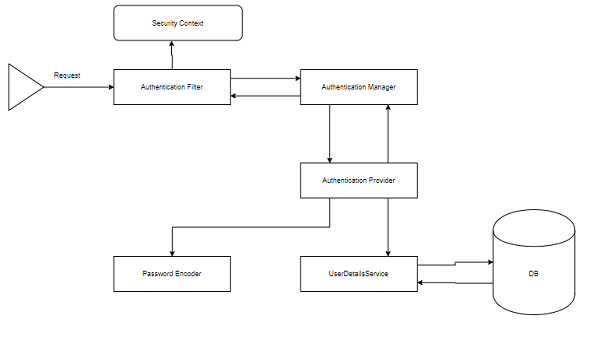
SPRING SECURITY-



UserDetailService is responsible for connecting with the database and taking user data from database.Password encoder done with hashing.At authentication provider it is decoded what is the form of authentication.

### AuthenticationFilter

This is the filter that intercepts requests and attempts to authenticate it. In Spring Security, it converts the request to an Authentication Object and delegates the authentication to the AuthenticationManager.

### AuthenticationManager

It is the main strategy interface for authentication. It uses the lone method authenticate() to authenticate the request. The authenticate() method performs the authentication and returns an Authentication Object on successful authentication or throw an AuthenticationException in case of authentication failure. If the method can’t decide, it will return null. The process of authentication in this process is delegated to the AuthenticationProvider which we will discuss next.

### AuthenticationProvider

The AuthenticationManager is implemented by the ProviderManager which delegates the process to one or more AuthenticationProvider instances. Any class implementing the AuthenticationProvider interface must implement the two methods – authenticate() and supports(). First, let us talk about the supports() method. It is used to check if the particular authentication type is supported by our AuthenticationProvider implementation class. If it is supported it returns true or else false. Next, the authenticate() method. Here is where the authentication occurs. If the authentication type is supported, the process of authentication is started. Here is this class can use the loadUserByUsername() method of the UserDetailsService implementation. If the user is not found, it can throw a UsernameNotFoundException.

On the other hand, if the user is found, then the authentication details of the user are used to authenticate the user. For example, in the basic authentication scenario, the password provided by the user may be checked with the password in the database. If they are found to match with each other, it is a success scenario. Then we can return an Authentication object from this method which will be stored in the Security Context, which we will discuss later.

### UserDetailsService

It is one of the core interfaces of Spring Security. The authentication of any request mostly depends on the implementation of the UserDetailsService interface. It is most commonly used in database backed authentication to retrieve user data. The data is retrieved with the implementation of the lone loadUserByUsername() method where we can provide our logic to fetch the user details for a user. The method will throw a UsernameNotFoundException if the user is not found.

### PasswordEncoder

Until Spring Security 4, the use of PasswordEncoder was optional. The user could store plain text passwords using in-memory authentication. But Spring Security 5 has mandated the use of PasswordEncoder to store passwords. This encodes the user’s password using one its many implementations. The most common of its implementations is the BCryptPasswordEncoder. Also, we can use an instance of the NoOpPasswordEncoder for our development purposes. It will allow passwords to be stored in plain text. But it is not supposed to be used for production or real-world applications.

### Spring Security Context

This is where the details of the currently authenticated user are stored on successful authentication. The authentication object is then available throughout the application for the session. So, if we need the username or any other user details, we need to get the SecurityContext first. This is done with the SecurityContextHolder, a helper class, which provides access to the security context. We can use the setAuthentication() and getAuthentication() methods for storing and retrieving the user details respectively.

Moving on, let’s now discuss the three custom implementations we are going to use for our application.

### Form Login

When we add Spring Security to an existing Spring application it adds a login form and sets up a dummy user. This is Spring Security in auto-configuration mode. In this mode, it also sets up the default filters, authentication-managers, authentication-providers, and so on. This setup is an in-memory authentication setup. We can override this auto-configuration to set up our own users and authentication process. We can also set up our custom login method like a custom login form. Spring Security only has to made aware of the details of the login form like – the URI of the login form, the login processing URL, etc.. It will then render our login form for the application and carry out the process of authentication along with the other provided configurations or Spring’s own implementation.

This custom form setup will only have to abide by certain rules to be integrated with Spring Security. We need to have a username parameter and a password parameter and the parameter names should be “username” and “password” since those are the default names. In case, we use our own parameter names for these fields in the custom we have to inform Spring Security of those changes using the usernameParameter() and passwordParameter() methods. Similarly, for every change we do to the login form or the form login method, we will have to inform Spring Security of those changes with appropriate methods so that it can integrate them with the authentication process.

### Login with a Database

As we discussed, Spring Security automatically provides an in-memory authentication implementation by default. We can override this by authenticating users whose details are stored in a database. In this case, while authenticating a user, we can verify the credentials provided by the user against those in the database for authentication. We can also let new users register in our application and store their credentials in the same database. Also, we can provide methods to change or update their passwords or roles or other data. As a result, this provides us with persistent user data which can be used for longer periods of time.

### Login Attempts Limit

To limit login attempts in our application we can use Spring Security’s isAccountNonLocked property. Spring Security’s UserDetails provides us with that property. We can set up an authentication method wherein, if any user or someone else provides incorrect credentials for more than a certain number of times, we can lock their account. Spring Security disables authentication for a locked user even if the user provides correct credentials. This is an in-built feature provided by Spring Security. We can store the number of incorrect login attempts in our database. Then against each incorrect authentication attempt, we can update and check with the database table. When the number of such attempts exceeds a given number, we can lock the user out of their account. Consequently, the user will not be able to log in again until their account is unlocked.