

# Spring Security Flow Explained

explain this in details so that i am able to understand it well and do not have doubt on this

explain in depth of this so that i donot have any doubts on this and have great knowlege of this screenshot

Great question, Piyush 🙌

This diagram is **one of the most important mental models in Spring Security**.

If you understand this **once, deeply**, 80% of Spring Security confusion disappears.

I'll explain this **slowly, layer by layer, step by step**, using **real flow**, **why each component exists**, and **what happens internally** — exactly matching your screenshot.

## 0 Big Picture (Before Details)

What is Spring Security doing here?

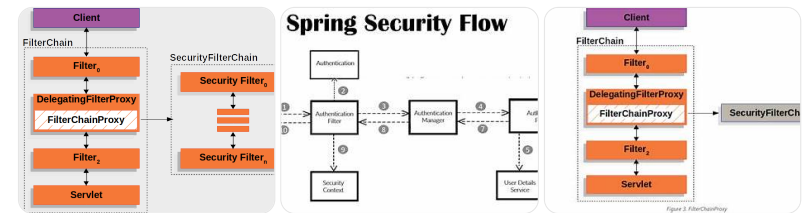
👉 For **every HTTP request**, Spring Security decides:

- Who is the user?
- Is the user authenticated?
- Is the user authorized to access this endpoint?

This is done using **filters**, **authentication managers**, **providers**, and **security context**.

Your diagram shows the **complete authentication flow**.

## 1 Request enters the server (Tomcat → Servlet Filter Chain)



## Step 1: Incoming request

arduino

Browser / Postman / Frontend

Tomcat Server

Tomcat is just a **web server**.

It does **not** know anything about Spring Security.

## Servlet Filter Chain (Generic Filters)

diff

Servlet filter chain:

- CharacterEncodingFilter
- HiddenHttpMethodFilter
- RequestContextFilter
- ForwardedHeaderFilter
- CorsFilter

These are **normal servlet filters**, NOT security filters.

✦ Important:

- Security is not here yet
- These filters prepare the request (encoding, headers, CORS, etc.)

## 2 FilterChainProxy – Entry point of Spring Security

## This is the MOST IMPORTANT box

markdown

```
ServletFilterChain
  ↓
  FilterChainProxy ← Spring Security starts here
```

### What is `FilterChainProxy` ?

- It is **Spring Security's gateway**
- Registered as **one servlet filter**
- Delegates request to **Security Filter Chain**

✦ Think of it as:

"If security is enabled, every request must pass through me."

## 3 Security Filter Chain (Heart of Spring Security)

diff

```
Security Filter Chain:
- WebAsyncManagerIntegrationFilter
- SecurityContextPersistenceFilter
- HeaderWriterFilter
- LogoutFilter
- UsernamePasswordAuthenticationFilter
- ...
```

### Important idea

Spring Security is **FILTER-BASED**, not controller-based.

Each filter has **one responsibility**.

### 🔑 Key filters you MUST understand

#### 1 `SecurityContextPersistenceFilter`

- Loads `SecurityContext` at the start

- Saves it at the end
- ✦ If user was already logged in:
- It loads authentication from session/JWT

#### 2 `UsernamePasswordAuthenticationFilter`

👉 **THIS is where login happens**

Triggered when:

bash

```
POST /login
```

It:

- Extracts username & password
- Creates an `Authentication` object (unauthenticated)
- Passes it to `AuthenticationManager`

## 4 AuthenticationManager (The Boss)

markdown

```
UsernamePasswordAuthenticationFilter
  ↓
  AuthenticationManager
```

### What is AuthenticationManager?

- It does **not authenticate itself**
- It **delegates** authentication

✦ Think of it as:

"I don't know how to authenticate, I know WHO can authenticate."

## 5 AuthenticationProvider (Actual Authentication)

markdown

```

AuthenticationManager
  ↓
AuthenticationProvider

```

Examples:

- `DaoAuthenticationProvider`
- `InMemoryAuthenticationProvider`
- `OAuth2AuthenticationProvider`

How selection happens?

Each provider has:

java

```
boolean supports(Authentication auth)
```

The manager picks the matching provider.

## 6 DaoAuthenticationProvider (Most common)

markdown

```

DaoAuthenticationProvider
  ↓      ↓
UserDetailsService PasswordEncoder

```

Step-by-step:

### 1 Load user

java

```
UserDetails user = userDetailsService.loadUserByUsername(username);
```

- Fetches user from DB
- Returns:
  - username
  - password (hashed)

- roles/authorities

### 2 Verify password

java

```
passwordEncoder.matches(rawPassword, encodedPassword);
```

✓ If match → success

✗ If not → `BadCredentialsException`

## 7 Authentication success flow (Very Important)

After successful authentication:

pgsql

```

AuthenticationProvider
  ↓
AuthenticationManager
  ↓
Security Filter Chain

```

What is returned?

A FULLY AUTHENTICATED Authentication object:

- principal (UserDetails)
- authorities
- authenticated = true

## 8 SecurityContextHolder (Logged-in user stored)

markdown

```

SecurityContextHolder
  ↓
SecurityContext

```

## What happens here?

Spring Security stores the authenticated user in:

```
java
SecurityContextHolder.getContext().setAuthentication(auth);
```

✦ This means:

- User is now **logged in**
- Available anywhere in the app

Example:

```
java
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
```

## 9 DispatcherServlet & Controllers

```
kotlin
SecurityContext is set
    ↓
DispatcherServlet
    ↓
@Controller / @RestController
```

Now:

- Controllers can run
- Authorization checks happen:
  - `@PreAuthorize`
  - `hasRole()`
  - URL-based security

## Request Flow Summary (One Line)

```
rust
```

```
Request
→ Tomcat
→ Servlet Filters
→ FilterChainProxy
→ Security Filters
→ AuthenticationManager
→ AuthenticationProvider
→ SecurityContextHolder
→ Controller
```

## 🔒 Why this design is powerful

### ✅ Pluggable authentication

- DB
- LDAP
- OAuth
- JWT

### ✅ Stateless or Stateful

- Session-based
- JWT-based

### ✅ Clean separation

- Filters → request handling
- Providers → authentication logic

## 💬 Common Doubts (Answered)

### ? Why filters, not controllers?

Security must run **before controllers**, for every request.

### ? Why AuthenticationManager if Provider exists?

- Manager = coordinator
- Provider = executor

### ? Where is authorization?

- After authentication
- In filters like `FilterSecurityInterceptor`

### ? Why SecurityContextHolder?

- Thread-safe storage of authenticated user
- Accessible anywhere

### ✓ Final Mental Model (Remember This)

Spring Security is a filter-based authentication pipeline that delegates authentication to providers and stores the authenticated user in a thread-local security context before hitting controllers.

If you want, next I can:

- Explain JWT flow using the same diagram
- Convert this into real Spring Boot code
- Explain OAuth2 flow visually
- Explain multiple `SecurityFilterChain` with `@Order`

Just tell me 🙌

explain more in depth of this

Perfect 🙌

Now we'll go one level deeper than most tutorials ever go.

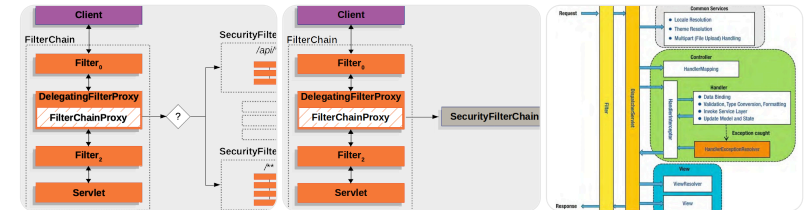
I'll explain what happens internally, why each class exists, how data moves, what objects look like in memory, and how this changes for JWT / sessions — all mapped exactly to your screenshot.

Read this slowly once or twice. After this, Spring Security will feel logical, not magical.

## 1 Two different filter chains (MOST IMPORTANT CLARITY)

Your screenshot shows **two filter chains**. Many people confuse them.

### Servlet Filter Chain (Container-level)



This is **NOT** Spring Security.

Examples:

- `CharacterEncodingFilter`
- `CorsFilter`
- `HiddenHttpMethodFilter`

★ Facts:

- Managed by Tomcat
- Runs for **every** request
- Knows nothing about authentication or users

### Security Filter Chain (Spring Security-level)

This starts **ONLY** when request reaches `FilterChainProxy`.

★ Key rule:

Spring Security = just **ONE** servlet filter (`FilterChainProxy`)  
Inside it → many security filters

## 2 FilterChainProxy (Why it exists)

Why not directly register security filters?

Because:

- You may have **multiple security configurations**
- Different URLs → different security rules

Example:

```
java
SecurityFilterChain apiChain // /api/**
SecurityFilterChain webChain // /login, /logout
```

✦ **FilterChainProxy**:

- Matches request URL
- Chooses correct **SecurityFilterChain**
- Executes filters in **exact order**

### 3 Security Filter Chain – Order matters (A LOT)

Spring Security filters are **ordered deliberately**.

Example simplified order:

```
markdown
1. SecurityContextPersistenceFilter
2. UsernamePasswordAuthenticationFilter
3. Authorization Filters
4. ExceptionTranslationFilter
```

✦ If order breaks → security breaks.

### 4 SecurityContextPersistenceFilter (Deep Internals)

This filter runs **TWICE**:

- At request start
- At request end

At request **START**:

```
java
```

```
SecurityContext context = repository.loadContext(request);
SecurityContextHolder.setContext(context);
```

Where does it load from?

- Session (STATEFUL)
- JWT (STATELESS)
- Empty context (new user)

✦ This answers:

“How does Spring know I’m already logged in?”

At request **END**:

```
java
repository.saveContext(context, request, response);
SecurityContextHolder.clearContext();
```

✦ Why clear?

- Thread reuse in Tomcat
- Prevent user leak between requests

### 5 UsernamePasswordAuthenticationFilter (Login engine)

Triggered **ONLY** when:

```
bash
POST /login
```

Internally it does:

```
java
String username = request.getParameter("username");
String password = request.getParameter("password");

Authentication authRequest =
```

```
new UsernamePasswordAuthenticationToken(username, password);
```

```
authenticationManager.authenticate(authRequest);
```

#### ✦ Important:

- This token is **NOT** authenticated
- `authenticated = false`

## 6 Authentication object (Understand this clearly)

Authentication has 4 core things:

```
java

principal    → UserDetails / username
credentials → password
authorities  → roles
authenticated → true/false
```

Before authentication:

```
ini

principal = "piyush"
credentials = "rawPassword"
authorities = null
authenticated = false
```

After authentication:

```
ini

principal = UserDetails object
credentials = null
authorities = [ROLE_USER]
authenticated = true
```

## 7 AuthenticationManager (Coordinator, NOT worker)

Internally:

```
java

for (AuthenticationProvider provider : providers) {
    if (provider.supports(auth.getClass())) {
        return provider.authenticate(auth);
    }
}
throw new ProviderNotFoundException();
```

#### ✦ AuthenticationManager:

- Does **NOT** know DB
- Does **NOT** know password logic
- Just delegates

## 8 AuthenticationProvider (Actual brain)

DaoAuthenticationProvider (Most used)

It performs 3 critical checks:

### ♦ Step 1: Load user

```
java

UserDetails user =
    userDetailsService.loadUserByUsername(username);
```

If not found:

```
java

UsernameNotFoundException
```

### ♦ Step 2: Password check

```
java
```

```
passwordEncoder.matches(raw, encoded);
```

✖ mismatch → `BadCredentialsException`

### ◆ Step 3: Create authenticated token

```
java
return new UsernamePasswordAuthenticationToken(
    userDetails,
    null,
    userDetails.getAuthorities()
);
```

✧ authenticated = true

## 9 Back to Security Filters (Post-authentication)

Once authentication succeeds:

```
nginx
AuthenticationProvider
→ AuthenticationManager
→ UsernamePasswordAuthenticationFilter
```

Now this line executes:

```
java
SecurityContextHolder.getContext().setAuthentication(auth);
```

✧ This is the LOGIN MOMENT

## 10 SecurityContextHolder (ThreadLocal magic)

What is it?

- Uses `ThreadLocal`

- One user per request thread
- ✧ Why ThreadLocal?
  - Fast
  - No passing user object everywhere
  - Safe per request

## 1 1 Authorization phase (Often ignored but crucial)

After authentication:

- `FilterSecurityInterceptor`
- `AuthorizationFilter`

They check:

```
java
hasRole("ADMIN")
hasAuthority("READ")
```

If fails:

```
java
AccessDeniedException → 403
```

## 1 2 DispatcherServlet & Controllers

Now request reaches:

```
java
@RestController
@GetMapping("/profile")
```

You can access user:

```
java
```



```
Authentication auth =
    SecurityContextHolder.getContext().getAuthentication();
```

Or:

```
java

Principal principal
```

### 1 3 Stateful vs Stateless (CRITICAL DIFFERENCE)

#### Session-based:

- SecurityContext saved in HTTP Session
- Cookie → JSESSIONID

#### JWT-based:

- Token sent every request
- Custom filter reads JWT
- Sets SecurityContext manually

✦ But core flow remains SAME.

### 1 4 Why Spring Security is designed this way

- ✓ Filter-based → secure everything
- ✓ Delegation → extensible
- ✓ Providers → pluggable auth
- ✓ Context → centralized user access

### 🧠 FINAL SUPER-STRONG MENTAL MODEL

Spring Security intercepts every request using filters, delegates authentication to providers, stores the authenticated user in a thread-local context, then enforces authorization before controllers execute.

If you want next, I can:

- Map **JWT flow** on same diagram
- Write **custom AuthenticationProvider**
- Explain **multiple SecurityFilterChain** with **@Order**
- Explain **OAuth2 login** using same lifecycle
- Convert this into **debug-level logs walkthrough**

Just tell me 🔥

explain this in details so that i am ablt to understand it well and do not have doubts in this

Servlet filter chain:

- CharacterEncodingFilter
- HiddenHttpMethodFilter
- RequestContextFilter
- ForwardedHeaderFilter
- CorsFilter

Excellent — this is a **very important foundation topic**, and many Spring devs skip it, which later causes confusion in **Spring Security, REST APIs, CORS, proxies, and HTTP methods**.

I'll explain **each filter in depth**, in **simple language**, with:

- **What problem it solves**
- **Why it exists**
- **What happens if it's missing**
- **Real examples**
- **Relation to Spring Boot & Spring Security**

No shortcuts. No assumptions.

### ♦ What is the *Servlet Filter Chain* (First, clear this)

Before Spring Security even starts, **every HTTP request** goes through:

SCSS

```

Tomcat
↓
Servlet Filter Chain ← (what you asked about)
↓
FilterChainProxy (Spring Security)

```

- ✦ These filters are **NOT Spring Security filters**
- ✦ They are **Servlet-level infrastructure filters**

Think of them as **request preparation filters**.

## 1 CharacterEncodingFilter

### ♦ What problem does it solve?

By default:

- HTTP requests **may not use UTF-8**
- This causes:
  - Broken special characters
  - Wrong data in forms
  - Corrupted JSON / request body

Example problem:

```

pgsql

Name = "Piyush गाँव"
Stored as = "Piyush ????"

```

### ♦ What this filter does

It forces encoding for:

- Request
- Response

Typically:

```
java
```

```
UTF-8
```

Internally:

```

java

request.setCharacterEncoding("UTF-8");
response.setCharacterEncoding("UTF-8");

```

### ♦ Why it runs FIRST

Encoding must be set **before reading request data**.

If request body is already read → encoding change is useless.

### ♦ What happens if you remove it?

- ✗ Unicode breaks
- ✗ International characters fail
- ✗ JSON parsing issues

### ♦ When you really feel its importance

- Multilingual apps
- JSON APIs
- Form submissions
- Mobile apps

## 2 HiddenHttpMethodFilter

### ♦ What problem does it solve?

HTML forms support only:

```
sql
```

```
GET, POST
```

But REST APIs need:

```
sql
```

```
PUT, DELETE, PATCH
```

### ◆ How this filter works

You send:

```
html

<form method="POST">
  <input type="hidden" name="_method" value="DELETE"/>
</form>
```

This filter:

- Reads `_method`
- Converts request into:

```
bash

DELETE /resource
```

Internally:

```
java

if (_method exists) {
    request.setMethod("DELETE");
}
```

### ◆ Why it exists (real-world reason)

Older browsers + HTML limitations.

### ◆ Does it affect APIs?

✗ No (Postman / frontend frameworks don't need it)

✓ Mainly useful for:

- Thymeleaf
- JSP
- MVC apps

### ◆ What if removed?

- DELETE/PUT won't work from forms
- REST APIs unaffected

## 3 RequestContextFilter

### ◆ What problem does it solve?

Spring needs request-related data available everywhere, even:

- Services
- Repositories
- Async threads

But:

- Servlet request is **thread-bound**
- Spring beans don't get request directly

### ◆ What this filter does

It binds request & response to Spring context.

Internally:

```
java

RequestContextHolder.setRequestAttributes(...)
```

### ◆ What does this enable?

Now you can do:

```
java
ContextHolder.currentRequestAttributes()
```

Or:

```
java
@Autowired HttpServletRequest
```

Even outside controllers.

### ◆ Why Spring Security needs this

Security filters need:

- Request
- Headers
- Session
- IP address

This filter makes it accessible.

### ◆ If missing?

- ✗ Request-scoped beans fail
- ✗ Async processing breaks
- ✗ Security context access issues

### 4 ForwardedHeaderFilter

#### ◆ This one is VERY IMPORTANT in real production

#### ◆ What problem does it solve?

In real apps:

```
pgsql
Client → Load Balancer → Reverse Proxy → App
```

Original request:

```
arduino
https://myapp.com
```

But app sees:

```
arduino
http://localhost:8080
```

- ✗ Wrong protocol
- ✗ Wrong IP
- ✗ Broken redirects
- ✗ OAuth failures

### ◆ What this filter does

Reads headers like:

```
css
X-Forwarded-For
X-Forwarded-Proto
X-Forwarded-Host
```

And fixes request info.

Example:

```
h
X-Forwarded-Proto: https
```

App now sees:

```
arduino
```

```
request.isSecure() = true
```

### ◆ Why Spring Security needs it

- OAuth redirect URIs
- HTTPS enforcement
- IP-based security
- CORS

### ◆ If missing?

- ✗ OAuth redirect mismatch
- ✗ Wrong URL generation
- ✗ Security issues

## 5 CorsFilter

### ◆ What problem does it solve?

Browsers block:

```
less

Frontend (localhost:3000)
→ Backend (localhost:8080)
```

Unless CORS headers exist.

### ◆ What is CORS?

Cross-Origin Resource Sharing

Controls:

- Which origins can call API
- Which headers allowed

- Which methods allowed

### ◆ What this filter does

Adds response headers:

```
http

Access-Control-Allow-Origin
Access-Control-Allow-Methods
Access-Control-Allow-Headers
```

Also handles:

```
scss

OPTIONS (preflight)
```

### ◆ Why it runs BEFORE security

Browser sends:

```
bash

OPTIONS /api
```

This is NOT a real request.

If security blocks it:

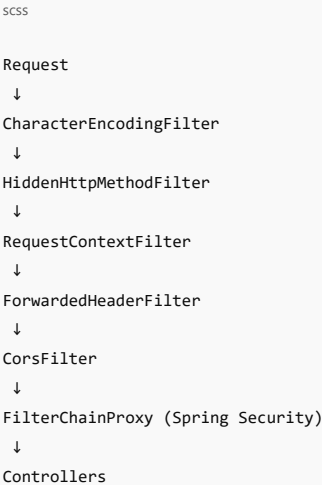
- ✗ Frontend breaks

So CORS must allow it FIRST.

### ◆ If missing?

- ✗ Frontend cannot call backend
- ✗ Browser errors
- ✗ API works in Postman but fails in browser

🔄 Full Execution Order (Mental Model)



🧠 FINAL CLEAR SUMMARY (No Doubts Version)

Filter	Purpose
CharacterEncodingFilter	Fix encoding (UTF-8)
HiddenHttpMethodFilter	Enable PUT/DELETE via forms
RequestContextFilter	Bind request to Spring context
ForwardedHeaderFilter	Fix proxy / load balancer issues
CorsFilter	Allow frontend ↔ backend

🔥 Golden Rule to Remember

Servlet filters prepare the request.  
Spring Security filters secure the request.

If you want next, I can:

- Show **real debug logs**
- Explain **what happens when a request fails**
- Explain **OPTIONS request flow**
- Connect this with **JWT authentication**
- Show **production vs local setup**

Just tell me 🙏

explain this in details so that i am able to understand it well and do not have doubts in this

Security Filter Chain:  
- WebAsyncManagerIntegrationFilter  
- SecurityContextPersistenceFilter  
- HeaderWriterFilter  
- LogoutFilter  
- UsernamePasswordAuthenticationFilter  
- ...

Perfect 🙌

Now we are entering the **core of Spring Security**.

I'll explain **each filter in the Security Filter Chain in very deep detail**, step-by-step, with:

- **Why the filter exists**
- **What problem it solves**
- **What exactly happens internally**
- **What breaks if it's missing**
- **How it fits in the full request lifecycle**

After this, you should be able to **mentally simulate Spring Security for any request**.

## 🔒 What is the *Security Filter Chain* (First, crystal clear)

Before controllers are called, Spring Security runs a **chain of security filters**.

```
pgsql
Request
↓
Servlet Filters (encoding, CORS, etc.)
↓
FilterChainProxy
↓
Security Filter Chain ← YOU ARE HERE
↓
DispatcherServlet
↓
Controllers
```

🔑 Key rule:

**Spring Security is 100% filter-based**

No controller, no annotation, no interceptor runs **before** these filters.

### 1 WebAsyncManagerIntegrationFilter

#### ♦ Why this filter exists

Spring supports **async requests**, like:

- `@Async`
- `Callable`
- `DeferredResult`
- `CompletableFuture`

Problem:

Spring Security stores authentication in a **ThreadLocal**

Async execution uses **different threads** ❌

So authentication would be **lost** ❌

#### ♦ What this filter does

It copies the **SecurityContext** from:

- Main request thread
- To async execution threads

Internally (simplified):

```
java
SecurityContext context = SecurityContextHolder.getContext();
WebAsyncManager.registerCallableInterceptor(context);
```

#### ♦ Real example

```
java
@GetMapping("/async")
public Callable<String> test() {
    return () -> {
        Authentication auth =
            SecurityContextHolder.getContext().getAuthentication();
        return auth.getName();
    };
}
```

Without this filter:

❌ `auth == null`

With this filter:

✅ Authentication available

#### ♦ If this filter is missing

- Async controllers break
- `@Async` loses user info
- Security bugs appear silently

## 2 SecurityContextPersistenceFilter (MOST IMPORTANT FILTER)

🔥 This is the backbone of Spring Security

### ♦ What problem it solves

Spring Security needs:

- User authentication available **during request**
- Persisted **between requests**

But:

- HTTP is stateless
- Threads are reused

### ♦ What this filter does (VERY IMPORTANT)

This filter runs **TWICE**:

#### ▶ At request START

```
java
SecurityContext context = repository.loadContext(request);
SecurityContextHolder.setContext(context);
```

Loads authentication from:

- HTTP Session (stateful)
- JWT / token (stateless)
- Empty context (anonymous user)

#### ▶ At request END

```
java
repository.saveContext(context, request, response);
SecurityContextHolder.clearContext();
```

### ✦ Clearing context is **critical**

- Prevents user leak between threads
- Avoids severe security bugs

### ♦ If this filter is missing

- ✗ User never stays logged in
- ✗ SecurityContext always empty
- ✗ Authentication lost every request

### ♦ Why it runs early

Other filters **need authentication info**:

- Authorization
- Logout
- Header decisions

## 3 HeaderWriterFilter

### ♦ What problem it solves

Modern security requires HTTP security headers.

Browsers protect apps using headers like:

- X-Frame-Options
- X-Content-Type-Options
- Strict-Transport-Security
- Content-Security-Policy

### ♦ What this filter does

It **adds security headers** to every response.

Example:



```
h
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
```

### ◆ Why this is important

Prevents:

- Clickjacking
- MIME sniffing
- XSS
- Downgrade attacks

### ◆ If missing

- ✗ App vulnerable to browser-based attacks
- ✗ Security audits fail
- ✗ Production security weak

### ◆ Important note

This filter:

- Does **NOT** authenticate
- Does **NOT** authorize
- Only **hardens** responses

### 4 LogoutFilter

### ◆ What problem it solves

Logout is **not just** deleting a cookie.

Spring Security must:

- Clear SecurityContext
- Invalidate session

- Delete cookies
- Trigger logout handlers

### ◆ When this filter activates

Default:

```
bash
POST /logout
```

### ◆ What happens internally

- 1 Match logout URL
- 2 Clear `SecurityContextHolder`
- 3 Invalidate HTTP session
- 4 Delete cookies (JSESSIONID)
- 5 Redirect or return response

Internally:

```
java
SecurityContextHolder.clearContext();
session.invalidate();
```

### ◆ If missing

- ✗ User never logs out properly
- ✗ Session fixation risks
- ✗ Old authentication remains

### ◆ Important

Logout happens **before authentication filters**

Because logout does **not require re-authentication**

## 5 UsernamePasswordAuthenticationFilter (LOGIN ENGINE)

🔥 This filter performs LOGIN

### ◆ When this filter runs

Only when:

```
bash
POST /login
```

Other requests:

➡ This filter is skipped

### ◆ What this filter does (FULL FLOW)

#### Step 1: Extract credentials

```
java
username = request.getParameter("username");
password = request.getParameter("password");
```

#### Step 2: Create Authentication object (unauthenticated)

```
java
new UsernamePasswordAuthenticationToken(username, password);
```

✦ At this point:

```
ini
authenticated = false
```

#### Step 3: Delegate to AuthenticationManager

```
java
authenticationManager.authenticate(authRequest);
```

#### Step 4: On success

```
java
SecurityContextHolder.getContext()
    .setAuthentication(authenticatedAuth);
```

#### Step 5: Continue filter chain

User is now logged in ✅

### ◆ If authentication fails

- `BadCredentialsException`
- `AuthenticationException`
- Error response / redirect

### ◆ If this filter is missing

- ✗ Form login does not work
- ✗ `/login` does nothing
- ✗ No username-password auth

## 6 Other filters ( . . . ) – What they generally do

These include:

- `AnonymousAuthenticationFilter`

- `ExceptionTranslationFilter`
- `FilterSecurityInterceptor`

#### Their responsibilities:

- Handle anonymous users
- Convert exceptions to 401/403
- Enforce authorization rules

✳ Authentication happens **before** authorization.

### FULL EXECUTION ORDER (CLEAR PICTURE)

```
sql

Request
↓
WebAsyncManagerIntegrationFilter
↓
SecurityContextPersistenceFilter
↓
HeaderWriterFilter
↓
LogoutFilter
↓
UsernamePasswordAuthenticationFilter
↓
Authorization Filters
↓
DispatcherServlet
↓
Controller
```

### FINAL NO-DOUBT MENTAL MODEL

Spring Security loads the user, keeps the user safe across threads, authenticates on login, applies security headers, handles logout, enforces authorization, and only then allows the request to reach controllers.

### One-line purpose of each filter

Filter	Purpose
<code>WebAsyncManagerIntegrationFilter</code>	Preserve auth in async
<code>SecurityContextPersistenceFilter</code>	Load & save user
<code>HeaderWriterFilter</code>	Add security headers
<code>LogoutFilter</code>	Perform logout
<code>UsernamePasswordAuthenticationFilter</code>	Perform login

If you want next, I can:

- Explain **authorization filters** in same depth
- Walk through **JWT login flow** step-by-step
- Show **debug logs** for each filter
- Explain **custom authentication filter**
- Explain **multiple SecurityFilterChain** with `@Order`

Just tell me 🙌