What is Spring Security?

- Spring Security is a framework that provides **authentication**, **authorization**, and protection against **common attacks**.
- It is the de-facto standard for securing Spring-based applications.

Maven Dependency

- Spring Boot provides a spring-boot-starter-security starter that aggregates Spring Security-related dependencies.
- By default, the Authentication gets enabled for the Application.
- Also, content negotiation is used to determine if **basic** or **formLogin** should be used.

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

Security Properties

- If we don't configure the password using the predefined property spring.security.user.password and start the application, a **default password** is **randomly** generated and printed in the **console log**.
- We can specify username and password in **application.properties** using the following properties.

```
spring.security.user.name=test
spring.security.user.password=test
```

Authentication

- Spring Security provides comprehensive support for authentication.
- 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.

Authorization

- Spring Security provides comprehensive support for authorization.
- Authorization is determining who is allowed to access a particular resource.
- Spring Security provides request based authorization and method based authorization.

Protection Against Exploits

- Spring Security provides protection against common exploits.
- Whenever possible, the protection is enabled by default.
- Spring Security protects against:
  - ✓ **CSRF** - Spring provides comprehensive support for protecting against Cross Site Request Forgery (CSRF) attacks
  - ✓ **HTTP Headers** - Spring Security provides a default set of security related HTTP response headers to provide secure defaults.
  - ✓ **HTTP Requests**

Spring Boot Auto Configuration

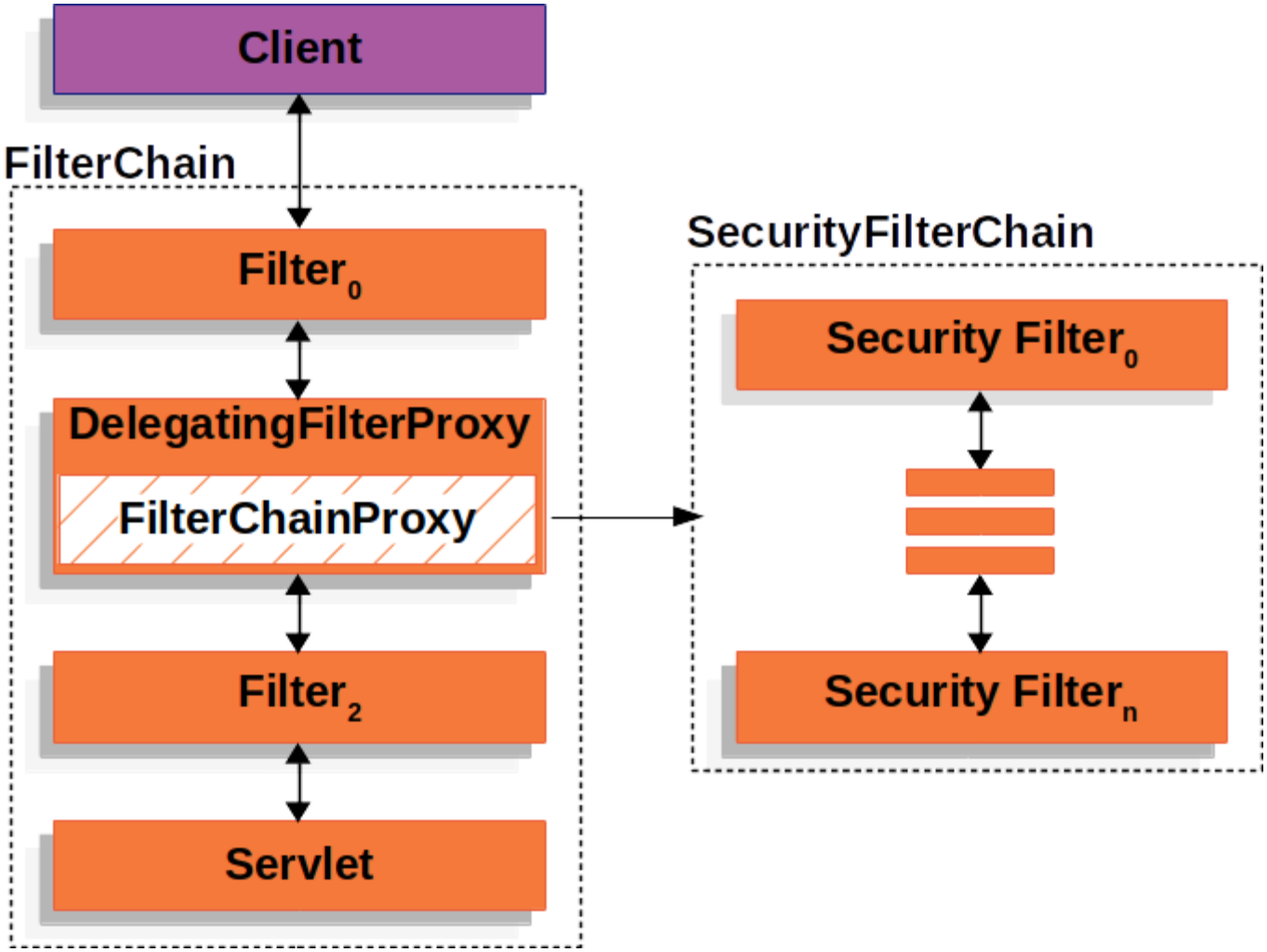
- When we include spring-boot-starter-security in the project, the following steps are taken by Spring Boot:
  - ✓ Creates a servlet Filter as a bean named **springSecurityFilterChain**.
  - ✓ This bean is responsible for all the security (protecting the application URLs, validating submitted username and passwords, redirecting to the login form, and so on) within your application.
  - ✓ Creates a **UserDetailsService** bean with a username of user and a randomly generated password that is logged to the console.
  - ✓ Registers the Filter with a bean named springSecurityFilterChain with the Servlet container for every request.

Servlet API Methods

- The following Servlet API methods in **HttpServletRequest** are integrated with security:
  - ✓ `getRemoteUser()`
  - ✓ `getUserPrincipal()`
  - ✓ `isUserInRole(java.lang.String)`
  - ✓ `login(java.lang.String, java.lang.String)`
  - ✓ `logout()`

Configure Security using SecurityFilterChain

- SecurityFilterChain is used by FilterChainProxy to determine which Spring Security Filter instances should be invoked for the current request.



Basic Authentication

- Spring Security provides support for Basic HTTP Authentication
- When a user makes an unauthenticated request a resource, Spring Security indicates that the unauthenticated request is denied and sends a **WWW-Authenticate** header.
- When a client receives the WWW-Authenticate header, it knows it should take username and password and send them back for authentication.
- By default Basic Authentication is enabled.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
       // code
      http.httpBasic();
      return http.build();
}
```

Form Authentication

- Spring Security provides support for username and password being provided through an HTML form.
- When a user makes an unauthenticated request to the resource, Spring Security redirects user to login page.
- The login page takes username and password and submits, Spring Security authenticates given credentials and if valid then allows user to proceed to resource.
- By default Form Authentication is enabled.

```
public SecurityFilterChain filterChain(HttpSecurity http) {
     http.formLogin(form ->
             form.loginPage("/login")  // URL for login form
                 .permitAll() );
     // ...
}
```

HttpSecurity Class

- It allows configuring web based security for specific http requests.
- By default it will be applied to all requests, but can be restricted using **requestMatcher**(RequestMatcher) or other similar methods.

```
@Configuration
@EnableWebSecurity
public class MySecurityConfiguration {
 @Bean
 public SecurityFilterChain filterChain(HttpSecurity http)
         throws Exception {
   http.httpBasic();
   http.authorizeHttpRequests()
      .requestMatchers("/api/*")
      .permitAll();           // allow all
   http.authorizeHttpRequests()
       .requestMatchers("/admin/*")
       .authenticated();    // allow only authenticated users
   return http.build();
 }
}
```

In-Memory Authentication

- Spring Security's **InMemoryUserDetailsManager** implements **UserDetailsService** to provide support for username/password based authentication that is stored in memory.
- InMemoryUserDetailsManager provides management of **UserDetails** by implementing the **UserDetailsManager** interface.

```
@Bean public UserDetailsService users() {
 // The builder will ensure the passwords are encoded before
 // saving in memory
 UserBuilder users = User.withDefaultPasswordEncoder();
 UserDetails user = users
                    .username("user").password("user")
                    .roles("USER").build();
 UserDetails admin = users
                    .username("admin").password("admin")
                    .roles("USER", "ADMIN")
                    .build();
 return new InMemoryUserDetailsManager(user, admin);
}
```

In-Memory Authentication with Password Encoding

```
@Configuration
@EnableWebSecurity
public class InMemorySecurityDemo {
    @Bean
    public InMemoryUserDetailsManager userDetailsService
                        (PasswordEncoder passwordEncoder) {
        UserDetails user = User.withUsername("user")
            .password(passwordEncoder.encode("password"))
            .roles("USER")
            .build();

        UserDetails admin = User.withUsername("admin")
            .password(passwordEncoder.encode("admin"))
            .roles("USER", "ADMIN")
            .build();

        return new InMemoryUserDetailsManager(user, admin);
    }
    @Bean
    public PasswordEncoder passwordEncoder() {
        PasswordEncoder encoder =
          PasswordEncoderFactories.createDelegatingPasswordEncoder();
        return encoder;
    }
}
```

UserDetails Interface

- Provides core user information.
- They simply store user information which is later encapsulated into Authentication objects.

Collection<? extends **GrantedAuthority**> **getAuthorities**()
Returns the authorities granted to the user.

String getPassword()
Returns the password used to authenticate the user.

String getUsername()
Returns the username used to authenticate the user.

boolean isEnabled()
Indicates whether the user is enabled or disabled.

User Class

- Models core user information retrieved by a UserDetailsService.
- Implements UserDetails interface.
- The following method return **UserBuilder**, which is a nested class used to build users.

```
static User.UserBuilder builder()

static User.UserBuilder withUserDetails
            (UserDetails userDetails)

static User.UserBuilder withUsername(String username)
```

UserBuilder Class

- Builds the user to be added.
- At minimum the username, password, and authorities should provided.
- The remaining attributes have reasonable defaults.

```
User.UserBuilderauthorities(GrantedAuthority... authorities)
Populates the authorities.
UserDetails build()
Builds UserDetails object.
User.UserBuilder disabled(boolean disabled)
Defines if the account is disabled or not.
User.UserBuilder password(String password)
Populates the password.
User.UserBuilder roles(String... roles)
Populates the roles.
User.UserBuilder username(String username)
Populates the username.
```

UserDetailsService Interface

- Core interface which loads user-specific data.
- It is used throughout the framework as a user DAO

```
UserDetails loadUserByUsername(String username)
```

Locates the user based on the username.

Password Encoding

- Spring Security introduces **DelegatingPasswordEncoder**, which solves all of the problems by:
  - ✓ Ensuring that passwords are encoded by using the current password storage recommendations
  - ✓ Allowing for validating passwords in modern and legacy formats
  - ✓ Allowing for upgrading the encoding in the future
- Spring Security uses DelegatingPasswordEncoder by default. However, you can customize this by exposing a **PasswordEncoder** as a Spring bean.
- You can easily construct an instance of DelegatingPasswordEncoder by using PasswordEncoderFactories:

```
PasswordEncoder passwordEncoder =
 PasswordEncoderFactories.createDelegatingPasswordEncoder();
```

Creating users with Password encoding

```
UserBuilder users = User.withDefaultPasswordEncoder();
User user = users.username("user")
                 .password("password")
                 .roles("USER")
                 .build();

User admin = users.username("admin")
                 .password("password")
                 .roles("USER","ADMIN")
                 .build();
```

- The above code does hash the password that is stored, but the passwords are still exposed in memory and in the compiled source code.
- For production, you should hash your passwords externally.

Method Security

- We can enable annotation-based security using the @**EnableMethodSecurity** annotation on any @**Configuration** instance.
- Adding an annotation like @**PreAuthroize** to a method (on a class or interface) would then limit the access to that method accordingly.

```
@Configuration
@EnableMethodSecurity
public class MySecurityConfiguration {
     // code
}
```

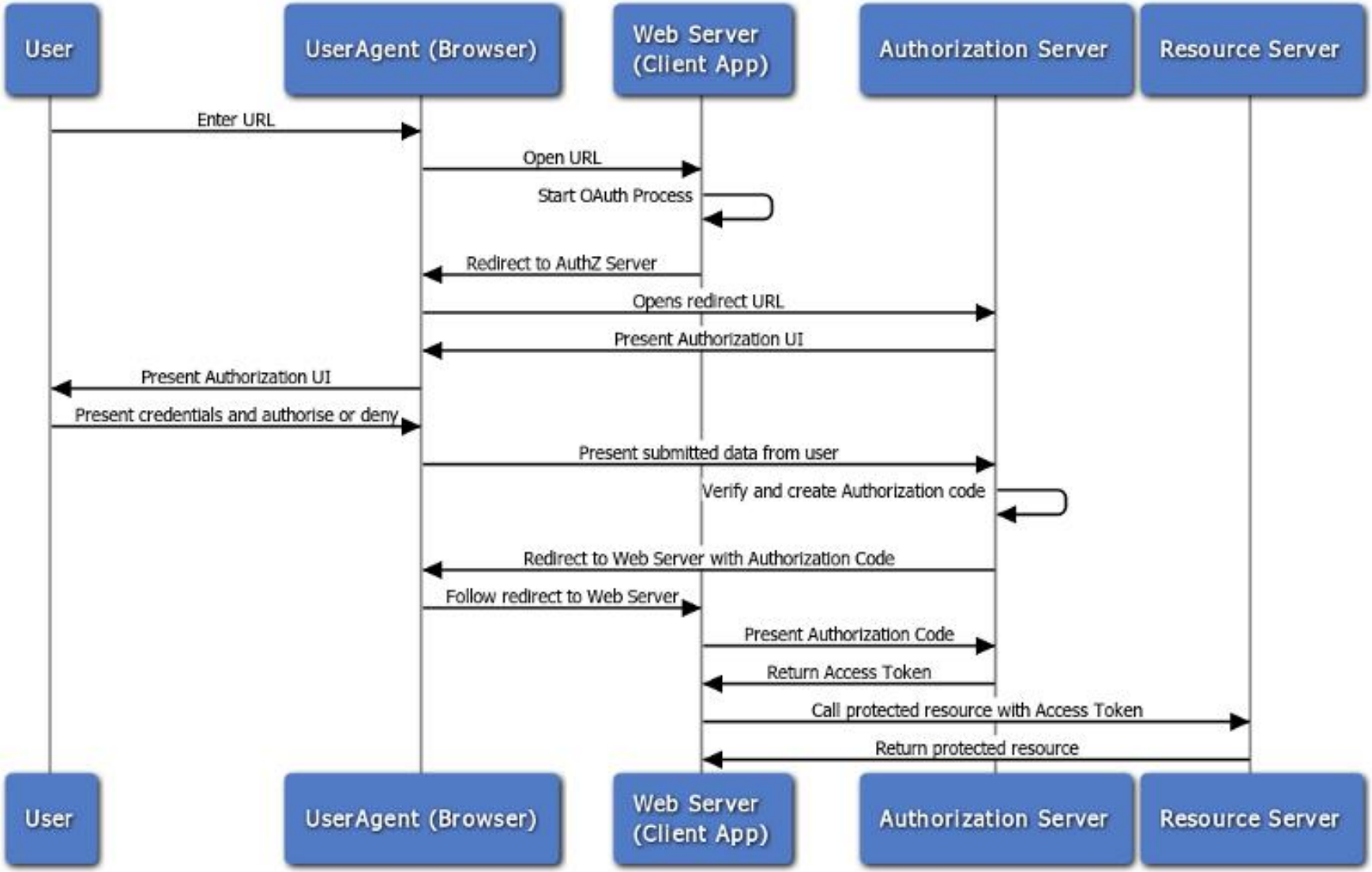
```
@RestController
public class UserController {

   @GetMapping("/createuser")
   @PreAuthorize("hasRole('ADMIN')")
   public String createUser() {
        //code
   }
}
```

OAuth 2

- OAuth stands for Open Authorization.
- OAuth 2.0 is the industry-standard protocol for authorization.
- Designed to allow website or application to access resources on behalf of a user.
- It works by delegating user authentication to the service that hosts a user account and authorizing third-party applications to access that user account.

OAuth 2 Flow


User
UserAgent (Browser)
Web Server (Client App)
Authorization Server
Resource Server
Enter URL
Open URL
Start OAuth Process
Redirect to AuthZ Server
Opens redirect URL
Present Authorization UI
Present Authorization UI
Present credentials and authorise or deny
Present submitted data from user
Verify and create Authorization code
Redirect to Web Server with Authorization Code
Follow redirect to Web Server
Present Authorization Code
Return Access Token
Call protected resource with Access Token
Return protected resource
User
UserAgent (Browser)
Web Server (Client App)
Authorization Server
Resource Server

OAuth 2 Flow

- An OAuth Access Token transaction requires three players:
  - ✓ End user,
  - ✓ Application (API),
  - ✓ Resource (service provider that has stored your privileged credentials).

- The application asks for authorization from the resource by providing the user's verified identity as proof.
- After the authorization has been authenticated, the resource grants an **Access Token** to the API, without having to divulge usernames or passwords.
- Tokens come with access permission for the API. These permissions are called scopes and each token will have an authorized scope for every API. The application gets access to the resource only to the extent the scope allows.

OAuth Roles

- **Resource Owner**: The resource owner is the user who authorizes an application to access their account. The application's access to the user's account is limited to the scope of the authorization granted (e.g. read or write access)
- **Client**: The client is the application that wants to access the user's account. Before it may do so, it must be authorized by the user, and the authorization must be validated by the API.
- **Resource Server**: The resource server hosts the protected user accounts.
- **Authorization Server**: The authorization server verifies the identity of the user then issues access tokens to the application.

Steps using Github for Authentication

- Create a Spring Boot Application with following dependencies.

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

- Add application to GitHub using *Settings -> Developer Settings -> New OAuth App*
- While adding new application in Github, you have to provide details of Application (next slide).
  - ✓ Name : Test
  - ✓ Homepage URL : http://localhost:8080
  - ✓ Application Description : Testing OAuth
  - ✓ Authorization Callback URL: http://localhost:8080/login/oauth2/code/github
- Github provides Client ID and Client Secret.
- Modify Spring Boot Application's properties file (application.properties) with details.

Registering App with GitHub

Register a new OAuth application

Application name *

Something users will recognize and trust.

Homepage URL *

The full URL to your application homepage.

Application description

Application description is optional

This is displayed to all users of your application.

Authorization callback URL *

Your application's callback URL. Read our OAuth documentation for more information.

☐ **Enable Device Flow**

Allow this OAuth App to authorize users via the Device Flow.

Read the Device Flow documentation for more information.

Register application Cancel

Application.Properties

- Add the following properties in application.properties file.

```
spring.security.oauth2.client.registration.springboottestapp.client-id=
<client-id>
spring.security.oauth2.client.registration.springboottestapp.client-secret=
<client-secret>
spring.security.oauth2.client.registration.springboottestapp.client-name=
Spring Boot Test App
spring.security.oauth2.client.registration.springboottestapp.provider=
github
spring.security.oauth2.client.registration.springboottestapp.scope=
user
spring.security.oauth2.client.registration.springboottestapp.redirect-uri=
http://localhost:8080/login/oauth2/code/github
```