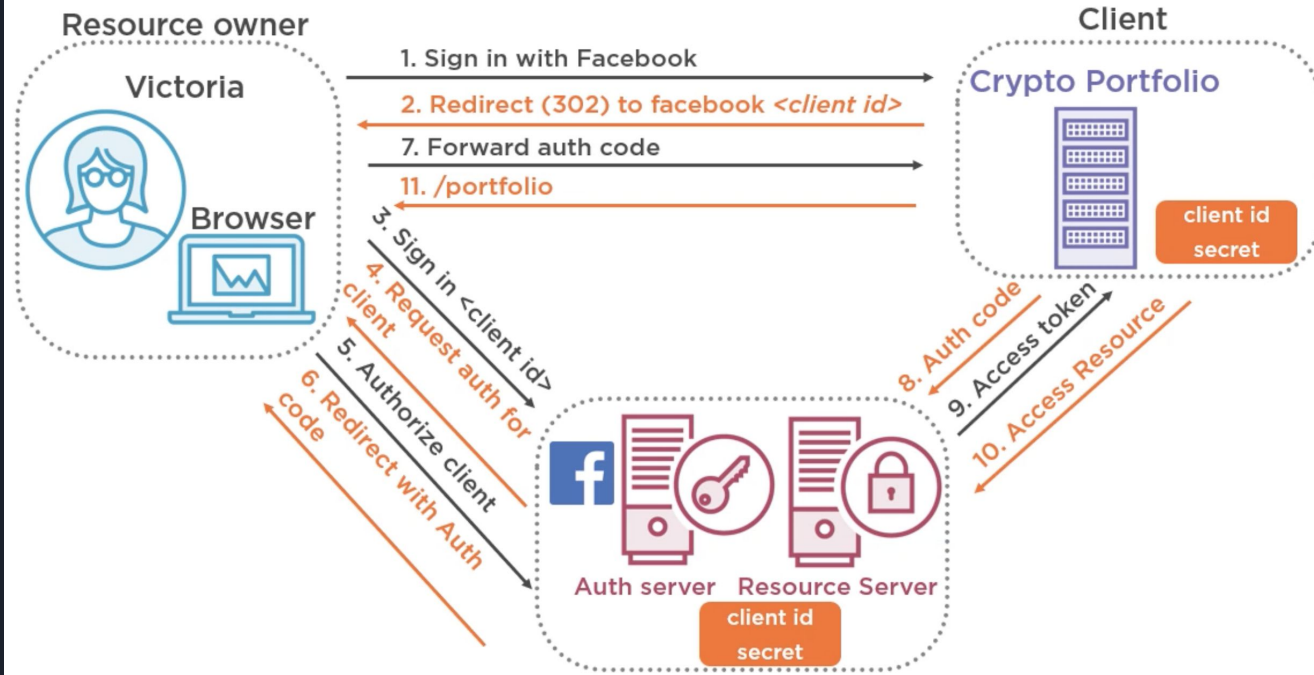
JSON Structure

- Header
- Payload
- Signature

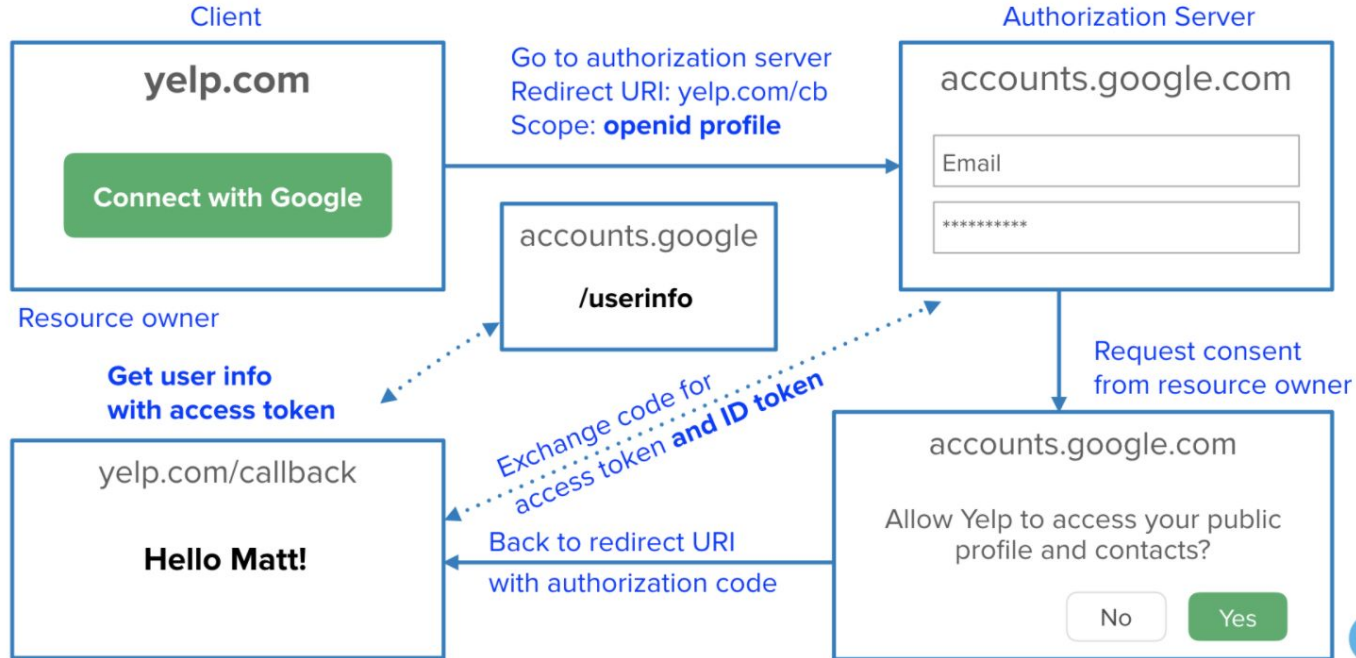
Field	Description
iat	Indicates the token was "issued at"
jit	JSON Token ID
iss	Issuer of the token
exp	Expiry time
sub	Subject
aud	Audience



Oauth2 Authorization Code Grant



OIDC Authorization Code Flow



Code:

<https://github.com/nsardana-bny/spring-security-oauth2-okta-sample>

Final Thoughts



Core Principles of Application Security

- Minimize attack surface area
- Establish secure defaults
- The principle of least privilege
- The principle of defense in depth
- Fail securely
- Don't trust services
- Separation of duties
- Avoid security by obscurity
- Keep security simple
- Fix security issues correctly



References

[Medium Blog - Spring Security Code Walkthrough](#)

[Spring Design Principles](#)

[Top Courses on Spring Security](#)

[OWASP Top 10](#)

[Spring Security Reference](#)

[OAuth 2.1](#)

Picture Credits: <https://tinyurl.com/fp5pfux7> (Pluralsight.com)

Thank you!

Twitter: @nehasardana09