**Resolving Circular Reference in Spring Boot**

**Introduction**

In a Spring Boot application, circular dependencies can often lead to issues where a bean depends on itself either directly or indirectly. One common scenario arises when using components like PasswordEncoder within the configuration classes of Spring Security. This article will explain why circular dependency issues occur, how they affect the application, and how to resolve them by creating a separate configuration class for PasswordEncoder.

**Understanding Circular Dependency in Spring Boot**

**What is a Circular Dependency?**

A **circular dependency** occurs when two or more beans depend on each other, directly or indirectly, creating an endless loop during the bean creation process. Spring Boot, as a dependency injection framework, manages the creation and wiring of beans. When such cycles are detected, it fails with an error indicating that a bean is currently in creation.

**Example of Circular Dependency Issue**

Consider the following simple scenario where SpringSecurity depends on PasswordEncoder:

1. **SpringSecurity Class**:
2. @Configuration
3. @EnableWebSecurity
4. public class SpringSecurity {
5. private final PasswordEncoder passwordEncoder;
6. public SpringSecurity(PasswordEncoder passwordEncoder) {
7. this.passwordEncoder = passwordEncoder;
8. }
9. // Additional configurations and security settings
10. }
11. **PasswordEncoder Class**:
12. @Configuration
13. public class PasswordEncoderConfig {
14. @Bean
15. public PasswordEncoder passwordEncoder() {
16. return new BCryptPasswordEncoder();
17. }
18. }

In this example, Spring Boot tries to create the SpringSecurity bean, which in turn depends on the PasswordEncoder bean, creating a circular reference.

**Issue Explanation**

When Spring Boot encounters such circular dependencies, it fails with the following exception:

org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'springSecurity': Requested bean is currently in creation: Is there an unresolvable circular reference?

This happens because:

* SpringSecurity requires PasswordEncoder to be available, and
* PasswordEncoderConfig depends on SpringSecurity indirectly, creating a recursive dependency.

**Resolving Circular Dependency with a Separate Configuration Class**

To solve this issue, we can decouple the creation of PasswordEncoder from the SpringSecurity configuration by introducing a separate configuration class for PasswordEncoder.

**Solution: Create a Separate PasswordEncoder Configuration**

1. **PasswordEncoderConfig Class**:
2. @Configuration
3. public class PasswordEncoderConfig {
4. @Bean
5. public PasswordEncoder passwordEncoder() {
6. return new BCryptPasswordEncoder();
7. }
8. }
9. **Using PasswordEncoder in SpringSecurity**:
10. @Configuration
11. @EnableWebSecurity
12. public class SpringSecurity {
13. private final PasswordEncoder passwordEncoder;
14. public SpringSecurity(PasswordEncoder passwordEncoder) {
15. this.passwordEncoder = passwordEncoder;
16. }
17. public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
18. auth.userDetailsService(userDetailsService()).passwordEncoder(passwordEncoder);
19. }
20. protected void configure(HttpSecurity http) throws Exception {
21. http
22. .authorizeRequests()
23. .antMatchers("/public/\*\*").permitAll()
24. .anyRequest().authenticated()
25. .and()
26. .formLogin().permitAll()
27. .and()
28. .logout().permitAll();
29. }
30. private Object userDetailsService() {
31. // Implement userDetailsService logic
32. return null;
33. }
34. }

In this solution:

* The PasswordEncoderConfig class provides the necessary PasswordEncoder bean, breaking the circular dependency.
* SpringSecurity then injects the PasswordEncoder dependency without creating a cycle.

**Conclusion**

Circular dependencies are a common challenge when configuring Spring Boot applications, especially when dealing with security components like PasswordEncoder. By creating a dedicated configuration class for PasswordEncoder, we ensure that each component is independently defined and avoids creating unresolvable circular references. This approach helps maintain a clean and scalable Spring Boot application.