**54.1. Spring Security JDBC**

**Database Access**:

Here we will learn how to use Spring Security with user accounts stored in database. This is an advanced feature of Spring security.

**Database Support in Spring Security**:

1. Spring Security can read user account info from database
2. By default, we have to follow Spring Security's predefined table schemas.

The advantage is, (the following schemas) Spring Security includes all of the JDBC code to actually read information from the database. We have to write very little Java code for reading the information from the database.

We have to simply setup the configuration, create the appropriate tables, and Spring Security will do all heavy work for developer in the background.

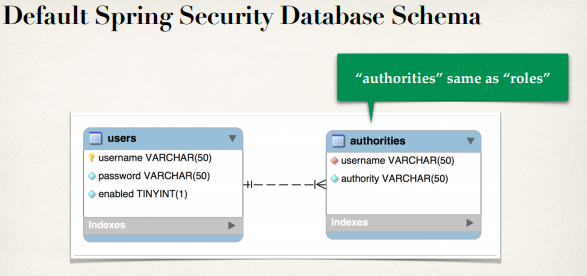
**Customize Database Access with Spring Security**:

1. Can also customize the table schemas.
2. Useful if we have custom tables specific to our project / customer.
3. We will be responsible for developing the code to access data.
   1. JDBC, Hibernate, etc.

**Development Process (Step-by-Step)**:

1. Develop SQL Script to set up database tables
2. Add database support to Maven POM file
3. Create JDBC properties file
4. Define DataSource in Spring Configuration
5. Update Spring Security Configuration to use JDBC

**Default Spring Security Database Schema**:



* We need to provide two tables. One called "**users**" and another called "**authorities**".
* Need exact table names and column names (**This is very important**).

**Step 1: Develop SQL Script to setup database tables**:

DROP DATABASE IF EXISTS `spring\_security\_demo\_plaintext`;

CREATE DATABASE IF NOT EXISTS `spring\_security\_demo\_plaintext`;

USE `spring\_security\_demo\_plaintext`;

DROP TABLE IF EXISTS `users`;

CREATE TABLE `users` (

`username` varchar(50) NOT NULL,

`password` varchar(50) NOT NULL,

`enabled` tinyint(1) NOT NULL,

PRIMARY KEY (`username`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--

-- Inserting data for table `users`

--

INSERT INTO `users`

VALUES

('john','{noop}test123',1),

('mary','{noop}test123',1),

('susan','{noop}test123',1);

--

-- Table structure for table `authorities`

--

DROP TABLE IF EXISTS `authorities`;

CREATE TABLE `authorities` (

`username` varchar(50) NOT NULL,

`authority` varchar(50) NOT NULL,

UNIQUE KEY `authorities\_idx\_1` (`username`,`authority`),

CONSTRAINT `authorities\_ibfk\_1` FOREIGN KEY (`username`) REFERENCES `users` (`username`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--

-- Inserting data for table `authorities`

--

INSERT INTO `authorities`

VALUES

('john','ROLE\_EMPLOYEE'),

('mary','ROLE\_EMPLOYEE'),

('mary','ROLE\_MANAGER'),

('susan','ROLE\_EMPLOYEE'),

('susan','ROLE\_ADMIN');

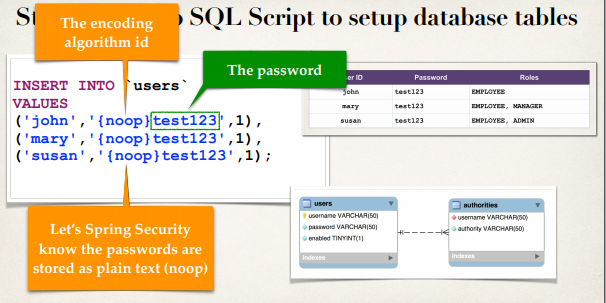
**Spring Security Password Storage**:

In Spring Security 5, passwords are stored in database using a specific format

**{id}encodedPassword**

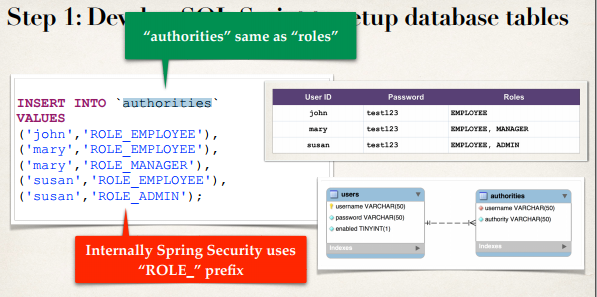
|  |  |
| --- | --- |
| ID | Description |
| noop (means no operation) | Plain text password |
| bcrypt | BCrypt password hashing |
| … | ... |

**User Table details**:



**Authorities Table details**:

We setting the " **authorities**" table for roles. The authorities table is same as roles. We insert the role in authorities table as "ROLE\_Role\_Name"



**Step 2: Add Database Support to Maven POM file**:

<!-- Add MySQL and C3p0 support -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.16</version>

</dependency>

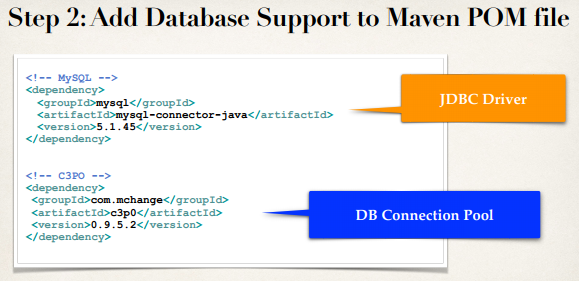
<dependency>

<groupId>com.mchange</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.5.4</version>

</dependency>



**Step 3: Create JDBC Properties File**:

**File: src/main/resources/persistence-mysql.properties**

#

# JDBC connection properties

#

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/spring\_security\_demo\_bcrypt?useSSL=false

jdbc.user=springstudent

jdbc.password=springstudent

#

# Connection pool properties

#

connection.pool.initialPoolSize=5

connection.pool.minPoolSize=5

connection.pool.maxPoolSize=20

connection.pool.maxIdleTime=3000

**Step 4: Define DataSource in Spring Configuration**:

1. Add "**@PropertySorce**" annotation in "**DemoAppConfig.class**".
2. Add "@ Autowired" annotation to "**Environment**" in "**DemoAppConfig.class**".
3. Define a @Bean in "**DemoAppConfig.class**".

**a) Add "@PropertySorce" annotation**:

Add "**@PropertySorce**" annotation to " DemoAppConfig.class". This will automatically read the **persistence-mysql.properties** file from "**src/main/resources**" and copied to classpath during Maven build.

**b) Add "@ Autowired" annotation to "Environment" in "DemoAppConfig.class"**:

In our "**DemoAppConfig.class**" class we have to add "**@Autowired**" in "**Environment**" class. This "**Environment**" is basically a Spring helper class that's can hold the data that was read from the property file. We already loaded from the property file. That data will be injected into this "**Environment**" item.

**import** org.springframework.context.annotation.PropertySource;

**import** org.springframework.core.env.Environment;

@PropertySource("classpath:persistence-mysql.properties")

**public** **class** DemoAppConfig {

// set up variable to hold the properties files

@Autowired

**private** Environment env;

**}**

**c) Define a @Bean in "DemoAppConfig.class"**:

Now we define a @Bean and inside the bean we will do the following thing

* Create connection pool
* Set the JDBC driver
* Set database connection props

// define a bean for our security datasource

@Bean

**public** DataSource securityDataSource() {

// create connection pool

ComboPooledDataSource securityDataSource = **new** ComboPooledDataSource();

// set the JDBC driver class

**try** {

securityDataSource.setDriverClass(env.getProperty("jdbc.driver"));

} **catch** (PropertyVetoException exe) {

**throw** **new** RuntimeException(exe);

}

// set database connection props

securityDataSource.setJdbcUrl(env.getProperty("jdbc.url"));

securityDataSource.setUser(env.getProperty("jdbc.user"));

securityDataSource.setPassword(env.getProperty("jdbc.password"));

// set connection pool props

securityDataSource.setInitialPoolSize(getIntProperty("connection.pool.initialPoolSize"));

securityDataSource.setMinPoolSize(getIntProperty("connection.pool.minPoolSize"));

securityDataSource.setMaxPoolSize(getIntProperty("connection.pool.maxPoolSize"));

securityDataSource.setMaxIdleTime(getIntProperty("connection.pool.maxIdleTime"));

**return** securityDataSource;

}

// need a helper method

// read environment property and convert to int

**private** **int** getIntProperty(String propName) {

String propVal = env.getProperty(propName);

// now convert them to int

**int** intPropVal = Integer.*parseInt*(propVal);

**return** intPropVal;

}

**Step 5: Update Spring Security to use JDBC**:

**public** **class** DemoSecurityConfig **extends** WebSecurityConfigurerAdapter {

// add a reference to our security data source

@Autowired

**private** DataSource securityDataSource;

@Override

**protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {

// use JDBC authentication

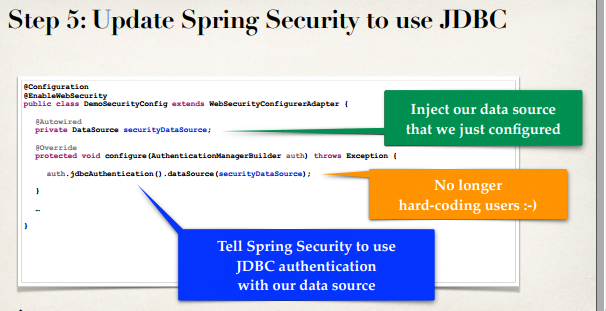
// tell Spring-Security to use JDBC authentication with our data source

// no longer hard-coding user, dataSource point to the database table (user, authentication)

auth.jdbcAuthentication().dataSource(securityDataSource);

}

}



54.1. Spring Security JDBC - Overview