14

Mãos na massa

Aqui, você pode verificar, resumidamente, as alterações feitas no código do projeto neste capítulo.

Primeiramente, vamos alterar a estratégia do `HibernateMapping` para `update` em *JpaConfigurator*:

```
public class JpaConfigurator {
    ...
    @Bean
    public LocalContainerEntityManagerFactoryBean getEntityManagerFactory(DataSource dataSource

        ...
        props.setProperty("hibernate.hbm2ddl.auto", "update");
        ...
    }
}
```

Para testar o *cache*, precisamos de um `EntityManager` . Para isso, faremos um `lookup` ao contexto do Spring e com o EntityManager pronto para usar, podemos realizar um `find` pelo `id` do produto que acabamos de cadastrar no banco:

```
public class TesteCache {
    public static void main(String[] args) {

        ApplicationContext ctx = new AnnotationConfigApplicationContext(JpaConfigurator.class);

        EntityManagerFactory emf = (EntityManagerFactory) ctx.getBean(EntityManagerFactory.clas
        EntityManager em = emf.createEntityManager();

        Produto produto = em.find(Produto.class, 1);
        System.out.println("Nome: " + produto.getNome());
    }
}
```

Adicionamos mais um `select` , para testarmos o **cache**:

```
Produto outroProduto = em.find(Produto.class, 1);
        System.out.println("Nome: " + outroProduto.getNome());
```

Agora, podemos testar com 2 `EntityManager` . Será que a resposta é diferente? Altere/crie as seguintes linhas:

```
EntityManager em2 = emf.createEntityManager(); // criando o segundo EntityManager
Produto outroProduto = em2.find(Produto.class, 1); // buscando a mesma entidade com o segundo E
```

Agora que terminamos o teste do cache de primeiro nível podemos retornar o `HibernateMapping` para `create-drop` .

Feito isso, no nosso `JpaConfigurator` podemos configurar o *cache de segundo nível*:

```
public class JpaConfigurator {
    ...
    @Bean
    public LocalContainerEntityManagerFactoryBean getEntityManagerFactory(DataSource dataSource

        ...
        props.setProperty("hibernate.cache.use_second_level_cache", "true");
        ...
    }
}
```

Com persistence.xml adicionaríamos uma tag com a mesma propriedade:

Precisamos dizer, agora, qual provedor de cache estamos usando. Para isso, precisamos adicionar novamente no nosso `JpaConfigurator` a propriedade:

```
props.setProperty("hibernate.cache.region.factory_class", "org.hibernate.cache.ehcache.Singleto
```

Para começarmos a fazer uso do cache de segundo-nível, vamos anotar a nossa classe `Produto` com `@Cache` e passar a estratégia que será utilizada:

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Produto {
```

Podemos anotar, também, as associações da nossa classe Produto:

```
@ManyToMany
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
private List<Categoria> categorias = new ArrayList<>();
```

E na nossa classe de `Categoria` :

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Categoria {
```

E também na classe de `Loja` :

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Loja {
```

Agora, para podermos usar o *cache de queries*, precisamos habilitá-lo em nossas configurações, `JpaConfigurator`, adicionando a seguinte propriedade

```
props.setProperty("hibernate.cache.use_query_cache", "true");
```

Depois disso, precisamos avisar ao Hibernate, para ele salvar no cache os resultados de alguma query. Para isso, iremos passar a seguinte *pista* em nosso método `getProdutos()` do `ProdutoDao`:

```
public List<Produto> getProdutos(String nome, Integer categoriaId, Integer lojaId) {
    ...

    TypedQuery<Produto> typedQuery = em.createQuery(query.where(conjuncao));
    typedQuery.setHint("org.hibernate.cacheable", "true");

    return typedQuery.getResultList();
}
```