

2-Spring-Security Configuration Explained



```

10 @Configuration
11 public class ProjectSecurityConfig {
12
13     @Bean
14     SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
15         //http.authorizeHttpRequests((requests) -> requests.anyRequest().permitAll());
16         //http.authorizeHttpRequests((requests) -> requests.anyRequest().denyAll());
17         http.authorizeHttpRequests((requests) -> requests
18             .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards").authenticated()
19             .requestMatchers("/notices", "/contact", "/error").permitAll());
20         http.formLogin(withDefaults());
21         http.httpBasic(withDefaults());
22         return http.build();
23     }
24 }
25
26

```

Kindly explain the above code and How internally Spring security is working



The provided code is a Spring Security configuration class. Let me break it down for you and explain how Spring Security works internally in this case.

Code Breakdown

1. `@Configuration` Annotation:

- This indicates that the class is a Spring configuration class. It contains bean definitions and is processed during application startup.

2. `@Bean` Annotation:

- This marks the `defaultSecurityFilterChain` method as a bean. The return value of this method will be managed by the Spring container.

3. `SecurityFilterChain`:

- This is a Spring Security interface that defines the security filter chain for the application.

4. `authorizeHttpRequests()`:

- This method configures which HTTP requests are allowed or require authentication.
- The commented-out lines demonstrate two possible configurations:
 - `permitAll()`: Allow all requests without authentication.

- `denyAll()`: Deny all requests.

5. Active Configuration:

```
java

http.authorizeHttpRequests((requests) ->
    requests
        .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards").authenticated()
        .requestMatchers("/notices", "/contact", "/error").permitAll()
    );
```

- `requestMatchers()`: Specifies URL patterns to configure.
- `.authenticated()`: Requires authentication for specified URL patterns.
- `.permitAll()`: Allows unrestricted access to the specified URL patterns.

6. `formLogin(withDefaults())`:

- Enables form-based authentication with default settings.

7. `httpBasic(withDefaults())`:

- Enables basic authentication with default settings.

8. `http.build()`:

- Builds the `SecurityFilterChain` object using the defined configurations.

How Spring Security Works Internally

1. Filter Chain:

- Spring Security uses a chain of filters to handle security concerns for HTTP requests. These filters intercept incoming requests and decide whether to allow or deny them.

2. AuthenticationManager:

- This is the core component responsible for processing authentication requests.
- It uses **AuthenticationProviders** to validate credentials.

3. Authorization:

- After authentication, Spring Security checks the user's roles and authorities to determine if they have access to a specific resource.
- This is defined by `authorizeHttpRequests()` in the code.

4. **SecurityContext:**

- Once authenticated, the user's authentication information is stored in the `SecurityContext`. This context is associated with the current thread and request lifecycle.

5. **Default Behavior:**

- By default, Spring Security applies a "deny-all" policy. You explicitly configure access rules for resources, as shown in the code.

6. **Form and Basic Authentication:**

- **Form Authentication:** Shows a login page to the user if they're not authenticated.
- **Basic Authentication:** Uses HTTP headers (`Authorization`) for authentication, suitable for APIs or non-browser clients.

Flow for a Request

1. **Incoming HTTP Request:**

- The request hits the Spring Security filter chain.

2. **Filter Processing:**

- Each filter in the chain evaluates the request.
- The `authorizeHttpRequests()` rules determine whether the request is authenticated or permitted.

3. **Authentication:**

- If authentication is required, the filter directs the request to the `AuthenticationManager`.

4. **Decision:**

- If authentication and authorization pass, the request proceeds to the controller.
- Otherwise, Spring Security returns an appropriate response (e.g., 403 Forbidden or a login page).

Let me know if you'd like further details on any part of this!



How AuthenticationManager and Provider are getting called



In Spring Security, the `AuthenticationManager` and `AuthenticationProvider` are central to the authentication process. Let's break down how they are invoked when a request is processed:

1. Authentication Flow Overview

1. User Sends Request:

- A request (e.g., login or accessing a protected resource) is sent to the server.
- This request passes through the Spring Security filter chain.

2. Filter Chain Interception:

- A specific filter in the chain (e.g., `UsernamePasswordAuthenticationFilter` for form login) intercepts the request and attempts to authenticate the user.

3. AuthenticationManager:

- The filter delegates the authentication process to the `AuthenticationManager`.

4. AuthenticationProvider:

- The `AuthenticationManager` delegates the actual authentication logic to one or more `AuthenticationProviders`.

5. Authentication Success/Failure:

- If an `AuthenticationProvider` successfully authenticates, an `Authentication` object is returned.
 - If no provider succeeds, an exception is thrown.
-

2. Key Components in Detail

A. `AuthenticationManager`

- **Role:** Orchestrates the authentication process by delegating to configured `AuthenticationProviders`.
- **How It's Called:**
 - It's usually invoked by a filter like `UsernamePasswordAuthenticationFilter` via `AuthenticationManager.authenticate(Authentication authentication)`.
- **Default Implementation:**
 - The most common implementation is `ProviderManager`, which supports multiple `AuthenticationProviders`.

B. `AuthenticationProvider`

- **Role:** Contains the actual authentication logic (e.g., validating username/password, tokens).
 - **How It's Called:**
 - The `AuthenticationManager` iterates over a list of `AuthenticationProviders` and calls `authenticate(Authentication authentication)` on each until one successfully authenticates or all fail.
-

3. Step-by-Step Execution

A. User Logs In (Form-Based Authentication Example)

1. User Submits Login Form:

- The form data (username and password) is sent in an HTTP POST request to the login endpoint.

2. Filter Interception:

- `UsernamePasswordAuthenticationFilter` intercepts the request.
- It extracts the username and password from the request and creates an `Authentication` object (e.g., `UsernamePasswordAuthenticationToken`).

3. AuthenticationManager Invoked:

- The filter calls `AuthenticationManager.authenticate(authentication)` with the token.

4. AuthenticationProvider Invoked:

- The `AuthenticationManager` (typically a `ProviderManager`) iterates over its list of `AuthenticationProviders`.
- For each provider:
 - It calls `provider.authenticate(authentication)`.

5. Authentication Validation:

- For example, a `DaoAuthenticationProvider`:
 - Retrieves user details (e.g., from a database) using a `UserDetailsService`.
 - Compares the provided password with the stored password (after encoding).

6. Authentication Success:

- If a provider authenticates successfully, it returns a populated `Authentication` object.
- The filter stores this object in the `SecurityContext`.

7. Authentication Failure:

- If no provider authenticates, an exception is thrown, and the user is returned an error response.

B. Accessing a Protected Resource

1. Request Interception:

- A filter (e.g., `BasicAuthenticationFilter`) extracts the authentication token (e.g., a username and password or a session token).

2. SecurityContext Check:

- If the `SecurityContext` already contains an `Authentication` object, the user is considered authenticated.
- Otherwise, the filter delegates to the `AuthenticationManager` for authentication.

4. Code Example

Here's an example of how this process can be configured in Spring Security:

Configuration:

```
java

@Configuration
public class SecurityConfig {

    @Bean
    public AuthenticationManager authManager(AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }

    @Bean
    public DaoAuthenticationProvider authenticationProvider(UserDetailsService userDetailsService, PasswordEncoder passwordEncoder) {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder);
        return provider;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((requests) ->
                requests.anyRequest().authenticated()
            )
            .formLogin(withDefaults());
        return http.build();
    }
}
```

Key Points:

1. **AuthenticationManager:**

- Automatically configured by `AuthenticationConfiguration`.
- Delegates authentication to `DaoAuthenticationProvider`.

2. **DaoAuthenticationProvider:**

- Uses a `UserDetailsService` to load user details.
- Validates the credentials using a `PasswordEncoder`.

3. **Filters:**

- The `UsernamePasswordAuthenticationFilter` will call the `AuthenticationManager` when a login request is submitted.
-

5. Internals of `AuthenticationManager` and `AuthenticationProvider`

AuthenticationManager:

- Implements `authenticate(Authentication authentication)`.
- Delegates the call to a list of `AuthenticationProviders`.

AuthenticationProvider:

- Implements `authenticate(Authentication authentication)` to perform specific checks.
 - Example providers:
 - `DaoAuthenticationProvider`: Validates against a database.
 - `JwtAuthenticationProvider`: Validates JWT tokens.
-

Sequence of Method Calls

1. **Filter Chain:**

- `UsernamePasswordAuthenticationFilter.doFilter()`

2. **AuthenticationManager:**

- `ProviderManager.authenticate()`

3. AuthenticationProvider:

- `DaoAuthenticationProvider.authenticate()`
- Retrieves user details from the `UserDetailsService`.
- Validates credentials.

4. Result:

- A valid `Authentication` object or an exception.

Let me know if you'd like a deeper dive into any specific part of this flow!



By default, Spring Security framework protects all the paths present inside the web application. This behaviour is due to the code present inside the method `defaultSecurityFilterChain(HttpSecurity http)` of class `SpringBootWebSecurityConfiguration`

```
@Bean
@Order(SecurityProperties.BASIC_AUTH_ORDER)
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests().anyRequest().authenticated();
    http.formLogin();
    http.httpBasic();
    return http.build();
}
```

Kindly explain the above



The image describes the **default security configuration** in Spring Security, particularly inside the Spring Boot framework. It shows how Spring Security protects all application paths by default using the `SecurityFilterChain`. Let's break this down in detail:

What Is Happening in the Code?

1. Method: `defaultSecurityFilterChain`

- This is a method that configures Spring Security's default behavior.
- It defines a **security filter chain** to protect the application's HTTP endpoints.

2. Annotations Used

1. `@Bean`:`

- Registers this method's return value (`SecurityFilterChain``) as a bean in the Spring application context.
- This allows Spring to manage and apply this security configuration automatically.

2. `@Order(SecurityProperties.BASIC_AUTH_ORDER)`:`

- Sets the order in which this filter chain is applied relative to other filter chains.
- By default, **Spring Security applies this as the primary filter chain for authentication and authorization.**

3. Code Configuration

```
java
```

```
http.authorizeHttpRequests().anyRequest().authenticated();
```

- `authorizeHttpRequests()`:`
 - Configures rules for authorizing HTTP requests.
- `anyRequest().authenticated()`:`
 - Ensures that **all HTTP requests require authentication** by default.
 - Users need to log in to access any endpoint.

```
java
```

```
http.formLogin();
```

- `formLogin()`:`

- Enables form-based authentication.
- When a user accesses a protected resource, they are redirected to a login form.
- Spring Security provides a default login page if no custom page is configured.

```
java

http.httpBasic();
```

- `httpBasic()`:
 - Enables HTTP Basic Authentication.
 - Credentials (username and password) are sent as a base64-encoded HTTP header (`Authorization`) for each request.
 - This is commonly used for APIs or simple client-server interactions.

```
java

return http.build();
```

- `http.build()`:
 - Finalizes and builds the security filter chain configuration.
 - This filter chain is then applied to incoming requests.

Default Behavior in Spring Security

1. All Paths Are Protected:

- By default, Spring Security applies a **deny-all** policy for all application paths.
- Only authenticated users can access resources.

2. Login Mechanisms:

- Both form-based login (`formLogin()`) and basic authentication (`httpBasic()`) are enabled.
- A user can either:
 - Log in via the default login form.

- Provide credentials in the ``Authorization`` header for HTTP Basic authentication.

3. Preconfigured Settings:

- Spring Security uses built-in defaults (like the default login page and error handling).
- These defaults can be overridden by customizing the security configuration.

How It Works Internally

1. Filter Chain Activation:

- When the application starts, Spring Security builds and registers the ``SecurityFilterChain`` defined in this method.

2. Intercepting Requests:

- Every incoming HTTP request is passed through the filter chain.
- The chain checks whether the request matches a rule (in this case, ``authenticated()`` for all requests).

3. Enforcing Authentication:

- If the user is not authenticated:
 - The ``formLogin()`` redirects the user to the default login page.
 - The ``httpBasic()`` challenges the user for credentials via a ``401 Unauthorized`` response.

4. Authentication and Authorization:

- Once the user logs in successfully, Spring Security stores their authentication details in the ``SecurityContext``.
- Future requests use this context to determine access permissions.

Summary

This default configuration is provided by Spring Security's ``SpringBootWebSecurityConfiguration`` class to secure all paths by requiring authentication. It enables both form-based and basic authentication mechanisms out of the box. You can customize this behavior by overriding or extending the configuration in your application.

Would you like to know how to customize this configuration or more details about any part?



CUSTOM SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

**eazy
bytes**

We can secure the web application APIs, Paths as per our custom requirements using Spring Security framework like shown below,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests()
            .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards").authenticated()
            .requestMatchers("/notices", "/contact").permitAll()
            .and().formLogin()
            .and().httpBasic();

        return http.build();
    }
}
```

DENY ALL SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

**eazy
bytes**

NOT RECOMMENDED
FOR PRODUCTION

We can deny all the requests coming towards our web application APIs, Paths using Spring Security framework like shown below,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests()
            .anyRequest().denyAll()
            .and().formLogin()
            .and().httpBasic();

        return http.build();
    }
}
```

PERMIT ALL SECURITY CONFIGURATIONS

INSIDE SPRING SECURITY FRAMEWORK

**eazy
bytes**

NOT RECOMMENDED
FOR PRODUCTION

We can permit all the requests coming towards our web application APIs, Paths using Spring Security framework like shown below,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests()
            .anyRequest().permitAll()
            .and().formLogin()
            .and().httpBasic();

        return http.build();
    }
}
```

CONFIGURE USERS

USING `InMemoryUserDetailsManager`

eazy
byteNOT RECOMMENDED
FOR PRODUCTION

Instead of defining a single user inside `application.properties`, as a next step we can define multiple users along with their authorities with the help of `InMemoryUserDetailsManager` & `UserDetails`

Approach 1 where we use
`withDefaultPasswordEncoder()` method
while creating the user details

```
@Bean
public InMemoryUserDetailsManager userDetailsService(){
    UserDetails admin = User.withDefaultPasswordEncoder()
        .username("admin")
        .password("12345")
        .authorities("admin")
        .build();
    UserDetails user = User.withDefaultPasswordEncoder()
        .username("user")
        .password("12345")
        .authorities("read")
        .build();
    return new InMemoryUserDetailsManager(admin, user);
}
```




Kindly explain the above images in detail



The provided images showcase various **Spring Security configurations** and illustrate ways to control access to your application, including **custom security configurations**, **permit-all or deny-all policies**, and **user management with `InMemoryUserDetailsManager`**. Here's a detailed explanation of each image:

1. Custom Security Configurations

This image illustrates a custom configuration to secure specific API paths based on requirements.

Code Explanation:

```
java
```

```
http.authorizeHttpRequests()
    .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards").authenticated()
    .requestMatchers("/notices", "/contact").permitAll()
    .and().formLogin()
    .and().httpBasic();
```

- ``requestMatchers()``:
 - Specifies URL patterns and access rules:
 - ``"/myAccount", "/myBalance", "/myLoans", "/myCards"``: These endpoints require authentication.
 - ``"/notices", "/contact"``: These endpoints are accessible to everyone (no authentication required).
- ``authenticated()``:
 - Ensures that only authenticated users can access these endpoints.
- ``permitAll()``:
 - Allows unrestricted access to specific endpoints.
- ``formLogin()`` and ``httpBasic()``:
 - Enables both form-based and HTTP Basic authentication for the application.

Use Case:

- You can define fine-grained access control for different endpoints based on application requirements.

2. Deny-All Security Configurations

This configuration blocks access to all application paths.

Code Explanation:

```
java

http.authorizeHttpRequests()
    .anyRequest().denyAll()
    .and().formLogin()
    .and().httpBasic();
```

- `anyRequest().denyAll()`:
 - Denies access to every incoming HTTP request, regardless of the user's authentication status.
- `.formLogin()` and `.httpBasic()`:
 - Login options are enabled, but since all requests are denied, even authenticated users cannot access resources.

Use Case:

- This configuration can be used for maintenance or debugging purposes, but it is **not recommended for production** since it blocks all application functionality.

3. Permit-All Security Configurations

This configuration allows unrestricted access to all endpoints.

Code Explanation:

```
java

http.authorizeHttpRequests()
    .anyRequest().permitAll()
```

```
.and().formLogin()
.and().httpBasic();
```

- ``anyRequest().permitAll()``:
 - Grants access to all requests without requiring authentication.
- ``formLogin()`` and ``httpBasic()``:
 - Even though login mechanisms are enabled, they are not needed because all endpoints are accessible.

Use Case:

- This configuration is useful during development or for public-facing applications without any restricted endpoints, but it is **not recommended for production** due to security risks.

4. Configure Users with `InMemoryUserDetailsManager` (Approach 1)`

This example shows how to configure multiple users with roles using an `InMemoryUserDetailsManager``.

Code Explanation:

```
java

@Bean
public InMemoryUserDetailsManager userDetailsService() {
    UserDetails admin = User.withDefaultPasswordEncoder()
        .username("admin")
        .password("12345")
        .authorities("admin")
        .build();

    UserDetails user = User.withDefaultPasswordEncoder()
        .username("user")
        .password("12345")
        .authorities("read")
        .build();
}
```

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

- ``UserDetails``:
 - Represents a user with a username, password, and granted authorities (roles).
- ``withDefaultPasswordEncoder()``:
 - Simplifies password encoding for development but is **not recommended for production**.
- ``InMemoryUserDetailsManager``:
 - Manages users in memory (no database integration).

Use Case:

- Quick setup for testing or small applications with minimal security needs.

5. Configure Users with ``InMemoryUserDetailsManager`` (Approach 2)

This approach improves upon Approach 1 by creating a separate bean for password encoding.

Code Explanation:

```
java

@Bean
public InMemoryUserDetailsManager userDetailsService() {
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

    UserDetails admin = User.withUsername("admin")
        .password("12345")
        .authorities("admin")
        .build();

    UserDetails user = User.withUsername("user")
        .password("12345")
        .authorities("read")
        .build();

    manager.createUser(admin);
}
```

```
manager.createUser(user);
    return manager;
}

@Bean
public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
}
```

- `InMemoryUserDetailsManager``:
 - Similar to Approach 1, but users are explicitly created using `createUser()`.
- `PasswordEncoder``:
 - The `NoOpPasswordEncoder`` is used to avoid password encoding for simplicity.
 - **Important:** Do not use `NoOpPasswordEncoder`` in production.

Use Case:

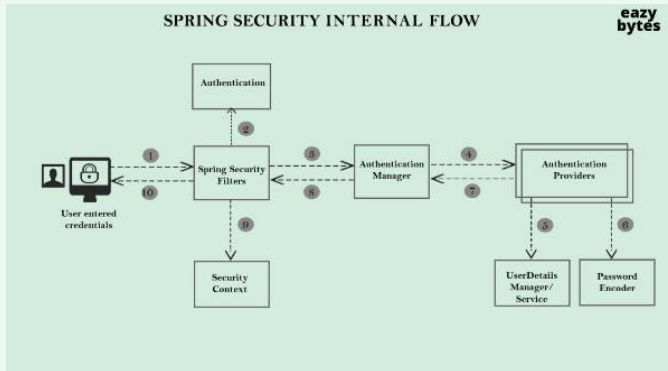
- Provides better separation of concerns and is suitable for development environments.

Conclusion

- **Custom Security Configurations:**
 - Use these to control access to different parts of your application based on roles or permissions.
- **Deny-All/Permit-All:**
 - Useful for special scenarios but **not suitable for production**.
- **InMemoryUserDetailsManager:**
 - A quick way to manage users for testing or prototyping. For production, consider integrating a persistent user store (e.g., database).

Let me know if you want to explore any specific part in more detail!





Kindly explain the given image in details



The provided image illustrates the **internal flow of Spring Security** during an authentication process. It details the steps involved in verifying user credentials and managing security in a web application. Let's break down each step:

Flow Breakdown

1. User Enters Credentials

- The user sends a request with credentials (username and password) to access a protected resource. This request is intercepted by **Spring Security filters**.

2. Spring Security Filters

- The incoming request is processed through the **Spring Security filter chain**. Key filters include:
 - ``UsernamePasswordAuthenticationFilter``: Handles form-based authentication.
 - ``BasicAuthenticationFilter``: Handles HTTP Basic authentication.
- These filters extract the user's credentials (e.g., username and password) and create an ``Authentication`` object, which is passed to the ``AuthenticationManager``.

3. Authentication Manager

- The `AuthenticationManager` orchestrates the authentication process by delegating the request to one or more `AuthenticationProviders`.
 - Common implementation: `ProviderManager`, which supports multiple `AuthenticationProviders`.
-

4. Authentication Providers

- `AuthenticationProvider` validates the credentials.
 - For example, a `DaoAuthenticationProvider`:
 - Uses a `UserDetailsService` to fetch user information.
 - Uses a `PasswordEncoder` to verify the password.
 - The `AuthenticationProvider` is responsible for actual authentication logic. It either:
 - Returns a fully authenticated `Authentication` object, or
 - Throws an exception if authentication fails.
-

5. UserDetails Manager/Service

- The `UserDetailsService` fetches user information (e.g., username, password, roles) from a persistent store such as:
 - In-memory storage (e.g., `InMemoryUserDetailsManager`).
 - Database (custom implementation).
 - It returns a `UserDetails` object, which contains user information required for authentication.
-

6. Password Encoder

- The `PasswordEncoder` validates the provided password by encoding it and comparing it to the stored password.
 - Common implementations:
 - `BCryptPasswordEncoder` (recommended for production).
 - `NoOpPasswordEncoder` (only for development).
-

7. Returning to Authentication Manager

- The `AuthenticationProvider` returns the authenticated `Authentication` object to the `AuthenticationManager` if the credentials are valid.
-

8. Returning to Security Filters

- The `AuthenticationManager` returns the authenticated `Authentication` object to the filter that initiated the request.
 - The filter stores the `Authentication` object in the `SecurityContext`.
-

9. Security Context

- The `SecurityContext` holds the authenticated user's details (username, roles, etc.) for the duration of the session or request lifecycle.
 - Subsequent requests use the `SecurityContext` to check user permissions.
-

10. Response to the User

- If authentication is successful:
 - The user is granted access to the requested resource.
 - If authentication fails:
 - An appropriate error response (e.g., `401 Unauthorized`) is returned.
-

Flow Summary

1. The user sends credentials via a request.
 2. Spring Security filters intercept the request.
 3. Credentials are sent to the `AuthenticationManager`.
 4. The `AuthenticationManager` delegates to `AuthenticationProviders`.
 5. The `AuthenticationProvider` validates credentials using the `UserDetailsService` and `PasswordEncoder`.
 6. A valid `Authentication` object is returned and stored in the `SecurityContext`.
 7. The user gains access to the protected resource.
-

Key Components Explained

1. **Spring Security Filters:**

- The entry point for all security-related processing.
- Filters are applied in a chain, where each filter performs a specific function (e.g., authentication, authorization).

2. **AuthenticationManager:**

- Central coordinator for authentication.
- Delegates to multiple `AuthenticationProviders`.

3. **AuthenticationProviders:**

- Perform the actual authentication.
- Can be custom implementations or predefined providers like `DaoAuthenticationProvider`.

4. **UserDetailsService:**

- Fetches user details from a database or in-memory store.

5. **PasswordEncoder:**

- Ensures secure password storage and validation.

6. **SecurityContext:**

- Holds the authenticated user's details for use in the application.

Practical Use Case

When a user logs in, their credentials (username and password) are checked against stored values (e.g., in a database). If valid, Spring Security grants access to resources based on their roles or permissions.

This modular approach allows for flexibility, customizations, and integration with external systems (e.g., LDAP, OAuth2).

Let me know if you'd like further clarification on any step!