



Ciência da **Computação**

Programação Orientada a Objetos Avançado
Prof. Luciano Rodrigo Ferretto

Alguns conceitos importantes!!!



Arquitetura em Camadas

Separação de responsabilidades é uma das boas práticas mais importantes em sistemas backend. Vocês estruturaram o projeto em:

- **Entidade (UserEntity)**: representa os dados persistidos no banco.
- **Repositório (UserRepository)**: interface de acesso aos dados, usando o Spring Data JPA.
- **Serviço (UserService)**: onde ficam as regras de negócio (ex: validações do cadastro).
- **Controlador (UserController)**: onde chegam as requisições HTTP, atuando como ponte entre cliente e lógica do sistema.



O que é um Bean no Spring?

- Um Bean no Spring é um objeto que o Spring cria, gerencia e disponibiliza automaticamente para ser usado onde for necessário.
-  **Definição prática:**
- “Bean é um componente gerenciado pelo contêiner do Spring. Ele é criado, configurado e injetado automaticamente para você.”



O que é um Bean no Spring?

- **O Spring é um Contêiner de Injeção de Dependência**
 - O Spring funciona como um "faz-tudo" que cria objetos para você, cuida de suas dependências, e entrega tudo pronto. Isso é chamado de IoC (Inversão de Controle).
- Exemplo:
- Ao invés de escrever `new UserService()`, você deixa o Spring cuidar disso e apenas pede:

```
@Autowired  
private UserService userService;
```

ou (o ideal):

```
private final UserService userService;  
public AuthController(UserService userService) {  
    this.userService = userService;  
}
```



Como o Spring sabe que algo deve ser um Bean?

- Usamos anotações para dizer ao Spring que uma classe é um componente que ele deve gerenciar.

@Component

Marca qualquer classe como um Bean

@Service

Marca uma classe de regra de negócio

@Repository

Marca uma classe de acesso a dados

@Controller ou @RestController

Indica uma classe que recebe requisições HTTP

- “Sempre que vocês anotam uma classe com *@Service*, *@Component* ou *@Repository*, estão dizendo ao Spring: ‘Ei, crie esse objeto pra mim, gerencie ele, e injete onde for preciso.’”



Quando os Beans são criados?

- Os Beans geralmente são criados **na inicialização da aplicação** (por padrão, são **singleton**, ou seja, uma única instância por aplicação), e ficam prontos para uso até o fim da execução.



Beans via Método com @Bean

- O que é **@Bean**?
- A anotação `@Bean` é usada em um método dentro de uma classe `@Configuration` para dizer ao Spring:
- **“O objeto retornado por esse método deve ser um Bean gerenciado.”**

- Analogia simples:
- “É como se você estivesse dizendo ao Spring: ‘crie esse objeto exatamente assim, com esse passo a passo, e me entregue pronto quando eu precisar.’”

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

- 🧠 Explicação:
- Esse método cria um PasswordEncoder específico (BCryptPasswordEncoder) e registra como **Bean**.
- Assim, qualquer lugar que pedir um PasswordEncoder, o Spring sabe qual entregar.

```
@Service  
public class UserService implements UserDetailsService{  
  
    private final UserRepository repository;  
    private final PasswordEncoder passwordEncoder;  
  
    public UserService(UserRepository repository, PasswordEncoder passwordEncoder) {  
        super();  
        this.repository = repository;  
        this.passwordEncoder = passwordEncoder;  
    }  
}
```

```
@Configuration
public class ConfigSecurity {

    @Bean
    SecurityFilterChain getSecurity(HttpSecurity http, AuthTokenFilter filter) throws Exception {
        http.csrf(csrf -> csrf.disable())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/ws**", "/ws/**").authenticated()
                .anyRequest().permitAll())
            .addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
}
```

-  **Explicação:**
- Esse método monta toda a configuração de **segurança HTTP** e registra como **Bean**.
- **O Spring vai usar esse Bean para saber como proteger as rotas da sua aplicação.**



Entendendo a Segurança no Spring – Cadeia de Filtros

-  Objetivo do Spring Security:
- Interceptar todas as requisições e decidir, com base em regras e tokens, quem pode acessar o quê.
-  O que é a cadeia de filtros (**Filter Chain**)?
- O Spring Security funciona como uma grande esteira de segurança, composta por vários filtros
- **Objetos** que inspecionam e manipulam a requisição antes que ela chegue ao **Controller**.
-  Filtros são executados em ordem. Cada um pode:
 - Deixar a requisição passar
 - Interromper (ex: se o token for inválido)
 - Adicionar informações (como o usuário autenticado)



Configuração da Cadeia de Filtros no Spring Security

```
@Bean
SecurityFilterChain getSecurity(HttpSecurity http, AuthTokenFilter filter) throws Exception {
    http.csrf(csrf -> csrf.disable())
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/ws**", "/ws/**").authenticated()
            .anyRequest().permitAll())
        .addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```



Configuração da Cadeia de Filtros no Spring Security

```
http.csrf(csrf -> csrf.disable())
    .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
```

- **Desativa a proteção CSRF**, que não é necessária em APIs REST stateless (sem sessão e sem cookies).
- Define que a aplicação **não usará sessão HTTP**. Cada requisição deve conter todos os dados (ex: o JWT), pois o servidor **não armazenará nada entre uma requisição e outra**.



Configuração da Cadeia de Filtros no Spring Security

```
.authorizeHttpRequests(auth -> auth
    .requestMatchers("/ws**", "/ws/**").authenticated()
    .anyRequest().permitAll())
```

- Define quais rotas precisam de autenticação:
- Rotas que começam com `/ws` exigem usuário autenticado.
- Qualquer outra rota (`anyRequest()`) está liberada.



Configuração da Cadeia de Filtros no Spring Security

```
        .addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

- Adiciona o seu filtro (**AuthTokenFilter**, que valida o JWT) na cadeia de filtros:
 - Ele será executado antes do filtro padrão de autenticação via formulário (UsernamePasswordAuthenticationFilter).
 - Ou seja, ele verifica se já existe um JWT válido e, se sim, autentica o usuário antes que o Spring tente qualquer outra coisa.
- **return http.build();**
 - Finaliza e constrói a cadeia de segurança com todas as configurações aplicadas.



Seu filtro JWT: AuthTokenFilter

- Esse filtro é um componente essencial da segurança JWT, pois:
Valida o token, autentica o usuário e injeta a autenticação no contexto do Spring Security.

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
    throws ServletException, IOException, java.io.IOException {
String jwt = JWTUtils.getJwtFromRequest(request);
if (jwt != null) {
    String email = JWTUtils.validateToken(jwt);
    if (email != null) {
        var user = userService.loadUserByUsername(email);
        UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(user, null, null);
        auth.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(auth);
    }
}
filterChain.doFilter(request, response);
}
```



Seu filtro JWT: AuthTokenFilter

- Esse filtro é um componente essencial da segurança JWT, pois:
Valida o token, autentica o usuário e injeta a autenticação no contexto do Spring Security.

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
    throws ServletException, IOException, java.io.IOException {
String jwt = JWTUtils.getJwtFromRequest(request);
if (jwt != null) {
    String email = JWTUtils.validateToken(jwt);
    if (email != null) {
        var user = userService.loadUserByUsername(email);
        UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(user, null, null);
        auth.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(auth);
    }
}
filterChain.doFilter(request, response);
}
```



Seu filtro JWT: AuthTokenFilter

```
String jwt = JWTUtils.getJwtFromRequest(request);
if (jwt != null) {
    String email = JWTUtils.validateToken(jwt);
```

- **1** Pega o token JWT do cabeçalho Authorization (ex: "Bearer eyJhbGci...")
- **2** Se o token estiver presente, tenta validar
- **3** Valida o token e extrai o e-mail do usuário (ou outro identificador)

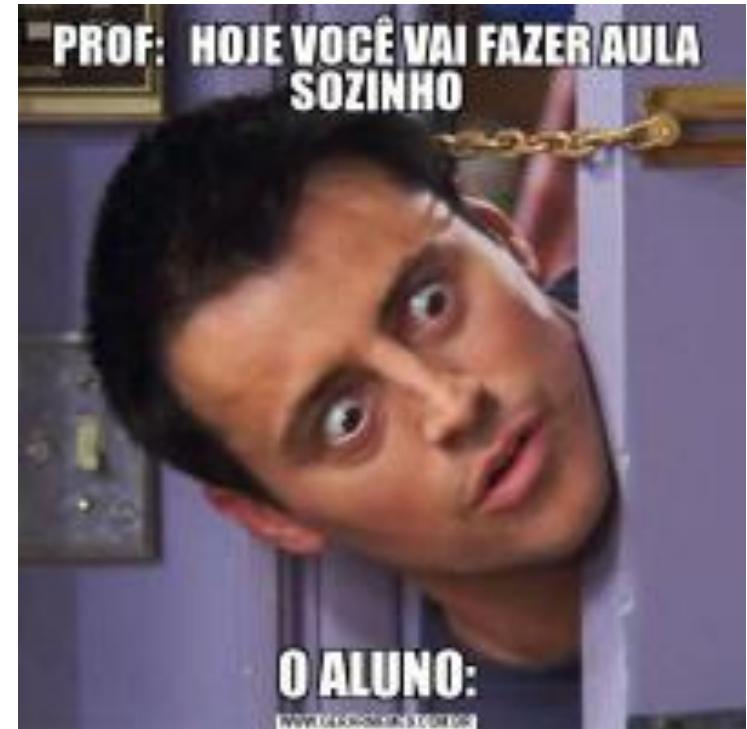


Seu filtro JWT: AuthTokenFilter

```
    if (email != null) {
        var user = userService.loadUserByUsername(email);
        UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(user, null, null);
        auth.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(auth);
    }
}
filterChain.doFilter(request, response);
```

- 4 Se o token for válido e o e-mail extraído
- 5 Carrega os dados do usuário (implementação de UserDetails)
- 6 Cria um objeto de autenticação do Spring (com o usuário autenticado)
- 7 Define detalhes adicionais da requisição (como IP, sessão, etc.)
- 8 Injeta a autenticação no contexto de segurança do Spring
- 9 Continua a cadeia de filtros (obrigatório!)

Agora é com vocês!!!!



Ciência da
Computação

Criando a Entidade PointEntity

1. Crie uma nova classe de entidade chamada PointEntity.
2. A tabela no banco deve se chamar tb_point.
3. A entidade deve conter os seguintes campos:
 - Id - Um identificador do tipo UUID, gerado automaticamente.
 - latitude e longitude - Campos para latitude e longitude, com precisão suficiente para coordenadas reais (ex: 17 dígitos com 14 casas decimais).
 - Description - Uma descrição (texto) com no máximo 250 caracteres, obrigatória.
4. A entidade deve conter um relacionamento do tipo muitos-para-um com a entidade de usuário (UserEntity), representando o dono do ponto cadastrado.
 - Esse iremos fazer juntos

Criando o repositório PointRepository

Crie uma interface chamada `PointRepository` no pacote adequado do seu projeto.

1. Essa interface deve estender `JpaRepository`, indicando que será um repositório JPA para a entidade `PointEntity`.
2. O tipo da chave primária da entidade é `UUID`.

 Dica: não é necessário implementar nenhum método extra neste momento — apenas o básico para já permitir persistência com Spring Data JPA.

Criando o serviço PointService

1. Crie uma classe PointService.
2. A classe deve injetar a dependência para o PointRepository via construtor.
3. Implemente um método `save(PointEntity point)` que:
 - Verifique se o objeto recebido é nulo.
 - Valide os campos `description`, `latitude` e `longitude`.
 - Descrição não pode ser nula ou vazia.
 - Latitude deve estar entre -90 e 90.
 - Longitude deve estar entre -180 e 180.
 - Associe o ponto ao usuário autenticado (recuperando via SecurityContextHolder).
 - Salve o ponto no banco e retorne o resultado.
4. Implemente um método `deleteById(UUID id)` que:
 - Busque o ponto pelo ID. Se não encontrar, lance uma exceção.
 - Verifique se o ponto pertence ao usuário autenticado. Se não pertencer, lance uma exceção.
 - Caso pertença, remova do banco.
5. Implemente também um método `findAll()` que:
 - Retorne todos os pontos cadastrados no banco.
6.  Dica: Use `@Transactional` nos métodos de escrita (`save`, `deleteById`), e trate exceções de forma clara.
7.  Dica extra: use o `SecurityContextHolder` para acessar o usuário autenticado e associá-lo ao ponto.

Criando a controladora PointController

Crie uma classe PointController e mapeada para o caminho base /ws/point.

1. Injeite a dependência PointService via construtor.
2. Crie um record chamado PointDTO contendo os campos: double latitude, double longitude, String descricao
3. Implemente os seguintes endpoints:
 - GET /ws/point
 - Retorna a lista de todos os pontos registrados no banco. >> Utilize o método findAll() da service.
 - POST /ws/point
 - Recebe um PointDTO no corpo da requisição. >> Converte o DTO em entidade. >> Chama o método save() da service. >> Retorna o ponto salvo.
 - DELETE /ws/point/{id}
 - Recebe o ID de um ponto via @PathVariable. >> Remove o ponto. >> Retorna uma mensagem de sucesso.
4. Implemente um @ExceptionHandler para capturar exceções e retornar HTTP 400 com a mensagem da exceção como resposta limpa (sem quebras de linha).