



I want you to act as a Spring boot security expert and analyze the image I have uploaded and create notes of each steps to make it understand. Follow the sequence, Don't get or make me confuse, and if you don't know anything just admit instead of wrong of vague information



Good diagram. It's **mostly correct**, but a few labels are sloppy and one term is outright wrong. I'll walk you through it **step-by-step in the exact sequence shown**, cleanly, without mixing concepts. I'll also **call out mistakes** so you don't memorize garbage.

Spring Security Request Flow — Step-by-Step (Based on Your Image)

STEP 1: Incoming HTTP Request

What happens

- A client sends an HTTP request (browser, Postman, frontend app).
- This request **does NOT go directly to your controller**.

Reality check

- If you think `@Controller` or `@RestController` is first → **wrong**.
- Spring Security always gets first crack.

STEP 2: Request enters the Security Filter Chain

What happens

- The request enters Spring Security's **Filter Chain**.
- This is a chain of servlet filters executed **in a strict order**.

Important filters shown in your image

- `SecurityContextPersistenceFilter`
- `HeaderWriterFilter`
- `LogoutFilter`
- `UsernamePasswordAuthenticationFilter`
- etc.

Hard truth

- These are **Servlet Filters**, not Spring MVC interceptors.
- They execute **before DispatcherServlet**.

STEP 3: Authentication is delegated to AuthenticationManager

Your diagram says: *"authentication reprehensibility is delegated"*

✗ **This wording is wrong.**

✓ **Correct term: authentication responsibility is delegated**

What actually happens

- A filter (usually `UsernamePasswordAuthenticationFilter`) extracts credentials.
- It creates an `Authentication` object (unauthenticated).
- It calls:

```
java
authenticationManager.authenticate(authentication)
```



Key point

- Filters **do not authenticate users themselves**.
- They delegate.

STEP 4: AuthenticationManager chooses the AuthenticationProvider

What happens

- `AuthenticationManager` (usually `ProviderManager`) loops through registered `AuthenticationProvider`s.
- It picks the **first provider that supports the Authentication type**.

Examples shown

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

Hard rule

- If no provider supports the token → authentication fails.

- Order of providers **matters**.

STEP 5: AuthenticationProvider performs authentication

Focus: DaoAuthenticationProvider (shown on right)

What it does

1. Calls `UserDetailsService.loadUserByUsername()`
2. Gets `UserDetails`
3. Uses `PasswordEncoder` to compare passwords

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

 Copy code

Critical understanding

- `UserDetailsService` **does NOT** validate passwords
- `PasswordEncoder` **does NOT** fetch users
- Each has a single responsibility

If you mix this up in interviews → instant red flag.

STEP 6: AuthenticationProvider returns an authenticated Authentication

What happens

- If credentials are valid:
 - Provider returns a **fully authenticated** Authentication object
 - `isAuthenticated = true`
 - Authorities are populated

If invalid

- Throws `AuthenticationException`
- Filter chain stops
- 401 / 403 returned

No magic. No retries. No silent failures.

STEP 7: AuthenticationManager returns authenticated user to Filter

What happens

- `AuthenticationManager` hands the authenticated object back to the filter.
- The filter now trusts the result.

Important

- Authentication happens **only once per request** (unless stateless JWT etc.)

STEP 8: SecurityContext is set in SecurityContextHolder

What happens

- The filter does:

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

 Copy code

Where this lives

- Stored in `ThreadLocal` by default.
- For stateful apps, persisted via `SecurityContextPersistenceFilter`.

Why this matters

- This is how `@AuthenticationPrincipal` works.
- This is how `hasRole()` works.
- No context → no security.

STEP 9: Request proceeds to DispatcherServlet & Controllers

What happens

- Only now does the request reach:
 - `DispatcherServlet`
 - Controllers
 - `@PreAuthorize`, `@Secured`, etc.

Authorization

- Happens **after authentication**
- Uses data from `SecurityContext`

Rid Mistakes People Make (Don't Be One of Them)

Big mistakes people make (don't be one of them)

- ✗ Thinking `UserDetailsService` authenticates users
- ✗ Thinking filters know about controllers
- ✗ Confusing authentication with authorization
- ✗ Believing `SecurityContextHolder` is global (it's thread-local)
- ✗ Assuming Spring Security is MVC-based (it's Servlet-based)

One Honest Limitation

Your diagram **does not show**:

- Stateless JWT flow
- OAuth2 login redirect flow
- Remember-Me services
- `ExceptionTranslationFilter` behavior

That's fine — **this diagram is for classic username/password auth**, and within that scope it's solid.

If you want, next we can:

- Rewrite this diagram in **plain English interview answer**
- Map **each filter to real code**
- Or simulate this flow with a **debug log walkthrough**

Your call.



Rewrite this diagram in plain English interview answer



Alright. Here's the **plain-English, interview-ready explanation** of that diagram.

No fluff. No buzzword soup. This is exactly how you should say it in a senior-level interview.

Spring Security Request Flow — Plain English Interview Answer

“What happens when a request comes to a Spring Boot application secured with Spring Security?”

1. The HTTP request first hits Spring Security, not the controller

When a request comes in, it **does not go to the controller directly**.

It first enters Spring Security's **filter chain**, which is a series of servlet filters that execute before Spring MVC.

2. Security filters decide whether authentication is required

Each filter has a specific responsibility.

If the request needs authentication (like a login request or a protected API), the relevant filter—such as

`UsernamePasswordAuthenticationFilter`—takes over.

3. The filter does not authenticate the user itself

The filter only:

- extracts credentials (username/password, token, etc.)
- creates an **unauthenticated Authentication object**
- delegates authentication to the **AuthenticationManager**

This separation is intentional—filters don't contain authentication logic.

4. AuthenticationManager delegates to an AuthenticationProvider

The `AuthenticationManager` doesn't validate credentials either.

It checks its registered `AuthenticationProvider`s and selects the one that supports the authentication type.

For example:

- username/password → `DaoAuthenticationProvider`
- OAuth → OAuth provider
- in-memory → `InMemory provider`

5. AuthenticationProvider performs the actual authentication

The selected `AuthenticationProvider` does the real work.

For `DaoAuthenticationProvider`:

- It calls `UserDetailsService` to load the user
- It uses `PasswordEncoder` to verify the password

- It builds a **fully authenticated Authentication object** if successful

If anything is wrong, it throws an exception and authentication fails immediately.

6. Authenticated user is returned back up the chain

Once authentication succeeds:

- The provider returns the authenticated object to the `AuthenticationManager`
- The manager returns it back to the filter

At this point, the user is considered authenticated.

7. Authentication is stored in the SecurityContext

The filter stores the authenticated user in the `SecurityContextHolder`.

This is critical because:

- Spring Security uses this context for authorization
- Controllers, annotations, and access rules rely on it
- By default, it's stored per thread

8. The request finally reaches the controller

Only after authentication is complete:

- The request proceeds to `DispatcherServlet`
- Controllers are executed
- Authorization checks like `@PreAuthorize` or `hasRole()` are applied

If the user isn't authorized, access is denied here.

One-Line Summary (Very Interview-Friendly)

"Spring Security intercepts the request using filters, delegates authentication to an `AuthenticationManager`, which uses an `AuthenticationProvider` to validate credentials, stores the authenticated user in the `SecurityContext`, and only then allows the request to reach the controller."

If You Want to Sound Senior, Add This

"Authentication is handled in the filter layer, authorization happens after authentication using the `SecurityContext`, and controllers never deal with authentication logic directly."

If you want next:

- A 30-second ultra-short version
- A code-mapped explanation
- Or JWT vs session flow comparison

Say it.



Is this conversation helpful so far?



+ Ask anything



ChatGPT can make mistakes. Check important info. See [Cookie Preferences](#).