Aspect Oriented Programming

2018-2019
Course 8

Course 8 Contents

- Design and implementation of security using (Spring) AOP

Security

- Security is an important consideration in modern, highly connected software systems.
- Most applications need to expose functionality through multiple interfaces to allow access to the business data and make complex integration possible.
- They need to do so in a secured manner.
- Security consists of many components such as authentication, authorization, auditing, protection against web site attacks, and cryptography.
- We focus on authentication and authorization.

Security with conventional techniques

- Implementing security using conventional programming techniques requires the modification of multiple modules to add authentication and authorization code.
- APIs such as Java Authentication and Authorization Service (JAAS) and Spring Security provide a layer of abstraction over the underlying mechanism and let you separate the configuration from the code.
- The overall goal of these APIs is to reduce complexity and provide agile implementations. Yet such APIs offer only a part of the solution. The developers still have to call these APIs from many places in the code.

Security with conventional techniques

- The conventional solutions can be classified in two categories:
  - the *do-it-yourself* approach, where you explicitly invoke security APIs in appropriate places;
  - the *framework-based* approach, where the application framework provides support for common security use cases.

Do-it-yourself Approach

- The classes implementing the business functionality directly call the authentication and authorization methods.
- Business methods that require an access check directly invoke the security system APIs:

```
public void businessMethod() {
    SecurityAttribute sa = ...
    accessManager.checkPermission(sa);
    ... business operation ...
}
```

Creates security attribute

Checks for permission

- Security attributes specify the privileges needed to execute the business operations.
- The access manager decides whether the user accessing the method has sufficient privileges.
- If the user does not have sufficient privileges, the checkPermission() method throws an exception.

Do-it-yourself Approach

- The do-it-yourself approach has many problems:
  - It quickly gets repetitious.
  - It obscures the intention of business logic and prevents reuse with different security constraints.
  - It clutters unit-testing code.
  - It is easy to overlook places where the policy is implemented incorrectly.

Framework Approach

- Frameworks often target a specific class of applications and can take advantage of the nature of the applications to offer simplified security solutions.
- A framework may use AOP as the underlying mechanism as it is done in Spring Security.
- The EJB framework handles authorization by separating the security attributes into the deployment descriptor or annotations, where you specify the required roles for methods in the EJBs. But you may face situations that require a custom solution for authentication and authorization.
- Current EJB frameworks do not offer a good solution to problems that demand for example, data-driven authorization, where an authorization check considers not only the identity of the user and the functionality accessed but also the data involved.

Modularizing security using AOP

- AOP offers a simplified and flexible solution given the crosscutting nature of security.
- Security aspects separate the security-management logic from the core business logic. Business methods can focus on their core logic.

```
public void businessMethod() {
    ... business operation
}
```

- The security aspect needs to do the following:
  - Select join points that need authentication or authorization.
  - Advise the selected join points to perform authentication and authorization.

Modularizing security using AOP

```
aspect SecurityAspect {
   private AccessDecisionManager accessManager;
   pointcut secured() : ...
   before() : secured() {
    SecurityAttribute sa = ...
    accessManager.checkPermission(sa);
   }
}
```

- Computation of the security attributes:
  - In the do it yourself approach, each method creates a separate security attribute object.
  - In an AOP solution, the same advice applies to all advised join points, yet each join point may require a different attribute. Therefore, the advice may need some cooperation from the advised code or some external configuration to compute a correct attribute at each join point. One way to achieve this collaboration is to use annotations.

Modularizing security using AOP

- You can apply security aspects using either the proxy-based or byte-code based AOP.
- Proxy-based AOP does have some limitations:
  - Only spring managed beans can be advised.
  - Only public methods can be advised.
  - Any self-invocations (even to public methods) won't be advised.
- Byte-code based AOP can be used in any situations (not just for methods, but also for fields, etc.)
- The AOP solution relies on an appropriate implementation of the underlying infrastructure. The security aspect acts as a controller that mediates between the core system and the security subsystem.

Spring Security Overview

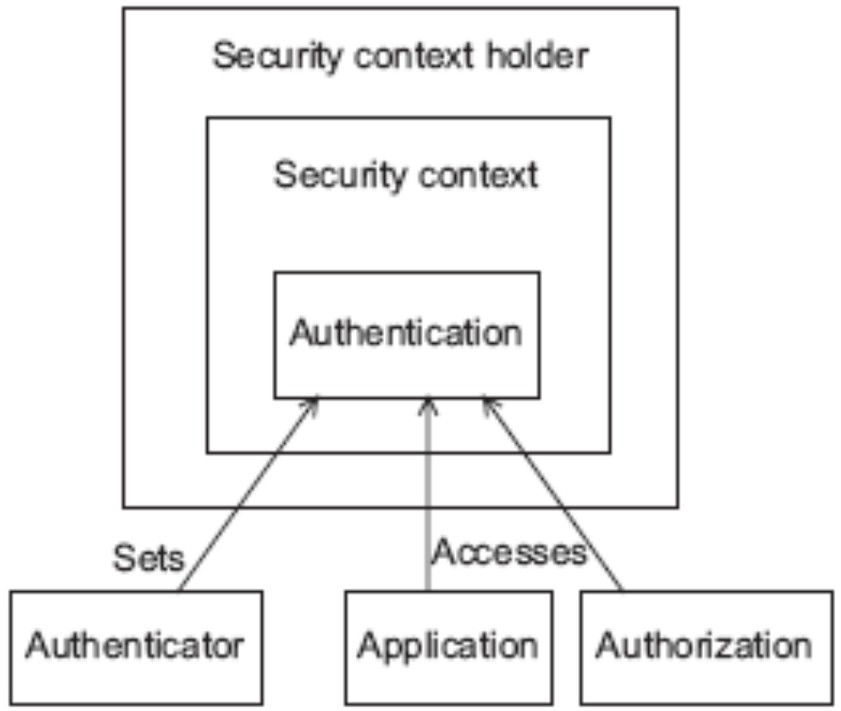
- Security requirements vary widely, and many implementations exist to meet these needs.
- An authentication requirement may vary from simple web-based authentication to a single sign-on solution. Storage for credentials (such as passwords) and authorities (typically roles such as ADMIN or USER) varies widely as well—from a simple text file to a database or Lightweight Directory Access Protocol (LDAP).
- Spring Security provides ready-made solutions for a few common scenarios that allow you to implement certain security requirements by including just a few lines of configuration.

Authentication

- *Authentication* is a process that verifies that the user (human or machine) is indeed whom they claim to be.
- E.g., the system may challenge the user with a username and password. When the user enters that information, the system verifies it against the stored credentials.
- Spring Security supports authentication using a wide range of authentication schemes such as basic, forms, database, LDAP, JAAS, and single sign-on.
- You can also roll your own authentication to support the specific scheme that your organization is using (and still use the rest of the framework, including authorization).
- After the user credentials have been authenticated, the authenticated user (more generally known as the *principal*) is stored in the security context.

Authentication

- The authentication system stores an Authentication object inside a SecurityContext.
- The SecurityContextHolder holds on to one SecurityContext object per scope, which may be a thread-local, an inheritable thread-local, or global.

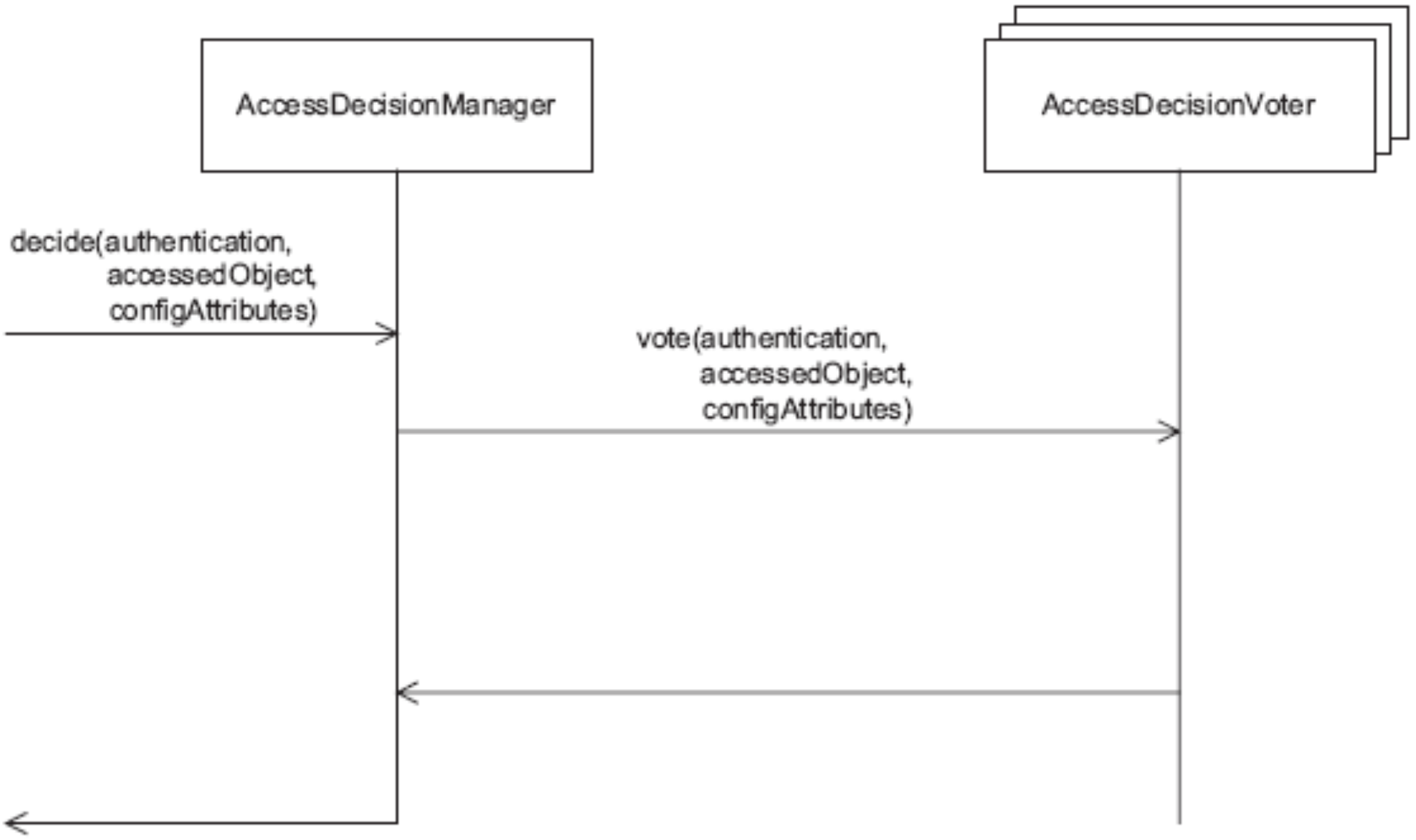


- The authenticator stores the authentication object in the security context. Other parts of the system, including the authorization subsystem, may access the authentication object from the security context.
- The authenticated user may then be accessed by other parts of the system.
- E.g., a service layer may access the principal authenticated by the web layer so that it can determine whether the user has appropriate privileges for the invoked services.

Authorization

- *Authorization* is a process that establishes whether an authenticated user has sufficient privileges to access certain resources.
- For example, only users with the admin privilege may access certain web pages or invoke certain business methods.
- Spring Security provides role-based and object-level authorization. To accommodate potentially complex custom requirements, it provides several components that you can customize.
- An *AccessDecisionManager* has the overall responsibility to determine access to a given object. It makes the determination based on the current Authentication object, the accessed resource (which may be a domain object, an object representing a method, or even a join point), and the security attributes associated with the accessed object. The security attributes specify properties such as the roles required for an authorized access.

Authorization



Authorization

- **`AccessDecisionManager`**, in turn, delegates the responsibility to **`AccessDecisionVoters`** by calling their **`vote()`** method.
- Each **`AccessDecisionVoter`** uses the same information that is passed to the **`AccessDecisionManager's decide()`** method. Voters may vote to allow, deny, or abstain.
- Implementations of **`AccessDecisionManager`** count all the votes and issue the final verdict.
- Spring Security comes with three **`AccessDecisionManager`** implementations based on how they count votes:
  - AffirmativeBased (at least one voter returns an affirmative response),
  - ConsensusBased (a majority of voters return an affirmative response),
  - UnanimousBased (all voters return an affirmative response).

AOP Security Solution- Authentication

- Authentication aspects
- When do you authenticate a user?
  - you can do an up-front login when a user visits the web site or starts a UI or console application,
  - when a user visits a secure web page, or when a user accesses a specific functionality leading to a secured method invocation, etc.
- Depending on your choice, AOP may or may not be the right implementation technique (e.g., if you need to perform authentication only at one specific point, you may not need AOP).
- We focus on the cases where authentication is performed in response to a method being invoked.

AOP Security Solution- Authentication

- Base authentication aspect (using @AspectJ syntax)

```
package ajia.security;
import ...
@Aspect
public abstract class AbstractAuthenticationAspect {
    //support class dependency
    private AuthenticationSupport authenticationSupport;
 @Pointcut  //Authentication pointcut
public abstract void authenticationRequired();
@Before("authenticationRequired()") //Authentication advice
public void authenticate() {
 authenticationSupport.authenticate();
}
public void setAuthenticationSupport(
 AuthenticationSupport authenticationSupport) {
 this.authenticationSupport = authenticationSupport;
}}
```

AOP Security Solution- Authentication

- Base authentication aspect (using @AspectJ syntax):
  - Most of the authentication-related implementation is moved to a support class, which is declared as a dependency (`AuthenticationSupport`). This dependency can be injected through configuration.
  - The abstract `authenticationRequired()` pointcut lets subaspects specify its definition.
  - The before advice to the `authenticationRequired()` pointcut performs authentication by delegating it to the injected dependency. If authentication fails, the support class throws `AuthenticationException`, which is an unchecked exception.

AOP Security Solution- Authentication

- Support class for authentication

```
package ajia.security;
import ...
public class AuthenticationSupport {
   private AuthenticationManager authenticationManager;
   private LoginService loginUI;
   public void authenticate() {
    Authentication authentication=
   SecurityContextHolder.getContext().getAuthentication();
    if(authentication != null) {return;}
    Authentication authenticationToken=
                                        loginUI.getAuthenticationToken();
authentication=authenticationManager.authenticate(authenticationToken);
SecurityContextHolder.getContext().setAuthentication(authentication);
   }//cont ...
```

AOP Security Solution- Authentication

```
public void setAuthenticationManager(AuthenticationManager am)
    {this.authenticationManager = am; }
public void setLoginUI(LoginService loginUI)
    {this.loginUI = loginUI;}
}
```

- The class declares a dependency on Spring Security's `AuthenticationManager`. The injected object needs to be configured with the required objects, such as a credential source. This way, you can plug in any authentication mechanism such as a database, LDAP, or single sign-on.

AOP Security Solution- Authentication

- The class also declares a dependency on a **`LoginService`** object.
- The **`LoginService`** provides the **`getAuthenticationToken()`** method that returns an **`Authentication`** object.
- It also has a dependency on a strategy that obtains the username and password as the default implementation. For a rich UI client application, the implementation could put up a dialog box to ask for the username and password.
- The authentication logic first checks if there is already an authenticated user in the security context. If not, it obtains an authentication token using the login service. It then passes that token to the authentication manager to validate it. The authentication manager throws **`AuthenticationException`** if the credentials don't match. Otherwise, the method stores the validated authentication object in the **`SecurityContext`** so that other parts of the system may obtain it.

AOP Security Solution- Authentication

- Example subaspect

```
package concurs.aspects.security;

import ...

@Aspect
public class ConcursAuthenticationAspect extends
    AbstractAuthenticationAspect {
   //selects adauga methods from ConcursService
   @Pointcut("execution(* concurs.model.ConcursService.adauga*(..))”)
   public void authenticationRequired() { }
}
```

AOP Security Solution- Authentication

- Spring configuration file

```
<security:authentication-manager alias="authenticationManager">
 <security:authentication-provider>
    <security:user-service>
     <security:user name="admin" password="admin"
          authorities="ROLE_USER,ROLE_SUPERVISOR"/>
     <security:user name="test" password="test"
          authorities="ROLE_USER"/>
    </security:user-service>
 </security:authentication-provider>
</security:authentication-manager>
<bean id="authenticationSupport" class="ajia.security.AuthenticationSupport">
   <property name="authenticationManager"    ref="authenticationManager"/>
   <property name="loginUI" ref="loginService"/>
</bean>
```

AOP Security Solution- Authentication

```
<bean id="loginService“ class="concurs.aspects.security.UILoginService"/>
<bean id="authenticationAspect"
   class="concurs.aspects.security.ConcursAuthenticationAspect"
   factory-method="aspectOf">
 <property name="authenticationSupport“ ref="authenticationSupport"/>
</bean>
```

- The `<security:authentication-manager>` element exposes an authentication manager. The embedded `<security:authentication-provider>` element creates a simple in-memory user service with two users: `admin` and `test`.
- You use `factory-method="aspectOf"` only if you use the AspectJ weaver and not when using Spring AOP.
- If you use Spring AOP, you also need an `<aop:aspectj-autoproxy>` declaration.

AOP Security Solution- Authorization

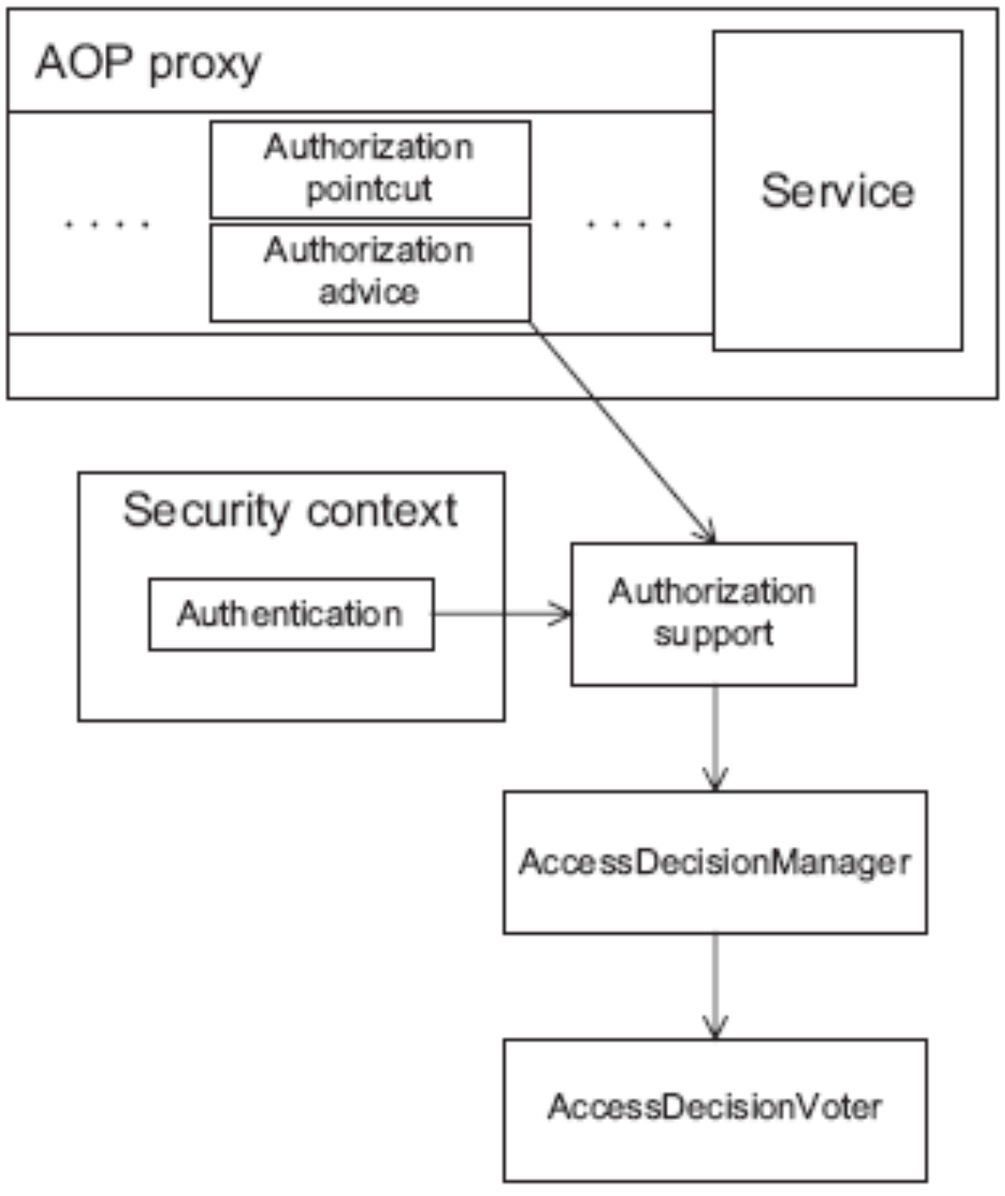
- An authorization aspect ensures that the user has sufficient privileges to access the advised join point.
- Two choices for weaving in the authorization logic: proxy-based (using Spring-AOP) or byte code–based (using AspectJ) weaving.
- Implementing security using proxy-based Spring AOP: Spring AOP creates a proxy around the service beans, and the security advice ensures authorized access.

AOP Security Solution- Authorization

- Securing the service layer using proxy-based AOP and Spring Security.

An aspect consists of a pointcut and an advice.

The advice delegates to a support class.



AOP Security Solution- Authorization

- Base authorization aspect

```
package ajia.security;
import ...
@Aspect
public abstract class AbstractAuthorizationAspect {
   private AuthorizationSupport authorizationSupport;
   @Pointcut
   public abstract void authorizationRequired();
   @Before("authorizationRequired()")
   public void authorize(JoinPoint jp) {
      authorizationSupport.authorize(jp);
   }
   public void setAuthorizationSupport(
        AuthorizationSupport authorizationSupport) {
      this.authorizationSupport = authorizationSupport;
   }
}
```

AOP Security Solution- Authorization

- The authorization aspect delegates to a support class.
- The abstract `authorizationRequired`() pointcut allows subaspects to select the join points that need to be authorized.
- The advice delegates the authorization logic to the injected support object.
- As authorization depends on the join points to compute the security attributes, it passes the current join point to the delegated method.

AOP Security Solution- Authorization

- Support class to simplify implementation:

```
package ajia.security;
import ...
public class AuthorizationSupport {
    private AccessDecisionManager accessDecisionManager;
    private SecurityMetadataSource securityMetadataSource;
    public void authorize(Object securedObject) {
     List<ConfigAttribute> attrs= securityMetadataSource.getAttributes(securedObject);
     Authentication authentication=
                              SecurityContextHolder.getContext().getAuthentication();
     accessDecisionManager.decide(authentication,securedObject, attrs);
    }
    public void setAccessDecisionManager(AccessDecisionManager adm)
     {    this.accessDecisionManager = adm;}

     public void setSecurityMetadataSource(SecurityMetadataSource smds)
     {    this.securityMetadataSource = smds; }
}
```

AOP Security Solution- Authorization

- The support class declares a dependency on an **`AccessDecisionManager`** that performs authorization and a **`SecurityMetadataSource`** that computes the security attributes.
- The **`authorize()`** method obtains a list of **`ConfigAttributes`**, which are the security attributes for the secured resource.
- It then obtains the current **`Authentication`** object from the **`SecurityContextHolder`**.
- Finally it passes all this information to the **`decide()`** method. If the accessing user does not have privileges matching the security attributes, the **`decide()`** method throws an **`AccessDeniedException`**.
- If the current user has not been authenticated yet, it throws an **`InsufficientAuthenticationException`**.
- Both of these are runtime exceptions.

AOP Security Solution- Authorization

- A few possible ways to write subaspects based on how you select join points to authorize.
- The two main selection choices are with or without annotations.
- Subaspect without annotations:
  - Subaspects that don't use annotations require no changes to the core code.
  - They rely on existing program structure such as the package structure, inheritance hierarchy, class and method name patterns, and even annotations not related to security.
  - The result is noninvasive security functionality. But these subaspects are a little complex to implement in the absence of security-related information available from the secured program elements.
  - They can be more fragile, because they often rely on fragile naming conventions.

AOP Security Solution- Authorization

- Simple aspect without annotations

```
package concurs.aspects.security;

import ...

@Aspect

public class ConcursAuthorizationAspect extends
    AbstractAuthorizationAspect {


    @Pointcut("execution(* concurs.model.ConcursService.adauga*(..))”)
    public void authorizationRequired() { }
}
```

- You need to implement a `SecurityMetadataSource` to obtain the security attributes for the advised join point.

AOP Security Solution- Authorization

- SecurityMetadataSource example:

```
package ajia.security;
import ...
public class MappingBasedSecurityMetadataSource implements SecurityMetadataSource {
    private Map<String, String> mapping;
public List<ConfigAttribute> getAttributes(Object jp) {
    String methodName = ((JoinPoint)jp).getSignature().getName();
    for (String pattern : mapping.keySet()) {
     if(PatternMatchUtils.simpleMatch(pattern, methodName)) {
         String role = mapping.get(pattern);
        return SecurityConfig.createList(role);
     }
    }
    throw new IllegalArgumentException("Unknown mapping for " + methodName);
}
public Collection<ConfigAttribute> getAllConfigAttributes() {return null;}
public boolean supports(Class clazz) {
    return clazz.isAssignableFrom(JoinPoint.class);
}
public void setRoleMapping(Map<String, String> mapping) {this.mapping = mapping;}
}
```

AOP Security Solution- Authorization

- The role mapping is configured externally (typically, through Spring's application context).
- The key in the map is a name pattern such as "add*", and the value is the required role to access a method matching the name.
- Authorization configuration for simple aspect:

```
<bean id="accessDecisionManager"
class="org.springframework.security.access.vote.UnanimousBased">
   <property name="decisionVoters">
   <list>
    <bean class="org.springframework.security.access.vote.RoleVoter"/>
   </list>
   </property>
</bean>
```

AOP Security Solution- Authorization

```
<bean id="securityMetadataSource"
    class="ajia.security.MappingBasedSecurityMetadataSource">
    <property name="roleMapping">
        <map>
    <!-- only supervisors can add new participants -->
          <entry key="adaugaP*" value="ROLE_SUPERVISOR" />
   <!-- any authenticated user can add new results -->
          <entry key="adaugaR*" value="ROLE_USER" />
        </map>
     </property>
</bean>
<bean id="authorizationSupport" class="ajia.security.AuthorizationSupport">
    <property name="accessDecisionManager" ref="accessDecisionManager"/>
    <property name="securityMetadataSource" ref="securityMetadataSource"/>
</bean>
<bean id="authorizationAspect"
    class="concurs.aspects.security.ConcursAuthorizationAspect"
     factory-method="aspectOf">
    <property name="authorizationSupport" ref="authorizationSupport"/>
</bean>
```

AOP Security Solution- Authorization

- You need to ensure that authentication happens before authorization if both take place at the same join point.
- The `SecurityCoordinationAspect` declares higher precedence for subaspects of AbstractAuthenticationAspect over subaspects of AbstractAuthorizationAspect

```
package ajia.security;
import ...
@Aspect
@DeclarePrecedence("AbstractAuthenticationAspect+,"
+ "AbstractAuthorizationAspect+")
public class SecurityCoordinationAspect {
}
```

- If you use Spring AOP, you make each aspect implement the `Ordered` interface or attach the `@Order` annotation to each aspect and configure the order for `ConcursAuthenticationAspect` to be a lower number than that for `ConcursAuthorizationAspect`

AOP Security Solution- Authorization

- Annotation-driven subaspect
  - The core idea behind the annotation-driven approach is to attach authorized types and/or methods with annotations that specify security-related information such as the expected roles.
  - The aspect can then select all methods that carry such annotations and compute the security attributes based on the annotation values.

AOP Security Solution- Authorization

- Annotation-driven authorization aspect

```
package ajia.security;

import ...

@Aspect

public class SecuredAnnotationAuthorizationAspect extends
   AbstractAuthorizationAspect {

// The wildcard in org.springframework..Secured is used
//  only for better formatting

@Pointcut(

"execution(@org.springframework..Secured * *(..))"

+ "|| execution(* (@org.springframework..Secured *).*(..))")

public void authorizationRequired() {}

}
```

AOP Security Solution- Authorization

- The only change compared to **`ConcursAuthorizationAspect`** is the pointcut expression.
- Instead of selecting every service method, this pointcut selects every method marked with the **`@Secured`** annotation as well as every method in a type marked with the **`@Secured`** annotation.
- Spring Security provides an implementation of **`SecurityMetadataSource`** in the form of **`SecuredMethodSecurityMetadataSource`** to extract the security attribute from the **`@Secured`** annotation.

```
@Secured("ROLE_USER")
public class InventoryServiceImpl implements InventoryService {



@Secured("ROLE_SUPERVISOR")
public void expireProduct(Product product) {...}
    ...
}
```

AOP Security Solution- Authorization

- Authorization configuration for a simple annotated aspect

```
<!--accessDecisionManager and authorizationSupport →

<bean id="securityMetadataSource“
   class="org.springframework.security.access.annotation.SecuredAnnotati
   onSecurityMetadataSource"/>
<bean id="authorizationAspect"
   class="ajia.security.SecuredAnnotationAuthorizationAspect"
   factory-method="aspectOf">
<property name="authorizationSupport“ ref="authorizationSupport"/>
</bean>
```

AOP Security Solution- Field Level

- There are situations when security is at the individual field level and not at the object level.
- Because the security-access logic applies to the fields of domain objects, proxy based AOP cannot be used. You must use AspectJ-based weaving.
- The decision to determine access may be based on roles, object state, or a combination of both.
- The core idea behind implementing field-access checks is similar to method access. The only real difference is that instead of using method-level annotations, you use field-level annotations. By using a field-access pointcut, you can enforce the access check.
- The use of annotations is not the only choice for field level security. It's valid to enforce access checks based on packages, inheritance hierarchy, and so on.

AOP Security Solution- Field Level

- Custom annotation to mark secured fields:

```
package ajia.security;

import ...

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Access {

   String[] value();

}
```

- This annotation is similar to the @Secured annotation. The only difference is that you allow the annotation target to be a field.

AOP Security Solution- Field Level

- Domain entity with secured fields

```
package ajia.domain;

import ...

public class Product extends DomainEntity {

   private String name;

   private String description;

   @Access("ROLE_SUPERVISOR") private double price;

   //...

}
```

AOP Security Solution- Field Level

- Aspect advising access to secured fields

```
package ajia.security;
import ...
@Aspect
public class AccessFieldSecurityAspect {
    private AuthorizationSupport securitySupport;
    @Pointcut("(get(@Access * *) || set(@Access * *)) "+
          "&& annotation(access)")
    public void secureFieldAccess(Access access) {}
    @Before("secureFieldAccess(access)")
    public void secure(Access access) {
     securitySupport.authorize(access);
    }
    public void setSecuritySupport(AuthorizationSupport securitySupport) {
     this.securitySupport = securitySupport;
    }
}
```

AOP Security Solution- Field Level

- Configuration of the aspect

```
<beans ...>
<bean id="securityMetadataSource"
   class="ajia.security.AccessAnnotationMetadataSource"/>
<bean id="accessDecisionManager"
 class="org.springframework.security.access.vote.UnanimousBased">
   <property name="decisionVoters">   <list>
   <bean class="org.springframework.security.access.vote.RoleVoter" />
   </list></property>
</bean>
<bean id="fieldSecurityAspect"
   class="ajia.security.AccessFieldSecurityAspect"
   factory-method="aspectOf">
   <property name="securitySupport" ref="securitySupport"/>
</bean>
```

AOP Security Solution- Field Level

- Configuration of the aspect

```
<bean id="securitySupport“ class="ajia.security.AuthorizationSupport">

   <property name="accessDecisionManager“
                                  ref="accessDecisionManager" />

   <property name="securityMetadataSource“

   ref="securityMetadataSource"/>

   </bean>

</beans>
```

Spring Security prebuilt solutions

- Spring Security provides ready-made solutions that enable developers to secure applications with a few lines of configuration.
- These solutions target different parts of application: web, service layer, and domain objects.

- Service level security:
  - Spring Security provides prebuilt aspects along with namespace-based configuration support to secure the service layer with minimal code (more precisely, you can secure any Spring bean, not just those in the service layer).
  - It offers two options to specify the access-control information: through the XML-based configuration and through annotations.

Spring Security prebuilt solutions

- External configuration option: XML code specifies the secured methods as well as the mapping between methods and roles.

```
<security:global-method-security>
   <security:protect-pointcut
   expression="execution(* ajia.service.*.delete*(..))"
   access="ROLE_ADMIN"/>
</security:global-method-security>
```

- The protect-pointcut element specifies a pointcut and the required privileges to access the join points selected by that pointcut.
- E.g., only users with the ADMIN role may access any method whose name starts with `delete` in a class in the `ajia.service` package.

Spring Security prebuilt solutions

- Annotation-based option: Annotations provide another way to enable method-level security. Instead of writing a pointcut, you attach types and methods with annotations that specify the security attributes.

```
package ajia.service.impl;
import...
@Secured("ROLE_SUPERVISOR")
public class InventoryServiceImpl implements InventoryService {
...
public void addProduct(Product product, int quantity) {...}
public void removeProduct(Product product, int quantity) {...}
@Secured("ROLE_USER")
public boolean isProductAvailable(Product product, int quantity) {...}
}
```

The XML configuration reduces to this:

```
<global-method-security secured-annotations="enabled">
```

Spring Security

- In order to use Spring Security core classes you have to add the following jar files to the project build path:
  - **spring-security-aspects-x.y.z.RELEASE.jar**
  - **spring-security-config-x.y.z.RELEASE.jar**
  - **spring-security-core-x.y.z.RELEASE.jar**
- You have to download the latest Spring Security archive from Spring homepage.