

3-Spring Security User Management



USER MANAGEMENT IMPORTANT CLASSES & INTERFACES

**eazy
bytes**

Core interface which loads user-specific data.

UserDetailsService
(Interface)

✓ loadUserByUsername(String username)

An extension of the UserDetailsService which provides the ability to create new users and update existing ones.

UserDetailsManager
(Interface)

✓ createUser(UserDetails user)
✓ updateUser(UserDetails user)
✓ deleteUser(String username)
✓ changePassword(String oldPwd, String newPwd)
✓ userExists(String username)

Sample implementation classes provided by the Spring Security team

InMemoryUser
DetailsManager

JdbcUser
DetailsManager

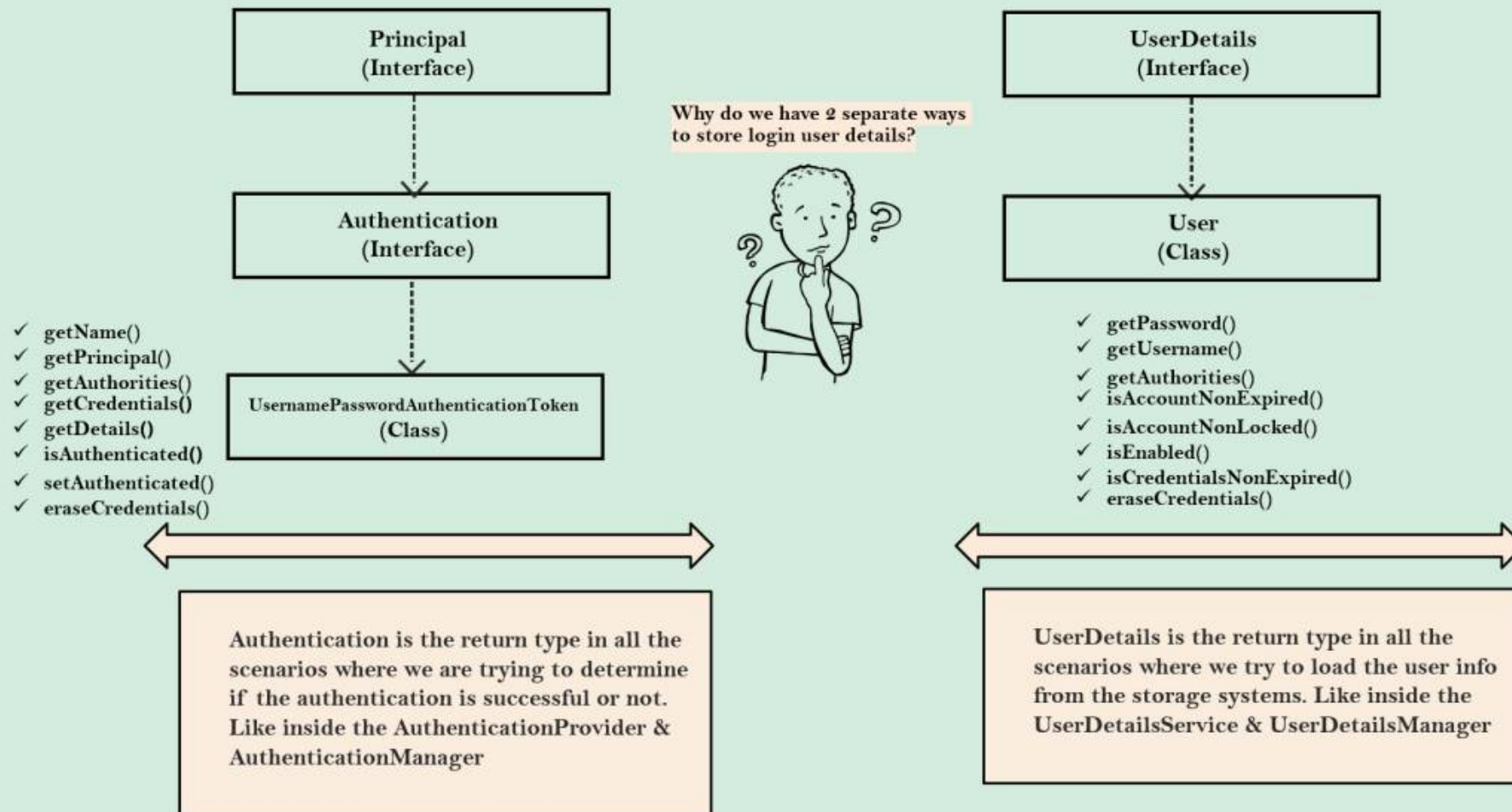
LdapUser
DetailsManager

UserDetails

All the above interfaces & classes uses an interface UserDetails & its implementation which provides core user information.

USERDETAILS & AUTHENTICATION

RELATION BETWEEN THEM



Kindly explain the above image in details



The diagrams you uploaded detail the user management and authentication mechanisms in **Spring Security**, a framework for securing Java applications. Let me explain each diagram in detail.

Diagram 1: User Management - Important Classes & Interfaces

Key Components:

1. `UserService` (Interface):

- The core interface in Spring Security that loads user-specific data.
- It has a single method:

```
java

UserDetails loadUserByUsername(String username);
```

- This is primarily used during authentication to retrieve user details from a data source.

2. `UserDetailsManager` (Interface):

- Extends `UserService` and adds methods for user management:
 - `createUser(UserDetails user)` – Adds a new user.
 - `updateUser(UserDetails user)` – Updates an existing user.
 - `deleteUser(String username)` – Deletes a user.
 - `changePassword(String oldPwd, String newPwd)` – Changes a user's password.
 - `userExists(String username)` – Checks if a user exists.

3. Implementations of `UserDetailsManager`:

- `InMemoryUserDetailsManager`: Stores user details in memory. Useful for testing or simple use cases.
- `JdbcUserDetailsManager`: Stores and retrieves user details from a database using JDBC.
- `LdapUserDetailsManager`: Manages users in an LDAP (Lightweight Directory Access Protocol) system.

4. `UserDetails` Interface:

- Represents core user information. All the above interfaces and implementations rely on this interface to represent a user.
- Common methods include:
 - `getUsername()`
 - `getPassword()`

- `getAuthorities()`
- Account and credential status checks (e.g., `isAccountNonLocked()`, `isEnabled()`).

Diagram 2: UserDetails & Authentication - Relation Between Them

Key Components:

1. Authentication Flow:

- **`Principal` (Interface):**
 - Represents the currently authenticated user. It is part of the Java security package.
- **`Authentication` (Interface):**
 - Represents the authentication request or the authenticated user's information.
 - Common methods:
 - `getName()` – Retrieves the username.
 - `getPrincipal()` – Retrieves the user details (often a `UserDetails` object).
 - `getAuthorities()` – Retrieves roles or permissions of the user.
 - `isAuthenticated()` – Checks if the user is authenticated.
- **`UsernamePasswordAuthenticationToken` (Class):**
 - A concrete implementation of `Authentication` used for username-password-based authentication.

2. `UserDetails` & `User` Class:

- **`UserDetails` Interface:**
 - Represents user information loaded by the `UserService`.
 - Contains methods such as:
 - `getUsername()`
 - `getPassword()`
 - `isAccountNonExpired()`
 - `isCredentialsNonExpired()`
- **`User` Class:**
 - A concrete implementation of `UserDetails` provided by Spring Security.
 - Used as a standard implementation for user information.

Why Two Separate Mechanisms (`Authentication` vs. `UserDetails`):

- ``Authentication``:
 - Used to determine if a user is authenticated.
 - Represents the result of the authentication process.
- ``UserDetails``:
 - Represents the user's information retrieved from the storage layer.
 - This is used to load user data before authentication.

Key Takeaways:

- **Authentication Process:**
 1. User credentials are submitted.
 2. ``UserDetailsService`` loads user information from storage.
 3. ``AuthenticationManager`` verifies the credentials and returns an ``Authentication`` object if valid.
- **User Management:**
 - ``UserDetailsManager`` extends ``UserDetailsService`` to allow creating, updating, and deleting users.

Let me know if you'd like further clarification or examples!



Kindly explain in details How does user management works in Spring security



User management in **Spring Security** involves handling user data, such as creating, updating, deleting, authenticating, and retrieving user-specific details. Here's a detailed explanation of how user management works:

1. Core Components of User Management

A. ``UserDetails``

- Represents user-specific information required for authentication and authorization.
- Methods:

- `getUsername()` – Returns the username.
- `getPassword()` – Returns the password.
- `getAuthorities()` – Returns roles/permissions granted to the user.
- `isAccountNonLocked()`, `isEnabled()` – Check account status.
- Spring provides a default implementation called `User` class.

B. `UserDetailsService`

- The core interface used to retrieve user information from a data source (database, in-memory, etc.).
- Method:

```
java

UserDetails loadUserByUsername(String username);
```

- During authentication, Spring Security calls this method to fetch user details.

C. `UserDetailsManager`

- Extends `UserDetailsService` and adds user management capabilities.
- Methods:
 - `createUser(UserDetails user)` – Add a new user.
 - `updateUser(UserDetails user)` – Update existing user details.
 - `deleteUser(String username)` – Delete a user by username.
 - `changePassword(String oldPwd, String newPwd)` – Change the user's password.
 - `userExists(String username)` – Check if a user exists.

D. Implementations of `UserDetailsManager`

1. `InMemoryUserDetailsManager`:

- Manages users stored in memory.
- Suitable for testing or simple applications.

2. `JdbcUserDetailsManager`:

- Manages users stored in a relational database using JDBC.
- Uses default schemas for user and authority tables but can be customized.

3. `LdapUserDetailsManager``:

- Manages users in an LDAP system.
-

2. User Authentication Flow

Spring Security uses `UserDetailsService`` or `UserDetailsManager`` to handle user data during authentication. Here's how the process works:

A. Submitting Credentials:

- The user submits their username and password via a login form.
- A filter (e.g., `UsernamePasswordAuthenticationFilter``) intercepts the login request.

B. Loading User Data:

- Spring Security calls the `loadUserByUsername(String username)`` method of `UserDetailsService`` to retrieve the user's details (username, password, roles, etc.) from the data source.

C. Authenticating User:

- The retrieved user details are validated against the submitted credentials.
 - If valid, Spring Security creates an `Authentication`` object, marking the user as authenticated.
 - The `Authentication`` object is stored in the `SecurityContext``.
-

3. Customizing User Management

You can customize user management in Spring Security by implementing or configuring specific components:

A. Custom `UserDetailsService``

- Create a class that implements `UserDetailsService`` to fetch user data from a custom data source:

java

```
@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        // Fetch user from database or custom source
        return new User("admin", "{noop}password", List.of(new SimpleGrantedAuthority("ROLE_ADMIN")));
    }
}
```

B. Custom `UserDetails` Implementation

- Create a custom class that implements `UserDetails` for additional user fields:

java

```
public class CustomUserDetails implements UserDetails {
    private String username;
    private String password;
    private List<GrantedAuthority> authorities;

    // Additional fields like email, phone, etc.

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }

    // Implement other required methods...
}
```


C. Using `UserDetailsManager`

- Configure an `InMemoryUserDetailsManager` or `JdbcUserDetailsManager` to manage users:

```
java

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {
        return new JdbcUserDetailsManager(dataSource);
    }
}
```

4. User Management with Databases

When using `JdbcUserDetailsManager`, Spring Security expects specific database tables for users and authorities:

Default Table Structure:

1. Users Table:

- Columns: `username`, `password`, `enabled`.

2. Authorities Table:

- Columns: `username`, `authority`.

Example Queries:

- Query to fetch user details:

```
sql

SELECT username, password, enabled FROM users WHERE username = ?;
```

- Query to fetch authorities:

```
sql
```

```
SELECT username, authority FROM authorities WHERE username = ?;
```

Customizing Queries:

- You can override the default queries by configuring the `JdbcUserDetailsManager`:

```
java
```

```
JdbcUserDetailsManager manager = new JdbcUserDetailsManager();
manager.setUsersByUsernameQuery("SELECT ... FROM my_users WHERE ...");
manager.setAuthoritiesByUsernameQuery("SELECT ... FROM my_authorities WHERE ...");
```

5. Password Management

Spring Security supports password encoding for enhanced security:

- Use the `PasswordEncoder` interface to encode and validate passwords.
- Common implementations:
 - `BCryptPasswordEncoder` (recommended).
 - `NoOpPasswordEncoder` (for testing, no encryption).

Example:

```
java
```

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

6. Authorization with User Management

- The roles/authorities returned by `UserDetails.getAuthorities()` determine the user's access.
- Configure access rules using Spring Security DSL:

```
java

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/user/**").hasRole("USER")
        .anyRequest().authenticated();
}
```

7. Key Features of User Management

- Centralized user storage: Allows managing users in a single location (database, LDAP, etc.).
- Easy customization: Allows overriding default implementations to fit your application needs.
- Password security: Built-in support for password encoding and validation.
- Scalable: Can handle large-scale applications using databases or LDAP.

Let me know if you'd like code examples for any specific part of this process!



What is the relation between UserDetails & Authentication Kindly explain in details also with reference to above image



The relation between `UserDetails` and `Authentication` in Spring Security is critical to understanding how user information is managed during authentication and authorization. The above image illustrates this relationship visually. Here's a detailed explanation:

Understanding `UserDetails`

1. Definition:

- `UserDetails` is an interface in Spring Security that represents core user information.
- It is used by the `UserService` to load user information from a persistent storage (like a database or LDAP).

2. Key Responsibilities:

- Provides user-specific data required for authentication and authorization.
- Includes methods such as:
 - `getUsername()` – Returns the username.
 - `getPassword()` – Returns the encrypted password.
 - `getAuthorities()` – Returns roles/permissions granted to the user.
 - Account and credential status checks (e.g., `isAccountNonLocked()`, `isEnabled()`).

3. Role in Authentication:

- The `UserDetails` **object** is returned by the `UserService.loadUserByUsername(String username)` method during authentication.
- It contains all the necessary information about the user, which is then validated against the credentials provided by the user.

Understanding `Authentication`

1. Definition:

- `Authentication` is an interface in Spring Security that represents the authentication request or the authenticated user's information.
- It is a central part of Spring Security's `SecurityContext`.

2. Key Responsibilities:

- Contains details about the authentication process, including:
 - Principal (user's identity, often a `UserDetails` object).
 - Credentials (user's password, usually cleared post-authentication for security).
 - Authorities (roles/permissions granted to the user).
- Methods include:
 - `getPrincipal()` – Returns the user's identity.
 - `getAuthorities()` – Retrieves the roles/permissions of the user.
 - `isAuthenticated()` – Checks if the user is authenticated.

3. Role in Authentication:

- The `Authentication` **object** is created during the authentication process to represent the result of the authentication attempt.
- It is stored in the `SecurityContext` and is used for subsequent authorization decisions.

Relation Between `UserDetails` and `Authentication`

1. `UserDetails` is Part of `Authentication`:

- When a user tries to log in, their credentials are submitted.
- Spring Security's authentication mechanism retrieves the user's information from a data source using `UserService`, returning a `UserDetails` object.
- This `UserDetails` object is then used to populate the `Authentication` object.
 - Specifically, the `UserDetails` object is stored as the `principal` in the `Authentication` object.

Example:

```
java

Authentication auth = SecurityContextHolder.getContext().getAuthentication();
UserDetails userDetails = (UserDetails) auth.getPrincipal();
```

2. `UserDetails` is for User Information; `Authentication` is for Authentication Result:

- `UserDetails` represents the **data** (username, password, roles) required for authentication.
- `Authentication` represents the **state** of the authentication process (whether the user is authenticated or not).

3. Interaction During the Authentication Process:

- The `AuthenticationManager` or `AuthenticationProvider` validates the user's credentials.
- If valid, an `Authentication` object is created, and the `UserDetails` object is embedded in it as the `principal`.
- The `Authentication` object is then stored in the `SecurityContext` for subsequent access.

Explanation with Reference to the Image

1. Left Side (Principal and Authentication):

- **`Principal` Interface:**
 - Represents the identity of the user.
 - In Spring Security, this is typically the `UserDetails` object.
- **`Authentication` Interface:**

- Extends `Principal`.
- Includes additional methods to handle credentials, authorities, and authentication status.
- `UsernamePasswordAuthenticationToken` is an implementation of `Authentication`, used during username-password-based authentication.

2. Right Side (UserDetails and User):

- **`UserDetails` Interface:**
 - Provides the structure for storing user information required for authentication.
 - Implemented by the `User` class, which is a default Spring Security implementation.
- **`User` Class:**
 - Implements `UserDetails` and provides basic fields like `username`, `password`, and `authorities`.

3. Why Two Separate Ways to Store Login User Details?

- `UserDetails` is used by `UserDetailsService` to **fetch user data from a storage system**.
- `Authentication` is used to **store the result of authentication** and contains additional information like whether the user is authenticated or not.
- This separation ensures clear responsibilities:
 - `UserDetails` for data storage.
 - `Authentication` for the current state of authentication.

4. Arrows Between Components in the Image:

- The `UserDetails` **interface** connects to the `Authentication` **object** as part of its `principal`.
- The `Authentication` **object** references the `UserDetails` for its core user data during authorization decisions.

Authentication Process Summary

1. User submits credentials (username and password).
2. Spring Security retrieves `UserDetails` via `UserDetailsService`.
3. The `AuthenticationProvider` or `AuthenticationManager` validates the credentials using the `UserDetails` object.
4. On successful authentication:
 - An `Authentication` object is created with `UserDetails` as the `principal`.
 - This `Authentication` object is stored in the `SecurityContext`.

Let me know if you'd like further clarification or a specific example!



AUTHENTICATION

USING `JdbcUserDetailsManager`

eazy bytes

Instead of creating users inside the memory of web server, we can store them inside a DB and with the help of `JdbcUserDetailsManager`, we can perform authentication.

```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
    return new JdbcUserDetailsManager(dataSource);
}

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

Please note to create table as per the `JdbcUserDetailsManager` class & insert user records inside them. `NoOpPasswordEncoder` is not recommended for prod apps.

USERDETAILSSERVICE IMPLEMENTATION FOR CUSTOM USER FETCHING LOGIC

**eazy
bytes**

When we want to load the user details based on our own tables, columns, custom logic, then we need to create a bean that implements `UserDetailsService` and overrides the method `loadUserByUsername()`

```
@Service
public class EazyBankUserDetails implements UserDetailsService {

    1 usage
    @Autowired
    private CustomerRepository customerRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        List<Customer> customer = customerRepository.findByEmail(username);
        if (customer.size() == 0) {
            throw new UsernameNotFoundException("User details not found for the user : " + username);
        }
        return new SecurityCustomer(customer.get(0));
    }
}
```

Kindly explain the above code and when it will be called



Explanation of the Uploaded Code and Its Execution Context:

The images showcase two approaches to user authentication in **Spring Security**:

1. Using `JdbcUserDetailsManager` for database-backed user management.
2. Implementing a custom `UserService` for custom user retrieval logic.

Image 1: Using `JdbcUserDetailsManager`

Code Explanation:

1. Definition of `JdbcUserDetailsManager`:

- The `JdbcUserDetailsManager` is an out-of-the-box implementation of `UserService` and `UserDetailsManager`.
- It manages user details in a relational database using JDBC.

2. Key Annotations and Beans:

- `@Bean public UserService userDetailsService(DataSource dataSource)`:
 - Creates a bean of `JdbcUserDetailsManager` using the provided `DataSource`.
 - The `DataSource` points to the database where user details (username, password, roles) are stored.
- `@Bean public PasswordEncoder passwordEncoder()`:
 - Defines a password encoder.
 - `NoOpPasswordEncoder` is used here, which stores passwords in plain text (not recommended for production).

3. Steps to Use `JdbcUserDetailsManager`:

- Ensure the database contains the required tables:
 - **Users table**: For storing user credentials.
 - **Authorities table**: For storing user roles or authorities.
- Insert user details into these tables.
- When a user logs in, `JdbcUserDetailsManager` queries the database to retrieve the user's credentials and authorities.

4. When It Is Called:

- During the authentication process:
 - The `JdbcUserDetailsManager`'s `loadUserByUsername(String username)` method is called automatically by Spring Security.
 - It fetches the user details from the database and validates the credentials.

Image 2: Custom `UserService` Implementation

Code Explanation:

1. Custom Implementation of `UserDetailsService`:

- This code defines a custom class (`EazyBankUserDetails`) that implements `UserDetailsService`.
- This allows you to customize how user information is fetched, for example, from a custom database schema, an external API, or other storage mechanisms.

2. Key Code Sections:

- **`@Service` Annotation:**
 - Marks this class as a Spring service bean, allowing it to be injected where required.
- **`@Autowired private CustomerRepository customerRepository`:**
 - Injects a repository class (`CustomerRepository`) to fetch user details from a database.
- **`loadUserByUsername(String username)` Method:**
 - Overrides the method from the `UserDetailsService` interface.
 - Fetches the user from the database using the repository.
 - Throws a `UsernameNotFoundException` if no user is found.
 - Wraps the fetched user into a custom `UserDetails` implementation (`SecurityCustomer`) and returns it.

3. When It Is Called:

- Spring Security invokes this method during authentication to retrieve user information.
- For example:
 - A user attempts to log in with a username and password.
 - Spring Security calls `loadUserByUsername(username)`.
 - The `CustomerRepository` retrieves the user data from the database.
 - The returned `SecurityCustomer` object contains user credentials and authorities for authentication.

4. Advantages of Custom `UserDetailsService`:

- Flexibility to use custom database schemas, table structures, or external systems.
- Add additional business logic during user retrieval.

Comparison of the Two Approaches:

Feature	<code>JdbcUserDetailsManager</code>	Custom <code>UserDetailsService</code>
Use Case	Standard database schema for users and authorities.	Custom logic or non-standard database schemas.
Ease of Use	Easy to configure with minimal coding.	Requires implementing custom logic.

Feature	<code>JdbcUserDetailsManager</code>	Custom <code>UserService</code>
Customization	Limited (requires overriding default queries).	Highly customizable for complex requirements.
When to Use	For simple use cases or quick prototyping.	For custom business logic or complex schemas.

Practical Example:

- If you're using Spring Security to authenticate users for an enterprise banking application:
 - `JdbcUserDetailsManager`: Useful if the database follows Spring Security's default schema (users, authorities tables).
 - **Custom `UserService`**: Necessary if the database has a custom schema or you need additional fields like email, phone number, etc., in the user object.

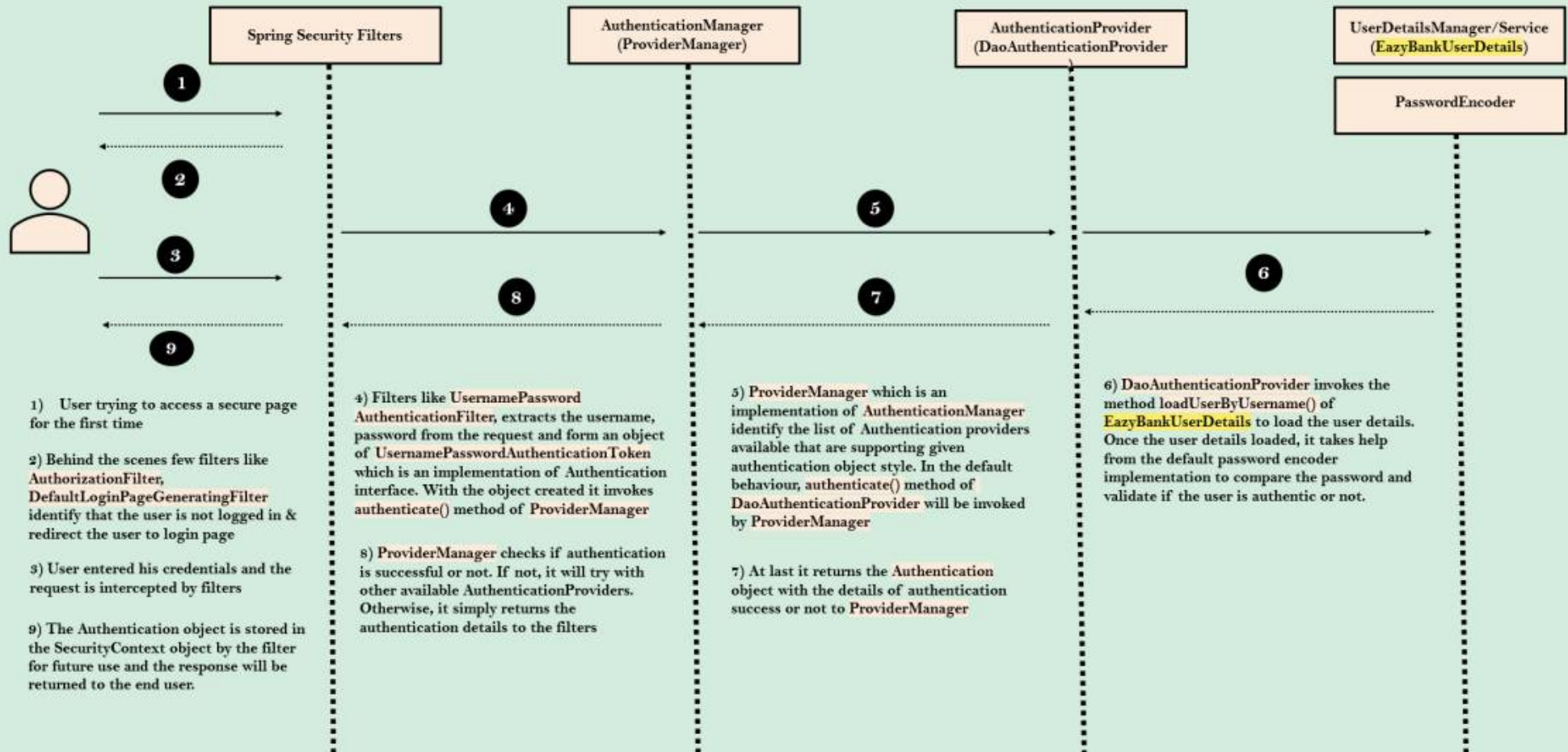
Let me know if you'd like further clarification or additional examples!



SEQUENCE FLOW

WITH OUR OWN USERDETAILSSERVICE IMPLEMENTATION

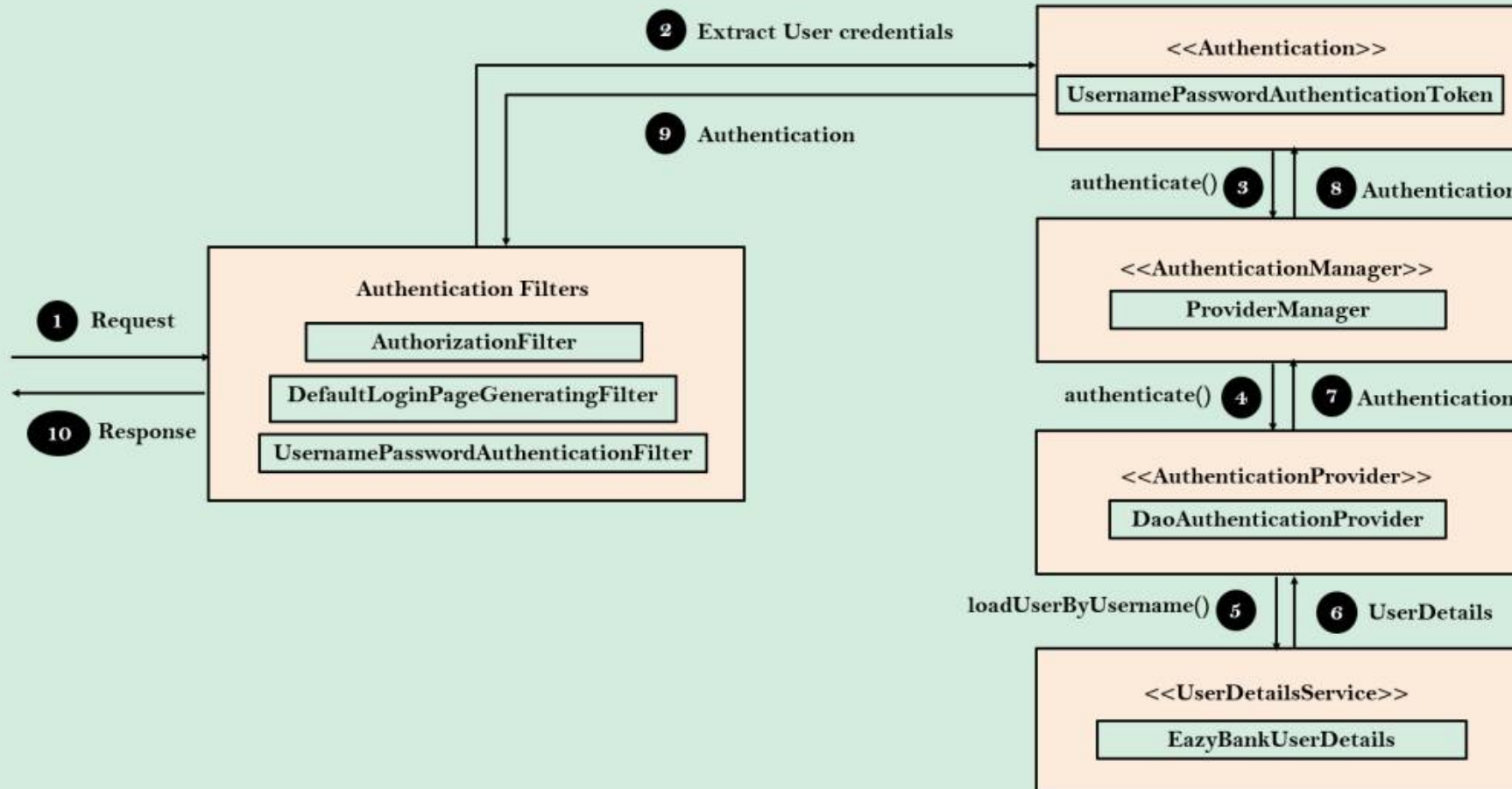
easy
bytes



SEQUENCE FLOW

WITH OUR OWN USERDETAILSSERVICE IMPLEMENTATION

easy bytes



Kindly explain the above images along with detailed explanation



The images represent the **sequence flow of authentication** in **Spring Security** using a custom ``UserDetailsService`` implementation. Here's a detailed explanation of the steps involved:

Image 1: Sequence Flow with Custom ``UserDetailsService`` Implementation

This image explains the **step-by-step process** of how authentication works when a custom ``UserDetailsService`` is implemented.

Step-by-Step Sequence:

1. User Request (Step 1):

- A user tries to access a secure resource for the first time.
- Spring Security identifies that the user is not authenticated.

2. Filters Intercept the Request (Step 2):

- Behind the scenes, Spring Security filters (e.g., ``AuthorizationFilter``, ``DefaultLoginPageGeneratingFilter``) intercept the request.
- The user is redirected to the login page to submit their credentials.

3. User Submits Credentials (Step 3):

- The user submits their username and password through the login form.
- The ``UsernamePasswordAuthenticationFilter`` captures the credentials and creates an instance of ``UsernamePasswordAuthenticationToken`` (an implementation of the ``Authentication`` interface).

4. Authentication Begins (Step 4):

- The ``UsernamePasswordAuthenticationToken`` is passed to the ``ProviderManager`` (the default ``AuthenticationManager`` implementation).
- The ``ProviderManager`` invokes its ``authenticate()`` method.

5. ProviderManager Delegates to DaoAuthenticationProvider (Step 5):

- The ``ProviderManager`` checks its list of ``AuthenticationProviders`` to find one that supports the token type.
- By default, the ``DaoAuthenticationProvider`` is invoked.

6. UserDetailsService is Called (Step 6):

- The ``DaoAuthenticationProvider`` calls the ``loadUserByUsername(String username)`` method of the custom ``UserDetailsService`` implementation.
- In this example, the ``EazyBankUserDetails`` service retrieves the user information (e.g., username, password, roles) from the data source (e.g., database).
- The retrieved user details are wrapped in a ``UserDetails`` object.

7. Password Validation (Step 7):

- The `DaoAuthenticationProvider` uses the configured `PasswordEncoder` to compare the submitted password with the stored password.
- If the password matches, the user is authenticated.

8. Authentication Success (Step 8):

- On successful authentication, an `Authentication` object is created.
- The `Authentication` object contains details about the authenticated user (e.g., roles, username, etc.).

9. Storing Authentication (Step 9):

- The `Authentication` object is stored in the `SecurityContext` by the filter.
- Future requests from the user will use this stored context to identify the user without requiring re-authentication.

10. Response to User (Step 10):

- The secure resource is now accessible, and a response is returned to the user.

Image 2: Simplified Sequence Flow

This image represents the same flow in a more **technical and diagrammatic manner**, breaking down the components.

Key Components:

1. Authentication Filters:

- **AuthorizationFilter**: Checks if the user is authorized to access the requested resource.
- **DefaultLoginPageGeneratingFilter**: Redirects the user to the login page if they are not authenticated.
- **UsernamePasswordAuthenticationFilter**: Captures the username and password from the login form and initiates the authentication process.

2. UsernamePasswordAuthenticationToken:

- This is the token created with the user's submitted credentials.
- It is passed to the `ProviderManager` for authentication.

3. ProviderManager:

- The main entry point for authentication in Spring Security.
- Delegates the authentication process to an `AuthenticationProvider`.

4. DaoAuthenticationProvider:

- A default implementation of `AuthenticationProvider` that uses `UserDetailsService` to fetch user details and validate credentials.
- Calls `UserDetailsService.loadUserByUsername()`.

5. Custom UserDetailsService (EazyBankUserDetails):

- Retrieves the user's information (e.g., username, password, roles) from the database or other data sources.
- Returns a `UserDetails` object containing this information.

6. PasswordEncoder:

- Used by the `DaoAuthenticationProvider` to encode and validate the password securely.

7. Authentication Success:

- An `Authentication` object is created and stored in the `SecurityContext`.
-

Key Takeaways:

1. Custom `UserService`:

- Allows retrieving user information from a custom data source or using a custom schema.
- Must implement the `loadUserByUsername(String username)` method.

2. Role of Filters:

- Spring Security filters are responsible for intercepting requests and triggering the authentication process.

3. Separation of Concerns:

- Filters handle requests.
- The `ProviderManager` delegates authentication to providers.
- The `DaoAuthenticationProvider` validates credentials using the `UserService` and `PasswordEncoder`.

4. Password Handling:

- Passwords are never stored in plain text; they are hashed using a `PasswordEncoder`.
- Password validation ensures secure comparison.

5. Authentication Object:

- Created on successful authentication.
 - Stored in the `SecurityContext` to identify the user in subsequent requests.
-

Let me know if you need clarification or additional details on any step!