Spring Security

Table of Contents

[1 General Web Security Concepts 4](#_Toc78639148)

[1.1 Cryptography 4](#_Toc78639149)

[1.1.1 Hash Functions 4](#_Toc78639150)

[1.1.2 Encryption Functions 4](#_Toc78639151)

[1.1.3 Use Cases 4](#_Toc78639152)

[1.2 Storing Passwords, Salt and Pepper 6](#_Toc78639153)

[1.2.1 Protecting Passwords with Salt and Pepper 6](#_Toc78639154)

[1.3 Basic access authentication (Http Basic Auth) 7](#_Toc78639155)

[2 Spring Security Concepts 9](#_Toc78639156)

[2.1 Authentication 9](#_Toc78639157)

[2.1.1 Knowledge Based 9](#_Toc78639158)

[2.1.2 Possession Based 9](#_Toc78639159)

[2.1.3 Multi-Factor Authentication 9](#_Toc78639160)

[2.2 Authorization 10](#_Toc78639161)

[2.3 Principal 10](#_Toc78639162)

[2.4 Granted Authority 10](#_Toc78639163)

[2.5 Roles 10](#_Toc78639164)

[3 Adding Spring Security to a Project 11](#_Toc78639165)

[4 Configure Authentication 11](#_Toc78639166)

[4.1 In-memory Authentication 12](#_Toc78639167)

[4.2 JDBC Authentication 13](#_Toc78639168)

[4.3 LDAP Authentication 13](#_Toc78639169)

[4.4 Kerberos Authentication 14](#_Toc78639170)

[4.5 JWT Authentication 14](#_Toc78639171)

[4.6 Oauth2 Authentication? 14](#_Toc78639172)

[5 Configure Authentication / Login Mechanisms? 14](#_Toc78639173)

[5.1 Form Login 14](#_Toc78639174)

[5.2 Basic Http Auth / Login 14](#_Toc78639175)

[6 Configure Authorization 15](#_Toc78639176)

[7 How Spring Security Authentication works 15](#_Toc78639177)

[7.1 Authentication Details 16](#_Toc78639178)

[7.1.1 AuthenticationProvider 16](#_Toc78639179)

[7.1.2 AuthenticationManager 17](#_Toc78639180)

[7.1.3 UserDetailsService 17](#_Toc78639181)

[8 Custom Login Page 19](#_Toc78639182)

[9 Registration with Spring Security 20](#_Toc78639183)

[9.1 Password Encoding 20](#_Toc78639184)

Resources used:

* Five Spring Security Concepts - Authentication vs authorization - Java Brains Brain Bytes: <https://www.youtube.com/watch?v=I0poT4UxFxE>
* Adding Spring Security to new Spring Boot project - Java Brains Brain Bytes: <https://www.youtube.com/watch?v=PhG5p_yv0zs>
* How to configure Spring Security Authentication - Java Brains: <https://www.youtube.com/watch?v=iyXne7dIn7U>
* Protecting passwords with Salt & Pepper | CISSP Answers: <https://www.youtube.com/watch?v=eicDtA9Yu-A>
* Adding Salt to Hashing: A Better Way to Store Passwords: <https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>
* Passwords & hash functions (Simply Explained): <https://www.youtube.com/watch?v=cczlpiiu42M>
* Registration with Spring Security – Password Encoding: <https://www.baeldung.com/spring-security-registration-password-encoding-bcrypt>
* How Spring Security Authentication works - Java Brains: <https://www.youtube.com/watch?v=caCJAJC41Rk>
* Spring Security Architecture: <https://spring.io/guides/topicals/spring-security-architecture>
* Spring Security | FULL COURSE: <https://www.youtube.com/watch?v=her_7pa0vrg>

1. General Web Security Concepts
   1. Cryptography

Asd

* + 1. Hash Functions

They provide a mapping between an arbitrary length input, and a (usually) fixed length (or smaller length) output. It can be anything from a simple crc32, to a full blown cryptographic hash function such as **MD5** or **SHA1/2/256/512**. The point is that **there's a one-way mapping going on, they are ONE WAY functions.** It's **always a many:1** **mapping** (meaning there will always be collisions) since every function produces a smaller output than it's capable of inputting (If you feed every possible 1mb file into MD5, you'll get a ton of collisions). They are **hard** (or **impossible** **in practicality**) to **reverse**.

* + 1. Encryption Functions

They provide a **1:1 mapping** between an arbitrary length input and output**, TWO WAY function**. And **the result are always reversible**. The important thing to note is that it's reversible using some method. And it's always 1:1 for a given key. Now, there are multiple input:key pairs that might generate the same output (in fact there usually are, depending on the encryption function). Good encrypted data is indistinguishable from random noise. This is different from a good hash output which is always of a consistent format.

* + 1. Use Cases

Use a **hash function** when you want to compare a value but can't store the plain representation (for any number of reasons). Passwords should fit this use-case very well since you don't want to store them plain-text for security reasons (and shouldn't). But what if you wanted to check a filesystem for pirated music files? It would be impractical to store the 3 MB per music file in order to compare it later. So instead, take the hash of the file, and store that (md5 would store 16 bytes instead of 3MB). That way, you just hash each file and compare to the stored database of hashes (This doesn't work as well in practice because of re-encoding, changing file headers, etc, but it's an example use-case).

Use a **hash function** when you're checking validity of input data. That's what they are designed for. If you have 2 pieces of input, and want to check to see if they are the same, run both through a hash function. The probability of a collision is astronomically low for small input sizes (assuming a good hash function). That's why it's recommended for passwords. For passwords up to 32 characters, md5 has 4 times the output space. SHA1 has 6 times the output space (approximately). SHA512 has about 16 times the output space. You don't really care what the password was, you care if it's the same as the one that was stored. That's why you should use hashes for passwords.

**Hash** **functions** are also great for signing data. For example, if you're using HMAC, you sign a piece of data by taking a hash of the data concatenated with a known but not transmitted value (a secret value). So, you send the plain-text and the HMAC hash. Then, the receiver simply hashes the submitted data with the known value and checks to see if it matches the transmitted HMAC. If it's the same, you know it wasn't tampered with by a party without the secret value. This is commonly used in secure cookie systems by HTTP frameworks, as well as in message transmission of data over HTTP where you want some assurance of integrity in the data.

**Hashing** is useful if you want to send someone a file. But you are afraid that someone else might intercept the file and change it. So a way that the recipient can make sure that it is the right file is if you post the hash value publicly. That way the recipient can compute the hash value of the file received and check that it matches the hash value.

**Encryption** is good if you say have a message to send to someone. You encrypt the message with a key and the recipient decrypts with the same (or maybe even a different) key to get back the original message. Credits

Use **encryption** whenever you need to get the input data back out. Notice the word need. If you're storing credit card numbers, you need to get them back out at some point, but don't want to store them plain text. So instead, store the encrypted version and keep the key as safe as possible.

**A note on hashes for passwords:**

A key feature of cryptographic hash functions is that they should be very fast to create, and very difficult/slow to reverse (so much so that it's practically impossible). This poses a problem with passwords. If you store sha512(password), you're not doing a thing to guard against **rainbow tables** or **brute force attacks**. Remember, the hash function was designed for speed. So it's trivial for an attacker to just run a dictionary through the hash function and test each result.

**Adding a** **salt** helps matters since it adds a bit of unknown data to the hash. So instead of finding anything that matches **md5(foo)**, they need to find something that when added to the known salt produces **md5(foo + salt)** (which is very much harder to do). But it still doesn't solve the speed problem since if they know the salt it's just a matter of running the dictionary through.

Text

Description automatically generatedSo, there are ways of dealing with this. One popular method is called **key strengthening** (or **key stretching**). Basically, you iterate over a hash many times (thousands usually). This does two things. First, it slows down the runtime of the hashing algorithm significantly. Second, if implemented right (passing the input and salt back in on each iteration) actually increases the entropy (available space) for the output, reducing the chances of collisions. A trivial implementation is:

* The bottom line, hash(password) is not good enough. hash(password + salt) is better, but still not good enough... Use a stretched hash mechanism to produce your password hashes…

**Another note on trivial stretching**

Text

Description automatically generatedDo not under any circumstances feed the output of one hash directly back into the hash function:

* 1. Storing Passwords, Salt and Pepper

Passwords can be basically stored in 3 ways in a database:

* **Plain text**: This is really reaaally bad for obvious reasons.
* **Encrypted Passwords**: Encrypting the password with a secret key and storing the result in a database is also a bad idea. The encryption key needs to be stored somewhere on the system. Since Encrypting is two way, if an attacker has access to the key, then he can decrypt ALL the stored passwords of every user.
* **Storing the hash value**: This is the preferred way. Since hashing is one way, the resulting string cannot be reversed to access the original password. Not even the system will know the plain password yet we can still easily authenticate users by hashing their passwords and then comparing the resulting string to the entries in the database.

But there is another attack vector called **Rainbow Tables**.

Hackers gather all the leaked/stolen passwords into a big database, which is called a rainbow table. In this table, there are a ton of commonly used and also leaked password in plain text and the results of all the hashing algorithms hashing that password. An attacker could get access to a companies database. It will be extremely fast to compare the hash values of all the entries.

**Rainbow Table versus Dictionary/Brute Force**

A rainbow table is generally an offline only attack. In a brute force attack or dictionary attack, you need to spend time either sending your guess to the real system to running through the algorithm offline. Given a slow hashing or encryption algorithm, this wastes time. Also, the work being done cannot be reused.

* + 1. Protecting Passwords with Salt and Pepper

Salt and pepper are used to solve the Rainbow Table problem.

* **Salt**: A unique, non-secret value which is appended to the user’s password before it is hashed.   
  Salts are stored in the database (in plain text). Each password in the database will have its own, randomly generated salt value associated with it.
* **Pepper**: A secret value which is appended to the user’s password before it is hashed.  
  Peppers are not stored in the database. Peppers are NOT unique for each password. It can be thought as a secret key that is stored in a secure system file. It is typically the same value appended to every password.

Most hash functions are optimized for speed. A relatively new system (SHA3 – GTX 970) can do 300 million hashed/second. The speed of a hash function is a positive thing in certain areas however when it comes to storing/hashing passwords, we don’t want the hash algorithm to be fast.

If an attacker accesses a DB with salts, he won’t be able to do his ms fast rainbow table lookups because the hashes will be different. When the salt is unique for each hash, we inconvenience the attacker by now having to compute a hash table for each user hash. To make it even slower, the best practice is to use a hash algorithm which is deliberately being slowed down. This creates a big bottleneck for the attacker.

Examples for such “slow” hashing algorithms are: **bcrypt**, **scrypt**, **argon2**

* bcrypt(password, salt, cost)

These algorithms take the password, the salt and a cost argument. The cost defines the number of rounds the algorithm goes through. This effectively slows it down. We can increase the cost over time to counter computers getting faster.

While the attacker may be able to crack one password, cracking all passwords will be unfeasible. Regardless, when a company experiences a data breach, it is impossible to determine which passwords could have been cracked and therefore all passwords must be considered compromised. A request to all users to change their passwords should be issued by the company right away. Upon password change, a new salt should be generated for each user as well.

* More details: <https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>

A picture containing text, iPod

Description automatically generatedA really good protection is done if multiple of these techniques are used together.   
For example, Dropbox uses this multi-layer protection.

1. Get your password and hash it without a salt
2. They run the resulting value through a bcrypt algorithm, with a salt and a cost of 10 (slows down brute force attacks)
3. The resulting hash is then encrypted with AES (Advanced Encryption Standard). The encryption key is not stored in the database but kept separately.

If an attacker breaches the Dropbox database, they will have to peel away each protective layer to get to the plain text password.

* Password Hashing: add salt + pepper or is salt enough?: <https://security.stackexchange.com/questions/3272/password-hashing-add-salt-pepper-or-is-salt-enough>
  1. Basic access authentication (Http Basic Auth)

In the context of an HTTP transaction, basic access authentication is a method for an HTTP user agent (e.g. a web browser) to provide a **user name** and **password** when making a request. In basic HTTP authentication, a request contains a **header field** of the form Authorization: Basic <credentials>, where credentials is the **base64** encoding of id and password joined by a single colon (username:password).

HTTP Basic authentication (BA) implementation is the simplest technique for enforcing access controls to web resources because **it does not require** cookies, session identifiers, or login pages; rather, HTTP Basic authentication uses standard fields in the HTTP header, removing the need for handshakes.

The BA mechanism provides **no confidentiality protection** for the transmitted credentials. They are merely encoded with Base64 in transit, but not encrypted or hashed in any way. Therefore, Basic Authentication is typically used in conjunction with HTTPS to provide confidentiality.

* Because the BA field has to be sent in the header of **each HTTP request** (there is no sessions, every new request require the credentials to be passed again), the web browser needs to cache credentials for a reasonable period of time to avoid constantly prompting the user for their username and password. **Caching policy differs between browsers**. Microsoft Internet Explorer caches the credentials for 15 minutes by default.

HTTP does not provide a method for a web server to instruct the client to "**log** **out**" the user. However, there are a number of methods to clear cached credentials in certain web browsers. One of them is redirecting the user to a URL on the same domain containing credentials that are intentionally incorrect. However, this behavior is inconsistent between various browsers and browser versions. Microsoft Internet Explorer offers a dedicated JavaScript method to clear cached credentials:



**Server Side**

When the server wants the user agent to authenticate itself towards the server, the server must respond appropriately to unauthenticated requests. To unauthenticated requests, the server should return

* a response whose header contains a **HTTP 401 Unauthorized** status and
* a **WWW-Authenticate** field: The WWW-Authenticate field for basic authentication is constructed as following: WWW-Authenticate: Basic realm="User Visible Realm" The server may choose to include the charset parameter from RFC 7617: WWW-Authenticate: Basic realm="User Visible Realm", charset="UTF-8" This parameter indicates that the server expects the client to use UTF-8 for encoding username and password (see below).

**Client Side**

When the user agent wants to send authentication credentials to the server, it may use the **Authorization** field. The Authorization field is constructed as follows:

1. The **username** and **password** are combined with a **single colon** (:). This means that the username itself cannot contain a colon.
2. The resulting string is encoded into an octet sequence. The character set to use for this encoding is by default unspecified, as long as it is compatible with US-ASCII, but the server may suggest use of UTF-8 by sending the charset parameter.
3. The resulting string is encoded using a variant of **Base64**.
4. The authorization method and a space (e.g. "Basic ") is then prepended to the encoded string.

For example, if the browser uses Aladdin as the username and OpenSesame as the password, then the field's value is the base64-encoding of Aladdin:OpenSesame, or QWxhZGRpbjpPcGVuU2VzYW1l. Then the Authorization header will appear as:



**URL Encoding**

A client may avoid a login prompt when accessing a basic access authentication by prepending **username:password@** to the hostname in the URL. For example, the following would access the page index.html at the web site www.example.com with the secure HTTPS protocol and provide the username Aladdin and the password OpenSesame credentials via basic authorization:

This has been deprecated by RFC 3986: Use of the format "user:password" in the user info field is deprecated. **Some modern browsers therefore no longer support URL encoding of basic access credentials**. This prevents passwords from being sent and seen prominently in plain text, and also eliminates (potentially deliberately) confusing URLs.

Asd

asd

1. Spring Security Concepts
   1. Diagram

      Description automatically generatedAuthentication

Spring security can be imagined as a security guard which stands between the users and the application. When a user request something from the application, the guard asks:

* Who are you? (Authentication: answering this question)
* What do you want?

You answer the first question by telling the guard who you are but that is ofc not enough, you also have to prove it. The proof happens usually by providing some kind of an ID. Something that only you can provide and an impersonator can’t.

And then you tell what do you want or what you would like to do. You don’t need to provide any proof for the second part because there is no point in anyone lying here.

Most applications have an ID for each user stored in the system. So when you access the system, you have to tell it which one of those millions of accounts in the system belongs to you. But you also need to prove that you really are who you say you are (authentication).

* + 1. Knowledge Based

The most common way of proving that is telling the password which you set when you created your account. This type of authentication is called knowledge base authentication because it is based on something only you can know. This can be:

* Password
* Pin code
* Answer to a secret / personal question

It is simple to implement and use but the disadvantage is that if someone steals your password then they can impersonate you.

* + 1. Possession Based

This is for instance when an app sends you a text message with an OTP. The system checks if you the phone which only the user of this account can possess. Phones are harder to steal than passwords.

* Phone / Text messages
* Key cards and badges
* Access token device
  + 1. Multi-Factor Authentication

Some apps do the combinations of these two: enter your password and then also verify you possess the phone by entering the secret in the text we’ve just sent you.

* 1. Authorization

This is the security guard examining what you want and deciding if you are allowed to do that. It is generally a yes or no answer. Everyone are allowed to login to the system but only certain people are allowed to see some web pages.

* The rules that determine who is allowed to do what.
* For authorization, you first need authentication: you need to be sure who this person really is to be able to then check if that person is allowed to do certain operations
  1. Principal

This is a spring security concept but also used in some other frameworks.

* The principal is the person you have identified through the process of authentication. In other words, the principal is the currently logged in user/account.

After a user gives his user id and password, the app establishes the principal and it remembers it. This is the reason why you authenticate with an application only once and don’t need to enter your credentials again.

* One user can have multiple ID’s / accounts. When the user logs in using account A, then the principal corresponds to account A. The user can switch the currently logged in account to account B and the principal will change to account B.
  1. Granted Authority

How does authorization happen? How does the app decide whether or not to allow an operation for a given user?

There are pre-defined permissions which are set for users. This is called **Granted Authority**. The app can allow / authorize the user to do certain actions, only if the user has granted that authority to do so.

* Chart, bubble chart

  Description automatically generatedThese are fine grained permissions what a user can do.
* In Spring Security: Permissions can be named anything we want.

**Example Granted authorities**:

* PERMISSION\_PROFILE\_READ,
* ACCESS\_PROFILE\_EDIT,
* AUTHORITY\_PROFILE\_DELETE
  1. Roles

Fine grained permissions are authorities and more coarse grained group of authorities are roles

* In Spring Security we can name permissions anything we want, but for roles **we have to prefix the name**   
  with ‘**ROLE**’ prefix.

**Example Roles**:

* ROLE\_ADMIN,
* ROLE\_USER,
* ROLE\_SALES,
* ROLE\_MANAGEMENT.

In a real world app, only a role based authorization system might not be flexible enough. If the system has a ton of features and if we need to allow only certain ones for users then it might be difficult with only roles. Different users might have the same role but different privileges. In those scenarios it’s more common to define permission, instead of creating new roles.

1. Adding Spring Security to a Project

Graphical user interface, text

Description automatically generated

Adding the following dependency is enough to add spring security to your project:

Graphical user interface, application

Description automatically generatedMost spring boot starter dependencies won’t have an effect when you add them to the classpath. You also have to configure them before you can do something with them.

* Spring Security immediately starts working by default. It forwards each request to the /login endpoint and expects the user to authenticate himself.

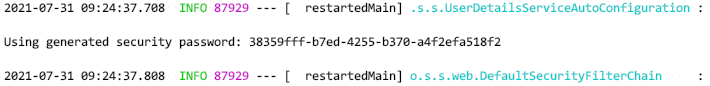
Spring security uses **Filters**. This is a common concept of servlet applications. All of these Spring applications are servlet applications after all.

Diagram

Description automatically generatedNormally a spring applications looks at a requests and then routes that request to the corresponding servlet which will process it. Filters are the mechanism which stand between the requests and the servlets. They intercept the requests, first get the request and then do some operations on them.

* Log each incoming request, stop certain requests before they reach the server.
* Check every request for a particular header or add a header

While servlets are mapped to URLs, filters can be applied to all requests, so they intercept every request. That is exactly what spring security does. Here is the default behavior:

* Adds mandatory authentication for all URLs (except the error page /error)
* Adds a login form (under /login and also a /logout endpoint)
* Validates and handles login error
* Creates a user and sets a default password (if no users are configured). Default user is user and a new random password is generated each time the application starts. See the logs.



You can customize this user in the application.properties:

1. Configure Authentication

We want to configure spring security so it doesn’t rely on only the default user but uses for example the user-data inside a database.

* The way to configure authentication is by effecting the AuthenticationManager. It has a method authenticate() and that either returns a successful authentication or throws an exception. This method does the authentication.

The way to affect what the AuthenticationManager does is working with the AuthenticationManagerBuilder class. For the most part you don’t directly work with the AuthenticationManager.

AuthenticationManagerBuilder asks:

1. “What type of authentication do you want?” -> In memory
2. Okay, then tell me the username, password and roles of your in-memory user are.

After we’ve given these configurations a new AuthenticationManager is built, which has the values we set.

**Step 1: Get hold of AuthenticationManagerBuilder**

**Step 2: Set the configuration on it**

Spring Security provides some configuration helpers to quickly get common authentication manager features set up in your application. The most commonly used helper is the AuthenticationManagerBuilder, which is great for setting up:

* in-memory,
* JDBC,
* LDAP user details
* or for adding a custom UserDetailsService
  1. Diagram, text, letter

     Description automatically generatedIn-memory Authentication

Spring security provides Hooks for developers so they can extend them and get access to certain functionality like Authentication or Authorization.

* Extending **WebSecurityConfigurerAdapter**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

* Since we’ve configured a user, spring security won’t create that default user anymore.
  1. JDBC Authentication

TODO How to setup JDBC authentication with Spring Security from scratch - Java Brains <https://www.youtube.com/watch?v=LKvrFltAgCQ>

* 1. LDAP Authentication

TODO

Spring Boot + Spring Security + LDAP from scratch - Java Brains: <https://www.youtube.com/watch?v=-wDUChgvYgU>

* 1. Kerberos Authentication

TODO

Kerberos Authentication Explained | A deep dive: <https://www.youtube.com/watch?v=5N242XcKAsM>

* 1. JWT Authentication

TODO

Spring Boot + Spring Security + JWT from scratch - Java Brains : <https://www.youtube.com/watch?v=X80nJ5T7YpE>

* 1. Oauth2 Authentication?

TODO

Implementing login with Facebook and Github from scratch - Java Brains: <https://www.youtube.com/watch?v=CWiwpvpCrro>

1. Configure Authentication / Login Mechanisms?

Asd

* 1. Form Login

Text

Description automatically generatedTODO

* 1. Basic Http Auth / Login

TODO

Graphical user interface, text, application

Description automatically generated

TODO - Any other login mechanisms?

1. Configure Authorization

Diagram, text, letter

Description automatically generatedGraphical user interface, text, application, email

Description automatically generatedWe will again extend the class **WebSecurityConfigurerAdapter** as we saw before when configuring the Authentication.

1. How Spring Security Authentication works

A **Filter** is a construct in a Spring application which lets you intercept requests. Diagram

Description automatically generatedNormally a spring application looks at an incoming request and then routes that request to the corresponding servlet which will process it. It is usually a 1 to 1 mapping. Filters are the mechanism which stand between the requests and the servlets. They intercept the requests, first get the request and then do some operations on them.

* Log each incoming request
* Stop certain requests before they reach the server.
* Check every request for a particular header or add a header

While servlets are mapped to URLs, filters can be applied to all requests, so they intercept every request.

Graphical user interface, text, application, email

Description automatically generatedThat is exactly what spring security does. When we add the spring security dependency to our project, it does the filter mapping to intercept ALL the requests (“/\*\*”) and maps it to spring securities own filter called DelegatingFilterProxy. If you were working on a non-Spring Boot app or wanted to add it manually, this is what it looks like (this is only for web applications. Doing method level security in non-web applications looks different):

A picture containing text

Description automatically generatedThe reason why this works has a clue in the name DelegatingFilterProxy. This catch all filter is actually just delegates the requests to other spring security specific filters, it does not do the job itself. Even in an unconfigured spring security app there are around 5-6 of these filters.

One of these filters is the AuthenticationFilter which intercepts all authentication requests and initiates the authentication process. There are also Authorization relevant filters.

* 1. Diagram

     Description automatically generatedAuthentication Details

Think of authentication as inputs and outputs. Typically, the inputs are the credentials from the user who is trying to authenticate. The output could be a Boolean but it is a little more than that. When the authentication is successful, it returns the **Principal**, the information about the logged in user.

Spring security keeps track of both the input and the output, using an object of type Authentication. Authentication is an internal spring security interface and the object is meant to hold credentials before authentication and once the user is authenticated, it holds the principal.

* + 1. AuthenticationProvider

Diagram

Description automatically generatedWhat is the actual thing which does the authentication? There are several ways to do it in a spring application.

The most common pattern is using **Providers**. The interface called AuthenticationProvider, the instance responsible for doing the actual authentication. It has a method called authenticate() and any application needs to have an implementation of this method and then tell spring security about it. Spring security will then call this authenticate() method to authenticate users.

The authenticate() method takes credentials as input, in an instance of Authentication and after it successfully authenticates, it clears out the credentials, puts the currently logged in user/principal in an instance of Authentication and returns it back.

Graphical user interface, text, application, Teams

Description automatically generated

* And the Authentication interface has getPrincipal(), getCredentials() and similar methods
* To tell if a given Authentication object holds credentials or the principle, we can call isAuthenticated() which returns a Boolean.

But a typical webapp can have multiple ways to authenticate. Credentials, Oauth, SSO, LDAP etc. As a result, an application can have multiple AuthenticationProvider s. Each one, knowing how to authenticate with its specific authentication mechanism. Each one knows what to expect as credentials and how to authenticate with those credentials.

* + 1. Diagram

       Description automatically generated AuthenticationManager

And how do those different AuthenticationProvider s coordinate with each other?

In order to coordinate all of them, there is a special type called AuthenticationManager. It too has an authenticate() method. There are different ways to implement this but we are looking at the ProviderManager, with the Provider pattern, which does this delegation to AuthenticationProvider s, depending on which ever provider supports the authentication. The ProviderManager doesn’t do the work itself. It just coordinates.

Given an attempted authentication, it asks all the available AuthenticationProvider s who supports the incoming auth type until it finds the right provider and delegates the work.

However, in order to do that, each AuthenticationProvider needs to say what kind of auth type it supports.

* AuthenticationProvider also have the supports() method.

This is the method which the AuthenticatinManager calls to find the right provider each time.

* + 1. Text, letter

       Description automatically generatedUserDetailsService

Each AuthenticationProvider needs to be able to access the identity store, to look up the user information based on the user that is sent to you. Once you get that information, you can then verify the credentials to see if they match, and return the principal.

This is pretty much all the AuthenticationProvider s must do. Whether you do LDAP or database store authentication, the only part that might change is how you retrieve this user from the system and where you retrieve it. But once the user is retrieved, the checks you make, like the verification of authorization is all the same.

* Text, letter

  Description automatically generatedWhich is why spring security extracted the part of retrieving the user information out into its own entity, called the UserDetailsService.

This service takes in a username and it returns an object UserDetails, which contains all the information about the user. Once this is return, the auth provider has everything it needs for doing the authentication. In fact, most of the AuthenticationProvider s use this same object as the UserDetails object directly as the principal object that gets put into the Authentication object.

Diagram

Description automatically generatedThis is the whole flow. If the auth was not successful then the AuthenticationProvider throws an AuthenticationException and this will bubble all the way up to the filter and either something catches it or the user will see the exception.

What happens when the filter gets the Principal after a successful auth?

It saves that object in the thread context (ThreadLocal). In every step of the way for a request, the app needs to know who the currently logged in user is. There is a security context associated with the current thread and this Authentication object is put there for you to use later on.

This pattern works the same irrespective of what the actual auth mechanism is.

What about subsequent requests? It can’t hold that in ThreadLocal, it has to be in the users session, isn’t it?

There is also a mechanism to allow for this context to be associated with the session. This is the reason why you authenticate only once and then access the app for the duration of that session.

This work is done by another spring security filter, which manages the users session. It takes the authenticated Principal and associates it with the users session. For every subsequent request this filter is responsible for taking that context from the session and saving it to ThreadLocal again so that it is available for the framework.

Asd

1. Custom Login Page

TODO

How to write a custom login page in Spring Security and Spring Boot - Java Brains: <https://www.youtube.com/watch?v=yoTohM2jYhs>

1. Registration with Spring Security

TODO – <https://www.baeldung.com/registration-with-spring-mvc-and-spring-security>

* 1. Password Encoding

Graphical user interface, text, application

Description automatically generated

TODO - <https://www.baeldung.com/spring-security-registration-password-encoding-bcrypt>