J2EE
Lecture 9: Securing Web Applications and Methods

Dr. Ming Qiu
Xiamen University
Software School
mingqiu@xmu.edu.cn

9.1 Getting started with Spring Security

- Spring Security is a security framework that provides declarative security for your Spring-based applications.
  - handling authentication (who are you) and authorization (what are you allowed to do) at both the web request level and at the method invocation level.

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

9.1 Getting started with Spring Security

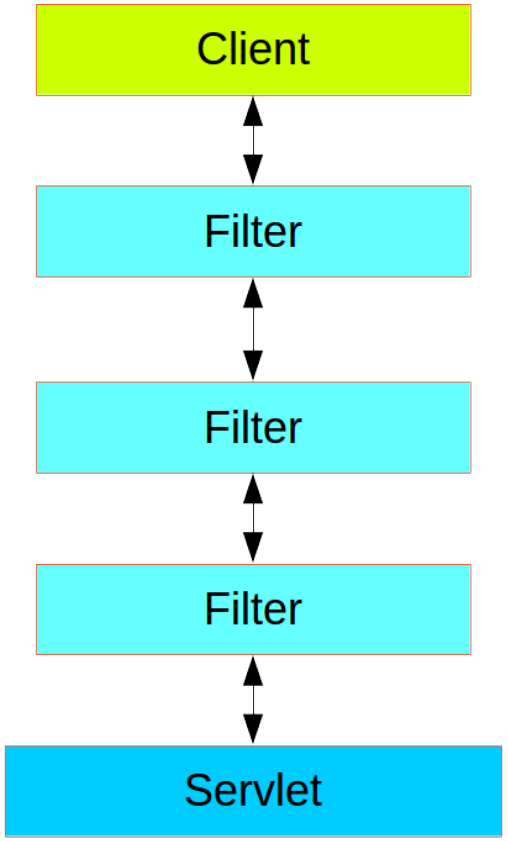
- Unders

Table 9.1 Spring Security is partitioned into eleven modules

Module	Description
ACL	Provides support for domain object security through access control lists (ACLs).
Aspects	A small module providing support for AspectJ-based aspects instead of standard Spring AOP when using Spring Security annotations.
CAS Client	Support for single sign-on authentication using Jasig's Central Authentication Service (CAS).
Configuration	Contains support for configuring Spring Security with XML and Java. (Java configuration support introduced in Spring Security 3.2.)
Core	Provides the essential Spring Security library.
Cryptography	Provides support for encryption and password encoding.
LDAP	Provides support for LDAP-based authentication.
OpenID	Contains support for centralized authentication with OpenID.
Remoting	Provides integration with Spring Remoting.
Tag Library	Spring Security's JSP tag library.
Web	Provides Spring Security's filter-based web security support.

9.1 Getting started with Spring Security

- the typical layering of the handlers for a single HTTP request.



9.1 Getting started with Spring Security

- Filtering web requests
  - DelegatingFilterProxy is a single physical Filter but delegates processing to a chain of internal filters

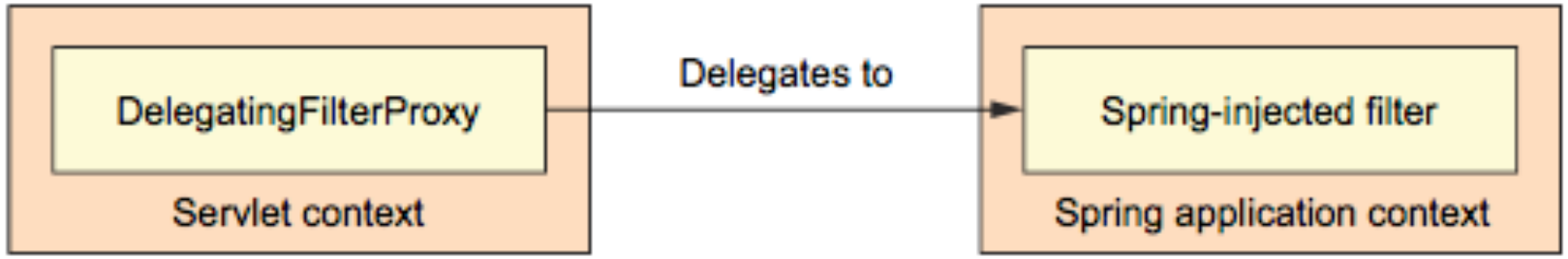
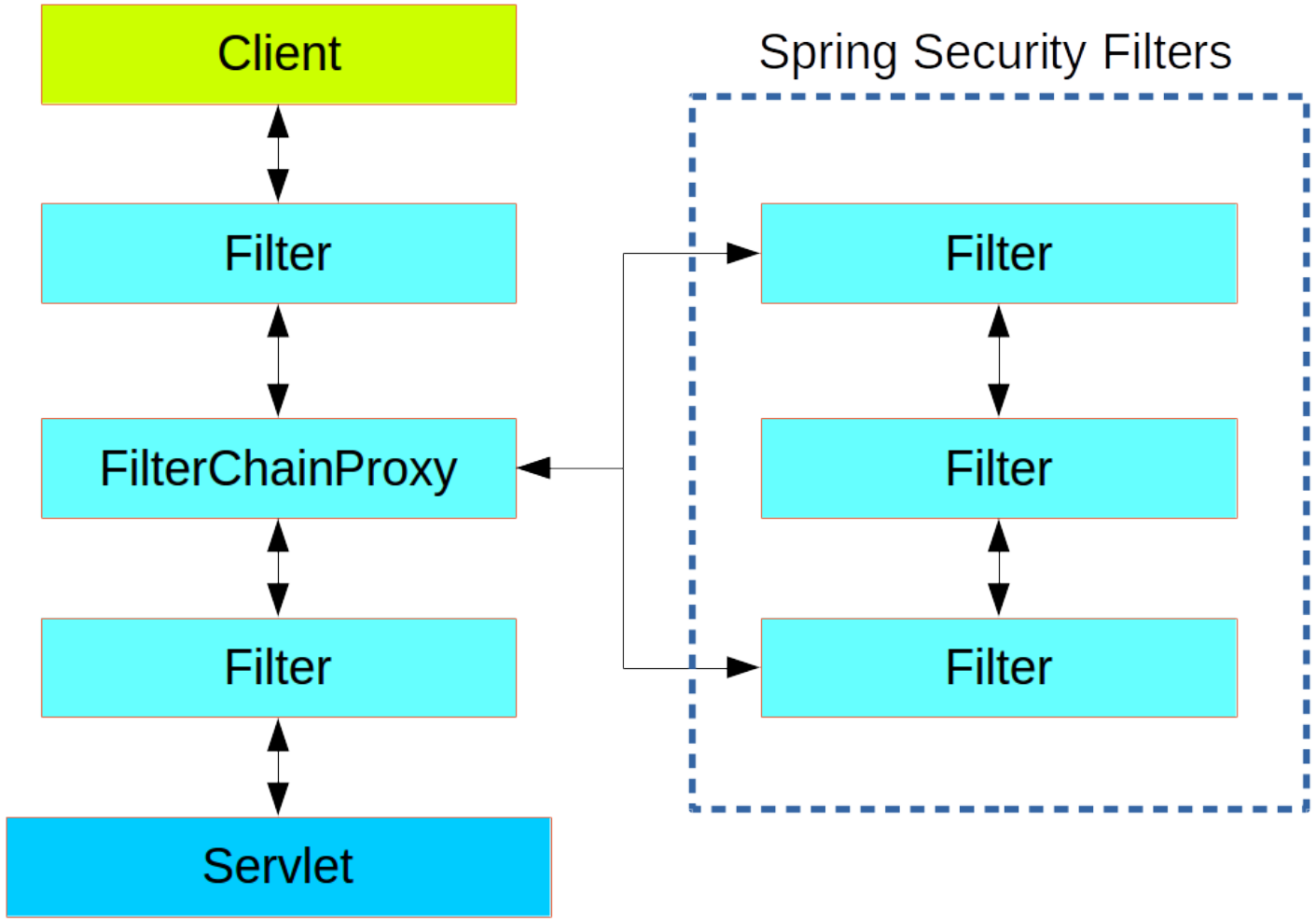
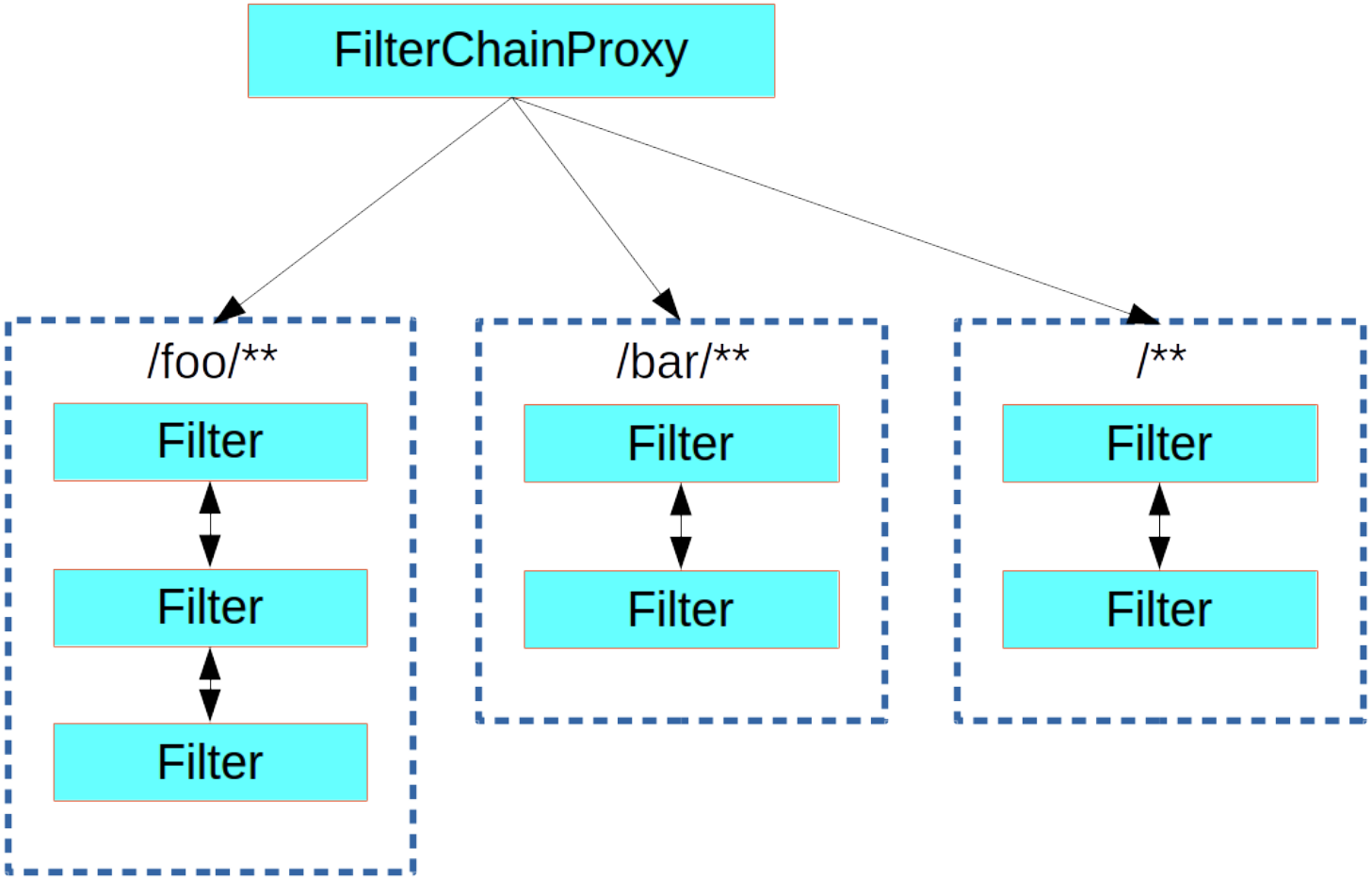


Figure 9.1 `DelegatingFilterProxy` proxies filter handling to a delegate filter bean in the Spring application context.

9.1 Getting started with Spring Security



9.1 Getting started with Spring Security



9.1 Getting started with Spring Security

- Writing a simple security configuration

Listing 9.2 The simplest configuration class to enable web security for Spring MVC

```
package spitter.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.
                                configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.annotation.web.servlet.
                                        configuration.EnableWebMvcSecurity;

@Configuration
@EnableWebMvcSecurity
 public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

Enable Spring MVC security

9.1 Getting started with Spring Security

- Writing a simple security configuration

Table 9.2 Overriding `WebSecurityConfigurerAdapter`'s `configure()` methods

Method	Description
`configure(WebSecurity)`	Override to configure Spring Security's filter chain.
`configure(HttpSecurity)`	Override to configure how requests are secured by interceptors.
`configure(AuthenticationManagerBuilder)`	Override to configure user-details services.

9.1 Getting started with Spring Security

- Writing a simple security configuration
  - Configure a user store
  - Specify which requests should and should not require authentication, as well as what authorities they require
  - Provide a custom login screen to replace the plain default login screen

9.2 Selecting user details services

- Configuring a custom user service

```
public interface UserDetailsService {
       UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException;
}
```

9.2 Sele

- Confi

Listing 9.4 Retrieve a UserDetails object from a SpitterRepository

```
package spittr.security;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.
                                                SimpleGrantedAuthority;
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 spittr.Spitter;
import spittr.data.SpitterRepository;

public class SpitterUserService implements UserDetailsService {

  private final SpitterRepository spitterRepository;

  public SpitterUserService(SpitterRepository spitterRepository) {
     this.spitterRepository = spitterRepository;
  }

  @Override
  public UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException {
    Spitter spitter = spitterRepository.findByUsername(username);
     if (spitter != null) {
      List<GrantedAuthority> authorities =
          new ArrayList<GrantedAuthority>();
      authorities.add(new SimpleGrantedAuthority("ROLE_SPITTER"));

      return new User(
           spitter.getUsername(),
          spitter.getPassword(),
          authorities);
    }
```

Inject SpitterRepository

Look up Spitter

Create authorities list

Return a User

```
package spitter.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.
            configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.annotation.web.servlet.
                                configuration.EnableWebMvcSecurity;
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Autowired
   SpitterRepository spitterRepository;

   @Override
   protected void configure(HttpSecurity http) throws Exception {
         http.authorizeRequests().
             antMatchers("/spitters/me").authenticated()
             .antMatchers(HttpMethod.POST, "/spittles").
             authenticated().anyRequest().permitAll();
   }

   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.userDetailsService(new SpitterUserService(spitterRepository));
  }
}
```

9.3 Intercepting requests

Table 9.4 Configuration methods to define how a path is to be secured

Method	What it does
`access(String)`	Allows access if the given SpEL expression evaluates to `true`
`anonymous()`	Allows access to anonymous users
`authenticated()`	Allows access to authenticated users
`denyAll()`	Denies access unconditionally
`fullyAuthenticated()`	Allows access if the user is fully authenticated (not remembered)
`hasAnyAuthority(String…)`	Allows access if the user has any of the given authorities
`hasAnyRole(String…)`	Allows access if the user has any of the given roles
`hasAuthority(String)`	Allows access if the user has the given authority
`hasIpAddress(String)`	Allows access if the request comes from the given IP address
`hasRole(String)`	Allows access if the user has the given role
`not()`	Negates the effect of any of the other access methods
`permitAll()`	Allows access unconditionally
`rememberMe()`	Allows access for users who are authenticated via remember-me

9.3 Intercepting requests

```
@Override
protected void configure(HttpSecurity http) throws Exception {

   http.authorizeRequests()
        .antMatchers("/spitter/me").hasRole("SPITTER")
        .antMatchers(HttpMethod.POST, "/spittles").hasRole("SPITTER")
        .anyRequest().permitAll();



  .antMatchers("/spitter/me")
  .access("hasRole('ROLE_SPITTER') and hasIpAddress('192.168.1.2')")
```

9.3 Intercepting requests

- Enforcing channel security

Listing 9.5 The `requiresChannel()` method enforces HTTPS for select URLs

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/spitter/me").hasRole("SPITTER")
      .antMatchers(HttpMethod.POST, "/spittles").hasRole("SPITTER")
      .anyRequest().permitAll();
    .and()
    .requiresChannel()
      .antMatchers("/spitter/form").requiresSecure();      <-- Require HTTPS
 }
```

9.3 Intercepting requests

- Preventing cross-site request forgery (CSRF)
  - a CSRF attack happens when one site tricks a user into submitting a request to another server,

```
<form method="POST" action="http://www.spittr.com/spittles">
   <input type="hidden" name="message" value="I'm stupid!" />
   <input type="submit" value="Click here to win a new car!" />
</form>
```

9.3 Intercepting requests

- Preventing cross-site request forgery
  - Spring Security implements CSRF protection with a synchronizer token.
    - State- changing requests (for example, any request that is not GET, HEAD, OPTIONS, or TRACE) will be intercepted and checked for a CSRF token.

9.4 Authenticating users

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http.formLogin().loginPage("/login")
    .and().logout().logoutSuccessUrl("/")
    ...
}
```

9.5 Securing methods with annotations

- Spring Security provides three different kinds of security annotations:
  - Spring Security's own @Secured
  - JSR-250's@RolesAllowed
  - Expression-driven annotations, with @PreAuthorize, @PostAuthorize, @PreFilter, and @PostFilter

9.5 Securing methods with annotations

- Restricting method access with @Secured

```
@SpringBootApplication
@EnableGlobalMethodSecurity(securedEnabled  = true)
public class SampleSecureApplication {
}
```

9.5 Securing methods with annotations

- Restricting method access with @Secured

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled=true)
public class MethodSecurityConfig
   extends GlobalMethodSecurityConfiguration {
}

@Secured({"ROLE_SPITTER", "ROLE_ADMIN"})

public void addSpittle(Spittle spittle) {
     // ...
}
```

9.6 Using expressions for method-level security

Table 14.1 Spring Security 3.0 offers four new annotations that can be used to secure methods with SpEL expressions.

Annotations	Description
`@PreAuthorize`	Restricts access to a method before invocation based on the result of evaluating an expression
`@PostAuthorize`	Allows a method to be invoked, but throws a security exception if the expression evaluates to `false`
`@PostFilter`	Allows a method to be invoked, but filters the results of that method based on an expression
`@PreFilter`	Allows a method to be invoked, but filters input prior to entering the method

9.6 Using expressions for method-level security

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
public void addSpittle(Spittle spittle) {
   // ...
}
```

```
@PreAuthorize( "(hasRole('ROLE_SPITTER')
   and #spittle.text.length() <= 140)" +
   "or hasRole('ROLE_PREMIUM')")
public void addSpittle(Spittle spittle) {
    // ...
}
```

9.6 Using expressions for method-level security

```
@PostAuthorize("returnObject.spitter.username == principal.username")
public Spittle getSpittleById(long id) {
     // ...
}
```

9.6 Using expressions for method-level security

- Filtering method inputs and outputs

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER',
'ROLE_ADMIN'})") @PostFilter( "hasRole('ROLE_ADMIN') ||
" + "filterObject.spitter.username == principal.name")
public List<Spittle> getOffensiveSpittles() {
...
}

@PreAuthorize("hasAnyRole({'ROLE_SPITTER',
'ROLE_ADMIN'})") @PreFilter( "hasRole('ROLE_ADMIN') || "
+ "targetObject.spitter.username == principal.name")
public void deleteSpittles(List<Spittle> spittles) {
 ...
}
```

9.6 Using expressions for method-level security

- Filtering method inputs and outputs

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER',
'ROLE_ADMIN'})")
@PreFilter("hasPermission(targetObject, 'delete')")
public void deleteSpittles(List<Spittle> spittles) {
...
}
```

9.6 Using expressions for method-level security

- Filtering method inputs and outputs

```
@PreAuthorize("hasAnyRole({'ROLE_SPITTER',
'ROLE_ADMIN'})")
@PreFilter("hasPermission(targetObject, 'delete')")
public void deleteSpittles(List<Spittle> spittles) {
...
}
```