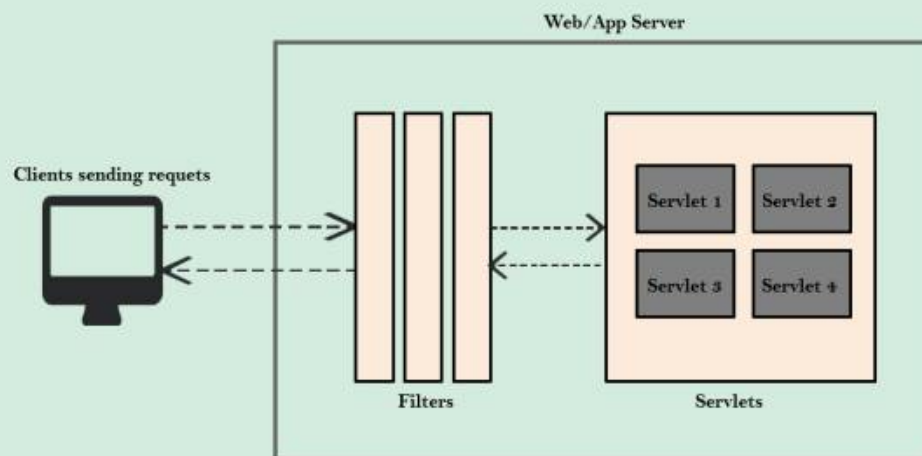


# 1-Servlets and Spring Security



## SERVLETS & FILTERS

**eazy  
bytes**



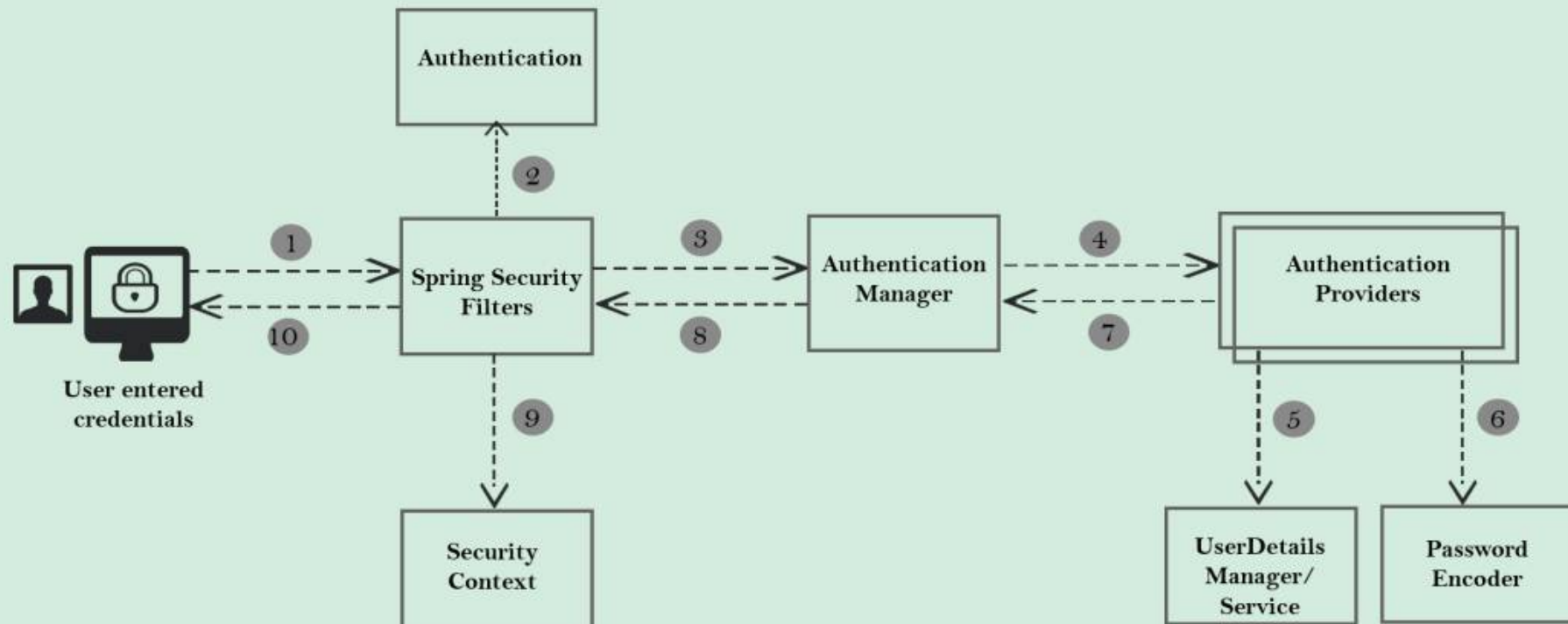
### ★ Typical Scenario inside a web application

*In Java web apps, Servlet Container (Web Server) takes care of translating the HTTP messages for Java code to understand. One of the mostly used servlet container is Apache Tomcat. Servlet Container converts the HTTP messages into ServletRequest and hand over to Servlet method as a parameter. Similarly, ServletResponse returns as an output to Servlet Container from Servlet. So everything we write inside Java web apps are driven by Servlets*

### ★ Role of Filters

*Filters inside Java web applications can be used to intercept each request/response and do some pre work before our business logic. So using the same filters, Spring Security enforce security based on our configurations inside a web application.*

## SPRING SECURITY INTERNAL FLOW



# SPRING SECURITY INTERNAL FLOW

## ★ Spring Security Filters

*A series of Spring Security filters intercept each request & work together to identify if Authentication is required or not. If authentication is required, accordingly navigate the user to login page or use the existing details stored during initial authentication.*

## ★ Authentication

*Filters like UsernamePasswordAuthenticationFilter will extract username/password from HTTP request & prepare Authentication type object. Because Authentication is the core standard of storing authenticated user details inside Spring Security framework.*

## ★ AuthenticationManager

*Once received request from filter, it delegates the validating of the user details to the authentication providers available. Since there can be multiple providers inside an app, it is the responsibility of the AuthenticationManager to manage all the authentication providers available.*

## ★ AuthenticationProvider

*AuthenticationProviders has all the core logic of validating user details for authentication.*

## ★ UserDetailsManager/UserDetailsService

*UserDetailsManager/UserDetailsService helps in retrieving, creating, updating, deleting the User Details from the DB/storage systems.*

## ★ PasswordEncoder

*Service interface that helps in encoding & hashing passwords. Otherwise we may have to live with plain text passwords ☹*

## ★ SecurityContext

*Once the request has been authenticated, the Authentication will usually be stored in a thread-local SecurityContext managed by the SecurityContextHolder. This helps during the upcoming requests from the same user.*

Kindly explain the above images in details



The uploaded images explain concepts of **Servlets and Filters** and the **Spring Security Internal Flow** in Java-based web applications. Here's a detailed explanation for each image:

## Image 1: SERVLETS & FILTERS

### 1. Typical Scenario in a Web Application:

- Java web applications operate using a **Servlet Container** (e.g., Apache Tomcat) that processes HTTP requests and responses.
- The Servlet Container performs the translation of HTTP messages into objects Java can understand:
  - **ServletRequest**: Encapsulates request details.
  - **ServletResponse**: Handles the response to the client.
- The container invokes the appropriate **Servlet** to process the request based on the application's configuration.
- In essence, every Java-based web application is heavily dependent on Servlets for functionality.

### 2. Role of Filters:

- Filters are pre-processing or post-processing components that intercept HTTP requests and responses before or after Servlets.
- They serve tasks such as:
  - Logging
  - Request validation
  - Response modification
  - Implementing security policies (e.g., in **Spring Security**, filters enforce rules to authenticate/authorize requests).
- Filters allow for centralized logic that can act across multiple servlets, enhancing modularity and maintainability.

## Image 2: SPRING SECURITY INTERNAL FLOW

This image outlines how **Spring Security** manages the authentication process internally.

### 1. Steps in the Process:

- **Step 1**: The user submits credentials (e.g., username/password) via a login form.
- **Step 2**: These credentials are intercepted by **Spring Security Filters**, such as `UsernamePasswordAuthenticationFilter``.
- **Step 3**: The filter passes the credentials to the **Authentication Manager**.
- **Step 4**: The Authentication Manager delegates the task of verifying credentials to the appropriate **Authentication Provider**.
- **Steps 5 & 6**: The Authentication Provider interacts with the `UserDetailsService`` to fetch user information and uses the `PasswordEncoder`` to validate the password.
- **Step 7**: Based on the validation result, the Authentication Provider responds back to the Authentication Manager.
- **Step 8**: The Authentication Manager relays the status back to the filter.
- **Step 9**: If authentication is successful, a `SecurityContext`` is updated to store the user's authenticated state.
- **Step 10**: The user is either redirected to the intended resource or shown an error (if authentication fails).

## Image 3: Spring Security Flow Components

This image elaborates on the components involved in the **Spring Security Internal Flow**:

### 1. Spring Security Filters:

- A chain of filters that handle request authentication/authorization.
- Examples include:
  - ``UsernamePasswordAuthenticationFilter``
  - ``BasicAuthenticationFilter``
  - ``SecurityContextPersistenceFilter``

### 2. Authentication:

- Represents the credentials (e.g., username/password) of a user during login.
- Authentication is the core of ensuring a user's identity in Spring Security.

### 3. AuthenticationManager:

- Central interface that handles authentication requests.
- Delegates the actual validation to one or more **Authentication Providers**.

### 4. AuthenticationProvider:

- Contains the core logic for user validation.
- Supports multiple implementations, e.g., database-based authentication, LDAP, or custom logic.

### 5. UserDetailsManager/UserDetailsService:

- Service used to retrieve user information from storage (e.g., database).
- It fetches user details like username, roles, etc.

### 6. PasswordEncoder:

- Encodes and verifies passwords.
- Supports secure hashing algorithms like BCrypt to ensure password security.

### 7. SecurityContext:

- A thread-local storage that holds the authenticated user's details.
- Helps in maintaining user session information across subsequent requests.

---

## How These Images Relate:

- **Servlets and Filters** provide the underlying structure for web applications.
- **Spring Security Filters** enhance this structure by incorporating authentication and authorization workflows into the filter chain.
- The entire process ensures secure access to resources in a web application by validating user credentials and managing security contexts.

Would you like a more specific example or code snippet for any part of this explanation?

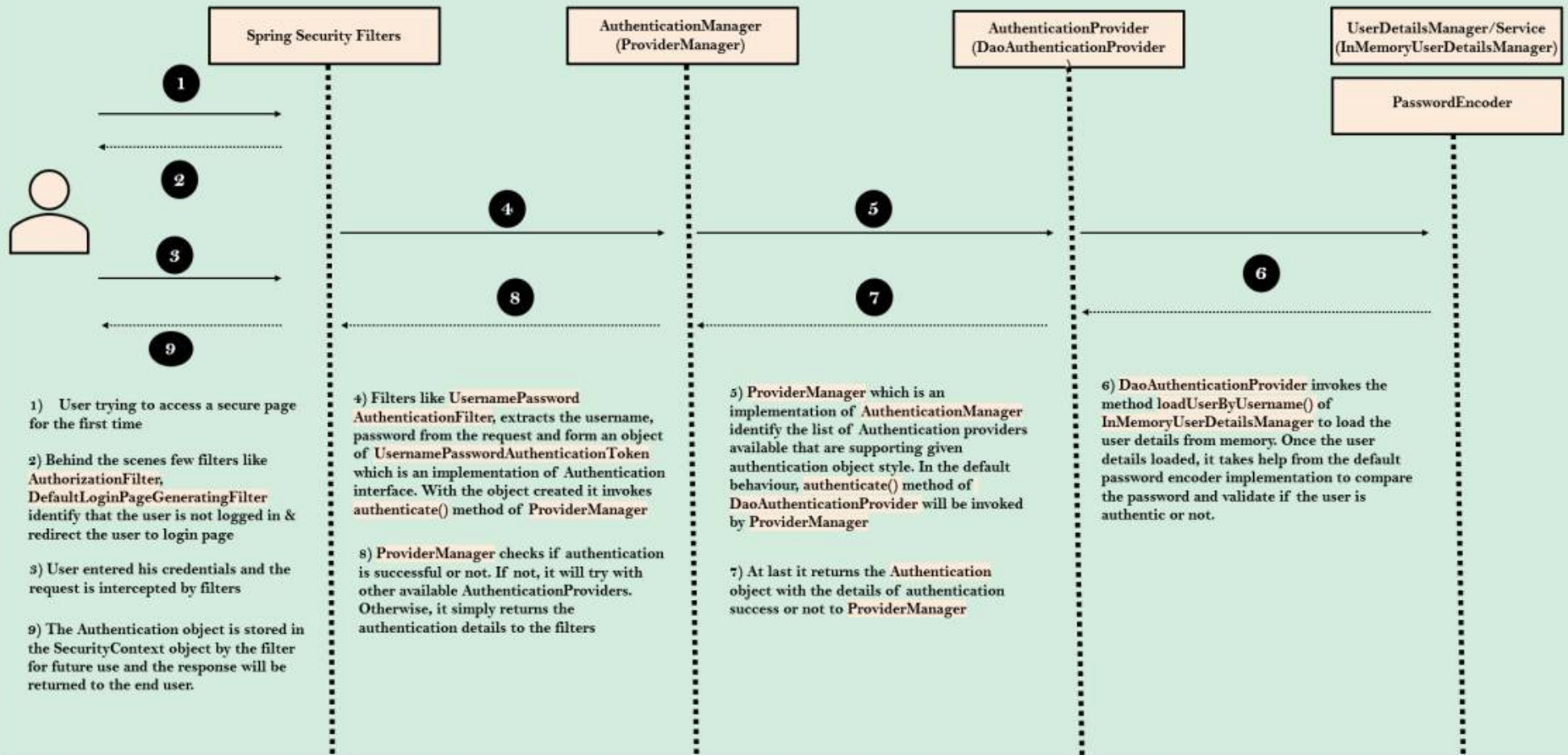




# SEQUENCE FLOW

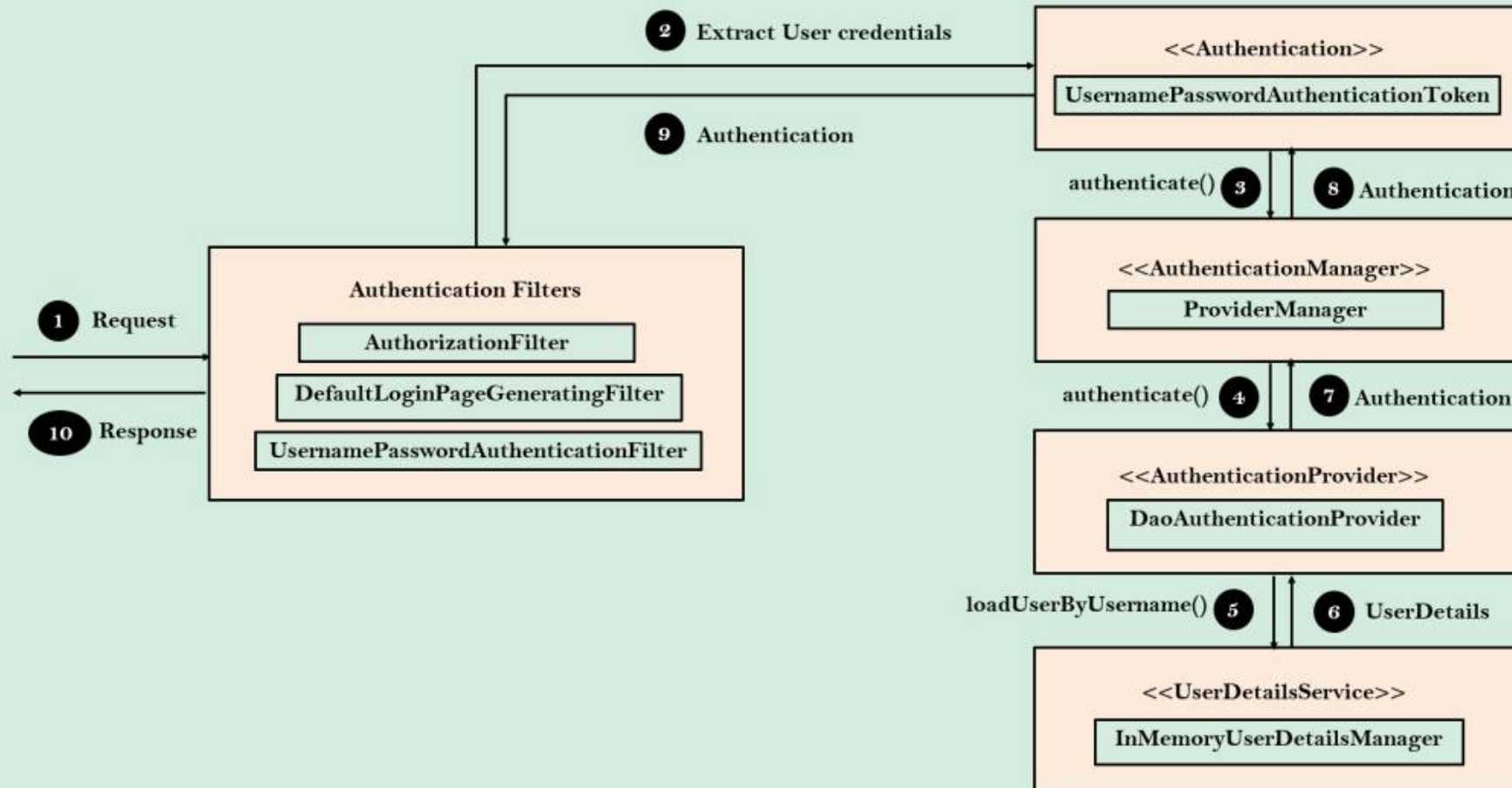
## SPRING SECURITY DEFAULT BEHAVIOUR

easy bytes



## SEQUENCE FLOW SPRING SECURITY DEFAULT BEHAVIOUR

easy  
bytes



Kindly explain the above images in details



The images provide a detailed view of the **sequence flow of Spring Security default behavior**. Here's the explanation:



## Image 1: Sequence Flow - Spring Security Default Behavior

### 1. User Attempts to Access a Secure Page:

- The user tries to access a secure resource within the application.
- If the user is not authenticated, Spring Security's **AuthorizationFilter** identifies this.

### 2. Redirect to Login Page:

- Filters like ``DefaultLoginPageGeneratingFilter`` redirect the user to the login page since they are not authenticated.

### 3. User Submits Credentials:

- The user provides their username and password on the login page.
- The **UsernamePasswordAuthenticationFilter** intercepts this request.

### 4. Token Creation:

- The filter extracts the username and password and creates a ``UsernamePasswordAuthenticationToken``.
- This token is passed to the **ProviderManager** for authentication.

### 5. Authentication Manager:

- The ``ProviderManager``, an implementation of ``AuthenticationManager``, identifies the appropriate **AuthenticationProvider** (e.g., ``DaoAuthenticationProvider``).

### 6. User Details Retrieval:

- The ``DaoAuthenticationProvider`` uses the ``UserDetailsService`` (e.g., ``InMemoryUserDetailsManager``) to fetch user details based on the provided username.
- If the username matches, the details (like roles, password, etc.) are retrieved.

### 7. Password Validation:

- The password provided by the user is compared with the encoded password stored using the **PasswordEncoder**.
- If the password matches, authentication succeeds.

### 8. Authentication Confirmation:

- The ``ProviderManager`` confirms authentication and returns an **Authentication object**.

### 9. Store in SecurityContext:

- The authentication details are stored in the ``SecurityContext`` for subsequent requests.

### 10. Return Response:

- The user is granted access to the originally requested resource, completing the flow.

## Image 2: Visual Flow - Default Spring Security Behavior

This image represents the components visually:

### 1. Request Interception:

- Filters like `AuthorizationFilter`, `DefaultLoginPageGeneratingFilter`, and `UsernamePasswordAuthenticationFilter` handle the request before proceeding to business logic.

### 2. UsernamePasswordAuthenticationToken:

- Once credentials are provided, a `UsernamePasswordAuthenticationToken` is created and passed to the `ProviderManager`.

### 3. ProviderManager:

- Delegates authentication to the `DaoAuthenticationProvider`.

### 4. DaoAuthenticationProvider:

- Uses `UserDetailsService` (e.g., `InMemoryUserDetailsManager`) to fetch user information.
- Calls the `PasswordEncoder` to validate the password.

### 5. UserDetailsService:

- Loads user details using the `loadUserByUsername` method.

### 6. Authentication Result:

- If valid, an `Authentication` object is created and stored in the `SecurityContext`.

### 7. Response Sent:

- The user is authenticated and redirected to the requested resource.

## Key Components:

### 1. Filters:

- Intercepts requests and redirects unauthorized users to the login page.
- Examples: `AuthorizationFilter`, `UsernamePasswordAuthenticationFilter`.

### 2. AuthenticationManager (ProviderManager):

- Handles delegation to multiple `AuthenticationProvider` implementations.

### 3. AuthenticationProvider:

- Validates credentials and fetches user details.

#### 4. UserDetailsService:

- Loads user details based on the username.

#### 5. PasswordEncoder:

- Ensures passwords are securely hashed and validated.

This flow ensures robust security in Java-based web applications by systematically validating user credentials and securing sensitive resources. Let me know if you'd like code snippets or further details!



What are the different types of Filters in spring Security. Kindly explain it in details along with its purpose



In **Spring Security**, filters are a crucial part of the security infrastructure. They intercept HTTP requests and perform various pre-processing or post-processing tasks, such as authentication, authorization, and session management. Below is a detailed explanation of the **different types of filters in Spring Security** and their purposes:

#### 1. `SecurityContextPersistenceFilter`

- **Purpose:**
  - Manages the `SecurityContext` for each request.
  - Ensures that the `SecurityContext` is loaded at the beginning of a request and saved at the end.
- **Key Features:**
  - Retrieves the `SecurityContext` from the `SecurityContextRepository` at the start of a request.
  - Saves the updated `SecurityContext` back to the repository after the request is processed.
- **Example:** Used to store the security context in a session for subsequent requests.

#### 2. `UsernamePasswordAuthenticationFilter`

- **Purpose:**
  - Handles form-based authentication.

- Extracts username and password from an HTTP request and attempts to authenticate the user.
  - **Key Features:**
    - Creates a ``UsernamePasswordAuthenticationToken``.
    - Delegates authentication to the ``AuthenticationManager``.
    - If authentication is successful, stores the result in the ``SecurityContext``.
  - **Example:** Triggered when a user submits a login form with credentials.
- 

### 3. ``BasicAuthenticationFilter``

- **Purpose:**
    - Handles HTTP Basic Authentication.
    - Extracts credentials from the ``Authorization`` header of the HTTP request.
  - **Key Features:**
    - Checks if the ``Authorization`` header contains a username and password in Base64 encoding.
    - Delegates authentication to the ``AuthenticationManager``.
  - **Example:** Used for APIs where credentials are passed directly in the request header.
- 

### 4. ``BearerTokenAuthenticationFilter``

- **Purpose:**
    - Handles authentication based on OAuth2 Bearer tokens.
    - Extracts the token from the ``Authorization`` header or other configured locations.
  - **Key Features:**
    - Passes the token to the ``AuthenticationManager`` for validation.
    - Typically used with OAuth2 resource servers.
  - **Example:** Used in REST APIs secured with OAuth2.
- 

### 5. ``ExceptionHandlerFilter``

- **Purpose:**
  - Translates Spring Security-specific exceptions into appropriate HTTP responses.
- **Key Features:**
  - Handles exceptions like ``AuthenticationException`` and ``AccessDeniedException``.

- Redirects unauthenticated users to a login page or sends a `401 Unauthorized` response for APIs.
  - **Example:** Ensures proper handling of unauthorized access attempts.
- 

## 6. `FilterSecurityInterceptor`

- **Purpose:**
    - Handles authorization decisions for requests.
    - Enforces access control rules specified in the security configuration.
  - **Key Features:**
    - Intercepts requests and invokes the `AccessDecisionManager`.
    - Checks if the authenticated user has the required permissions to access the requested resource.
  - **Example:** Used to secure URLs based on roles or authorities.
- 

## 7. `CsrfFilter`

- **Purpose:**
    - Protects against Cross-Site Request Forgery (CSRF) attacks.
  - **Key Features:**
    - Validates the presence of a CSRF token in requests that modify state (e.g., POST, PUT, DELETE).
    - Rejects requests without a valid CSRF token.
  - **Example:** Ensures that only requests originating from the same domain are processed.
- 

## 8. `SessionManagementFilter`

- **Purpose:**
    - Manages the user's session in the application.
  - **Key Features:**
    - Prevents session fixation attacks by creating a new session upon authentication.
    - Limits the number of concurrent sessions for a user.
    - Handles session expiration and invalidation.
  - **Example:** Ensures secure session handling in web applications.
-

## 9. `AnonymousAuthenticationFilter`

- **Purpose:**
    - Assigns an anonymous `Authentication` object to requests that are not authenticated.
  - **Key Features:**
    - Ensures that unauthenticated requests are handled gracefully.
    - Assigns default roles or permissions to anonymous users.
  - **Example:** Provides access to public resources without requiring authentication.
- 

## 10. `LogoutFilter`

- **Purpose:**
    - Handles user logout functionality.
  - **Key Features:**
    - Invalidates the user's session.
    - Clears the `SecurityContext`.
    - Redirects the user to a post-logout URL.
  - **Example:** Ensures a clean logout process.
- 

## 11. `OAuth2AuthorizationRequestRedirectFilter`

- **Purpose:**
    - Handles OAuth2 authorization requests.
  - **Key Features:**
    - Redirects the user to the OAuth2 authorization server.
    - Used for social login (e.g., Google, Facebook).
  - **Example:** Triggers the OAuth2 login flow.
- 

## 12. `CorsFilter`

- **Purpose:**
  - Manages Cross-Origin Resource Sharing (CORS) settings.
- **Key Features:**



- Configures which domains are allowed to send requests to the server.
  - Adds necessary headers to HTTP responses.
  - **Example:** Used in APIs accessed from different domains.
- 

### 13. `RequestCacheAwareFilter`

- **Purpose:**
    - Handles cached requests that require authentication.
  - **Key Features:**
    - Saves the original request before redirecting to the login page.
    - Restores the request after successful authentication.
  - **Example:** Ensures the user is redirected back to the original URL after login.
- 

### 14. `AuthenticationTokenProcessingFilter`

- **Purpose:**
    - Processes authentication based on tokens (e.g., JWT).
  - **Key Features:**
    - Extracts the token from the request header.
    - Validates the token and populates the `SecurityContext`.
  - **Example:** Used in stateless authentication scenarios.
- 

### How the Filters Work Together:

- Filters in Spring Security are executed in a specific **chain** defined by the framework.
- Each filter has a distinct role and works sequentially to process requests.
- You can customize the filter chain to add or modify filters as needed.

Would you like a deeper explanation of any specific filter or example implementation?