The provided information demonstrates how to configure Spring Boot Security with JDBC authentication. Here's a breakdown of the steps involved:

Step 1: Setup Database tables with required data

- This step involves creating two tables: `users` and `authorities`.
- The `users` table stores user information such as username, password, and enabled status.
- The `authorities` table stores the username and corresponding authority (role) of each user.

Step 2: Create Boot application with required dependencies

- Add the necessary dependencies to your Spring Boot application, including:
- `web-starter`: for building web applications
- `security-starter`: for Spring Security support
- `data-jdbc`: for JDBC database connectivity
- `mysql-connector`: for MySQL database support
- `lombok`: for reducing boilerplate code
- `devtools`: for development-time tools and automatic restarts

Step 3: Configure Data source properties

- In the `application.properties` file, specify the MySQL database connection properties, including the URL, username, password, and driver class name.

Step 4: Create a Rest Controller with required methods

- Create a `UserRestController` class that defines the REST endpoints for different user roles, such as `/admin`, `/user`, and `/`.

Step 5: Create a Security Configuration class with JDBC Authentication Manager

- Create a `SecurityConfiguration` class and annotate it with `@Configuration` and `@EnableWebSecurity`.

- Autowire the `DataSource` bean and the `AuthenticationManagerBuilder`.
- In the `authManager` method, configure JDBC authentication by specifying the data source, password encoder, and queries to retrieve user information and authorities from the database.
- Define a `SecurityFilterChain` bean that configures the authorization rules using the `HttpSecurity` object.
- In the `securityConfig` method, use the `HttpSecurity` object to define the authorization rules based on URL patterns and roles.
- In this example, the `/admin` endpoint requires the `ROLE_ADMIN` role, the `/user` endpoint requires either `ROLE_ADMIN` or `ROLE_USER` roles, and the `/` endpoint is accessible to all.
- The `formLogin()` method is used to enable form-based authentication.

Note: Make sure to include the necessary imports and dependencies in your code.

Let's go through the provided code and explain each section:

```
@RestController

public class UserRestController {

@GetMapping(value = "/admin")

public String admin() {
return "<h3>Welcome Admin :)</h3>";

}
@GetMapping(value = "/user")

public String user() {
return "<h3>Hello User :)</h3>";

}

@GetMapping(value = "/")

public String welcome() {
```

```
return  "<h3>Welcome :)</h3>";

}
}
```

This code defines a `UserRestController` class annotated with `@RestController`, indicating that it's responsible for handling REST API requests. It contains three request mapping methods:

1. `admin()`: This method handles GET requests for the `/admin` endpoint. It returns a simple HTML string `<h3>Welcome Admin :)</h3>`.
2. `user()`: This method handles GET requests for the `/user` endpoint. It returns a simple HTML string `<h3>Hello User :)</h3>`.
3. `welcome()`: This method handles GET requests for the root `/` endpoint. It returns a simple HTML string `<h3>Welcome :)</h3>`.

These methods are called when a request is made to the corresponding endpoints, and the returned HTML strings will be sent as the response.

```

@Configuration

@EnableWebSecurity

public  class  SecurityConfiguration {
private  static  final  String  ADMIN = "ADMIN";
private  static  final  String  USER = "USER";



@Autowired

private  DataSource  dataSource;



@Autowired

public  void  authManager(AuthenticationManagerBuilder  auth) throws  Excep
auth.jdbcAuthentication()
.dataSource(dataSource)
.passwordEncoder(new  BCryptPasswordEncoder())
.usersByUsernameQuery("select username,password,enabled from users where us
.authoritiesByUsernameQuery("select username,authority from authorities whe

}

@Bean

public  SecurityFilterChain  securityConfig(HttpSecurity  http) throws  Exc

http.authorizeHttpRequests( (req) -> req
.antMatchers("/admin").hasRole(ADMIN)
.antMatchers("/user").hasAnyRole(ADMIN,USER)
.antMatchers("/").permitAll()
.anyRequest().authenticated()
).formLogin();

return  http.build();

}



}
```

This code defines a `SecurityConfiguration` class responsible for configuring Spring Security.

- The `@Configuration` annotation marks it as a configuration class, and `@EnableWebSecurity` enables Spring Security for the application.
- Two constants, `ADMIN` and `USER`, are defined to represent the role names.
- The `DataSource` is autowired, which will be used to configure JDBC authentication.
- The `authManager` method is annotated with `@Autowired` and accepts an `AuthenticationManagerBuilder` as a parameter. This method configures the authentication manager with JDBC authentication.
- The `jdbcAuthentication()` method configures JDBC-based authentication, specifying the data source, password encoder, and queries to retrieve user information and authorities from the database.

- The `usersByUsernameQuery()` method specifies the SQL query to retrieve the user details (username, password, enabled) based on the provided username.
- The `authoritiesByUsernameQuery()` method specifies the SQL query to retrieve the user's authorities (roles) based on the username.
- The `securityConfig` method is annotated with `@Bean` and accepts an `HttpSecurity` object as a parameter. This method configures the security filter chain using `HttpSecurity`.
- The `authorizeHttpRequests()` method configures the authorization rules for different URLs.
- The `antMatchers()` method is used to match specific URLs and apply role-based access restrictions.
- In this example, the `/admin` endpoint requires the `ADMIN` role, the `/user` endpoint requires either the `ADMIN` or `USER` role, and the `/` endpoint is accessible to all (no authentication required).
- The `permitAll()` method allows unrestricted access to the root `/` endpoint.
- The `anyRequest().authenticated()` method specifies that any other request requires authentication.
- The `formLogin()` method enables form-based authentication.

The `SecurityConfiguration` class configures the authentication and authorization aspects of Spring Security for the application.