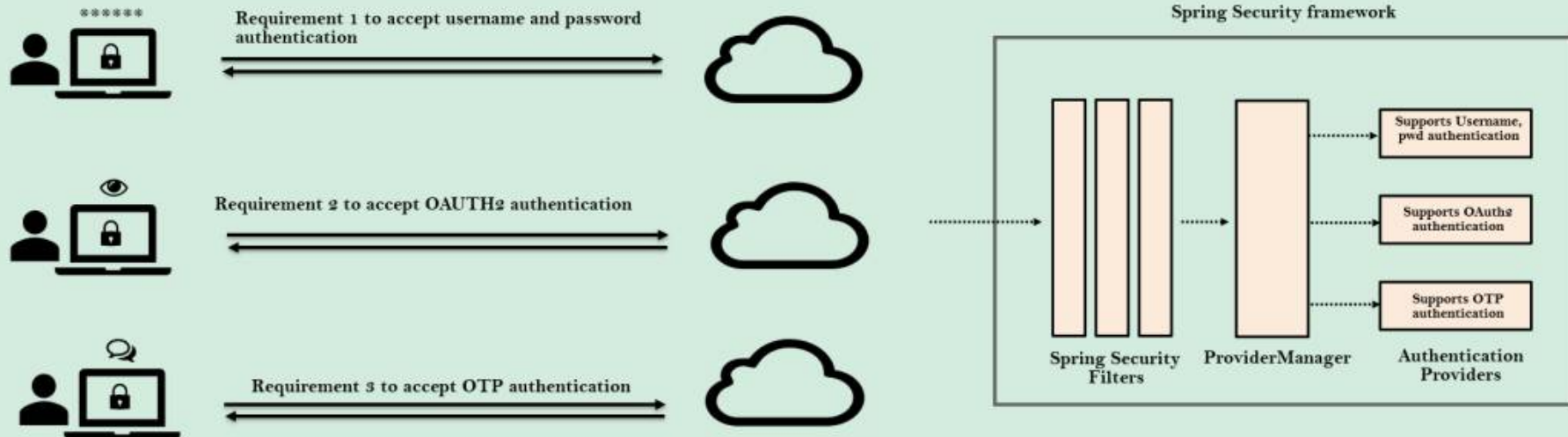


5-Authentication Providers Explained



AUTHENTICATION PROVIDER WHY DO WE NEED IT?

**eazy
bytes**



- ✓ The AuthenticationProvider in Spring Security takes care of the authentication logic. The default implementation of the AuthenticationProvider is to delegate the responsibility of finding the user in the system to a UserDetailsServiceImpl & PasswordEncoder for password validation. But if we have a custom authentication requirement that is not fulfilled by Spring Security framework, then we can build our own authentication logic by implementing the AuthenticationProvider interface.
- ✓ It is the responsibility of the ProviderManager which is an implementation of AuthenticationManager, to check with all the implementations of Authentication Providers and try to authenticate the user.

DETAILS OF AUTHENTICATION PROVIDER

Methods inside AuthenticationProvider Interface

```
public interface AuthenticationProvider {  
  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
  
    boolean supports(Class<?> authentication);  
}
```

- ★ The `authenticate()` method receives and returns authentication object. We can implement all our custom authentication logic inside `authenticate()` method.
- ★ The second method in the `AuthenticationProvider` interface is `supports(Class<?> authentication)`. You'll implement this method to return true if the current `AuthenticationProvider` supports the type of the `Authentication` object provided.

Kindly explain the above image in details with respect to Authentication provider



The images you provided explain the concept of **Authentication Providers** in the Spring Security framework. Below is a detailed explanation based on the content in the images:

1. Why Do We Need an Authentication Provider?

- Authentication is the process of verifying the identity of a user. Different systems and use cases require different types of authentication.
- Examples:
 - **Requirement 1:** Accept username and password-based authentication.
 - **Requirement 2:** Accept OAuth2-based authentication for federated login.
 - **Requirement 3:** Accept OTP-based authentication for enhanced security.

The **Spring Security Framework** provides a flexible and extensible way to handle these diverse authentication requirements using **Authentication Providers**.

Key Components:

- **Spring Security Filters:** These are part of the security filter chain and are responsible for intercepting HTTP requests and initiating the authentication process.
- **ProviderManager:** This manages multiple **Authentication Providers**, delegating the authentication process to the appropriate provider.
- **Authentication Providers:** Each provider handles a specific authentication mechanism (e.g., username/password, OAuth2, OTP).

2. How Authentication Provider Works

- **AuthenticationProvider Interface:** This is a contract in Spring Security that defines how to authenticate a user.
- It has two key methods:
 1. `authenticate(Authentication authentication)`: This is where the authentication logic resides. It receives an `Authentication` object (e.g., containing username and password) and returns an authenticated `Authentication` object upon successful validation or throws an `AuthenticationException` otherwise.
 2. `supports(Class<?> authentication)`: This method is used to determine whether the current provider can handle the type of `Authentication` object passed.

Example Workflow:

- A user submits credentials (username/password).
- The **ProviderManager** forwards the request to the relevant **AuthenticationProvider**.
- The provider authenticates the user and either:
 - Returns an authenticated `Authentication` object, or
 - Throws an `AuthenticationException` if validation fails.

3. Default vs. Custom Authentication

- **Default Implementation:** Spring Security provides a default implementation that uses a **UserDetailsService** and a **PasswordEncoder** for handling username/password authentication.
- **Custom Implementation:** If your requirements (e.g., OTP or OAuth2) are not met by the default implementation, you can create your own provider by implementing the `AuthenticationProvider` interface.

Example:

- For OTP-based authentication:
 - Implement the `AuthenticationProvider` interface.
 - Use the `authenticate()` method to validate the OTP entered by the user.
 - Use the `supports()` method to ensure this provider only handles OTP-based `Authentication` objects.

4. Responsibilities of ProviderManager

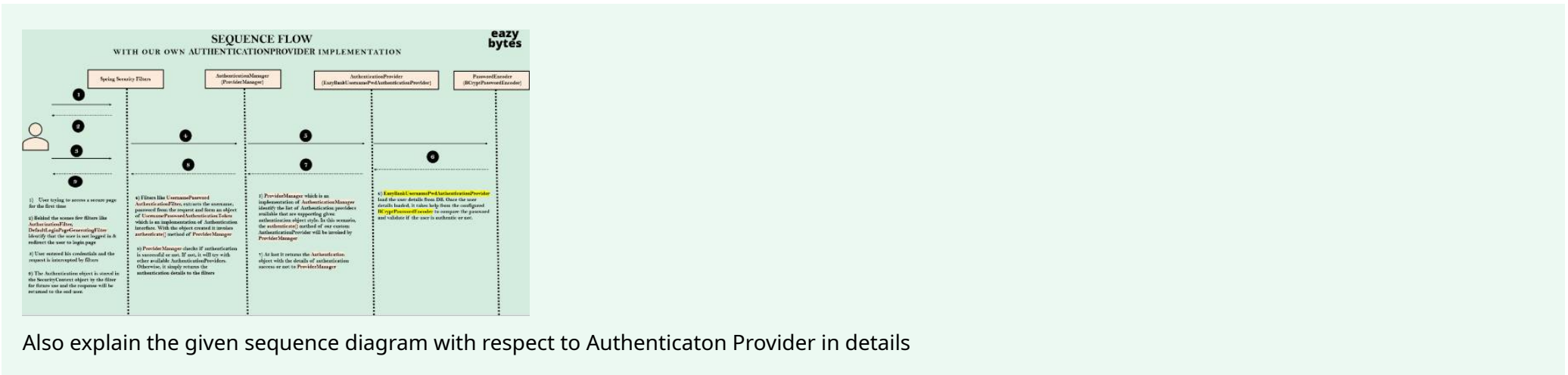
- The **ProviderManager** acts as the main entry point for authentication.
- It checks each configured **AuthenticationProvider** to find one that supports the given authentication type.
- If no provider can handle the authentication request, it throws an exception.

Summary:

- **Flexibility:** Spring Security's use of **Authentication Providers** allows the framework to support diverse authentication mechanisms.
- **Extensibility:** Developers can add custom authentication logic by implementing the `AuthenticationProvider` interface.
- **Central Management:** The **ProviderManager** ensures all authentication logic is managed centrally and cohesively.

If you'd like further clarification or examples, feel free to ask!





Also explain the given sequence diagram with respect to Authenticaton Provider in details



The sequence diagram provided explains the flow of a custom implementation of an **Authentication Provider** in Spring Security. Here's a detailed explanation of the flow:

Sequence Flow with Custom Authentication Provider

1. User Accesses a Secure Page

- The user tries to access a secured page for the first time.
- The request is intercepted by **Spring Security Filters**.

2. Filters Identification

- Filters such as:
 - **AuthorizationFilter**: Checks if the user has the required authorization.
 - **DefaultLoginPageGeneratingFilter**: Redirects the user to the login page if not authenticated.
- These filters detect that the user is not logged in.

3. User Submits Credentials

- The user provides their username and password on the login page and submits the credentials.
- The request is intercepted by the **UsernamePasswordAuthenticationFilter**.

4. Authentication Token Creation

- **UsernamePasswordAuthenticationFilter:**

- Extracts the username and password from the request.
- Creates a `UsernamePasswordAuthenticationToken` object, which implements the `Authentication` interface.
- Passes this token to the **ProviderManager** for authentication.

5. ProviderManager Identifies Authentication Providers

- The **ProviderManager** identifies the available **Authentication Providers** that can handle the type of `Authentication` object (in this case, `UsernamePasswordAuthenticationToken`).
- It delegates the authentication request to the appropriate provider. For example:
 - Here, the custom `EazyBankUsernamePwdAuthenticationProvider` is invoked.

6. Custom Authentication Provider Logic

- The **EazyBankUsernamePwdAuthenticationProvider:**
 - Retrieves user details (e.g., username, password) from the database or another data source.
 - Uses the configured **BCryptPasswordEncoder** to validate the provided password against the stored hash.

7. Authentication Result

- If the authentication is successful:
 - The **EazyBankUsernamePwdAuthenticationProvider** returns an `Authentication` object with the user details to the **ProviderManager**.
- If unsuccessful:
 - It throws an `AuthenticationException`.
 - The **ProviderManager** may try other **Authentication Providers**, if available.

8. ProviderManager Finalizes Authentication

- The **ProviderManager** confirms whether the authentication is successful.
- If not, it returns an error response.

9. Store Authentication and Respond

- Upon successful authentication:
 - The authenticated `Authentication` object is stored in the **SecurityContext**.
 - The response is returned to the user, granting access to the secure page.

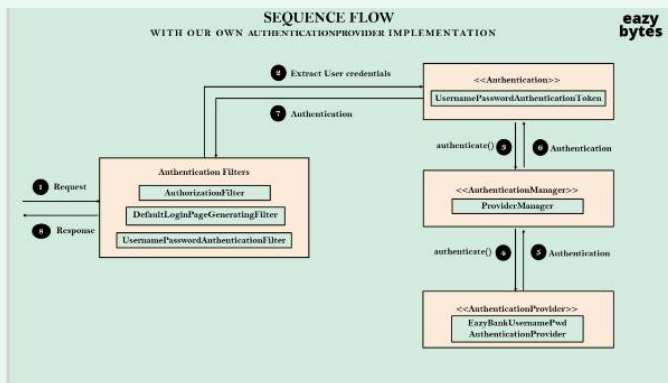
Key Components in the Diagram

- **Spring Security Filters:** Responsible for intercepting and processing HTTP requests.
- **AuthenticationManager (ProviderManager):** Manages and delegates authentication to configured **Authentication Providers**.
- **AuthenticationProvider:**

- Implements the actual authentication logic.
- Custom provider (**EazyBankUsernamePwdAuthenticationProvider**) handles the username and password validation.
- **PasswordEncoder**: Encodes and matches passwords securely. Here, **BCryptPasswordEncoder** is used.

Conclusion

This sequence flow illustrates how Spring Security's authentication process can be customized using a custom **AuthenticationProvider**. By implementing your own provider, you can handle authentication logic tailored to specific requirements (e.g., custom validation rules, integration with a database). The **ProviderManager** ensures a consistent process by coordinating between various authentication providers.



Kindly explain the above diagram. And also mention the steps to implement own Custom Authentication Provider with example



The provided diagram illustrates the **sequence flow** for implementing a custom **Authentication Provider** in Spring Security. It depicts how authentication requests are processed in a Spring Security application using custom authentication logic.

Explanation of the Diagram

1. Request:

- A user makes a request to access a secured resource.

2. Authentication Filters:

- The request passes through various filters in the **Spring Security Filter Chain**, such as:
 - **AuthorizationFilter**: Checks if the user has sufficient permissions.
 - **DefaultLoginPageGeneratingFilter**: Redirects unauthenticated users to a login page.
 - **UsernamePasswordAuthenticationFilter**: Extracts user credentials (username and password) from the request and creates a ``UsernamePasswordAuthenticationToken``.

3. Token Creation:

- The ``UsernamePasswordAuthenticationToken`` is created to encapsulate the credentials provided by the user.

4. ProviderManager Delegation:

- The **ProviderManager** (an implementation of ``AuthenticationManager``) receives the token and determines which **AuthenticationProvider** is capable of handling the token.

5. Custom AuthenticationProvider:

- The **EazyBankUsernamePwdAuthenticationProvider** (custom provider) is selected based on the token type.
- It performs the actual authentication by:
 - Retrieving user details from the database.
 - Validating the provided password using the configured ``PasswordEncoder``.

6. Authentication Response:

- The custom provider returns an authenticated ``Authentication`` object if the user is successfully authenticated.
- If authentication fails, it throws an ``AuthenticationException``.

7. AuthenticationManager Response:

- The **ProviderManager** receives the authentication result and passes it back to the filter.

8. Response:

- If authentication succeeds, the user is granted access to the requested resource.
- Otherwise, an error response (e.g., "401 Unauthorized") is sent.

Steps to Implement a Custom Authentication Provider

1. Create the Custom ``AuthenticationProvider``:

- Implement the ``AuthenticationProvider`` interface and define the ``authenticate()`` and ``supports()`` methods.

Example:


```
java
```

```
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.PasswordEncoder;

public class CustomAuthenticationProvider implements AuthenticationProvider {

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;

    public CustomAuthenticationProvider(UserDetailsService userDetailsService, PasswordEncoder passwordEncoder) {
        this.userDetailsService = userDetailsService;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String username = authentication.getName();
        String password = authentication.getCredentials().toString();

        UserDetails user = userDetailsService.loadUserByUsername(username);

        if (passwordEncoder.matches(password, user.getPassword())) {
            return new UsernamePasswordAuthenticationToken(username, password, user.getAuthorities());
        } else {
            throw new BadCredentialsException("Invalid username or password");
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

2. Configure the Custom Provider in Security Configuration:

- Register the custom `AuthenticationProvider` in the Spring Security configuration.

Example:

```
java
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class SecurityConfig {

    private final UserDetailsService userDetailsService;

    public SecurityConfig(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

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

    @Bean
    public AuthenticationProvider customAuthenticationProvider() {
        return new CustomAuthenticationProvider(userDetailsService, passwordEncoder());
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration configuration) throws Exception {
        return configuration.getAuthenticationManager();
    }
}
```

3. Define UserDetailsService:

- Implement a `UserDetailsService` to load user details from the database.

Example:

```
java
```

```
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
```

```
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        // Fetch user details from the database
        if ("user".equals(username)) {
            return User.builder()
                .username("user")
                .password("$2a$10$...") // BCrypt encrypted password
                .roles("USER")
                .build();
        }
        throw new UsernameNotFoundException("User not found");
    }
}
```

4. Test the Authentication:

- Use an endpoint that requires authentication and test the custom authentication logic.

By following these steps, you can implement and integrate a custom **AuthenticationProvider** into your Spring Security application, tailored to your specific authentication requirements.



Write the logic with implementing own AuthrenticationManager
and Authenticatio Provider



To implement your own `AuthenticationManager` and `AuthenticationProvider` in Spring Security, you need to override the default implementations and create custom authentication logic. Here's how you can do it:

Custom `AuthenticationProvider`

The `AuthenticationProvider` is responsible for performing the actual authentication logic. Below is an implementation:

```
java
```

```
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.PasswordEncoder;

public class CustomAuthenticationProvider implements AuthenticationProvider {

    private final UserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;

    public CustomAuthenticationProvider(UserDetailsService userDetailsService, PasswordEncoder passwordEncoder) {
        this.userDetailsService = userDetailsService;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String username = authentication.getName();
        String password = authentication.getCredentials().toString();

        UserDetails user = userDetailsService.loadUserByUsername(username);

        if (user != null && passwordEncoder.matches(password, user.getPassword())) {
            return new UsernamePasswordAuthenticationToken(username, password, user.getAuthorities());
        } else {
            throw new AuthenticationException("Invalid username or password") {};
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

Custom `AuthenticationManager`

The `AuthenticationManager` manages authentication by delegating the task to an `AuthenticationProvider`.

java

```
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;

import java.util.List;

public class CustomAuthenticationManager implements AuthenticationManager {

    private final List<AuthenticationProvider> providers;

    public CustomAuthenticationManager(List<AuthenticationProvider> providers) {
        this.providers = providers;
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        for (AuthenticationProvider provider : providers) {
            if (provider.supports(authentication.getClass())) {
                return provider.authenticate(authentication);
            }
        }
        throw new AuthenticationException("No suitable AuthenticationProvider found") {};
    }
}
```

Integration in Configuration

You can integrate the custom `AuthenticationManager` and `AuthenticationProvider` into your Spring Security configuration.

java

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

import java.util.Collections;
```

```
@Configuration
public class SecurityConfig {

    private final UserDetailsService userDetailsService;

    public SecurityConfig(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

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

    @Bean
    public AuthenticationProvider customAuthenticationProvider() {
        return new CustomAuthenticationProvider(userDetailsService, passwordEncoder());
    }

    @Bean
    public AuthenticationManager customAuthenticationManager() {
        return new CustomAuthenticationManager(Collections.singletonList(customAuthenticationProvider()));
    }
}
```

Example of `UserDetailsService`

The `UserDetailsService` fetches user details from a data source.

```
java

import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        // Mock user data
        if ("user".equals(username)) {
            return User.builder()

```

```
        .username("user")
        .password("$2a$10$...") // BCrypt-encoded password
        .roles("USER")
        .build();
    }
    throw new UsernameNotFoundException("User not found");
}
```

Testing

To test the custom implementation:

- Configure a secured endpoint.
- Use a client (e.g., Postman) or a browser to authenticate with the username and password.
- Validate that the `CustomAuthenticationProvider` and `CustomAuthenticationManager` are invoked.

This setup gives you full control over the authentication process and is highly flexible for custom authentication scenarios.



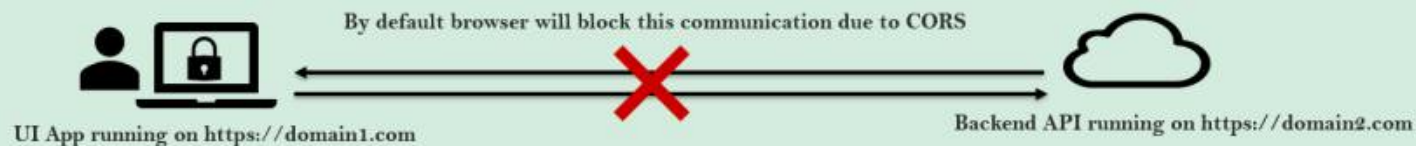
CROSS-ORIGIN RESOURCE SHARING (CORS)

CORS is a protocol that enables scripts running on a browser client to interact with resources from a different origin. For example, if a UI app wishes to make an API call running on a different domain, it would be blocked from doing so by default due to CORS. It is a specification from W3C implemented by most browsers.

So CORS is not a security issue/attack but the default protection provided by browsers to stop sharing the data/communication between different origins.

"other origins" means the URL being accessed differs from the location that the JavaScript is running from, by having:

- a different scheme (HTTP or HTTPS)
- a different domain
- a different port



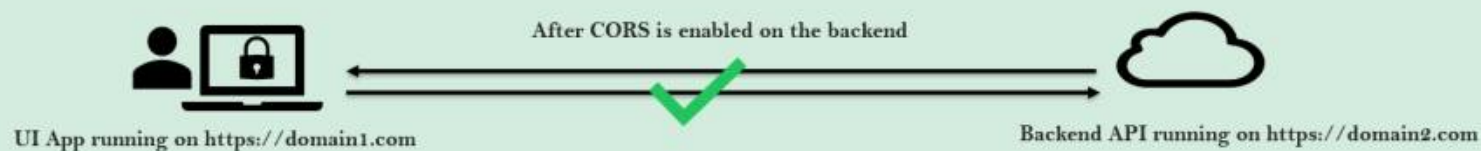
SOLUTION TO HANDLE CORS

If we have a valid scenario, where a Web APP UI deployed on a server is trying to communicate with a REST service deployed on another server, then these kind of communications we can allow with the help of `@CrossOrigin` annotation. `@CrossOrigin` allows clients from any domain to consume the API.

`@CrossOrigin` annotation can be mentioned on top of a class or method like mentioned below,

```
@CrossOrigin(origins = "http://localhost:4200") // Will allow on specified domain
```

```
@CrossOrigin(origins = "**") // Will allow any domain
```



What is CORS. Kindly explain it in details along with detailed explanation



What is CORS (Cross-Origin Resource Sharing)?

CORS is a protocol or security mechanism that allows restricted resources on a web page (such as APIs, fonts, or AJAX requests) to be accessed from a domain different from the one on which the web page is running. CORS is a W3C standard implemented by most browsers to ensure secure interaction between different origins.

Key Concepts

1. Origin:

- Defined by three components: **scheme**, **domain**, and **port**.
- Two URLs have the same origin if all three components match.

Example:

- ``https://domain1.com`` and ``https://domain2.com`` are **different origins** because their domains differ.

2. Default Browser Behavior:

- By default, browsers block cross-origin requests for security reasons (known as the **same-origin policy**).
- This restriction prevents unauthorized access to resources hosted on a different domain, ensuring malicious websites cannot fetch sensitive data from other domains.

3. What Triggers CORS? CORS is triggered when:

- A web app hosted on ``https://domain1.com`` tries to communicate with a server hosted on ``https://domain2.com``.
- Differences may occur in:
 - **Scheme:** ``http://`` vs ``https://``
 - **Domain:** ``domain1.com`` vs ``domain2.com``
 - **Port:** ``https://domain1.com:8080`` vs ``https://domain1.com:9090``

How CORS Works

1. CORS Preflight Requests:

- For certain types of requests (like ``PUT``, ``DELETE``, or requests with custom headers), browsers send a **preflight request** using the ``OPTIONS`` HTTP method.
- The server must respond with specific headers indicating whether the cross-origin request is allowed.

Example:

```
bash
```

```
OPTIONS /api/resource HTTP/1.1
Origin: https://domain1.com
Access-Control-Request-Method: POST
```

Response:

```
makefile
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://domain1.com
Access-Control-Allow-Methods: POST, GET, OPTIONS
```

2. Server's Role:

- The server determines which origins, HTTP methods, and headers are permitted.
- It responds with specific CORS headers like:
 - ``Access-Control-Allow-Origin``
 - ``Access-Control-Allow-Methods``
 - ``Access-Control-Allow-Headers``

Why is CORS Important?

- **Security:** Prevents unauthorized domains from accessing your resources.
- **Flexibility:** Allows legitimate cross-origin requests for APIs and other resources.

Solution to Handle CORS

1. Enable CORS in Backend APIs:

- Add CORS support on the server using headers or framework-specific configurations.
- In Spring Boot, use the `@CrossOrigin`` annotation.

2. Example of CORS Configuration in Spring Boot:

Using `@CrossOrigin` Annotation:

```
java

@RestController
@RequestMapping("/api")
@CrossOrigin(origins = "http://localhost:4200") // Allows only a specific origin
public class MyController {
    @GetMapping("/data")
    public String getData() {
        return "CORS Enabled";
    }
}
```

Global CORS Configuration:

```
java

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                    .allowedOrigins("http://localhost:4200")
                    .allowedMethods("GET", "POST", "PUT", "DELETE");
            }
        };
    }
}
```

3. Allowing All Origins (Not recommended for production):

```
java
```

```
@CrossOrigin(origins = "*")
```

Diagram Explanation

- **Without CORS:**

- A UI app hosted on `https://domain1.com` tries to access an API hosted on `https://domain2.com`.
- The browser blocks the request due to the same-origin policy.

- **With CORS Enabled:**

- After enabling CORS on the backend, the server responds with headers allowing cross-origin requests.
- The UI app can now communicate with the backend API successfully.

Key Takeaways

- **CORS is not a security vulnerability;** it is a security feature that protects resources from unauthorized access.
- Always configure CORS on the server to allow only trusted origins.
- Use the ``@CrossOrigin`` annotation or global configurations in Spring Boot to enable CORS.



SOLUTION TO HANDLE CORS

Instead of mentioning `@CrossOrigin` annotation on all the controllers inside our web app, we can define CORS related configurations globally using Spring Security like shown below,

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.cors().configurationSource(new CorsConfigurationSource() {
        @Override
        public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
            CorsConfiguration config = new CorsConfiguration();
            config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));
            config.setAllowedMethods(Collections.singletonList("*"));
            config.setAllowCredentials(true);
            config.setAllowedHeaders(Collections.singletonList("*"));
            config.setMaxAge(3600L);
            return config;
        }
    })
    .authorizeHttpRequests()
        .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
        .requestMatchers("/notices", "/contact", "/register").permitAll()
    .and().formLogin()
    .and().httpBasic()
    .return http.build();
}
```

Kindly explain the above image in details with detailed explanation



The provided image illustrates how to handle **CORS (Cross-Origin Resource Sharing)** configuration globally in a Spring Security application using the `SecurityFilterChain`. Let me explain it in detail:

Explanation of the Code

1. Global CORS Configuration:

- Instead of adding the `@CrossOrigin` annotation on individual controllers or methods, the image demonstrates how to configure CORS settings globally using Spring Security.
- This approach ensures a centralized configuration for all HTTP requests in the application.

2. `SecurityFilterChain` Bean:

- The `SecurityFilterChain` bean defines the security configuration for the application.
- Inside this configuration, CORS settings are specified using the `cors()` method and a `CorsConfigurationSource`.

3. Custom CORS Configuration:

- A custom implementation of `CorsConfigurationSource` is provided to define CORS rules.
- **Key CORS Properties Defined:**
 - **Allowed Origins:**

```
java

config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));
```

This allows requests only from `http://localhost:4200`.

- **Allowed Methods:**

```
java

config.setAllowedMethods(Collections.singletonList("*"));
```

All HTTP methods (`GET`, `POST`, `PUT`, `DELETE`, etc.) are permitted.

- **Allowed Headers:**

```
java

config.setAllowedHeaders(Collections.singletonList("*"));
```

All headers in the request are permitted.

- **Allow Credentials:**

```
java

config.setAllowCredentials(true);
```

Allows sending cookies or credentials with the request.

- **Max Age:**

```
java

config.setMaxAge(3600L);
```

Caches the CORS preflight response for 3600 seconds (1 hour).

4. Authorization Rules:

- Following the CORS configuration, the image defines HTTP authorization rules:
 - **Authenticated Endpoints:**

```
java

.requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
```

These endpoints require the user to be authenticated.

- **Public Endpoints:**

```
java

.requestMatchers("/notices", "/contact", "/register").permitAll()
```

These endpoints are accessible to everyone without authentication.

5. Authentication Mechanisms:

- Form-based login and HTTP Basic authentication are enabled:

```
java

.and().formLogin()
.and().httpBasic();
```

6. Returning the Configured `HttpSecurity`:

- The configured `HttpSecurity` object is returned at the end:

```
java

return http.build();
```

Advantages of Global CORS Configuration

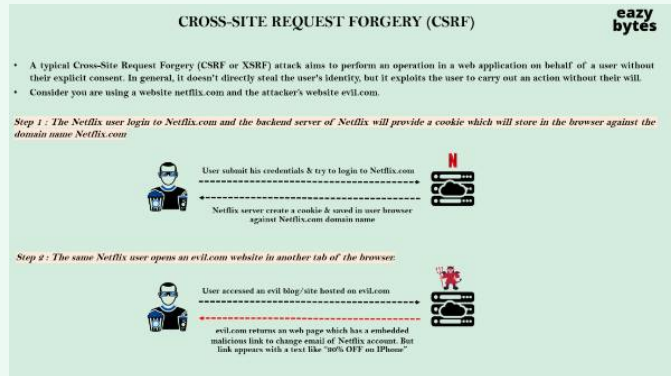
- **Centralized Management:**
 - CORS rules are managed in one place, making it easier to maintain and update.
 - **Consistency:**
 - Ensures uniform CORS behavior across all endpoints.
 - **Simplified Controllers:**
 - Reduces the need for repetitive `@CrossOrigin` annotations on individual controllers or methods.
-

Key Takeaways

- This global approach is ideal for projects where multiple endpoints interact with front-end clients hosted on different origins.
- By allowing credentials and customizing origins, methods, and headers, the configuration is both secure and flexible.
- Always ensure you restrict `setAllowedOrigins()` to trusted domains in production to avoid exposing your application to security risks. Using `""` for all origins is only recommended for development or when your API is intended to be publicly accessible.

This setup makes your Spring Security application CORS-compliant while maintaining robust security practices.





What is CSRF. Kindly explain the above image in details



What is CSRF (Cross-Site Request Forgery)?

Cross-Site Request Forgery (CSRF) is a type of attack that tricks an authenticated user into performing an unintended action on a web application without their explicit consent. While the attacker cannot directly access the user's data or credentials, they exploit the user's authenticated session to perform malicious actions.

Detailed Explanation of the Image

The image explains the CSRF attack flow using an example involving two websites:

1. **Netflix.com (the victim's legitimate site).**
2. **Evil.com (the attacker's malicious site).**

Steps in a CSRF Attack

Step 1: User Logs into Netflix

- The user visits `Netflix.com` and submits their credentials.
- After successful login:

- Netflix creates a **session cookie** and stores it in the user's browser.
- This cookie is tied to the ``Netflix.com`` domain and is used for authentication in subsequent requests.

Step 2: User Visits Malicious Website (Evil.com)

- In another browser tab, the user visits ``Evil.com``, a site controlled by an attacker.
- Evil.com contains a malicious script or link embedded in its content. This link:
 - **Targets Netflix.com.**
 - Contains commands or actions that the attacker wants to perform (e.g., changing the user's email, transferring funds).

Step 3: Exploitation

- When the user interacts with the malicious link (e.g., clicking on a "90% OFF on iPhone" button):
 - The browser automatically sends the **session cookie** for Netflix (since the browser includes cookies for all authenticated requests).
 - Netflix receives the request as if it were made by the legitimate user and executes the action (e.g., email change).
 - The malicious action is performed without the user's consent because Netflix relies on the session cookie for authentication, not the origin of the request.
-

Characteristics of CSRF

- **Exploitation of Trust:**
 - The server trusts the authenticated session of the user and executes the request.
 - **No Need for User Credentials:**
 - The attacker does not need to steal the user's credentials; they exploit the user's active session.
-

Why is CSRF Dangerous?

- CSRF can lead to:
 - Changing sensitive account information (e.g., passwords, emails).
 - Unauthorized fund transfers in financial applications.
 - Submitting fraudulent data or requests.
-

How to Prevent CSRF Attacks

1. CSRF Tokens:

- Use a **CSRF token** (unique, per-session token) in forms and AJAX requests.
- The server validates the token before processing the request.
- Example in Spring Security:

```
java

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

2. SameSite Cookies:

- Configure cookies with the `SameSite` attribute to prevent them from being sent with cross-origin requests.
- Example:
 - `SameSite=Strict` or `SameSite=Lax`.

3. Verify HTTP Referer Header:

- Validate the `Referer` header to ensure the request originated from the same domain.

4. Avoid GET Requests for Sensitive Actions:

- Ensure sensitive operations (e.g., password change) use POST or other non-idempotent methods.

5. User Awareness:

- Educate users to avoid clicking on suspicious links or visiting untrusted websites.

Key Takeaways from the Image

- CSRF attacks exploit an authenticated user's session cookie to perform unauthorized actions.
- Even though the user interacts with a malicious site (e.g., Evil.com), their authenticated session with Netflix.com is leveraged for exploitation.
- Effective CSRF prevention requires implementing server-side validation mechanisms like CSRF tokens, SameSite cookies, and secure HTTP headers.

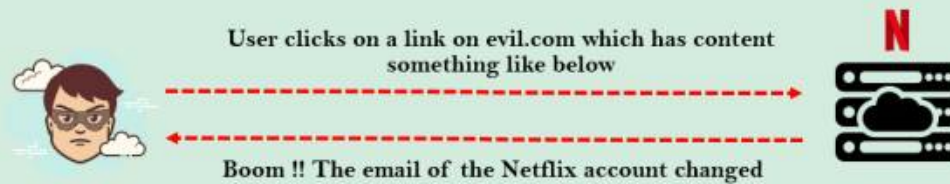
This example highlights the importance of robust CSRF protections in web applications.



CROSS-SITE REQUEST FORGERY (CSRF)

bytes

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com, the backend server of Netflix.com can't differentiate from where the request came. So here the evil.com forged the request as if it is coming from a Netflix.com UI page.



```
<form action="https://netflix.com/changeEmail"
  method="POST" id="form">
  <input type="hidden" name="email" value="user@evil.com">
</form>

<script>
  document.getElementById('form').submit()
</script>
```

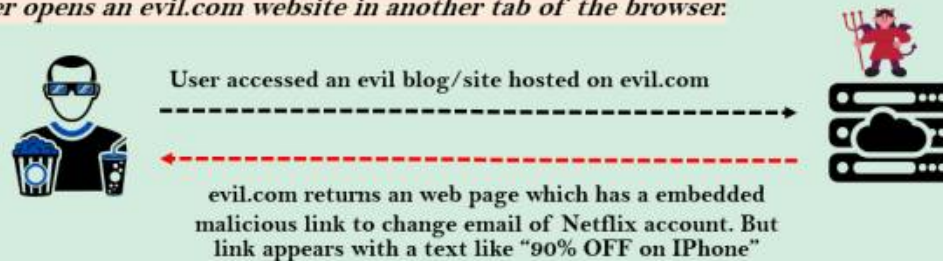
SOLUTION TO CSRF ATTACK

- To defeat a CSRF attack, applications need a way to determine if the HTTP request is legitimately generated via the application's user interface. The best way to achieve this is through a CSRF token. A CSRF token is a secure random token that is used to prevent CSRF attacks. The token needs to be unique per user session and should be of large random value to make it difficult to guess.
- Let's see how this solve CSRF attack by taking the previous Netflix example again,

Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com along with a randomly generated unique CSRF token for this particular user session. CSRF token is inserted within hidden parameters of HTML forms to avoid exposure to session cookies.

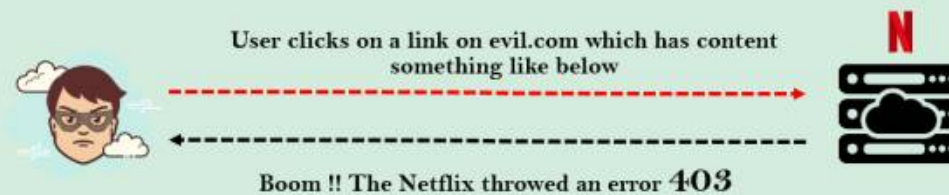


Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.



SOLUTION TO CSRF ATTACK

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com. This time the Netflix.com backend server expects CSRF token along with the cookie. The CSRF token must be same as initial value generated during login operation



The CSRF token will be used by the application server to verify the legitimacy of the end-user request if it is coming from the same App UI or not. The application server rejects the request if the CSRF token fails to match the test.

Kindly explain the above image in details with detailed explanation



The series of images explain **Cross-Site Request Forgery (CSRF)** attacks and the solution to mitigate them using **CSRF tokens**. Let's dive into the details.

CSRF Attack: Detailed Explanation

Step 3: How the Attack Happens

- A user is logged into a legitimate site, `Netflix.com`.
- While browsing, they unknowingly visit a malicious site, `Evil.com`.
- The malicious site has an embedded form or script:

```
html

<form action="https://netflix.com/changeEmail" method="POST" id="form">
  <input type="hidden" name="email" value="user@evil.com">
</form>
<script>
  document.getElementById('form').submit();
</script>
```

- This script automatically submits a request to `Netflix.com` to change the user's email to `user@evil.com`.
- Since the user's browser still has the authentication cookie for `Netflix.com`, the server processes the request as though it was initiated by the user.
- Result: The malicious action is executed without the user's consent.

Solution to CSRF Attack

Use of CSRF Tokens

To mitigate CSRF attacks, the application must generate and validate a unique **CSRF token** for every user session. Here's how it works:

Step 1: Generating a CSRF Token

- When the user logs into `Netflix.com`:
 - The server generates a unique, random **CSRF token** for that user session.
 - The token is included as a hidden field in all HTML forms or headers for AJAX requests.

Step 2: Embedding the CSRF Token

- For each form or request, the CSRF token is embedded:

```
html

<form action="https://netflix.com/changeEmail" method="POST">
  <input type="hidden" name="_csrf" value="123456ABCDEF">
  <input type="text" name="email" value="user@netflix.com">
  <button type="submit">Submit</button>
</form>
```

- The browser sends the CSRF token along with the request.

Step 3: Server Validation

- When the server receives the request:
 - It compares the CSRF token from the request with the token stored in the session or database.
 - If the token matches, the request is deemed legitimate and processed.
 - If the token is missing or invalid, the server rejects the request (e.g., with a **403 Forbidden** error).

Why CSRF Tokens Work

- The malicious site, `Evil.com`, cannot generate a valid CSRF token because:
 - Tokens are random and unique to each session.
 - Tokens are generated and validated by the legitimate site (`Netflix.com`) and are not accessible to other origins.

Outcome of CSRF Prevention

- If a user clicks on a malicious link on `Evil.com`:
 - The request sent to `Netflix.com` does not include a valid CSRF token.
 - The server rejects the request with a **403 Forbidden** error, ensuring no harm is done.

Implementation in Spring Security

In Spring Security, enabling CSRF protection is straightforward:

1. Default CSRF Protection:

- CSRF protection is enabled by default in Spring Security.
- It applies to all POST, PUT, DELETE, and other non-idempotent requests.

2. Customizing CSRF Protection:

- Configure CSRF protection to include custom headers or specific endpoints:

```
java

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf()
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            .and()
            .authorizeRequests()
            .antMatchers("/login", "/register").permitAll()
            .anyRequest().authenticated();
    }
}
```

3. Including CSRF Tokens in Forms:

- Spring automatically includes CSRF tokens in forms rendered by Thymeleaf:

```
html

<form action="/changeEmail" method="POST">
    <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
    <input type="text" name="email" />
    <button type="submit">Change Email</button>
</form>
```

Key Takeaways

- CSRF attacks exploit the trust a web application places in the user's browser session.
- CSRF tokens ensure that every request is intentional and originates from the legitimate application.
- Enabling CSRF protection in frameworks like Spring Security is essential to prevent these attacks.

This is how CSRF tokens protect applications from unauthorized actions and secure user data effectively.



DISABLE CSRF PROTECTION INSIDE SPRING SECURITY

**easy
bytes**

NOT RECOMMENDED
FOR PRODUCTION

By default Spring Security block all HTTP POST, PUT, DELETE, PATCH operations with an error of 403, if there is no CSRF solution implemented inside a web application. We can change this default behaviour by disabling the CSRF protection provided by Spring Security.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeHttpRequests()
            .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
            .requestMatchers("/notices", "/contact", "/register").permitAll()
        .and().formLogin()
        .and().httpBasic();
    return http.build();
}
```

CSRF ATTACK SOLUTION

INSIDE SPRING SECURITY

easy
bytes

With the below configuration of Spring Security, we can let the framework to generate a random CSRF token which can be sent to UI after successful login. The same token need to be sent by UI for every subsequent requests it is making to backend. For certain paths, we can disable CSRF with the help of ignoringRequestMatchers.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    CsrfTokenRequestAttributeHandler requestHandler = new CsrfTokenRequestAttributeHandler();
    requestHandler.setCsrfRequestAttributeName("_csrf");

    http.securityContext().requireExplicitSave(false)
        .and().sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
        .cors().configurationSource(new CorsConfigurationSource() {
            @Override
            public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {...}
        }).and().csrf((csrf) -> csrf.csrfTokenRequestHandler(requestHandler).ignoringRequestMatchers("/contact", "/register")
        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
        .addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
        .authorizeHttpRequests()
        .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
        .requestMatchers("/notices", "/contact", "/register").permitAll()
        .and().formLogin()
        .and().httpBasic();
    return http.build();
}
```

Kindly explain the above image in details with detailed explanation



The provided images demonstrate two approaches to managing **CSRF (Cross-Site Request Forgery)** protection in a Spring Security application: disabling CSRF entirely (not recommended for production) and implementing a robust CSRF solution with token generation.

1. Disabling CSRF Protection in Spring Security

Code Overview

The first image shows the configuration to disable CSRF protection:

```
java

@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeHttpRequests()
        .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
        .requestMatchers("/notices", "/contact", "/register").permitAll()
        .and().formLogin()
        .and().httpBasic();
    return http.build();
}
```

Details

1. `http.csrf().disable()`:

- Disables CSRF protection globally in the application.
- This allows HTTP POST, PUT, DELETE, and PATCH operations without requiring CSRF tokens.

2. Authorization Rules:

- Specifies that certain endpoints (e.g., `/myAccount`, `/myBalance`) require authentication.
- Public endpoints like `/notices`, `/contact`, and `/register` are accessible without authentication.

3. Authentication Mechanisms:

- Enables form-based login and HTTP Basic authentication.

Key Considerations

- **Not Recommended for Production:**
 - Disabling CSRF leaves the application vulnerable to CSRF attacks.
 - Any request with a valid session cookie can execute potentially harmful actions on behalf of the user.
- **Use Case:**
 - This configuration is only suitable for APIs where CSRF protection is unnecessary, such as stateless REST APIs using tokens (e.g., JWT).

2. Robust CSRF Solution with Spring Security

Code Overview

The second image demonstrates a configuration to enable CSRF protection with token generation:

```
java

@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    CsrfTokenRequestAttributeHandler requestHandler = new CsrfTokenRequestAttributeHandler();
    requestHandler.setCsrfRequestAttributeName("_csrf");

    http.securityContext().requireExplicitSave(false)
        .and().sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
        .cors().configurationSource(new CorsConfigurationSource() {
            @Override
            public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
                // Define CORS configuration
                return new CorsConfiguration();
            }
        })
        .and().csrf(csrf -> csrf.csrfTokenRequestHandler(requestHandler)
            .ignoringRequestMatchers("/contact", "/register")
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
        .addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
        .authorizeHttpRequests()
        .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
        .requestMatchers("/notices", "/contact", "/register").permitAll()
        .and().formLogin()
        .and().httpBasic();
    return http.build();
}
```

Details

1. CSRF Token Generation:

- The ``CsrfTokenRequestAttributeHandler`` generates a unique CSRF token and associates it with each session.
- The token is added to forms or HTTP requests via a cookie.

2. CORS Configuration:

- Enables Cross-Origin Resource Sharing (CORS) to allow specific origins or methods.
- A ``CorsConfigurationSource`` defines the allowed origins, methods, headers, etc.

3. Ignoring CSRF for Certain Endpoints:

- CSRF protection is bypassed for specific endpoints (``/contact`` and ``/register``) using:

```
java

    ignoringRequestMatchers("/contact", "/register")
```

4. CSRF Token Repository:

- Stores the CSRF token in cookies with:

```
java

    CookieCsrfTokenRepository.withHttpOnlyFalse()
```

5. Custom CSRF Filter:

- Adds a filter (``CsrfCookieFilter``) to manage CSRF tokens effectively.

6. Authorization Rules:

- Secured endpoints (e.g., ``/myAccount``, ``/myBalance``) require authentication.
- Public endpoints (``/notices``, ``/contact``, ``/register``) are accessible to all.

Comparison

Aspect	Disabling CSRF	CSRF Solution with Token
Security	Vulnerable to CSRF attacks	Protects against CSRF attacks
Use Case	Stateless APIs using JWT	Applications with session-based authentication
Ease of Implementation	Simple (<code>http.csrf().disable()</code>)	Requires configuration for token handling
Customization	No customization available	Allows token-based validation and exclusions

Key Takeaways

- **Disabling CSRF:**
 - Should be avoided in applications using session-based authentication.
 - Suitable for token-based APIs where CSRF protection is unnecessary.
- **CSRF Protection with Tokens:**
 - Adds an additional layer of security by ensuring every request is legitimate.
 - Essential for protecting applications from CSRF attacks.

By carefully implementing CSRF protection based on the application's architecture, you can safeguard user data and ensure secure request handling.