



# Securing Web Applications

---

Addressing Common Web Application  
Security Requirements

## Objectives

---

After completing this lesson, you should be able to

- Explain basic security concepts
- Set up Spring Security in a Web environment
- Use Spring Security to configure Authentication and Authorization
- Define Method-level Security



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

<http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/>

# Agenda

- **Security Overview**
- URL Authorization
- Configuring Web Authentication
- Method Security
- Lab
- Advanced Security



# Security Concepts

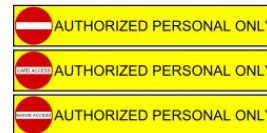
- **Principal**
  - User, device or system that performs an action
- **Authentication**
  - Establishing that a principal's credentials are valid
- **Authorization**
  - Deciding if a principal is allowed to perform an action
- **Authority**
  - Permission or credential enabling access (such as a role)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

# Authentication



- There are many authentication mechanisms
  - *Examples:* Basic, Digest, Form, X.509, OAuth
- There are many storage options for credential and authority information
  - *Examples:* in-memory (development), Database, LDAP

# 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



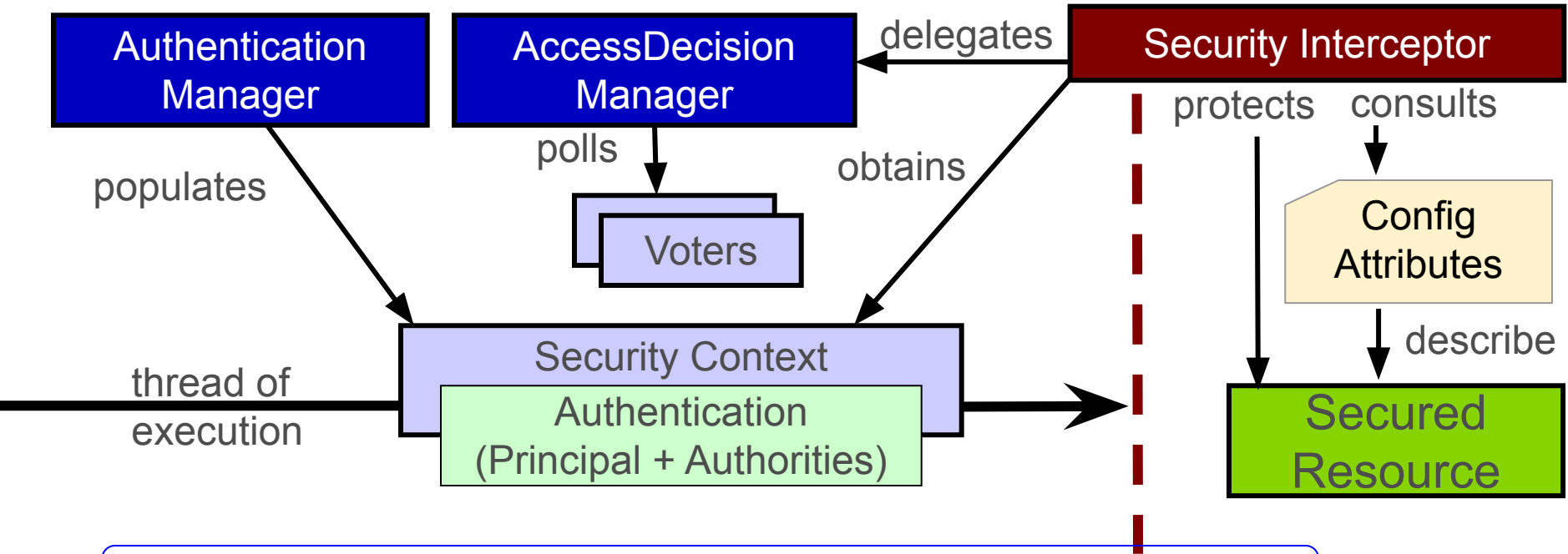
*A Role is simply a commonly used type of Authority.*

# Spring Security Project



- **Portable**
  - Secured archive (JAR, WAR, EAR) can be deployed as-is
- **Separation of Concerns**
  - Business logic is *decoupled* from security concerns
  - Authentication and Authorization are *decoupled*
    - Changes to authentication have *no impact* on authorization
- **Flexible & Extensible**
  - *Authentication*: Basic, Form, X.509, OAuth, Cookies, Single-Sign-On, ...
  - *Storage*: LDAP, RDBMS, Properties file, custom DAOs, ...
  - Highly customizable

# Spring Security – the Big Picture



<https://spring.io/guides/topicals/spring-security-architecture>



# Setup and Configuration

## Spring Security in a Web Environment



### Three steps

1. Setup Filter chain (Spring Boot does this for you)
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 – 1

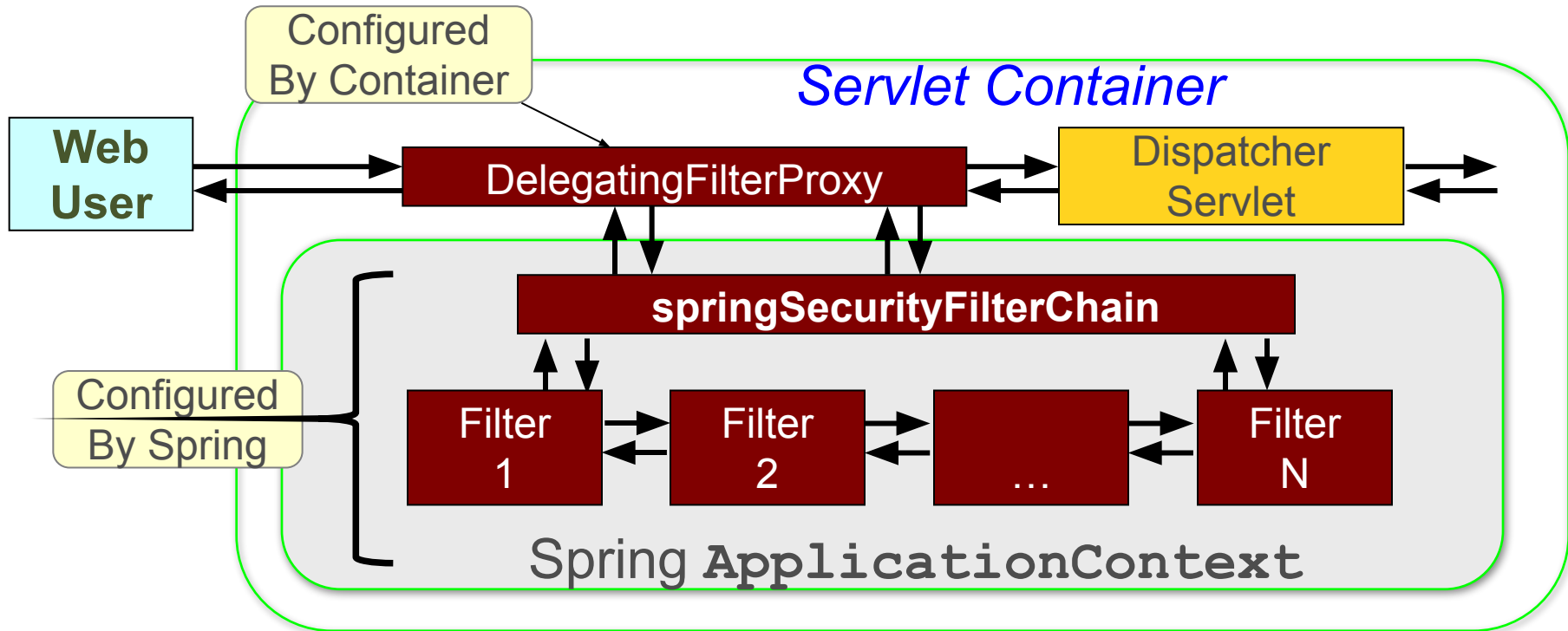


- Implementation is a *chain* of Spring configured *filters*
  - Requires a **DelegatingFilterProxy** which *must* be called *springSecurityFilterChain*
  - Chain consists of many filters (next slide)
- Setup filter chain using *one* of these options
  - Spring Boot does it automatically
  - Or subclass **AbstractSecurityWebApplicationInitializer**
  - Or declare as a **<filter>** in **web.xml**



For more details (and non-Boot examples) see “*Advanced security: working with filters*” at end of this topic.

# Spring Security Filter Chain – 2



 All implement `java.servlet.Filter`

# Spring Boot Default Security Setup



- Sets up a single in-memory user called “user”
- Auto-generates a UUID password
- Relies on Spring Security’s content-negotiation strategy to determine whether to use httpBasic or formLogin
- All URLs require a logged-in user

```
INFO : o.s.b.web.servlet.FilterRegistrationBean - Mapping filter: 'httpTraceFilter' to: [/*]  
INFO : o.s.b.web.servlet.FilterRegistrationBean - Mapping filter: 'webMvcMetricsFilter' to: [/*]  
INFO : o.s.b.w.servlet.ServletRegistrationBean - Servlet dispatcherServlet mapped to [/]  
INFO : o.s.b.a.w.s.WelcomePageHandlerMapping - Adding welcome page: class path resource [static/index.html]  
INFO : o.s.b.a.s.s.UserDetailsServiceAutoConfiguration -
```

Using generated security password: `f49a49f1-df8a-4da8-b3e8-89fb204bda24`

```
INFO : o.s.s.web.DefaultSecurityFilterChain - Creating filter chain: org.springframework.security.web.util.matcher.AnyReq  
INFO : o.s.b.d.a.OptionalLiveReloadServer - LiveReload server is running on port 35729
```

# Agenda

- Security Overview
- **URL Authorization**
- Configuring Web Authentication
- Method Security
- Lab
- Advanced Security



# Configuration in the Application Context

@Configuration

@EnableWebSecurity

**public class** SecurityConfig **extends** WebSecurityConfigurerAdapter {

@Override

**protected void** configure(HttpSecurity http) **throws** Exception {

}

@Autowired

**public void** configureGlobal(AuthenticationManagerBuilder auth)  
**throws** Exception {

}

}

Extend `WebSecurityConfigurerAdapter`

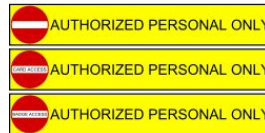
Optional for Spring Boot applications

Web-specific security settings

Global security settings  
(authentication manager, ...)

**Note: @Autowired**

# Authorizing URLs



- Define specific authorization restrictions for URLs
- Support “*Ant-style*” pattern matching
  - “`/admin/`” only matches “`/admin/xxx`”
  - “`/admin/**`” matches *any* path under `/admin`
    - Such as “`/admin/database/access-control`”

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .mvcMatchers("/admin/**").hasRole("ADMIN")  
        ...  
}
```

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

User must have  
**ADMIN** role

# More on authorizeRequests ()

- *Chain* multiple restrictions - evaluated in the order listed
  - First match is used, *put specific matches first*

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
            .mvcMatchers("/signup", "/about").permitAll()  
            .mvcMatchers("/accounts/edit*").hasRole("ADMIN")  
            .mvcMatchers("/accounts/**").hasAnyRole("USER", "ADMIN")  
            .anyRequest().authenticated();
```

Must be  
authenticated for  
any other request



Spring Security supports *roles* out-of-the-box – but *there are no predefined roles*.



# Warning: URL Matching



- Older code may use **antMatchers**

```
http.authorizeRequests()  
    // Only matches /admin  
    .antMatchers("/admin").hasRole("ADMIN")  
    // Matches /admin, /admin/, /admin.html, /admin.xxx  
    .mvcMatchers("/admin").hasRole("ADMIN")
```

They look identical  
– but are **not**

- Use **mvcMatchers**
  - Uses same matching rules as `@RequestMapping`
  - Newer API, less error-prone, *recommended*





# By-passing Security

- Some URLs need not be secure (such as static resources)
  - `permitAll()` allows open-access
    - But still processed by Spring Security
- Can by-pass Security completely

*Different `configure()` method than earlier*

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(WebSecurity web) throws Exception {
        web.ignoring().mvcMatchers("/css/**", "/images/**", "/javascript/**");
    }
}
```

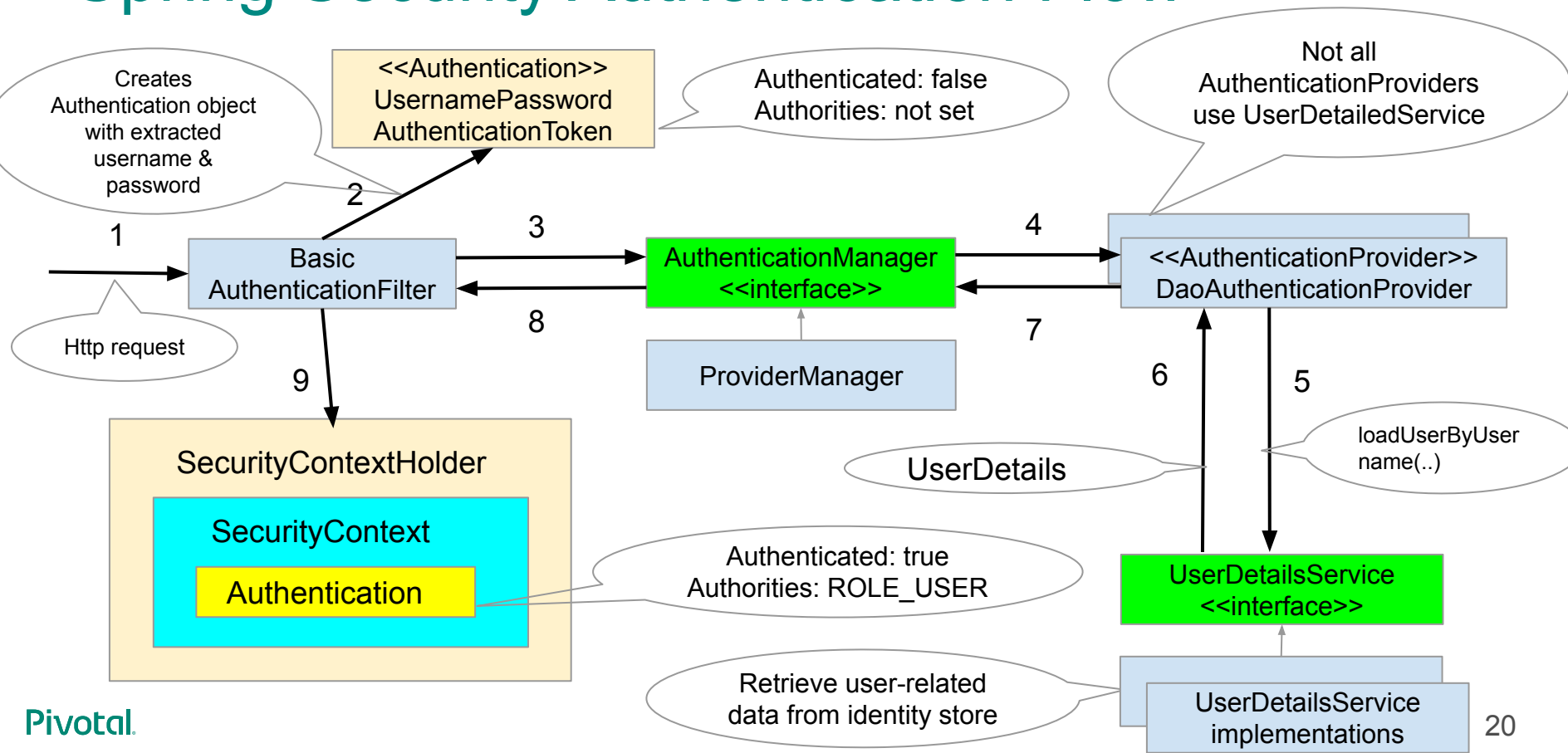
These URLs pass straight through, no checks

# Agenda

- Security Overview
- URL Authorization
- **Configuring Web Authentication**
- Method Security
- Lab
- Advanced Security



# Spring Security Authentication Flow



# In-Memory Authentication Manager



- Example of a built-in `UserService`
  - `configureGlobal()` – security for *whole* application
  - Can be shared by web and method security

**@Autowired**

```
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
    PasswordEncoder passwordEncoder =  
        PasswordEncoderFactories.createDelegatingPasswordEncoder();  
    auth  
        .inMemoryAuthentication()  
        .withUser("thor").password(passwordEncoder.encode("hammer")).roles("SUPPORT").and()  
        .withUser("loki").password(passwordEncoder.encode("trouble")).roles("USER").and()  
        .withUser("odin").password(passwordEncoder.encode("king")).roles("ADMIN");  
}
```

Not web-specific

Returns a *UserDetailsManagerConfigurer*

login

password

Supported roles

# Sourcing Users from a Database – 1

```
private DataSource dataSource;
```

```
@Autowired
```

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

```
@Autowired
```

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

Can customize queries using methods:

- usersByUsernameQuery(<custom-query>)
- authoritiesByUsernameQuery(<custom-query>)
- groupAuthoritiesByUsername(<custom-query>)

# 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

# Password Encoding – 1

**Note:** *sha* and *md5*  
only suitable for testing  
– too insecure

- Can encode passwords using a *one-way* 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...passwordEncoder(new BCryptPasswordEncoder(12));
```

Encoding using  
'strength' 12

- Store *only* encrypted passwords

```
auth.inMemoryAuthentication().withUser("hughie")  
    .password("$2a$10$aMxNkanIJ...IEuylt87PNlicYpl1y.IG0C.")  
    .roles("GENERAL")
```

# Challenges of Password Encoding Schemes

- Should be future-proof
  - Encoding schemes that are considered secure today will not provide the same level of security in the future
  - New encoding schemes will emerge in the future
- Should accommodate old password formats
  - Old format passwords should be able to be used with no/minimum effort
- Should allow usage of multiple password formats
  - Old and new format passwords should be able to co-exist

Spring Security framework should address these challenges.

# DelegatingPasswordEncoder to the Rescue

- Introduced in Spring Security 5 (and Spring Boot 2)
- Uses new password storage format: *{id}encodedPassword*
  - {id} represents a logical name of a specific encoder
- Delegates to another PasswordEncoder based upon a prefixed id
- Uses BCrypt as a default “best practice” encoding scheme for now

## @Autowired

```
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
    PasswordEncoder passwordEncoder =  
        PasswordEncoderFactories.createDelegatingPasswordEncoder();  
    auth  
        .inMemoryAuthentication()  
        .withUser("thor").password(passwordEncoder.encode("hammer")).roles("SUPPORT");  
}
```

Generates {bcrypt}\$2a\$10\$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1t1Ry.fqvM/BG

# Enabling HTTP Authentication - 1

- Use the `HttpSecurity` object again
  - *Example: HTTP Basic*


```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
            .mvcMatchers("/admin/**").hasRole("ADMIN")  
            .mvcMatchers("/accounts/**").hasAnyRole("USER","ADMIN")  
        .and()  
        .httpBasic();           // Enable HTTP Basic  
}
```

*Browser will prompt for username & password*

# Enabling HTTP Authentication - 2

*Form based  
login*

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
        .mvcMatchers("/admin/**").hasRole("ADMIN")...  
        .and()                                // method chaining!  
  
        .formLogin()                          // setup form-based authentication  
        .loginPage("/login")                 // 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  
}
```



Default: /login?logout

# An Example Login Page

URL that indicates an authentication request.

*Default:* POST to same URL used to display the form.

```
<form action="/login" method="POST">
  <input type="text" name="username"/>
  <br/>
  <input type="password" name="password"/>
  <br/>
  <input type="submit" name="submit" value="LOGIN"/>
</form>
```

The expected keys  
for generation of an  
authentication  
request token

*login.html*

# Other Authentication Options

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

Authorization is *not* affected by changes to Authentication!

# Agenda

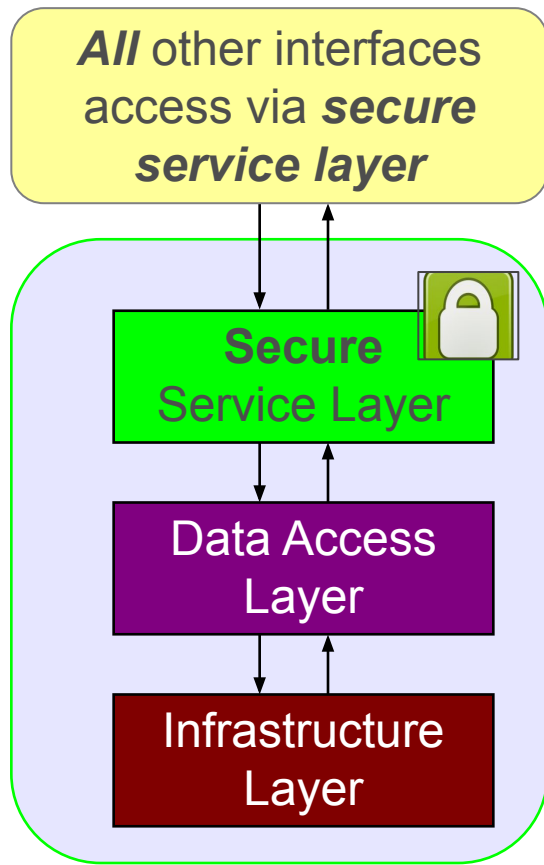
- Security Overview
- URL Authorization
- Configuring Web Authentication
- **Method Security**
- Lab
- Advanced Security





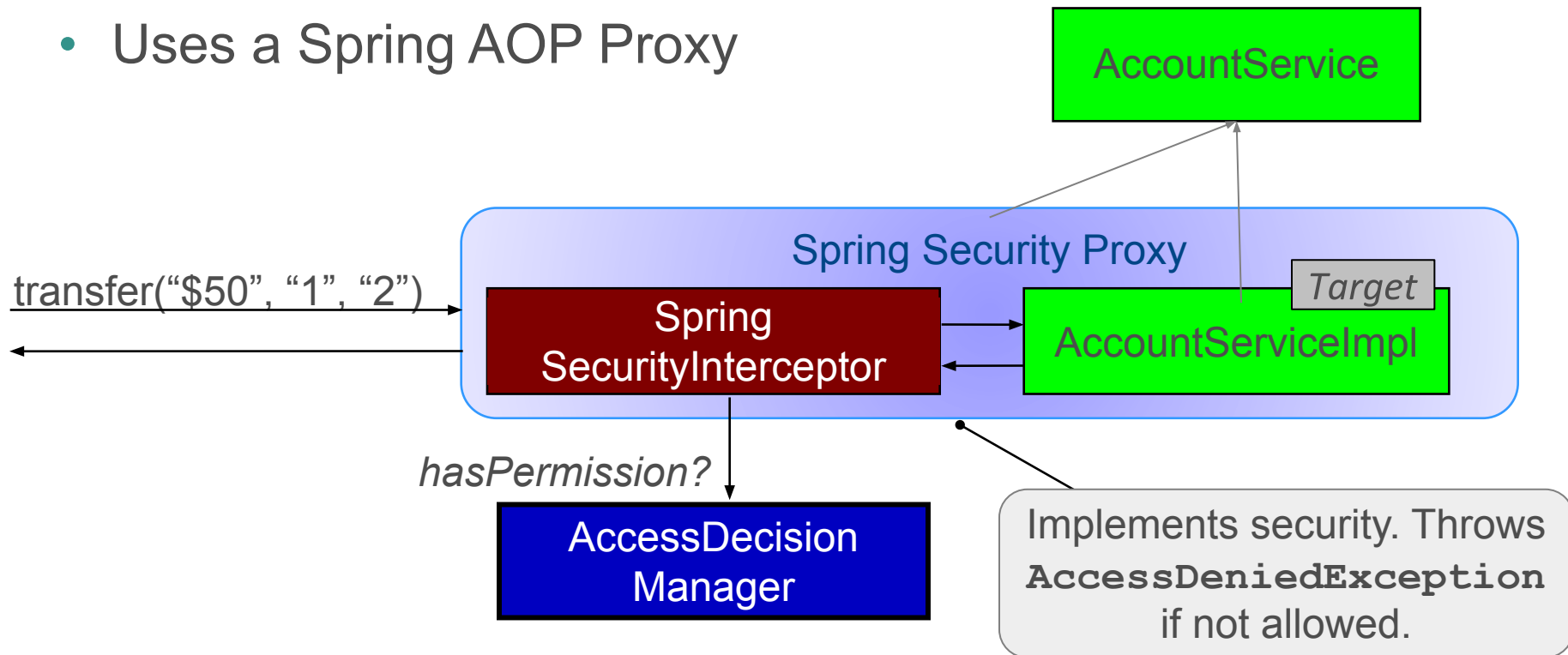
# Method Security

- Spring Security uses AOP for method-level security
  - Annotations: either Spring's own or JSR-250
- Recommendation:
  - Secure your services
  - Do *not* access other layers directly
    - Bypasses security (and probably transactions) on your service layer



# Method Security – How it Works

- Uses a Spring AOP Proxy



# Method Security - JSR-250

JSR-250 annotations  
must be enabled

- Only supports **role-based** security (hence the name)

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

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

```
public class ItemManager {  
    @RolesAllowed("ROLE_MEMBER")  
    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 you *must* use full name

# Method Security with SpEL

- Use Pre/Post annotations for SpEL

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

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

public class ItemManager {
    // Members may only find their own order items
    @PreAuthorize("hasRole('MEMBER') && " +
        "#order.owner.id == principal.user.id")
    public Item findItem(Order order, 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

A man with a beard and a woman are sitting at a desk, looking at a computer monitor. The man is pointing at the screen. The background is a blurred office environment.

---

## ***Lab: Securing a RESTful application***

---

**Lab project:**  
**40-security**

**Anticipated Lab time:**  
**20 Minutes**

**Optional Topics:** Filter Details, Configuration Choices, Legacy Apps

# Agenda

- Security Overview
- URL Authorization
- Configuring Web Authentication
- Method Security
- Lab
- **Advanced Security**
  - **Working with Filters**
  - Configuration Choices
  - Legacy Applications

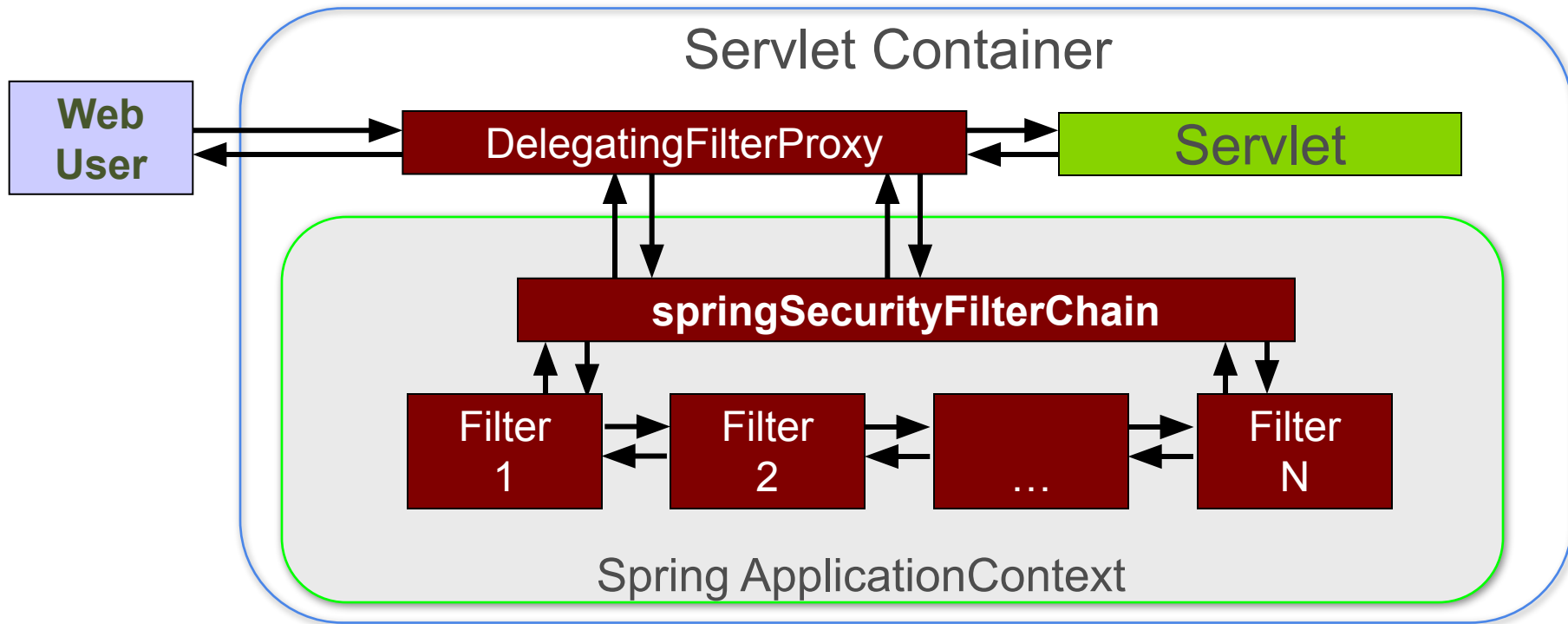


# Spring Security in a Web Environment

- *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



# Web Security Filter Configuration



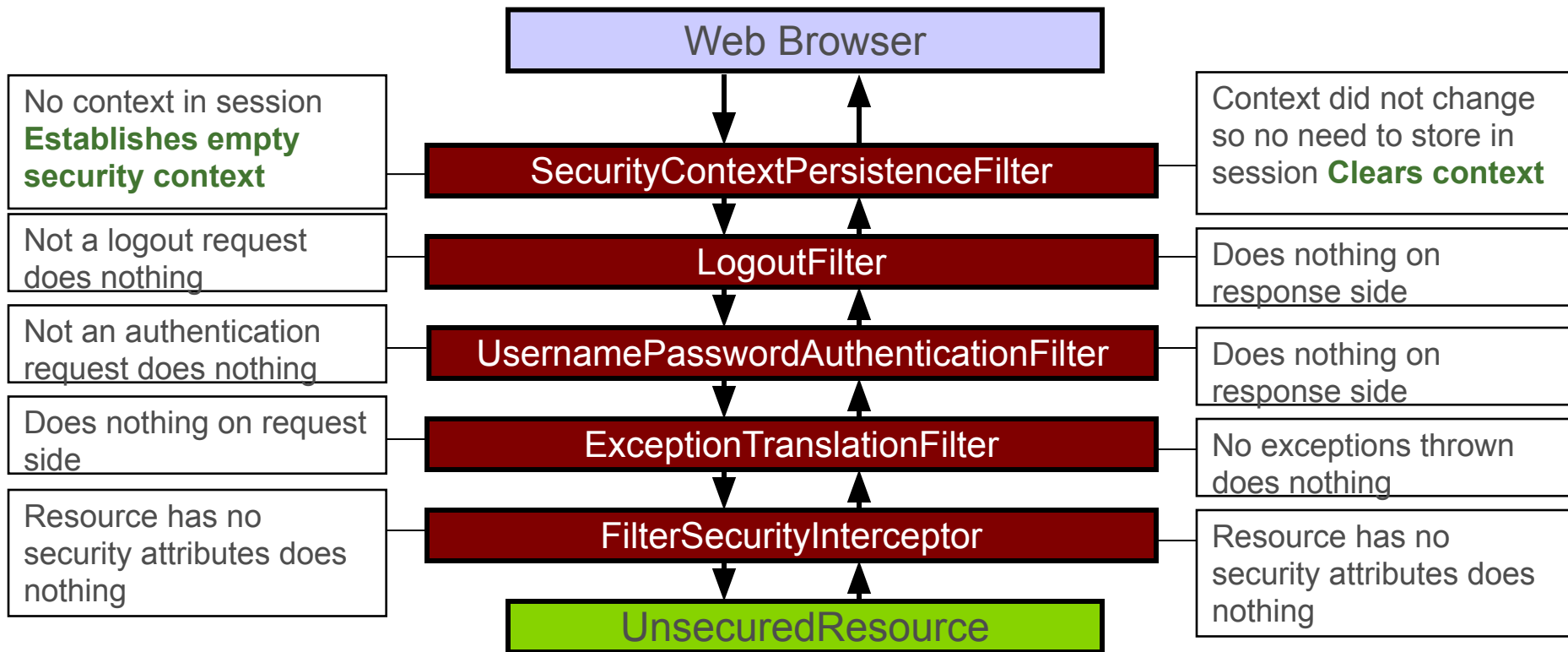
# The Filter Chain

- Spring Security uses a chain of many, many filters
  - Filters 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

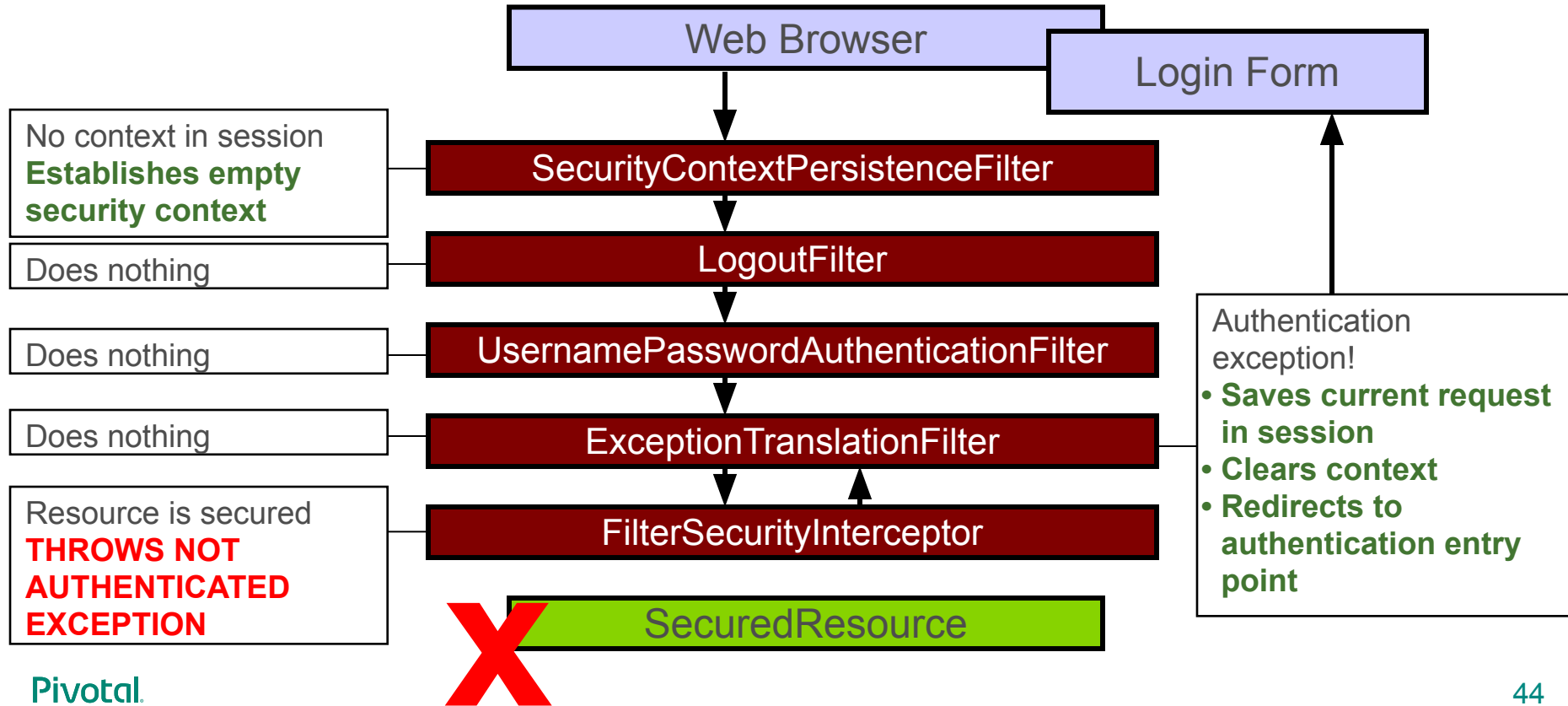


Spring Security originally developed independently of Spring – called *ACEGI Security* and involved far more manual configuration

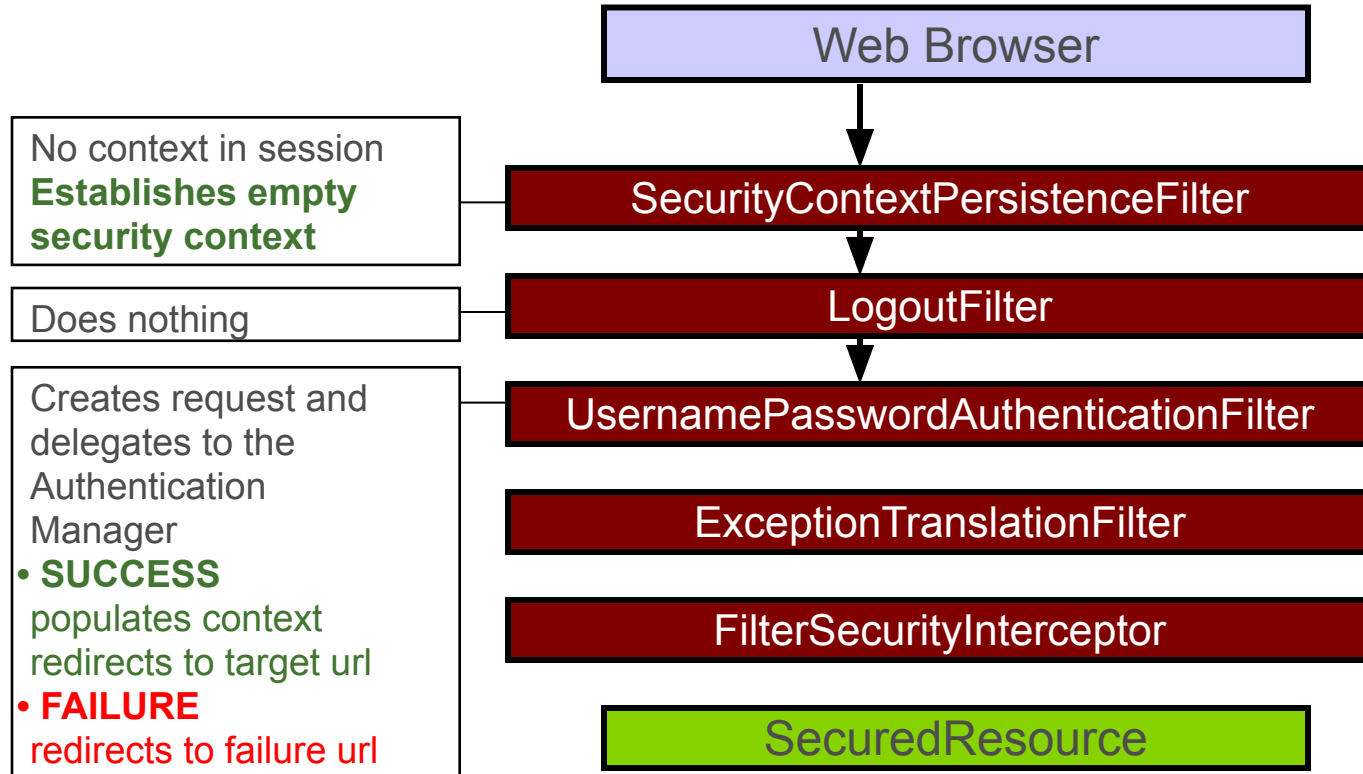
# Access Unsecured Resource Prior to Login



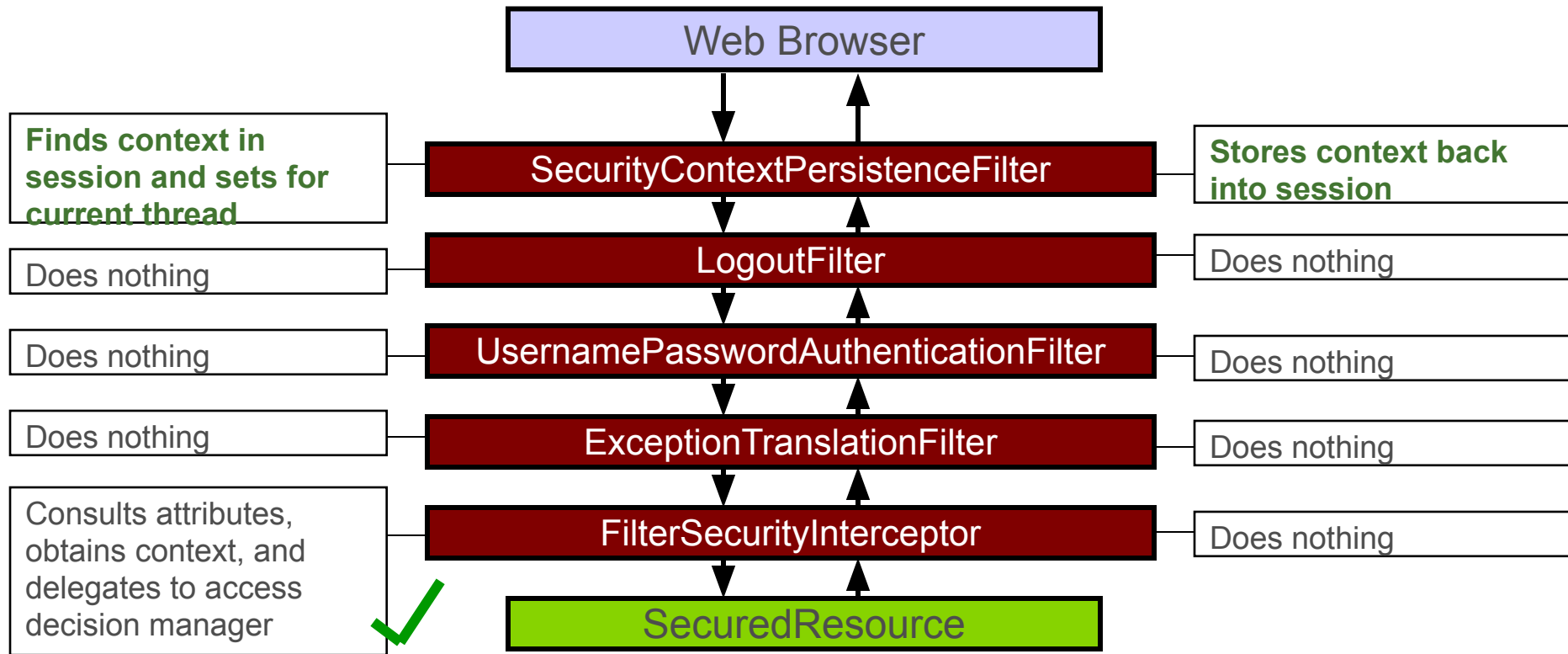
# Access Secured Resource Prior to Login



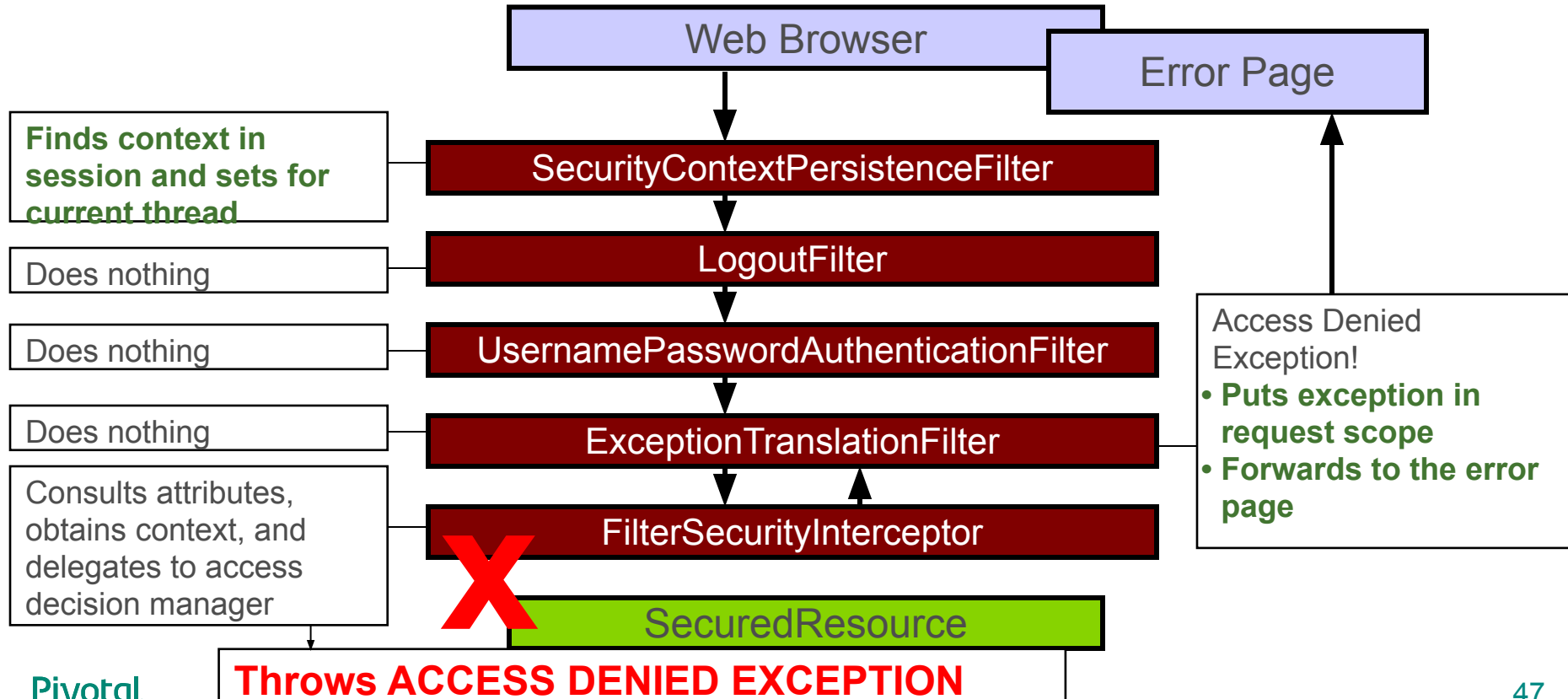
# Submit Login Request



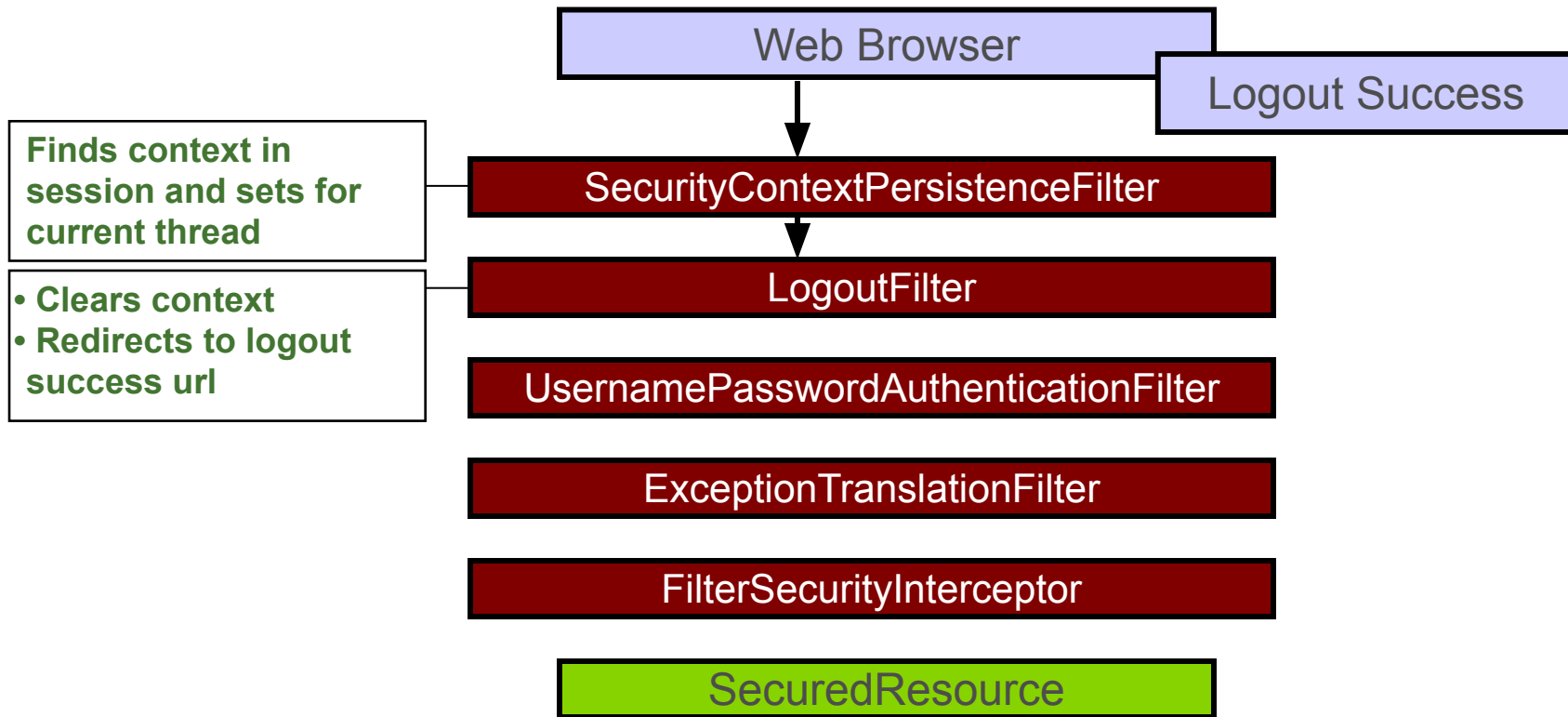
# Access Resource With Required Role



# 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
2	LogoutFilter	Clears SecurityContextHolder when logout requested
3	UsernamePassword AuthenticationFilter	Puts Authentication into the SecurityContext on login request.
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 implements Filter { ... }
```

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

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

# Agenda

- Security Overview
- URL Authorization
- Configuring Web Authentication
- Method Security
- Lab
- **Advanced Security**
  - Working with Filters
  - **Configuration Choices**
  - Legacy Applications



# Configuration Choices

1. Add an autowired method to your security configuration
  - As shown in these slides: **`configureGlobal(...)`**
2. Override **`WebSecurityConfigurerAdapter`**'s **`configure(AuthenticationManagerBuilder auth)`**
  - Defines users/roles for web-configuration *only*,
  - Users *would not* be recognized by method security
3. Extend **`GlobalAuthenticationConfigurerAdapter`**
  - Equivalent to option 1, more control
  - Can setup *multiple* authentication schemes

# @Profile with Security Configuration

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

@Configuration

@EnableWebSecurity

@Profile("development")



*Use in-memory provider*

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

# @Profile with Security Configuration

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

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

*Use database provider*

Use this profile when "development" *not* defined



# Agenda

- Security Overview
- URL Authorization
- Configuring Web Authentication
- Method Security
- Lab
- **Advanced Security**
  - Working with Filters
  - Configuration Choices
  - **Legacy Applications**





# Configuration without Spring Boot

## Servlet 2 using *web.xml*

- Define the DelegatingFilterProxy

This name is mandatory - delegates to a Spring bean with *same* name

```
<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.xml*

# Configuration without Spring Boot

## *Servlet 3 `WebApplicationInitializer`*

- Declare your own subclass of **`AbstractSecurityWebApplicationInitializer`**
  - Sets up the **`DelegatingFilterProxy`**
  - Automatically called by Spring because it implements **`WebApplicationInitializer`**

```
import org.springframework.security.web.  
    context.AbstractSecurityWebApplicationInitializer;  
  
public class SecurityWebApplicationInitializer  
    extends AbstractSecurityWebApplicationInitializer {  
}
```

Class is meant to be empty – nothing else is required

# Method Security - @Secured

You may see this in older applications

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

Annotation must be enabled

```
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. But SpEL **not** supported.*