Auditoria

Site: Moodle institucional da UTFPR
Curso: CETEJ34 - Sistemas De Banco De Dados - JAVA_XIX (2020_01)
Livro: Auditoria

Impresso por: Alexandre Tiago Ximenes
Data: segunda, 24 ago 2020, 20:28

Sumário

1. Auditoria

Quando se trabalha com banco de dados a segurança também é algo a ser levado em consideração. Um recurso bastante utilizado como parte desse objetivo é a Auditoria.

Essa prática tem como foco criar uma espécie de *log* para identificar, por exemplo, quando e quem realizou uma determinada ação no banco de dados. Deste modo, analisando o *log* é possível obter informações como, quando um usuário autenticou no sistema, se ele alterou, apagou ou inseriu algum registro, entre outras informações a se desejar.

Uma forma bastante comum de registrar *logs* de auditoria é por meio de *triggers*, já que grande parte dos sistemas gerenciadores de banco de dados não possuem um recurso próprio para este fim.

Porém, há quem diga que trabalhar com *triggers* seria inadequado para este caso, já que pode onerar o banco de dados por adicionar rotinas que são executadas a cada ação realizada.

Por conta disso, alguns desenvolvedores preferem trabalhar com a auditoria via aplicação, ao invés de usar *triggers* no banco de dados. Deste modo, os *logs* são registrados a partir das operações realizadas na aplicação, ou seja, se uma ação de inserção é executada, esta própria operação já registra o *log*.

Uma vantagem desta prática é que o desenvolvedor não precisa se preocupar com qual SGBD vai lidar, diferentemente de quando usa *triggers*.

O uso de *triggers* necessita que elas sejam criadas diretamente no SGBD e cada SGBD tem sua própria estrutura para criar uma *trigger*.

1.1. Auditando com o Spring Data JPA

Para aqueles que pretendem usar o Spring Data JPA em seus próximos projetos, o uso de auditoria vai se tornar bastante simples. Isto porque, o Spring fornece uma implementação para este recurso e há algumas diferentes formas de utilizá-la, por exemplo, via interface **org.springframework.data.domain.Auditable**:

```
public interface Auditable<U, ID extends Serializable>
      extends Persistbale<ID> {
   U getCreatedBy();
   void setCreatedBy(final U createdBy);
   DateTime getCreatedDate();
   void setCreatedDate(final DateTime creationDate);
   U getLastModifiedBy();
   void setLastModifiedBy(final U lastModifiedBy);
   DateTime getLastModifiedDate();
   void setLastModifiedDate(final DateTime lastModifiedDate);
}
```

Essa talvez seja a forma mais complicada de usar este recurso, já que você teria que implementar cada um dos métodos da interface. Então, uma opção mais simples é estender a classe abstrata **org.springframework.data.jpa.domain.AbstractAuditable** a qual é uma implementação da interface **Auditable**, como se pode ver a seguir:

```
@MappedSuperclass
public abstract class AbstractAduitable<U, PK extends Serializable>
      extends AbastractPersistable<PK>
      implements Auditable<U, PK> {
   private static final long seriaLVersionUID =
      141481953116476081L;
   @ManyToOne
   private U createdBy;
   @Temporal(TemporalType.TIMESTAMP)
   private Date createdDate;
   @ManyToOne
   private U lastModifiedBy;
   @Temporal(TemporalType.TIMESTAMP)
   private Date lastModifiedDate;
   // getters/setters foram omitidos nesta listagem
}
```

Outro ponto a destacar é que **AbstractAuditable** estende a classe **AbstractPersistable**, a qual já foi abordada. Então, seria possível substituir **AbstractPersistable** por **AbstractAuditable** sem ter nenhum impacto negativo em suas entidades.

Para usar o sistema de auditoria do Spring Data é necessário incluir ao projeto a dependência "**spring-aspects**" do Spring Framework:

```
<dependency>
   <groupId>org.springrfamework</groupId>
   <artifactID>spring-aspects</artifactID>
   <version>5.2.4.RELEASE</version>
</dependency>
```

Observando o código é possível notar que existem quatro atributos mapeados via JPA. Esses estarão fazendo a referência aos métodos **set / get** de **Auditable**. E a função dos atributos mapeados é registrar no log de auditoria a seguinte informação:

- createdBy - salva em uma coluna o nome do usuário que realizou a inserção da entidade;
- createdDate - registra em uma coluna a data e hora que a inserção foi realizada;
- lastModifiedBy - quando uma edição for realizada na entidade, esta instrução vai salvar em uma coluna o nome do usuário que realizou a alteração;
- lastModifiedDate - registra a data e hora que a alteração foi realizada na entidade.

Desta forma, cada tabela no banco de dados deverá ter quatro colunas adicionais para o *log* de auditoria. Mas ainda há outra maneira de lidar com este recurso, que é usando algumas anotações na própria classe de entidade, ao invés da classe **AbstractAuditable** ou da interface **Auditable**.

Veja o exemplo a seguir, na Listagem 7.1, que demonstra este processo na entidade **Contato**.

LISTAGEM 7.1: USANDO ANOTAÇÕES PARA AUDITAR.

```
@EntityListeners(AuditingEntityListener.class)
public class Contato extends AbstractPersistable<Long> {
   // atributos já apresentados foram omitidos nesta listagem
   @Column(name = "created_by")
   @CreatedBy
   private String createdBy;
   @Column(name = "created_date")
   @CreatedDate
   private Date createdDate;
   @Column(name = "modified_by")
   @LastModifiedBy
   private String modifiedBy;
   @Column(name = "modified_date")
   @LastModifiedDate
   private Date modifiedDate;
   // getters/setters foram omitidos nesta listagem
}
```

Analisando a classe **Contato**, observe que foram adicionados quatro atributos: **createdBy, createdDate, modifiedBy e modifiedDate**.

Esses atributos poderiam ser nomeados de qualquer outra forma, isto porque, o importante são as anotações do Spring que vão fazer a referência ao recurso de auditoria.

Estas anotações são a **@CreatedBy, @CreatedDate, @LastModifiedBy e @LastModifiedDate** e, o objetivo de cada uma delas é o mesmo dos atributos já apresentados na classe **AbstractAuditable**.

Além disso, é preciso ainda incluir um ouvinte na entidade para que o Spring saiba quando um objeto do tipo contato está sendo criado ou alterado e assim, atribuir automaticamente os valores que serão salvos na tabela do banco de dados nos atributos anotados.

O registro do ouvinte é realizado pela anotação da JPA **@EntityListeners** e, como valor, a anotação recebe a classe **AuditingEntityListener**.

Esta classe é fornecida pelo Spring Data JPA e está no pacote **org.springframework.data.jpa.domain.support**.

É claro que para ter no registro de log o usuário que está realizando uma ação no banco de dados, é necessário que a aplicação tenha um sistema de autenticação, por exemplo, o Spring Security. Neste caso, será preciso integrar os conceitos de autenticação e de auditoria. Para isso, o Spring tem dois passos necessários.

O primeiro, é criar uma classe que implemente a interface **AuditorAware** para ter acesso ao método **getCurrentAuditor()**. Este método vai fazer com que se tenha acesso ao nome do usuário que está autenticado na aplicação, para que o recurso de auditoria possa usá-lo no registro do *log*.

Entretanto, a implementação do corpo do método fica por conta do desenvolvedor e um exemplo simples que pode ser usado está presente na Listagem 7.2.

LISTAGEM 7.2 : O MÉTODO GETCURRENTAUDITOR().

```
import org.springframework.data.dmoain.AuditorAware;
import org.springframework.securtiy.core.Authentciation;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.securtiy.core.userdetalis.user;
public class SpringSecurityAduitorAware implements AuditorAware<String> {
  public String getCurentAuditor(){
      Authentication auth = SecurityContextHolder
         .getContext();
         .getAuthentication();
      if (auth == null || ! auth.isAuthenctaited()){
         return null;
      }
      return((User) auth.getPrincipal()).getUsername();
   }
}
```

Basicamente o processo do método é recuperar um objeto **Authentication** a partir do contexto do SpringSecurity. Com esse objeto recuperado, basta retornar o nome do usuário.

O Spring vai ter acesso ao usuário autenticado por meio da classe **UserDetailService**, usada pelo SpringSecurity para autenticar um usuário.

Feito isso, o passo seguinte é informar ao Spring que a aplicação vai trabalhar com o recurso de autenticação e também de onde o nome do usuário autenticado deve ser recuperado. Esse processo é bem fácil, como mostra a Listagem 7.3.

LISTAGEM 7.3 : HABILITANDO O SISTEMA DE AUDITORIA.

```
import org.springframework.context.annotation.Bean;
import org.springfrmaework.data.domain.AuditorAware;
import org.springfrmaewotk.data.jpa.repositoy.config.EnableJpaAuditing;
import org.springframework.stereotype.Component;
@Component
@EnableJpaAuditing
public class SpringAutditingConfig{
   @Bean
   public AuditorAware<String> auditorAware() {
      return new SpringSecuirtyAuditorAware();
   }
}
```

Veja que a classe **SpringAuditingConfig** foi anotada com **@Component** para transformá-la em um *bean* gerenciado pelo Spring e **@EnableJpaAuditing** para habilitar o recurso de auditoria.

Além disso, outro *bean*, o **auditorAware**, foi incluído na classe, como um método, com a finalidade de dizer em qual local o Spring deve recuperar o nome do usuário, neste caso a classe SpringSecurityAuditorAware apresentada na Listagem 7.2.

Com isso, o sistema de auditoria está configurado e pronto para ser usado na tabela de **Contatos**. Sempre que um usuário autenticado inserir ou alterar um registro, seu nome de usuário e a data das ações serão incluídos automaticamente na mesma operação de **insert** ou **update**.