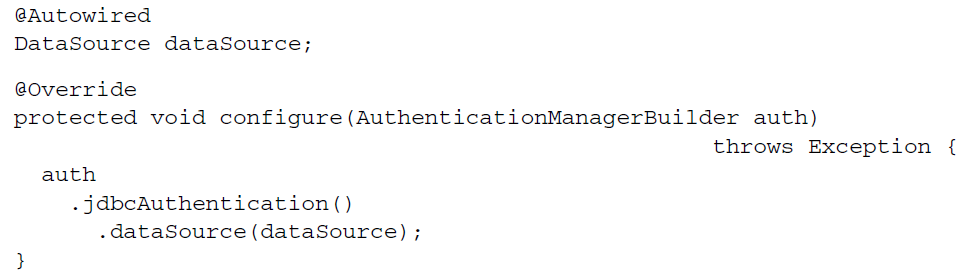
***Authenticating against database tables***

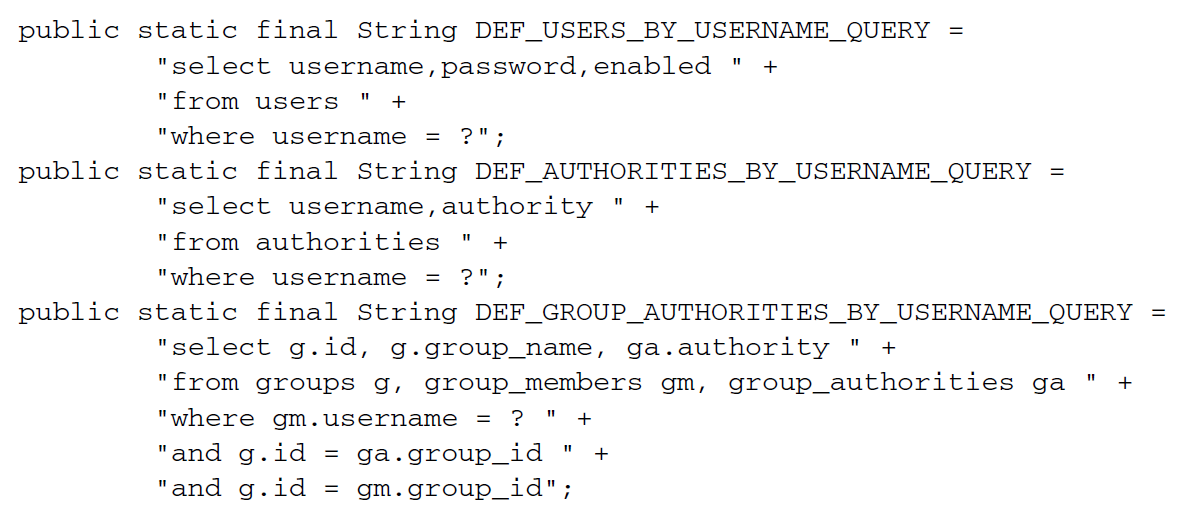
It’s quite common for user data to be stored in a relational database, accessed via JDBC. To configure Spring Security to authenticate against a JDBC-backed user store, you can use the jdbcAuthentication() method. The minimal configuration required is as follows:



The only thing you must configure is a DataSource so that it’s able to access the relational database. The DataSource is provided here via the magic of autowiring.

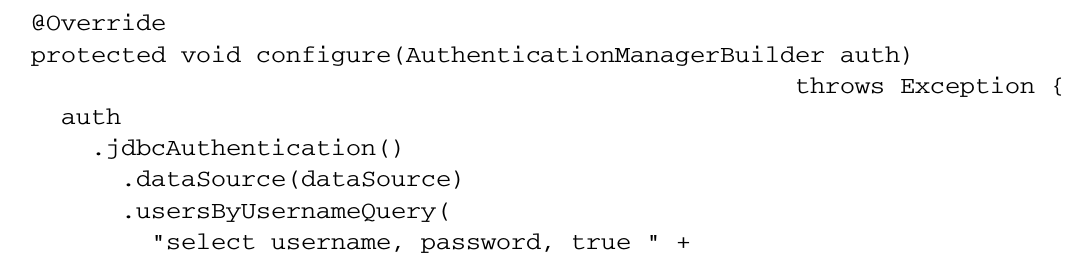
**OVERRIDING THE DEFAULT USER QUERIES**

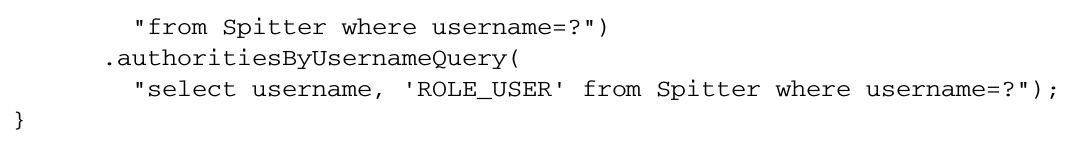
Although this minimal configuration will work, it makes some assumptions about your database schema. It expects that certain tables exist where user data will be kept. More specifically, the following snippet of code from Spring Security’s internals shows the SQL queries that will be performed when looking up user details:



The first query retrieves a user’s username, password, and whether or not they’re enabled. This information is used to authenticate the user. The next query looks up the user’s granted authorities for authorization purposes, and the final query looks up authorities granted to a user as a member of a group.

If you’re okay with defining and populating tables in your database that satisfy those queries, then there’s not much else for you to do. But chances are your database doesn’t look anything like this, and you’ll want more control over the queries. In that case, you can configure your own queries like this:





In this case, you’re only overriding the authentication and basic authorization queries. But you can also override the group authorities query by calling group-

AuthoritiesByUsername() with a custom query.

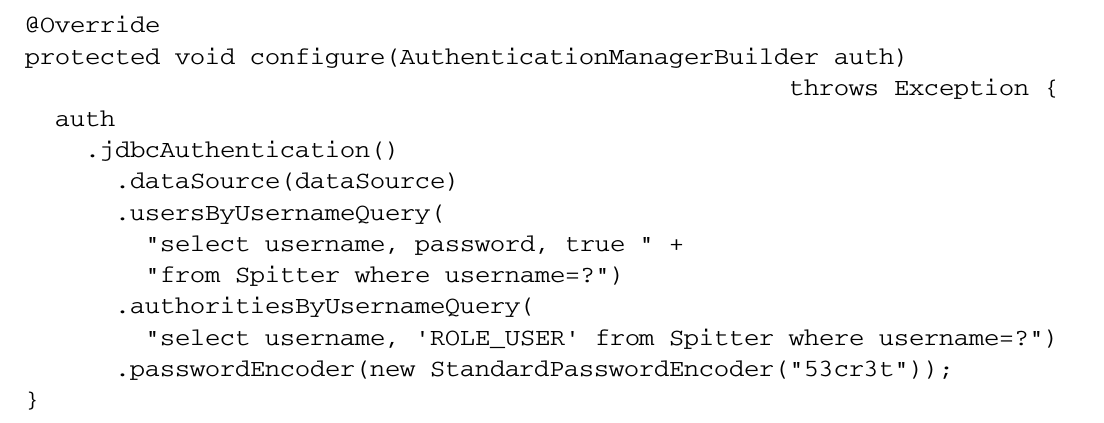
* When replacing the default SQL queries with those of your own design, it’s important to adhere to the basic contract of the queries. All of them take the username as their only parameter.
* The authentication query selects the username, password, and enabled status.
* The authorities query selects zero or more rows containing the username and a granted authority. And the group authorities query selects zero or more rows each with a group ID, group name, and an authority.

**WORKING WITH ENCODED PASSWORDS**

Focusing on the authentication query, you can see that user passwords are expected to be stored in the database. The only problem with that is that if the passwords are stored in plain text, they’re subject to the prying eyes of a hacker. But if you encode the password in the database, then authentication will fail because it won’t match the plain text password submitted by the user.

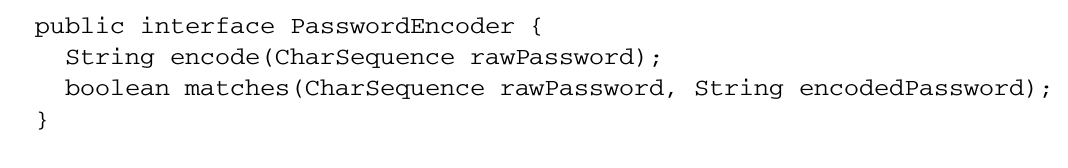
To remedy this problem, you need to specify a password encoder by calling the

passwordEncoder() method:



The passwordEncoder method accepts any implementation of Spring Security’s PasswordEncoder interface. Spring Security’s cryptography module includes three such implementations: BCryptPasswordEncoder , NoOpPasswordEncoder , and StandardPasswordEncoder.

The preceding code uses StandardPasswordEncoder . But you can always provide your own custom implementation if none of the out-of-the-box implementations meet your needs. The PasswordEncoder interface is rather simple:

No matter which password encoder you use, it’s important to understand that the password in the database is never decoded. Instead, the password that the user enters at login is encoded using the same algorithm and is then compared with the encoded password in the database. That comparison is performed in the PasswordEncoder’s matches() method.