

**Core Concepts**

* **Authentication:** it is basically providing proof of who you are, usually via username and password. This kind of authentication is called knowledge-based authentication.

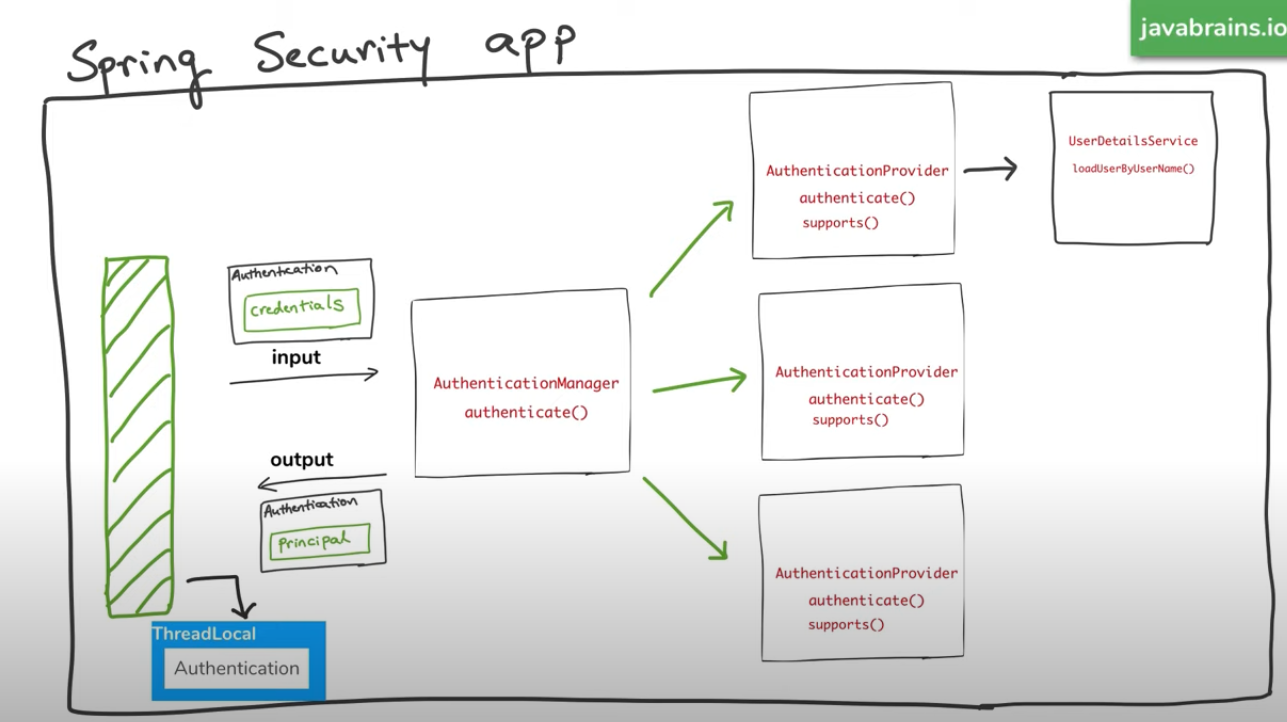
Other ways are Possession based authentication, for example phone text pin code entering.

Another way can be multi-factor authentication, which combines both of the above.

* **Authorization:** This decides what users can or can’t do.
* **Principle:** Principle is the currently logged in user in context of spring security. This is the reason we don’t need to input our username/pw for every request, cause spring remembers who we are once we login.
* **Granted Authority:** A user can only do something when a user has been granted authority to do so. For example, only admins can do certain stuff. So, admins are granted authority to do those stuff.
* **Role:** It is like a group of authorities that are assigned together. We can assign roles to users, and they get authority for those roles.

In most cases, Role and authority is interchangeable. Only difference is, authority is fine grained permissions, and coarse-grained grouping of permission are roles.

**Working Process**



* After user gives his credentials, authenticationmanager finds the right provider for the authentication using the supports method.
* After he find the right provider, it checks the authentication with authenticate method
* It looks up the user details using userdetailsservice
* This returns an instance of user details service
* AuthenticationProvider then verify this instance and then authentication happens.
* If successful, the authentication object is returned back with the principle and the authorities.
* If failed, authenticationprovider throws an exception, which is usually catch somewhere or user see the error in a. error page/exception thrown.
* After successful authentication, the authentication object is saved in the thread context. There is a security context in the threadlocal object (this object values can only be read/write by the same thread). This authentication object is placed in the security context to use for authorization or identifying the current principle.
* There is a mechanism for this security context to be associated with the session, this is done by another filter which takes in the user session and associates it with the user session.

**Overwriting the default configuration**

* By Default, when we add spring security, it will secure every route other than error by adding filters for all the incoming request. This filter actually delegates its work to bunch of filters, which does different things.
* It will create a default user with username = user, and a random password that we can see in the console.

Here **AuthenticationManager** manages authentication in spring security application. It has a method called authenticate (), that either returns a principle in case of success or throws an exception that it can’t authenticate.

Now to configure what authenticationmanager do, we need to use the class **AuthenticationManagerBuilder.**

After we configure the authentication manager builder, it will create an authenticationmanager for us with our configuration.

Now, to do this, we need to extend the class **WebSecurityConfigurerAdapter.** This has the configure method (with authenticationmanagerbuilder hook in parameter) with default options, but we can overwrite it and create our own configuration.

* First, we need to tell what kind of authentication do we need, then based on that we provide the inputs.
* Then, we need to use the annotation @**EnableWebSecurity,** this tells spring that it is a web security configuration. There are other ways of securing application like application/method level security.
* Passwords must be encoded, for this we can create a bean of type **PasswordEncoder,** here we can define any encoder we want.

**Configuring the Authorization**

We can use the HttpSecurity object to configure the paths and access restrictions for those paths. We can get a hook of this object on the **WebSecurityConfigurerAdapter** configure method, which we can overwrite in our security configure class that extends this configure adapter.

* Here using the http object, we use the authorizerequest method toinitiate the authorization.
* After this we can use antmatcher to match pattern of url.
* Then we can assign a single role with hasRole() or multiple role with hasAnyRole().
* After this we can specify which kind of login we want, for example formlogin.
* The convention is to start with configuration of most restrictive path to least restrictive path, for example from the admin only url to user etc.

**Setting up JDBC authentication**

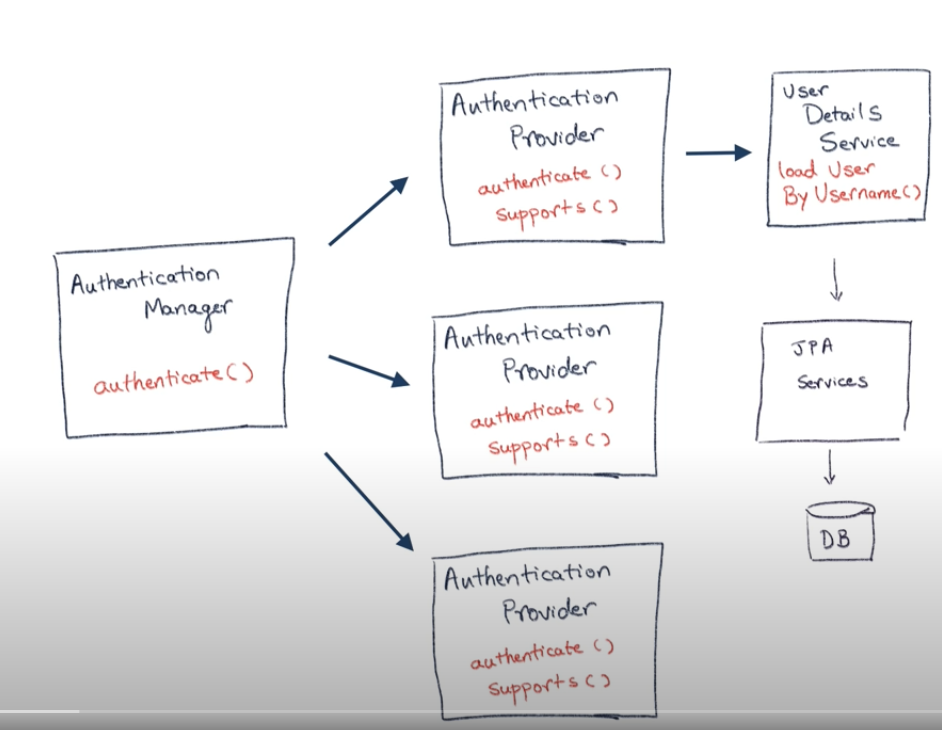
We can specify the data source in the properties file, and use the DataSource bean to inject as configuration for our authentication table.

For how to collect the user and authority from database, we can use **userByUsernameQuery** and **AuthoritiesByUsernameQuery** method chained to the auth object and specify the query to our needs.

We can also define a schema, but for now the other way seems better to me

**Setting up JPA authentication**

By default, JPA don’t have out of the box authentication. So, we have to implement the userDetailsService ourselves to configure the authentication. This service is actually not connected to the JPA at all, we can have it look up a text file if we want.



* We need to implement the **UserDetailsService** interface to configure our own implementation.
* There is a method inside that UserDetailsService interface called **loadUserByUsername**, which returns a **UserDetails** object.
* Whatever this method returns, spring will authenticate against it.