Spring Boot REST API Security Overview

You will learn how to …

- Secure Spring Boot REST APIs
- Define users and roles
- Protect URLs based on role
- Store users, passwords and roles in DB (plain-text -> encrypted)

Practical Results

- Cover the most common Spring Security tasks that you will need on daily projects
- Not an A to Z reference … for that you can see **Spring Security Reference Manual**

http://www.luv2code.com/spring-security-reference-manual

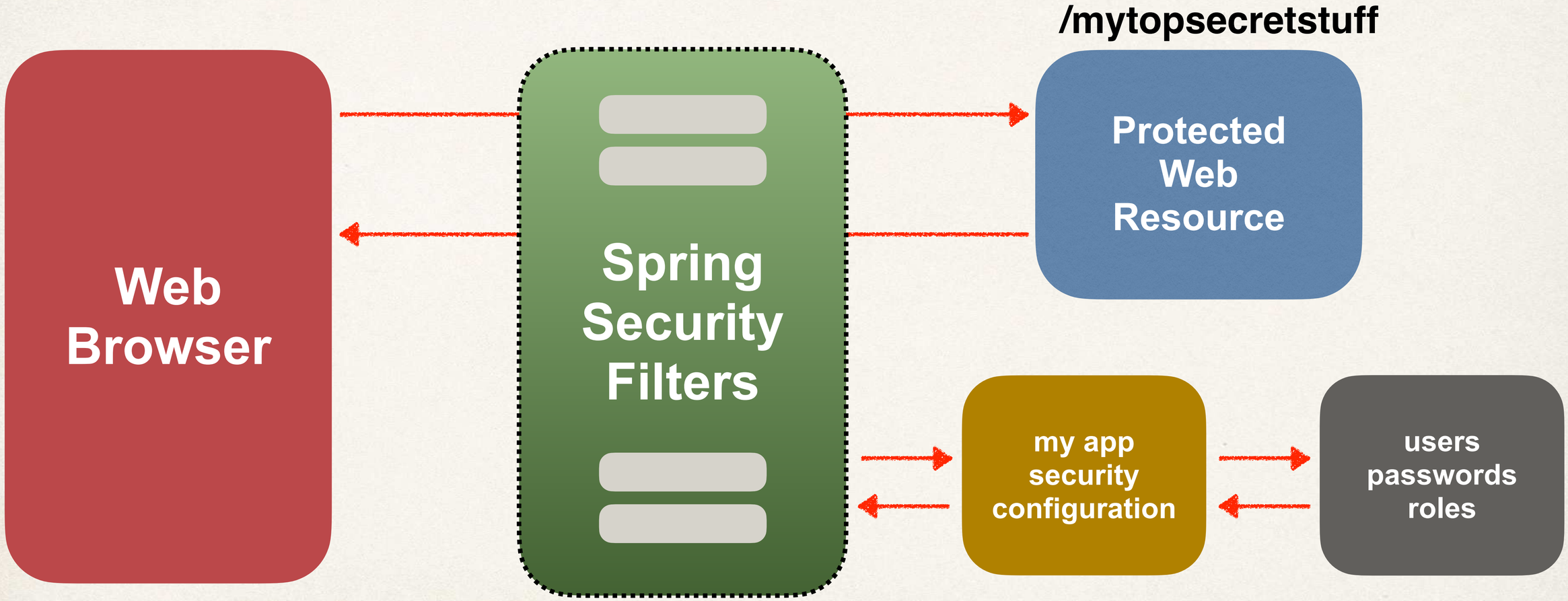
Spring Security Model

- Spring Security defines a framework for security
- Implemented using Servlet filters in the background
- Two methods of securing an app: declarative and programmatic

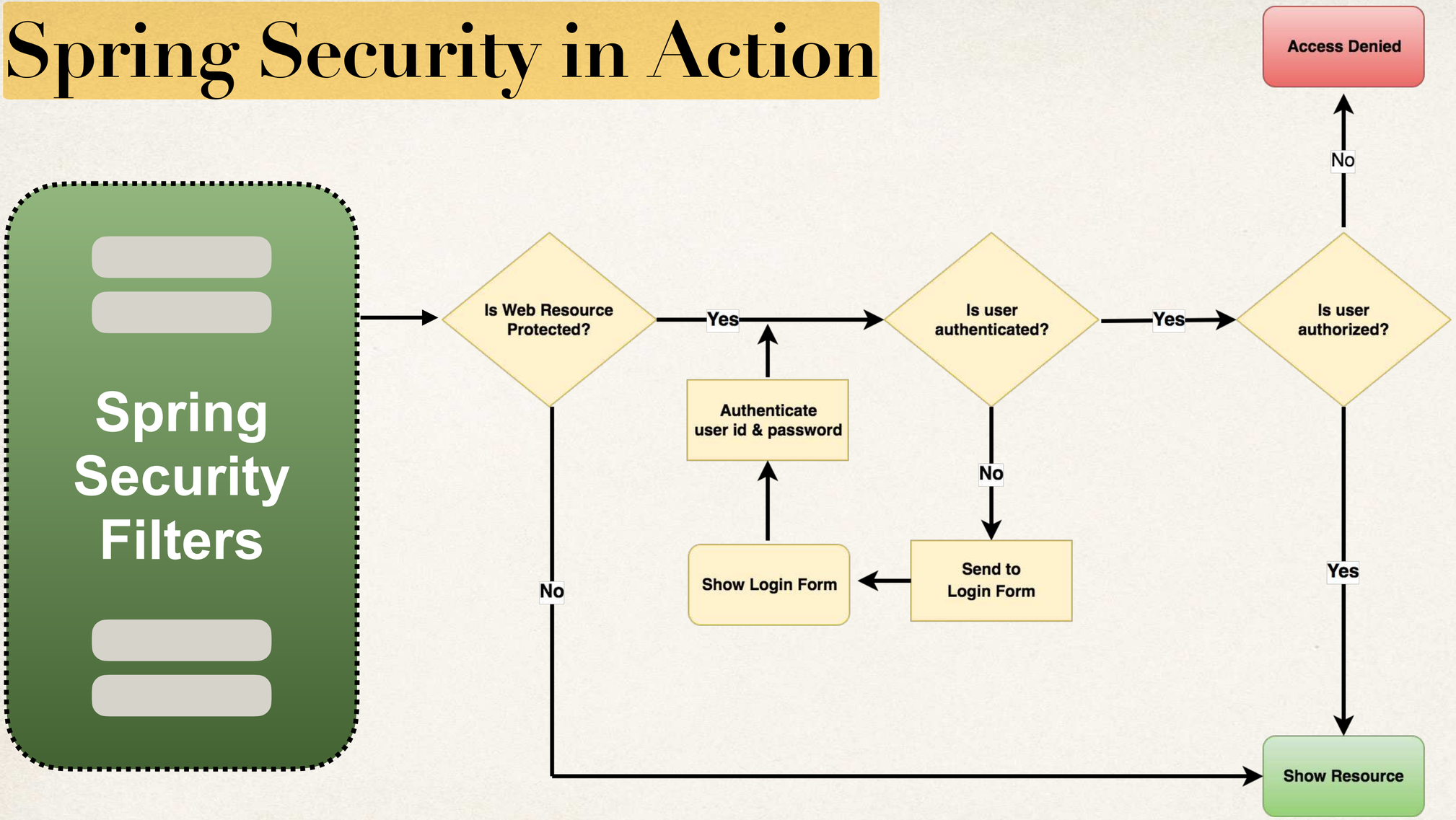
Spring Security with Servlet Filters

- Servlet Filters are used to pre-process / post-process web requests
- Servlet Filters can route web requests based on security logic
- Spring provides a bulk of security functionality with servlet filters

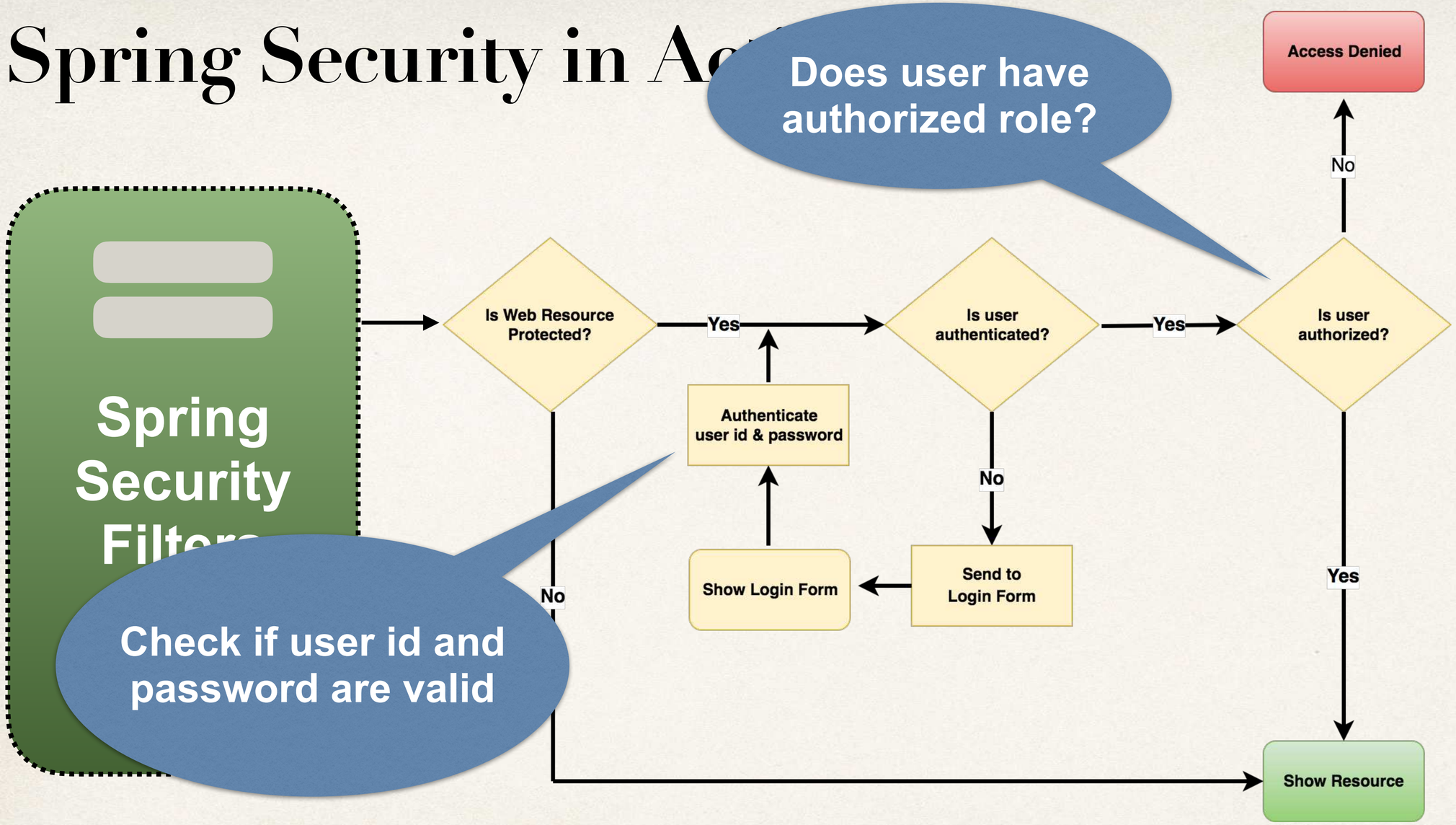
Spring Security Overview



Spring Security in Action



Spring Security in Ac



Security Concepts

- Authentication
  - Check user id and password with credentials stored in app / db
- Authorization
  - Check to see if user has an authorized role

Declarative Security

- Define application's security constraints in configuration
  - All Java config: @Configuration

- Provides separation of concerns between application code and security

Programmatic Security

- Spring Security provides an API for custom application coding
- Provides greater customization for specific app requirements

Enabling Spring Security

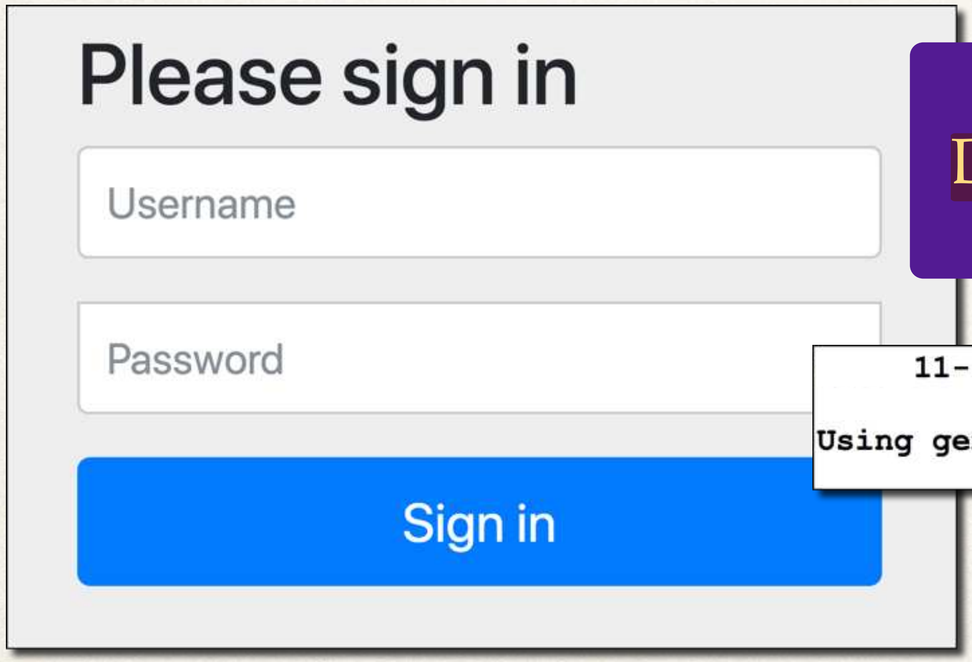
1. Edit **pom.xml** and add **spring-boot-starter-security**

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

2. This will *automagically* secure all endpoints for application

Secured Endpoints

- Now when you access your application
- Spring Security will prompt for login



Default user name: **user**

Check console logs for password

```
2022-11-02 21:05:57.074  INFO 24986 --- [           main] .s.s.UserDetailsSe

Using generated security password: 78fd68a6-c190-421d-934b-df7852fc7dc2
```

Spring Security configuration

- You can override default user name and generated password

File: src/main/resources/application.properties

```
spring.security.user.name=scott
spring.security.user.password=test123
```

Authentication and Authorization

- In-memory
- JDBC
- LDAP
- Custom / Pluggable
- *others …*

We will cover password storage in DB as plain-text AND encrypted

Configuring Basic Security

Our Users

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

We can give ANY names for user roles

Development Process

Step-By-Step

1. Create Spring Security Configuration (@Configuration)

2. Add users, passwords and roles

Step 1: Create Spring Security Configuration

File: DemoSecurityConfig.java

```
import org.springframework.context.annotation.Configuration;

@Configuration
public class DemoSecurityConfig {

  // add our security configurations here …

}
```

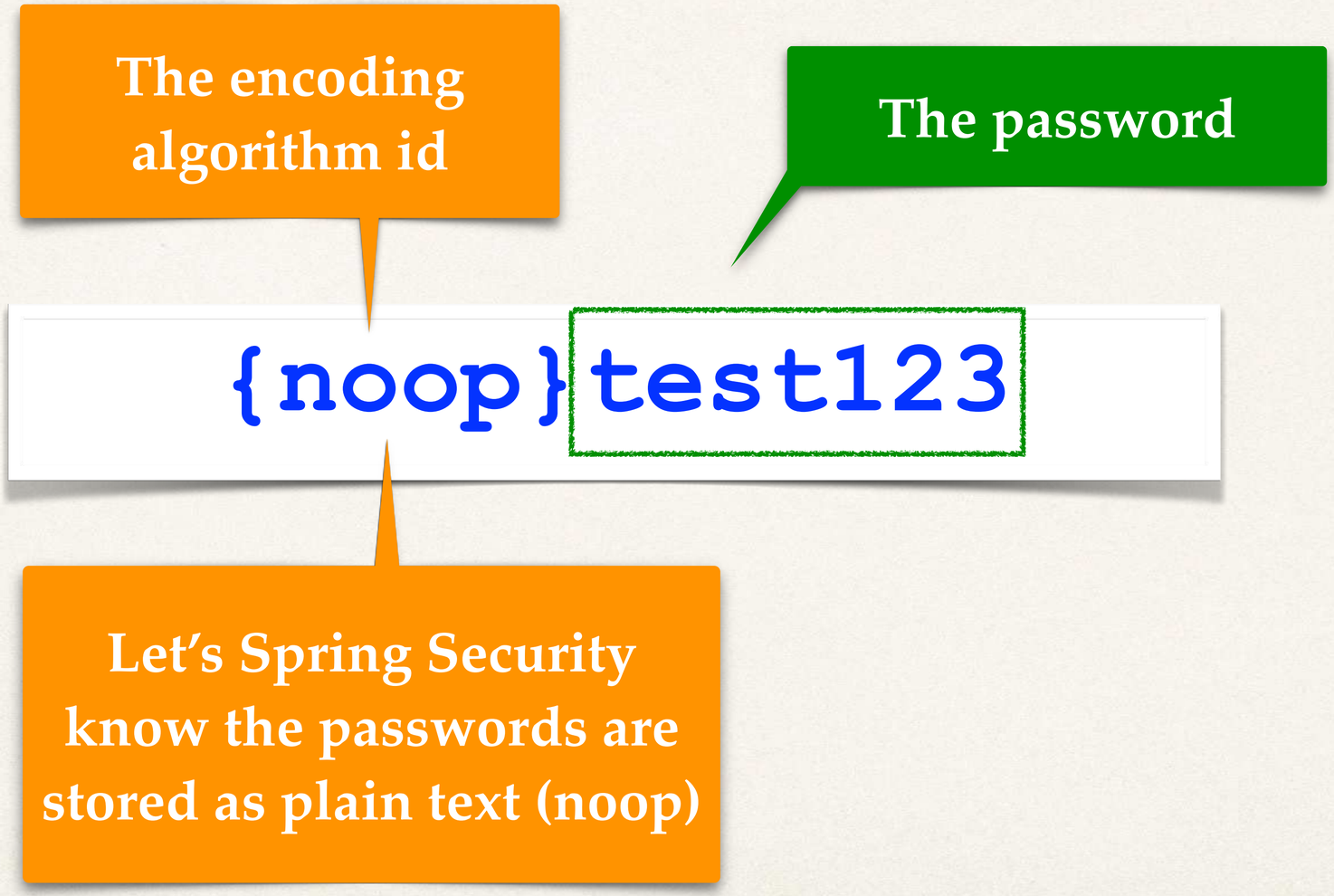
Spring Security Password Storage

- In Spring Security, passwords are stored using a specific format

```
{id}encodedPassword
```

ID	Description
noop	Plain text passwords
bcrypt	BCrypt password hashing
...	...

Password Example



Step 2: Add users, passwords and roles

File: DemoSecurityConfig.java

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public InMemoryUserDetailsManager userDetailsManager() {

        UserDetails john = User.builder()
                .username("john")
                .password("{noop}test123")
                .roles("EMPLOYEE")
                .build();

        UserDetails mary = User.builder()
                .username("mary")
                .password("{noop}test123")
                .roles("EMPLOYEE", "MANAGER")
                .build();

        UserDetails susan = User.builder()
                .username("susan")
                .password("{noop}test123")
                .roles("EMPLOYEE", "MANAGER", "ADMIN")
                .build();

        return new InMemoryUserDetailsManager(john, mary, susan);
    }
}
```

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

We will add DB support in later videos

(plaintext and encrypted)

Restrict Access Based on Roles

Our Example

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

Restricting Access to Roles

- General Syntax

Restrict access to a given path "/api/employees"

```
requestMatchers(<< add path to match on >>)
        .hasRole(<< authorized role >>)
```

Single role

"ADMIN"

Restricting Access to Roles

Specify HTTP method: GET, POST, PUT, DELETE …

Restrict access to a given path "/api/employees"

```
requestMatchers(<< add HTTP METHOD to match on >>, << add path to match on >>)
        .hasRole(<< authorized roles >>)
```

Single role

Restricting Access to Roles

```
requestMatchers(<< add HTTP METHOD to match on >>, << add path to match on >>)
    .hasAnyRole(<< list of authorized roles >>)
```

Any role

Comma-delimited list

Authorize Requests for EMPLOYEE role

```
requestMatchers(HttpMethod.GET, "/api/employees").hasRole("EMPLOYEE")
requestMatchers(HttpMethod.GET, "/api/employees/**").hasRole("EMPLOYEE")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

The ** syntax:
match on all sub-paths

Authorize Requests for MANAGER role

```
requestMatchers(HttpMethod.POST, "/api/employees").hasRole("MANAGER")
requestMatchers(HttpMethod.PUT, "/api/employees").hasRole("MANAGER")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

Authorize Requests for ADMIN role

```
requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN")
```

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

Pull It Together

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(configurer ->
                configurer
                        .requestMatchers(HttpMethod.GET, "/api/employees").hasRole("EMPLOYEE")
                        .requestMatchers(HttpMethod.GET, "/api/employees/**").hasRole("EMPLOYEE")
                        .requestMatchers(HttpMethod.POST, "/api/employees").hasRole("MANAGER")
                        .requestMatchers(HttpMethod.PUT, "/api/employees").hasRole("MANAGER")
                        .requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN"));

    // use HTTP Basic authentication
    http.httpBasic();

    return http.build();
}
```

Use HTTP Basic Authentication

HTTP Method	Endpoint	CRUD Action	Role
GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeId}	Read single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	Update	MANAGER
DELETE	/api/employees/{employeeId}	Delete employee	ADMIN

Cross-Site Request Forgery (CSRF)

- Spring Security can protect against CSRF attacks
- Embed additional authentication data/token into all HTML forms
- On subsequent requests, web app will verify token before processing
- Primary use case is traditional web applications (HTML forms etc …)

When to use CSRF Protection?

- The Spring Security team recommends
  - Use CSRF protection for any normal browser web requests
  - Traditional web apps with HTML forms to add/modify data
- If you are building a REST API for non-browser clients
  - you *may* want to disable CSRF protection
- In general, not required for stateless REST APIs
  - That use POST, PUT, DELETE and/or PATCH

Pull It Together

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(configurer ->
                configurer
                        .requestMatchers(HttpMethod.GET, "/api/employees").hasRole("EMPLOYEE")
                        .requestMatchers(HttpMethod.GET, "/api/employees/**").hasRole("EMPLOYEE")
                        .requestMatchers(HttpMethod.POST, "/api/employees").hasRole("MANAGER")
                        .requestMatchers(HttpMethod.PUT, "/api/employees").hasRole("MANAGER")
                        .requestMatchers(HttpMethod.DELETE, "/api/employees/**").hasRole("ADMIN"));

    // use HTTP Basic authentication
    http.httpBasic();

   // disable Cross Site Request Forgery (CSRF)
   http.csrf().disable();


    return http.build();
}
```

In general, CSRF is not required for stateless REST APIs that use POST, PUT, DELETE and/or PATCH

Spring Security
User Accounts Stored in Database

Database Access

- So far, our user accounts were hard coded in Java source code

- We want to add database access

Advanced

Recall Our User Roles

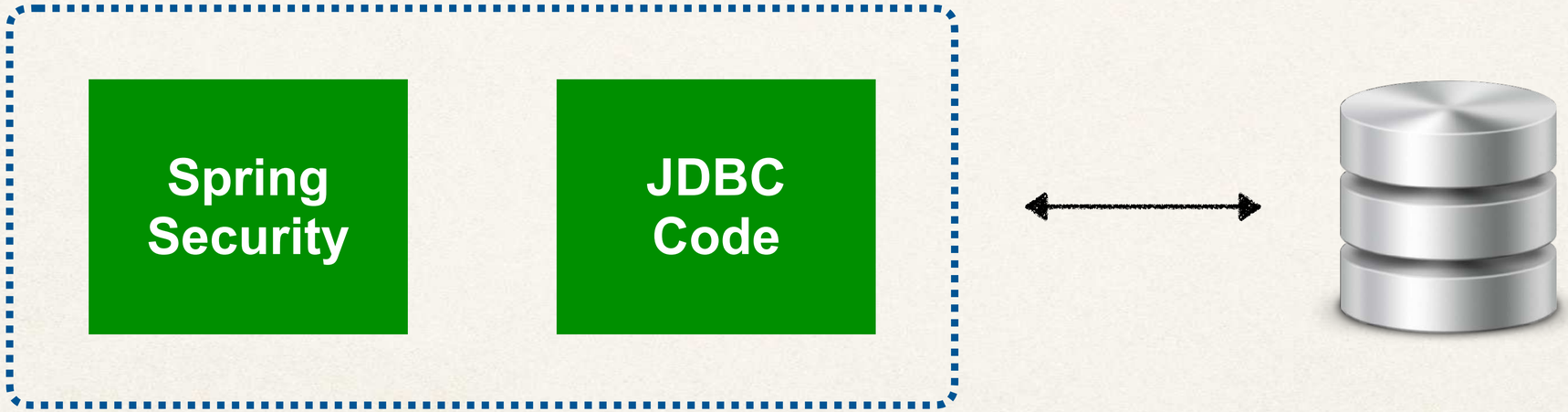
User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN

Database Support in Spring Security

Out-of-the-box

- Spring Security can read user account info from database

- By default, you have to follow Spring Security's predefined table schemas



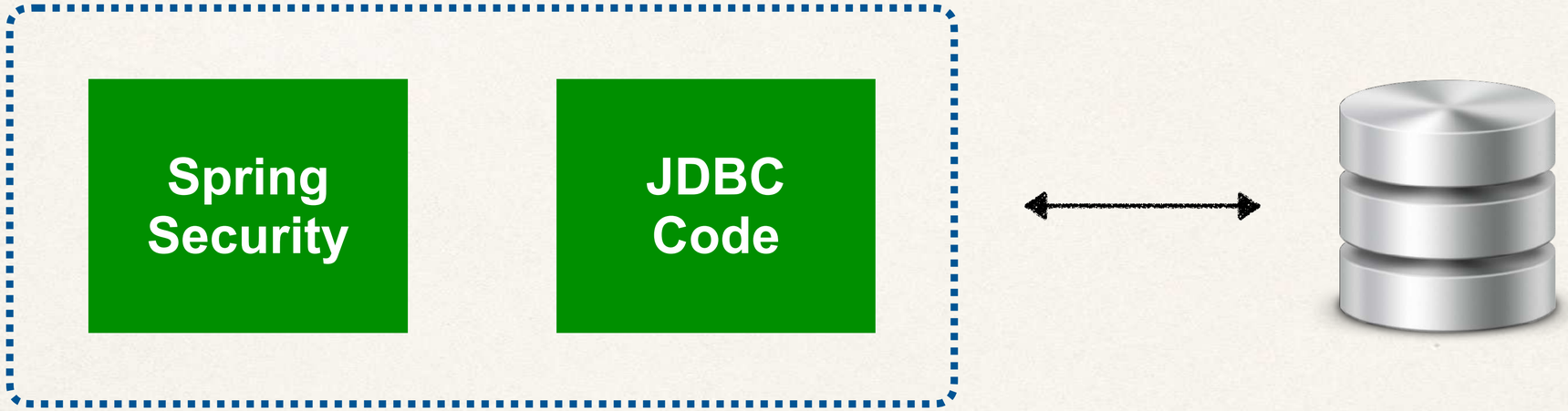
Customize Database Access with Spring Security

- Can also customize the table schemas
- Useful if you have custom tables specific to your project / custom
- You will be responsible for developing the code to access the data
  - JDBC, JPA/Hibernate etc …

Database Support in Spring Security

Out-of-the-box

- Follow Spring Security's predefined table schemas



Development Process

Step-By-Step

1. Develop SQL Script to set up database tables

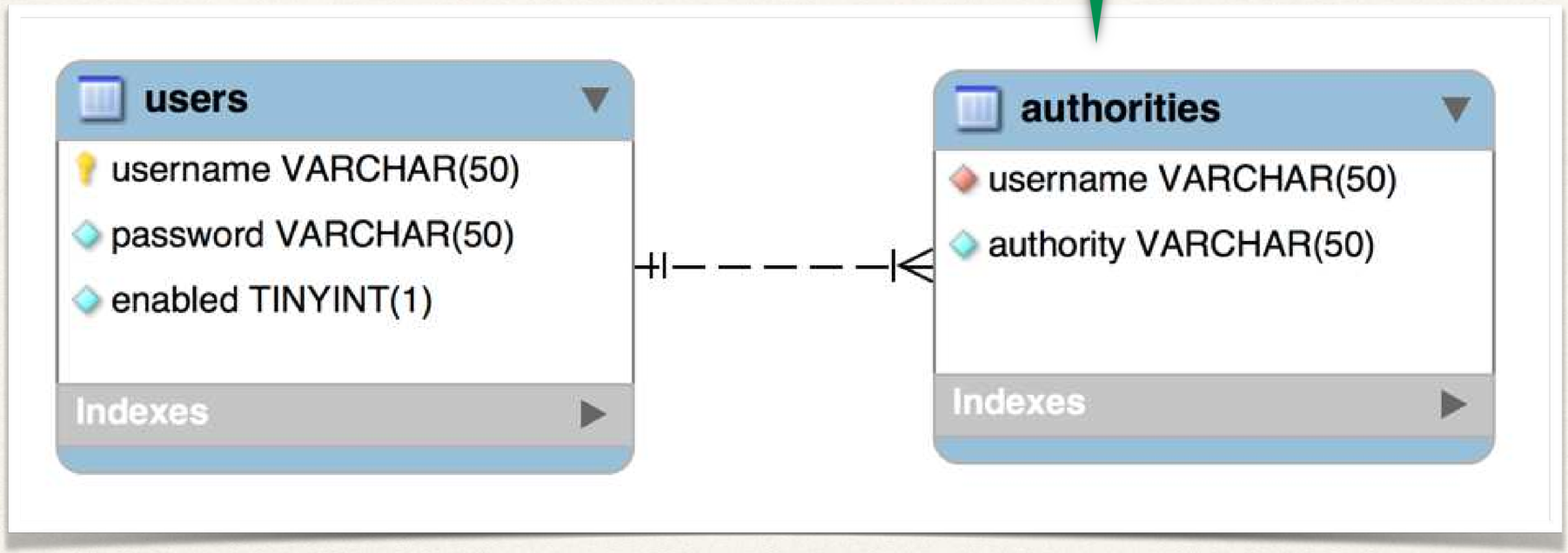
2. Add database support to Maven POM file

3. Create JDBC properties file

4. Update Spring Security Configuration to use JDBC

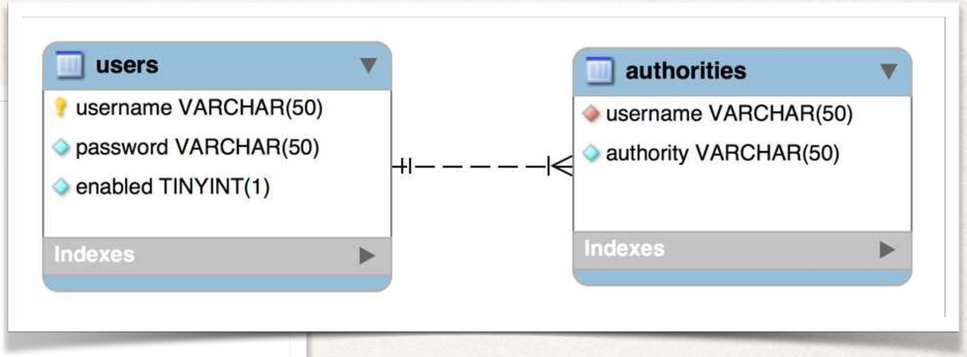
Default Spring Security Database Schema

"authorities" same as "roles"



Step 1: Develop SQL Script to setup database tables

```
CREATE TABLE `users` (
  `username` varchar(50) NOT NULL,
  `password` varchar(50) NOT NULL,
  `enabled` tinyint NOT NULL,

  PRIMARY KEY (`username`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```



Step 1: Develop SQL Script to setup database tables

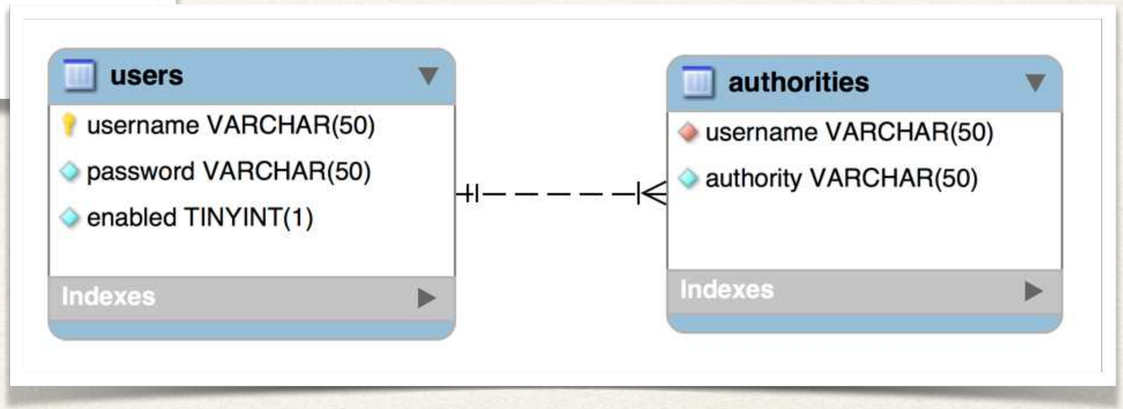
The encoding algorithm id

The password

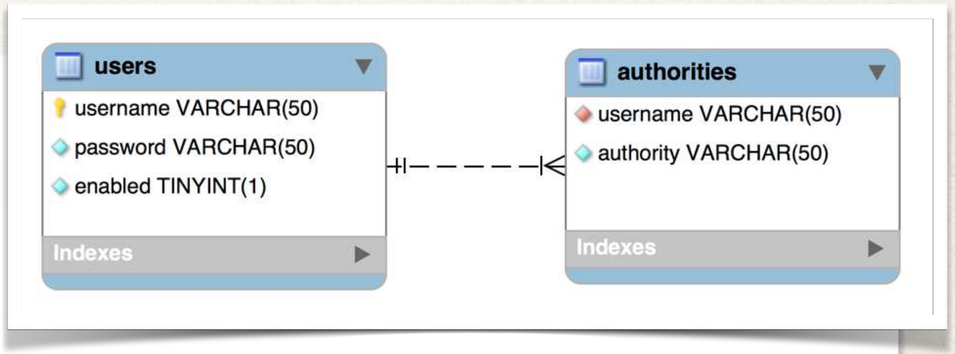
```
INSERT INTO `users`
VALUES
('john','{noop}test123',1),
('mary','{noop}test123',1),
('susan','{noop}test123',1);
```

Let's Spring Security know the passwords are stored as plain text (noop)

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN



Step 1: Develop SQL Script to setup database tables



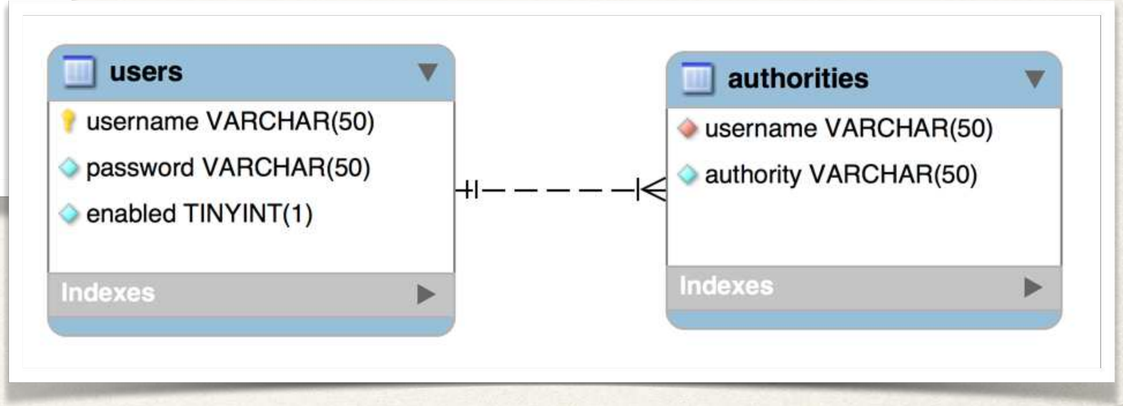

```
CREATE TABLE `authorities` (
  `username` varchar(50) NOT NULL,
  `authority` varchar(50) NOT NULL,

  UNIQUE KEY `authorities_idx_1` (`username`,`authority`),

  CONSTRAINT `authorities_ibfk_1`
  FOREIGN KEY (`username`)
  REFERENCES `users` (`username`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Step 1: Develop SQL Script to setup database tables

"authorities" same as "roles"

```
INSERT INTO `authorities`
VALUES
('john','ROLE_EMPLOYEE'),
('mary','ROLE_EMPLOYEE'),
('mary','ROLE_MANAGER'),
('susan','ROLE_EMPLOYEE'),
('susan','ROLE_MANAGER'),
('susan','ROLE_ADMIN');
```

User ID	Password	Roles
john	test123	EMPLOYEE
mary	test123	EMPLOYEE, MANAGER
susan	test123	EMPLOYEE, MANAGER, ADMIN



Internally Spring Security uses "ROLE_" prefix

Step 2: Add Database Support to Maven POM file

```
<!-- MySQL -->
<dependency>
   <groupId>com.mysql</groupId>
   <artifactId>mysql-connector-j</artifactId>
   <scope>runtime</scope>
</dependency>
```

JDBC Driver

Step 3: Create JDBC Properties File

File: application.properties

```
#
# JDBC connection properties
#
spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory
spring.datasource.username=springstudent
spring.datasource.password=springstudent
```

Step 4: Update Spring Security to use JDBC

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {

        return new JdbcUserDetailsManager(dataSource);
    }

    …
}
```

Inject data source
Auto-configured by Spring Boot

No longer
hard-coding users :-)

Tell Spring Security to use
JDBC authentication
with our data source

Spring Security
Password Encryption

© luv2code LLC

Password Storage

- So far, our user passwords are stored in plaintext … yikes!

username	password	enabled
john	{noop}test123	1
mary	{noop}test123	1
susan	{noop}test123	1

- Ok for getting started … but not for production / real-time project :-(

© luv2code LLC

Password Storage - Best Practice

Best Practice

- The best practice is store passwords in an encrypted format

username	password	enabled
john	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1
mary	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1
susan	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1

Encrypted version of password

© luv2code LLC

Spring Security Team Recommendation

- Spring Security recommends using the popular **bcrypt** algorithm

- bcrypt
  - Performs one-way encrypted hashing
  - Adds a random salt to the password for additional protection
  - Includes support to defeat brute force attacks

© luv2code LLC

Bcrypt Additional Information

- Why you should use bcrypt to hash passwords

  www.luv2code.com/why-bcrypt

- Detailed bcrypt algorithm analysis

  www.luv2code.com/bcrypt-wiki-page

- Password hashing - Best Practices

  www.luv2code.com/password-hashing-best-practices

© luv2code LLC

How to Get a Bcrypt password

You have a plaintext password and you want to encrypt using bcrypt

- Option 1: Use a website utility to perform the encryption
- Option 2: Write Java code to perform the encryption

© luv2code LLC

How to Get a Bcrypt password - Website

- Visit: **`www.luv2code.com/generate-bcrypt-password`**
- Enter your plaintext password
- The website will generate a bcrypt password for you

© luv2code LLC

DEMO

© luv2code LLC

Development Process

Step-By-Step

1. Run SQL Script that contains encrypted passwords

- Modify DDL for password field, length should be 68

THAT'S IT … no need to change Java source code :-)

© luv2code LLC

Spring Security Password Storage

- In Spring Security, passwords are stored using a specific format

```
{bcrypt}encodedPassword
```

Password column must be at least 68 chars wide

{bcrypt} - 8 chars
***encodedPassword* - 60 chars**

username	password	enabled
john	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1
mary	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1
susan	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1

© luv2code LLC

Modify DDL for Password Field

```
CREATE TABLE `users` (
  `username` varchar(50) NOT NULL,
  `password`     char(68) NOT NULL,
  `enabled` tinyint NOT NULL,

  PRIMARY KEY (`username`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Password column must be at least 68 chars wide

{bcrypt} - 8 chars
***encodedPassword* - 60 chars**

© luv2code LLC

Step 1: Develop SQL Script to setup database tables

The encoding algorithm id

The encrypted password: fun123

```
INSERT INTO `users`
VALUES
('john','{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q',1),

...
```

Let's Spring Security know the passwords are stored as encrypted passwords: bcrypt

© luv2code LLC

Step 1: Develop SQL Script to setup database tables

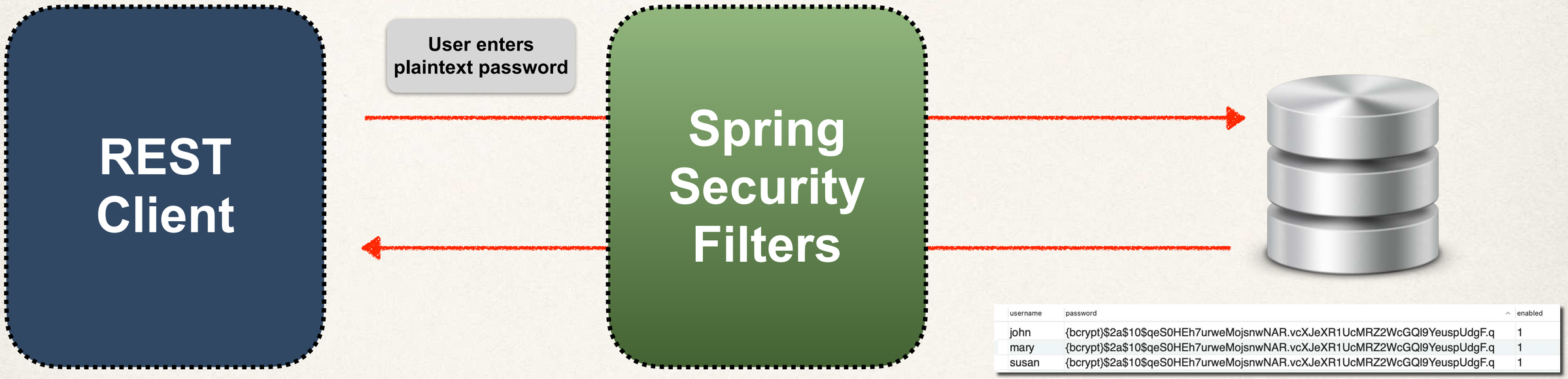
The encoding algorithm id

The encrypted password: fun123

```
INSERT INTO `users`
VALUES
('john','{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q',1),

('mary','{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2IwjyWefYkMpiBEFlpBwDH.5PM0K',1),

('susan','{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2IwjyWefYkMpiBEFlpBwDH.5PM0K',1);
```


© luv2code LLC

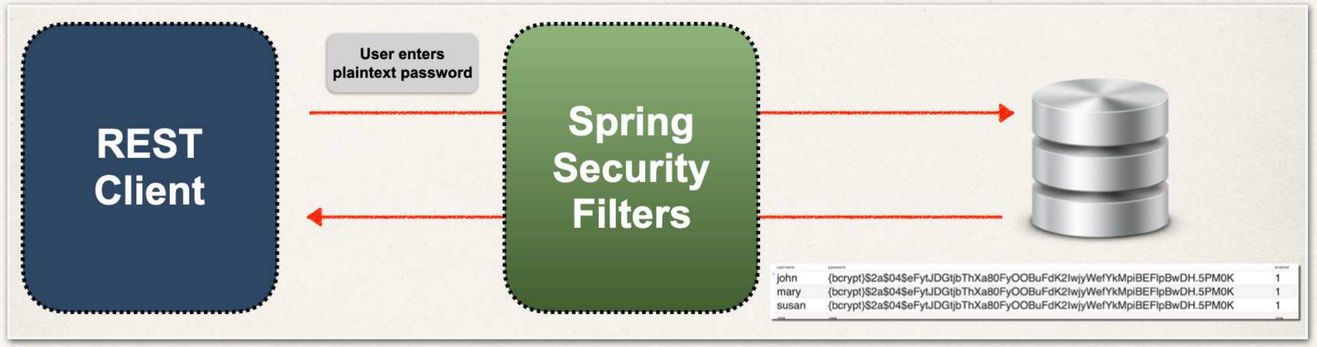
Spring Security Login Process



username	password	enabled
john	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1
mary	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1
susan	{bcrypt}$2a$10$qeS0HEh7urweMojsnwNAR.vcXJeXR1UcMRZ2WcGQl9YeuspUdgF.q	1

© luv2code LLC

Spring Security Login Process



1. Retrieve password from db for the user
2. Read the encoding algorithm id (bcrypt etc)
3. For case of bcrypt, encrypt plaintext password from login form (using salt from db password)
4. Compare encrypted password from login form WITH encrypted password from db
5. If there is a match, login successful
6. If no match, login NOT successful

Note:
The password from db is NEVER decrypted

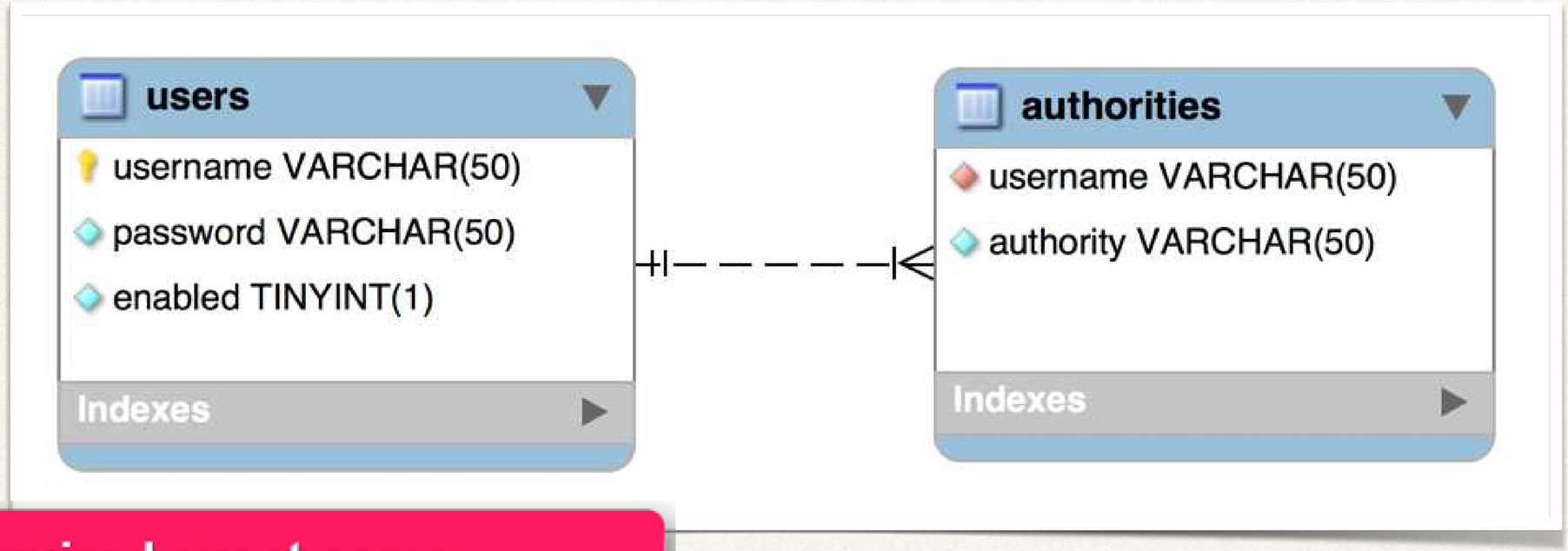
Because bcrypt is a one-way encryption algorithm

© luv2code LLC

Spring Security
Custom Tables

© luv2code LLC

Default Spring Security Database Schema

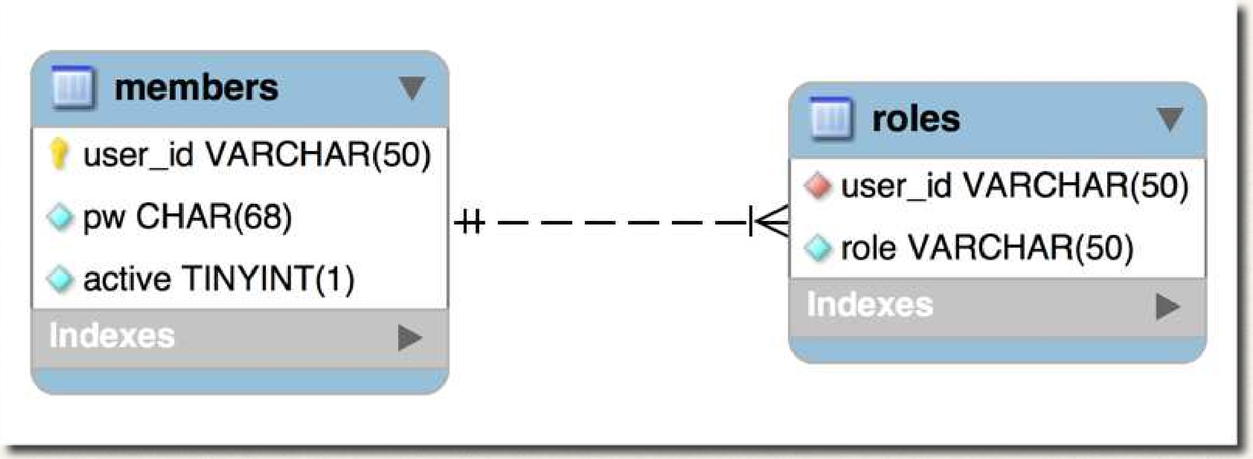


Required exact same table names and column names

© luv2code LLC

Custom Tables

- What if we have our own custom tables?
- Our own custom column names?



This is all custom
Nothing matches with default Spring Security table schema

© luv2code LLC

For Security Schema Customization

- Tell Spring how to query your custom tables
- Provide query to find user by user name
- Provide query to find authorities / roles by user name

© luv2code LLC

Development Process

Step-By-Step

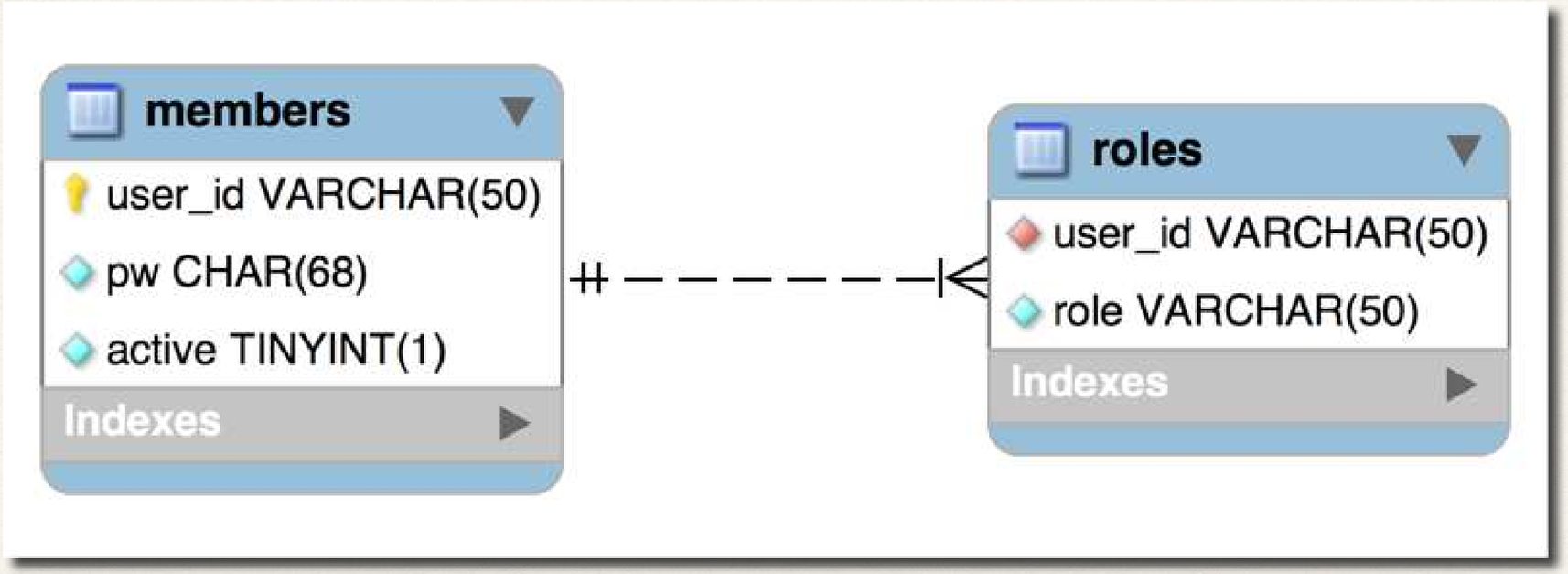
1. Create our custom tables with SQL

2. Update Spring Security Configuration

   - Provide query to find user by user name
   - Provide query to find authorities / roles by user name

© luv2code LLC

Step 1: Create our custom tables with SQL

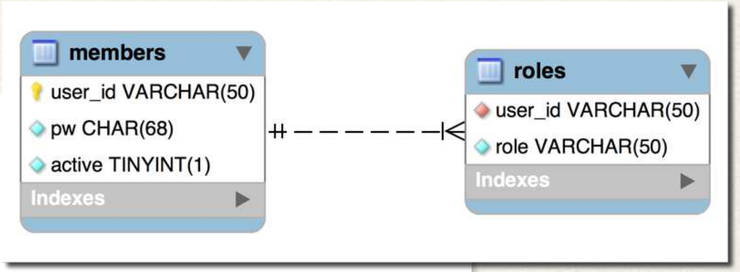


This is all custom

Nothing matches with default Spring Security table schema

© luv2code LLC

Step 2: Update Spring Security Configuration

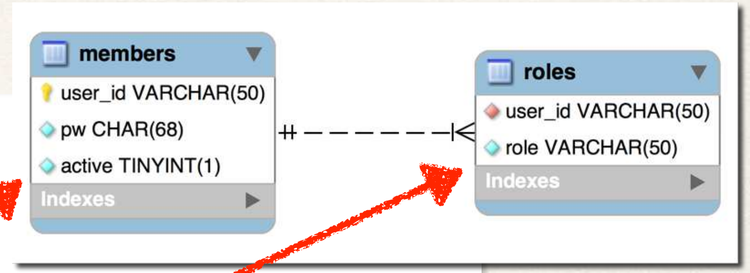



```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {
        JdbcUserDetailsManager theUserDetailsManager = new JdbcUserDetailsManager(dataSource);

        theUserDetailsManager
           .setUsersByUsernameQuery("select user_id, pw, active from members where user_id=?");

        theUserDetailsManager
           .setAuthoritiesByUsernameQuery("select user_id, role from roles where user_id=?");

        return theUserDetailsManager;
    }

    ...

}
```


© luv2code LLC

Step 2: Update Spring Security Configuration

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {
        JdbcUserDetailsManager theUserDetailsManager = new JdbcUserDetailsManager(dataSource);

        theUserDetailsManager
           .setUsersByUsernameQuery("select user_id, pw, active from members where user_id=?");

        theUserDetailsManager
           .setAuthoritiesByUsernameQuery("select user_id, role from roles where user_id=?");

        return theUserDetailsManager;
    }

    ...

}
```



How to find users

How to find roles

Question mark "?"

Parameter value will be the user name from login

© luv2code LLC