

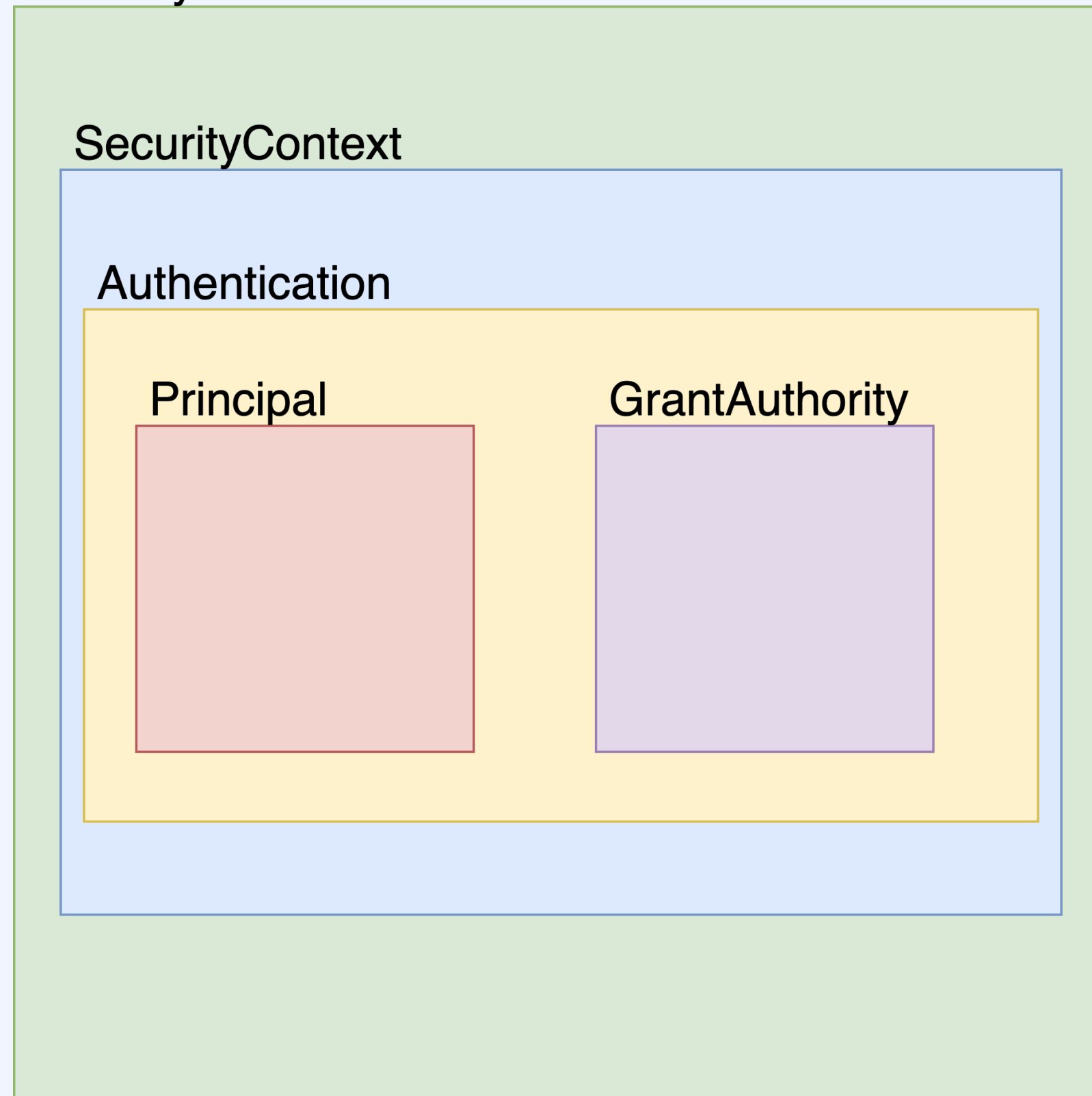
Spring Security

3 아키텍처

Spring Security의 내부구조

3.
아키텍처

SecurityContextHolder



그림으로 보는 내부구조

```
SecurityContext context = SecurityContextHolder.getContext(); // Security Context
Authentication authentication = context.getAuthentication(); // authentication
authentication.getPrincipal();
authentication.getAuthorities();
authentication.getCredentials();
authentication.getDetails();
authentication.isAuthenticated();
```

코드로 보는 내부구조

SecurityContextHolder -> SecurityContext -> Authentication -> Principal & GrantAuthority

Spring Security의 내부구조

3.

아키텍처

SecurityContextHolder

SecurityContextHolder는 SecurityContext를 제공하는 **static 메소드(getContext)**를 지원합니다.

SecurityContext

SecurityContext 는 접근 주체와 인증에 대한 정보를 담고 있는 Context 입니다.

즉, Authentication 을 담고 있습니다.

Authentication

Principal과 GrantAuthority를 제공합니다.

인증이 이루어 지면 해당 Authentication이 저장됩니다.

Principal

유저에 해당하는 정보입니다.

대부분의 경우 Principal로 UserDetails를 반환합니다.

GrantAuthority

ROLE_ADMIN, ROLE_USER등 Principal이 가지고 있는 권한을 나타냅니다.

prefix로 'ROLE_' 이 붙습니다.

인증 이후에 인가를 할 때 사용합니다.

권한은 여러개 일수 있기 때문에 Collection<GrantedAuthority>형태로 제공합니다.

ex) ROLE_DEVELOPER, ROLE_ADMIN

Spring Security는 이런 정보들을 사용해서
인증, 인가를 판단합니다.

ThreadLocal

우리가 WebMVC 기반으로 프로젝트를 만든다는 가정하에 대부분의 경우에는 요청 1개에 Thread 1개가 생성됩니다.

요청1개 : Thread1개

이때 ThreadLocal을 사용하면 Thread마다 고유한 공간을 만들수가 있고 그곳에 SecurityContext를 저장할 수 있습니다.

요청1개 : Thread1개 : Security Context1개

그러나 ThreadLocal만 강제로 사용해야하는 것은 아니며 원하면 SecurityContext 공유 전략을 바꿀 수 있습니다.

MODE_THREADLOCAL

ThreadLocalSecurityContextHolderStrategy를 사용합니다.

ThreadLocal을 사용하여 같은 Thread안에서 SecurityContext를 공유합니다.

기본 설정 모드입니다.

MODE_INHERITABLETHREADLOCAL

InheritableThreadLocalSecurityContextHolderStrategy를 사용합니다.,
InheritableThreadLocal을 사용하여 자식 Thread까지도 SecurityContext를 공유합니다.

MODE_GLOBAL

GlobalSecurityContextHolderStrategy를 사용합니다.

Global로 설정되어 애플리케이션 전체에서 SecurityContext를 공유합니다.

ThreadLocal

3.

아키텍처

Spring Security의 기본적인 Security Context 관리 전략은 ThreadLocal을 사용하는 ThreadLocalSecurityContextHolderStrategy 입니다.

변수는 지역변수, 전역변수와 같은 유효한 Scope을 가집니다. 마찬가지로 ThreadLocal은 Thread마다 고유한 영역을 가지고 있는 곳에 저장된 변수로 각각의 Thread안에서 유효한 변수입니다. 일반적인 서버의 경우에는 외부로부터 요청이 오면 그 요청마다 Thread 1개가 할당됩니다. 따라서 ThreadLocal로 SecurityContext를 관리하게 되면 SecurityContext는 요청마다 독립적으로 관리될 수 있습니다.

```
A ThreadLocal-based implementation of SecurityContextHolderStrategy.
See Also: ThreadLocal, org.springframework.security.core.context.web.
SecurityContextPersistenceFilter

final class ThreadLocalSecurityContextHolderStrategy implements SecurityContextHolderStrategy {

    private static final ThreadLocal<SecurityContext> contextHolder = new ThreadLocal<>();

    @Override
    public void clearContext() { contextHolder.remove(); }

    @Override
    public SecurityContext getContext() {
        SecurityContext ctx = contextHolder.get();
        if (ctx == null) {
            ctx = createEmptyContext();
            contextHolder.set(ctx);
        }
        return ctx;
    }

    @Override
    public void setContext(SecurityContext context) {
        Assert.notNull(context, "message: \"Only non-null SecurityContext instances are permitted\"");
        contextHolder.set(context);
    }
}
```

PasswordEncoder

3.

아키텍처

Password를 안전하게 관리하자

우리는 Spring Security를 사용하면서 유저의 Password를 관리해야할 필요가 있습니다.

Password를 관리할 때는 일단 두가지가 만족되어야합니다.

1. 회원가입할 때 Password를 입력받으면 그 값을 암호화해서 저장해야합니다.
2. 로그인할 때 입력받은 Password와 회원가입할 때의 Password를 비교할 수 있어야합니다.

이 두가지를 만족하기 위해서는 보통 해시 함수라는 알고리즘 방식을 이용합니다.

해시 함수는 암호화는 비교적 쉽지만 복호화가 거의 불가능한 방식의 알고리즘입니다.

이것을 사용하면 아래와 같은 방식으로 password를 관리할수 있습니다.

1. 회원가입할 때 password를 해시함수로 암호화해서 저장합니다.
2. 로그인할 때 password가 들어오면 같은 해시함수로 암호화합니다.
3. 저장된 값을 불러와서 2번의 암호화된 값과 비교합니다.
4. 동일하면 같은 암호로 인지합니다.

PasswordEncoder

PasswordEncoder는 어떤 내용을 담고 있는지 보겠습니다.

```
public interface PasswordEncoder {
    /**
     * @param rawPassword 평문 패스워드
     * @return 암호화된 패스워드
     */
    String encode(CharSequence rawPassword);
    /**
     *
     * @param rawPassword 평문 패스워드
     * @param encodedPassword 암호화된 패스워드
     * @return rawPassword를 암호화한 값과 encodedPassword의 일치여부
     */
    boolean matches(CharSequence rawPassword, String encodedPassword);
}
```

Interface인 PasswordEncoder는 실제로 어떻게 구현되어있을까요?

실제로 어떤 녀석인지 보기 위해서 예제코드에 break point를 걸고 확인해보겠습니다.

```
@Service
@RequiredArgsConstructor
public class UserService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
```

```
passwordEncoder = {DelegatingPasswordEncoder@9508}
> idForEncode = "bcrypt"
> passwordEncoderForEncode = {BCryptPasswordEncoder@9519}
> idToPasswordEncoder = {HashMap@9520} size = 11
> defaultPasswordEncoderForMatches = {DelegatingPasswordEncoder@9521}
```


PasswordEncoder

다양한 PasswordEncoder 전략이 있는데 그 종류는 아래와 같습니다.

종류	예시
NoOpPasswordEncoder	{noop}password
BcryptPasswordEncoder	{bcrypt}\$2a\$10\$dXJ3SW6G7P50IGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
StandardPasswordEncoder	{sha256}97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605abcbcb0
Pbkdf2PasswordEncoder	{pbkdf2}5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca763d8dc
SCryptPasswordEncoder	{scrypt}\$e0801\$8bWJaSu2IKSn9Z9kM+TPXfOc/ 9bdYSrN1oD9qfVThWEwdRTnO7re7Ei+fUZRJ68k9lTyuTeUp4of4g24hHnazw==\$OAOec05+bXxvuu/ 1qZ6NUR+xQYvYv7BeL1QxwRpY5Pc=

스프링 시큐리티는 DelegatingPasswordEncoder라는 표의 모든 PasswordEncoder를 선택할수 있는 대표 PasswordEncoder를 따로 만들어서 사용하고 있습니다.

예시를 보면 뒤쪽의 난해한 문자들은 암호화된 값 같이 보입니다.

그런데 {noop} {bcrypt} {sha256} {pbkdf2} {scrypt} 같은 앞의 단어들은 무엇일까요?

DelegatingPasswordEncoder는 어떤 PasswordEncoder를 선택했는지 알려주기 위해서 앞에 암호화 방식을 표현하고 있습니다. 그렇기 때문에 어떤 암호는 bcrypt로 암호화되고 다른 암호는 sha256 되었다고 하더라도 DelegatingPasswordEncoder는 둘다 지원할 수 있습니다.

참고로 DelegatingPasswordEncoder는 인코딩 전략으로 Bcrypt를 기본 Encoder로 사용하고 있습니다.

PasswordEncoder

PasswordEncoder 종류

NoOpPasswordEncoder

암호화하지 않고 평문으로 사용합니다.

password가 그대로 노출되기 때문에 현재는 deprecated 되었고 사용하지 않기를 권장합니다.

BcryptPasswordEncoder

Bcrypt 해시 함수를 사용한 PasswordEncoder입니다.

Bcrypt는 애초부터 패스워드 저장을 목적으로 설계되었습니다.

Password를 무작위로 여러번 시도하여 맞추는 해킹을 방지하기 위해 암호를 확인할 때 의도적으로 느리게 설정되어있습니다.

BcryptPasswordEncoder는 강도를 설정할 수 있는데 강도가 높을수록 오랜 시간이 걸립니다.

Pbkdf2PasswordEncoder

Pbkdf2는 NIST(National Institute of Standards and Technology, 미국표준기술연구소)에 의해서 승인된 알고리즘이고, 미국 정부 시스템에서도 사용합니다.

ScryptPasswordEncoder

Scrypt는 Pbkdf2와 유사합니다.

해커가 무작위로 password를 맞추려고 시도할 때 메모리 사용량을 늘리거나 반대로 메모리 사용량을 줄여서

느린 공격을 실행할 수밖에 없도록 의도적인 방식을 사용합니다.

따라서 공격이 매우 어렵고 Pbkdf2보다 안전하다고 평가받습니다.

보안에 아주 민감한 경우에 사용할 수 있습니다.

Security Filter

3.

아키텍처

Spring Security의 동작은 사실상 Filter로 동작한다고 해도 무방합니다.

다양한 필터들이 존재하는데 이 Filter들은 각자 다른 기능을 하고 있습니다.

이런 Filter들은 제외할 수도 있고 추가할 수도 있습니다. 필터에 동작하는 순서를 정해줘서 원하는대로 유기적으로 동작할 수 있습니다.

필터의 종류는 많지만 많이 쓰이는 필터는 아래와 같습니다.

SecurityContextPersistenceFilter

BasicAuthenticationFilter

UsernamePasswordAuthenticationFilter

CsrfFilter

RememberMeAuthenticationFilter

AnonymousAuthenticationFilter

FilterSecurityInterceptor

ExceptionTranslationFilter

Filter

3.

아키텍처

SecurityContextPersistenceFilter를 보면 아래처럼 GenericFilterBean를 상속하고 있고 GenericFilterBean는 Filter를 상속하고 있습니다.

즉, SecurityContextPersistenceFilter는 Filter를 구현합니다.

```
public class SecurityContextPersistenceFilter extends GenericFilterBean {
```

```
public abstract class GenericFilterBean implements Filter
```

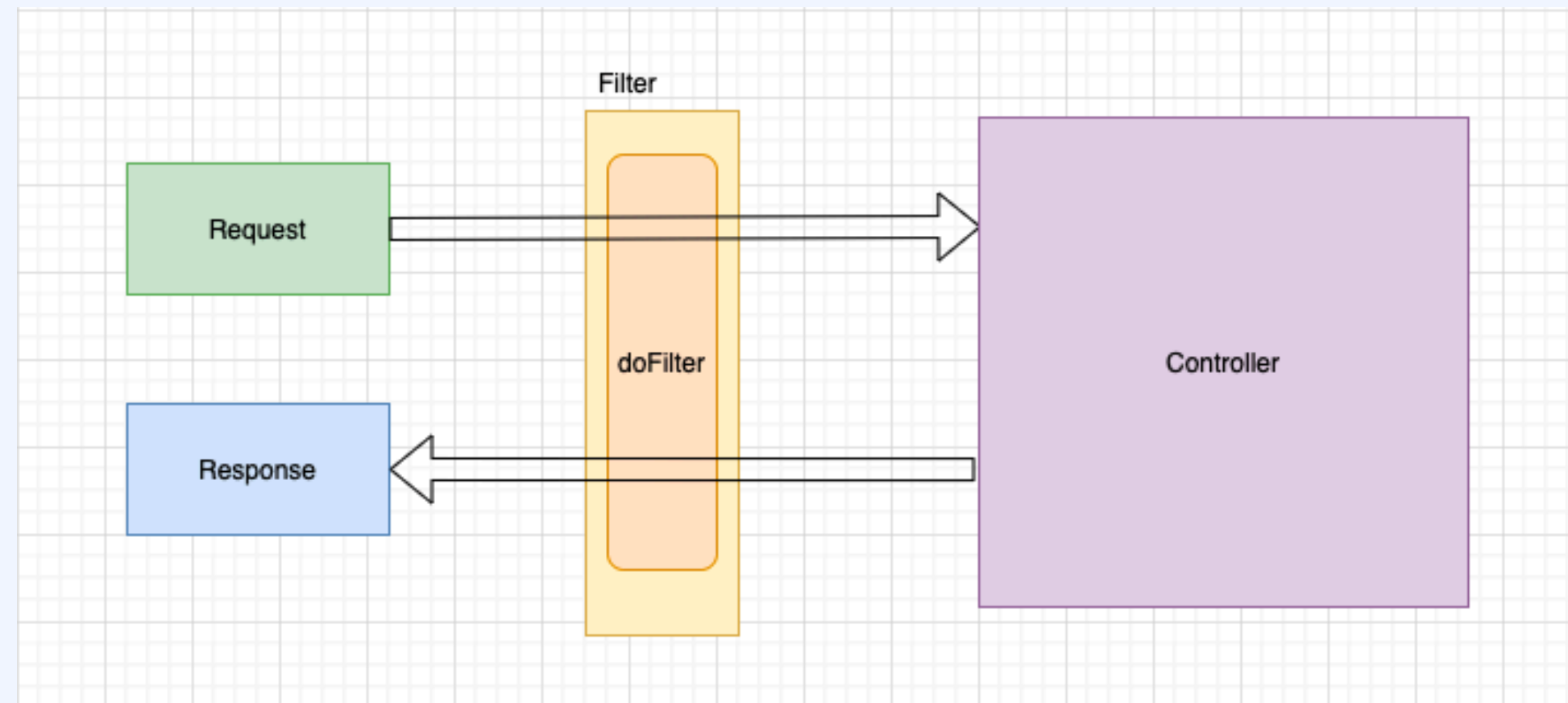
SpringSecurity에는 다양한 Filter들이 존재하는데 그중에 하나가 SecurityContextPersistenceFilter 일 뿐입니다. Filter들의 종류와 기능들을 알기에 앞서 Filter가 무엇인지 파악해보겠습니다.

필터는 요청이나 응답 또는 둘 다에 대해 필터링 작업을 수행하는 개체입니다. 필터는 doFilter 메소드에서 필터링을 수행합니다. 즉 Filter는 doFilter를 구현해야합니다.

```
public interface Filter {  
    public default void init(FilterConfig filterConfig) throws ServletException {}  
    public void doFilter(ServletRequest request, ServletResponse response,  
        FilterChain chain) throws IOException, ServletException;  
    public default void destroy() {}  
}
```

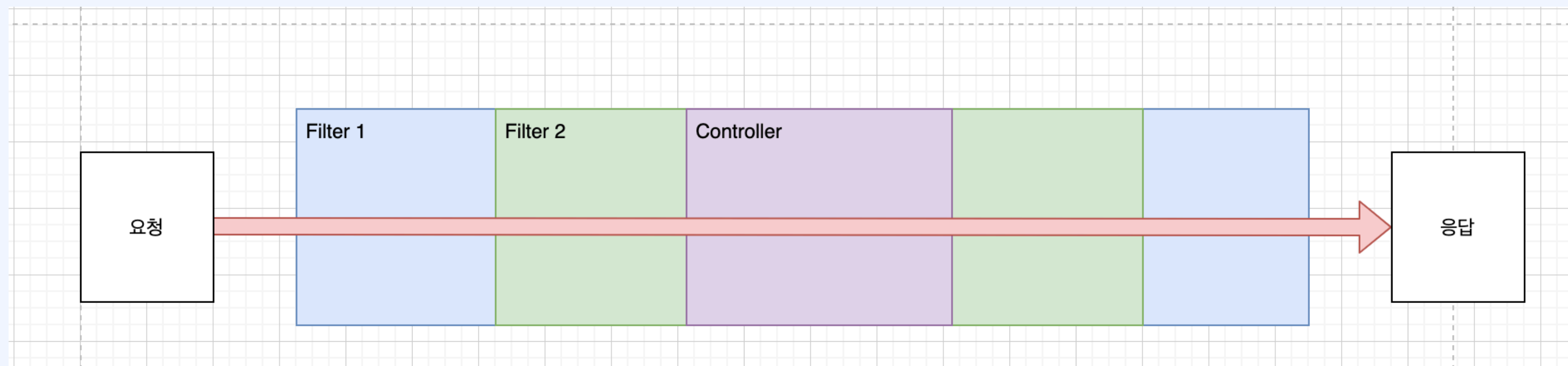
Filter

간단히 생각하면 요청 전, 응답 후 어떤 작업을 하도록 하는게 Filter라고 생각하면 됩니다.



Filter가 여러개인 상황이면 아래처럼 동작합니다.

마지막 순서 필터가 안쪽부터 첫번째 필터의 가장 밖까지 감싸고 있는 형태라고 볼수 있습니다.



Security Filter

3.

아키텍처

다양한 필터들과 그 필터들의 동작 순서는 FilterChainProxy Class에서 doFilterInternal에 break point를 걸어서 디버깅해보면 알 수 있습니다.

아래 그림에는 12가지의 필터가 적용된 것을 볼수 있는데 우리가 구현했던 SpringSecurityConfig를 통해서 어떤 필터가 적용되는지 선택됩니다.

```
private void doFilterInternal(ServletRequest request, ServletResponse response,
    throws IOException, ServletException {
    FirewallRequest firewallRequest = this.firewall.getFirewalledRequest(request);
    HttpServletResponse firewallResponse = this.firewall.getFirewalledResponse(response);
    List<Filter> filters = getFilters(firewallRequest);
    if (filters == null || filters.size() == 0) {
        if (logger.isTraceEnabled()) {
            logger.trace(LogMessage.of() -> "No security for " + requestLine());
        }
        firewallRequest.reset();
    }
}
```

Variables

- firewallRequest = {StrictHttpFirewall\$StrictFirewalledRequest@10882} "FirewalledRequest"
- firewallResponse = {FirewalledResponse@10883}
- filters = {ArrayList@12664} size = 12
 - 0 = {WebAsyncManagerIntegrationFilter@12669}
 - 1 = {SecurityContextPersistenceFilter@12670}
 - 2 = {HeaderWriterFilter@12671}
 - 3 = {CsrfFilter@12672}
 - 4 = {LogoutFilter@12673}
 - 5 = {UsernamePasswordAuthenticationFilter@12674}
 - 6 = {RequestCacheAwareFilter@12675}
 - 7 = {SecurityContextHolderAwareRequestFilter@12676}
 - 8 = {AnonymousAuthenticationFilter@12677}
 - 9 = {SessionManagementFilter@12678}
 - 10 = {ExceptionTranslationFilter@12679}
 - 11 = {FilterSecurityInterceptor@12680}

Security Filter

Filter의 순서는 어디에 정의되었는지는 FilterOrderRegistration 를 보면 알 수 있습니다.

Filter들은 100번 부터 시작해서 100씩 증가됩니다.

즉, 100이 가장 먼저 200, 300, 400 이런 순서대로 필터가 적용됩니다.

굳이 이렇게 만든 이유는 100이라는 공백 사이사이에 커스텀 필터를 넣을 수 있도록 한 것입니다.

```
private static final int INITIAL_ORDER = 100;
```

```
private static final int ORDER_STEP = 100;
```

```
private final Map<String, Integer> filterToOrder = new HashMap<>();
```

```
FilterOrderRegistration() {
    Step order = new Step(INITIAL_ORDER, ORDER_STEP);
    put(ChannelProcessingFilter.class, order.next());
    order.next(); // gh-8105
    put(WebAsyncManagerIntegrationFilter.class, order.next());
    put(SecurityContextPersistenceFilter.class, order.next());
}
```


SecurityContextPersistenceFilter

SecurityContextPersistenceFilter는 보통 두번째로 실행되는 필터입니다.

(첫번째로 실행되는 필터는 Async 요청에 대해서도 SecurityContext를 처리할수 있도록 해주는 WebAsyncManagerIntegrationFilter 입니다.)

SecurityContextPersistenceFilter는 SecurityContext를 찾아와서 SecurityContextHolder에 넣어주는 역할을 하는 Filter입니다. 만약에 SecurityContext를 찾았는데 없다면 그냥 새로 하나 만들어줍니다.

```
public class SecurityContextPersistenceFilter extends GenericFilterBean {
    private void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        SecurityContext contextBeforeChainExecution = this.repo.loadContext(holder);
        try {
            SecurityContextHolder.setContext(contextBeforeChainExecution);
            chain.doFilter(holder.getRequest(), holder.getResponse());
        }
        finally {
            SecurityContext contextAfterChainExecution = SecurityContextHolder.getContext();
            // Crucial removal of SecurityContextHolder contents before anything else.
            SecurityContextHolder.clearContext();
            this.repo.saveContext(contextAfterChainExecution, holder.getRequest(), holder.getResponse());
        }
    }
}
```

SecurityContextPersistenceFilter

3.

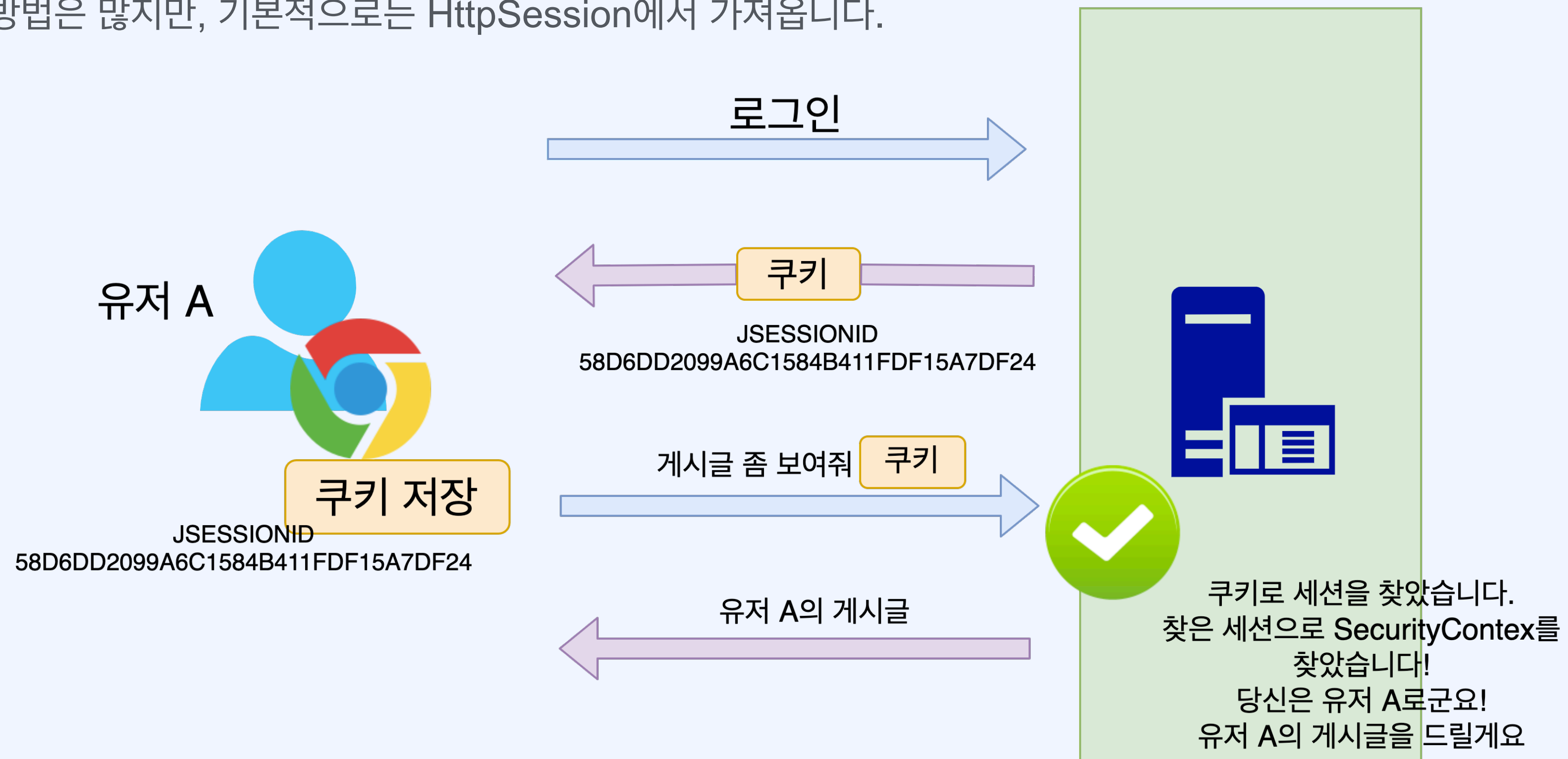
아키텍처

HttpSession

SecurityContextPersistenceFilter는 SecurityContext가 있으면 그걸 사용가져오고 없으면 새로 만든다고 했습니다.

그럼 가져온다는건 대체 어디서 가져온다는 말일까요?

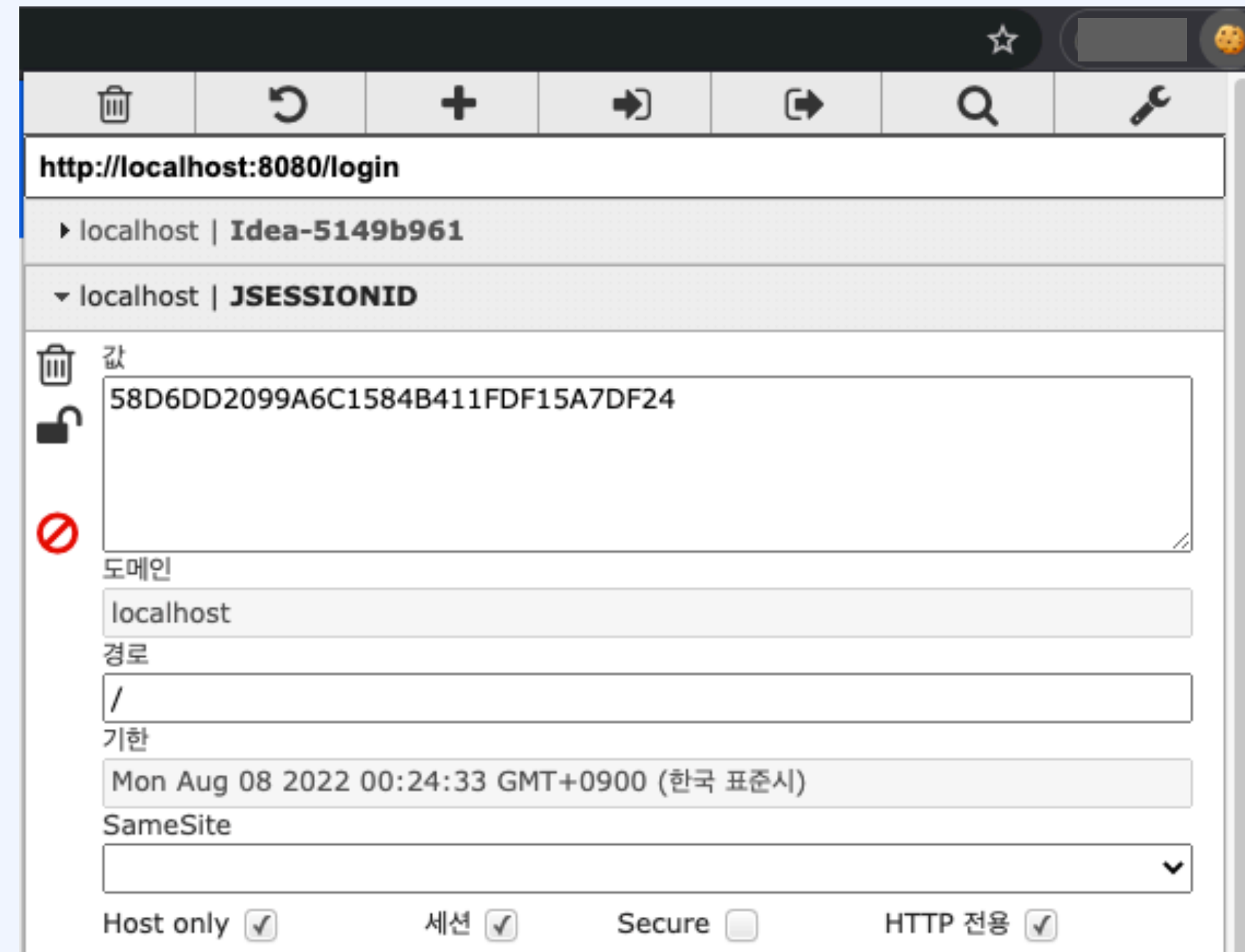
가져올수 있는 방법은 많지만, 기본적으로는 HttpSession에서 가져옵니다.



SecurityContextPersistenceFilter

JSESSIONID

세션 유지에 필요한 Session ID를 쿠키로 가지고 있어야 합니다.
그 값은 JSESSIONID라는 key에 넣어서 가지고 있습니다.



크롬의 확장프로그램 EditThisCookie를 통해 확인한 쿠키

<https://chrome.google.com/webstore/detail/editthiscookie/fngmhnnpilhplaeedifhccceomclgfbg?hl=ko>

BasicAuthenticationFilter

바로 실습을 통해서 어떤 필터인지 알아보겠습니다.

먼저, 우리 프로젝트를 실행 시킨뒤

아래처럼 username과 password를 추가해서 curl로 개인노트 페이지를 요청해보겠습니다.

```
curl -u user:user -L http://localhost:8080/note
```

결과는 로그인 페이지로 redirect 시켜줍니다. 로그인이 되지 않아봅니다.

이번에는 SpringSecurityConfig에 `http.httpBasic();` 추가해서 **BasicAuthenticationFilter**를 활성화합니다.

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    // basic authentication  
    http.httpBasic();  
}
```

다시 시도해봅니다.

```
curl -u user:user -L http://localhost:8080/note
```

이번에는 정상적으로 개인노트 페이지가 불러와집니다.

BasicAuthenticationFilter

3.

아키텍처

직접 필터를 적용해보면 따로 로그인이라는 과정을 하지 않았는데도 일회성으로 페이지를 불러올 수 있었습니다.

이처럼 우리가 로그인이라고 부르는 과정이 없어도 username : user123 / password : pass123 라는 로그인 데이터를 Base64로 인코딩해서 모든 요청에 포함해서 보내면 BasicAuthenticationFilter는 이것을 인증합니다.

그렇기 때문에 세션이 필요 없고 요청이 올 때마다 인증이 이루어 집니다. (즉 stateless합니다.)

이런 방식은 요청할 때마다 아이디와 비밀번호가 반복해서 노출되기 때문에 보안에 취약합니다.

그렇기 때문에 BasicAuthenticationFilter를 사용할 때는 반드시 https를 사용하도록 권장됩니다.

BasicAuthenticationFilter

BasicAuthenticationFilter를 사용하지 않을 것이라면 명시적으로 disable시켜주는 것이 좋습니다.

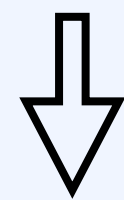
```
@Override
protected void configure(HttpSecurity http) throws Exception {
    // basic authentication
    http.httpBasic(); // basic authentication filter 활성화
    http.httpBasic().disable(); // basic authentication filter 비활성화
}
```


UsernamePasswordAuthenticationFilter

UsernamePasswordAuthenticationFilter는 Form 데이터로 username, password 기반의 인증을 담당하고 있는 필터입니다.

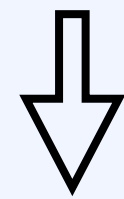
UsernamePasswordAuthenticationFilter

username, password 인증 필터



ProviderManager
(AuthenticationManager)

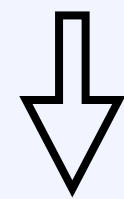
인증 정보 제공 관리자



AbstractUserDetailsAuthenticationProvider

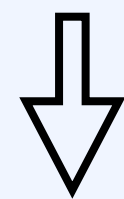
인증 정보 제공

계정의 상태나 패스워드 일치 여부등을 파악



DaoAuthenticationProvider

유저 정보 제공



UserDetailsService

유저 정보 제공하는 Service

UsernamePasswordAuthenticationFilter

ProviderManager (AuthenticationManager)

인자로 받은 authentication이 유효한지 확인하고 authentication을 반환합니다.

인증하면서 계정에 문제가 있는 것이 발견되면 AuthenticationException를 throw할 수 있습니다.

AuthenticationManager는 authenticate 하나만 구현하면 됩니다.

```
public interface AuthenticationManager {
    Authentication authenticate(Authentication authentication) throws AuthenticationException;
}
```

이런 AuthenticationManager를 구현한 Class가 ProviderManager입니다.

ProviderManager는 Password가 일치하는지, 계정이 활성화 되어있는지를 확인한 뒤 authentication을 반환합니다.

```
public class ProviderManager implements AuthenticationManager, MessageSourceAware, InitializingBean {
    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        // 생략...
        result = provider.authenticate(authentication);
        // 생략...
    }
}
```

UsernamePasswordAuthenticationFilter

DaoAuthenticationProvider (AbstractUserDetailsAuthenticationProvider)

유저를 인증하기 위해서는 어쨌든 먼저 유저정보를 가져와야 합니다. DaoAuthenticationProvider는 유저정보를 가져오는 Provider입니다. 사실 우리가 구현한 UserDetailsService를 불러오는게 거의 전부이긴 합니다.

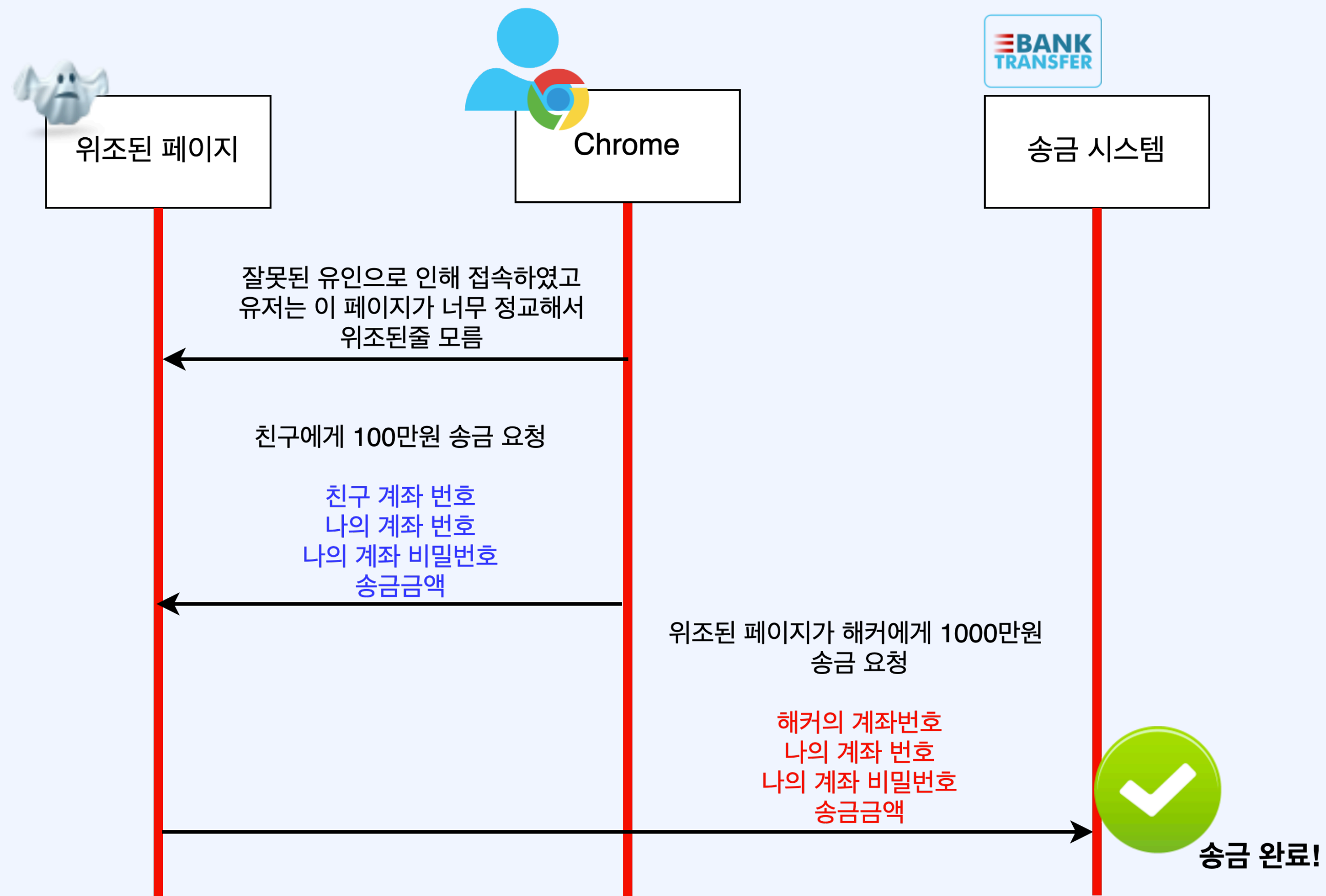
```
public abstract class AbstractUserDetailsAuthenticationProvider {
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String username = determineUsername(authentication);
        user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authentication);
    }
    public class DaoAuthenticationProvider extends AbstractUserDetailsAuthenticationProvider {
        protected final UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken authentication) {
            UserDetails loadedUser = this.getUserDetailsService().loadUserByUsername(username);
            return loadedUser;
        }
        public UserDetailsService userDetailsService() {
            return username -> {
                User user = userService.findByUsername(username);
                if (user == null) {
                    throw new UsernameNotFoundException(username);
                }
                return user;
            }
        }
    }
}
```

CsrfFilter

3. 아키텍처

CsrfFilter는 CsrftAttack을 방어하는 Filter입니다.

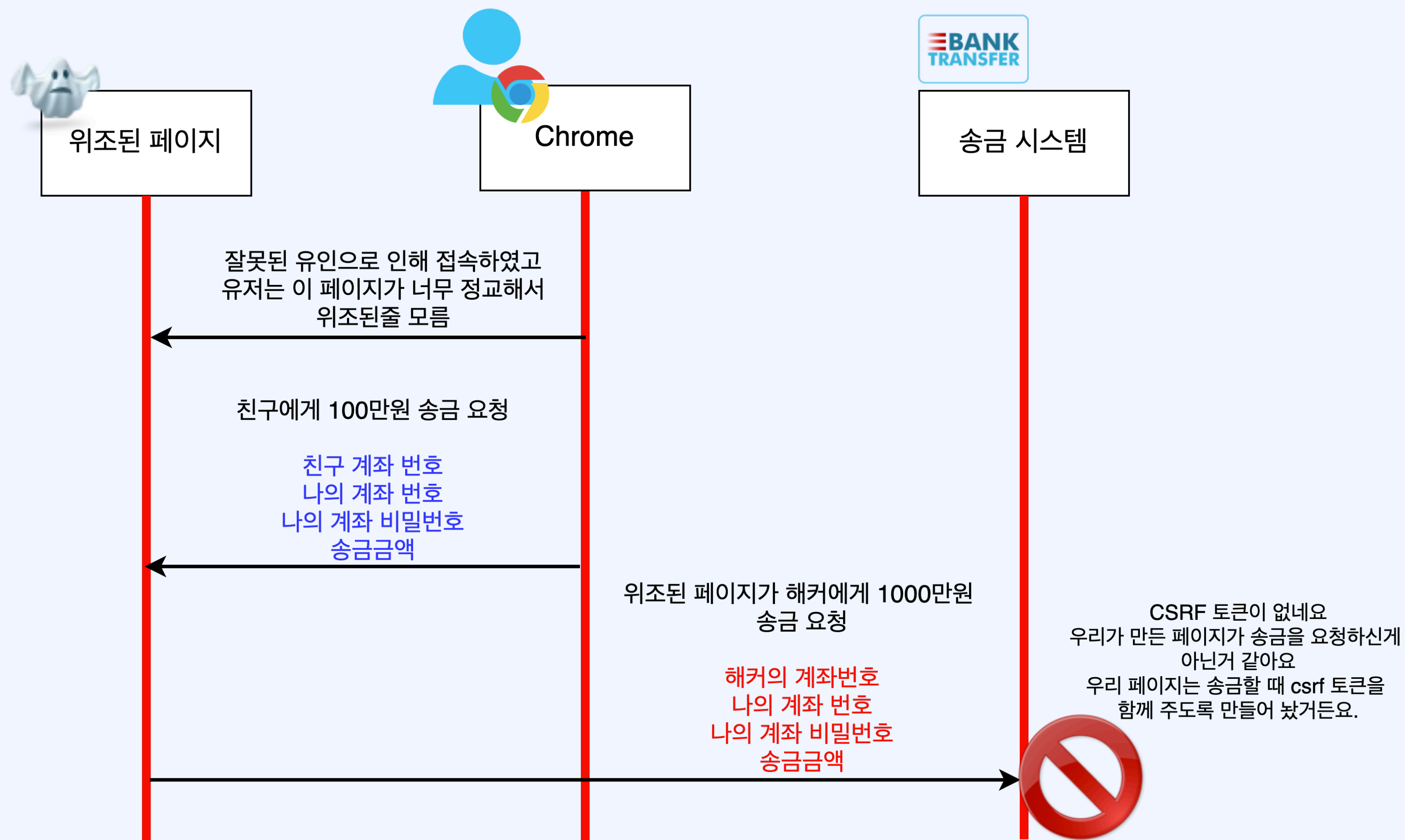
Csrf Attack이란...?



CsrfFilter

3. 아키텍처

Csrf Attack을 막기 위해서...



CsrfFilter

CsrfFilter는 Csrf Token을 사용하여 위조된 페이지의 악의적인 공격을 방어합니다.

정상적인 페이지는 Csrf Token이 있을 것이고

위조된 페이지는 Csrf Token이 없거나 잘못된 Csrf Token을 가지고 있습니다.

따라서 정상적인 페이지에는 Csrf Token 값을 알려줘야 하는데 Tymeleaf에서는 페이지를 만들때 자동으로 Csrf Token을 넣어줍니다.

따로 추가하지 않았는데 아래와 같은 코드가 form tag안에 자동으로 생성됩니다.

대신 굳이 사용자에게 보여줄 필요가 없는 값이기 때문에 hidden으로 처리됩니다.

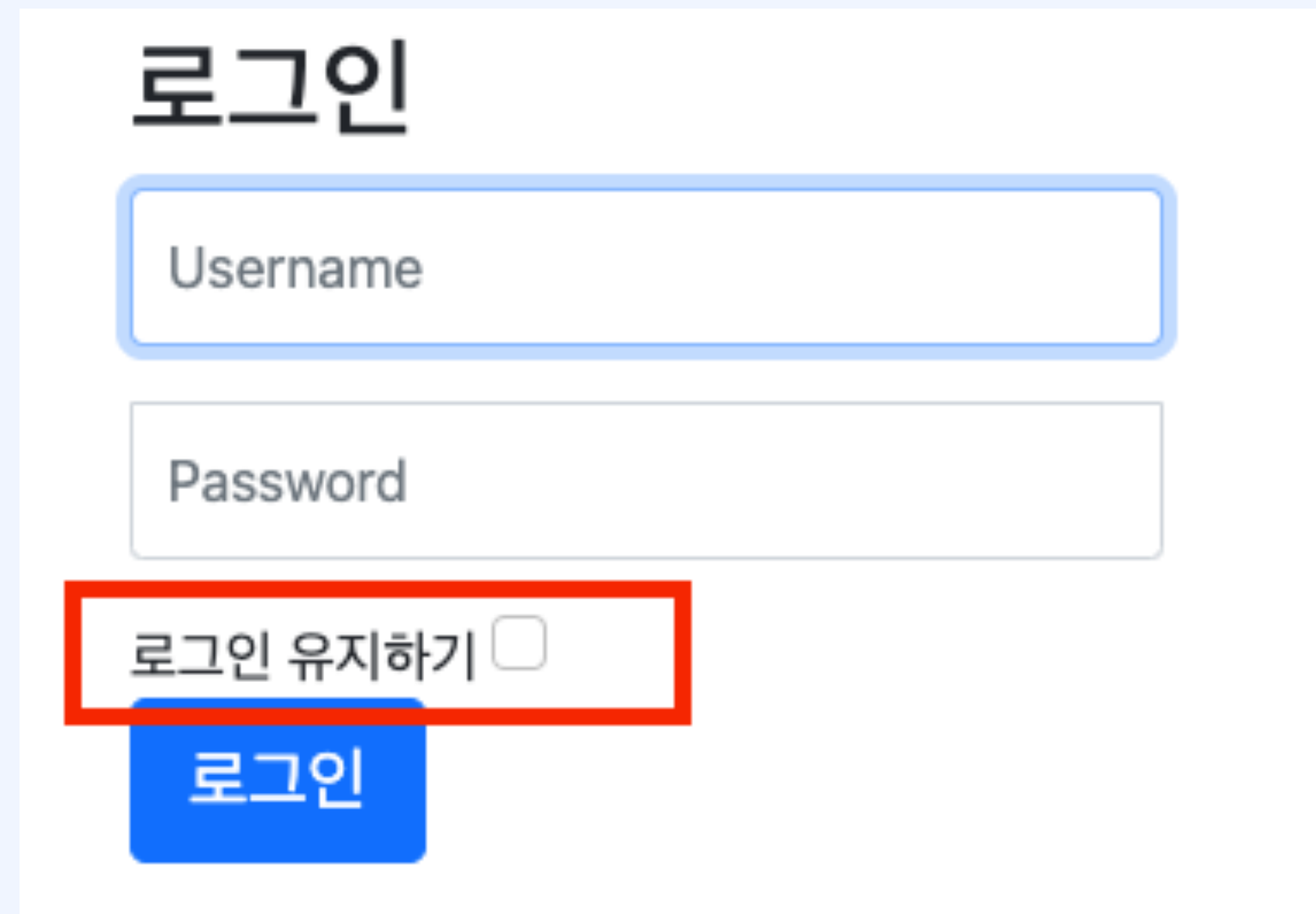
```
<input type="hidden" name="_csrf" value="594af42a-63e9-4ef9-aeb2-3687f12cdf43"/>
```

Csrf Filter는 자동으로 활성화되어있는 Filter지만 명시적으로 On 하기 위해서는 `http.csrf();` 코드를 추가합니다.

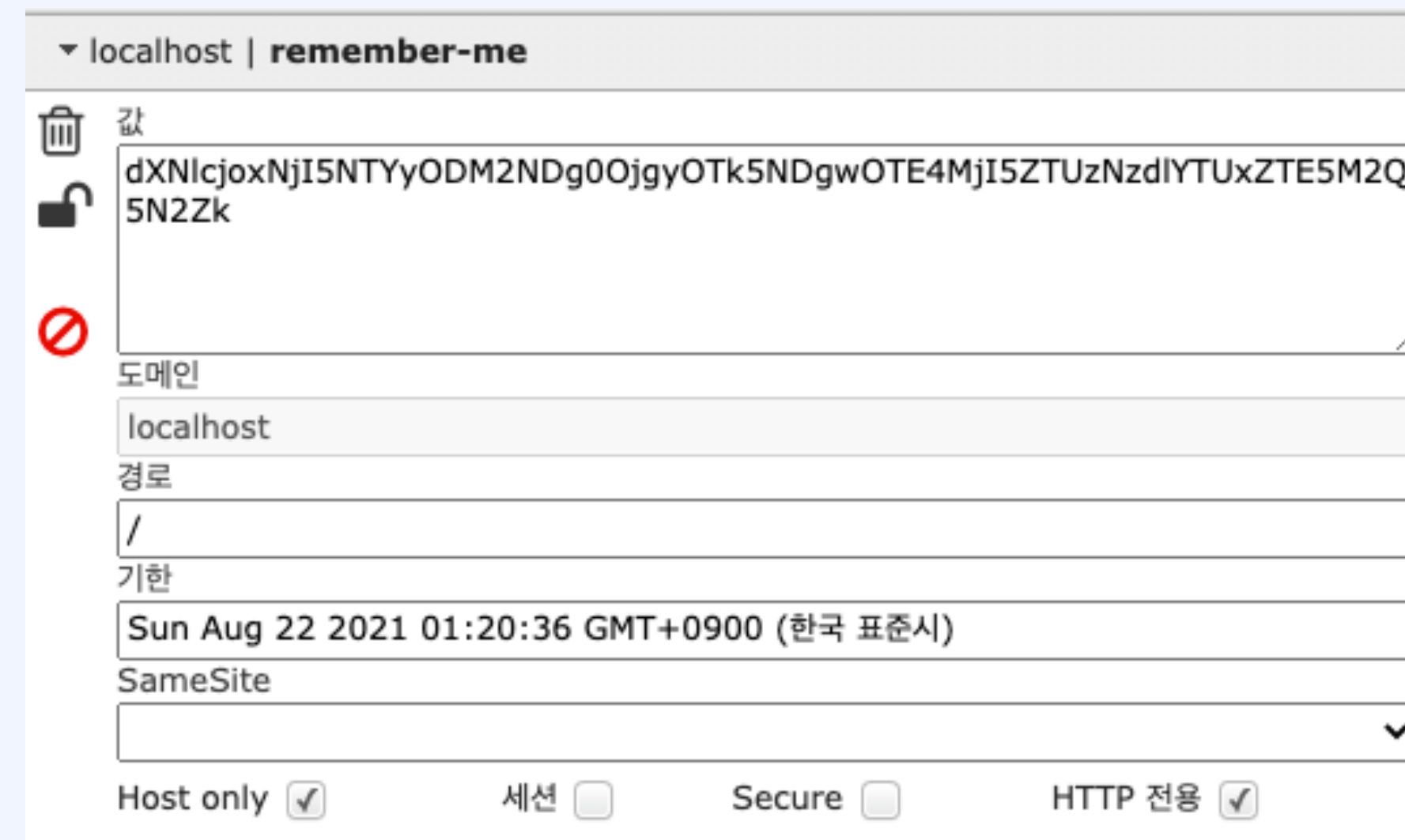
Off하기 위해서는 `http.csrf().disable();` 해줍니다.

RememberMeAuthenticationFilter

RememberMeAuthenticationFilter는 일반적인 세션보다 훨씬 오랫동안 로그인 사실을 기억할 수 있도록 해줍니다. Session의 세션 만료 시간은 기본 설정이 30분이지만 RememberMeAuthenticationFilter의 기본 설정은 2주 입니다.



The image shows a login form titled '로그인' (Login). It contains two input fields: 'Username' and 'Password'. Below these fields is a checkbox labeled '로그인 유지하기' (Remember Me), which is highlighted with a red rectangular border. At the bottom of the form is a blue button labeled '로그인' (Login).



The image shows the details of a cookie named 'remember-me' on the 'localhost' domain. The cookie value is a long alphanumeric string: 'dXNlcjoxNjI5NTYyODM2NDg0OjgyOTk5NDgwOTE4MjI5ZTUzNzdIYTUxZTE5M2Q5N2Zk'. The domain is 'localhost' and the path is '/'. The expiration date is 'Sun Aug 22 2021 01:20:36 GMT+0900 (한국 표준시)'. The 'SameSite' attribute is set to 'Host only' (checked), 'Secure' is unchecked, and 'HTTP 전용' (HTTP only) is checked.

우리 프로젝트는 이미 RememberMeAuthenticationFilter를 사용할 수 있도록 되어있습니다.

RememberMeAuthenticationFilter

3. 아키텍처

RememberMeAuthenticationFilter를 ON 하는 방법

```
// remember-me  
http.rememberMe();
```

```
<div>  
  <span>로그인 유지하기</span>  
  <input type="checkbox" id="remember-me" name="remember-me" class="form-check-input mt-0" autocomplete="off">  
</div>
```

AnonymousAuthenticationFilter

인증이 안된 유저가 요청을 하면 Anonymous(익명) 유저로 만들어 Authentication에 넣어주는 필터입니다. 인증되지 않았다고 하더라도 Null을 넣는게 아니라 기본 Authentication을 만들어 주는 개념으로 보면 됩니다. 다른 Filter에서 Anonymous유저인지 정상적으로 인증된 유저인지 분기 처리를 할 수 있습니다.

Anonymous 유저의 SecurityContext

```

result = {AnonymousAuthenticationToken@11121} "AnonymousAuthenticationToken [Principal=anonymousUser, Credentials=[PROT
  > f principal = "anonymousUser"
  f keyHash = 4143904
  v f authorities = {Collections$UnmodifiableRandomAccessList@11124} size = 1
    v 0 = {SimpleGrantedAuthority@11128} "ROLE_ANONYMOUS"
      > f role = "ROLE_ANONYMOUS"
  > f details = {WebAuthenticationDetails@11125} "WebAuthenticationDetails [RemoteIpAddress=0:0:0:0:0:0:0:1, SessionId=null]"
  f authenticated = true

```

AnonymousAuthenticationFilter 활성화

```

// anonymous
http.anonymous().principal("anonymousUser");

```

FilterSecurityInterceptor

3.

아키텍처

이름만 봐서는 Interceptor로 끝나지만 Filter 중에 하나입니다.

앞서 본 SecurityContextPersistenceFilter, UsernamePasswordAuthenticationFilter, AnonymousAuthenticationFilter 에서 SecurityContext를 찾거나 만들어서 넘겨주고 있다는 걸 확인했습니다.

FilterSecurityInterceptor에서는 이렇게 넘어온 authentication의 내용을 기반으로 최종 인가 판단을 내립니다.

그렇기 때문에 대부분의 경우에는 필터중에 뒤쪽에 위치합니다.

먼저, 인증(Authentication)을 가져오고 만약에 인증에 문제가 있다면 AuthenticationException를 발생합니다.

인증에 문제가 없다면 해당 인증으로 인가를 판단합니다.

이때 인가가 거절된다면 AccessDeniedException를 발생하고 승인된다면 정상적으로 필터가 종료됩니다.

FilterSecurityInterceptor

3.
아키텍처

FilterSecurityInterceptor.doFilter()



AbstractSecurityInterceptor.beforeInvocation()



AbstractSecurityInterceptor.authenticateIfRequired()



AbstractSecurityInterceptor.attemptAuthorization()

인증에 문제가 있다면...? **AuthenticationException**

인가에 문제가 있다면...? **AccessDeniedException**

FilterSecurityInterceptor

3.

아키텍처

```
public class FilterSecurityInterceptor extends AbstractSecurityInterceptor implements Filter {  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  
        throws IOException, ServletException {  
        invoke(new FilterInvocation(request, response, chain));  
    }  
  
    public void invoke(FilterInvocation filterInvocation) throws IOException, ServletException {  
        InterceptorStatusToken token = super.beforeInvocation(filterInvocation);  
  
        protected InterceptorStatusToken beforeInvocation(Object object) {  
            attemptAuthorization(object, attributes, authenticated);  
        }  
  
        private void attemptAuthorization(Object object, Collection<ConfigAttribute> attributes,  
            Authentication authenticated) {  
            try {  
                this.accessDecisionManager.decide(authenticated, object, attributes);  
            }  
        }  
  
        public void decide(Authentication authentication, Object object, Collection<ConfigAttribute> configAttributes)  
            throws AccessDeniedException {  
                int result = voter.vote(authentication, object, configAttributes);  
            }  
}
```


ExceptionTranslationFilter

앞서 본 FilterSecurityInterceptor에서 발생할 수 있는 두가지 Exception을 처리해주는 필터입니다.

1. AuthenticationException : 인증에 실패할 때 발생
2. AccessDeniedException : 인가에 실패할 때 발생

즉, 인증이나 인가에 실패했을 때 어떤 행동을 취해야하는지를 결정해주는 Filter입니다.

ExceptionTranslationFilter의 handleSpringSecurityException는 Exception의 종류에 따른 로직을 분산합니다.

```
private void handleSpringSecurityException(HttpServletRequest request, HttpServletResponse response,
    FilterChain chain, RuntimeException exception) throws IOException, ServletException {
    if (exception instanceof AuthenticationException) {
        handleAuthenticationException(request, response, chain, (AuthenticationException) exception);
    }
    else if (exception instanceof AccessDeniedException) {
        handleAccessDeniedException(request, response, chain, (AccessDeniedException) exception);
    }
}
```

ExceptionTranslationFilter

현재 우리 프로젝트에서는 (기본 설정)

AuthenticationException 발생 또는 Anonymous의 AccessDeniedException 발생 : Login Page로 이동

기명 유저의 AccessDeniedException 발생 : 403 Forbidden Whitelabel Error Page로 이동

Exception에 대한 대처 방식은 변경할 수 있습니다.

AuthenticationException

혹은 Anonymous유저가
AccessDeniedException를 발생한 경우

로그인

Username

Password

로그인 유지하기 ☐

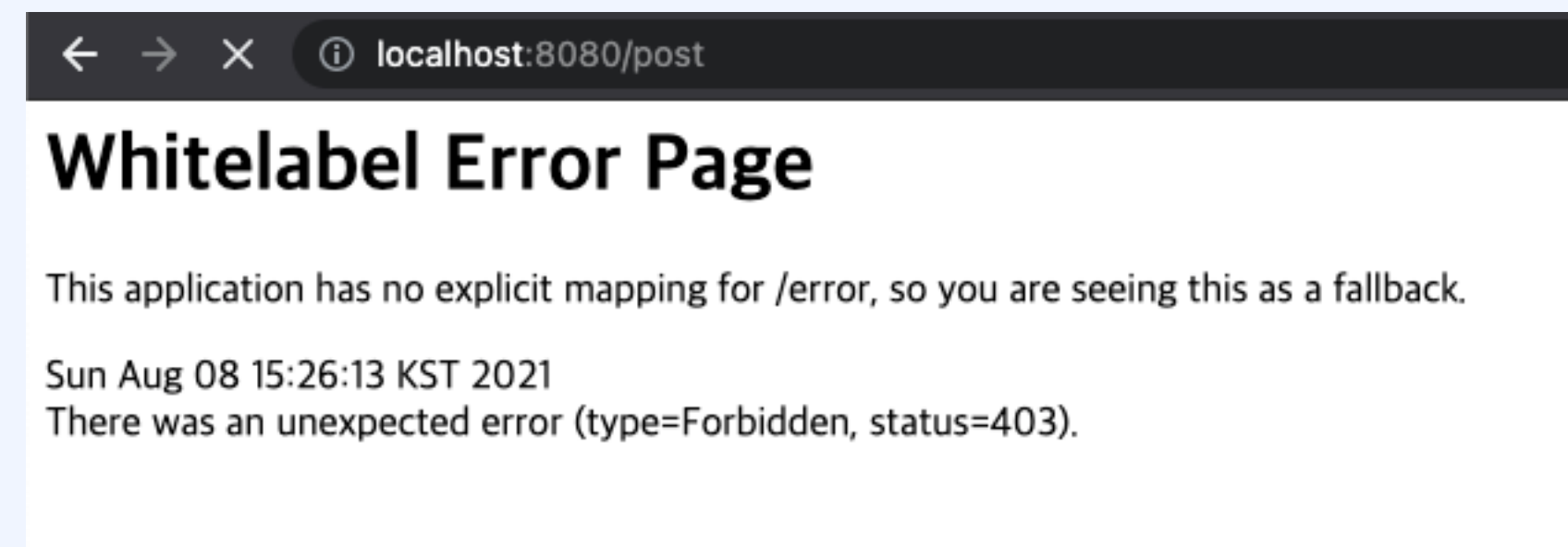
로그인

LoginUrlAuthenticationEntryPoint

ExceptionTranslationFilter

handleSpringSecurityException()

AccessDeniedException



AccessDeniedHandlerImpl