UNIVERSIDADE DO OESTE DE SANTA CATARINA

VITOR ZACHI JUNIOR

ALLIGATOR: *framework* para *multitenancy* baseado em JPA

Xanxerê
2012

VITOR ZACHI JUNIOR

ALLIGATOR: *framework* para *multitenancy* baseado em JPA

Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação da Universidade do Oeste de Santa Catarina, campus de Xanxerê.

Professor Orientador: Cristiano Agosti
Co-orientador: André Luiz Forchesatto

Xanxerê
2012

VITOR ZACHI JUNIOR

ALLIGATOR: *framework* para *multitenancy* baseado em JPA

Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação da Universidade do Oeste de Santa Catarina, campus de Xanxerê.

BANCA EXAMINADORA

_____

_____

_____

RESUMO

Com a atual expansão da internet, surgiu recentemente o conceito de *cloud computing*, ou computação nas nuvens. Com o *cloud computing*, é possível ter disponível, em qualquer lugar com acesso à internet, seus aplicativos como se eles estivessem em uma máquina local. Pensando nesse sentido, as empresas de software podem explorar um ramo do cloud computing chamado SaaS(*Software as a Service*), ou Software como um Serviço. Nessa modalidade, o software não é instalado nos computadores do usuário, mas sim disponibilizado via internet. Sendo assim, o usuário está utilizando o software como um serviço. Agora, ao invés de ter uma instalação do software para cada cliente, teremos uma única instalação do sistema “na nuvem”, que deverá tratar os dados de cada usuário separadamente. Certamente, o modelo de dados deverá ser modificado para que cada registro armazenado no banco de dados possua um identificador do usuário a que pertence, implicando em mudanças na maioria das consultas SQL efetuadas. O framework Alligator pretende, de forma simples, interceptar a persistência e a consulta dos dados, procurando impactar o mínimo possível no código já existente, de modo a manter a integridade e a separação lógica dos dados de cada usuário, baseando-se na especificação JPA 2.

Palavras-chave: *multitenancy.* SQL. JPA. Internet. *Software as a Service.*

ABSTRACT

With the current expansion of the Internet, has recently emerged concept of cloud computing. With the cloud computing, it's possible to access your applications, in anywhere with Internet access, as if they were in a local computer. Thinking in this way, software companies can explore a model of cloud computing, called SaaS, or Software as a Service. In this model, the software is not installed in the user computer, but available in the Internet. Now, instead of having a software installation for each client, we will have a single installation of the system in the cloud, which will process the data of each user separately. The data model should be modified so that each record stored in the database has a user identifier that belongs, resulting in changes in most SQL queries. The Alligator framework intends to intercept the persistence and the queries, with minimally impact on existing code, in order to maintain the integrity and logical separation of data for each user, based on JPA 2 specification.

Keywords: multitenancy. SQL. JPA. Internet. software as a service.

LISTA DE FIGURAS

Figura 1: A pirâmide da computação em nuvem e quem interage com cada camada. 14
Figura 2: Representação gráfica do modelo *multitenancy* via *container*.....18
Figura 3: Representação gráfica do modelo *multitenancy* via compartilhamento total de software.....19
Figura 4: Exemplo de uso de anotações na JPA.....21
Figura 5: Ciclo de requisição de uma fábrica de EntityManager.....22
Figura 6: Representação de um comando para salvar um objeto e o SQL gerado.. .23
Figura 7: Comando de consulta em JPA e sua tradução para SQL.....24
Figura 8: Comando de consulta JPA com restrição e seu SQL gerado.....25
Figura 9: Diferença entre as tabelas após o ajuste para suporte à *multitenancy*.....28
Figura 10: Comando para busca de tipos de animais por nome no protótipo para único inquilino.....29
Figura 11: Comando da figura 10 ajustado para *multitenancy*, filtrando dados por empresa.....29
Figura 12: Comando para recuperar classes de entidade filhas de *AbstractTenantModel*.....31
Figura 13: Fluxo do comando *persist* executado pelo Alligator.....32
Figura 14: Fluxograma de funcionamento do Alligator para operações de consulta(*select*).....33
Figura 15: Esquema de classes de processamento do Alligator.....34
Figura 16: Modelo de relacionamento entre as classes da aplicação de teste do Alligator.....36
Figura 17: Camadas de uma aplicação com o framework Alligator.....37
Figura 18: Código da classe utilitária que fornece o *EntityManager* para o restante da aplicação.....38
Figura 19: Classe de entidade com herança de *AbstractTenantModel*.....38
Figura 20: Ajuste de chaves únicas.....38
Figura 21: Definição da classe que representa o inquilino através da interface *TenantOwner*.....39
Figura 22: Definição do *TenantOwner* ao iniciar a aplicação.....39

Figura 23: Configuração do módulo web do Alligator.....39
Figura 24: Instrução SQL com filtro por empresa injetado automaticamente.....40
Figura 25: Representação de dois inquilinos distintos acessando um sistema com o Alligator ao mesmo tempo.....40
Figura 26: Tabela de cadastro de atividades da aplicação com o Alligator.....41

LISTA DE TABELAS

Tabela 1: Comparativo de tempo gasto entre aplicação com e sem o Alligator.....42

LISTA DE ABREVIATURAS

IaaS	Infrastructure as a Service
JDBC	Java DataBase Connectivity
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSF	Java Server Faces
ORM	Object Relational Mapping
PaaS	Platform as a Service
SaaS	Software as a Service
SQL	Structured Query Language
XML	Extensible Markup Language

SUMÁRIO

1 **INTRODUÇÃO**.....10
1.1 OBJETIVOS.....11
1.1.1 **Objetivo geral**.....11
1.1.2 **Objetivos específicos**.....11
1.2 JUSTIFICATIVA.....11
2 **REFERENCIAL TEÓRICO**.....13
2.1 COMPUTAÇÃO EM NUVEM.....13
2.1.1 **Software como um Serviço(SaaS)**.....14
2.1.2 **Plataforma como um Serviço(PaaS)**.....15
2.1.3 **Infraestrutura como um Serviço(IaaS)**.....15
2.2 MULTITENANCY.....15
2.2.1 **Tenant Isolado**.....16
2.2.2 **Multitenancy via hardware compartilhado**.....17
2.2.3 **Multitenancy via container**.....17
2.2.4 **Multitenancy via compartilhamento total de software**.....18
2.3 TECNOLOGIAS UTILIZADAS.....19
2.3.1 **Mapeamento ORM**.....19
2.3.2 **Java Persistence API**.....20
2.3.3 **Anotações**.....20
2.3.4 **EntityManager e EntityManagerFactory**.....21
2.3.5 **Consultas (Query) em JPA**.....23
2.3.6 **Reflection**.....25
2.3.7 **Teoria de desenvolvimento de frameworks**.....25
3 **METODOLOGIA**.....27
4 **DESENVOLVIMENTO DO FRAMEWORK**.....28
5 **RESULTADOS**.....36
6 **CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS** 42
REFERÊNCIAS BIBLIOGRÁFICAS.....45

1 **INTRODUÇÃO**

A computação em nuvem ganhou força nos últimos anos, juntamente com o aumento da popularização da internet. Esse processo tem feito muitas empresas migrarem seus sistemas para esse novo formato, modificando assim a forma como estes são comercializados e a arquitetura utilizada para a concepção dos mesmos.

Porém existe um problema, onde “atualmente, a maioria dos softwares existentes foi desenhada para operar nos data centers das empresas, sujeitos a contratos específicos de licença de uso”(TAURION, 2010), necessitando assim que estas empresas disponibilizem toda a infraestrutura necessária para manter o sistema em funcionamento. Taurion(2010) também cita que as linguagens Java e .Net foram desenhadas para operar nesse modelo.

Com a chegada da computação em nuvem, esse processo pôde ser modificado. O software pode ser disponibilizado e acessado pelos seus usuários através da internet, livrando destes e da empresa que utiliza o software a responsabilidade em manter a infraestrutura requerida para deixar o sistema funcionando.

Nessa hora entra o Software como um Serviço, conhecido também por SaaS, onde o sistema pode ser oferecido como um serviço a ser utilizado por várias organizações. Essa arquitetura de software recebe o nome de *multitenancy,* o que significa muitos inquilinos.

Como na maioria das vezes o código-fonte da aplicação estava planejado para dar suporte somente a um cliente, boa parte da estrutura do sistema precisa ser remodelada para esta nova arquitetura e garantir que os clientes fiquem logicamente separados, mesmo compartilhando a mesma instância de aplicação e de banco de dados.

Grande parte da remodelagem consiste em incluir uma referência nos registros indicando a quem este pertence, e ajustar as consultas *Structured Query Language*(SQL) espalhadas na aplicação para se adaptarem à nova estrutura. Além de gerar um custo de refatoração de código relativamente grande, existe ainda a possibilidade de alguma consulta SQL ficar esquecida pela equipe de programadores e ser executada sem o devido filtro, podendo assim um usuário

visualizar os dados de outro usuário desconhecido, comprometendo a segurança e integridade da aplicação. Para manter essa integridade, o sistema deve ser capaz de saber qual é a empresa que está acessando o sistema, e efetuar operações somente sobre dados daquela empresa.

Neste trabalho pretende-se estudar a arquitetura dos sistemas para um cliente(inquilino) e do modelo *multitenancy*, construindo assim um *framework*, baseado em JPA, que auxilie o sistema na tarefa de saber a empresa que está acessando o sistema e efetuar operações sobre dados daquela empresa. Com o auxílio do *framework* construído, as consultas ao banco de dados passam por um filtro, o qual separa logicamente os dados de cada inquilino que utiliza o sistema, sem necessidade de efetuar grandes refatorações no código-fonte da aplicação já existente.

1.1 OBJETIVOS

1.1.1 **Objetivo geral**

Criar um framework para gerenciar a disponibilização dos dados em uma aplicação *multitenancy* nas nuvens utilizando *Java Persistence API*(JPA).

1.1.2 **Objetivos específicos**

- Estudar o padrão JPA de persistência.
- Demonstrar as diferenças entre o modelo de sistema para um cliente(inquilino) e o modelo *multitenancy*.
- Criar uma aplicação de teste para visualização das diferenças.
- Avaliar a viabilidade de uso do framework em aplicações *multitenancy*.

1.2 JUSTIFICATIVA

A internet se expandiu muito nos últimos tempos. Muitos foram os fatores que facilitaram essa expansão, que ainda está acontecendo. Alguns sistemas antes

desenvolvidos para *desktop* já estão sendo reescritos para serem executados na internet, tornando possível o acesso à aplicação a partir de qualquer local com acesso à internet.

Essa expansão da internet trouxe como consequência uma maior propagação do conceito de computação em nuvem (*cloud computing*), onde existe uma infraestrutura de servidores e de rede para disponibilizar serviços e recursos como se estivessem instalados em seu próprio computador, mas sem a necessidade de instalação de outros programas além de um navegador de internet.

A computação em nuvem pode trazer grandes vantagens para ambos os lados. Os clientes estarão pagando pela quantidade de recurso consumido, evitando uma depreciação desnecessária de seu patrimônio caso tivesse de manter toda a infraestrutura disponível para instalação e disponibilização do sistema.

Algumas empresas desenvolvedoras de software estão vendo na computação em nuvem uma forma de disponibilizar mais facilmente os softwares por elas desenvolvidos, pois os clientes não teriam a preocupação com instalação do sistema, basta acessá-lo pela internet. Sistemas desse tipo são chamados de *Software as a Service* ou somente SaaS, que traduzindo para o português significa Software como um Serviço. SaaS é um tipo de computação em nuvem que muda um pouco a forma de negócios entre uma empresa desenvolvedora de softwares e o seu cliente.

Normalmente, quando um cliente necessita de uma solução, a fábrica de software constrói uma solução e a vende, bem como serviços de suporte e outros mais. Já com SaaS, o valor que o cliente pagará poderá ser calculado de acordo com o uso de recursos do sistema, processamento e outros. Sendo assim, o cliente paga pelo serviço disponibilizado, e não pelo sistema em si.

Para aproveitar ao máximo a estrutura alocada no servidor virtual para o sistema, geralmente vários clientes diferentes acessam a mesma aplicação. Neste caso, deve-se ter como prioridade a segurança de dados, impedindo que um inquilino acesse dados de outro. O *framework* aqui implementado pretende diminuir a refatoração de código e modificação de consultas SQL, além de possibilitar uma segurança maior na execução de consultas sobre dados que são restritos somente ao seu proprietário.

2 **REFERENCIAL TEÓRICO**

2.1 COMPUTAÇÃO EM NUVEM

O termo “computação em nuvem” surgiu em 2006 em uma palestra de Eric Schmidt, da Google, sobre como sua empresa gerenciava seus *data centers* (local onde são concentrados os computadores e sistemas responsáveis pelo processamento de dados de uma empresa ou organização) (TAURION, 2009).

Segundo RUSCHEL *et al* (2010), “a computação em nuvem é a ideia de utilizarmos, em qualquer lugar e independente de plataforma, os mais variados tipos de aplicações através da internet com a mesma facilidade de tê-las instaladas em nossos próprios computadores”. Uma grande vantagem desse modelo é a disponibilidade de acesso aos dados em qualquer local do mundo que possua acesso à internet. Apesar do conceito ser um tanto quanto recente, o setor deve manter-se em crescimento, com previsão de movimentar cerca de US$ 16 bilhões no ano de 2013(CAETANO, 2010).

A computação em nuvem é dividida em serviços. Dentre eles merece destaque o Software como um Serviço (SaaS), Plataforma como um Serviço (PaaS) e Infraestrutura como um Serviço (IaaS). Estas três categorias estão dispostas em uma hierarquia, onde na base está a infraestrutura(IaaS) necessária e fornecida por servidores, na maioria das vezes virtualizado. Sobre a infraestrutura é instalada uma plataforma(PaaS) que fornece o sistema operacional e outros serviços necessários para o desenvolvimento. O software, por sua vez, é instalado sobre a plataforma disponível e interage com os usuários finais. A figura a seguir demonstra essa estrutura.

Figura 1: A pirâmide da computação em nuvem e quem interage com cada camada.
Fonte: o autor.

Estas categorias de computação em nuvem serão abordadas detalhadamente nas subseções a seguir.

2.1.1 **Software como um Serviço(SaaS)**

O SaaS, ou Software como um Serviço, é uma categoria de computação em nuvem onde, como o próprio nome diz, um software é fornecido como um serviço onde os clientes ou usuários podem usufruir das funcionalidades oferecidas. Um bom e simples exemplo de SaaS é o famoso *Google Docs*, que fornece serviço de edição e armazenamento de arquivos de texto, planilhas e apresentações(RUSHEL *et al*, 2010).

Para o usuário existe uma grande vantagem no SaaS: todo o custo com a infraestrutura necessária para o software, como a compra de servidores, espaço físico, energia elétrica e configuração da rede, é reduzido à zero. Esta despesa é diluída no pagamento pelo uso do serviço, tornando o seu valor muito menor se comparado à aquisição da infraestrutura, e ainda economizando com custos de depreciação de todo o equipamento adquirido.

Essa redução no custo possibilita que empresas de pequeno e médio porte também tenham acesso à essa tecnologia, antes inviável pelo alto custo de

manutenção. Com a abertura para esse novo mercado de pequenas e médias empresas, a indústria do software consegue atingir um setor que compõe grande parte do mercado atual.

2.1.2 **Plataforma como um Serviço(PaaS)**

O PaaS, ou Plataforma como um Serviço, oferece uma estrutura de alto nível de integração para implementar aplicações na nuvem. Também fornece um sistema operacional, linguagens de programação e ambientes de desenvolvimento para as aplicações, o que auxilia a implementação dos softwares, já que contém ferramentas de desenvolvimento e colaboração entre desenvolvedores(RUSHEL *et al*, 2010).

2.1.3 **Infraestrutura como um Serviço(IaaS)**

Para definir melhor o conceito de IaaS, recorremos à seguinte afirmação de Martinez(2010):

> O IaaS refere-se ao fornecimento da infraestrutura computacional (geralmente em ambientes virtualizados) como um serviço. Em vez de o cliente comprar servidores para uma determinada aplicação, ele contrata um serviço dentro de um *datacenter* proporcional aos seus requisitos de infraestrutura e tem acesso completo à plataforma e ao software. Esse tipo de serviço é cobrado de acordo com a utilização ou pela reserva de recursos contratados.

Além disso, o serviço deve possibilitar alteração das configurações de processamento e memória, além de espaço em disco, para garantir a escalabilidade da aplicação e evitar que horários de pico tornem instáveis os serviços fornecidos sobre a infraestrutura. Toda essa disponibilidade e escalabilidade de recursos possibilitou o surgimento de uma nova arquitetura, mais voltada ao aproveitamento destes recursos, chamada de *multitenancy*.

2.2 *MULTITENANCY*

Multitenancy, ou simplesmente multi inquilino, se refere basicamente a uma

arquitetura de software onde tem-se uma única instância de um sistema rodando em um servidor, e atendendo a múltiplos clientes(chamados de inquilinos) diferentes, cada um com suas informações e o acesso pelo usuário se dá de forma transparente, dando a entender que a aplicação está servindo somente à ele. Do ponto de vista de quem fornece o serviço, esta arquitetura representa uma economia no custo com recursos de hardware e simplifica a criação do ambiente inicial para os novos usuários (FABBRO, 2011).

Já do ponto de vista do consumidor, ou usuário, a simplicidade e agilidade garante que o serviço esteja disponível logo após a sua adesão. Ou seja, não há interferência sobre o processo de criação de um novo ambiente à cada novo cliente que adere ao sistema. Ele deve ter a capacidade de criar e configurar automaticamente o ambiente necessário para um novo cliente.

Existem basicamente quatro modelos de *multitenancy*, sendo eles o modelo de *tenant*(inquilino) isolado, *multitenancy* via hardware compartilhado, *multitenancy* via *container* e *multitenancy* via compartilhamento total. Alguns autores desconsideram como modelo de *multitenancy* o modelo de *tenant* isolado, por ele não prover compartilhamento de instâncias de software ou hardware. Todavia, este modelo foi o precursor da computação em nuvem (TAURION, 2010), e consequentemente dos modelos atuais de *multitenancy*, tendo também a sua importância na evolução desta tecnologia.

2.2.1 ***Tenant* Isolado**

Neste modelo não existe compartilhamento de recursos entre os inquilinos. Cada um deles tem reservada uma instância da aplicação e seu banco de dados. Semelhante aos modelos comuns de hospedagem existentes atualmente, este modelo não é *multitenancy*, diferenciando-se dos modelos de hospedagem tradicionais apenas por disponibilizar a aplicação para todos os inquilinos a partir do mesmo *datacenter*, mas cada inquilino possui seu conjunto de recursos. Segundo Taurion (2009), “este modelo carece de agilidade e elasticidade, pois cada novo inquilino requer provisionamento de mais uma instância de hardware e software”.

2.2.2 *Multitenancy* via hardware compartilhado

Este modelo prevê que cada inquilino tenha o seu conjunto de tecnologias, mas rodando sobre o mesmo hardware via virtualização. É parecido com o modelo anterior, mas diferencia-se por poder alocar e liberar recursos de hardware dinamicamente.

Geralmente este modelo é utilizado por empresas provedoras de software, que neste caso são os inquilinos. Cada inquilino tem uma ou mais máquinas virtuais alocadas sobre o hardware disponibilizado pelo *datacenter*, e os recursos de memória, processamento e espaço em disco podem ser ajustados dinamicamente e de forma diferente para cada máquina virtual, conforme a necessidade de cada inquilino(TAURION, 2009).

2.2.3 *Multitenancy* via container

Este é um modelo geralmente utilizado e oferecido por provedores de software aos seus clientes, que agora são considerados inquilinos. Aqui os inquilinos compartilham o mesmo hardware(no caso uma máquina virtual) e a mesma instância da aplicação, porém cada um tem a sua instância de banco de dados. Este modelo garante a separação dos dados entre os inquilinos, mas exige que a aplicação mantenha o controle o acesso aos dados, direcionando o fluxo de informações de cada inquilino para o seu devido banco de dados. Este esquema está representado graficamente na figura 2.

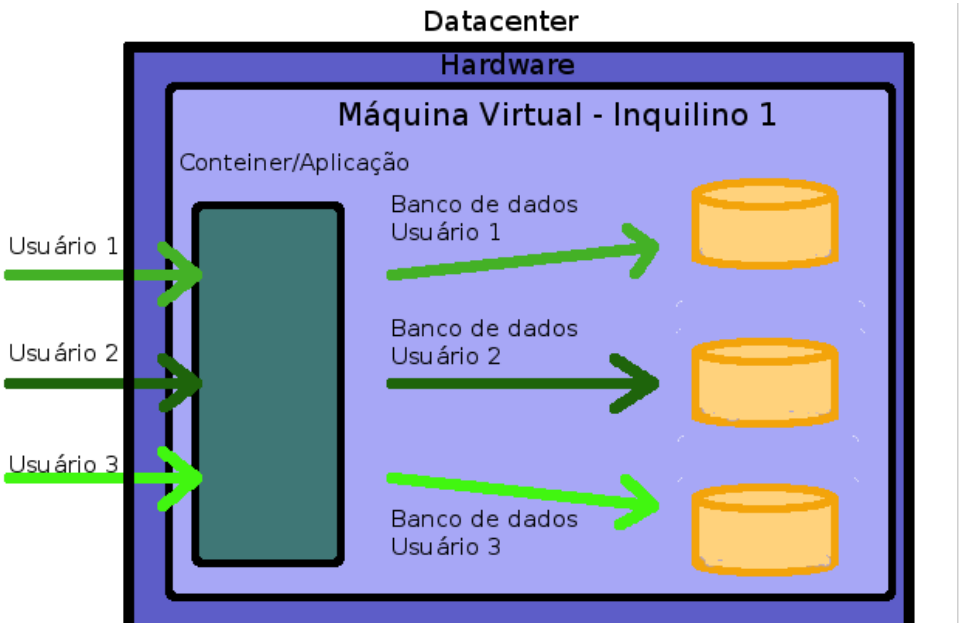

Figura 2: Representação gráfica do modelo *multitenancy* via *container*.
Fonte: o autor.

Uma grande vantagem é que, caso o provedor do software efetuar uma atualização no software, automaticamente todos os inquilinos irão usufruir da mesma sem a necessidade de repetir o processo de atualização em cada inquilino, pois todos estão compartilhando a mesma instância da aplicação. O mesmo já não pode ser dito quando se refere à mudanças na estrutura de dados, pois como cada inquilino possui uma instância diferente de banco de dados, a atualização deverá ser repetida para cada instância banco de dados existente.

2.2.4 ***Multitenancy* via compartilhamento total de software**

É baseado no modelo anterior, porém o banco de dados também é compartilhado. Neste modelo, a aplicação precisa manter os inquilinos logicamente separados, garantindo a execução de filtros sobre os dados que são requisitados pelos inquilinos, evitando que o princípio da confidencialidade dos dados seja quebrado. Ou seja, “deve-se proteger a informação contra leitura e/ou cópia por alguém que não tenha sido explicitamente autorizado pelo proprietário daquela informação”(SOUSA, PUTTINI, 2010). Este modelo está representado graficamente na figura 3.

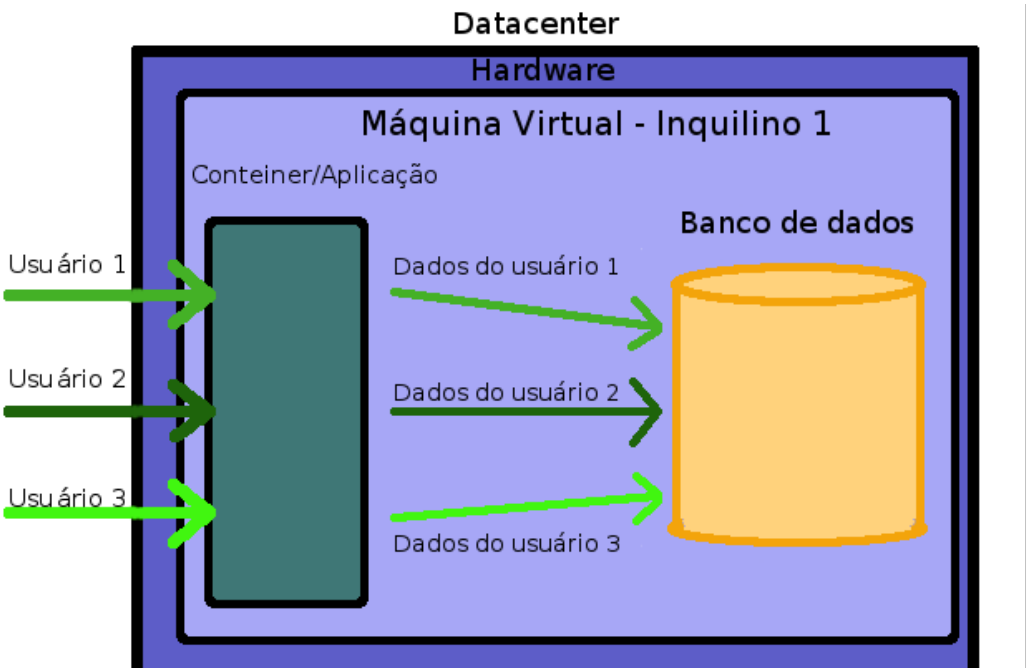

Figura 3: Representação gráfica do modelo *multitenancy* via compartilhamento total de software.
Fonte: o autor.

Este modelo e o *multitenancy* via *container* são modelos mais avançados de computação em nuvem, e provavelmente serão os mais utilizados. Por esse motivo, ele servirá de base para o *framework* desenvolvido neste trabalho.

2.3 TECNOLOGIAS UTILIZADAS

2.3.1 **Mapeamento ORM**

Atualmente os bancos de dados relacionais são os tipos de banco de dados mais comuns utilizados pelas empresas se comparados com outros formatos, por exemplo o banco de dados orientado a objetos(HEMRAJANI, 2009). Já no lado das linguagens, a programação orientada a objetos prevalece como sendo a mais utilizada. Um Mapeamento Objeto Relacional(ORM, do inglês *Object Relational Mapping*) é a persistência de objetos de forma automatizada e transparente de dentro de um aplicativo para as tabelas de um banco de dados relacional (BAUER, KING, 2005). Esse processo é feito através da utilização de metadados, hoje descrito em anotações sobre classes e atributos, que descrevem o mapeamento

entre os objetos e seus atributos com o banco de dados, transformando objetos carregados com valores em linhas de registro em tabelas e vice-versa.

Existem vários *frameworks* de ORM para a linguagem Java, sendo que o mais conhecido é o Hibernate. Pelo fato de que cada *frameworks* pode implementar os processos de persistência de forma diferente, buscou-se padronizar e incorporar essa padronização na linguagem. Surgia dessa forma a *Java Persistence API*, ou simplesmente JPA.

2.3.2 **Java Persistence API**

Uma das grandes funcionalidades de um sistema é a sua interação com banco de dados, onde pode-se armazenar e recuperar informações. Esse processo de armazenamento de dados é chamado de persistência.

O JPA é uma especificação criada para permitir a persistência objeto relacional, mais conhecida como ORM ou *Object-Relational Mapping*, com bancos de dados relacionais. Em vez de operar diretamente com dados de tabelas, o sistema trabalha com objetos carregados com valores (GOMES, 2008, p. 11). Ao efetuar operações de persistência, como leitura e escrita no banco de dados, o sistema exige que que se trabalhe com os objetos, e o *framework* que implementa a especificação JPA converte as instruções orientadas a objeto na linguagem do banco de dados, ou seja, em *Structured Query Language* (Linguagem de Consulta Estruturada ou simplesmente SQL). Com a vinda do padrão JPA, foi possível efetuar o mapeamento através de anotações simples nas classes, eliminando as configurações maçantes antes efetuadas via arquivos XML.

Vale lembrar que a JPA por si só não efetua persistência de dados, pois ela é um conjunto de interfaces e classes utilitárias responsáveis por padronizar o processo de interação com bancos de dados em uma aplicação. Para que isso ocorra, é necessário que haja no projeto uma implementação, sendo o Hibernate, mantido pela JBoss, e o TopLink, mantido pela Oracle, as implementações JPA mais conhecidas na comunidade Java.

2.3.3 **Anotações**

As anotações Java são um tipo especialmente definido com objetivo de simplificar algumas tarefas, através de uma anotação colocada acima ou à frente de algum elemento, que pode ser uma classe, atributo ou método, evitando ou diminuindo a quantidade de arquivos externos de configuração. Quando algum elemento é anotado, o compilador lê a informação e pode retê-la durante a execução do programa, conforme definido na anotação. Caso o valor ficar retido, o mesmo pode ser lido em tempo de execução através de reflexão (GONÇALVES, 2008, p. 102).

Na JPA, as anotações servem para definir entidades, que são as classes que serão mapeadas para o banco de dados e poderão ser persistidas, bem como as suas configurações. Para isso são utilizados anotações como *@Entity*, a qual define uma classe como entidade, e *@Table*, que comporta configurações como o nome da tabela no banco de dados e chaves únicas compostas. A figura 4 exemplifica o uso de anotações sobre uma classe Java.

```
@Entity
@Table(name = "CIDADE")
public class Cidade implements Serializable {
```

Figura 4: Exemplo de uso de anotações na JPA.
Fonte: o autor.

São usadas também anotações para representar relacionamentos e cardinalidade, como *@OneToMany*, *@ManyToOne*, *@ManyToMany* e *@OneToOne*, sendo que para estas anotações é possível definir operações em cascata, e a propriedade de mapeamento bidirecional, caso exista. Graças às anotações, foi possível abandonar os arquivos de mapeamento em XML por algo mais consistente e que garante maior entendimento do código.

2.3.4 ***EntityManager* e *EntityManagerFactory***

Para poder utilizar o mecanismo de persistência, precisamos de uma instância de *EntityManager*, ou gerenciador de entidades. O *EntityManager* é uma interface da especificação JPA e é responsável por disponibilizar métodos que

permitem a interação da aplicação com o banco de dados. Para conseguirmos uma instância de *EntityManager*, precisamos de uma fábrica para criá-lo. A imagem a seguir demonstra o ciclo para requisitarmos uma fábrica que irá criar essa instância, gerada com auxílio da classe *Persistence.java,*disponibilizada pela JPA. A figura 5 exemplifica o ciclo de requisição de uma fábrica de *EntityManager* (*EntityManagerFactory*).

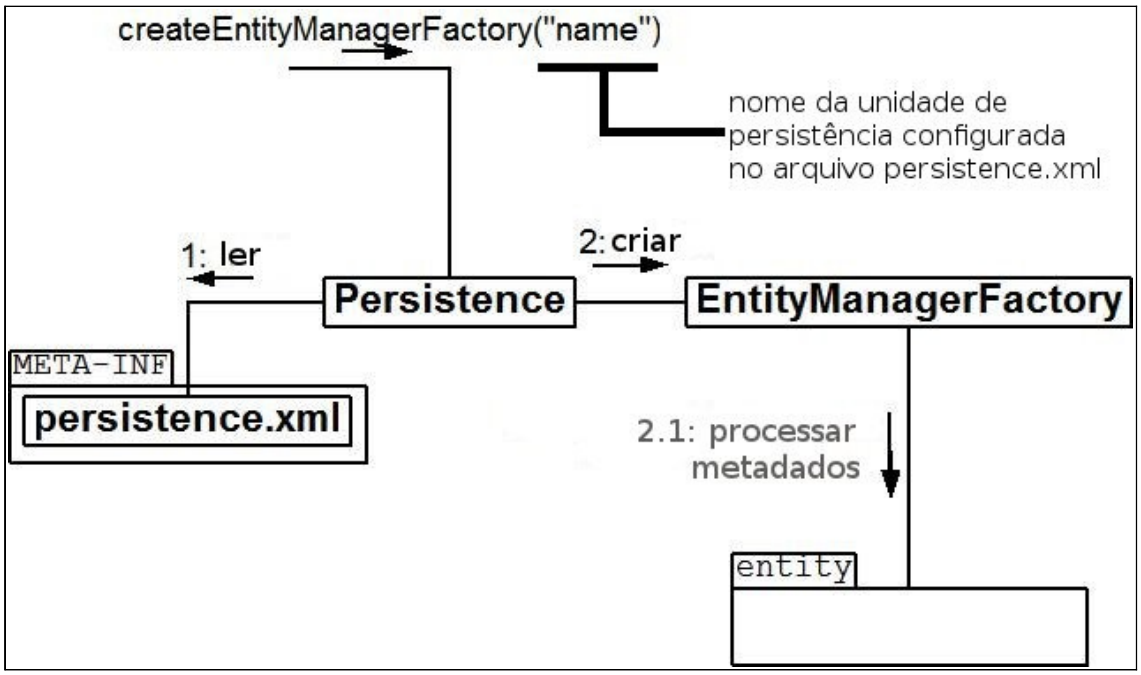

Figura 5: Ciclo de requisição de uma fábrica de *EntityManager*.
Fonte: traduzido de JPA Tutorial(2011).

O ciclo consiste em invocar o método *createEntityManagerFactory* da classe *Persistence.java*. Este método irá ler as configurações contidas em um arquivo chamado *persistence.xml* contido na pasta *META-INF*, que por sua vez deve estar localizada na pasta raiz dos pacotes de código-fonte, sendo isso uma condição obrigatória para o seu funcionamento. No arquivo *persistence.xml* estão as configurações necessárias para conexão com o banco de dados, como usuário, senha e a URL de conexão, além de conter as propriedades sobre o tipo de *driver* JDBC[1](*Java Database Conectivity*) a ser utilizado e qual é a classe que fornecerá a

1 JDBC(Java Database Connectivity) é um conjunto de classes e interfaces que fornecem um padrão para os desenvolvedores de bancos de dados. Com isso, evita-se a necessidade de ter um sistema escrito para banco Oracle, outro para MySql, já que é tudo padronizado pelo JDBC. (traduzido e adaptado de *What is JDBC?*)

implementação da JPA. Neste arquivo também é especificado um nome para a unidade de persistência (*persistenceUnit*), e este nome é necessário ao invocar o método *createEntityManagerFactory.*

Após efetuada a leitura do arquivo de configurações, é iniciado o processo de criação da fábrica. Este processo pode criar as tabelas do banco de dados automaticamente, através do processamento dos metadados(anotações) contidos nas entidades, conforme as propriedades configuradas no arquivo *persistence.xml.* O próximo passo é retornar uma instância de *EntityManagerFactory*, responsável por gerar instâncias de *EntityManager*.

Uma instância de *EntityManager* permite operações de criação(*persist*), alteração(*merge*), remoção(*remove*) e consulta(*createQuery*), além de outros métodos não muito relevantes neste trabalho.

Todo o processo de persistência é orientado a objeto, portanto, para os métodos de criação, alteração e remoção, é necessário um objeto (a entidade) como parâmetro. A imagem a seguir ilustra uma situação parecida, onde o método *persist* é chamado (em A) e o comando é traduzido para a linguagem SQL(em B).

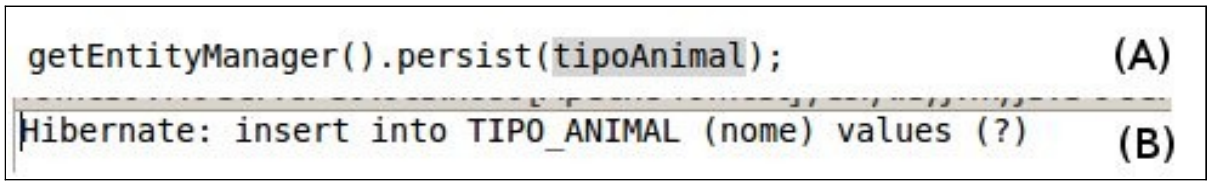

Figura 6: Representação de um comando para salvar um objeto e o SQL gerado.
Fonte: o autor.

Em alguns casos, é necessário selecionar somente um conjunto de dados e restringindo outros. Para esses casos são criadas consultas, mais chamadas de *queries*.

2.3.5 **Consultas (*Query*) em JPA**

Assim como os outros métodos, as consultas também são orientadas a objeto, utilizando-se a *Java Persistence Query Language* (JPQL), sendo posteriormente traduzidas para a linguagem do banco de dados. Uma consulta básica é formada por uma instrução “*select alias from Entidade alias*”, sendo que a parte “*select alias*” pode ser omitida. Em termos básicos, “*from Entidade*” indica de

qual entidade os dados serão selecionados. Geralmente, é necessário definir um pseudônimo ou apelido, mais chamado de *alias*, para a entidade presente na instrução, a fim de referenciá-la novamente na consulta, como por exemplo em uma consulta com restrição (*where*).

Para utilizarmos esse recurso na JPA, é necessário requisitar a criação de uma *Query* ao *EntityManager*, chamando *entityManager.createQuery,* passando a instrução JPQL por parâmetro. Este procedimento retorna um objeto do tipo *Query*, que nos dá acesso à métodos como por exemplo *getSingleResult* e *getResultList*, os quais retornam um único objeto ou uma coleção de objetos, respectivamente. A imagem 7 contém um exemplo de comando de consulta em JPA(em A) e a sua tradução para SQL(em B).

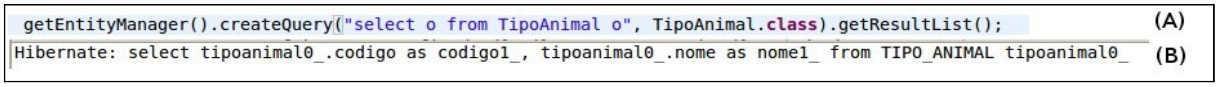

Figura 7: Comando de consulta em JPA e sua tradução para SQL.
Fonte: o autor.

As consultas com restrição são úteis quando queremos omitir certos resultados, selecionando somente aqueles que satisfazem as restrições impostas na instrução. Uma restrição é imposta através de uma cláusula "*where 'condição'*", onde a condição deve retornar um valor verdadeiro ou falso. Supondo que a classe *TipoAnimal.java* contenha um atributo chamado "nome", a instrução "*select o from TipoAnimal o where o.nome like 'cachorro'*" retorna todos os tipos de animais em que o nome é "cachorro".

Geralmente, as restrições são dinâmicas. No caso do exemplo anterior, provavelmente a restrição da consulta será um valor desconhecido, que o usuário irá definir ao realizar a pesquisa. Para estes casos, o objeto *Query* também permite a inserção de parâmetros na instrução, para formação deste tipo de consulta, através do método *setParameter*. Na instrução JPQL, substitui-se o parâmetro estático da instrução por um nome precedido por ":"(dois pontos), ficando "*select o from TipoAnimal o where o.nome like :nomePesquisa*". A instrução JPQL(A) e a tradução para SQL(B) produzida podem ser visualizados na imagem a seguir.

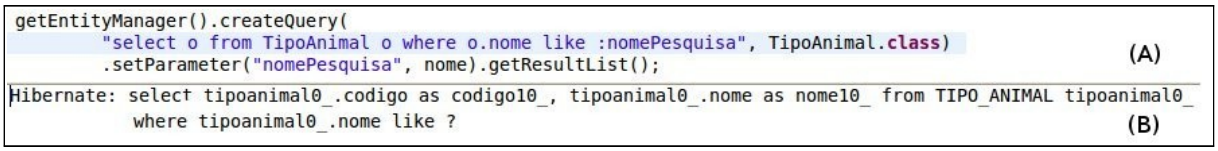

Figura 8: Comando de consulta JPA com restrição e seu SQL gerado.
Fonte: o autor.

É possível também inserir mais de um parâmetro, caso necessário. Pode-se também passar um objeto como parâmetro, desde que este seja compatível com a restrição, já que as consultas são orientadas a objeto.

2.3.6 ***Reflection***

A reflexão permite ao programa saber informações sobre as classes em tempo de execução, possibilitando a obtenção de informações sobre métodos e acessando-os dinamicamente durante a execução do programa (TRAIL: The Reflection API). A reflexão é um recurso muito utilizado em *frameworks* de Injeção de Dependência e de Mapeamento Objeto Relacional, justamente por prover esses recursos.

É utilizando a reflexão que também se torna possível o acesso às anotações e seus valores, possibilitando que seja efetuado algumas operações conforme o conteúdo destas anotações. Também é possível recuperar e definir valores aos atributos através da reflexão, e esse processo é chamado de introspecção.

Embora a reflexão seja muito útil, o tempo para ela ser executada é bem maior do que se houvesse o acesso direto aos objetos. Portanto, deve-se ter muito cuidado e utilizá-la somente quando necessário.

2.3.7 **Teoria de desenvolvimento de *frameworks***

Um *framework* é “um conjunto de classes que constitui um projeto abstrato para a solução de uma família de problemas”(Johnson e Foote,1988, *apud* Torres da Silva, 2010). Existem várias definições de *framework*, porém esta definição é a que mais se encaixa no propósito deste trabalho.

Conforme citado no parágrafo anterior, um *framework* basicamente resolve uma família específica de problemas, através de classes (concretas ou abstratas) e

interfaces, colaborando entre si a fim de atender o objetivo de resolver o problema proposto. Devem ser flexíveis e extensíveis, bem documentados e pouco intrusivos, possibilitando assim a construção de aplicações sem muito esforço e sem muita intervenção no código já existente.

Desenvolver um *framework* não é uma tarefa fácil. É preciso abstrair corretamente a lógica do problema, adaptando-se às diferentes maneiras de programar de cada desenvolvedor, além de possibilitar a extensão e customização de algumas partes, a fim de permitir o seu uso em alguns casos específicos que demandam um tratamento diferenciado do problema.

Os desafios de um *framework* não param por aí. Ele também é um software, e é preciso mantê-lo, atualizando-o constantemente com correções de *bugs*, além de poder contemplar novas funcionalidades e manter a documentação sempre atualizada(SOMMERVILLE, 2007). Todo esse processo necessita de tempo, e tem um custo, o que na maioria das vezes ocasiona a descontinuação do desenvolvimento, podendo causar prejuízos aos usuários do *framework*.

3 **METODOLOGIA**

Para início dos trabalhos, primeiramente foi realizado uma pesquisa bibliográfica, a fim de buscar informações sobre o modelo de sistemas *multitenancy* e verificar as diferenças existentes entre este e os modelos atuais.

Com intuito de obter maior clareza sobre estas diferenças detectadas, elaborou-se um estudo de caso, baseado em um protótipo de sistema para empresas de *petshop*, construído no modelo para comportar apenas um inquilino.

Sobre este protótipo, foram efetuadas modificações para adaptar a modelagem inicial ao modelo *multitenancy*. Em uma modificação, todo o código foi refatorado, adicionando-se os relacionamentos, reescrevendo os métodos e *queries* de consulta.

Baseado nas adaptações feitas na segunda versão, detectou-se os pontos onde o *framework* deveria atuar e como poderia atuar. A modelagem do mesmo foi pensada de modo que a sua utilização não impactasse muito no código já existente na aplicação, ou seja, não intrusivo.

A terceira versão do protótipo da aplicação sofreu adaptações para comportar o *framework* resultante deste trabalho, a fim de verificar o correto funcionamento do mesmo. Para isso, tomou-se como base o protótipo da primeira versão, tendo assim uma maior proximidade com a realidade das aplicações a que se destina o uso do *framework*.

Em todos os protótipos de aplicação, foi utilizado a linguagem Java, juntamente com a tecnologia *Java Server Faces* (JSF), banco de dados MySql e Apache Tomcat como servidor de aplicações.

4 DESENVOLVIMENTO DO *FRAMEWORK*

Para um melhor entendimento de como o *framework* irá atuar, tomemos como exemplo uma classe do protótipo chamada *TipoAnimal.java*, que representa os tipos de animais que cada empresa atende. Sendo assim, uma empresa não deve ter acesso aos tipos de animais de outras empresas.

A partir do protótipo da aplicação para um inquilino, foram feitas adaptações para adicionar esta restrição, como a inclusão de um atributo na classe para representar a empresa a que ele pertence. Esta modificação refletiu nas tabelas, onde este novo campo também foi adicionado à estrutura. A imagem 9 mostra a diferença entre as tabelas, onde está um fragmento da tabela antes(A) e depois(B) da alteração.

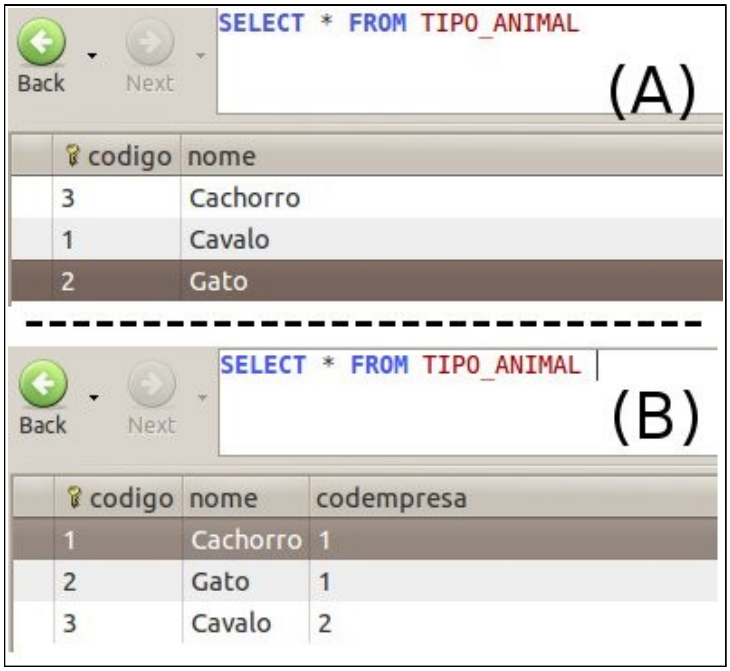

Figura 9: Diferença entre as tabelas após o ajuste para suporte à multitenancy.
Fonte: o autor.

Alguns métodos também sofreram modificações, onde os comandos de consulta JPQL foram reescritos para adicionar a restrição por empresa. A imagem a seguir mostra um comando de consulta por nome, da forma como desenvolvido no protótipo para único inquilino.

```
getEntityManager().createQuery(
        "select o from TipoAnimal o where o.nome like :nome",
        TipoAnimal.class).setParameter("nome", nome).getResultList();
```

Figura 10: Comando para busca de tipos de animais por nome no protótipo para único inquilino.
Fonte: o autor.

Já na figura 11 o comando contido na figura 10 foi modificado para filtrar efetuar a busca somente sobre os tipos de animais da empresa passada por parâmetro, através da adição de mais uma restrição(*"and o.empresa = :empresa"*) na cláusula *"where"*.

```
getEntityManager().createQuery(
        "select o from TipoAnimal o where o.nome like :nome and o.empresa=:empresa",
        TipoAnimal.class).setParameter("nome", nome).setParameter("empresa", empresa).getResultList();
```

Figura 11: Comando da figura 10 ajustado para multitenancy, filtrando dados por empresa.
Fonte: o autor.

Sem o *framework* desenvolvido neste trabalho, essa modificação precisaria ser efetuada em todas as operações de consulta envolvendo classes que possuem esta mesma característica (que devem sofrer filtragem por empresa) da classe *TipoAnimal.java*. Em uma aplicação pequena, com poucas classes, a quantidade de retrabalho não é tão grande, mas, e se o sistema tivesse 50 classes? E se algum programador esquecer de adicionar a restrição por empresa?

O *framework* teve seu desenvolvimento guiado sobre esses problemas e suas possíveis soluções, de uma forma que não fosse necessário percorrer por todas os comandos de persistência e de consulta aos dados para adicionar a restrição por inquilino, e que, não houvesse a possibilidade de executar alguma consulta sobre dados que não podem ser acessados por outros inquilinos sem que essa restrição fosse aplicada.

Para o desenvolvimento do *framework*, chamado de Alligator, primeiramente foi estudado o modelo de relacionamento entre tabelas de banco de dados de uma aplicação normal e como o modelo poderia ser adaptado para suportar uma aplicação *multitenancy*. Neste estudo definiu-se que cada registro do banco de dados iria possuir um código identificador do proprietário do registro, simulando um relacionamento entre este registro e seu proprietário. Este código identificador

estaria presente somente nos registros das tabelas onde existem dados que devem ser separados ou filtrados por inquilino, já que em uma base de dados podem existir tabelas onde faz sentido compartilhar os dados contidos nela, como por exemplo uma tabela contendo as unidades federativas do país. Ela será a mesma tabela para todos os inquilinos e não faria muito sentido replicar os registros para cada um deles.

Tendo definido esta modificação, iniciou-se um estudo sobre a maneira de aplicar as alterações nas entidades da aplicação Java, seguindo o padrão orientado a objetos e utilizando-se dos recursos de persistência presentes na JPA. Então, para definir o código identificador em cada registro, foi criado uma classe abstrata(*AbstractTenantModel*) contendo este atributo, e as classes de entidade que necessitam de separação entre os inquilinos devem herdar desta classe abstrata. Ao efetuar operações com o banco de dados, esse atributo terá seu valor atribuído automaticamente.

Para mapear corretamente esta classe na JPA, foi adicionado a anotação *@MappedSuperclass*. Esta anotação define que os atributos contidos nela serão replicados nas entidades-filho, não sendo criado uma tabela específica para esta classe(Annotation Type MappedSuperclass, 2007).

Após este passo, foi estudado uma forma de interceptar as operações de persistência (neste caso o método *persist*) efetuadas com estes objetos. Para esta tarefa, optou-se por criar uma espécie de cópia(padrão *proxy*[2]) da implementação do *EntityManager*, efetuando-se alterações nos métodos necessários e mantendo as mesmas funcionalidades da implementação nos métodos restantes. Antes de efetuar a interceptação, é preciso saber se o objeto a ser persistido é um objeto que possui um proprietário (herda de *AbstractTenantModel*) e, caso sim, o código de seu proprietário é injetado, antes de prosseguir com o comando de persistência.

Para simplificar e agilizar a busca de informações importantes e necessárias no processo de interceptação das operações, foi criado uma classe contendo essas informações, inicializadas uma só vez durante a aplicação. Nesta classe de contexto estão armazenadas as *NamedQueries* presentes nas classes de entidade, além de

2 O padrão *proxy* é um padrão de projeto com propósito de prover uma maneira de controlar o acesso à um objeto a partir de outro objeto(SOUZA, 2006). No caso deste trabalho, o acesso aos métodos do *EntityManager* "original" ocorre através de outro objeto, responsável por analisar a necessidade de filtro ou não.

conter quais são as classes que necessitam de interceptação, encontradas com o auxílio de um outro *framework* chamado Trugger, que foi incorporado ao Alligator. O *Trugger* é um framework desenvolvido por Marcelo Varella e distribuído sob Licença Apache, e tem por objetivo tornar o código mais claro e objetivo, através de uma sintaxe autoexplicativa. Para este projeto, o Trugger foi utilizado para percorrer e encontrar classes de entidade e que herdem de *AbstractTenantModel*, através de recursos de reflexão. O comando para recuperar estas classes está demonstrado na figura a seguir.

```
entities.addAll(ClassScan.findAll().annotatedWith(Entity.class).assignableTo(AbstractTenantModel.class)
        .recursively().in(namePackage));
```

Figura 12: Comando para recuperar classes de entidade filhas de AbstractTenantModel.
Fonte: o autor.

Este comando retorna uma coleção com objetos do tipo *java.lang.Class*, que satisfazem à restrição aplicada, ou seja, devem estar anotadas com *@Entity*, ser filhas de *AbstractTenantModel* e pertencer ao pacote contido na variável *namePackage*.

O contexto também é responsável por manter uma instância alterada da implementação do *EntityManager* utilizado na aplicação, funcionando basicamente como uma espécie de *proxy*. O contexto necessita receber em sua criação o nome do pacote onde se encontram as classes de entidade para rastrear as classes filhas de *AbstractTenantModel*, juntamente com o nome da unidade de persistência definido no arquivo de configurações da JPA(“persistence.xml”). Nesta função, será criado uma instância do *EntityManager* utilizando os recursos da JPA, e posteriormente disponibilizado a sua cópia para uso no código da aplicação. Caso já exista uma instância do *EntityManager* criada, pode-se utilizar outra função de inicialização, onde esta instância pode ser passada como parâmetro para a criação do seu *proxy*.

Essa instância alterada do *EntityManager* efetua interceptações nas operações de inserção(*persist*), inclusive em atributos do objeto a ser persistido, caso estes tenham a propriedade de inserção e/ou alteração em cascata. Esta introspecção é efetuada por um conjunto de classes específicas para este fim, e só é efetuada no primeiro nível de profundidade. Ou seja, o processo de introspecção só

é executado em atributos do objeto que será persistido, não atingindo os campos dos atributos existentes no objeto a ser persistido. Este fluxo é ilustrado na imagem 13.

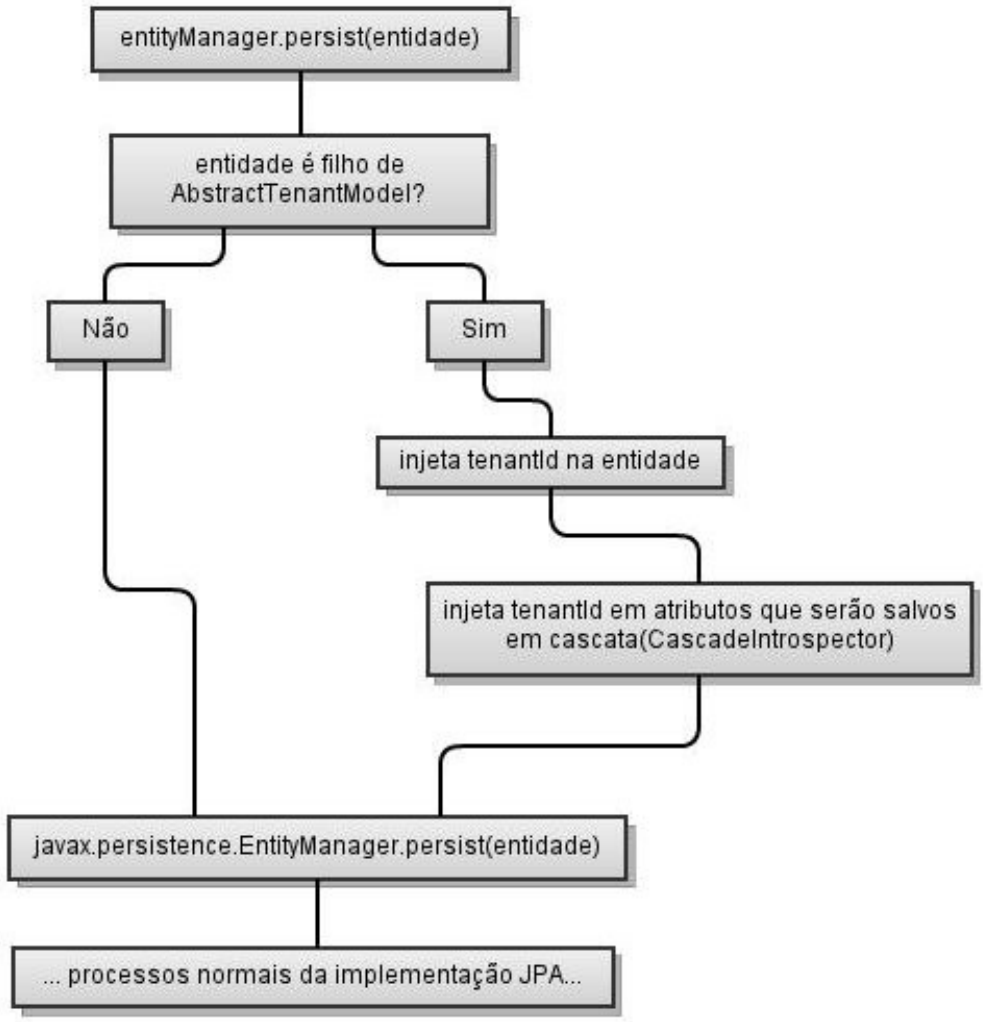

Figura 13: Fluxo do comando persist executado pelo Alligator.
Fonte: o autor.

Após ajustar a interceptação ao efetuar operações de inserção, foi necessário interceptar a leitura e recuperação de dados(*select*). Esta fase pode ser considerada como a mais difícil, pois envolve análise de SQL em busca de padrões nas instruções contidas no comando, a fim de encontrar o local correto para injetar o parâmetro de filtro(condição "*where*") quando necessário.

A seguir, será descrito com mais detalhes o fluxo de processamento do *framework* para os comandos de consulta, ilustrado na imagem a seguir.

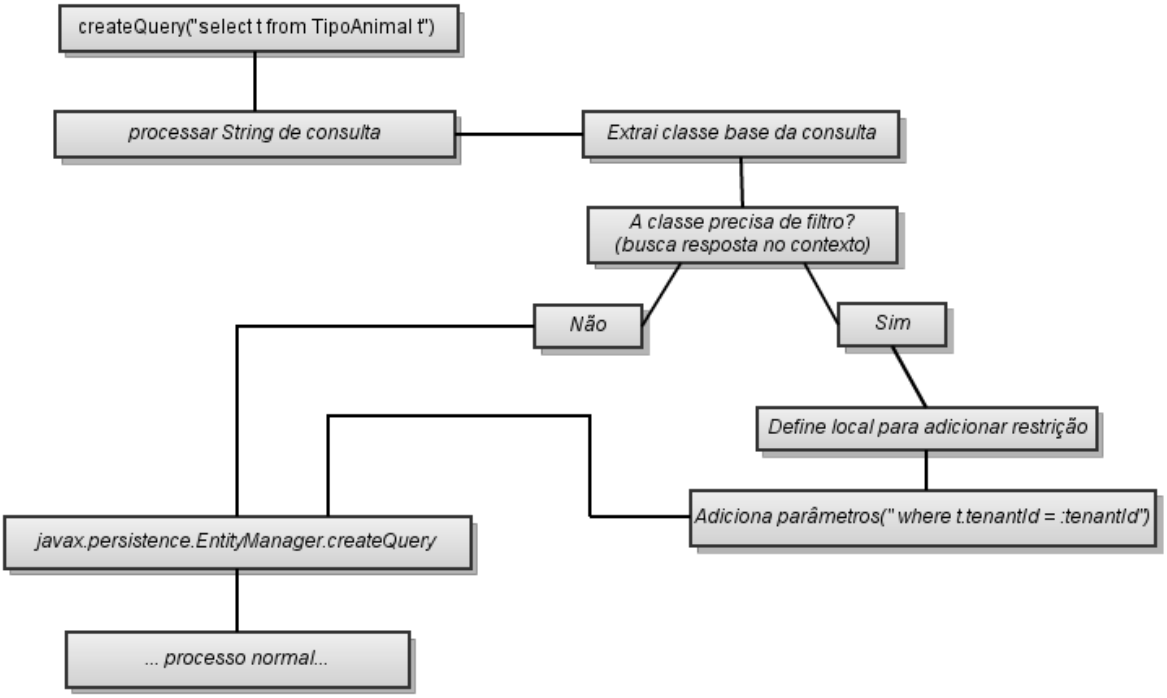

Figura 14: Fluxograma de funcionamento do Alligator para operações de consulta(select).
Fonte: o autor.

Para essa finalidade, foi criado uma classe para o processamento de *queries*, a qual analisa a instrução, definindo a necessidade de injeção da restrição e inserindo-a no local correto. As classes são identificadas seguindo o padrão da instrução "*from Entidade alias*", sendo obrigatório o uso deste padrão nas instruções. Encontrado as entidades e seus devidos pseudônimos, é efetuado uma busca no contexto, a fim de verificar se alguma classe necessita o filtro e, caso necessário, este é adicionado na instrução de consulta. Após esse processamento, a instrução é devolvida para o *EntityManager* continuar com a operação normal de execução da mesma.

O suporte às funcionalidades descritas anteriormente são dadas pelo esquemas de classes ilustradas a seguir.

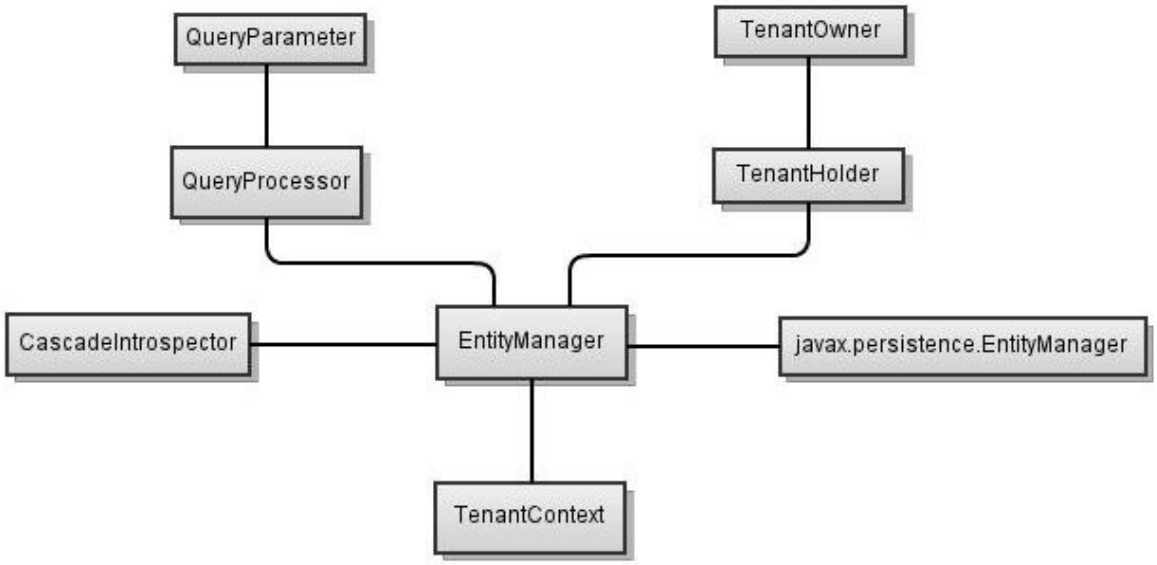

Figura 15: Esquema de classes de processamento do Alligator.
Fonte: o autor.

Cada classe realiza possui certas responsabilidades, sendo:

- *TenantContext.java*: classe abstrata responsável por armazenar informações sobre classes em que deve ser aplicado o filtro, além de armazenar uma coleção de *NamedQuery* encontradas na aplicação. Contém métodos de inicialização e de acesso ao *EntityManager*.
- *EntityManager.java*: classe concreta que armazena a implementação do *EntityManager* do JPA, possibilitando o tratamento de certas operações antes de chamar as operações normais da implementação.
- *CascadeIntrospector.java*: classe concreta responsável por injetar o código do inquilino nos campos que serão salvos em cascata.
- *QueryProcessor.java*: interface padrão para um processador de consultas SQL. A implementação padrão analisa o comando da consulta e verifica se há necessidade de injetar parâmetros para filtragem, representados por uma implementação de *QueryParameter.java.*
- *QueryParameter.java*: interface padrão para um parâmetro a ser adicionado no comando de consulta. A implementação padrão fornece o parâmetro para filtro pelo campo "*tenantId*".
- *TenantOwner.java*: interface que deve ser implementada pela classe

que servirá de base para o filtro. Representa o proprietário dos registros.

- *TenantHolder.java*: classe concreta responsável por informar ao Alligator qual é o *TenantOwner* responsável pela operação corrente, possibilitando que o filtro seja executado baseado nessa informação.

Para teste do *framework* foi desenvolvido uma pequena aplicação web, simulando um *petshop*. No modo normal, todos os usuários podem ver todos os dados presentes na aplicação, mostrando com isso o que acontece com o código de aplicações criadas no modelo para suportar somente um cliente. Como não há mecanismo para separar os dados, estes ficam disponíveis para todos usuários. Foi criado uma réplica desta aplicação, inserido o *framework* Alligator e efetuado testes para verificar o comportamento do *framework*, que se mostrou eficiente para as *queries* simples contidas na aplicação.

5 **RESULTADOS**

Para o teste do *framework* em produção, foi criado uma pequena aplicação *web* simulando atividades básicas de um *petshop*. O diagrama de classes da aplicação pode ser visualizado através da figura 16.

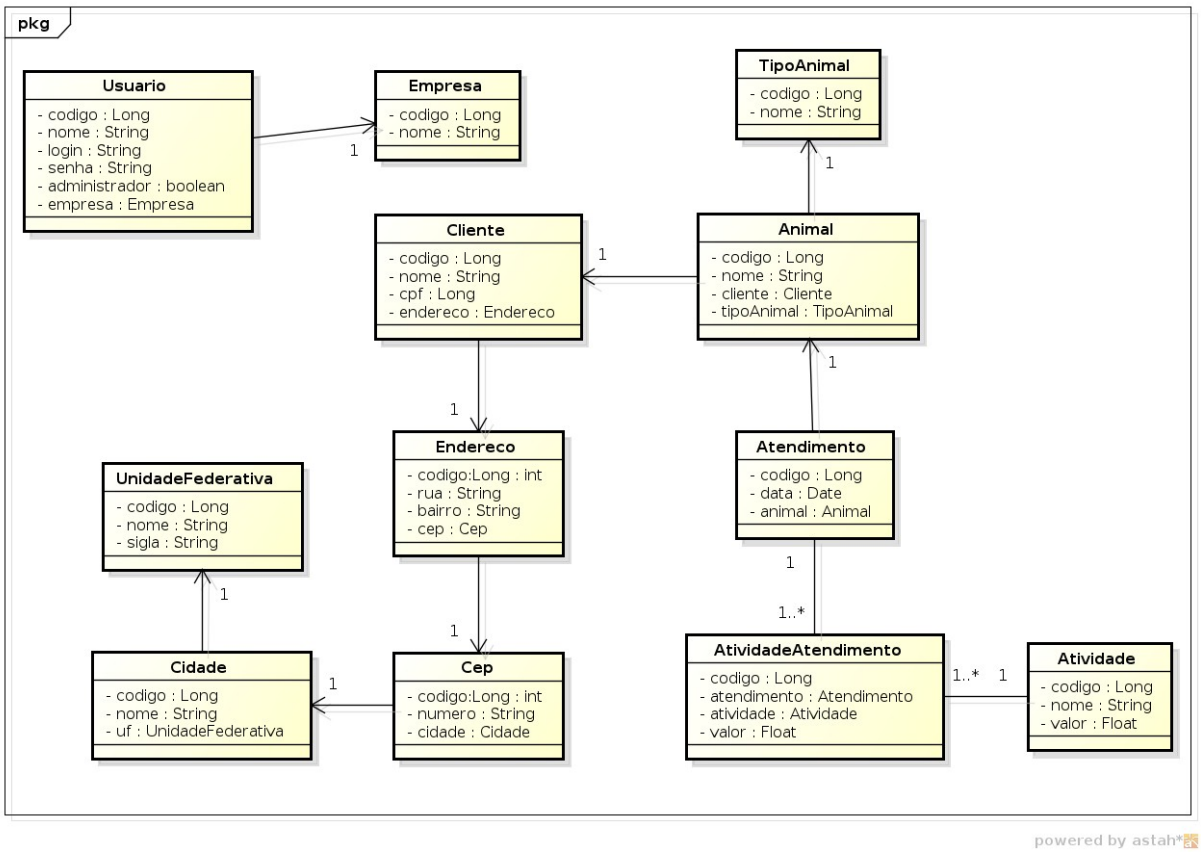

Figura 16: Modelo de relacionamento entre as classes da aplicação de teste do Alligator.
Fonte: o autor.

Com este modelo, pode-se simular boa parte das operações que um sistema real pode fazer, inclusive operações sobre listas e em cascata, para os testes de introspecção de primeiro nível. Para este caso, onde a aplicação foi modelada para servir apenas um cliente por instância, não é necessário uma ligação entre o cliente e a empresa, pois ela fica subentendida.

A aplicação gerada por este modelo passou por algumas modificações para trabalhar com o Alligator, e pôde-se perceber o seu funcionamento observando as consultas geradas pelo *framework* ORM com a condição do filtro inserida e executada corretamente. Através da imagem 17 é possível notar que, com o Alligator, a aplicação ganhou mais uma camada situada antes da camada de

persistência, para possibilitar que as alterações do *framework* sejam executadas corretamente.

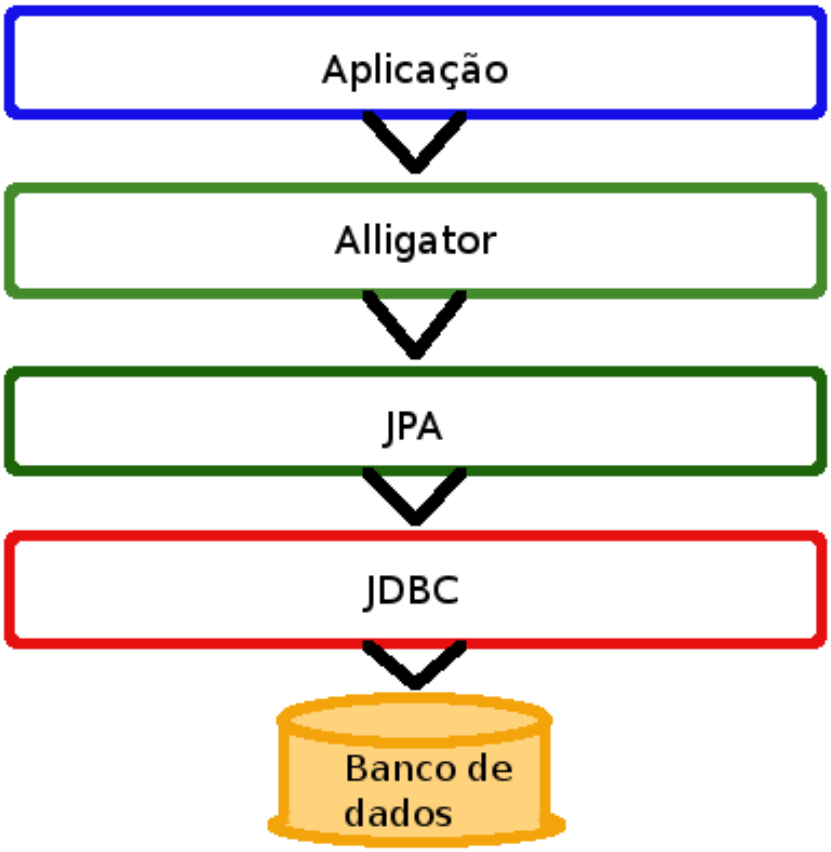

Figura 17: Camadas de uma aplicação com o framework Alligator.
Fonte: o autor.

Após incluir a biblioteca do Alligator no projeto, a primeira mudança se refere à obtenção do *EntityManager*, que deve ser obtida através da classe *TenantContext.java* contida no Alligator. Para tanto, é necessário a sua inicialização. Pode-se inicializar o contexto de várias maneiras, e uma delas recebe como parâmetro uma instância normal de *EntityManager*(para criar o *proxy*) e o nome do pacote onde se encontram as classes de entidade. Após a inicialização, pode-se obter o *EntityManager* através do comando *TenantContext.getEntityManager()*. Esta alteração é demonstrada na imagem 18.

```java
public class JPAUtil {

    private static EntityManagerFactory emf = null;
    private static EntityManager em = null;

    private static void createEntityManagerFactory() {
        if (emf == null) {
            emf = Persistence.createEntityManagerFactory("unitPetshop");
        }
    }

    public static EntityManager getEntityManager() {
        createEntityManagerFactory();
        if (em == null) {
            em = emf.createEntityManager();
            /* inicialização do contexto */
            TenantContext.initialize(em, "br.edu.unoesc.petshop.model");
        }
        /* retorno do EntityManager */
        return TenantContext.getEntityManager();
    }

}
```

Figura 18: Código da classe utilitária que fornece o EntityManager para o restante da aplicação.
Fonte: o autor.

O próximo passo é alterar as classes que necessitam ser separadas por inquilino, fazendo-as herdar de *AbstractTenantModel*, como no exemplo da imagem abaixo.

```java
@Entity
@Table(name = "TIPO_ANIMAL")
public class TipoAnimal extends AbstractTenantModel implements Serializable {
```

Figura 19: Classe de entidade com herança de AbstractTenantModel.
Fonte: o autor.

Nesta etapa, deve-se também verificar a existência de campos únicos nas classes filhas de *AbstractTenantModel* e, caso houver algum, esta condição deverá ser retirada do atributo e transferida para uma chave única composta, fazendo par com "*tenantId*", conforme a imagem 20.

```java
@Entity
@Table(name = "TIPO_ANIMAL", uniqueConstraints={@UniqueConstraint(columnNames={"nome", "tenantId"})})
public class TipoAnimal extends AbstractTenantModel implements Serializable {
```

Figura 20: Ajuste de chaves únicas.
Fonte: o autor.

Após este passo, é necessário informar qual classe representará o inquilino, proprietário dos registros filhos de *AbstractTenantModel*. Isso é feito fazendo com que esta classe implemente a interface *TenantOwner* e, consequentemente, seus métodos abstratos, que disponibilizarão o código de identificação. No caso do nosso protótipo, a classe que representa o inquilino é a *Empresa.java*, e sua declaração ficou como a imagem 21.

```
@Entity
@Table(name = "EMPRESA")
public class Empresa implements TenantOwner, Serializable {
```

Figura 21: Definição da classe que representa o inquilino através da interface TenantOwner.
Fonte: o autor.

Feito isso, agora é só informar ao Alligator com qual *TenantOwner* o framework irá trabalhar. Como no protótipo a classe *Usuário.java* possui um relacionamento com a *Empresa.java*(que implementa *TenantOwner*), assim que é criada uma sessão para o usuário, é indicado ao Alligator que o inquilino será a empresa do usuário, conforme o comando da figura 22.

```
@PostConstruct
public void setTenantOwner(){
    TenantHolder.setTenantOwner(getUsuarioLogado().getEmpresa());
}
```

Figura 22: Definição do TenantOwner ao iniciar a aplicação.
Fonte: o autor.

Em aplicações *web*, também é preciso adicionar a biblioteca do Alligator para suporte a este tipo de aplicação, além de adicionar o trecho de código da imagem 23 no arquivo "*web.xml*".

```
<filter>
    <filter-name>tenantPropagation</filter-name>
    <filter-class>br.edu.unoesc.alligator.webmodule.filter.TenantOwnerPropagationFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>tenantPropagation</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Figura 23: Configuração do módulo web do Alligator.
Fonte: o autor.

Após o *framework* em funcionamento, as instruções passaram a ser executadas com o devido filtro, como pode-se perceber através da imagem abaixo.

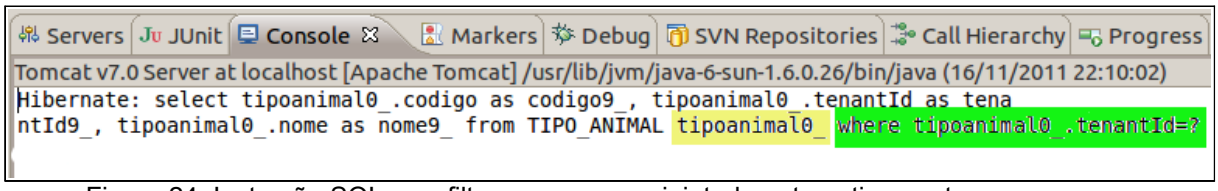

Figura 24: Instrução SQL com filtro por empresa injetado automaticamente.
Fonte: o autor.

Através do filtro do Alligator, foi possível suportar no mesmo sistema e com o mesmo banco de dados, duas empresas distintas sem que uma tivesse acesso aos dados da outra e sem reescrever a instrução de consulta. Na imagem 25 pode-se perceber o acesso efetuado por uma empresa(A) e o acesso simultâneo por outra empresa(B), onde são mostrados as atividades realizadas para cada animal no dia atual, e os comandos SQL gerados pela aplicação na saída do console(C).

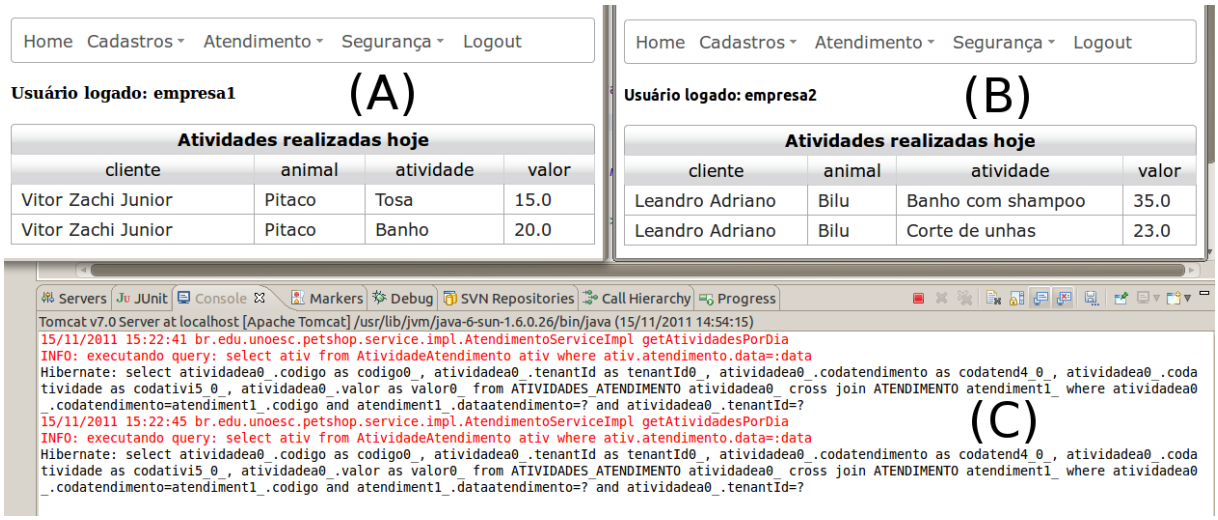

Figura 25: Representação de dois inquilinos distintos acessando um sistema com o Alligator ao mesmo tempo.
Fonte: o autor.

Complementando a imagem anterior, a seguir será visualizado a tabela de cadastro de atividades, que também serviu de base para os dados visualizados anteriormente, onde pode-se perceber uma coluna(*tenantId*) contendo o identificador de cada registro da tabela.

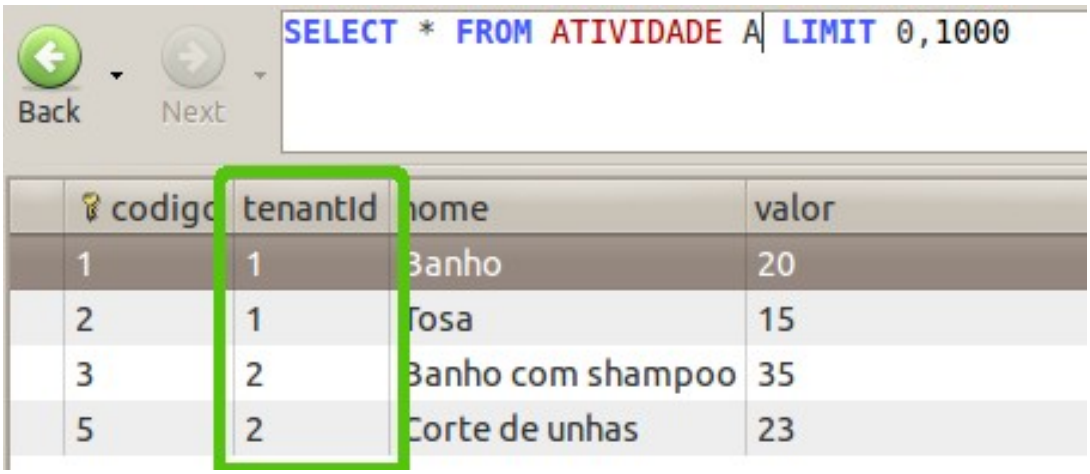

Figura 26: Tabela de cadastro de atividades da aplicação com o Alligator.
Fonte: o autor.

A seguir serão apresentadas as conclusões levantadas até o momento.

6 **CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS**

Com base nos três protótipos, foi possível perceber no código-fonte as diferenças entre os modelos analisados e a quantidade de retrabalho necessário para ajustar a aplicação para o suportar *multitenancy*, mesmo sendo um protótipo pequeno e com poucas classes .

Foi possível concluir também que o desenvolvimento do *framework* possibilitou uma transição do modelo de aplicação JPA para o modelo *multitenancy* sem transtornos, sendo que não foi necessário reescrever nenhuma instrução SQL da aplicação antiga. Com isso, ele se torna muito útil neste tipo de migração, onde o desenvolvedor pode manter o foco nas regras de negócio da aplicação, deixando para o *framework* a responsabilidade e garantia de executar o filtro nos dados requisitados ao banco de dados.

Foram feitos testes de performance do Alligator, comparando o tempo de execução de um comando JPA no protótipo sem o framework e no protótipo com o framework. Com base no resultado dos testes, pode-se concluir que a inclusão do Alligator não causou muito aumento no tempo das consultas (em média 1 milissegundo por *query*). A tabela 1 contém os dados comparativos do teste executado.

Tempo gasto(ms) no protótipo sem Alligator	Tempo gasto(ms) no protótipo com Alligator	Diferença	Número de consultas
4	4	0	1
2	2	0	1
2	3	1	1
83	118	35	11
35	41	6	4
222	268	46	50
3	4	1	1

Tabela 1: Comparativo de tempo gasto entre aplicação com e sem o Alligator.
Fonte: o autor.

Ainda há muito a fazer para otimizar o *framework*. Certamente haverá situações em que seja necessário a não execução do filtro e, como o funcionamento

deste é baseado na classe presente na consulta SQL, será preciso criar um mecanismo que possa “desativar” o recurso quando necessário.

Também é necessário analisar a situação das chaves únicas nos registros referentes às classes filhas de *AbstractTenantModel,* onde deve-se percorrer as classes e modificar a restrição para a chave única, formando uma chave composta pelo próprio atributo único e o código do inquilino(*tenantId*). Esse procedimento é necessário pois com a implementação *multitenancy*, a chave única não pode ser compartilhada. Supondo um exemplo com uma classe *Cliente* e que esta possua um atributo único chamado “cpf”. Caso a chave única permaneça compartilhada, um mesmo cliente cadastrado em um inquilino não pode ser cadastrado em outro, o que não é correto. Formando uma chave composta, esse procedimento poderia ser executado normalmente. Uma boa solução seria conseguir modificar o código-fonte das entidades de forma automática, criando estas chaves compostas automaticamente, e possibilitando o *framework* ORM executar estas mudanças no banco de dados.

Outro recurso interessante a ser implementado é a execução do filtro no segundo nível. Por exemplo, o Alligator consegue trabalhar tranquilamente onde uma classe *Empresa* é a proprietária dos registros e uma classe *Cliente* pertence à uma *Empresa*(os clientes são de uma empresa). Por outro lado, não consegue trabalhar com o filtro na classe Cliente caso seja inserido um outro nível de ligação, como por exemplo Empresa → Filial → Cliente, a menos que a classe *Filial* seja configurada como a proprietária dos registro, o que talvez não seja uma abstração correta da realidade, além de poder gerar certa confusão.

Outro motivo pelo qual o desenvolvimento deste *framework* tornou-se útil foi a possibilidade de implementação do filtro independente da implementação JPA que for inserida no projeto. Tomamos como exemplo os dois frameworks de ORM mais conhecidos e populares na comunidade Java, que são o Hibernate, mantido pela JBoss, e o TopLink, mantido pela Oracle. Ambos estão de acordo com o padrão JPA, sendo portanto candidatos a servir como implementação JPA em uma aplicação. O Hibernate possui um mecanismo de filtro por anotações sobre as entidades, que pode ser acessado e habilitado somente fora do padrão especificado pela JPA. Já o TopLink possui um mecanismo de filtro diferente do Hibernate, também acessado

somente fora da especificação JPA.

Já com o *framework* Alligator, a implementação JPA não interfere no funcionamento do mesmo, pois o comando JPQL é modificado antes de passar pela implementação. Teoricamente, uma consulta no padrão JPA deve ser executada da mesma forma em qualquer implementação.

REFERÊNCIAS BIBLIOGRÁFICAS

Annotation Type MappedSuperclass. 2007. Disponível em: <http://download.oracle.com/javaee/5/api/javax/persistence/MappedSuperclass.html >. Acesso em: 10 set. 2011.

BAUER,Christian; KING, Gaving. **Hibernate em ação**. Rio de Janeiro: Ciência Moderna, 2005. 517 p.

BEUGHLEY, Lynn. **Use a cabeça SQL**. Rio de Janeiro: Alta Books, 2008.

FABBRO, Luís Gustavo. **Sua solução SaaS é multitenant?**. 2011. Disponível em: <http://balaiotecnologico.blogspot.com/2011/03/sua-solucao-saas-e-multi-tenant.html>. Acesso em: 30 out. 2011.

HEMRAJANI, Anil. **Desenvolvimento ágil em Java com Spring, Hibernate e Eclipse**. São Paulo: Pearson Prentice Hall, 2009. 290 p.

GOMES, Yuri Marx Pereira. **Java na Web com JSF, Spring, Hibernate e Netbeans 6**. Rio de Janeiro: Ciência Moderna, 2008. 175 p.

GONÇALVES, Edson. **Dominando Java Server Faces e Facelets Utilizando Spring 2.5, Hibernate e JPA**. Rio de Janeiro: Ciência Moderna, 2008. 366 p.

MARTINEZ, Eduardo. **Você sabe o que é SaaS, PaaS e IaaS?**.2010. Disponível em: <http://webholic.com.br/2010/06/07/voce-sabe-o-que-e-saas-paas-e-iaas/>. Acesso em: 27 ago. 2011.

PLACED, Enrique. **Cloud computing**: Acronymns(IaaS, PaaS and SaaS). 2011. Disponível em: <http://www.haikumind.com/cloud-computing-acronyms-iaas-paas-and-saas/>. Acesso em: 16 nov. 2011.

RUSCHEL, Henrique; ZANOTTO, Mariana S.; DA MOTA, Wélton Costa. **Computação em nuvem**. 2010. Disponível em: <http://www.ppgia.pucpr.br/~jamhour/RSS/TCCRSS08B/Welton%20Costa%20da%20Mota%20-%20Artigo.pdf>. Acesso em: 27 out. 2011.

SOUSA Jr., Rafael T.; PUTTINI, Ricardo S. **Principais aspectos na segurança de redes de computadores**. Disponível em: <http://www.redes.unb.br/security/introducao/aspectos.html>. Acesso em: 07 nov. 2011.

SOUZA, Jefferson Teixeira de. **Padrões GoF**. Disponível em: <http://www.lia.ufc.br/~eti2005/menu/modulos/PS/ParteIII_GoFb.pdf>. Acesso em: 27

nov. 2011.

JPA tutorial – Getting started. Disponível em: <http://schuchert.wikispaces.com/JPA+Tutorial+1+-+Getting+Started>**.** Acesso em: 28 nov. 2011**.**

SOMMERVILLE, Ian. **Engenharia de software**. São Paulo: Pearson Prentice Hall, 2007. 552 p.

TAURION, Cezar. **Entendendo o modelo multitenancy**. 2010. Disponível em: <http://imasters.com.br/artigo/19067/cloud/entendendo_o_modelo_multi-tenancy/>. Acesso em: 08 nov. 2011.

_____. **Cloud Computing**:Computação em nuvem: Transformando o mundo da Tecnologia da Informação. Rio de Janeiro : Brasport, 2009.

TORRES DA SILVA, Viviane. **Frameworks**. 2010. Disponível em: <www.ic.uff.br/~viviane.silva/2010.1/es1/util/aula13.pdf>. Acesso em: 15 mar. 2012.

Trail: The Reflection API. Disponível em: <http://docs.oracle.com/javase/tutorial/reflect/index.html>. Acesso em: 28 out. 2011.

Trugger description. 2011. Disponível em: <http://sourceforge.net/projects/trugger/>. Acesso em: 25 set. 2011.

What is JDBC? Disponível em: <http://java.sun.com/docs/books/jdbc/intro.html>. Acesso em: 26 nov. 2011.