Spring Security

Web Application Security

Addressing Common Web Application Security Requirements

Topics in this Session

- **High-Level Security Overview**
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Security Concepts

- **Principal**
  - User, device or system that performs an action
- **Authentication**
  - Establishing that a principal's credentials are valid
- **Authorization**
  - Deciding if a principal is allowed to perform an action
- **Authority**
  - Permission or credential enabling access (such as a role)
- **Secured Item**
  - Resource that is being secured

Authentication

- There are many authentication mechanisms
  - e.g. basic, digest, form, X.509
- There are many storage options for credential and authority information
  - e.g. Database, LDAP, in-memory (development)

Authorization

- Authorization depends on authentication
  - Before deciding if a user can perform an action, user identity must be established
- Authorization determines if you have the required *Authority*
- The decision process is often based on roles
  - *ADMIN* can cancel orders
  - *MEMBER* can place orders
  - *GUEST* can browse the catalog

A Role is simply a commonly used type of Authority

Topics in this Session

- High-Level Security Overview
- **Motivations of Spring Security**
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

See: Spring Security Reference
http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/

Motivations - I

- **Portable**
  - Secured archive (WAR, EAR) can be deployed as-is
  - Also runs in standalone environments
  - Uses Spring for configuration
- **Separation of Concerns**
  - Business logic is decoupled from security concerns
  - Authentication and Authorization are decoupled
    - Changes to the authentication process have *no impact* on authorization

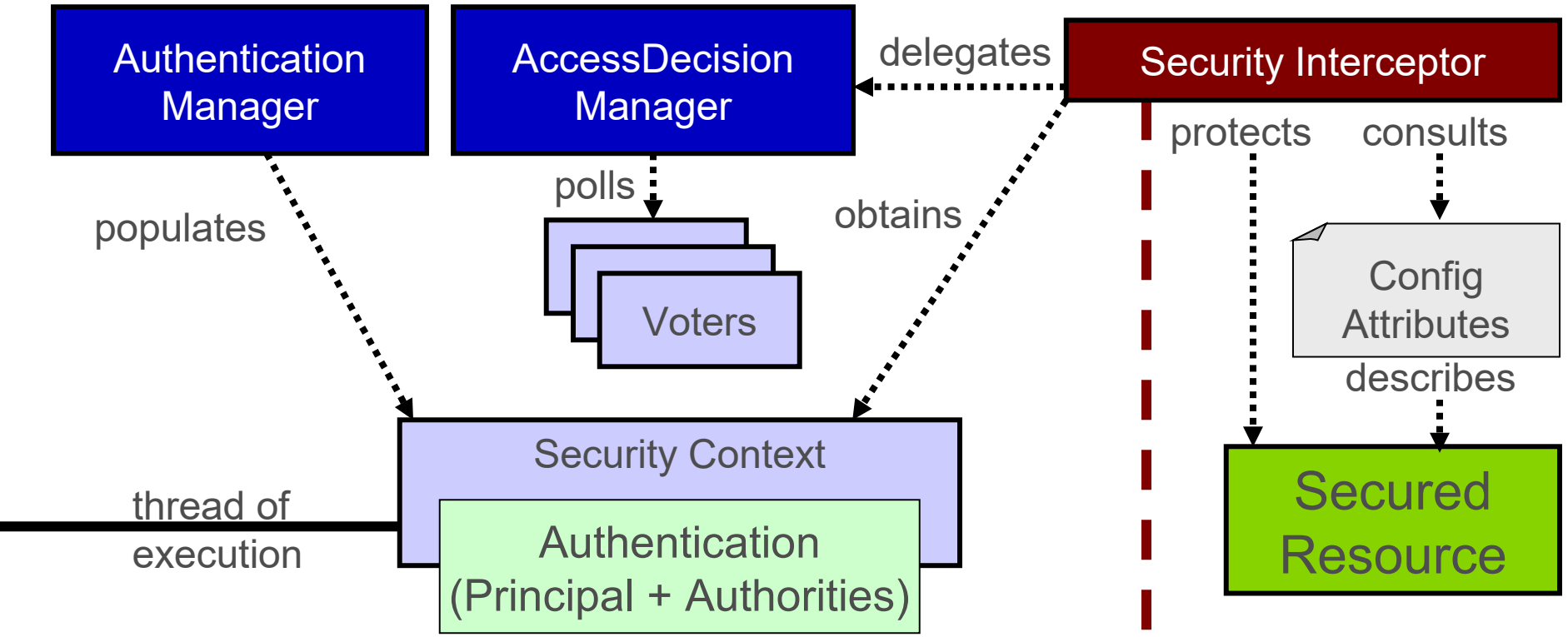
Motivations: II

- **Flexibility**
  - Supports all common authentication mechanisms
    - Basic, Form, X.509, Cookies, Single-Sign-On, etc.
  - Configurable storage options for user details (credentials and authorities)
    - Properties file, RDBMS, LDAP, custom DAOs, etc.
- **Extensible**
  - All the following can be customized
    - How a principal is defined
    - How authorization decisions are made
    - Where security constraints are stored

Consistency of Approach

- *The goal of authentication is always the same*
  - Regardless of the underlying mechanism
  - Establish a security context with the authenticated principal's information
  - Out-of-the-box this works for web applications
- *The process of authorization is always the same*
  - Regardless of the underlying mechanism
  - Consult the attributes of secured resource
  - Obtain principal information from security context
  - Grant or deny access

Spring Security – the Big Picture



Setup and Configuration
Spring Security in a Web Environment

Three steps

1) Setup filter chain
2) Configure security (authorization) rules
3) Setup Web Authentication

Spring Security is **not** limited to Web security, but that is all we will consider here, and it is configurable “out-of-the-box”

Spring Security Filter Chain

- Implementation is a *chain* of Spring configured filters
  - Requires a **`DelegatingFilterProxy`** which *must* be called *springSecurityFilterChain*
  - Chain consists of many filters

- Setup filter chain using *one* of these options
  - Spring Boot does it automatically
  - Use **`@EnableWebSecurity`**
  - Declare as a **`<filter>`** in **`web.xml`** in usual way

For more details (and a **`web.xml`** example) see "Advanced security: working with filters" at end of this topic.

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- **Spring Security in a Web Environment**
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Configuration in the Application Context

- Java Configuration (XML also available)
  - Extend *WebSecurityConfigurerAdapter* for more control

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

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

  }

  @Override
  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

  }
}
```

Web-specific security settings

General security settings (authentication manager, ...).

authorizeRequests()

- Adds specific authorization requirements to URLs
- Evaluated in the order listed
  - first match is used, put specific matches first

```
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/css/**","/images/**","/javascript/**").permitAll()
      .antMatchers("/accounts/edit*").hasRole("ADMIN")
      .antMatchers("/accounts/account*").hasAnyRole("USER","ADMIN")
      .antMatchers("/accounts/**").authenticated()
      .antMatchers("/customers/checkout*").fullyAuthenticated()
      .antMatchers("/customers/**").anonymous();
```

Match *all* URLs starting with **/customers** (ANT-style path)

Specifying login and logout

```
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .antMatchers("/aaa*").hasRole("ADMIN")
      .and()                                  //  method chaining!

    .formLogin()                              //  setup form-based authentication
      .loginPage("/login.jsp")                //  URL to use when login is needed
      .permitAll()                            //  any user can access
      .and()                                  //  method chaining!

    .logout()                                 //  configure logout
      .logoutSuccessUrl("/home")              //  go here after successful logout
      .permitAll();                           //  any user can access
}
```

An Example Login Page

URL that indicates an authentication request.
Default: POST against URL used to display the page.

```
<c:url var='loginUrl'value='/login.jsp' />
<form:form action="${loginUrl}" method="POST">
  <input type="text" name="username"/>
  <br/>
  <input type="password" name="password"/>
  <br/>
  <input type="submit" name="submit" value="LOGIN"/>
</form:form>
```

login-example.jsp

The expected keys for generation of an authentication request token

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- **Configuring Web Authentication**
- Using Spring Security's Tag Libraries
- Method security
- Advanced security: working with filters

Configure Authentication

- DAO Authentication provider is default
  - Expects a *UserDetailsService* implementation to provide credentials and authorities
    - Built-in: In-memory (properties), JDBC (database), LDAP
    - Custom
- Or define your *own* Authentication provider
  - *Example:* to get pre-authenticated user details when using single sign-on
    - CAS, TAM, SiteMinder ...
  - See online examples

Authentication Provider

- Use a *UserDetailsManagerConfigurer*
  - Three built in options:
    - LDAP, JDBC, in-memory (for quick testing)
  - Or use your own UserDetailsService implementation

```
@Override
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .inMemoryAuthentication()
            .withUser("hughie").password("hughie").roles("GENERAL").and()
            .withUser("dewey").password("dewey").roles("ADMIN").and()
            .withUser("louie").password("louie").roles("SUPPORT");
}
```

Not web-specific

Adds a *UserDetailsManagerConfigurer*

login

password

Supported roles

Sourcing Users from a Database – 1

```
public DataSource dataSource;

@Autowired
public void setDataSource(DataSource dataSource) throws Exception {
    this.dataSource = dataSource;;
}

@Override
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource);
}
```

Can customize queries using methods:
- usersByUsernameQuery()
- authoritiesByUsernameQuery()
- groupAuthoritiesByUsername()

Sourcing Users from a Database – 2

Queries RDBMS for users and their authorities

- Provides default queries
  - SELECT username, password, enabled FROM users WHERE username = ?
  - SELECT username, authority FROM authorities WHERE username = ?
- Groups also supported
  - groups, group_members, group_authorities tables
  - See online documentation for details
- Advantage
  - Can modify user info while system is running

Password Encoding – 1

- Can encode passwords using a hash
  - sha256, bcrypt, (sha, md5, …)
  - Use with *any* authentication mechanism

```
auth.inMemoryAuthentication()
  .passwordEncoder(new StandardPasswordEncoder());
```

SHA-256 by default

- Add a “salt” string to make encryption stronger
  - Salt prepended to password before hashing

```
auth.jdbcAuthentication()
  .dataSource(dataSource)
  .passwordEncoder(new StandardPasswordEncoder(“Spr1nGi$Gre@t”));
```

encoding with a 'salt' string

Password Encoding – 2

`BCryptPasswordEncoder` is recommended – uses Blowfish

- BCrypt is recommended over SHA-256
  - Secure passwords further by specifying a "strength" (N)
  - Internally the hash is rehashed 2^N times, default is 2^{10}

```
auth.inMemoryAuthentication()
  .passwordEncoder(new BCryptPasswordEncoder(12));
```

Encoding using '*strength*' 12

- Store *only* encrypted passwords

```
auth.inMemoryAuthentication()
   .withUser("hughie")
   .password("$2a$10$aMxNkanIJ...Ha.h5NKknelEuylt87PNlicYpI1y.IG0C.")
   .roles("GENERAL")
```

Other Authentication Options

- Implement a custom UserDetailsService
  - Delegate to an existing User repository or DAO
- LDAP
- X.509 Certificates
- JAAS Login Module
- Single-Sign-On
  - OAuth, SAML
  - SiteMinder, Kerberos
  - JA-SIG Central Authentication Service

Authorization is *not* affected by changes to Authentication!

@Profile with Security Configuration

```
public class SecurityBaseConfig extends WebSecurityConfigurerAdapter {
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/resources/**").permitAll();
  }
}
```

```
@Configuration
@EnableWebSecurity
@Profile("development")
public class SecurityDevConfig extends SecurityBaseConfig {
  @Override
  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
          .withUser("hughie").password("hughie").roles("GENERAL");
  }
}
```

Use in-memory provider

@Profile with Security Configuration

```
public class SecurityBaseConfig extends WebSecurityConfigurerAdapter {
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/resources/**").permitAll();
  }
}
```

```
@Configuration
@EnableWebSecurity
@Profile("production")
public class SecurityProdConfig extends SecurityBaseConfig {
  @Override
  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
     auth.jdbcAuthentication().dataSource(dataSource);
  }
}
```

Use database provider

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- **Using Spring Security's Tag Libraries**
- Method security
- Advanced security: working with filters

Tag library declaration

- The Spring Security tag library is declared as follows

available since Spring Security 2.0

```jsp
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags" %>
```

- Equivalent functionality for other View technologies
  - Velocity, Freemarker, Thymeleaf, JSF ...

Spring Security's Tag Library

- Display properties of the Authentication object

```jsp
You are logged in as:
<security:authentication property="principal.username"/>
```

- Hide sections of output based on role
  - Implementation on next slide

```jsp
<security:authorize ... >
    TOP-SECRET INFORMATION
    Click <a href="/admin/deleteAll">HERE</a>
    to delete all records.
</security:authorize>
```

Content hidden from unauthorized users

JSP Authorization – Using an intercept-url

- Restrict via URL permissions
  - *Must* specify **`@EnableWebSecurity`** (even with Spring Boot)

```
<security:authorize url="/admin/deleteAll">
   TOP-SECRET INFORMATION
   Click <a href="/admin/deleteAll">
   HERE</a>
</security:authorize>                                    jsp
```

Protected URL

Pattern that matches the URL to be protected

```
.antMatchers("/admin/*")
      .hasAnyRole("USER","ADMIN")
```

Matching roles

Topics in this Session

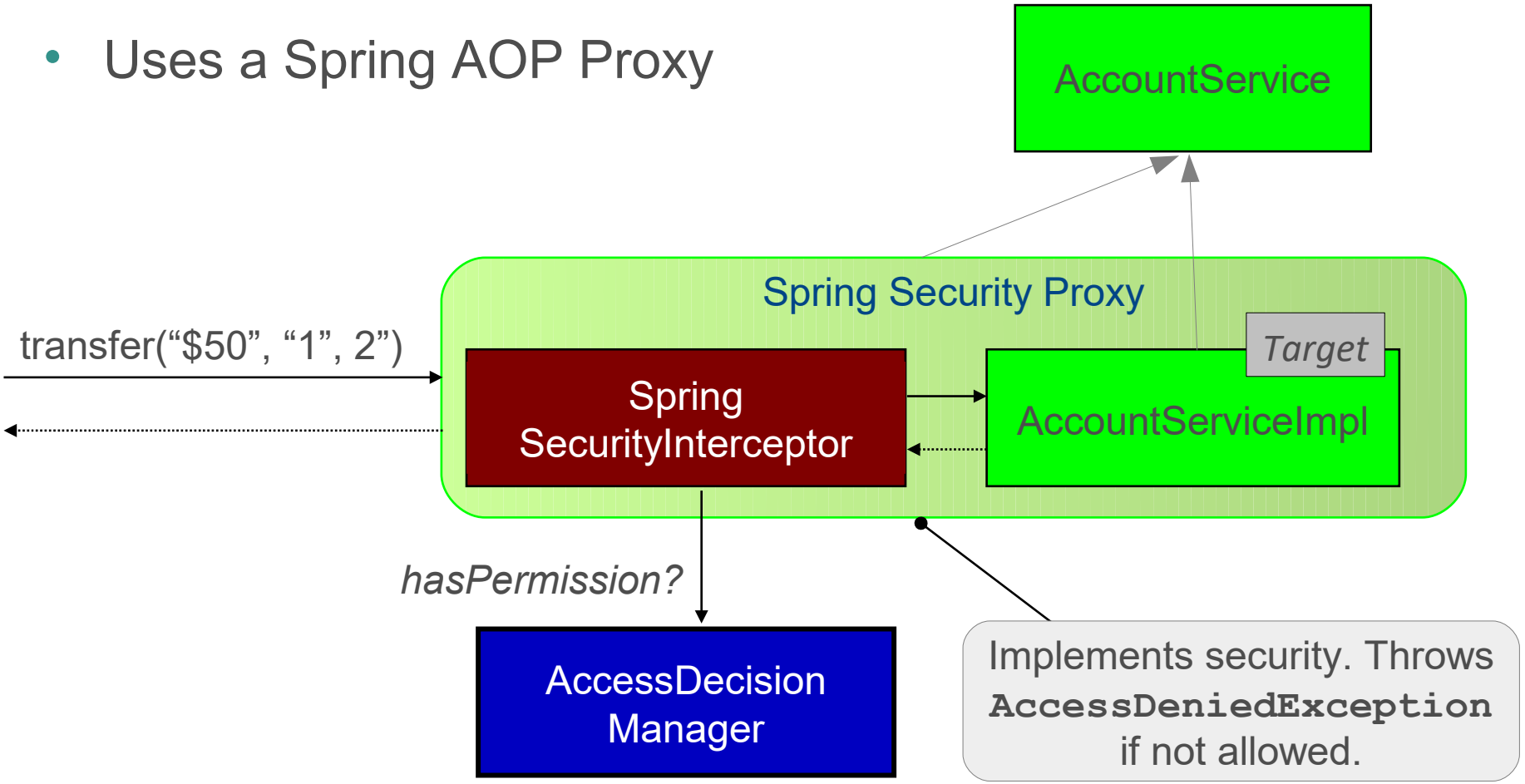
- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- **Method security**
- Advanced security: working with filters

Method Security

- Spring Security uses AOP for security at the method level
  - annotations based on Spring annotations or JSR-250 annotations
  - Java configuration to activate detection of annotations
- Typically secure your services
  - Do not access repositories directly, bypasses security (and transactions)

Method Security – How it Works

- Uses a Spring AOP Proxy



Method Security - JSR-250

- Only supports **role-based** security (hence the name)
  - JSR-250 annotations must be enabled

```
@EnableGlobalMethodSecurity(jsr250Enabled=true)
```

```
import javax.annotation.security.RolesAllowed;

public class ItemManager {
   @RolesAllowed({"ROLE_MEMBER", "ROLE_USER"})
   public Item findItem(long itemNumber) {
      ...
   }
}
```

Can also place at class level

```
@RolesAllowed({"ROLE_MEMBER", "ROLE_USER"})
```

Internally role authorities are stored with **`ROLE_`** prefix. APIs seen previously hide this. Here, and next slide, you *must* use full name

Method Security - @Secured

- Secured annotation should be enabled

```
@EnableGlobalMethodSecurity(securedEnabled=true)
```

```
import org.springframework.security.annotation.Secured;

public class ItemManager {
   @Secured("IS_AUTHENTICATED_FULLY")
   public Item findItem(long itemNumber) {
      ...
   }
}
```

Can also place at class level

```
@Secured("ROLE_MEMBER")
@Secured({"ROLE_MEMBER", "ROLE_USER"})
```

*Spring 2.0 syntax, **not** limited to roles. SpEL **not** supported.*

Method Security with SpEL

- Use Pre/Post annotations for SpEL

```
@EnableGlobalMethodSecurity(prePostEnabled=true)
```

```
import org.springframework.security.annotation.PreAuthorize;

public class ItemManager {
   @PreAuthorize("hasRole('MEMBER')")
   public Item findItem(long itemNumber) {
      ...
   }
}
```

Full role-names *not* required. `ROLE_` prepended automatically.

Summary

- Spring Security
  - Secure URLs using a chain of Servlet filters
  - And/or methods on Spring beans using AOP proxies
- Out-of-the-box setup usually sufficient – you define:
  - URL and/or method restrictions
  - How to login (typically using an HTML form)
  - Supports in-memory, database, LDAP credentials (and more)
  - Password encryption using familiar hashing techniques
  - Support for security tags in JSP views

Lab

Applying Spring Security to a Web Application

Topics in this Session

- High-Level Security Overview
- Motivations of Spring Security
- Spring Security in a Web Environment
- Configuring Web Authentication
- Using Spring Security's Tag Libraries
- Method security
- **Advanced Security**
  - **Working with Filters**

Spring Security in a Web Environment

- *SpringSecurityFilterChain*
  - **Always** first filter in chain
- This single proxy filter delegates to a chain of Spring-managed filters to:
  - Drive authentication
  - Enforce authorization
  - Manage logout
  - Maintain SecurityContext in HttpSession
  - and more

Example: Configuration in web.xml

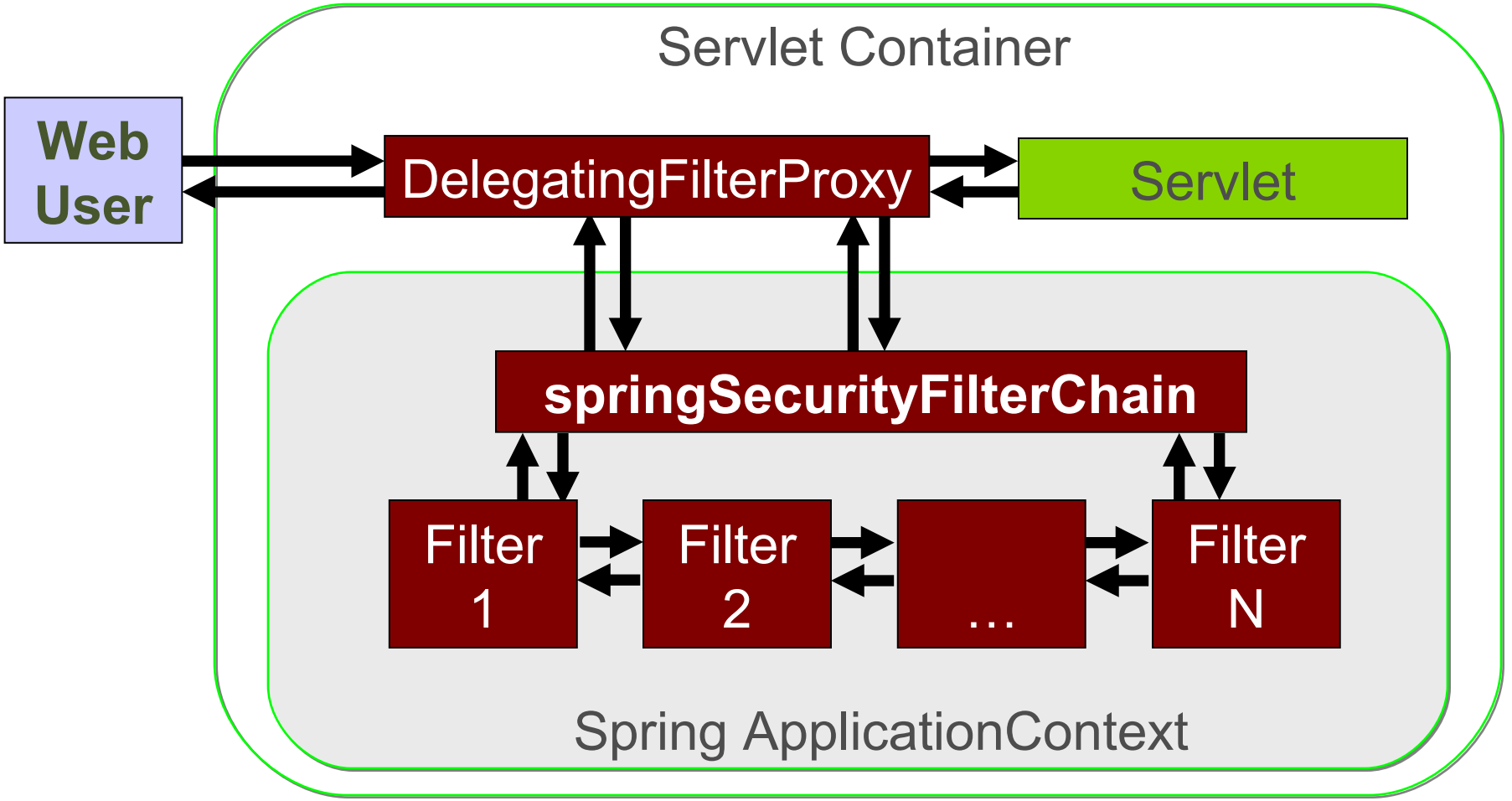
- Define the single proxy filter
  - *springSecurityFilterChain* is a mandatory name
    - Refers to an existing Spring bean with same name

web.xml

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
     org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

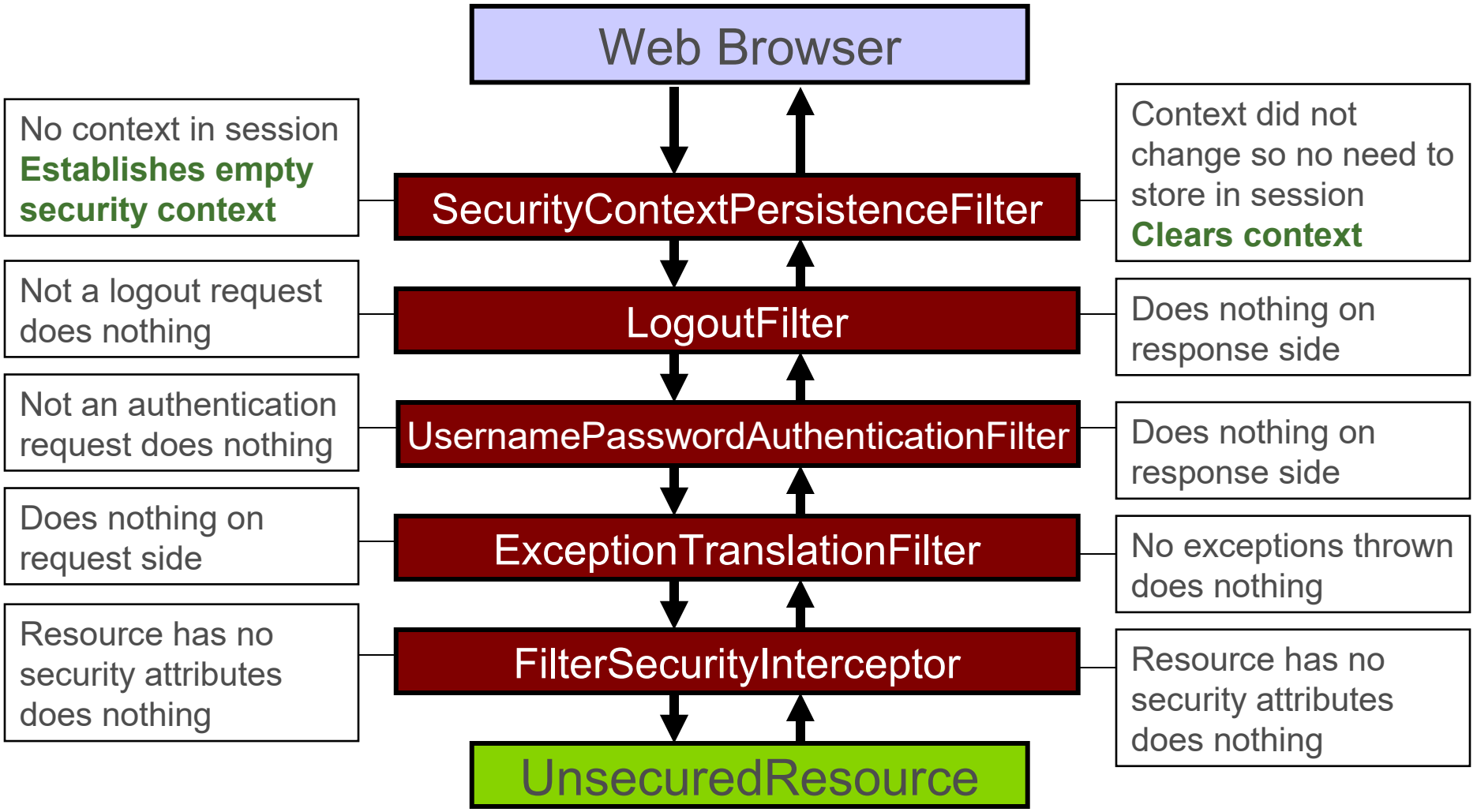
Web Security Filter Configuration



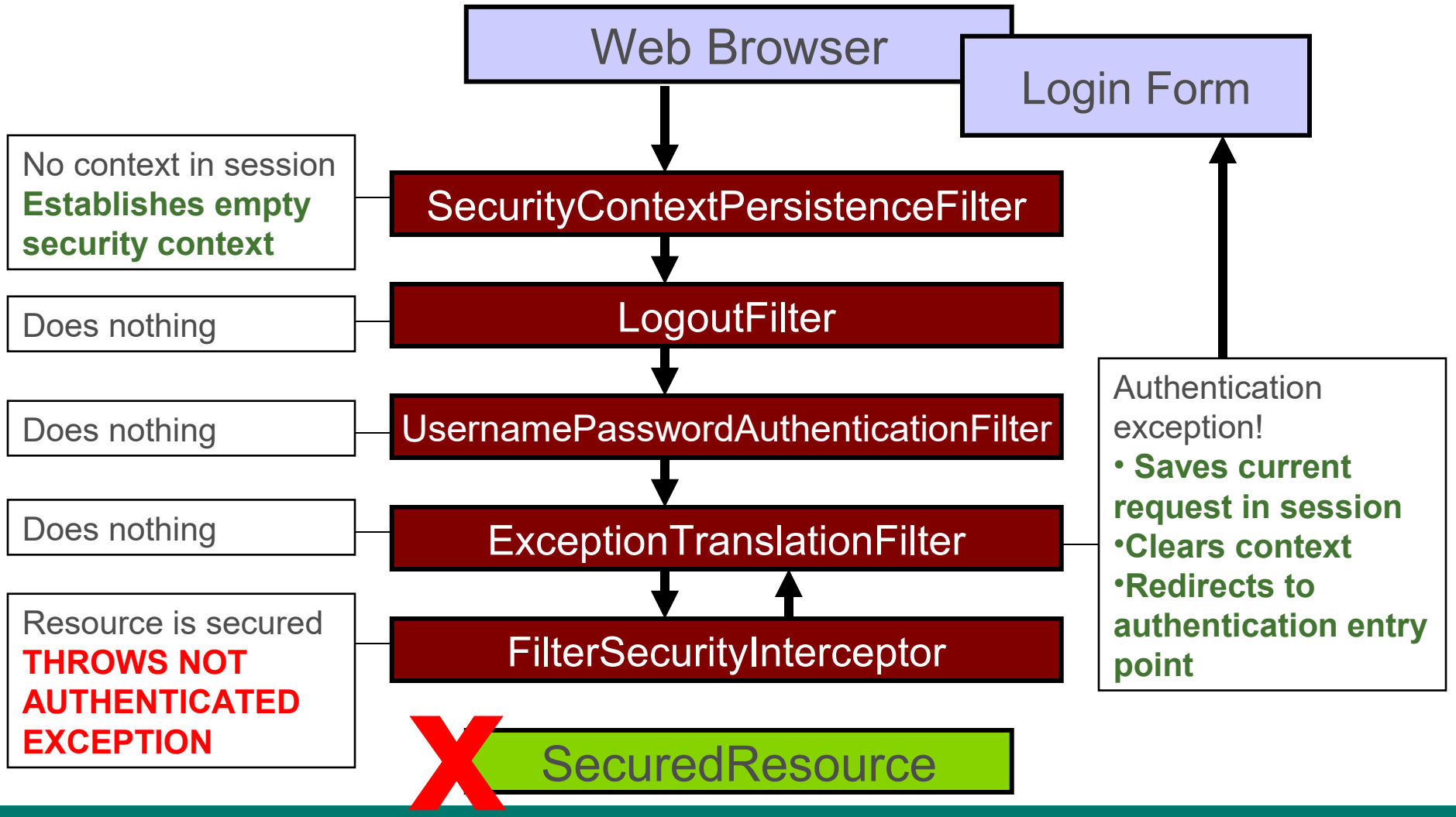
The Filter Chain

- With ACEGI Security 1.x
  - Filters were manually configured as individual <bean> elements
  - Led to verbose and error-prone XML
- Spring Security 2.x, 3.x, 4.x
  - Filters are initialized with correct values by default
  - Manual configuration is not required **unless you want to customize Spring Security's behavior**
  - It is still important to understand how they work underneath

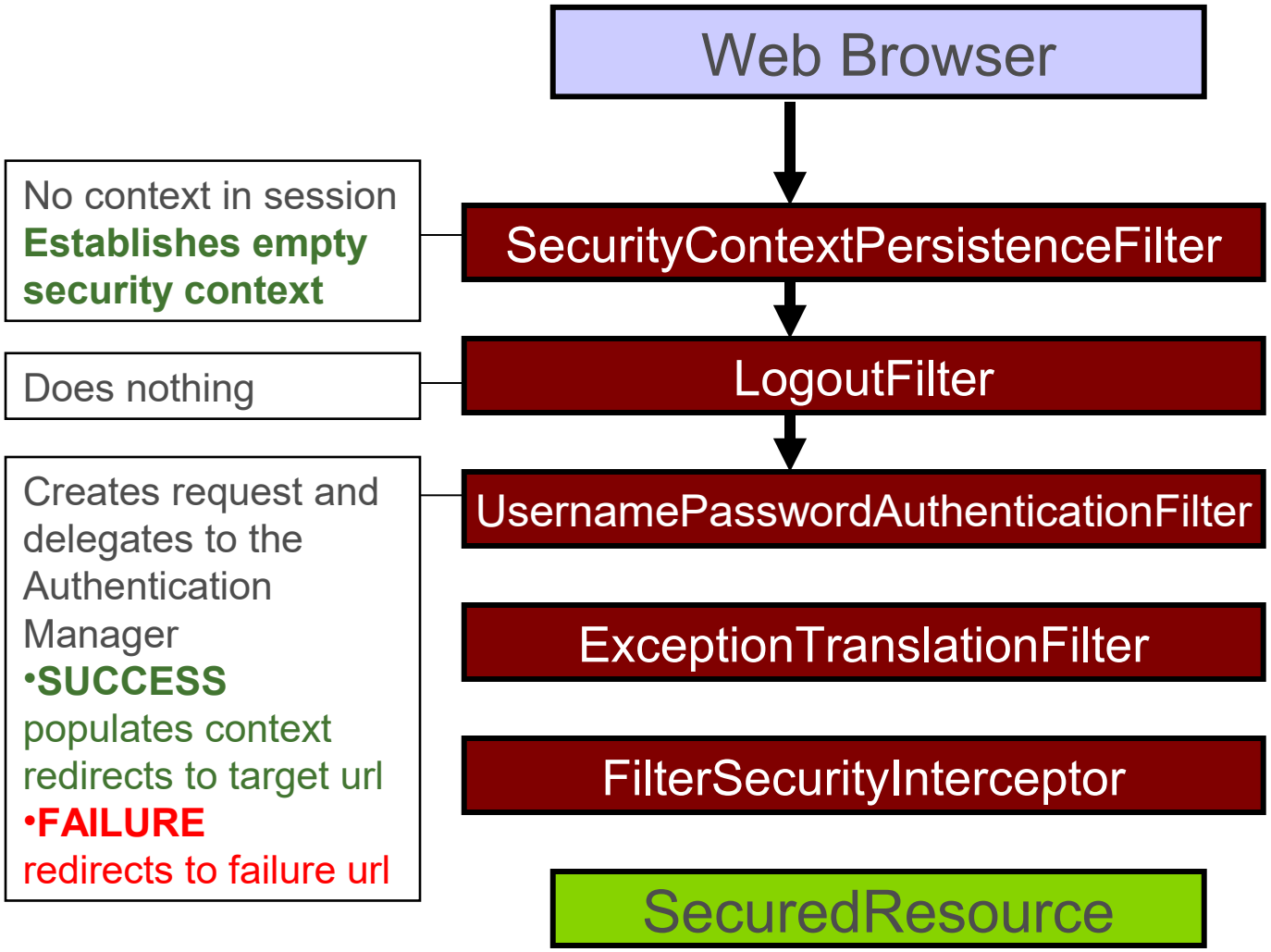
Access Unsecured Resource Prior to Login



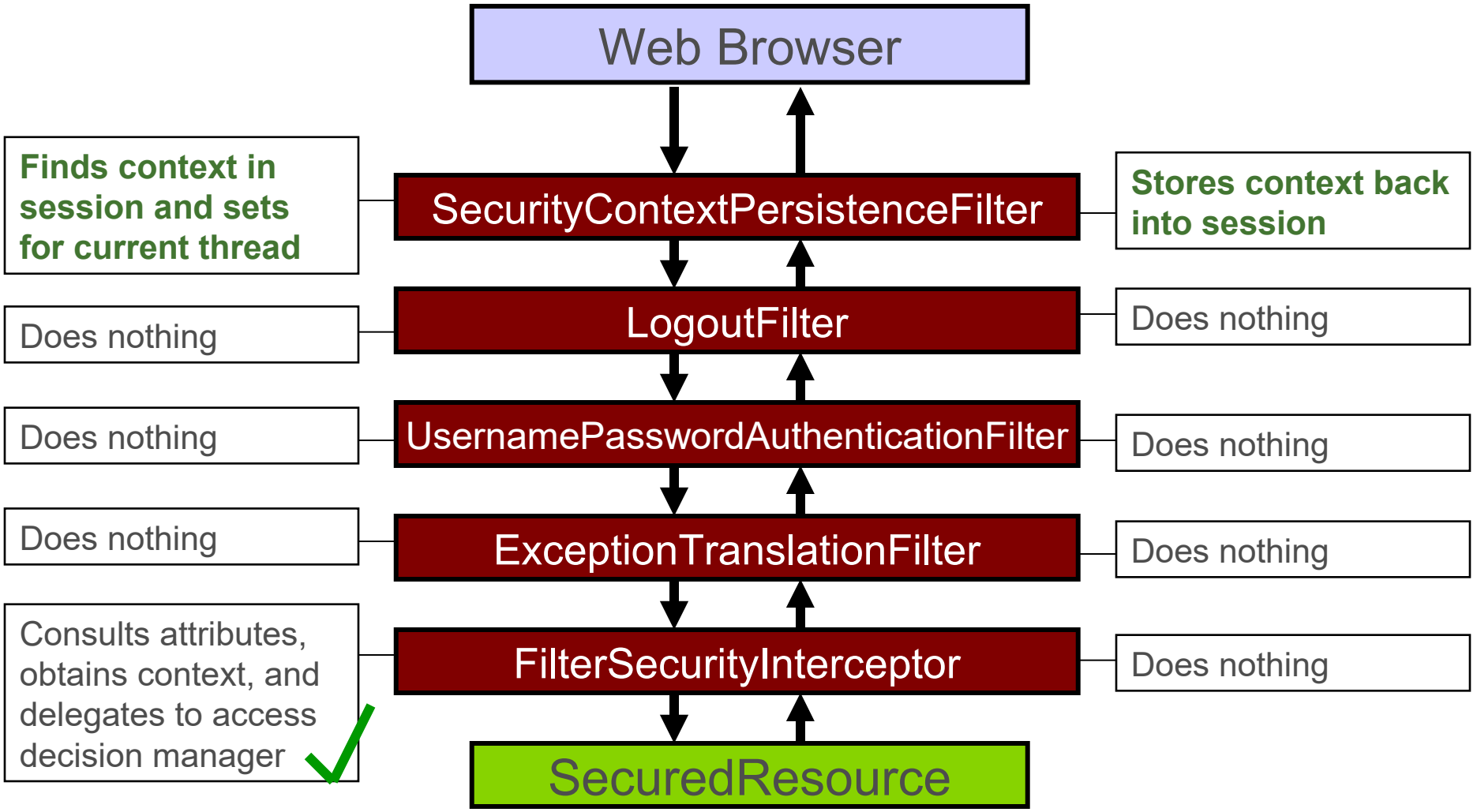
Access Secured Resource Prior to Login



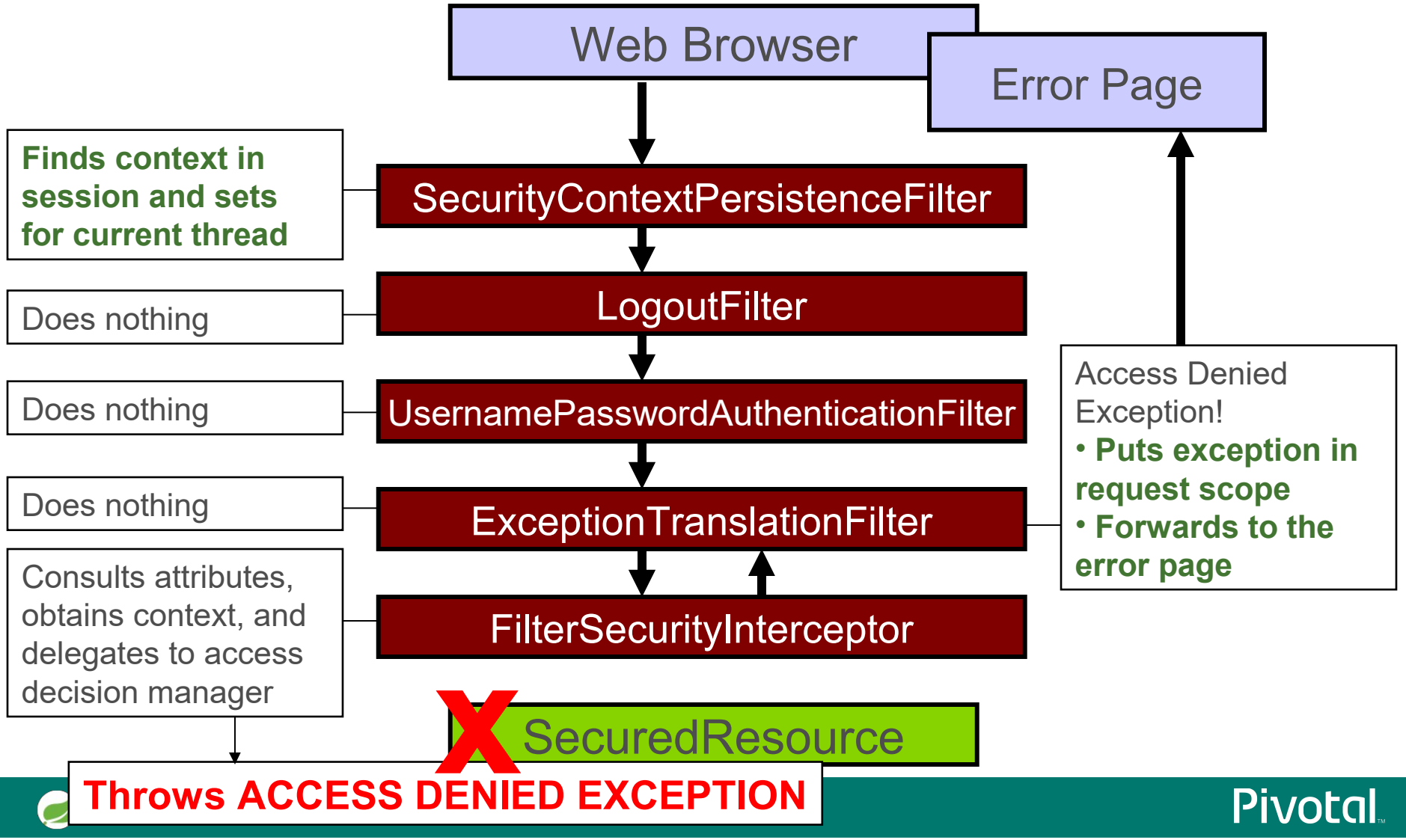
Submit Login Request



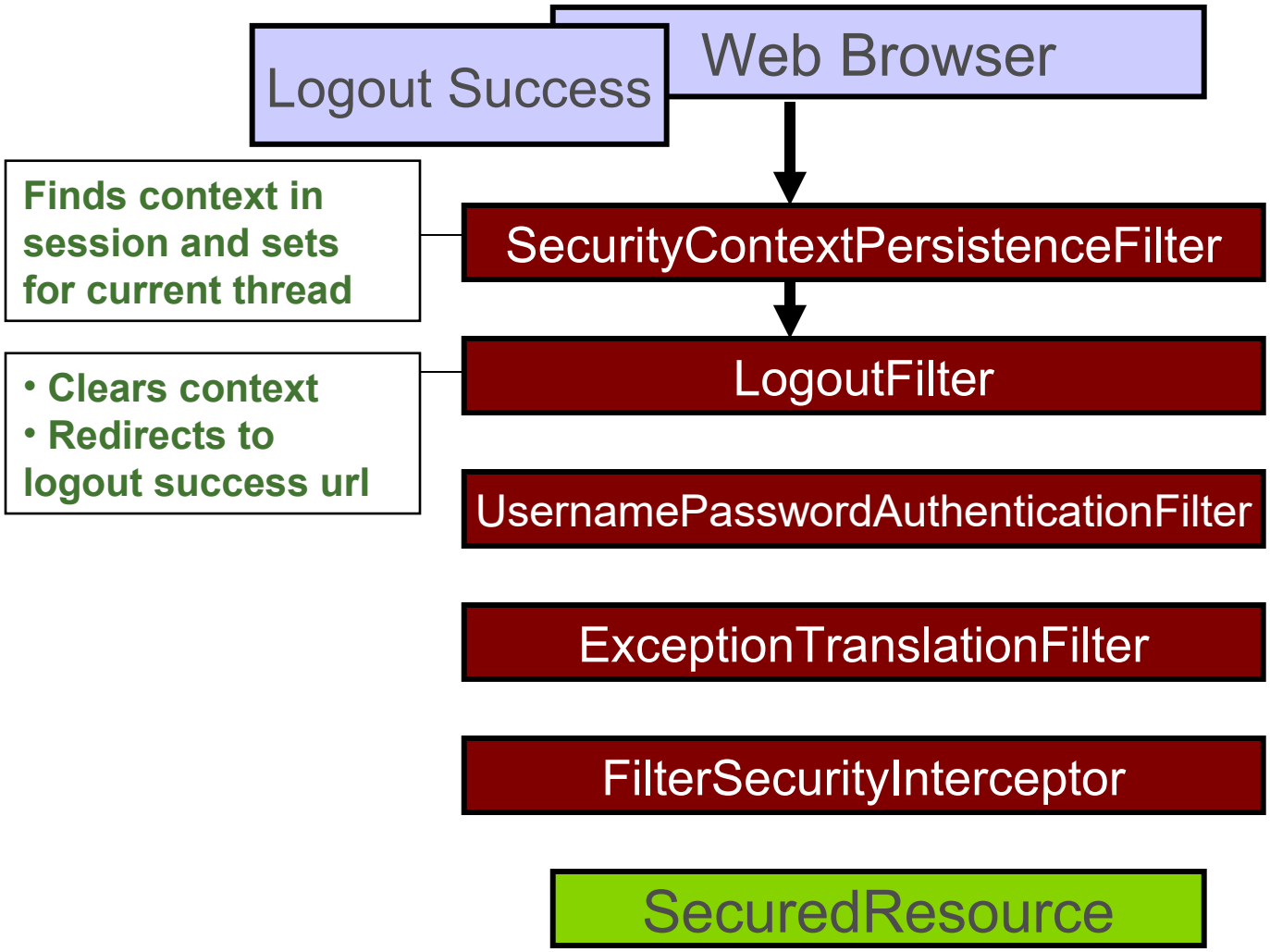
Access Resource With Required Role



Access Resource Without Required Role



Submit Logout Request



The Filter Chain: Summary

#	Filter Name	Main Purpose
1	SecurityContext IntegrationFilter	Establishes SecurityContext and maintains between HTTP requests *formerly: HttpSessionContextIntegrationFilter*
2	LogoutFilter	Clears SecurityContextHolder when logout requested
3	UsernamePassword AuthenticationFilter	Puts Authentication into the SecurityContext on login request *formerly: AuthenticationProcessingFilter*
4	Exception TranslationFilter	Converts SpringSecurity exceptions into HTTP response or redirect
5	FilterSecurity Interceptor	Authorizes web requests based on on config attributes and authorities

Custom Filter Chain – Replace Filter

- Filters can be **replaced** in the chain
  - Replace an existing filter with your own
    - Replacement must *extend* the filter being replaced

```
public class MyCustomLoginFilter
  extends UsernamePasswordAuthenticationFilter {}
```

```
@Bean
public Filter loginFilter() {
    return new MyCustomLoginFilter();
}
```

```
http.addFilter ( loginFilter() );
```

Custom Filter Chain – Add Filter

- Filters can be **added** to the chain
  - *After* any filter

```
public class MyExtraFilter extends Filter { ... }
```

```
@Bean
public Filter myExtraFilter() {
    return new MyExtraFilter();
}
```

```
http.addFilterAfter ( myExtraFilter(),
                      UsernamePasswordAuthenticationFilter.class );
```