Spring Security

Spring Security

Spring Security is an effective, customizable Java framework for authentication, authorization, and access control in Spring-primarily-based applications. It protects our applications from unauthorized access and also manages user permissions.

- It performs error handling, protects the resources, and manages login authentication. Those are the principal functions of Spring Security.
- It has a flexible configuration, secure applications, and smooth and easy integration.
- This is broadly used in enterprise applications.

Key features

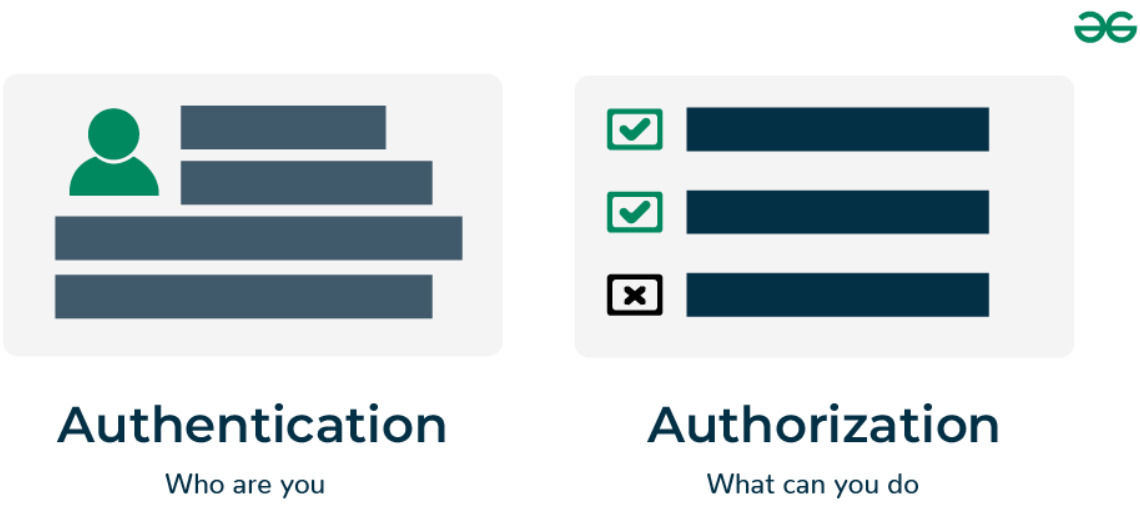
Some of the core features of Spring Security are depicted below:

- **Authentication**: Process of Identifying the person trying to access the application.
- **Authorization**: Process of Identifying if the person is allowed to do this operation.
- **Principal**: To know, is the person that we have identified through an authentication process. It is also called the Currently logged-in user that we store in session.
- **Protection**: This protects against most common threats like cross-site scripting and SQL injection.
- **Granted Authority**: These are the groups of permissions that are allowed for a user.

Difference between Authentication and Authorization

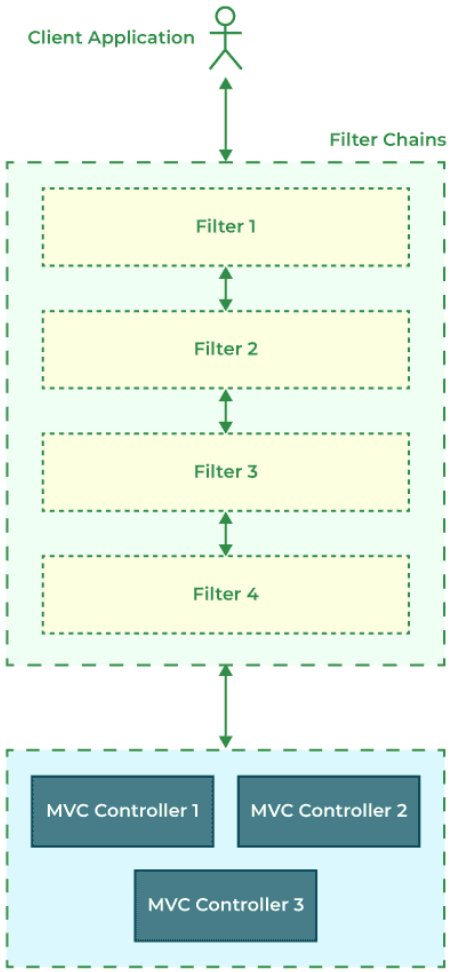
Features	Authentication	Authorization
Definition	It is a process used to verify user's identity.	It determines that within the application, which task a user is allowed to do.
Working	Checks credentials like username and password provided by user against stored credentials.	To determine resources, it uses user's identity and pre-defined access control rules.
Performance	It is fast.	After authentication, it authorizes.
Result	User gets authenticated and security token granted to them.	Users get access to specific features based on the rules.

Below is a simple diagram to know about Authentication and Authorization.



Filter Chain

To perform most of the security features, Spring Security utilizes the filter chain. Internally Spring Security maintains a filter chain with each filter having a specific responsibility. Filters are added or removed from the system depending on needed services. This is how the filter chain works in a web application:



In the above image,

- Client sends the request for a resource.
- The application container creates a filter chain to execute an incoming request.
- After that each requests i.e. the HttpServletRequest passes through that filter chain based upon the path of the request URI.

Password Encoder

Servlet support in Spring Security includes storing passwords securely by integrating a PasswordEncoder. We can configure the PasswordEncoder implementation of Spring Security by exposing the PasswordEncoder Bean. A sample code is given below.

There are various types of Password encoder available in Spring Security

- BCryptPasswordEncoder
- StandardPasswordEncoder
- NoOpPasswordEncoder
- Pbkdf2PasswordEncoder
- Argon2PasswordEncoder

@EnableWebSecurity

@EnableWebSecurity annotation gives signals to Spring to enable its web security support.

- This makes the application secured.
- This is used in conjunction along with the **@Configuration** annotation.

AuthenticationProvider

Authentication Providers perform the actual authentication work.

- AuthenticationManager is the API that describes that how Spring Security's filters perform authentication.
- ProviderManager is the mostly used implementation of AuthenticationManager.

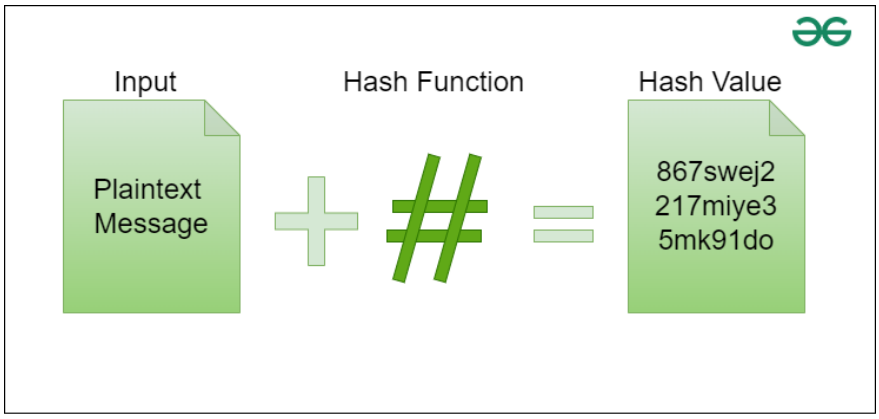
Difference between hasRole() and hasAuthority()

Features	hasRole()	hasAuthority()
Working	It is the shortcut for specifying URLs require for a particular role.	It specifies that URLs require a particular authority.
Prefix	Do not have to add role prefix (default “ROLE_”).	Have to add role prefix (default “ROLE_”).
Syntax	The role is required (i.e. ADMIN, USER etc.).	The authority to require (i.e. ROLE_ADMIN, ROLE_USER, etc.)

Hashing in Spring Security

Generally, end-users register the details at the same time provides password as well. That password we store in the database as it is that is called **Plain text**.

- Storing Plain text in the database is not recommended standard, so it should be converted to unreadable format that is called encrypting the password.
- Storing encrypted password in the database called **Password Hashing**.
- To convert it into unreadable format, there are so many hashing algorithms like Message Digester Algorithm (MD4, MD5), Security Hashing Algorithm – SHA (SHA256, SHA128) ETC.



Salting

- Salting is a process in Spring Security to combine random data with a password before password hashing.
- By increasing its uniqueness and complexity, it improves Hashing.

Benefits of Spring Security

- Centralized Security Management: Manage security configurations in a centralized manner, making it easier to maintain and update.
- Flexibility: Provides a flexible security architecture that can be extended and customized according to application requirements.
- Built-in Protection: Comes with built-in protections against common security threats such as CSRF, XSS, and session fixation.
- Seamless Integration: Works seamlessly with other Spring projects, enhancing the overall functionality and security of your applications.
- Community and Support: Backed by a large community and extensive documentation, making it easier to find solutions and best practices.

Basic Security Configuration:

Dependency

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

SecurityController.java

```
package com.rutuja.Security.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/security")
public class SecurityController {

    @GetMapping("/public")
    public String publicall()
    {
        return "Hello, Public API";
    }

    @GetMapping("/user")
    public String userall() {
        return "Hello, User API";
    }

    @GetMapping("/admin")
    public String adminall()
    {
        return "Hello,Admin API";
    }
}
```

SecurityConfiguration.java

```java
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws
Exception
    {
        return httpSecurity.authorizeHttpRequests(resistry->{
            resistry.requestMatchers("/security/public").permitAll();
            resistry.requestMatchers("/security/admin").hasRole("admin");
            resistry.requestMatchers("/security/user").hasRole("user");
        }).formLogin(AbstractAuthenticationFilterConfigurer::permitAll)
                .build();
    }

    @Bean
    public PasswordEncoder passwordEncoder()
    {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public UserDetailsService userDetailsService()
    {
        UserDetails userDetails1=User.builder().username("Rutuja")
        .password(passwordEncoder().encode("Rutuja@123"))
        .roles("admin").build();
        UserDetails userDetails2=User.builder().username("Aniket")
        .password(passwordEncoder().encode("Aniket@123"))
        .roles("user").build();
        return new InMemoryUserDetailsManager(userDetails1,userDetails2);
    }

}
```