Securing REST Application

Addressing Common Security Requirements

1.18.5

Objectives

After completing this lesson, you should be able to do the following

- 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/

Agenda

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

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 access a resource
- **Authority**
  - Permission or credential enabling access (such as a role)
- **Secured Resource**
  - Resource that is being secured

Authentication

- There are many authentication mechanisms
  - *Examples:* Basic, Digest, Form, X.509, OAuth 2.0 / OIDC
- There are many storage options for credential and authority data
  - *Examples:* in-memory (for development only), Database, LDAP

Authorization

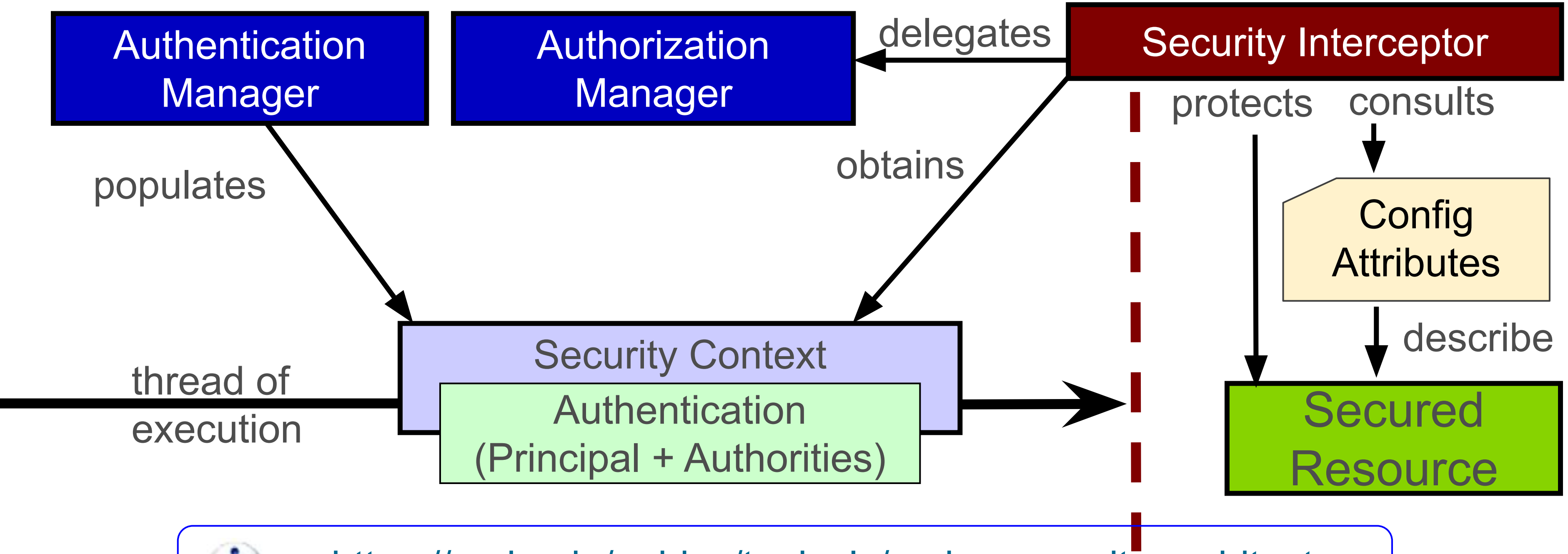
- Authorization depends on authentication
  - Before deciding if a user is permitted to access a resource, user identity must be established
- Authorization determines if you have the required *Authority*
- The decision process is often based on roles
  - *ADMIN* role can cancel orders
  - *MEMBER* role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

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

Spring Security

- Portable
  - Can be used on any Spring project
- Separation of Concerns
  - Business logic is *decoupled* from security concern
  - 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

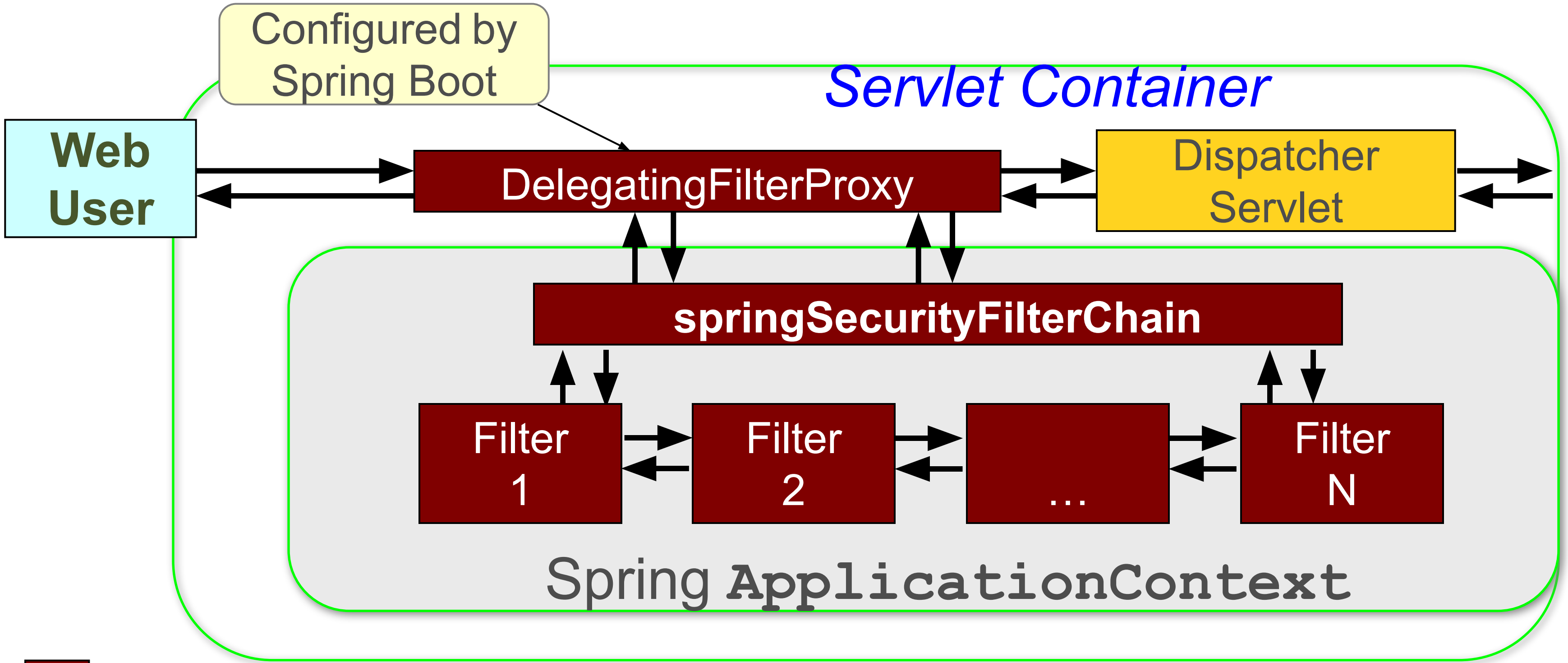
Setup and Configuration

Spring Security in a Web Environment

Three steps

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

Spring Security Filter Chain

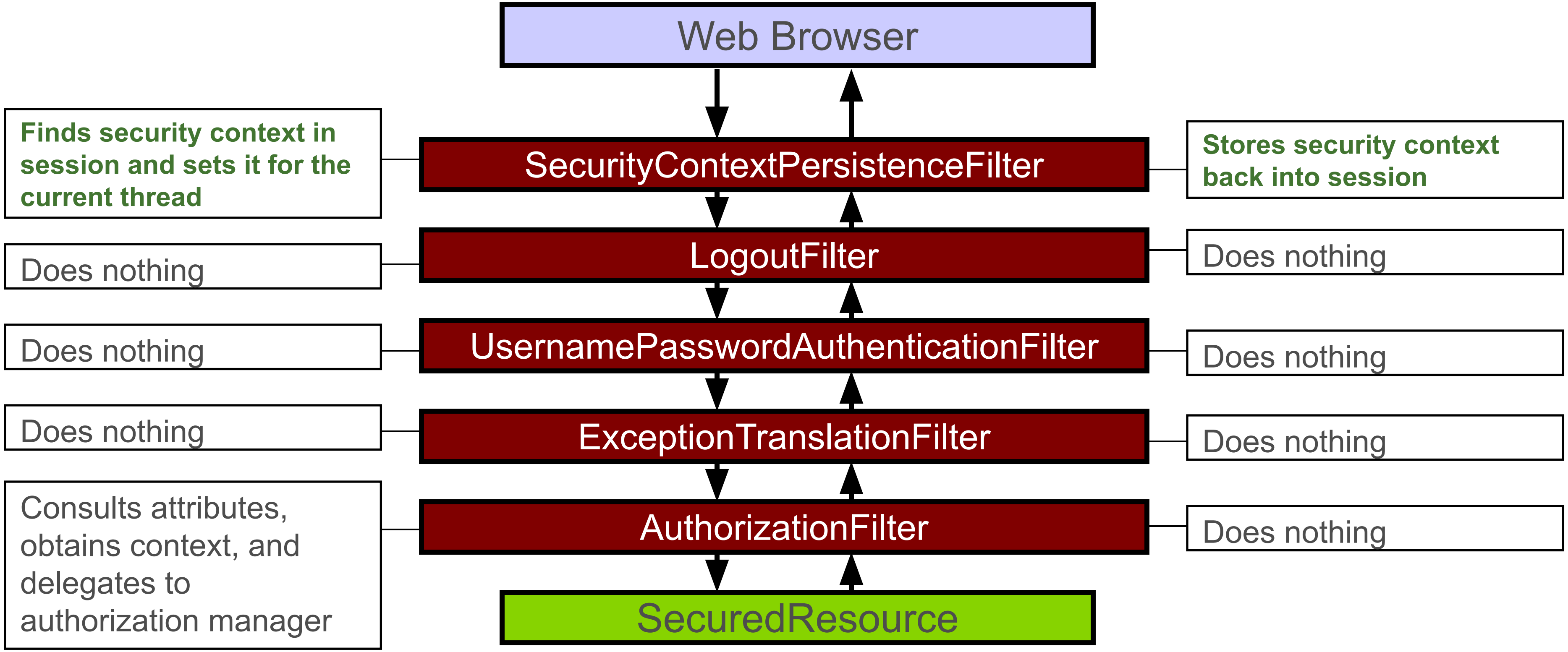


🟥 *All* implement `javax.servlet.Filter`

Spring Security Filters

#	Filter Name	Main Purpose
1	`SecurityContext PersistenceFilter`	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	`AuthorizationFilter`	Authorizes web requests based on config attributes and authorities

Example Filter: SecurityContextPersistenceFilter


Web Browser
Finds security context in session and sets it for the current thread
SecurityContextPersistenceFilter
Stores security context back into session
Does nothing
LogoutFilter
Does nothing
Does nothing
UsernamePasswordAuthenticationFilter
Does nothing
Does nothing
ExceptionTranslationFilter
Does nothing
Consults attributes, obtains context, and delegates to authorization manager
AuthorizationFilter
Does nothing
SecuredResource

Adding the Security dependency

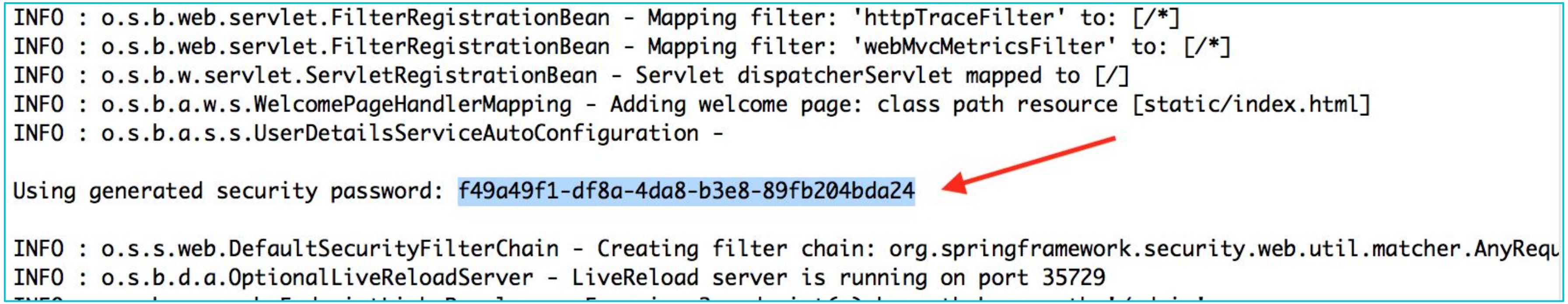
```xml
<dependencies>

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

</dependencies>
```

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.AnyRequ
INFO : o.s.b.d.a.OptionalLiveReloadServer - LiveReload server is running on port 35729
```

Agenda

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

Spring Security Configuration

```
@Configuration
public class SecurityConfig {

  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

  }

  @Bean
  public InMemoryUserDetailsService userDetailsService() {

  }
}
```

Configures the filter chain

Configures the AuthenticationManager

Authorizing URLs

- Define specific authorization restrictions for URLs
- Uses the Spring MVC matching rules if available, otherwise uses "*Ant-style*" pattern matching
  - `"/admin/*"` only matches `"/admin/xxx"`
  - `"/admin/**"` matches *any* path under `/admin`

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests((authz) -> authz
            .requestMatchers("/admin/**").hasRole("ADMIN")
            …
}
```

Match *all* URLs starting with **/admin**

User must have **ADMIN** role

More on `authorizeRequests()`

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

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Excepti
    http.authorizeHttpRequests((authz) -> authz
        .requestMatchers("/signup", "/about").permitAll()
        .requestMatchers(HttpMethod.PUT, "/accounts/edit*").hasRole("ADMIN")
        .requestMatchers("/accounts/**").hasAnyRole("USER","ADMIN")
        .anyRequest().authenticated());
    return http.build();
}
```

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 / mvcMatchers`**

```
http.authorizeHttpRequests((authz) -> authz
    // Only matches /admin
    .antMatchers("/admin").hasRole("ADMIN")
    // Matches /admin, /admin/
    .mvcMatchers("/admin").hasRole("ADMIN"))
```

These matchers are deprecated in Spring Security 5.8

- Use **`requestMatchers`**
  - Uses the most appropriate `RequestMatcher`
  - Newer API, more secure defaults, *recommended*

By-passing Security

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

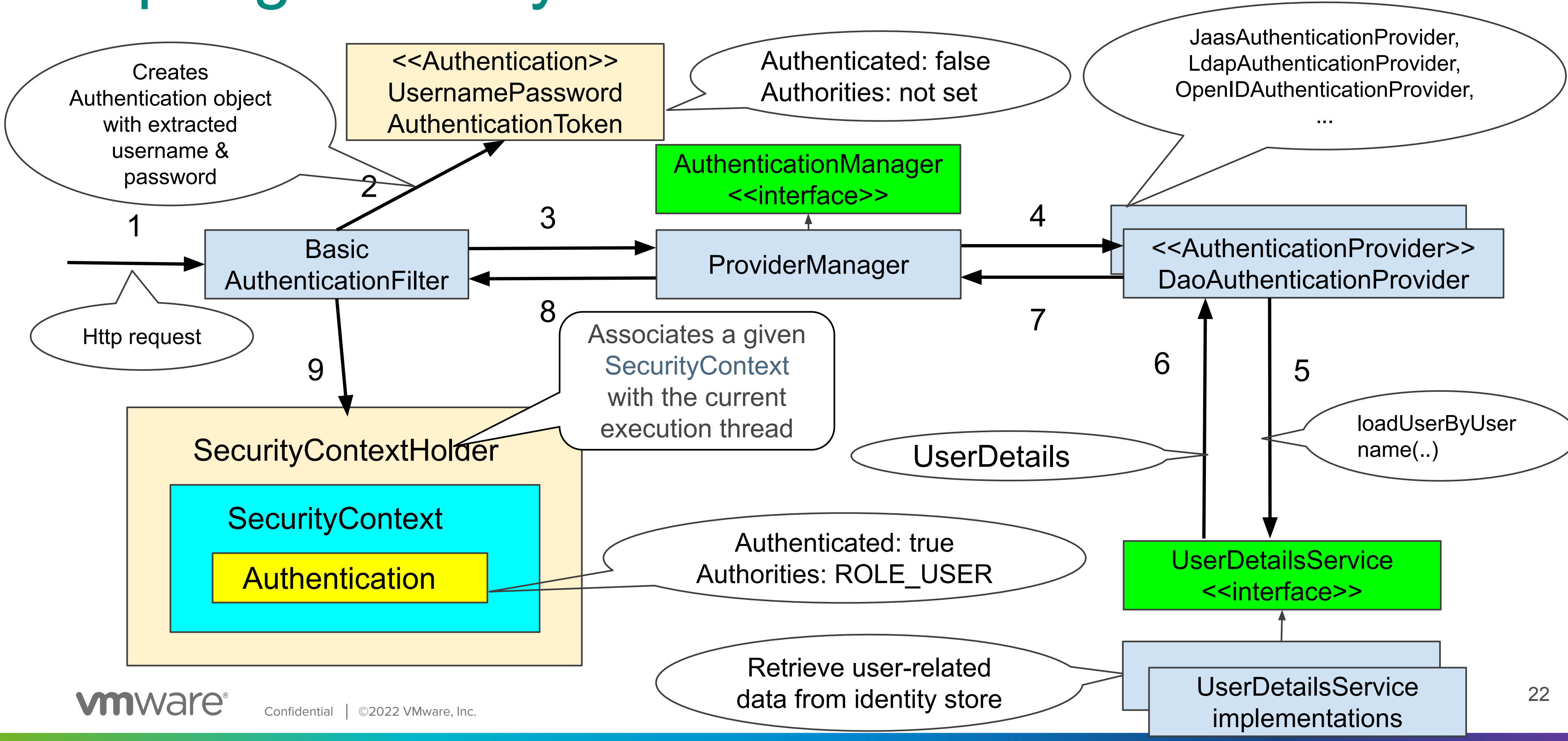
```
@Bean
public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> web.ignoring().requestMatchers("/ignore1", "/ignore2");
}
```

These URLs pass straight through, no checks

Agenda

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

Spring Security Authentication Flow



Creates Authentication object with extracted username & password
<<Authentication>> UsernamePassword AuthenticationToken
Authenticated: false
Authorities: not set
JaasAuthenticationProvider,
LdapAuthenticationProvider,
OpenIDAuthenticationProvider,
...
AuthenticationManager <<interface>>
1
2
3
4
Basic AuthenticationFilter
ProviderManager
<<AuthenticationProvider>> DaoAuthenticationProvider
Http request
8
7
6
5
9
Associates a given SecurityContext with the current execution thread
UserDetails
loadUserByUsername(..)
SecurityContextHolder
SecurityContext
Authentication
Authenticated: true
Authorities: ROLE_USER
UserDetailsService <<interface>>
Retrieve user-related data from identity store
UserDetailsService implementations

AuthenticationProvider & UserDetailsService

- Out-of-the-box **`AuthenticationProvider`** implementations
  - **`DaoAuthenticationProvider, LdapAuthenticatonProvider, OpenIDAuthenticationProvider, RememberMeAuthenticationProvider, etc.`**
- **`DaoAuthenticationProvider`** retrieves user details from a configured **`UserDetailsService`**
- Out-of-the-box **`UserDetailsService`** implementations
  - **`InMemoryUserDetailsManager`** uses in-memory identity store
  - **`JdbcUserDetailsManager`** uses database identity store
  - **`LdapUserDetailsManager`** uses Ldap identity store

In-Memory UserDetailsService

- Example of a built-in `UserDetailsService`
  - `InMemoryUserDetailsManager` implements `UserDetailsService` interface & `UserDetailsManager` interface

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {

  UserDetails user =
     User.withUsername("user").password(passwordEncoder.encode("user")).roles("USER").build();

  UserDetails admin =
     User.withUsername("admin").password(passwordEncoder.encode("admin")).roles("ADMIN").build();

  return new InMemoryUserDetailsManager(user, admin);
}
```

login

password

Supported roles

Database UserDetailsService – 1

- Another example of a built-in **`UserDetailsService`**
  - **`JdbcUserDetailsManager`** extends **`JdbcDaoImpl`** which implements the **`UserDetailsManager`** interface

```
@Bean
public UserDetailsManager userDetailsManager(DataSource dataSource) {
   return new JdbcUserDetailsManager(dataSource);
}
```

Sets up JdbcUserDetailsManager as UserDetailsService

Database UserDetailsService – 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

Implementing custom authentication

- Option #1: Implement custom **`UserDetailsService`** (using pre-configured **`DaoAuthenticationProvider`**)

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

- Option #2: Implement custom **`AuthenticationProvider`**

```
protected interface AuthenticationProvider {
        Authentication authenticate(Authentication authentication) throws AuthenticationException;
        boolean supports(Class<?> authentication);
}
```

Password Encoding

- Password must be stored in an encoded form
  - You cannot store password in plaintext form
- One-way transformation
  - You cannot decode it back to plaintext form
  - Authentication process compares user-provided password against the encoded one in the storage
- Spring Security supports multiple encoding schemes
  - *MD5PasswordEncoder* (Deprecated)
  - *SHAPasswordEncoder* (Deprecated)
  - *BCryptPasswordEncoder* (Currently recommended)

Challenges of Password Encoding Schemes

- Should be future-proof
  - Assume today’s encoding schemes will be insecure tomorrow
  - New ones will emerge in the future
- Should accommodate old password formats
  - Old format passwords should be able to 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

- Uses new password storage format: *{id}encodedPassword*
  - {id} = PasswordEncoder used to encrypt password
- Delegates to another PasswordEncoder based upon {id}
- BCrypt is current default

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {
  PasswordEncoder encoder  = PasswordEncoderFactories.createDelegatingPasswordEncoder();

  UserDetails user =
     User.withUsername("user").password(passwordEncoder.encode("user")).roles("USER").build();

  return new InMemoryUserDetailsManager(user);
}
```

```
Generates {bcrypt}$2a$10$qfHYt54ZGLkHH4/SXgvPiudiNR5s.5bXX0QtTSTvLNyK8/aGec4s2
```

Enabling HTTP Authentication - 1

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

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http.authorizeHttpRequests((authz) -> authz
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/accounts/**").hasAnyRole("USER","ADMIN")
            .anyRequest().authenticated())
        .httpBasic(withDefaults()); //  Enable HTTP Basic

   return http.build();
}
```

Browser will prompt for username & password

Enabling HTTP Authentication - 2

Form based login

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests((authz) -> authz
                .requestMatchers("/accounts/**").hasRole("USER")
                …
            )
            .formLogin(form -> form        //  setup form-based authentication
                    .loginPage("/login") //  URL to use when login is needed
                    .permitAll()          //  any user can access
            )
            .logout(logout -> logout                      //  configure logout
                    .logoutSuccessUrl("/home")  //  go here after successful logout
                    .permitAll()                        //  any user can access
            );
    return http.build();
}
```

Default: `/login?logout`

An Example Login Page

URL that indicates an authentication request.
Default: POST to same URL used to display the form.

The expected keys for generation of an authentication request token

```
<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>
```

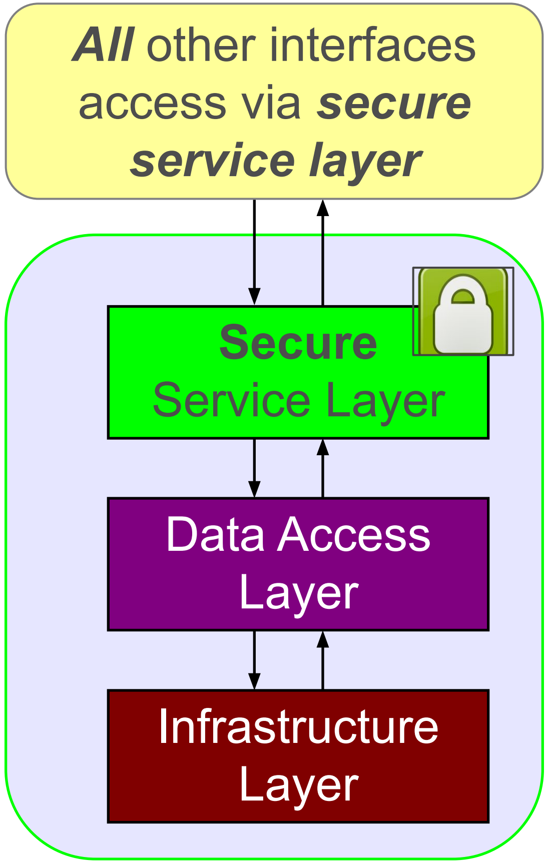
login.html

Agenda

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

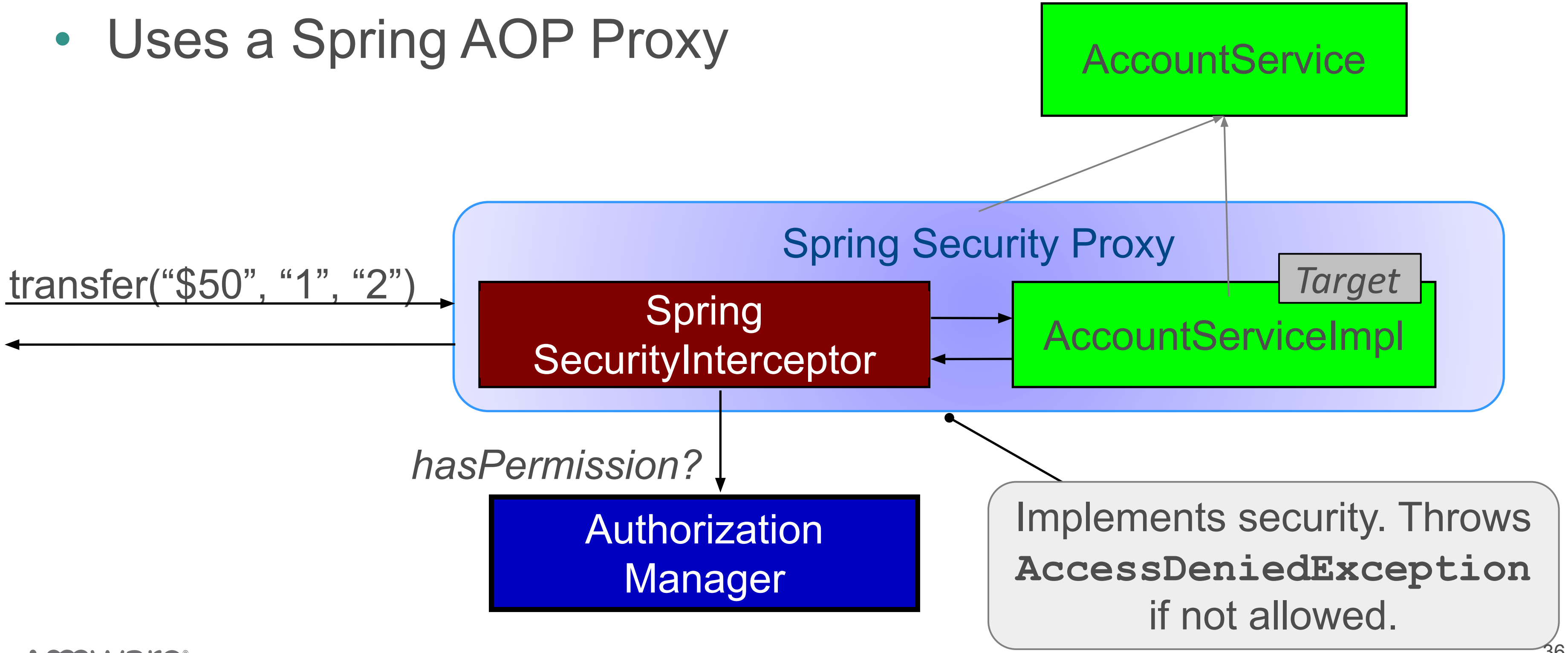
Method Security

- 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


All other interfaces access via ***secure service layer***
Secure Service Layer
Data Access Layer
Infrastructure Layer

Method Security – How it Works

- Uses a Spring AOP Proxy


AccountService
Spring Security Proxy
transfer("$50", "1", "2")
Spring SecurityInterceptor
Target
AccountServiceImpl
hasPermission?
Authorization Manager
Implements security. Throws AccessDeniedException if not allowed.

Apparté sur SpEL (Spring Expression Language)

```
@Configuration
public class AppConfiguration {

  @Value("${application.name}")
  private String appName

  private String appVersion;

  public AppConfiguration(@Value("${application.version}" String appVersion) {
    this.appVersion = appVersion;
  }
}
```

https://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html

Method Security with SpEL

- Use Pre/Post annotations for SpEL

```
@EnableMethodSecurity
```

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

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

Expression-based access control
https://docs.spring.io/spring-security/reference/servlet/authorization/expression-based.html#_overview

Agenda

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

MockMvc Testing - @WithMockUser

```
@WebMvcTest(AccountController.class)
@ContextConfiguration(classes = {RestWsApplication.class, SecurityConfig.class})
public class AccountControllerTests {

    @Test
    @WithMockUser(roles = {"INVALID"})
    public void accountSummary_with_invalid_role_should_return_403() throws Exception {
      mockMvc.perform(get("/accounts"))
                  .andExpect(status().isForbidden());
     }

    @Test
    @WithMockUser(roles = {"ADMIN"})
    public void accountDetails_with_ADMIN_role_should_return_200() throws Exception {
      mockMvc.perform(get("/accounts/0")).andExpect(status().isOk())
                  .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                  .andExpect(jsonPath("name").value("John  Doe"))
                  .andExpect(jsonPath("number").value("1234567890"))
    }
}
```

Use invalid role for testing

Use “ADMIN” role for testing

MockMvc Testing - @WithUserDetails

```
@WebMvcTest(AccountController.class)
@ContextConfiguration(classes = {RestWsApplication.class, SecurityConfig.class})
public class AccountControllerCustomUserDetailsServiceTests {

    @Test
    @WithUserDetails("mary")
    public void accountDetails_with_mary_credentials_should_return_200() throws Exception {
      mockMvc.perform(get("/accounts/0"))
                    .andExpect(status().isOk())
                    .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                    .andExpect(jsonPath("name").value("John   Doe"))
                    .andExpect(jsonPath("number").value("1234567890"))
    }
}
```

Use the user “mary” returned by the `UserDetailsService`

Security Testing (against a running app)

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class AccountClientTests {

  @Autowired
  private TestRestTemplate restTemplate;

  @Test
  public void listAccounts_using_invalid_user_should_return_401() {
     ResponseEntity<String> responseEntity
           = restTemplate.withBasicAuth("invalid", "invalid")
                               .getForEntity("/accounts", String.class);
     assertThat(responseEntity.getStatusCode()).isEqualTo(HttpStatus.UNAUTHORIZED);
  }

 @Test
 public void listAccounts_using_valid_user_should_succeed() {
   ResponseEntity<Account[]> responseEntity
        = restTemplate.withBasicAuth("admin", "admin")
                               .getForEntity("/accounts", Account[].class);
   assertThat(responseEntity.getStatusCode()).isEqualTo(HttpStatus.OK);
  }
}
```

Use invalid user credentials

Use “admin”/”admin” user credentials

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 *DelegatingPasswordEncoder*

Lab: Securing a RESTful application

https://github.com/Nimedas/imt-spring-2025

Branche : 3-solution

Optional Topics: Filter Details, Configuration Choices, Legacy Apps

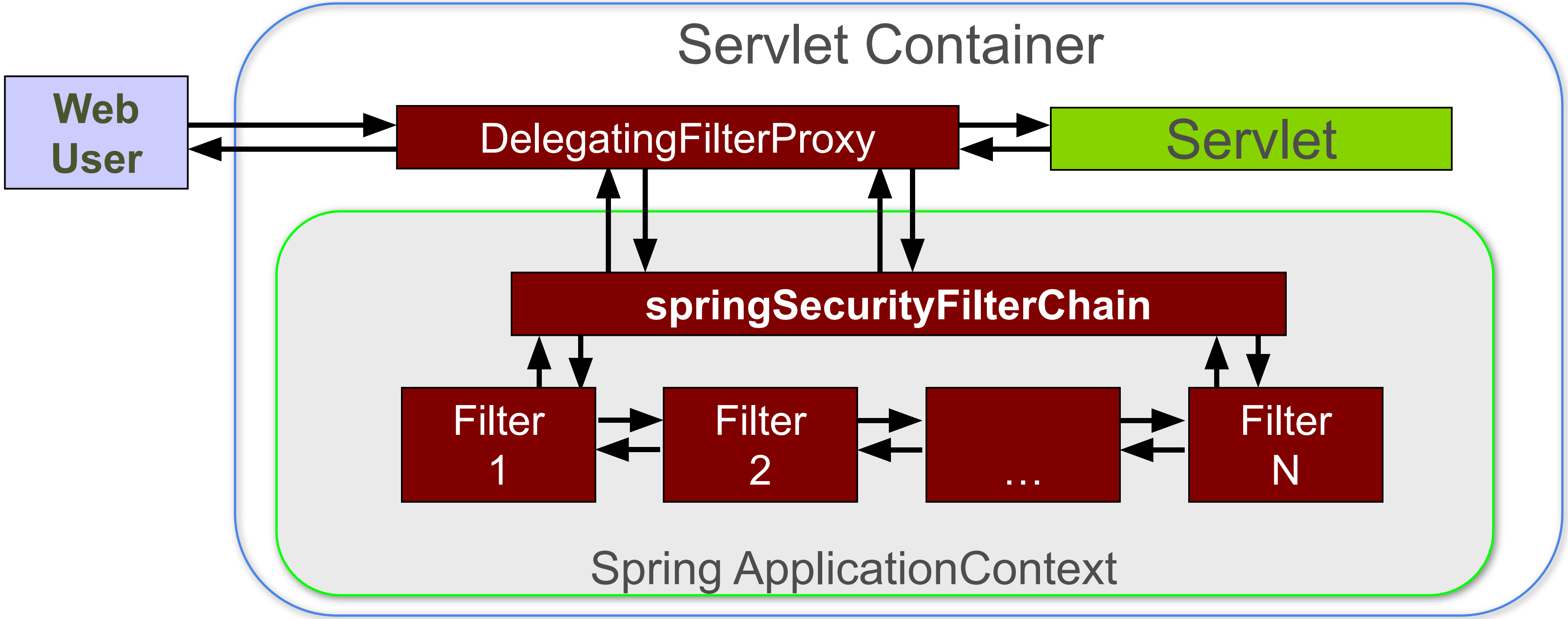
Agenda

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

Web Security Filter Configuration

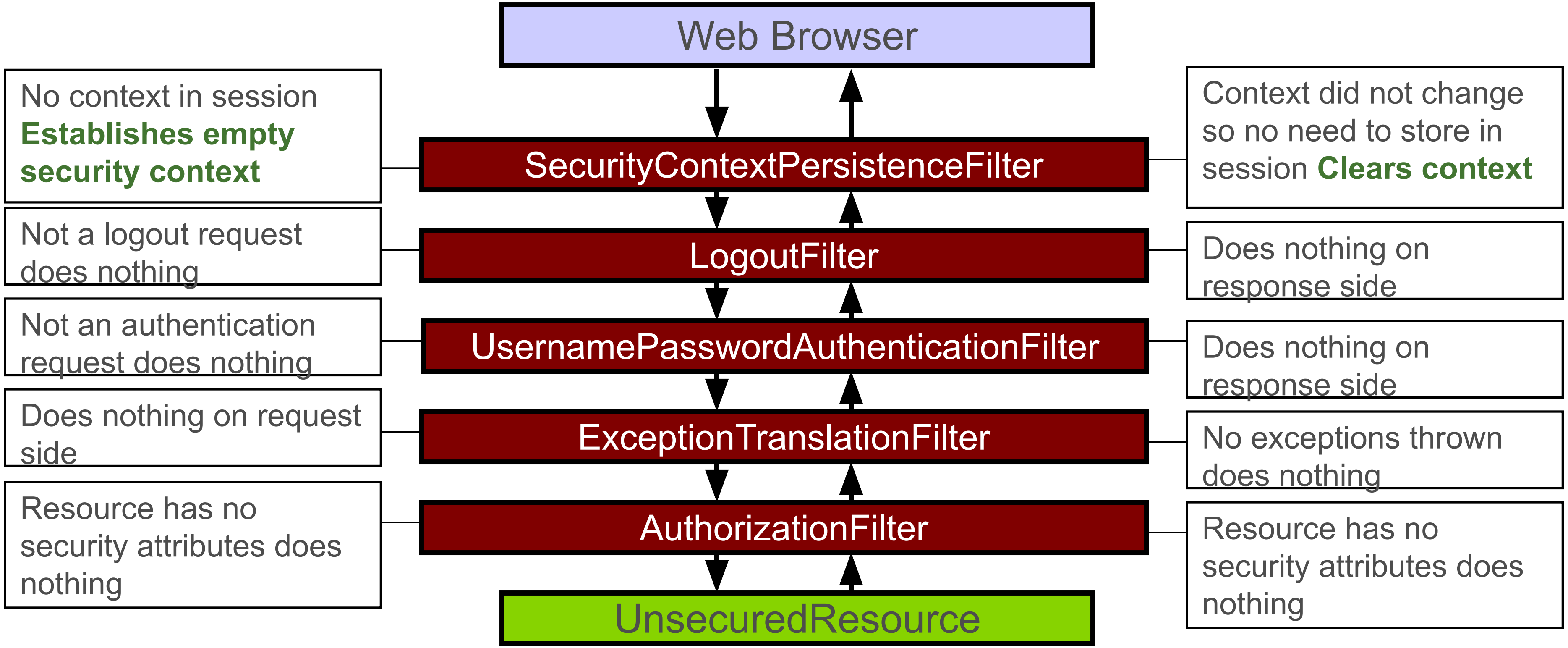


The Filter Chain

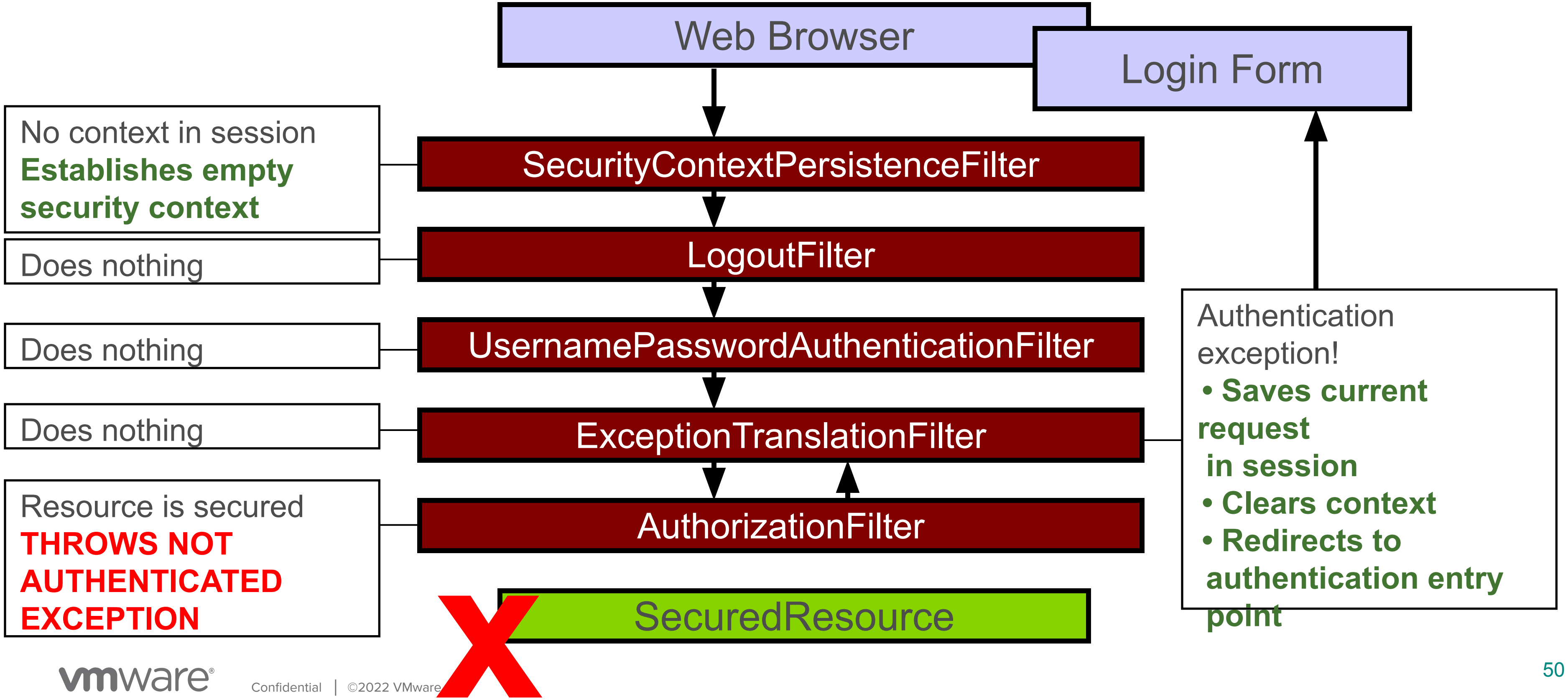
- Spring Security uses a chain of many, many filters
  - Filters initialized with correct values by default
  - Manual configuration is not required **unless you want to customize Spring Security's behavior**
  - It is still important to understand how they work underneath

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

Access Unsecured Resource Prior to Login

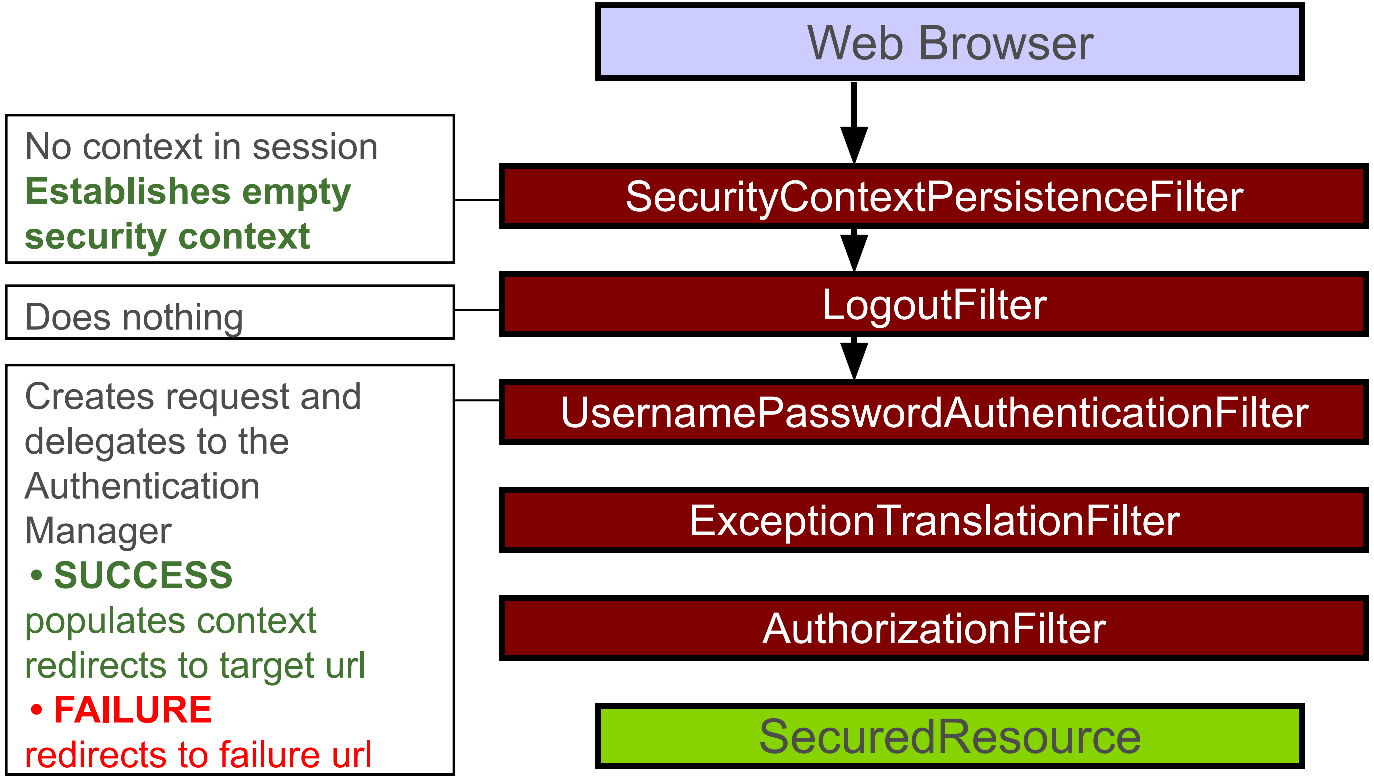

Web Browser
No context in session **Establishes empty security context**
SecurityContextPersistenceFilter
Context did not change so no need to store in session **Clears context**
Not a logout request does nothing
LogoutFilter
Does nothing on response side
Not an authentication request does nothing
UsernamePasswordAuthenticationFilter
Does nothing on response side
Does nothing on request side
ExceptionTranslationFilter
No exceptions thrown does nothing
Resource has no security attributes does nothing
AuthorizationFilter
Resource has no security attributes does nothing
UnsecuredResource

Access Secured Resource Prior to Login



Web Browser
Login Form
No context in session
Establishes empty security context
SecurityContextPersistenceFilter
Does nothing
LogoutFilter
Does nothing
UsernamePasswordAuthenticationFilter
Does nothing
ExceptionTranslationFilter
Resource is secured
THROWS NOT AUTHENTICATED EXCEPTION
AuthorizationFilter
SecuredResource
X
Authentication exception!
• Saves current request in session
• Clears context
• Redirects to authentication entry point

Submit Login Request


Web Browser
SecurityContextPersistenceFilter
LogoutFilter
UsernamePasswordAuthenticationFilter
ExceptionTranslationFilter
AuthorizationFilter
SecuredResource

No context in session
Establishes empty security context

Does nothing

Creates request and delegates to the Authentication Manager

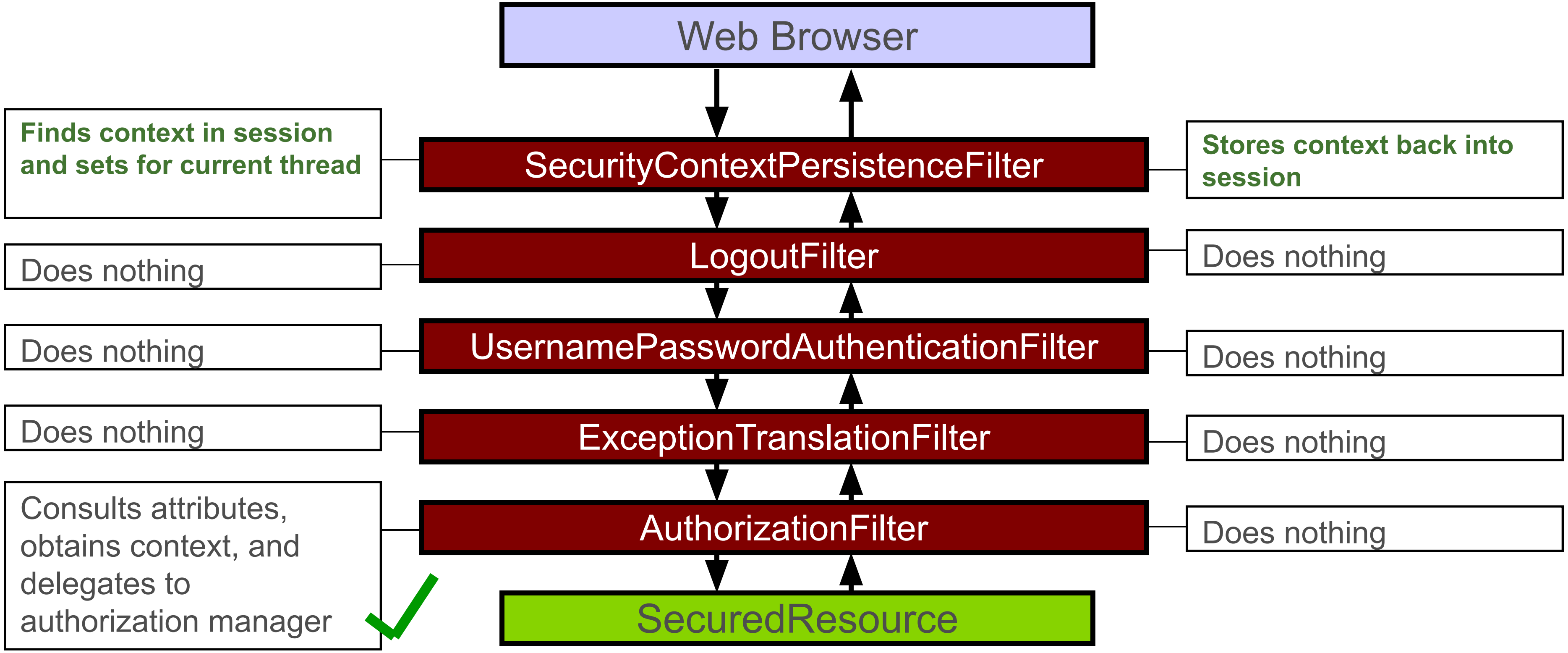
- **SUCCESS**

populates context
redirects to target url

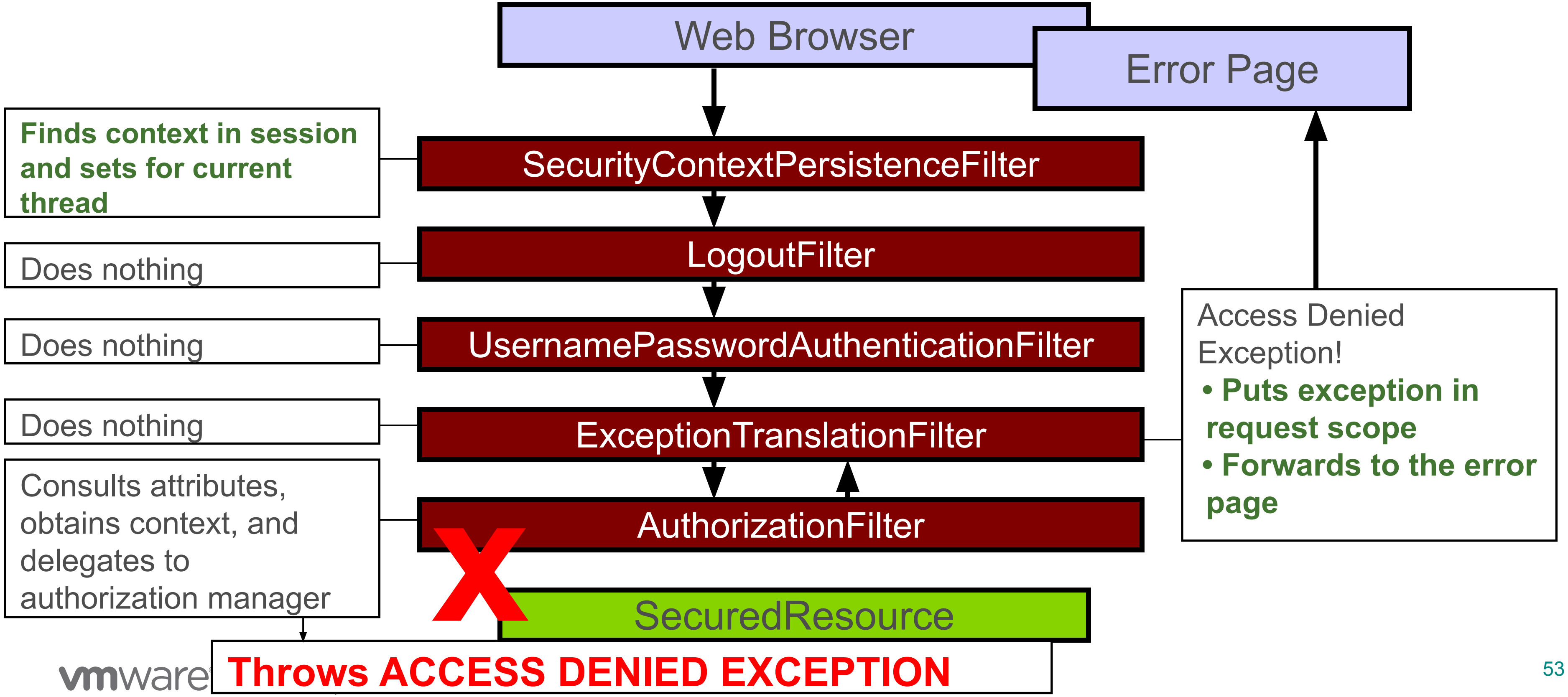
- **FAILURE**

redirects to failure url

Access Resource With Required Role


Web Browser
Finds context in session and sets for current thread
SecurityContextPersistenceFilter
Stores context back into session
Does nothing
LogoutFilter
Does nothing
Does nothing
UsernamePasswordAuthenticationFilter
Does nothing
Does nothing
ExceptionTranslationFilter
Does nothing
Consults attributes, obtains context, and delegates to authorization manager
AuthorizationFilter
Does nothing
SecuredResource

Access Resource Without Required Role


Web Browser
Error Page
SecurityContextPersistenceFilter
LogoutFilter
UsernamePasswordAuthenticationFilter
ExceptionTranslationFilter
AuthorizationFilter
SecuredResource
X

Finds context in session and sets for current thread

Does nothing

Does nothing

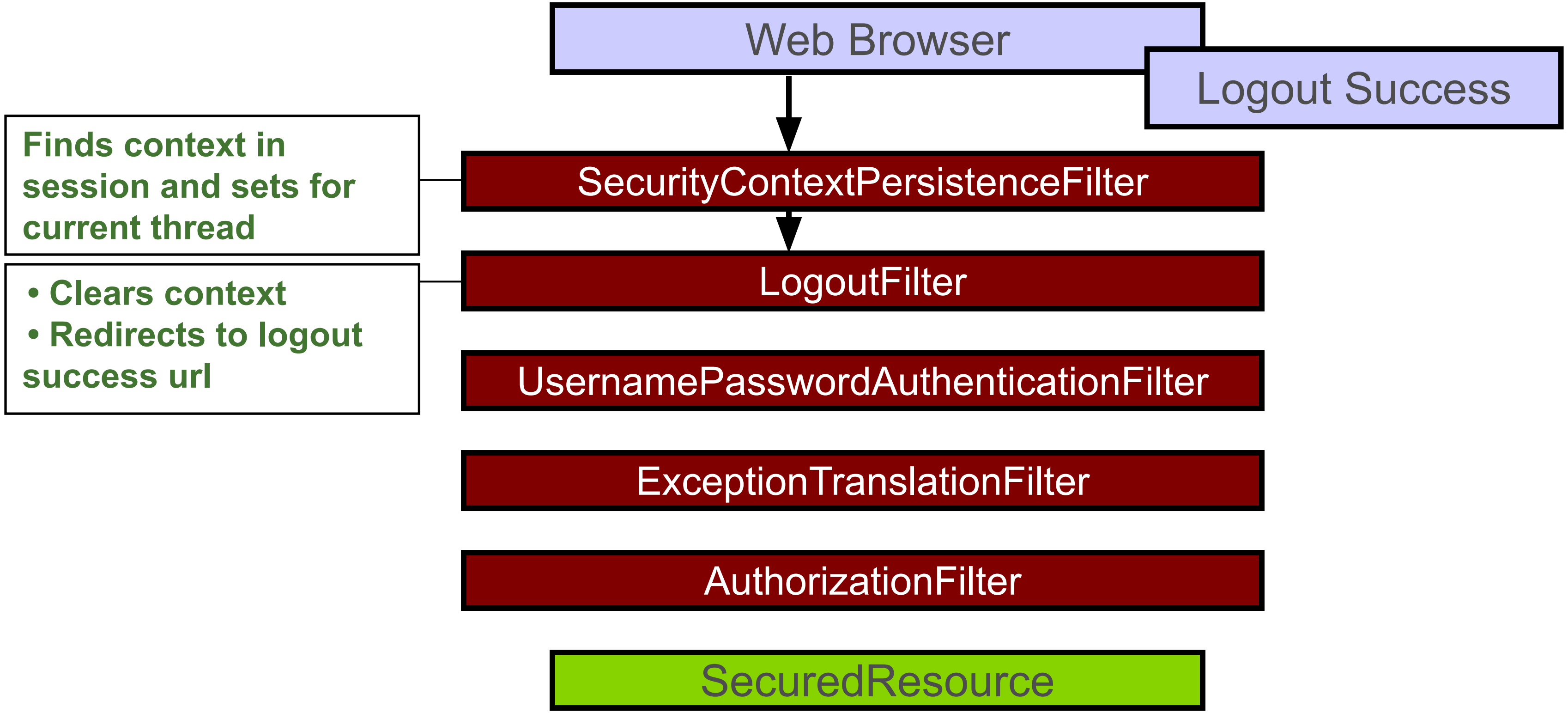
Does nothing

Consults attributes, obtains context, and delegates to authorization manager

Throws ACCESS DENIED EXCEPTION

Access Denied Exception!
- **Puts exception in request scope**
- **Forwards to the error page**

Submit Logout Request


Web Browser
Logout Success
Finds context in session and sets for current thread
SecurityContextPersistenceFilter
• Clears context
• Redirects to logout success url
LogoutFilter
UsernamePasswordAuthenticationFilter
ExceptionTranslationFilter
AuthorizationFilter
SecuredResource

The Filter Chain: Summary

#	Filter Name	Main Purpose
1	`SecurityContext PersistenceFilter`	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	`AuthorizationFilter`	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 );
```