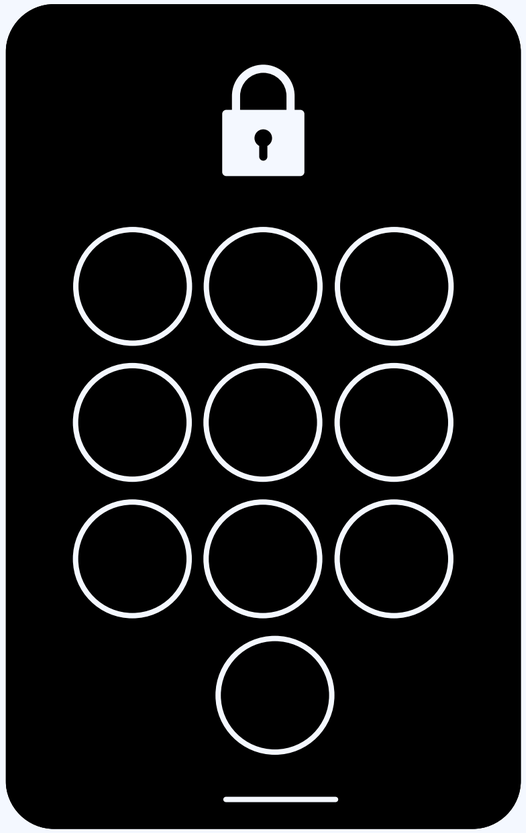
Authentication | Authorization | SecurityContext | Filter Chain | CSRF | JWT

SPRING SECURITY

Protecting the Digital World



QUESTIONS

SECURITY

How can I implement security to my web/mobile applications so that there won't be any security breaches in my application?

PASSWORDS

How to store passwords, validate them, encode, decode them using industry standard encryption algorithms?

USERS & ROLES

How to maintain the user level security based on their roles and grants associated to them?

MULTIPLE LOGINS

How can I implement a mechanism where the user will login only once and start using my application?

FINE GRAINED SECURITY

How can I implement security at each level of my application using authorization rules?

CSRF & CORS

What is CSRF attacks and CORS restrictions. How to overcome them?

JWT & OAuth2

What is JWT and OAuth2. How I can protect my web application using them?

PREVENTING ATTACKS

How to prevent security attacks like Brute force, stealing of data, session fixation?

SPRING SECURITY

SECURITY

Spring Security provides a comprehensive security framework that ensures your application is protected through authentication, authorization, and session management. Its configurable architecture helps safeguard against common vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF).

MULTIPLE LOGINS

Spring Security handles session management effectively, ensuring users stay logged in securely. It supports Single Sign-On (SSO) solutions and integrations with OAuth2 and SAML to allow seamless login across multiple applications.

USERS & ROLES

Spring Security supports role-based access control (RBAC) using annotations like @PreAuthorize and @Secured or URL-based authorization. You can define roles and permissions and map them to users to restrict access to resources or operations dynamically.

PASSWORDS

Spring Security includes tools like PasswordEncoder to securely hash and store passwords using algorithms such as BCrypt, SCrypt, and PBKDF2. These ensure passwords are never stored in plain text and provide mechanisms for secure password validation during login.

FINE-GRAINED SECURITY

Spring Security allows you to define fine-grained access rules at different layers of your application. With method-level security, URL-based rules, and global method security configurations, you can tailor security policies for specific resources and actions.

SPRING SECURITY

CSRF & CORS

Spring Security provides built-in protection against CSRF attacks by generating and validating CSRF tokens for sensitive operations. It also includes customizable CORS configuration, allowing you to control cross-origin requests securely.

JWT & OAuth2

Spring Security offers seamless integration with JWT and OAuth2. JWT ensures stateless authentication, while OAuth2 provides secure, token-based authorization for APIs and third-party access. Spring Security simplifies configuring these protocols for your application.

PREVENTING ATTACKS

How to prevent security attacks like brute force, data theft, session fixation?
Spring Security includes features to mitigate various attacks:

- Brute force: Account lockout mechanisms after repeated failed attempts.
- Data theft: Secure data transmission using HTTPS and encryption.
- Session fixation: Spring Security regenerates session IDs on successful authentication to protect against fixation attacks.

INTRODUCTION

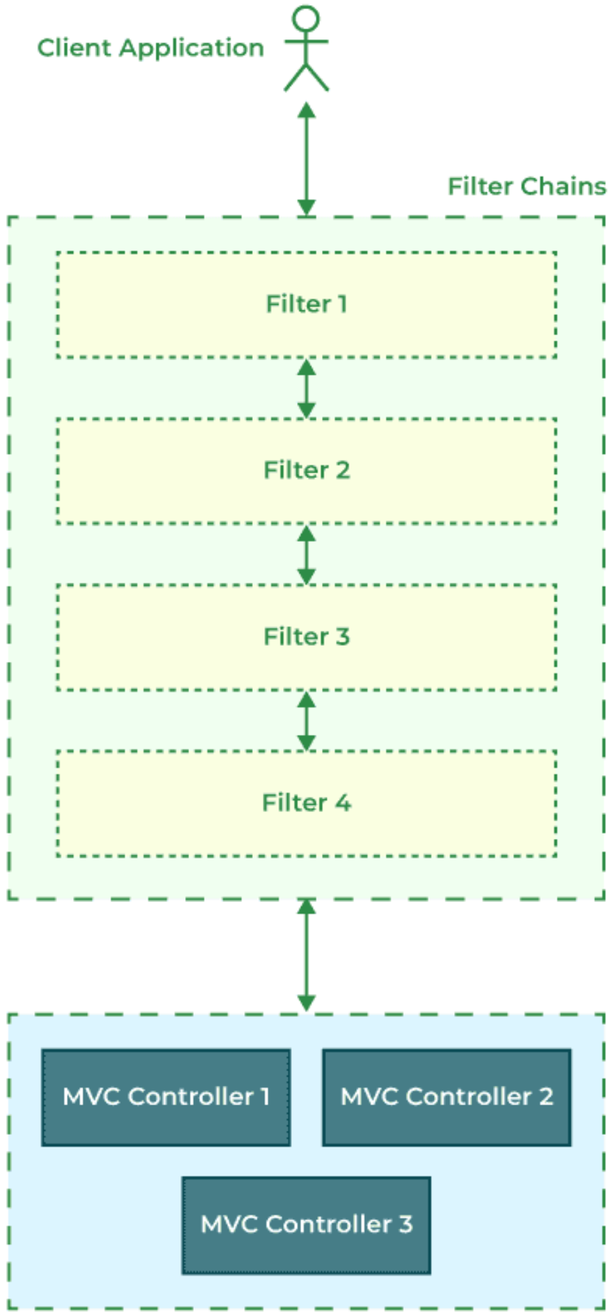
What is Spring Security?

Spring Security is a flexible framework that secures applications by managing authentication, authorization, and protection against common threats like CSRF and brute force. It supports role-based access, password encryption, JWT, OAuth2, and session management, offering everything needed for modern application security.

CORE FEATURES

- Authentication/Authorization
- Protection Against Common Vulnerabilities
- Password Management
- Session and CSRF Protection
- OAuth2 and OpenID Connect
- Filter Chain Architecture

SECURITY FILTERS



What are Filter in spring ?

Filters in Spring are components that intercept HTTP requests and responses, They are implemented as classes that implement the (javax/jakarta).servlet.Filter interface. They can intercept and manipulate requests and responses before they reach the servlet or after the servlet has processed them. Filters can be used for various purposes such as input validation, logging, authentication, authorization, and request/response modification. Filters are configured in the Spring application context and can be applied to specific URLs or all URLs in the application.

How Filters Work:

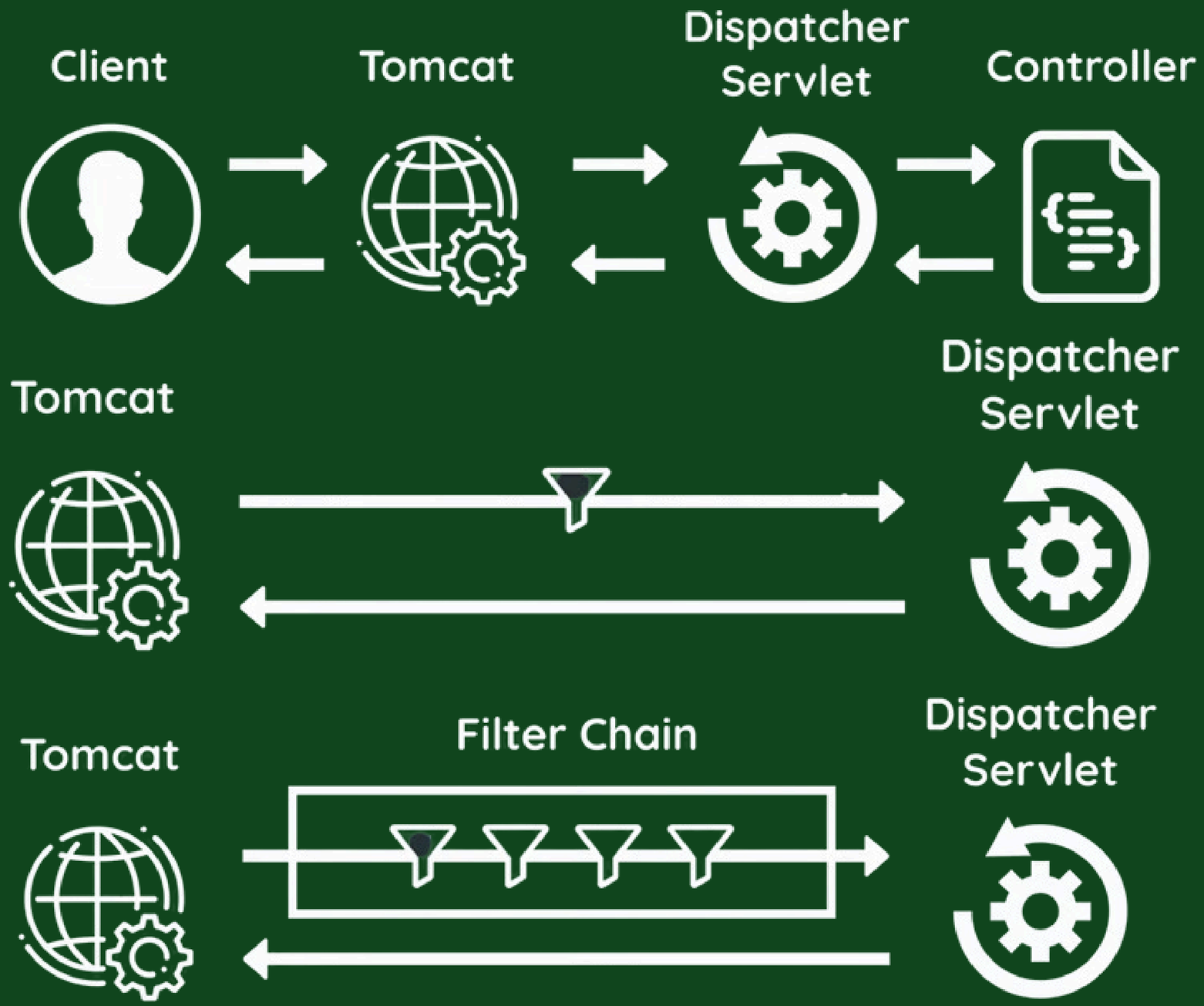
Request Flow: When a request arrives, it passes through a filter chain before reaching the DispatcherServlet or the target resource (e.g., servlet or static content).

Filter Chain: Filters in the chain are executed sequentially based on a configured order. Each filter can:

- Inspect or modify the request.
- Perform logic like authentication or logging.
- Decide whether to forward the request or terminate processing.

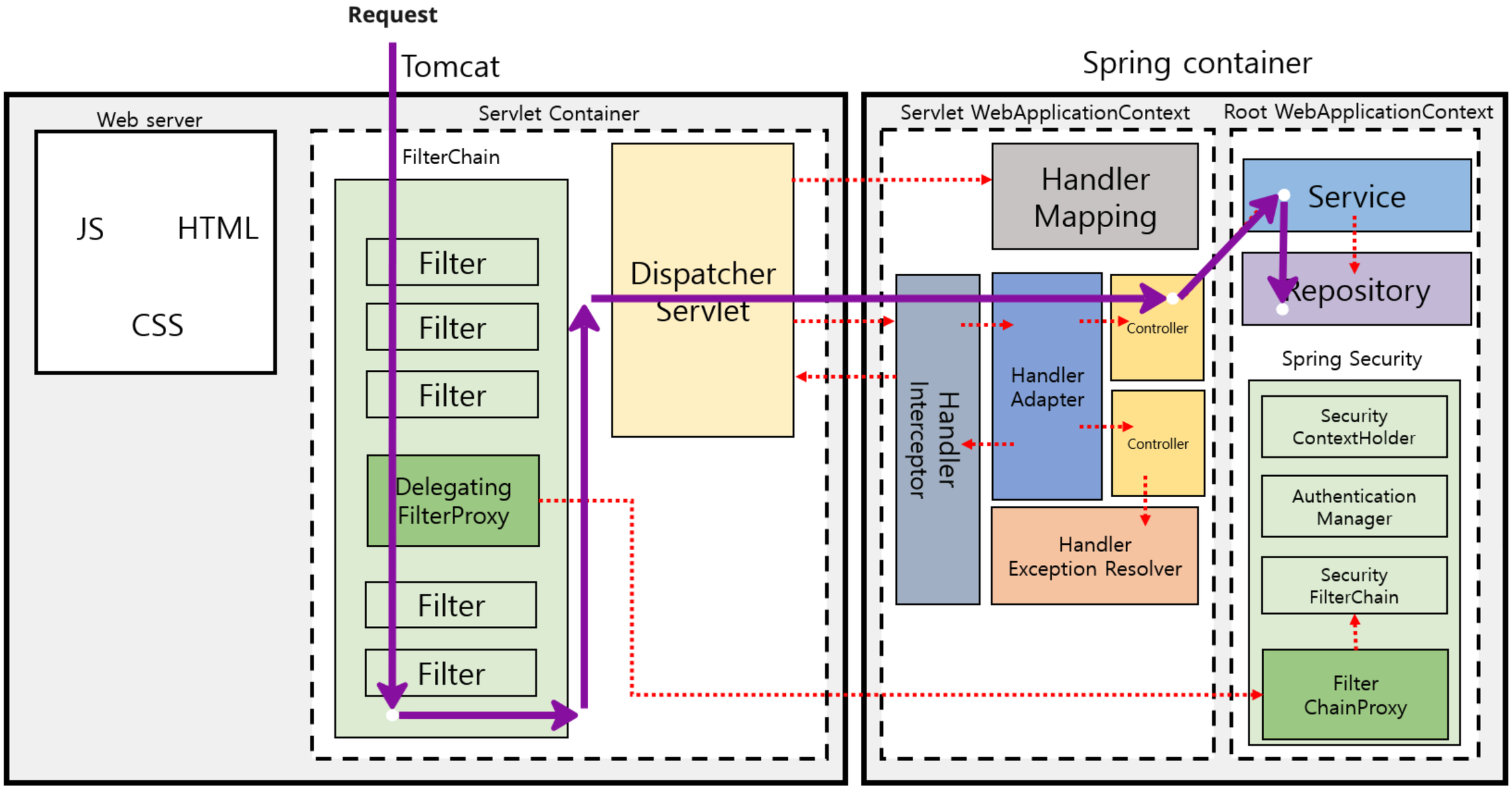
Response Flow: After the servlet processes the request, the response travels back through the filter chain for any final modifications.

Filters are configurable for specific URLs or all URLs in the application, enabling flexible and modular processing of HTTP requests and responses.

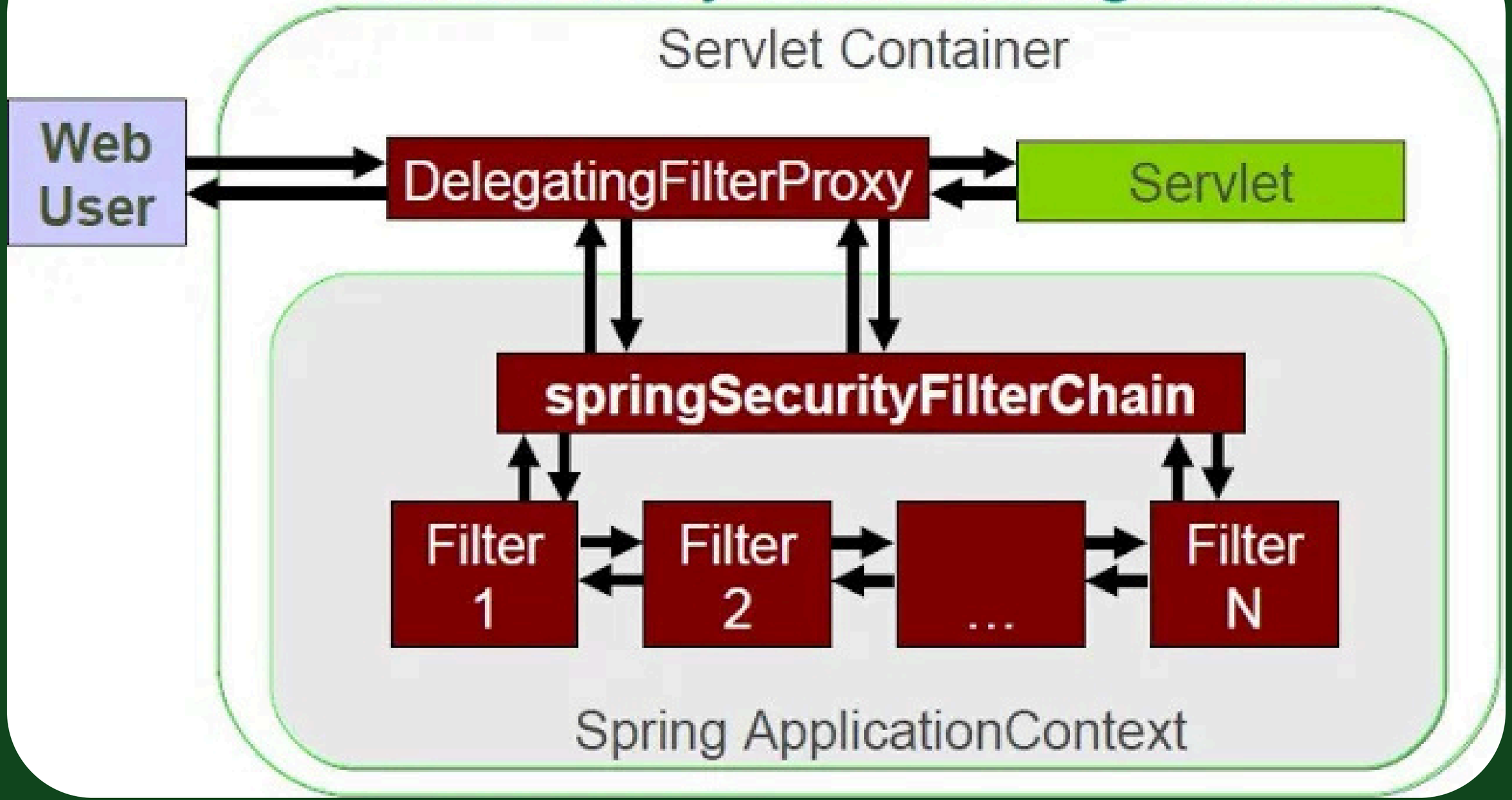


Filter Chain

FilterChain—Represents the chain of filters. We use the FilterChain object to invoke the next filter in the chain, or if the calling filter is the last filter in the chain, to invoke the resource at the end of the chain.


Request
Tomcat
Spring container
Web server
JS
HTML
CSS
Servlet Container
FilterChain
Filter
Filter
Filter
Delegating FilterProxy
Filter
Filter
Dispatcher Servlet
Servlet WebApplicationContext
Root WebApplicationContext
Handler Mapping
Handler Interceptor
Handler Adapter
Controller
Controller
Handler Exception Resolver
Service
Repository
Spring Security
Security ContextHolder
Authentication Manager
Security FilterChain
Filter ChainProxy

Web Security Filter Configuration



Purpose of Filters in Spring Security:

In Spring Security, filters play a critical role in securing the web application by intercepting HTTP requests, applying security policies, and modifying responses. Filters help enforce authentication, authorization, logging, and security rules. They act as checkpoints in the request flow where security operations are executed.

How Spring Security Implements Filters

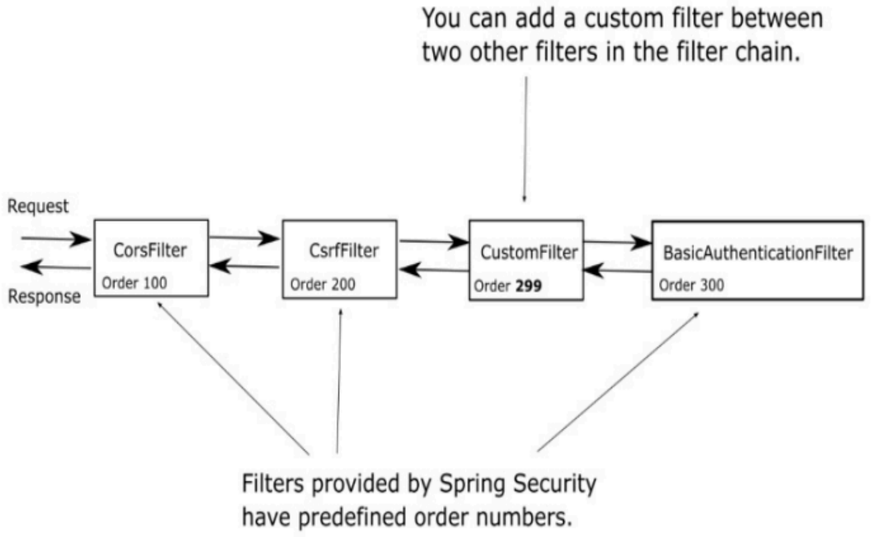
Spring Security Architecture Overview:
Spring Security uses a filter-based architecture, where various security-related concerns are implemented as filters. These filters are typically arranged in a filter chain and work together to secure the application.
DelegatingFilterProxy and Its Role:
DelegatingFilterProxy is a special filter provided by Spring that delegates filter logic to a Spring bean (typically a Filter implementation).
It is used to integrate Spring Security filters into the web application's filter chain.
Key Built-in Filters in Spring Security:

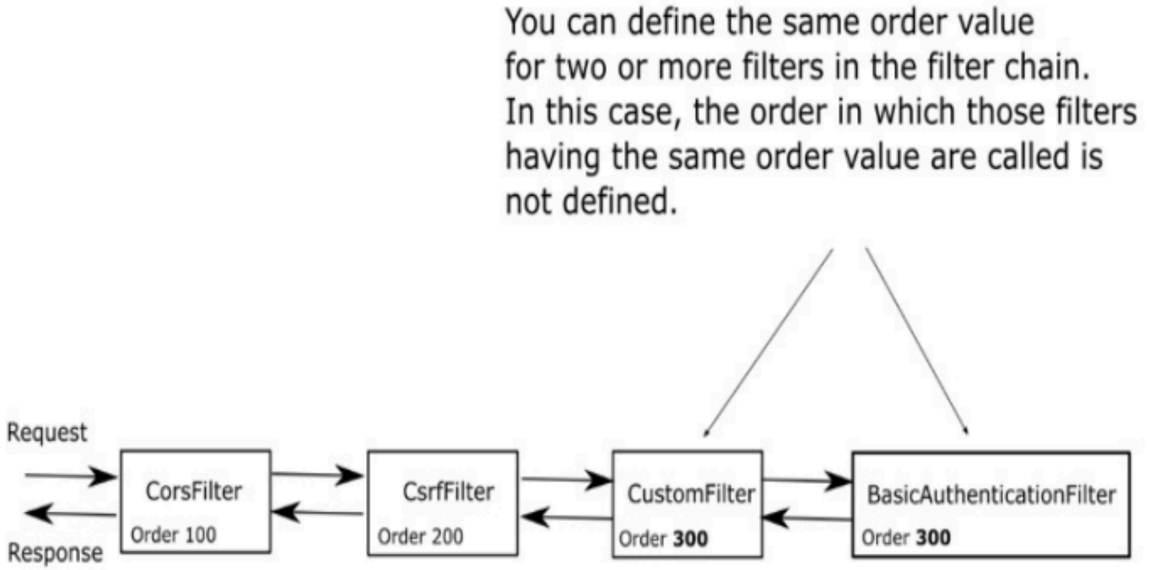
- **SecurityContextPersistenceFilter:** This filter ensures that the security context is stored and retrieved from the HTTP session, providing access to authentication information throughout a user's session.
- UsernamePasswordAuthenticationFilter: Handles form-based authentication by processing username and password login requests.
- **BasicAuthenticationFilter:** Deals with basic HTTP authentication by extracting credentials from the HTTP request header.
- **ExceptionTranslationFilter**: Catches authentication or authorization exceptions and translates them into HTTP responses (e.g., 403 Forbidden, 401 Unauthorized).
- **FilterSecurityInterceptor**: Performs access control decisions by verifying if the current user has the required permissions to access a resource.

Order and Execution of Filters in the Chain:

The filters in Spring Security are ordered in a chain, and they are executed in a specific sequence. The order is critical for correct behavior:

- Filters like SecurityContextPersistenceFilter should run early in the chain to store authentication context.
- Filters like FilterSecurityInterceptor should run after authentication and authorization filters to make access decisions.





AUTHENTICATION

Authentication is the process of verifying a user's identity.
Spring Security provides several ways to authenticate users:

- Username/password authentication
- OAuth2/OpenID Connect
- JWT (JSON Web Tokens)
- LDAP (Lightweight Directory Access Protocol)
- Social login (e.g., Google, Facebook)

Includes out-of-the-box support for login forms, HTTP Basic, and Digest authentication.
Supports custom authentication providers.

Core Components of Authentication in Spring Security

AuthenticationManager

- Central interface for processing authentication requests.
- Uses authenticate(Authentication authentication) to validate credentials.

AuthenticationProvider

- Performs the actual authentication logic.
- Supports different mechanisms such as database-based, LDAP, or custom authentication.

UserDetailsService

- Interface to load user-specific data.
- Commonly used to retrieve user details from a database.

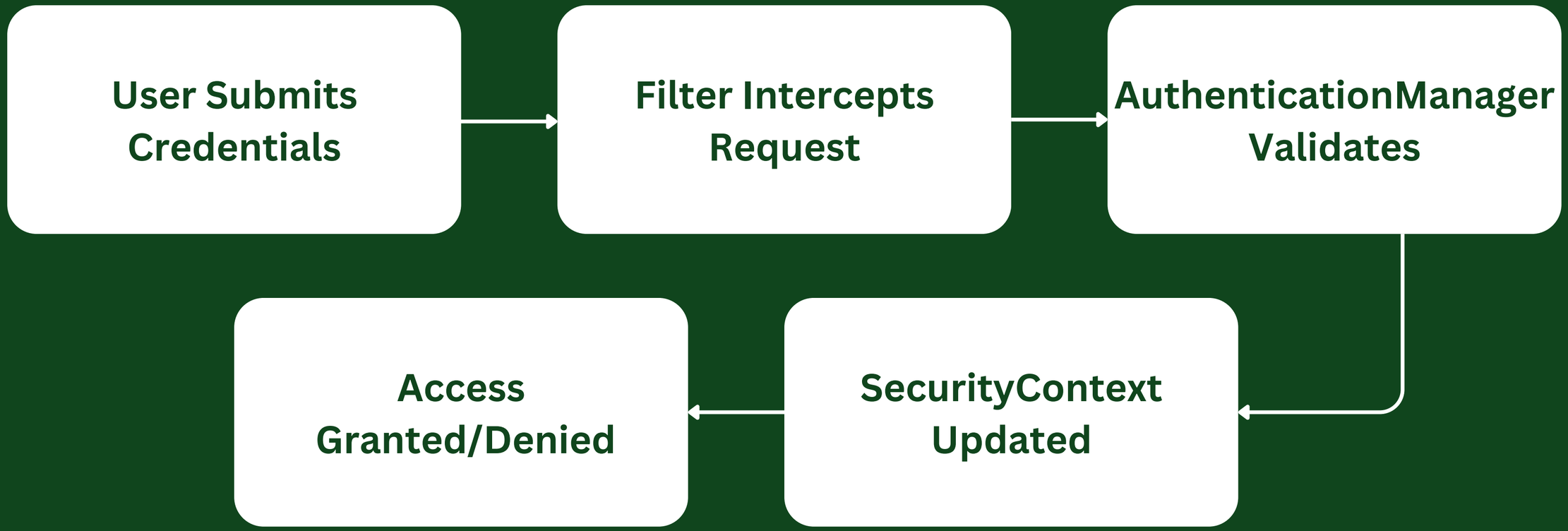
AuthenticationToken

Represents the credentials provided by the user (e.g., UsernamePasswordAuthenticationToken).

SecurityContext and SecurityContextHolder

Stores the security information of the currently authenticated user

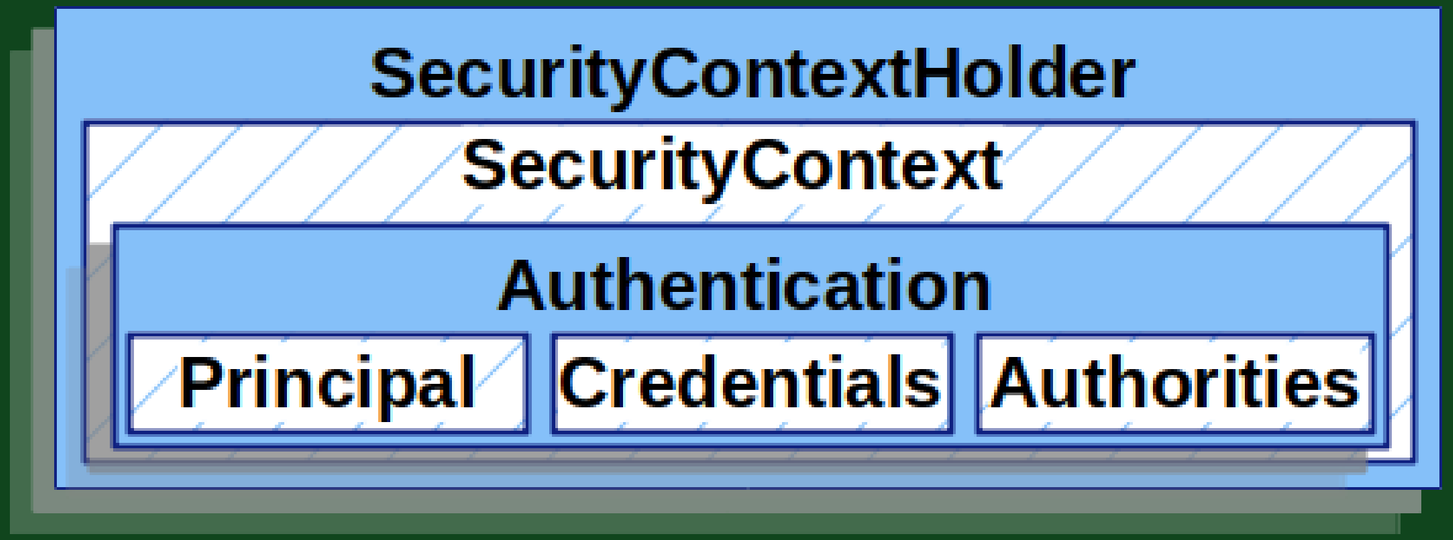
AUTHENTICATION WORKFLOW



User Submits Credentials: Credentials are sent from the client (e.g., username and password).
Filter Intercepts Request: The UsernamePasswordAuthenticationFilter or a custom filter intercepts the request.
AuthenticationManager Validates: Delegates authentication to one or more AuthenticationProvider.
SecurityContext Updated: If successful, the SecurityContextHolder stores the authentication details.
Access Granted/Denied: Based on the authentication result, the user can access protected resources.

Aspect	AuthenticationManager	AuthenticationProvider
Definition	Main interface for managing authentication in Spring Security.	Interface responsible for actual authentication logic.
Role	Delegates authentication requests to one or more AuthenticationProvider instances.	Validates the credentials and returns an authenticated Authentication object if successful.
Default Implementation	ProviderManager, which supports multiple AuthenticationProviders.	Common implementations include DaoAuthenticationProvider , LdapAuthenticationProvider, etc.
Custom Logic	Rarely customized. Usually configured to use multiple providers.	Often customized for specific authentication mechanisms (e.g., custom username/password validation).
Input	Takes an Authentication object with credentials.	Receives the same Authentication object from AuthenticationManager.
Output	Returns a fully populated Authentication object upon successful authentication.	Returns an authenticated Authentication object or throws an AuthenticationException on failure.
Exception Handling	Throws AuthenticationException if no provider can authenticate.	Throws AuthenticationException if validation fails for the handled authentication type.
Usage	Used to abstract and centralize the authentication process.	Handles the actual validation and is used by the AuthenticationManager.
Customization Example	Implement a custom AuthenticationManager to replace ProviderManager.	Implement a custom AuthenticationProvider to add logic for validating credentials against custom stores.
Supports Multiple Types	Yes, delegates to providers to handle different authentication mechanisms.	Typically handles a specific authentication mechanism, defined by the supports() method.

Security Context design



Security Context Holder

The **SecurityContextHolder** is a helper class provided by **Spring Security**. It serves as the gateway to access the **SecurityContext**, which holds the security information (like the authenticated user) for the current thread.

The **SecurityContextHolder** employs one of three strategies to store the **SecurityContext**. By default, it uses **ThreadLocal**, which associates a **SecurityContext** with the current thread, ensuring each thread handling a request has its own isolated **SecurityContext**.

In less common scenarios, it can operate in **Global Mode**, where the **SecurityContext** is stored globally rather than being thread-specific; however, this approach is **rarely** used in modern applications due to its limitations.

Alternatively, developers can define a **Custom Strategy**, creating their own mechanism for storing the **SecurityContext**, such as a custom store tailored to specific application needs.

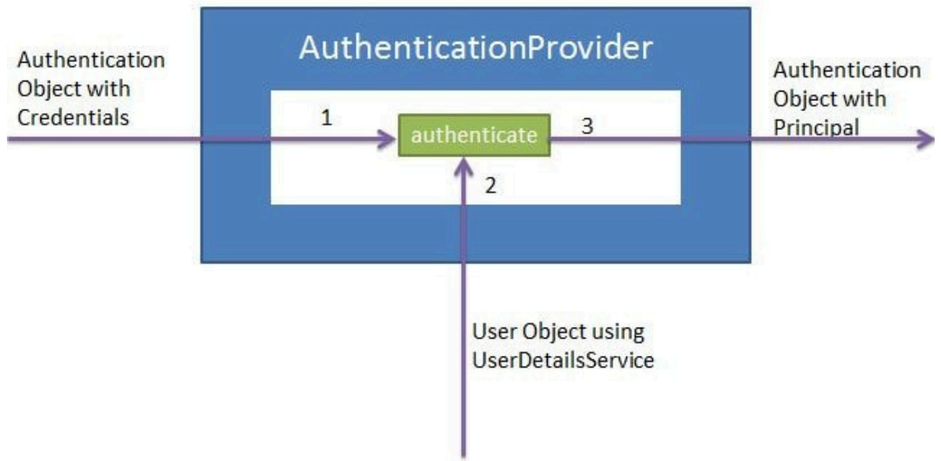
ThreadLocal

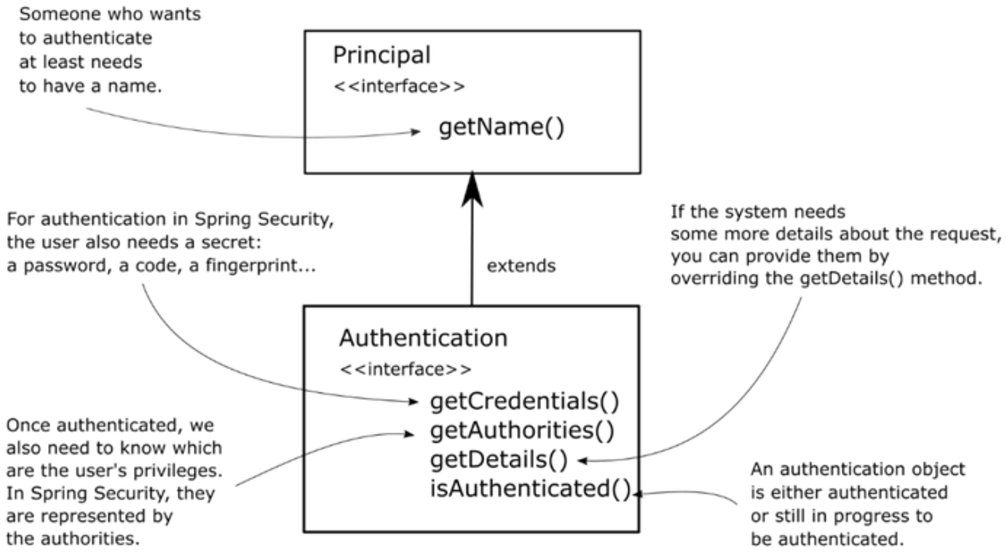
ThreadLocal is a class in Java that allows you to create variables that can only be **read** and **written** by the same thread.

This can be useful in situations where you have multiple threads accessing the same variable, but you want to ensure that each thread has its own isolated copy of the variable.

Authentication object or Context

The **Authentication** object in **Spring Security** represents the **principal** (user) and their **credentials**, **authorities**, and **authentication status** *i.e: isAuthenticated(): boolean method*. It acts as the key component to store and check user identity and access rights within the SecurityContext.





AUTHORIZATION

Authorization is the process of determining whether a user has permission to access a particular resource or perform a specific action.

Spring Security handles authorization in several ways:

- **Role-based access control (RBAC):** Users can be granted roles (e.g., ROLE_USER, ROLE_ADMIN), and access to certain URLs or resources can be restricted based on these roles.
- **Method-based security:** Spring Security also provides annotations like @PreAuthorize, @Secured, and @RolesAllowed to restrict access to methods based on roles or permissions.

The authorization process typically involves:

- Checking if the authenticated user has the required roles or permissions for a given resource.
- Configuring access rules using HttpSecurity (e.g., restrict access to /admin to users with ROLE_ADMIN).

Authentication Context Principal, Credentials and Authorities

The **principal** represents the user or system entity that has been authenticated. It identifies "who" the authenticated entity is.

The **credentials** are typically the sensitive information used to authenticate the principal. Examples include passwords, tokens, or other authentication data.
After **authentication**, credentials are often set to null for security reasons.

Authorities represent the roles or permissions granted to the principal. They define what actions or resources the user is authorized to access.
They are stored as a collection of **GrantedAuthority** objects.

Granted Authority

GRANTED AUTHORITY

Definition

A **GrantedAuthority** represents a role or permission granted to a user, defining their access rights within the application.
It plays a key role in authorization, ensuring users only access what they are allowed.

Where It Fits in Spring Security

GrantedAuthority is part of the **Authentication** object, stored in the authorities field.
Spring Security checks these authorities to determine whether a user can access a resource or perform an action. ***Examples:*** *"ROLE_ADMIN"* (role-based security), *"READ_PRIVILEGE"* (permission-based security).

GRANTED AUTHORITY

How GrantedAuthority Interacts in My App

GrantedAuthority interacts with various Spring Security components in my app. When a user logs in, the **AuthenticationManager** populates the **GrantedAuthority** list based on the user's roles and permissions.

These authorities are checked by the **AccessDecisionManager** whenever a user tries to access a protected resource. ***For example***, a user with *"ROLE_ADMIN"* can manage contents, while a user with *"ROLE_USER"* can only view them. These checks ensure that actions are restricted to authorized users.

What is AccessDecisionManager?

The primary purpose of the **AccessDecisionManager** is to determine if a user has the necessary **authorities** or **roles** to access a resource. It is invoked after all security filters and checks (like **AuthenticationManager**) have been executed.

The **AccessDecisionManager** evaluates whether the user's granted authorities match the required ones for accessing a particular method or resource.

GRANTED AUTHORITY

Implementations of AccessDecisionManager

- **AffirmativeBased**: Grants access if at least one voter agrees that the user should have access.
- **ConsensusBased**: Grants access only if the majority of voters approve.
- **UnanimousBased**: Grants access only if all voters approve.

How it works

- When a user attempts to access a **protected** resource *(like a URL or a method in your app)*, Spring Security needs to decide if they have sufficient permissions.

- This is done by evaluating the **granted authorities** attached to the current **Authentication** object *(which typically contains roles and permissions)*.

- The **AccessDecisionManager** makes this decision based on a series of **AccessDecisionVoters**.

GRANTED AUTHORITY

What is AccessDecisionVoter?

An **AccessDecisionVoter** is an **interface** in Spring Security that determines whether a user has permission to access a given resource.

It is responsible for evaluating a request against certain security criteria (such as roles, permissions, or other attributes) and voting on whether access should be granted, denied, or abstained.

Primary method of AccessDecisionVoters

The primary method is vote(Authentication authentication, Object object, Collection<ConfigAttribute> attributes), which checks the user's authentication against the given security attributes and returns one of three possible results:

- **ACCESS_GRANTED:** The voter approves access.
- **ACCESS_DENIED:** The voter denies access.
- **ACCESS_ABSTAIN:** The voter abstains from making a decision.

GRANTED AUTHORITY

Implementations of AccessDecisionVoter

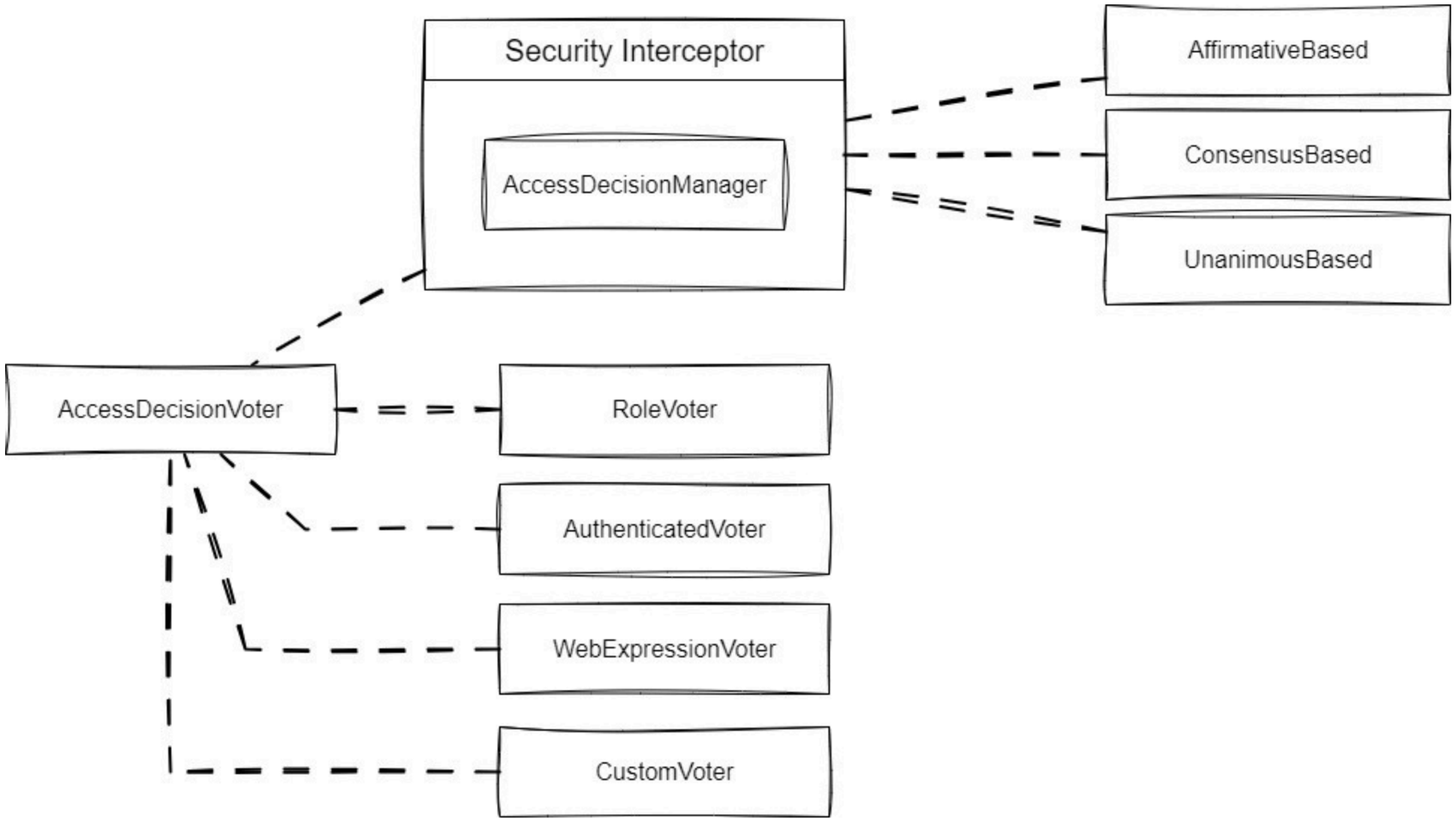
These are the components that the **AccessDecisionManager** uses, that exist inside the **AccessDecisionVoter** to evaluate the request. Voters make individual decisions about whether a user should be allowed access to a resource based on specific logic. ***For example:*** **Role-based voters** *(e.g., checking if the user has ROLE_ADMIN)*, **Permission-based voters** *(e.g., checking if the user has WRITE_PRIVILEGE)*.

There are different types of voters:

- **RoleVoter**: Checks for role-based permissions.
- **AuthenticatedVoter**: Decides access based on whether the user is authenticated.
- **WebExpressionVoter**: Uses SpEL *(Spring Expression Language)* to evaluate more complex access conditions.

GRANTED AUTHORITY

AccessDecisionManager Design



CODE

CONCLUSION

THANKYOU