Esfinge QueryBuilder

CAMADA DE PERSISTÊNCIA SIMPLES E RÁPIDA

Entenda como utilizar esse novo framework para persistência que diminui muito a quantidade de código necessária para criação de consultas.

Muitas vezes repetimos diversas vezes o mesmo pedaço de código ou uma mesma informação em nossa aplicação. De certa forma, repetimos as informações da entidade na classe e na tabela do banco de dados, repetimos o código para execução de consultas etc. De certa forma, o próprio nome do método quando é bem descritivo repete a informação que está em seu corpo. E se fosse possível que o programa soubesse somente pelo nome do método o que ele precisa fazer? É possível! Conheça o Esfinge QueryBuilder!

O projeto Esfinge é um grande guarda-chuva para diversos frameworks que são baseados na mesma filosofia. Seu objetivo é ao mesmo tempo prover flexibilidade de alteração de comportamento, aliado a um modelo que permita à equipe desenvolver o software com grande produtividade. Sua principal característica está em buscar nos metadados das próprias classes grande parte das informações necessárias para seu processamento, permitindo a diminuição de código redundante e repetitivo. Isso é associado à utilização de padrões de projeto que permitam que o comportamento configurado pelos metadados possa ser estendido e modificado.

Em 2011, iniciou-se um novo ciclo no desenvolvimento do framework Esfinge. O projeto original, desenvolvido em 2006, foi descontinuado e os frameworks estão todos sendo desenvolvidos do zero novamente. O primeiro framework a ser lançado foi o QueryBuilder que está em sua versão 1.3. Existem outros projetos do Esfinge 2 em desenvolvimento, porém nenhum deles ainda foi lançado. Dentre eles estão um framework para integração de aplicações, um framework para Adaptive Object Models e outro de comparação entre instâncias. O Esfinge é um framework open-source e pode ser baixado pelo endereço **http://esfinge.sf.net**.

Uma boa frase para definir o Esfinge QueryBuilder seria: "para o bom desenvolvedor, meia linha de código basta". Brincadeiras à parte, a ideia principal do QueryBuilder é a partir da assinatura de um método definido em uma interface gerar sua implementação. Através de um proxy dinâmico, o Esfinge QueryBuilder interpreta o nome do método e seus metadados associados, e gera a query correspondente em tempo de execução.

O objetivo deste artigo é apresentar o framework QueryBuilder e suas principais funcionalidades para a geração de consultas. O artigo começa dando uma visão geral do framework, explicando como configurar e utilizar o framework. Em seguida, são apresentados outros recursos como a criação de consultas dinâmicas, a definição de novos termos de domínio e a utilização de diversos tipos de comparação.

Esfinge QueryBuilder em 1 Minuto

A versão atual do QueryBuilder funciona junto com alguma implementação de JPA. Futuramente, pretende-se criar componentes plugáveis que acessem a base de dados através de outras bibliotecas ou que acessem bancos não-relacionais através de suas APIs. Sendo assim, o primeiro passo é possuir classes mapeadas para as tabelas do banco. Para os exemplos deste artigo, vamos utilizar as classes Person e Address apresentadas respectivamente nas Listagens 1 e 2. Observe que grande parte do mapeamento utilizou as convenções de nomenclatura.

Listagem 1. Classe Person mapeada para o banco de dados.

```
@Entity
public class Person {

    @Id
    private Integer id;
```

Eduardo Guerra | guerra@mundoj.com.br | @emguerra

Desenvolvedor de frameworks, pesquisador em design de software, editor-chefe da revista MundoJ e professor do ITA, onde concluiu sua graduação, mestrado e doutorado. Possui diversas certificações da plataforma Java e experiência como arquiteto de software nas plataformas Java SE, Java EE e Java ME. Participou de projetos open-source, como SwingBean, ClassMock e Esfinge, e acredita que um bom software se faz mais com criatividade do que com código.

```
    private String name;
    private String lastName;
    private Integer age;

    @ManyToOne
    @JoinColumn(name = "ADDRESS_ID",
       referencedColumnName = "ID")
    private Address address;

    //getters e setters omitidos
}
```

Listagem 2. Classe Address mapeada para o banco de dados.

```
@Entity
public class Address {

    @Id
    private int id;
    private String city;
    private String state;

    //getters e setters omitidos
}
```

Para utilizar o QueryBuilder, o primeiro passo é criar uma interface com métodos cujos nomes descrevam as consultas que devem ser feitas. Quando se deseja que sejam disponibilizadas operações CRUD para uma determinada classe, essa interface deve estender a interface Repository, porém isso não é obrigatório. A Listagem 3 mostra um exemplo de uma interface desse tipo.

Listagem 3. Interface para a geração das consultas com o QueryBuilder.

```
public interface PersonDAO extends Repository<Person>{
    public List<Person> getPersonByLastName(
       String lastname);
    public List<Person> getPersonByAddressCity(
       String city);
    public List<Person> getPersonOrderByAge();
}
```

O próximo (e último) passo é criar uma instância dessa interface e começar a utilizar. É só isso mesmo! A ideia do framework é buscar no nome do método e nos metadados da classe envolvida as informações necessárias para a construção da consulta. Observe, por exemplo, que "LastName" no nome do método se refere a uma propriedade da própria classe e "AddressCity" se refere à propriedade city da propriedade address. A sintaxe é a mesma, mas combinada com outras informações, é possível determinar exatamente a consulta que deve ser executada. Se os nomes já dizem a consulta que deve ser feita, pra que mais código? Abaixo segue o código necessário para criar uma instância que implementa a interface:

```
PersonDAO dao = QueryBuilder.create(PersonDAO.class);
```

O QueryBuilder utiliza o recurso de proxy dinâmico da API de reflexão do Java para gerar em tempo de execução uma classe que implementa a interface. A interpretação do nome do método e dos metadados é feita uma vez só e depois aproveitada para outras invocações do mesmo método, o que melhora o desempenho do framework. Caso o framework não reconheça o nome do método como uma consulta válida, será gerado um erro em sua invocação. As próximas seções entram em mais detalhes a respeito de como configurar o framework e das regras e recursos que podem ser utilizados na definição dos métodos da interface.

Configuração do QueryBuilder

Como foi dito anteriormente, a arquitetura interna do QueryBuilder foi criada de forma a possibilitar o seu uso com diversos frameworks de persistência, porém atualmente só possui versão de produção a implementação para JPA. Ela é compatível com o JPA 1, pois funciona através da geração de consultas JPA-QL e não da API Criteria, que foi inserida na versão 2. Sendo assim, a primeira configuração que precisa ser feita é a configuração do próprio JPA, além da inclusão do driver do banco de dados e das bibliotecas do JPA. Essa parte da configuração não será abordada no artigo.

Para configurar o Esfinge QueryBuilder, é necessário acrescentar os arquivos esfinge_querybuilder_core_1_X.jar e esfinge_querybuilder_jpa1_1_X.jar no classpath da sua aplicação. Caso sua aplicação rode fora de um servidor de aplicações ou de um servidor web, também é preciso acrescentar as bibliotecas el-api.jar e jasper-el.jar. Se você irá utiliza transações distribuídas com JTA, gerenciadas, por exemplo, pelo Spring ou pelo servidor de aplicações, essas são todas as bibliotecas que você precisará. Caso vá controlar as transações localmente, o arquivo esfinge_querybuilder_jpalocal1_1_X.jar também precisa ser adicionado.

100% TDD

Um fato interessante é que o Esfinge QueryBuilder foi construído totalmente utilizando TDD, mesmo sendo um framework. Toda modelagem das interfaces e dos módulos do framework foi feita através da definição de Mock Objects dentro de testes de unidade. Pelo fato do framework ser um tipo de software em que questões de design são tomadas a todo momento, optar por essa abordagem de desenvolvimento foi na verdade aceitar um grande desafio.

Felizmente, com o tempo, essa escolha se mostrou sábia, pois questões não previstas inicialmente criaram a necessidade de algumas refatorações que mudaram profundamente questões internas no framework. Com a presença dos testes de unidade, foi possível realizar as alterações necessárias de forma segura. Além disso, o framework hoje conta com uma suíte de testes com quase 100% de cobertura do código!

O próximo passo da configuração envolve a criação de uma classe que retorna o EntityManager ao framework e a criação de um arquivo que configura o uso dessa classe. A classe para prover o EntiryManager precisa implementar a interface EntityManagerProvider e implementar métodos que retornem o EntityManager e o EntityManagerFactory. A Listagem 4 apresenta um exemplo da implementação, o cenário do JPA esta sendo utilizado fora de um servidor de aplicação. Em outros cenários, é só recuperar essas instâncias onde elas estiverem (no contexto JNDI, por exemplo) e retorná-las. No site do projeto, existe um tutorial que ensina a configurar o QueryBuilder para recuperar o EntityManager do contexto do Spring.

Listagem 4. Exemplo de EntityManagerProvider.

```
public class TestEntityManagerProvider implements
EntityManagerProvider {
   @Override
   public EntityManager getEntityManager() {
      return getEntityManagerFactory().
         createEntityManager();
   }

   @Override
   public EntityManagerFactory
      getEntityManagerFactory() {
      return Persistence.createEntityManagerFactory(
         "database_test");
   }
}
```

O Esfinge utiliza a classe ServiceLoader (apresentada no artigo "Estratégias para Criação de Objetos Visando Modularidade" da edição 44 da MundoJ) para carregar suas classes. Por isso, é preciso criar um arquivo chamado "org.esfinge.querybuilder.jpa1.EntityManagerProvider" dentro do diretório META-INF/services. Esse arquivo deve conter apenas o nome completo da classe criada. Pronto! A configuração está feita! Agora é só colocar a "mão na massa" e utilizar o framework!

Assinatura dos métodos

Para que o QueryBuilder consiga entender o significado dos métodos, existe um padrão de nomenclatura que precisa ser seguido. Os nomes dos métodos devem indicar qual é a entidade que deverá ser retornada na consulta, quais os filtros devem ser utilizados e por qual propriedade a consulta deve ser ordenada. Abaixo, segue uma consulta que demonstra essa estrutura básica:

```
List<Person> getPersonByLastNameAndAddressCityO
rderByAge(String lastname, String city);
```

A tabela 1 apresenta cada parte da assinatura do método explicada.

Esse exemplo demonstrou a estrutura básica do nome de um método. As próximas duas subseções apresentam outras opções.

Tipos de comparação

Caso nada seja especificado, o filtro irá retornar os resultados que forem iguais ao parâmetro passado. Porém, existem outras formas de comparação que podem ser utilizadas. Por exemplo, pode-se desejar que se retorne o que for diferente do filtro passado. Para valores numéricos, pode-se desejar utilizar operadores como maior, menor, maior ou igual e menor ou igual. Já para texto, pode-se retornar valores começados, terminados ou que contenham o valor passado como parâmetro.

O Esfinge QueryBuilder permite que a definição do tipo de comparação seja feita de duas formas diferentes: uma utilizando anotações e outra utilizando o próprio nome do método. No caso da utilização de anotações, ela deve ser utilizada para configurar o parâmetro desejado. No caso da utilização do próprio nome do método, uma String com o tipo de comparação deve ser adicionada depois do nome parâmetro. As anotações possuem exatamente os mesmos nomes das Strings que devem ser adicionadas. Os valores suportados são: Equals (opção default), Lesser, Greater, LesserOrEquals, GreaterOrEquals, NotEquals, Contains, Starts, Ends. A Listagem 5 apresenta exemplos de consultas que utilizam anotações e a Listagem 6 apresenta exemplos que utilizam o nome do próprio método.

Listagem 5. Exemplo de comparações com uso de anotação.

```
List<Person> getPersonByAge(@Greater Integer age);
List<Person> getPersonByName(@Starts String name);
```

Estendendo o Esfinge QueryBuilder para outras fontes de dados

O Esfinge Query Builder é dividido em duas partes bem modularizadas: uma interpreta a assinatura dos métodos com as anotações e outra que cria as consultas da forma desejada e as executa na base de dados. Como foi dito, hoje existe apenas a implementação para JPA, porém quem quiser pode criar novas implementações para camadas de interface de persistência diferentes e até mesmo para bases de dados não relacionais. Não está no escopo do artigo detalhar como o framework pode ser estendido e em breve será criado um tutorial no site do projeto a respeito de como isso pode ser feito.

Listagem 6. Exemplo de comparações com inclusão no nome do método.

```
public List<Person> getPersonByAgeLesser(Integer age);

public List<Person>
getPersonByLastNameNotEquals(String name);

public List<Person>
getPersonByNameStartsAndAgeGreater(String name,
Integer age);
```

Opções de ordenação

Como foi mostrado inicialmente, a cláusula ”OrderBy” pode ser utilizada no final de um método para definir que uma consulta deve ser ordenada por uma determinada propriedade. Porém, no exemplo inicial, foi apresentada a utilização mais simples desse recurso.

Pode-se definir mais de uma propriedade para que seja feita a ordenação, sendo que nesse caso será feito primeiro a ordenação pela primeira propriedade, seguida pela segunda e assim por diante. As propriedades podem ser encadeadas utilizando “And”. A ordenação também pode ser ascendente ou descendente, bastando a inclusão, respectivamente, de “Asc” ou “Desc” depois do nome da propriedade. Caso nada seja incluído, será utilizada a opção ascendente. A Listagem 7 apresenta exemplos de consultas que utilizam esses recursos.

Listagem 7. Exemplo de opções de comparação.

```
public List<Person> getPersonByAgeOrderByNameDesc(
@Greater Integer age);

public List<Person>
getPersonOrderByNameAndLastName();
```

Métodos do repositório

Apesar de não ser uma de suas funcionalidades principais, o Esfinge QueryBuilder também disponibiliza funcionalidades que permitem a execução de operações CRUD. Para possuir essas operações dispo-

Tabela 1. Exemplo de nome de método explicado passo-a-passo.

Trecho de Código	Explicação
List<Person>	O retorno do método pode ter o tipo de uma lista da entidade ou o tipo da própria entidade. A partir do tipo declarado no retorno o método será inferido se a consulta espera um resultado ou uma lista de resultados.
get	Todo método de consulta deve começar com get.
Person	Nome da entidade que será retornada na consulta. O JPA deve reconhecer a entidade por esse nome.
By	Separa o nome da entidade do nome das propriedades que serão utilizadas para o filtro da consulta. Caso não haja nenhum filtro, essa parte pode ser emitida.
LastName	Em seguida do “By” deve vir uma propriedade a ser utilizada como filtro. O nome utilizado deve ser o nome da propriedade, porém começado com letra maiúscula, mesmo que ela seja formada por mais de um nome.
And	As propriedades podem ser separadas por “And” ou por “Or” na consulta, de acordo com a combinação desejada nos filtros.
AddressCity	É possível incluir em uma consulta propriedades de classes que compõem a entidade principal da consulta. No caso, será acessada a propriedade “city” que fica dentro da propriedade “address”. A sintaxe é equivalente a de propriedades com nomes compostos. O framework procura na estrutura da classe para ver qual é a propriedade que deverá ser utilizada.
OrderBy	Deve ser utilizado depois das propriedades que serão utilizadas como filtro, caso deseje-se ordenar os resultados por alguma propriedade. Caso não se deseje uma ordem específica, essa parte pode ser omitida.
Age	Depois do “OrderBy” são listadas as propriedades que devem ser consideradas na ordenação. Mais a frente, serão detalhadas as opções para a ordenação.
String lastname, String city	Os parâmetros do método devem seguir a mesma ordem que é especificada no nome do método. Eles também precisam ser do mesmo tipo da propriedade, pois se não forem será lançada uma exceção.

níveis, basta implementar a interface Repository conforme foi exemplificado na Listagem 3. Vale lembrar que a implementação dessa interface não é obrigatória. Essa interface deve ser implementada com o tipo genérico igual o da entidade para a qual se deseja as operações. Seguem os métodos disponibilizados nessa interface:

- E save(E obj) – Grava no banco de dados o objeto passado como parâmetro. O objeto é inserido caso não exista ou atualizado caso já exista.
- void delete(Object id) – O método exclui da base de dados a entidade cujo id foi passado como parâmetro.
- List<E> list() – Retorna uma lista com todas as entidades do banco de dados.
- E getById(Object id) – Retorna uma instância de acordo com o id passado como parâmetro.
- List<E> queryByExample(E obj) – Faz uma query que faz a busca de acordo com as propriedades populadas do objeto.

Quando o proxy dinâmico que é gerado em tempo de execução detecta que o método invocado pertence à interface Repository, ele redireciona para uma implementação dessa interface. O framework inclusive permite que essa implementação seja substituída, porém essa configuração mais avançada foge do escopo deste artigo.

Termos de domínio

Na verdade, o Esfinge QueryBuilder define uma DSL (Domain Specific Language) para o domínio de consultas de dados. Essa DSL é composta pelo padrão de nomenclatura utilizado no método, pelas convenções de código (como o retorno) e pelas anotações utilizadas. Uma funcionalidade interessante provida pelo framework está na possibilidade de definição de novos termos de domínio que podem ser utilizados como parte do nome dos métodos.

O termo de domínio é definido pela anotação @DomainTerm, que deve ser definida na própria interface onde os métodos estão sendo definidos. Esses termos de domínio representam termos que definem um subconjunto de uma determinada entidade. Por exemplo, se a entidade for uma ordem de compra, o termo "concluida" pode restringir somente às ordens que já foram entregues ao cliente.

Essa anotação deve definir o nome do termo que será utilizado e as condições que serão utilizadas para definir o subconjunto. A Listagem 8 apresenta um exemplo simples de uso de termos de domínio. A anotação define o termo "major" e, nesse caso, apenas uma condição é definida. Com essa definição, os termos podem ser adicionados nos nomes dos métodos após o nome da entidade, acrescentando, dessa forma, as condições que a definem na consulta que será gerada.

Listagem 8. Definição de um termo de domínio simples.

```
@DomainTerm(term="major",
conditions=@Condition(property="age",
      comparison=ComparisonType.GREATER_OR_EQUALS,
        value="18"))
    public interface PersonQuery{
  public List<Person> getPersonMajor();
}
```

Um termo de domínio pode possuir mais de um nome, sendo que se definir o termo "maior de idade", ele será utilizado como "MaiorDeIdade" no método. O termo de domínio também pode possuir mais de uma condição associada, bastando para isso encadear mais de um @Condition na propriedade "comparison". Mais de um termo de domínio pode ser utilizado na mesma consulta, bastando incluir um após o outro. Finalmente, também é possível definir mais de um termo de domínio na mesma interface, utilizando-se a anotação @DomainTerms. A Listagem 9 exemplifica todas essas situações. É importante ressaltar que os termos de domínio podem ser combinados com qualquer outro recurso de criação de consultas no QueryBuilder.

Listagem 9. Definições avançadas de termos de domínio.

```
@DomainTerms({
  @DomainTerm(term="teenager",
  conditions={@Condition(property="age",
        comparison=ComparisonType.
        GREATER_OR_EQUALS, value="13"),
  @Condition(property="age",
        comparison=ComparisonType.
     LESSER_OR_EQUALS, value="19")}),
        @DomainTerm(term="young child",
  conditions=@Condition(property="age",
        comparison=ComparisonType.LESSER
        ,value="7")),
        @DomainTerm(term="joseense",
  conditions=@Condition(property="address.city",
        comparison=ComparisonType.EQUALS ,
        value="São José dos Campos"))
})
public interface PersonQuery{
  public List<Person> getPersonTeenagerByName(@
Contains String name);
  public List<Person> getPersonYongChildByAddressCity
(String city);

  public List<Person> getPersonTeenagerJoseense();
}
```

Lidando com parâmetros nulos

Normalmente, as consultas definidas nos métodos do Esfinge QueryBuilder não esperam receber um valor nulo como parâmetro. Espera-se nesse caso que seja lançado um erro, porém o resultado irá depender

Esfinge QueryBuilder x Spring Data

Quando o Esfinge QueryBuilder começou a ser desenvolvido, não se tinha conhecimento do framework Spring Data, que segue basicamente a mesma ideia. Apesar de o conceito ser o mesmo, existem algumas diferenças em termos de funcionalidade. O Spring Data fornece a opção de se criar consultas paginadas, definição explícita de consultas e é possível passar a propriedade de ordenação como parâmetro, entre outras funcionalidades que ainda não estão presentes no QueryBuilder. Para o lado do QueryBuilder, o destaque fica no uso das anotações para configurações, no suporte a termos de domínio e na possibilidade de remover da consulta parâmetros que receberam nulo.

Os desenvolvedores Java só têm a ganhar tendo opções de framework diferentes de qualidade para a geração de consultas de forma simples e rápida. Porém, o Esfinge QueryBuilder não vai "deixar barato" e vai "brigar" sempre para fornecer mais funcionalidades com melhor desempenho!

da implementação JPA utilizada. O framework provê duas anotações que podem ser utilizadas para se lidar com valores nulos: @CompareToNull que realmente adiciona uma restrição na consulta do tipo IS NULL para aquela propriedade; e @IgnoreWhenNull que irá desconsiderar aquela restrição caso seja passado um valor nulo.

A Listagem 10 apresenta exemplos de assinaturas de métodos que utilizam esses recursos. Por exemplo, o método getPersonByCompany() irá buscar as pessoas que não possuem companhia configurada (valor NULL no banco de dados) caso receba um valor nulo como parâmetro. Já o segundo método, getPersonByNameAndLastName(), irá ignorar o parâmetro caso seja recebido o valor nulo. Por exemplo, caso sejam passados os parâmetros (null, "S"), serão buscadas as pessoas com sobrenome começado com S e caso seja passado ("S", null) serão buscadas as pessoas com o nome começado com S.

A utilização do @IgnoreWhenNull é muito útil para consultas que permitem que o usuário escolha dentre alguns parâmetros qual irá utilizar, sendo que os outros devem ser ignorados. Um fator que torna esse recurso do framework ainda mais atrativo é o fato desse tipo de consulta ser algo normalmente trabalhoso de ser criado manualmente.

Listagem 10. Consultas com as anotações @CompareToNull e @IgnoreWhenNull.

```
public interface PersonQuery{
   public List<Person> getPersonByCompany(
      @CompareToNull String company);
   public List<Person> getPersonByNameAndLastName(
      @Starts @IgnoreWhenNull String name,
      @Starts @IgnoreWhenNull String lastname);
}
```

O futuro do Esfinge QueryBuilder

O Esfinge Framework é um projeto que está "a todo vapor"! Além do QueryBuilder, existem vários outros módulos que estão sendo planejados e desenvolvidos. Desde que foi lançado, o QueryBuilder vem tendo releases frequentes, com intervalos variando entre um e dois meses. A cada nova versão, alguma funcionalidade interessante é acrescentada!

Dentre as funcionalidades que devem estar sendo acrescentadas no framework nas próximas versões estão:

- Funcionamento utilizando JDBC puro, sem a dependência ao JPA.
- Criação de consultas e operações CRUD em bancos NoSQL, como MongoDB.
- Paginação de consultas.
- Objetos complexos como parâmetro para consultas com muitas restrições.
- Recebimento da propriedade do Order By como parâmetro.
- Recebimento de listas e arrays nos parâmetros.

Considerações finais

Este artigo apresenta o framework Esfinge QueryBuilder, que a partir das assinaturas dos métodos de uma interface, consegue executar consultas em uma base de dados. O uso desse framework tem o potencial de diminuir significativamente a quantidade de código de persistência, consequentemente aumentando a produtividade da equipe e simplificando a manutenção da aplicação.

A ideia do Esfinge QueryBuilder não é permitir a criação de todas as consultas de um sistema, mas facilitar a criação de aproximadamente 90% delas. Esse ganho em produtividade já será significativo no projeto e compensará a utilização do framework.

Então, o que está esperando? Entre no site do projeto, baixe a última versão e experimente! Tenho certeza que irá considerar seriamente sua inclusão em seus próximos projetos! Se você gostou do projeto e está a fim de participar, entre em contato que sua ajuda será mais do que bem-vinda!

Agradecimentos

Aproveito a oportunidade para agradecer a empresa GSW pelo apoio que tem dado ao projeto Esfinge! Além de participar de seu desenvolvimento, a GSW ainda hospeda e mantém o portal do projeto. Espero que essa atitude sirva de exemplo para outras empresas brasileiras também investirem em projetos de código aberto.

/referências

> *Esfinge Framework – http://esfinge.sf.net*

> *Spring Data – http://www.springsource.org/spring-data*