**Spring Security**

Spring Security is a robust security framework that provides authentication, authorization, and other security features for Spring-based applications. It enables developers to easily implement security features, such as user authentication and role-based access control, using a set of standardized authentication and authorization mechanisms. With Spring Security, developers can secure web applications, RESTful APIs, and microservices using various authentication mechanisms, including form-based authentication, token-based authentication, OAuth, and OpenID Connect. It is widely used in enterprise applications to ensure the security of sensitive data and resources.

**Why Spring Security ?**

Spring security is a framework which is pre-written and considered most of the security scenarios. It is difficult for us to write the all custom code again from the scrath. With minium configuration we can secure our spring boot application. Spring security handles common vulnarabilties like CORS, CSRF etc.

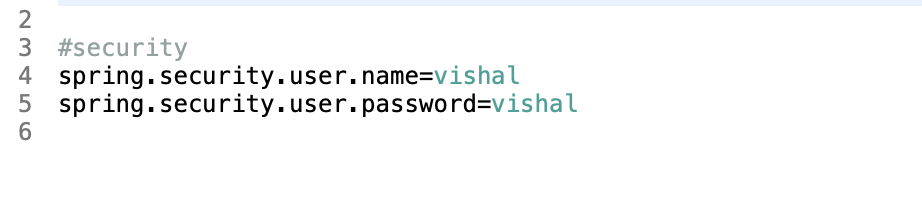
Adding Spring security in Spring boot Application

Create a spring boot application from spring.start.io selecting maven, java 17 and default selected spring boot version. Add spring web, lombok, spring security as a dependecies.

When you start the application, a password will be printed in the console. Create a Rest Api and try to hit it from the browser. A login screen appears which asks us to login.

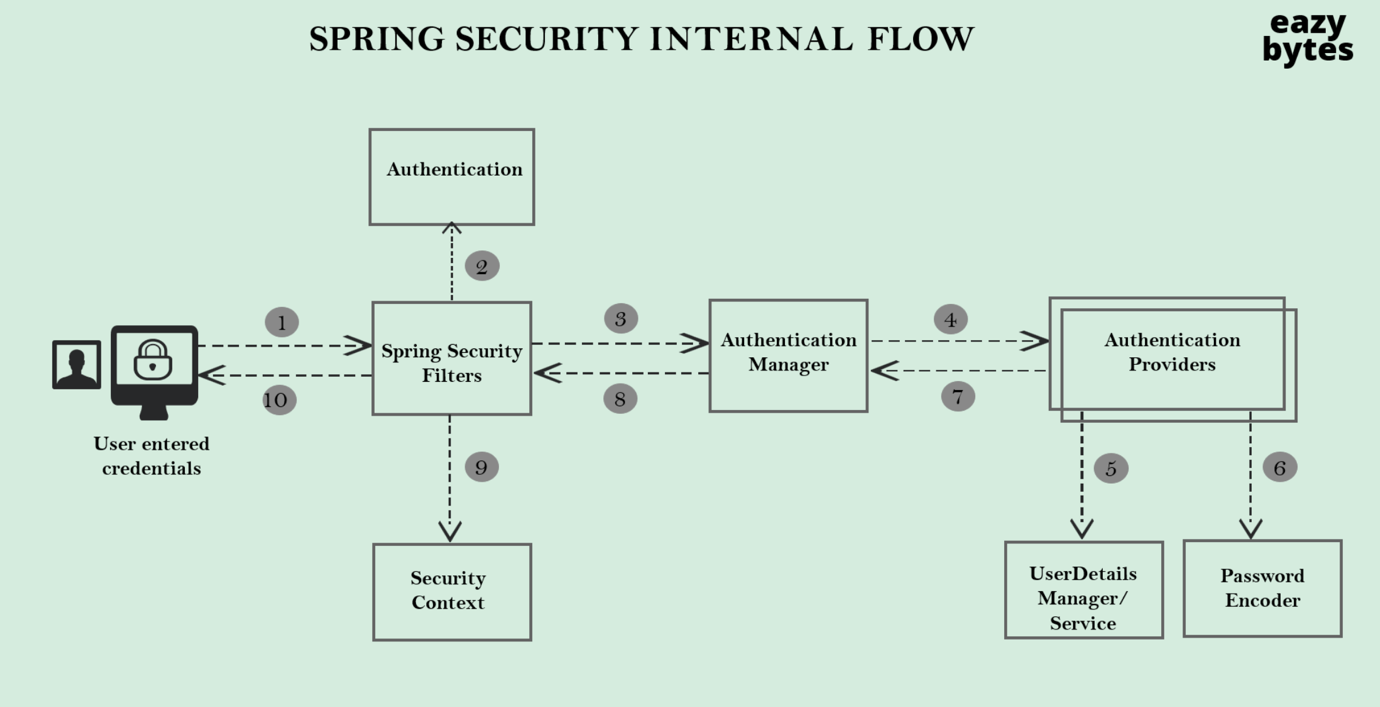
The default user name is “user” and password is printed in the console.

If you want to customize the user name and password we can configure that in the application.properties file.



Now you need use above username and password for authentication.

**Spring security Internal Flow**



**Spring Security Filters**

A series of Spring Securityfilters intercept each request & work together to identify if Authentication is required or not. If authentication is

required, accordingly navigate the user to login page or use the existingdetails stored during initial authentication.

**Authentication**

Filters like UsernamePasswordAuthenticationFilter will extract username/passwordfrom HTTP request &prepare Authentication type object. Because Authentication is the core standard of storing authenticated user details insdie Spring Security framework.

**AuthenticationManager**

Once received request from filter, it delegates the validating of the user details to the authentication providers available. Since there can be multiple providers inside an app, it is the responsibility of the Authentication Manager to manage all the authentication providers available.

**AuthenticationProvider**

AuthenticationProviders has all the core logic of validating user detailsfor authentication.

**UserDetailsManager/UserDetailsService**

UserDetailsManager/UserDetailsService helps in retrieving; creating; updating; deleting the User Details from the DB/storage systems.

**PasswordEncoder**

Service interface that helps in encoding & hashing passwords. Otherwise we may have to live with plain text passwords @

**SecurityContext**

Once the request has been authenticated, the Authentication will usually be stored in a thread-local Security Context managed by the SecurityContextHolder. This helps during the upcoming requestsfrom the same user.

We can configure InMemorUserNames and passwords with default password encoders instead of declaring them in application.proeprties.



We can configure InMemorUserNames and passwords with **NoOppasswordEncoders**. In this we convey to the spring security that we want our passwords to be in a plan text. But this way is not recommened in production applications.



In the Spring security interflow, Authentication providers take help from the UserDetailsService to get the information about the user from the InMemoryDatabase/RelationalDatabase.

We have InMemoryUserDetailsManager to handle the users which are configured via InMemory like in the above example, but in production on one uses it.

We have a JdbcUserDetailsManager to handle the users from the relational database, but asks the developer to create a database structure in a specific way which is again not recommended in the production application.

So we can implements UserDetailsService and override the “loadUserByUserName” method to write our custom authentication logic. In this logic we manually get the user from the database and return the UserDetails. Internally from the UserDetails Spring will do the password check and validate the credentials.

If we have a NoOpsPasswordEncoders in the classpath, the spring security simplies mathes the password entered by user with password in Database as a plain Text.



**Encoding vs Encryption vs Hashing**

**Encoding** is changing the plan text to a different string which is reversible and any one get the value of encoded string. Most Popular encoded mechanisms are Base64, ASCII, UNICODE.

**Encryption** is converting the plain text into different string using a secret key. It is also a reversible process and person with secret key can decrypt the password/strings. So in many web applications Encryption is not recommended.

**Hashing** is most used mechanism to convert the password/strings to hash string. It is not reversible so it is very secure. When user tries to login with plan text password, the algorithm converts the plain text password into hash string and compares the hash string with the hashed password in the database. If both the values are same then user can login in. So we can see even if the database is comprimised, it is almost impossible to guess the value of the hashed password as it is not reversible.

**Password Encoders**

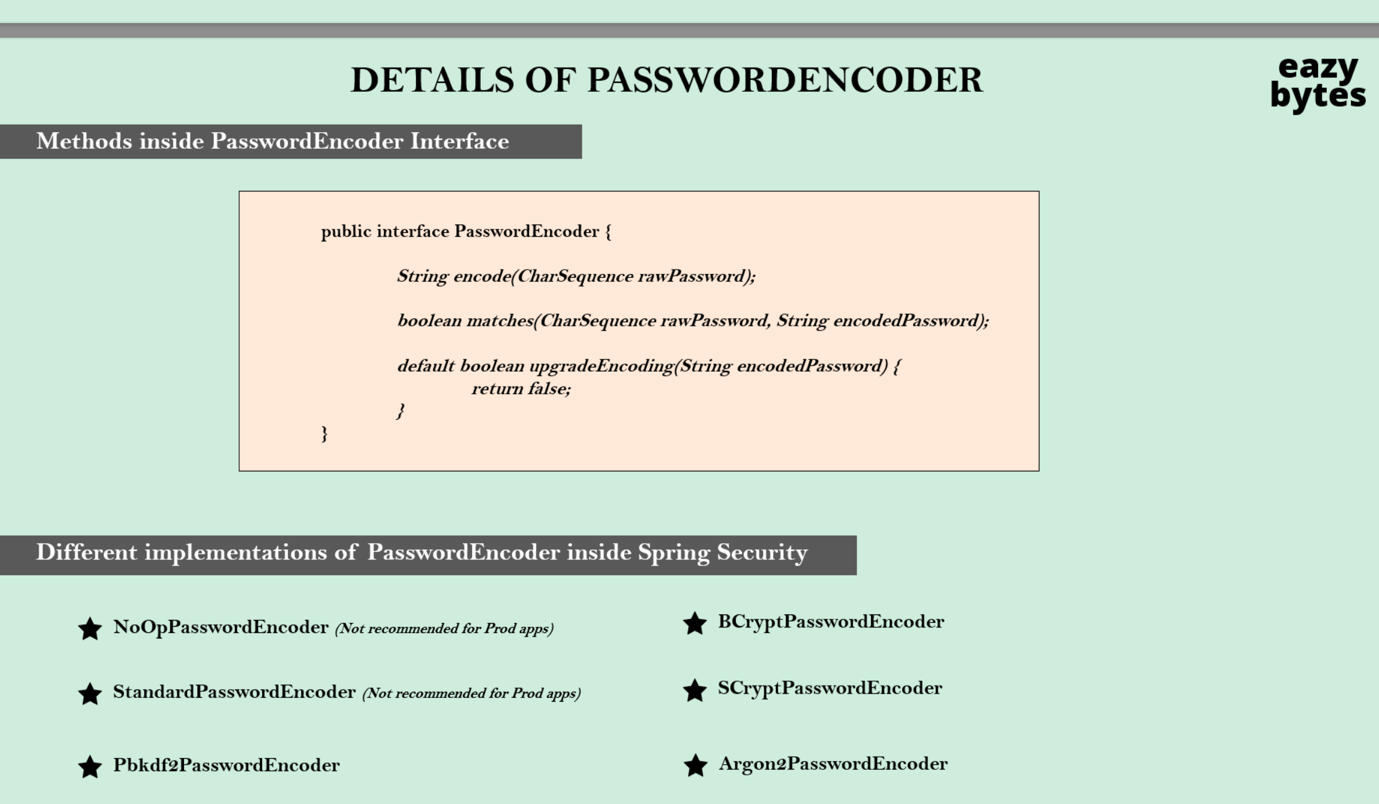
We have below 6 password Encoders in spring-security. In the below image left side password encoders are not recommended for the production.

**NoOpsPasswordEncoder**: Deals the password as a plain text.

**StandardPasswordEncoder**: Currently using in legacy application.

**Pbkdf2PasswordEncoder**: Used in 5 years back but now not recommended in production because a hacker with resources of high compuation power and memory can crack the passwords using “Brute-Force-Attack”

**A Brute-Force-Attack** is nothing from trail and error approach, if the hacker hacked the database and got the hashed password string, he will try random common passwords againt the password string, if he is lucky he might get the match.

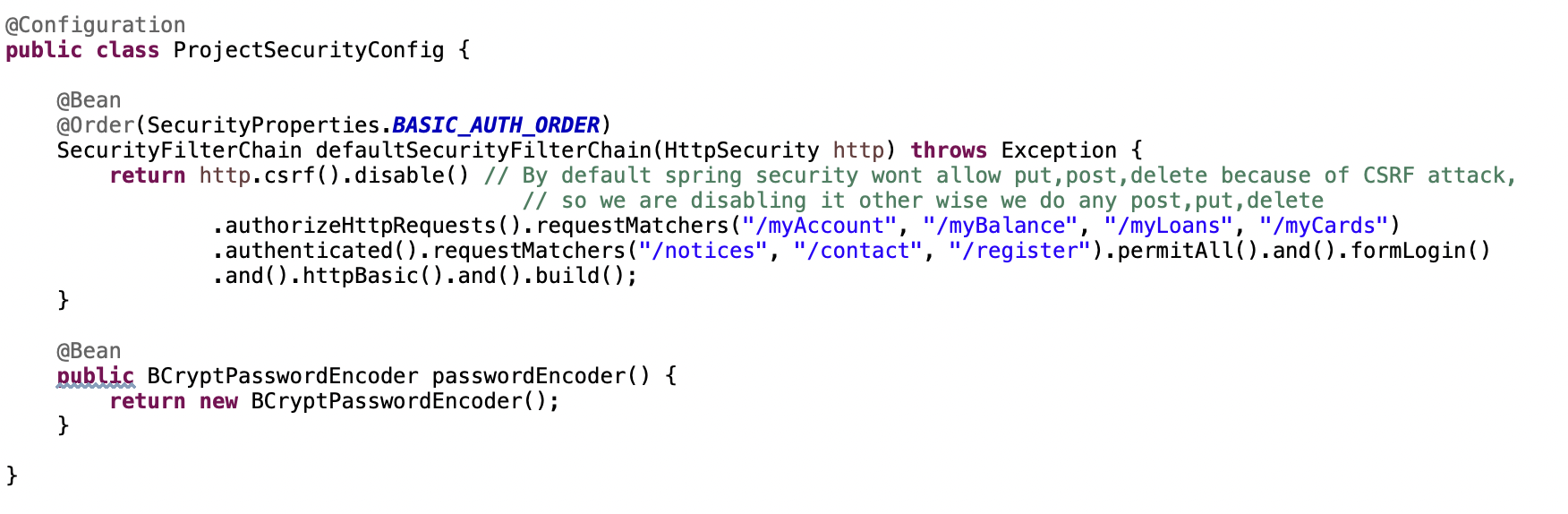


**BcryptPasswordEncoder** uses Bcrypt Hashing algorithm which uses computation power.It mostly widly used algorithm.

**ScryptPasswordEncoder** used Computation power and Memory.

**Argon2PasswordEncoder** uses computation power, memory and multiple threads.

**Implementing the BcryptPasswordEncoder in code**



We defined *BcryptedPasswordEncoder* bean in the spring container. We can use BcryptedPasswordEncoder to hash the password and store in the database.



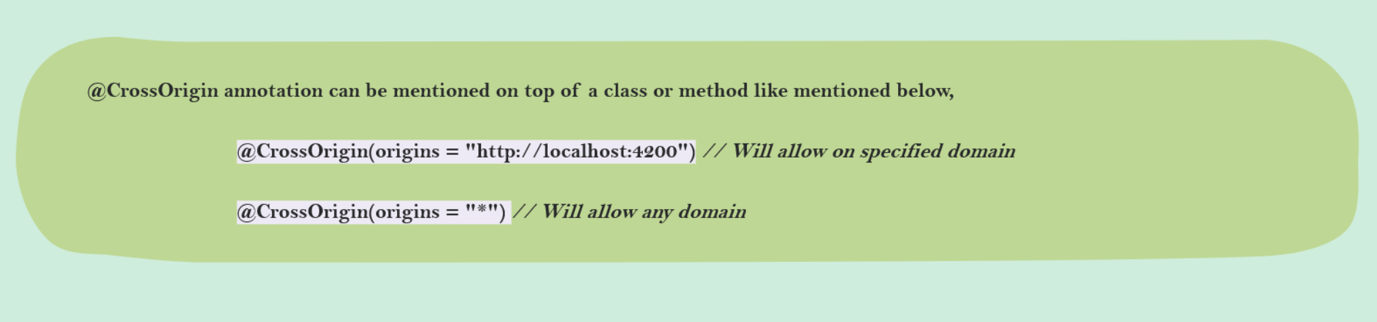
If we want to implement our custom authentication logic without using default DaoAuthenticationProvider can implement a Provider class.

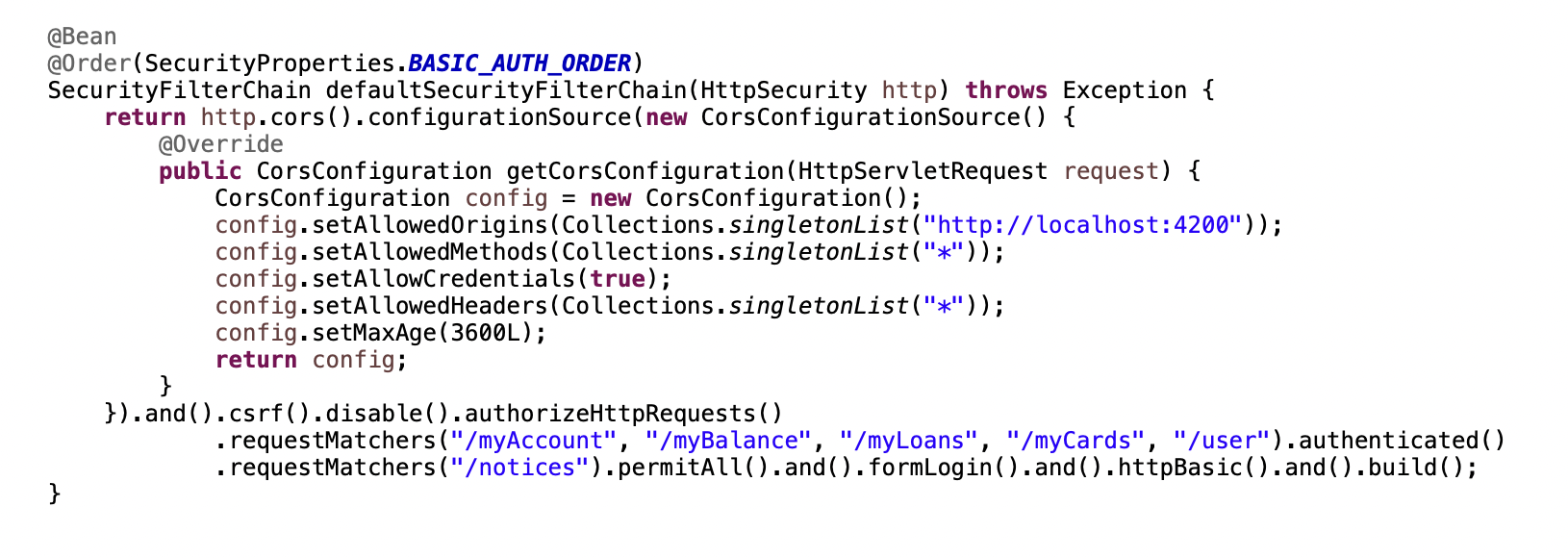


**CORS**

Cross origin resource sharing is a protocol that enables to servers to speak with each other. It is not an attack, it is an extra security from the browser end. For the first time, browser send an pre-flight request, in that request it says “hey backend port 4200 wanna send an api request to you its that ok” if backend says it ok then other requests to backend will get succeed.

Configuration of CORS in the backend can be 2 two ways. One with using @Crossorigin annotation and configuration class.





**CSRF**

Cross-Site Request Forgery (CSRF or XSRF) attack aims to perform an operation in a web application on behalf of a user without their explicit consent. In general, it doesn't directly steal the user's identity, but it exploits the user to carry out an action without their will.

Refere the author provided pdf for detailed information.

**Explanation of CSRF attack:**

Think you logged in your netflix account. Netflix backend will send you some cokkies that will store in the browser. This cookies will only be available to domain “netflix.com/\*”. Other domains cant acces this cookies. So attacker will create a page or an add which inside contains an html form element which sends a request to “netflix.com” to change the passowrd or update password, as its sends request in a form browser will attach the netflix which are saved before. So that’s how attackers might steal your data.

**Solution for CSRF attack**

We ask backend to send a CSRF cookie at the login time, after successful login we will send this CSRF cookie as header to the backend, backend validates this cookie and it is valid then only it will proceed the request to controller.The attacker don’t know from where/what we are expecting in the backend. So adding extra header in api request saves us from CSRF attack. Accordingly we need to configure the backend code.

CsrfTokenRequestAttributeHandler, CookieCsrfTokenRepository classes in the spring-security takes care of putting the cookie with name XSRF-TOKEN and reading header with name X-XSRF-TOKEN in request. They will validate the token from header it is valid then the request will be procedd to controller.

We need to configure the filter (CsrfCookieFilter) as well.



Configuring the custom CsrfCookieFilter



**Authentication and Authorization**

*Authentication* is all about checking the identity/access of the user to the application.

*Authorization* is how much access do you have in the application.

**Authorities and Roles**

*Authority* is like an individual privilege or an action. Restricting access in a fine-grained manner.

Ex: VIEWACCOUNT, VIEWCARDS

*ROLE* is a group of privileges/actions. Restricting access in a coarse-grained manner.

Ex: ROLE\_ADMIN, ROLE\_USER

**Code Implementation for Authorities and Roles**

First we need to have a table authorities and a foregin key to customer table.

Then create a Entity class for Authorities with @ManyToOne relationship to customer enity.

Then adding all the entites from database response to the UserDetails object in ProviderClass or UserDetailsService.







**Creating the Custom filters**

We can have our own filters which can be used to implement some custom logic before/after the in built Filters. We can configure accordingly.

