O que é @Component no Spring Boot

O @Component é uma **anotação do Spring** que marca uma classe como um **bean gerenciado pelo Spring**.
Quando o Spring Boot sobe a aplicação, ele faz um *scan* (varredura) em determinados pacotes e registra todas as classes anotadas com @Component (e outras anotações derivadas, como @Service, @Repository, @Controller).

🔍 **Principais pontos importantes**

1. **Registro automático no contexto**
    - Ao anotar uma classe com @Component, você não precisa instanciá-la manualmente com new.
    - O Spring cria e gerencia o ciclo de vida do objeto (singleton por padrão).
2. **Component Scan**
    - O Spring só encontra o @Component se ele estiver em um pacote **escaneado**.
    - Em um projeto Spring Boot, o *scan* padrão começa no pacote onde está a classe principal (@SpringBootApplication) e vai para os subpacotes.
    - Se estiver em outro pacote, precisa configurar o @ComponentScan.
3. **Estereótipos específicos**
    - @Service → usado para classes de lógica de negócio.
    - @Repository → usado para classes de acesso a dados.
    - @Controller ou @RestController → usado para classes que controlam requisições web.
    - Todos eles internamente são @Component.
4. **Injeção de Dependências**
    - Qualquer classe anotada com @Component pode ser **injetada** em outras usando @Autowired, @Inject ou *constructor injection*.
5. **Escopos**
    - Por padrão, um @Component é **singleton** (uma única instância para toda a aplicação).
    - Você pode mudar o escopo com @Scope("prototype"), @Scope("request"), etc.
6. **Evita código repetitivo**
    - Sem @Component, você teria que criar instâncias manualmente e gerenciar dependências sozinho.

O que é @Configuration

O @Configuration é uma anotação do Spring que marca uma classe como **classe de configuração**, ou seja, uma classe que **declara beans manualmente** usando métodos anotados com @Bean.

Enquanto o @Component serve para o Spring **detectar e instanciar classes automaticamente**, o @Configuration é usado quando **você quer criar e configurar beans manualmente no código**, em vez de deixar o Spring criar sozinho.

🔍 **Diferença principal entre @Component e @Configuration**

	@Component	**@Configuration**
Função	Marca uma classe para ser detectada automaticamente e instanciada pelo Spring.	Define manualmente um ou mais beans a serem gerenciados pelo Spring.
Criação de Bean	Automática (a própria classe é o bean).	Manual, através de métodos @Bean.
Quando usar	Classes de serviço, controladores, repositórios, etc.	Configurações, instâncias complexas ou bibliotecas externas.

🛠️ **Como funciona**

- Uma classe @Configuration também é um **bean do Spring** (na prática é um @Component especializado).
- Dentro dela, cada método anotado com @Bean cria e registra um objeto no *Application Context*.
- O Spring garante que esses métodos sejam chamados apenas uma vez (singleton por padrão).

Diferença básica

Anotação	**De onde vem o valor**	**Como aparece na URL**	**Exemplo de URL**
@PathVariable	Do **caminho** da URL (path)	Faz parte fixa do endereço	/usuarios/10
@RequestParam	Da **query string** (parâmetros após ?)	São pares chave=valor na URL	/usuarios?id=10

🔍 **Explicando**

1. **@PathVariable**
    - Usado quando o valor é parte **do caminho** da rota.
    - Geralmente para identificar recursos.
    - Mais "RESTful" e limpo.

Ex.: /produtos/123 → 123 é um *path variable*.

2. **@RequestParam**
    - Usado para pegar parâmetros **depois do ? na URL**.
    - Ótimo para filtros, paginação, buscas.
    - Pode ter vários parâmetros opcionais.

Ex.: /produtos?categoria=eletronicos&pagina=2

```java
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

    private final CustomUserDetailsService userDetailsService;

    public WebSecurityConfig(CustomUserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/register", "/login").permitAll() // Allow access to registration and login pages
                .requestMatchers("/greet").authenticated() // Secure the /greet endpoint
                .anyRequest().permitAll() // Allow access to all other endpoints
            )
            .formLogin(form -> form
                .loginPage("/login") // Custom login page
                .defaultSuccessUrl("/greet", true) // Redirect to /greet after successful login
                .permitAll()
            )
            .logout(logout -> logout
                .permitAll()
            );
        return http.build();
    }
```

- `@Configuration` : Marca esta classe como uma classe de configuração do Spring. Informa ao Spring que esta classe contém definições de beans e configurações.
- `@EnableWebSecurity` : Habilita o suporte à segurança web do Spring Security e fornece a integração com o Spring MVC. Permite personalizar as configurações de segurança para sua aplicação.
- `authorizeHttpRequests` : Configura regras de autorização baseadas em URL.
  - `.requestMatchers(\"/register\", \"/login\").permitAll()` : Permite acesso não autenticado aos endpoints `/register` e `/login` .
  - `.requestMatchers(\"/greet\").authenticated()` : Exige autenticação para o endpoint `/greet` . Apenas usuários autenticados podem acessá-lo.
  - `.anyRequest().permitAll()` : Permite acesso a todos os outros endpoints sem autenticação.

```
    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http) throws Exception {
        AuthenticationManagerBuilder authenticationManagerBuilder =
            http.getSharedObject(AuthenticationManagerBuilder.class);
        authenticationManagerBuilder
            .userDetailsService(userDetailsService) // Use your custom UserDetailsService
            .passwordEncoder(passwordEncoder()); // Use the password encoder
        return authenticationManagerBuilder.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

    private final Map<String, User> users = new HashMap<>();
    private final PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = users.get(username);
        if (user == null) {
            throw new UsernameNotFoundException("User not found");
        }

        return org.springframework.security.core.userdetails.User.builder()
            .username(user.getUsername())
            .password(user.getPassword())
            .roles("USER")
            .build();
    }

    public void registerUser(String username, String password) throws Exception {
        if(users.containsKey(username)) {
            throw new Exception("User already exists");
        } else {
            String encodedPassword = passwordEncoder.encode(password);
            users.put(username, new User(username, encodedPassword));
        }
    }
```

- @Service: Marca esta classe como um bean de serviço do Spring.
- implements UserDetailsService: Implementa a interface UserDetailsService, que é uma interface central no Spring Security para carregar dados específicos do usuário.
- `registerUser` é chamado quando o usuário se registra. A senha é codificada usando BCryptPasswordEncoder, e as credenciais do usuário são armazenadas no HashMap.
- `loadUserByUsername` é chamado quando um usuário tenta fazer login. O método recupera os detalhes do usuário do HashMap e retorna um objeto UserDetails.

```java
@Controller
public class GreetingController {

    private final CustomUserDetailsService userDetailsService;
    private final AuthenticationManager authenticationManager;

    public GreetingController(CustomUserDetailsService userDetailsService, AuthenticationManager authenticationManager) {
        this.userDetailsService = userDetailsService;
        this.authenticationManager = authenticationManager;
    }

    @GetMapping("/greet")
    public String greet(Model model) {
        // Get the authenticated user's username
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        String username = authentication.getName();
        System.out.println("Username from context "+username);

        // Add the username to the model
        model.addAttribute("username", username);

        // Return the Thymeleaf template name
        return "greet";
    }
```

```java
@GetMapping("/login")
public String login() {
    return "login"; // Returns the login.html template
}

@GetMapping("/register")
public String register() {
    return "register"; // Returns the register.html template
}

// POST endpoint to handle user registration and auto-login
@PostMapping("/register")
public String registerUser(
        @RequestParam String username, // Username from the form
        @RequestParam String password // Password from the form
) {
    // Register the user by storing their details in the HashMap
    try {
        userDetailsService.registerUser(username, password);
    } catch (Exception userExistsAlready) {
        // Redirect to the /register endpoint
        return "redirect:/register?error";
    }
```

```java
// Authenticate the user programmatically
Authentication authentication = authenticationManager.authenticate(
    new UsernamePasswordAuthenticationToken(username, password)
);

// Set the authentication in the SecurityContext
SecurityContextHolder.getContext().setAuthentication(authentication);

// Redirect to the /login endpoint
return "redirect:/login?success";
```

- O construtor recebe CustomUserDetailsService e AuthenticationManager. CustomUserDetailsService implementa o UserDetailsService do Spring Security. Ele é usado para carregar os detalhes do usuário durante a autenticação e registrar novos usuários. O componente AuthenticationManager é responsável por autenticar os usuários. Ambas as dependências são injetadas no controlador por meio da injeção de construtor.

Estes são os seguintes endpoints da API que o controlador gerencia.

- `@GetMapping("/greet")` para mostrar a página de saudações para o usuário logado
- `@GetMapping("/login")` para mostrar a página de login
- `@GetMapping("/register")` para mostrar a página de registro
- `@PostMapping("/register")` para registrar o usuário