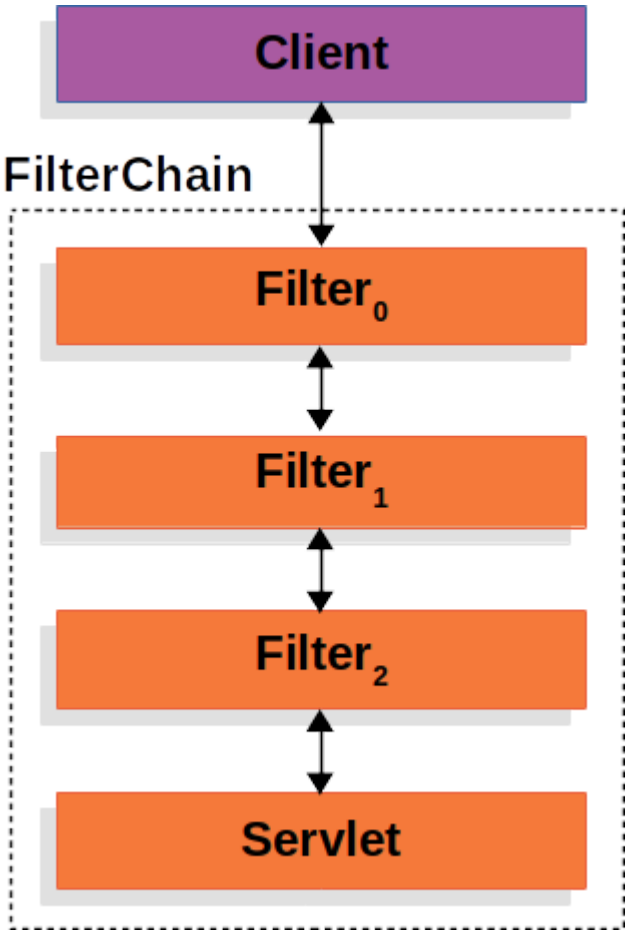
아키텍처

이 섹션에서는 Servlet 기반 애플리케이션에서 Spring Security의 고수준 아키텍처를 설명합니다. 이 고수준 이해를 바탕으로 이후 참고 자료의 인증(Authentication), 권한 부여(Authorization), 공격 방어(Protection Against Exploits) 섹션을 구축해 나갑니다.

필터에 대한 리뷰

Spring Security의 Servlet 지원은 Servlet 필터에 기반을 두고 있으므로, 필터의 역할에 대해 먼저 살펴보는 것이 좋습니다. 다음 이미지는 단일 HTTP 요청에 대한 핸들러의 일반적인 레이어 구성을 보여줍니다.



FilterChain

클라이언트가 애플리케이션에 요청을 보내면 컨테이너가 FilterChain을 생성합니다. 이 FilterChain은 요청 URI 경로를 기반으로 HttpServletRequest를 처리해야 하는 필터 인스턴스와 서블릿을 포함합니다. Spring MVC 애플리케이션에서는 서블릿이 DispatcherServlet 인스턴스입니다. 하나의 HttpServletRequest와 HttpServletResponse를 처리할 수 있는 서블릿은 최대 하나이지만, 여러 개의 필터를 사용하여 다음 작업을 수행할 수 있습니다:

- 하위 필터 인스턴스 또는 서블릿이 호출되지 않도록 방지합니다. 이 경우 필터는 일반적으로 HttpServletResponse에 기록합니다.
- 하위 필터 인스턴스와 서블릿에서 사용하는 HttpServletRequest 또는 HttpServletResponse를 수정합니다.

필터의 강력한 기능은 필터체인(FilterChain)에 의해 전달됩니다.

FilterChain 사용 예제

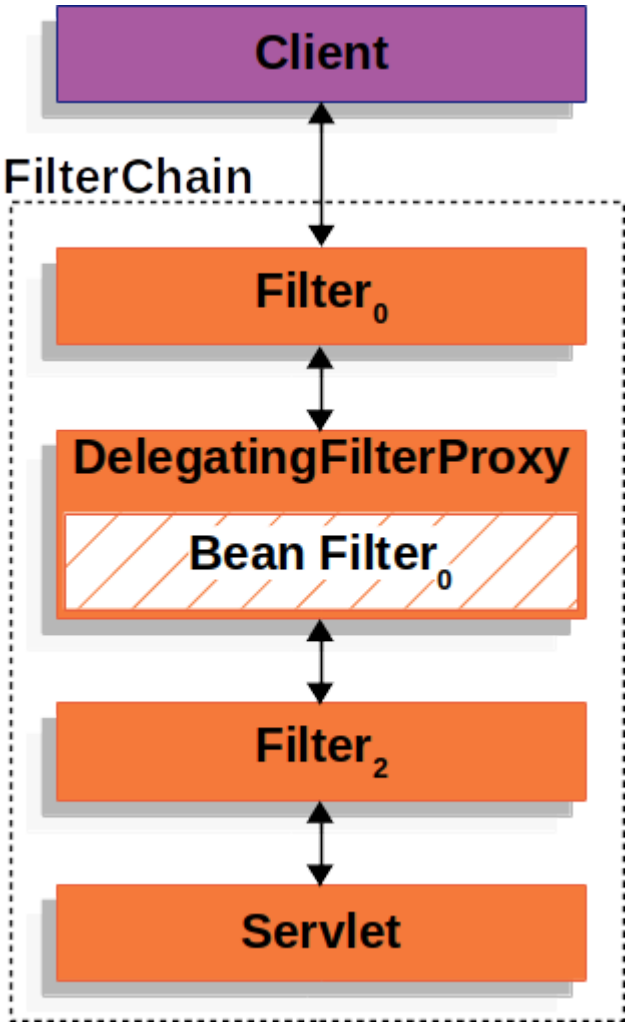
```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) {
    // 애플리케이션의 나머지 부분을 호출하기 전 작업 수행
    chain.doFilter(request, response); // 애플리케이션의 나머지 부분 호출
    // 애플리케이션의 나머지 부분을 호출한 후 작업 수행
}
```

필터는 하위 필터 인스턴스와 서블릿에만 영향을 미치므로 각 필터가 호출되는 순서가 매우 중요합니다.

DelegatingFilterProxy

Spring은 DelegatingFilterProxy라는 필터 구현을 제공하여 Servlet 컨테이너의 라이프사이클과 Spring의 ApplicationContext 간의 다리를 만들어줍니다. Servlet 컨테이너는 자체 표준을 사용하여 필터 인스턴스를 등록할 수 있지만, Spring에서 정의한 빈은 인식하지 못합니다. 표준 Servlet 컨테이너 메커니즘을 통해 DelegatingFilterProxy를 등록하고, 필터를 구현한 Spring Bean에 모든 작업을 위임할 수 있습니다.

DelegatingFilterProxy가 Filter 인스턴스 및 FilterChain에 어떻게 맞춰지는지 보여주는 그림이 아래에 있습니다.

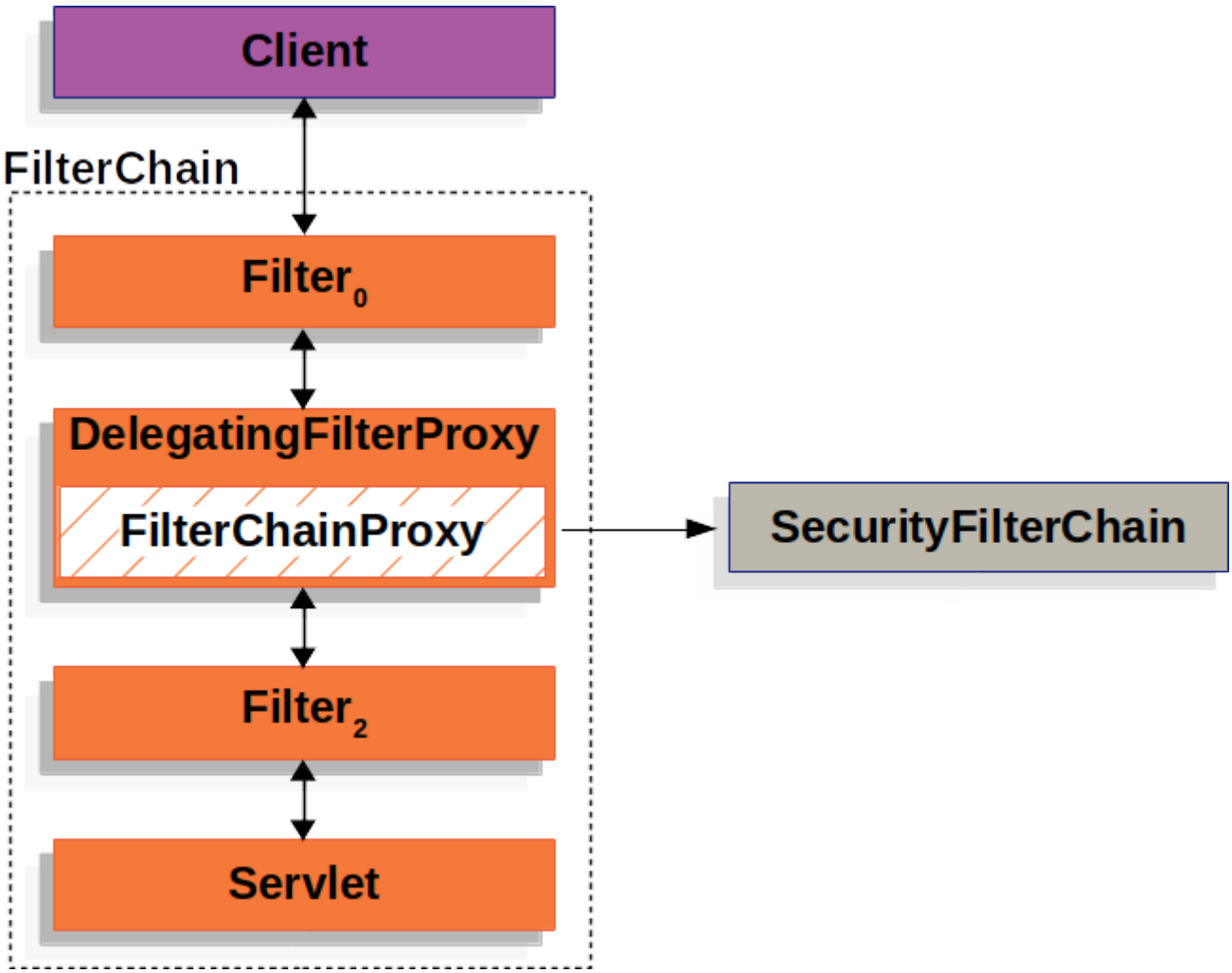


DelegatingFilterProxy의 Pseudo Code

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) {
    Filter delegate = getFilterBean(someBeanName);
    delegate.doFilter(request, response);
}
```

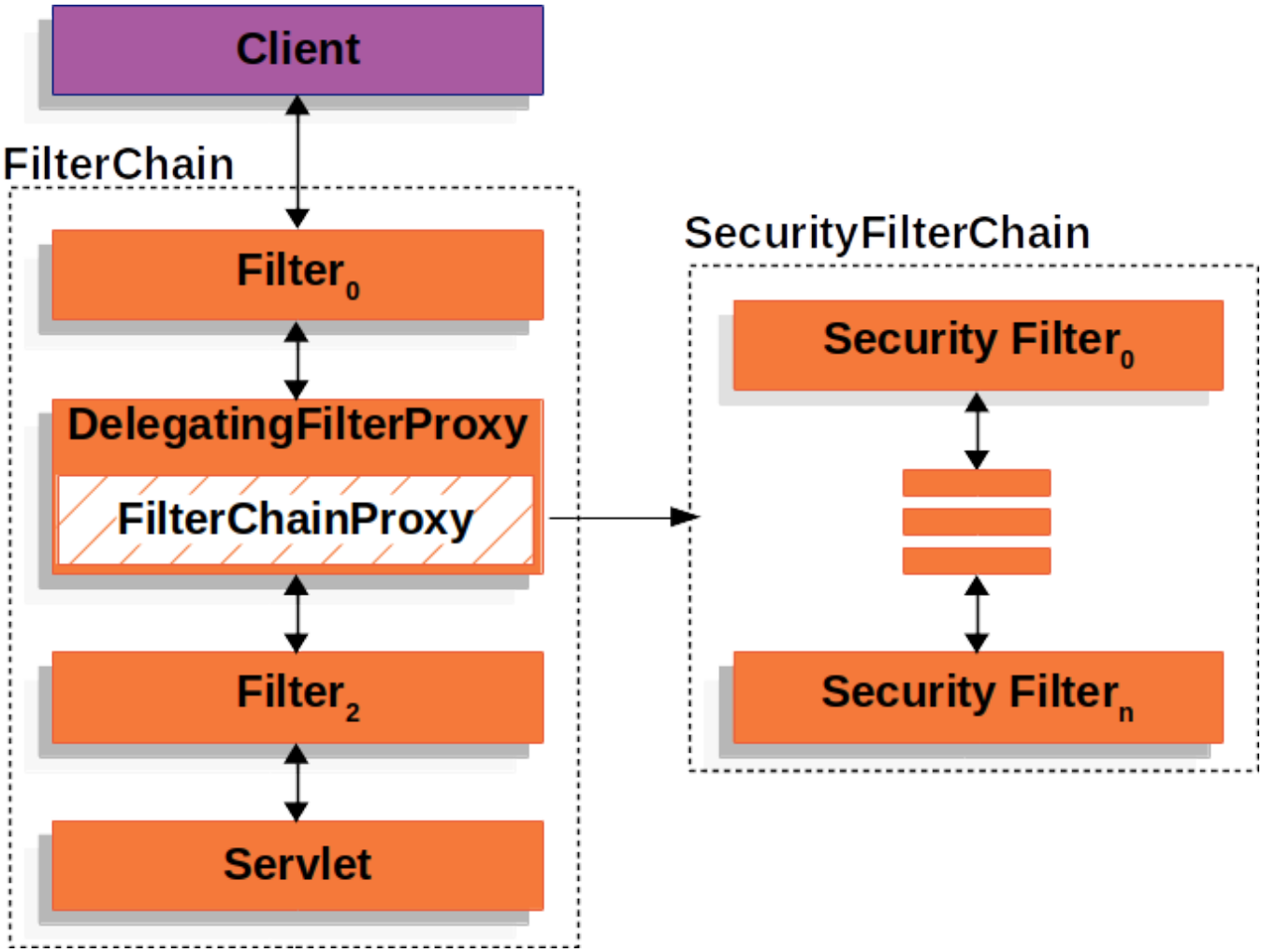
DelegatingFilterProxy는 Spring Bean으로 등록된 필터를 지연해서 가져올 수 있습니다. DelegatingFilterProxy의 이점 중 하나는 필터 빈 인스턴스의 검색을 지연할 수 있다는 점입니다. 이는 컨테이너가 시작되기 전에 필터 인스턴스를 등록해야 하기 때문에 중요합니다. 그러나 Spring은 일반적으로 ContextLoaderListener를 사용하여 Spring Bean을 로드하며, 필터 인스턴스를 등록해야 하는 시점 이후에 로드가 완료됩니다.

FilterChainProxy



Spring Security의 Servlet 지원은 FilterChainProxy 내에 포함되어 있습니다. FilterChainProxy는 Spring Security에서 제공하는 특수한 필터로, SecurityFilterChain을 통해 여러 필터 인스턴스에 위임할 수 있게 해줍니다. FilterChainProxy는 빈이므로 일반적으로 DelegatingFilterProxy로 감싸져 있습니다.

SecurityFilterChain

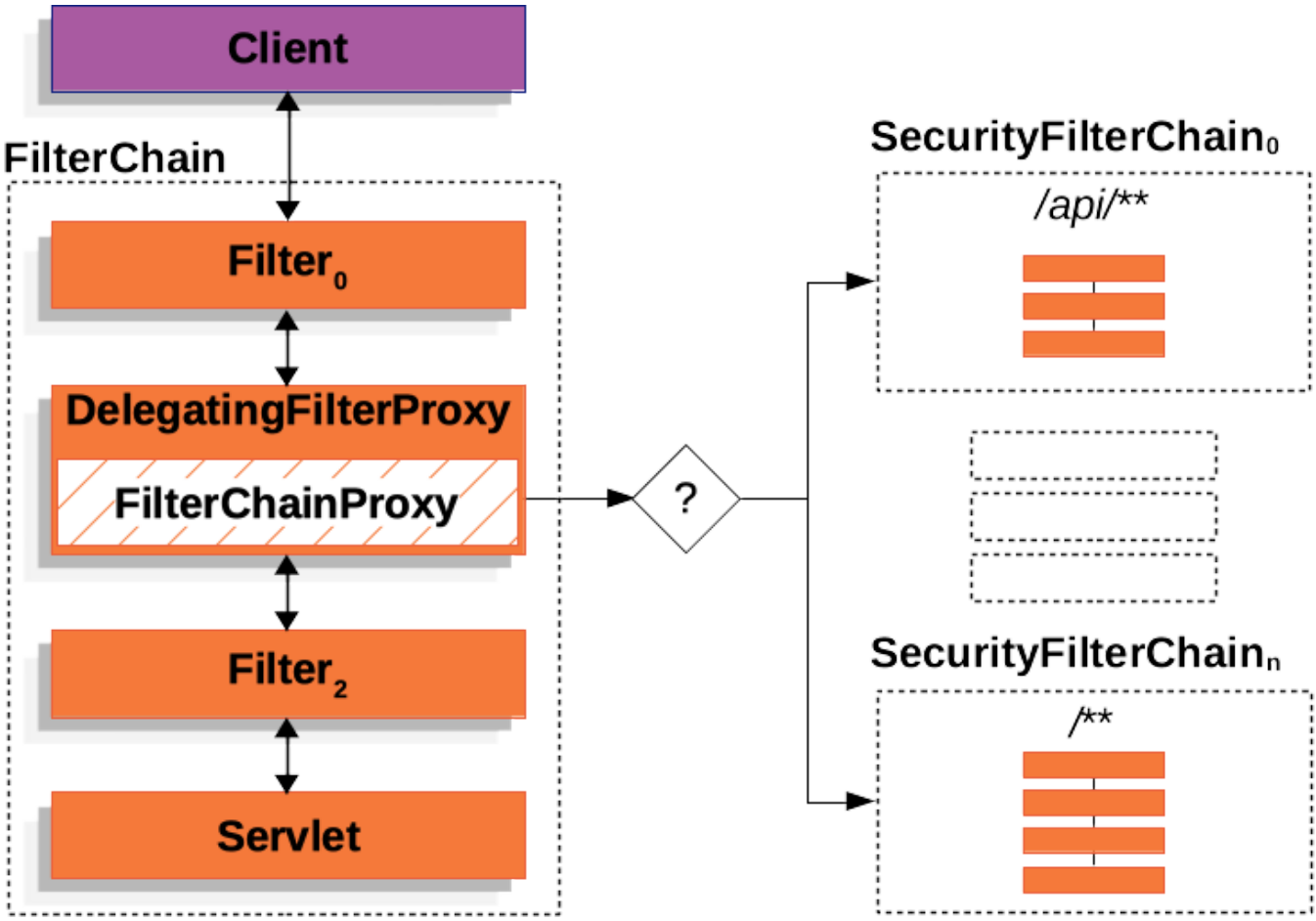


SecurityFilterChain은 FilterChainProxy가 현재 요청에 대해 호출해야 하는 Spring Security 필터 인스턴스를 결정하는 데 사용됩니다.

SecurityFilterChain 내의 보안 필터는 일반적으로 빈이지만, DelegatingFilterProxy 대신 FilterChainProxy에 등록됩니다. FilterChainProxy는 Servlet 컨테이너 또는 DelegatingFilterProxy에 직접 등록하는 것보다 몇 가지 이점을 제공합니다. 첫째, 이는 Spring Security의 Servlet 지원을 위한 시작 지점을 제공합니다. 따라서 Spring Security의 Servlet 지원을 문제 해결하려면 FilterChainProxy에 디버그 포인트를 추가하는 것이 좋은 방법입니다.

둘째, FilterChainProxy는 Spring Security의 사용에서 중심적인 역할을 하기 때문에 필수적이지 않은 작업을 수행할 수 있습니다. 예를 들어, 메모리 누수를 방지하기 위해 SecurityContext를 정리합니다. 또한 Spring Security의 HttpFirewall을 적용하여 특정 유형의 공격으로부터 애플리케이션을 보호합니다.

또한 FilterChainProxy는 SecurityFilterChain이 호출될 시점을 결정하는 데 더 큰 유연성을 제공합니다. Servlet 컨테이너에서는 URL만을 기준으로 필터 인스턴스가 호출되지만, FilterChainProxy는 RequestMatcher 인터페이스를 사용하여 HttpServletRequest의 모든 항목을 기준으로 호출을 결정할 수 있습니다.



위 그림에서 `FilterChainProxy`는 어느 `SecurityFilterChain`을 사용할지 결정합니다. 일치하는 첫 번째 `SecurityFilterChain`만 호출됩니다. 예를 들어, `/api/messages/` URL 요청이 들어오면 `/api/**` 패턴에 해당하는 `SecurityFilterChain0`과 먼저 일치하므로 `SecurityFilterChain0`만 호출됩니다. 비록 `SecurityFilterChainn`과도 일치하지만 호출되지 않습니다. 만약 `/messages/` URL이 요청되면 `/api/**` 패턴과 일치하지 않으므로 `FilterChainProxy`는 다른 `SecurityFilterChain`을 계속 확인합니다. 다른 `SecurityFilterChain` 인스턴스와 일치하는 것이 없다고 가정하면 `SecurityFilterChainn`이 호출됩니다.

참고로 `SecurityFilterChain0`에는 보안 필터 인스턴스가 세 개만 구성되어 있지만, `SecurityFilterChainn`에는 네 개의 보안 필터 인스턴스가 구성되어 있습니다. 각 `SecurityFilterChain`은 고유할 수 있으며 독립적으로 구성할 수 있다는 점이 중요합니다. 사실, Spring Security가 특정 요청을 무시하도록 하려면 보안 필터 인스턴스가 없는 `SecurityFilterChain`을 구성할 수도 있습니다.

보안 필터

보안 필터는 SecurityFilterChain API를 통해 FilterChainProxy에 삽입됩니다. 이러한 필터는 인증, 권한 부여, 공격 방지 등 여러 용도로 사용할 수 있습니다. 예를 들어, 인증을 수행하는 필터는 권한 부여를 수행하는 필터보다 먼저 호출되어야 합니다. Spring Security 필터의 순서를 반드시 알아야 할 필요는 없지만, 필요시 FilterOrderRegistration 코드를 통해 확인할 수 있습니다.

보안 설정 예제

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
```

```
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(Customizer.withDefaults())
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .httpBasic(Customizer.withDefaults())
            .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

위 구성에서는 필터가 다음 순서로 추가됩니다:

1. CsrfFilter: CSRF 공격으로부터 보호합니다.
2. UsernamePasswordAuthenticationFilter: 요청을 인증합니다.
3. BasicAuthenticationFilter: 요청을 기본 인증 방식으로 인증합니다.
4. AuthorizationFilter: 요청에 권한이 있는지 확인합니다.

보안 필터 목록 출력

특정 요청에 대해 호출되는 보안 필터 목록을 출력하여 특정 필터가 보안 필터 목록에 포함되었는지 확인할 수 있습니다. 필터 목록은 애플리케이션 시작 시 INFO 레벨로 콘솔에 출력되므로, 예를 들어 다음과 같은 콘솔 출력을 확인할 수 있습니다:

```
2023-06-14T08:55:22.321-03:00  INFO 76975 --- [           main]
o.s.s.web.DefaultSecurityFilterChain     : Will secure any request with [
org.springframework.security.web.session.DisableEncodeUrlFilter@404db674,
org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationF
ilter@50f097b5,
org.springframework.security.web.context.SecurityContextHolderFilter@6fc6deb7,
org.springframework.security.web.header.HeaderWriterFilter@6f76c2cc,
org.springframework.security.web.csrf.CsrfFilter@c29fe36,
...
```

필터 체인에 사용자 정의 필터 추가

대부분의 경우 기본 보안 필터가 애플리케이션의 보안을 제공하는 데 충분합니다. 그러나 때로는 보안 필터 체인에 사용자 정의 필터를 추가하고 싶을 수 있습니다.

예를 들어, 테넌트 ID 헤더를 가져와 현재 사용자가 해당 테넌트에 접근할 수 있는지 확인하는 필터를 추가하고 싶다고 가정해봅시다.

```
public class TenantFilter implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
```

```
            HttpServletRequest request = (HttpServletRequest) servletRequest;
            HttpServletResponse response = (HttpServletResponse) servletResponse;

            String tenantId = request.getHeader("X-Tenant-Id");
            boolean hasAccess = isUserAllowed(tenantId);
            if (hasAccess) {
                filterChain.doFilter(request, response);
                return;
            }
            throw new AccessDeniedException("Access denied");
        }
    }
```

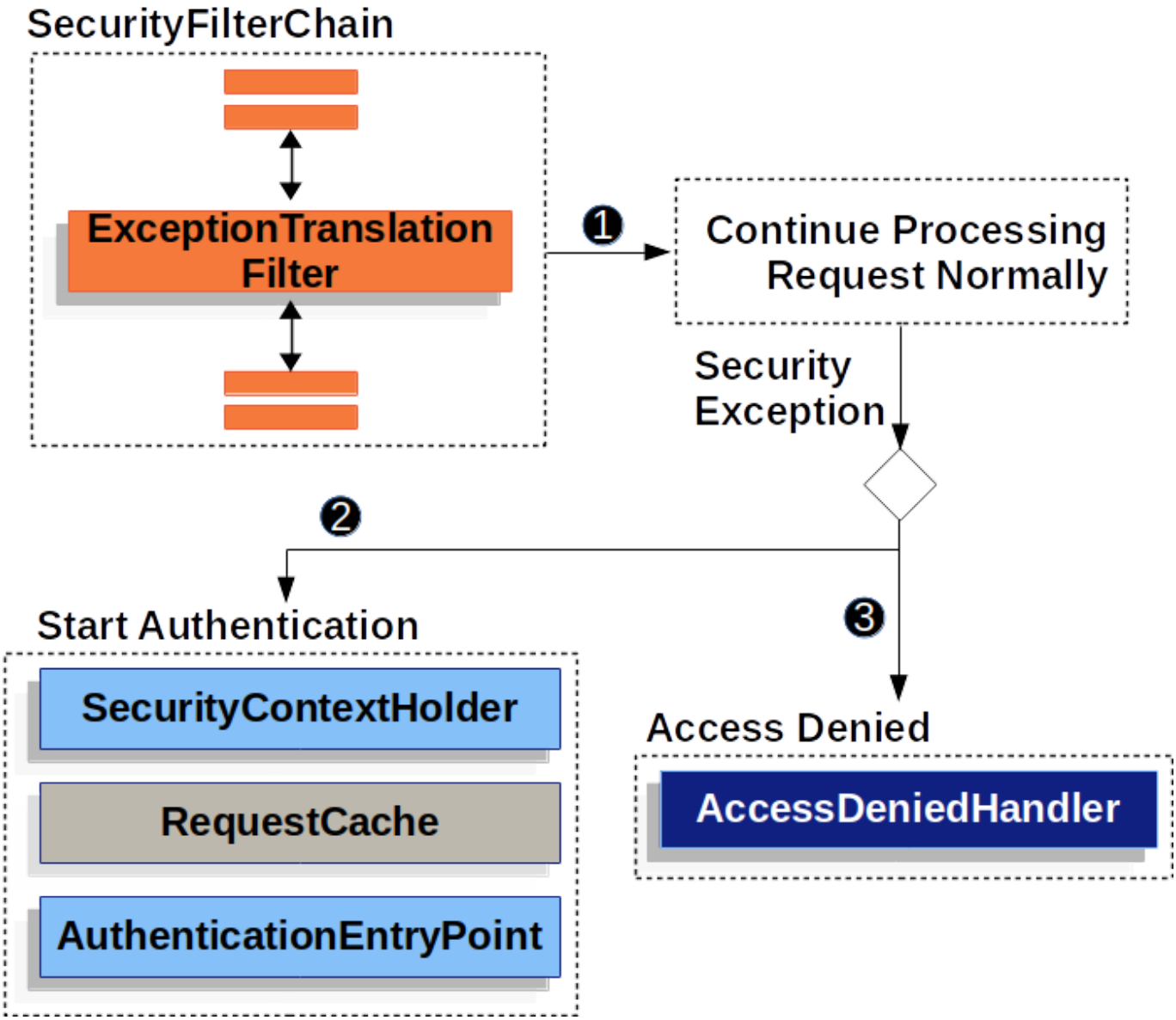
위 샘플 코드는 다음 작업을 수행합니다:

1. 요청 헤더에서 테넌트 ID를 가져옵니다.
2. 현재 사용자가 해당 테넌트 ID에 접근할 수 있는지 확인합니다.
3. 접근이 가능하면 필터 체인의 나머지 필터를 호출합니다.
4. 접근이 불가능하면 `AccessDeniedException`을 던집니다.

다음은 "Handling Security Exceptions"에 대한 번역입니다.

보안 예외 처리

`ExceptionTranslationFilter`는 `AccessDeniedException`과 `AuthenticationException`을 HTTP 응답으로 변환할 수 있도록 합니다. 이 `ExceptionTranslationFilter`는 `FilterChainProxy`의 보안 필터 중 하나로 삽입됩니다.



다음은 `ExceptionTranslationFilter`와 다른 구성 요소 간의 관계를 보여줍니다:

1. 먼저, `ExceptionTranslationFilter`는 `FilterChain.doFilter(request, response)`를 호출하여 애플리케이션의 나머지 부분을 호출합니다.
2. 만약 사용자가 인증되지 않았거나 `AuthenticationException`이 발생하면, "인증 시작" 절차가 진행됩니다.
    - `SecurityContextHolder`가 비워집니다.
    - `HttpServletRequest`가 저장되어, 인증이 성공한 후 원래 요청을 재실행할 수 있도록 합니다.
    - `AuthenticationEntryPoint`가 클라이언트에게 자격 증명을 요청하기 위해 사용됩니다. 예를 들어, 로그인 페이지로 리디렉션하거나 `WWW-Authenticate` 헤더를 보낼 수 있습니다.
3. 그렇지 않고 `AccessDeniedException`이 발생할 경우 "접근 거부" 처리가 이루어집니다. `AccessDeniedHandler`가 호출되어 접근 거부를 처리합니다.

애플리케이션에서 `AccessDeniedException`이나 `AuthenticationException`을 발생시키지 않으면 `ExceptionTranslationFilter`는 아무 작업도 수행하지 않습니다.

`ExceptionTranslationFilter`의 Pseudo Code 예시

```
try {
    filterChain.doFilter(request, response);
} catch (AccessDeniedException | AuthenticationException ex) {
    if (!authenticated || ex instanceof AuthenticationException) {
        startAuthentication();
    } else {
        accessDenied();
    }
}
```

필터 설명에서 언급한 것처럼, `FilterChain.doFilter(request, response)`를 호출하는 것은 애플리케이션의 나머지 부분을 호출하는 것과 같습니다. 즉, 애플리케이션의 다른 부분(예: `FilterSecurityInterceptor` 또는 메서드 보안)에서 `AuthenticationException` 또는 `AccessDeniedException`이 발생하면 여기서 잡아 처리합니다.

- 사용자가 인증되지 않았거나 `AuthenticationException`인 경우 인증 절차를 시작합니다.
- 그렇지 않으면 접근이 거부됩니다.

인증 간 요청 저장

처리 중 인증이 없고 인증이 필요한 리소스를 요청할 경우, 인증이 완료된 후 리소스에 다시 요청할 수 있도록 요청을 저장할 필요가 있습니다. Spring Security에서는 RequestCache 구현을 통해 이를 처리합니다.

RequestCache

HttpServletRequest는 RequestCache에 저장됩니다. 사용자가 성공적으로 인증하면 RequestCache가 원래 요청을 재생하는 데 사용됩니다. RequestCacheAwareFilter는 사용자가 인증한 후 저장된 HttpServletRequest를 가져오는 데 RequestCache를 사용하며, ExceptionTranslationFilter는 AuthenticationException을 감지한 후 로그인 엔드포인트로 리디렉션하기 전에 HttpServletRequest를 저장하는 데 사용합니다.

로깅

Spring Security는 DEBUG 및 TRACE 수준에서 모든 보안 관련 이벤트에 대한 포괄적인 로깅을 제공합니다. 보안상 Spring Security는 요청이 거부된 이유에 대한 세부 정보를 응답 본문에 추가하지 않습니다. 401 또는 403 오류가 발생하면 로그 메시지를 통해 무슨 일이 일어났는지 이해할 수 있는 메시지를 찾을 가능성이 높습니다.

application.properties 설정:

```
logging.level.org.springframework.security=TRACE
```

logback.xml 설정:

```
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <!-- ... -->
    </appender>
    <!-- ... -->
```

```
    <logger name="org.springframework.security" level="trace" additivity="false">
        <appender-ref ref="Console" />
    </logger>
</configuration>
```