Framework de Persistência para Implementação De Aplicações *Multi-tenant* em Java

Matheus de A. Cordeiro[1], Emanuell F. H. de Lucena [1]

[1]Faculdades Integradas de Patos (FIP)
58.700-250 – Patos – PB – Brasil

{matheus, emanuell}@ffm.com.br

Abstract. *The growing demand for cloud applications, due to high maintenance costs of the services offered by the traditional model of software offering, created the architectural model multi-tenancy, which allows the optimization of resources and infrastructure software systems sharing the same application instance and maintaining customer data logically separate. In the Java language, the JDBC API, widely used by the developer community, still does not natively support multi-tenant connections. This paper presents a framework that has these features to make possible the implementation of multi-tenant applications in Java using JDBC API for connecting to the database. At the beginning of this work, the methodology consists in performing literature searches on the central issue and related topics in order to acquire the necessary theoretical knowledge. Then, using the knowledge acquired in the previous step, are described the requirements and structure of a framework, applying the concepts previously seen. Finally, the creation and implementation of a case study using the framework presented.*

Resumo. *Com a crescente demanda de aplicações em nuvem, devido aos altos custos de manutenção dos serviços oferecidos pelo modelo tradicional de oferta de software, surge o modelo de arquitetura multi-tenancy, que permite a otimização de recursos de infraestrutura e sistemas de software compartilhando uma mesma instância de aplicação e mantendo os dados dos clientes separados de forma lógica. Na linguagem Java, a API JDBC, amplamente utilizada pela comunidade de desenvolvedores, ainda não dá suporte nativo a conexões multi-tenant. Este trabalho apresenta um framework que segue essas características para tornar possível a implementação de aplicações multi-tenant em Java utilizando a API JDBC para conexão com o banco de dados. Nas primeiras etapas deste trabalho, a metodologia adotada consiste na realização de pesquisas bibliográficas sobre o assunto central e temas relacionados, com o intuito de adquirir o conhecimento teórico necessário. Em seguida, utilizando-se dos conhecimentos adquiridos na etapa anterior, são descritos os requisitos e estrutura de um framework, aplicando os conceitos vistos anteriormente. Por fim, a criação e execução de um estudo de caso utilizando o framework apresentado.*

1. Introdução

As aplicações de *software* tradicionais, também chamadas de *stand-alone*, são construídas e disponibilizadas seguindo o modelo de arquitetura *single-tenant*, no qual corresponde ao modelo de manutenção que utiliza-se de um conjunto de licenças contratuais e custos anuais de suporte para um cliente em questão, seguindo todas as especificações desejadas.

Do ponto de vista do fornecedor, a aplicação se torna específica para um único cliente, e qualquer expansão ou reuso para novos clientes, que buscam funcionalidades semelhantes, exige um grande esforço de remodelagem e desenvolvimento.

Com o surgimento da *Cloud Computing*, virtualização de produtos e serviços computacionais, tornou-se possível a adoção do modelo de arquitetura *multi-tenancy*, onde uma aplicação, hospedada em servidores na Internet, é compartilhada entre vários usuários distintos, de forma que estes acessem a mesma aplicação e tenham seus dados independentes dos demais usuários. Com isso, o acesso aos dados se torna restrito, eliminando o risco de acessos não-autorizados.

Ao contratar o serviço, o usuário pode acessá-lo a partir de um navegador *Web*, economizando custos com licenciamento e recursos, tais como: pessoas, *hardware*, *software* e instalação. Neste modelo, o usuário paga apenas pelo que usar e os recursos da aplicação são de fácil expansão [Brito 2012].

Atualmente, a linguagem de programação Java já possui *frameworks* de conexão com banco de dados que podem possibilitar a separação necessária para a implementação de aplicações *multi-tenant*. Um *framework* é um conjunto de classes que incorporam um arcabouço para solucionar determinados problemas relacionados [Fayad et al. 1999]. Um dos *frameworks* Java mais populares é o *Hibernate* [Kabanov 2011]. Com ele, é possível realizar o mapeamento objeto-relacional de uma aplicação, o gerenciamento de conexões com o banco de dados, além do suporte à *multi-tenancy*.

Internamente o *Hibernate* utiliza-se de uma *Application Programming Interface* (API), que é um conjunto de classes e interfaces nativamente desenvolvidas, chamada de *Java Database Connectivity* (JDBC). Essa API tem a função de abstrair a forma de comunicação e envio das instruções da aplicação para o banco de dados. Todavia, a API JDBC ainda não dá suporte nativo à conexões *multi-tenant*. Por ser muito utilizada pela comunidade de desenvolvedores Java, é interessante que haja uma solução para esse novo modelo de aplicações o mais próximo da API nativa.

Portanto, o objetivo deste trabalho é construir um *framework* para implementação de *multi-tenancy* em aplicações Java que utilizam a API JDBC para conexão com banco de dados.

Nas primeiras etapas deste trabalho, a metodologia adotada consiste na realização de pesquisas bibliográficas sobre o assunto central e temas relacionados, com o intuito de adquirir o conhecimento teórico necessário. Em seguida, utilizando-se dos conhecimentos adquiridos na etapa anterior, são descritos os requisitos e estrutura de um framework, aplicando os conceitos vistos anteriormente. Por fim, a criação e execução de um estudo de caso utilizando o framework apresentado.

Para a apresentação da pesquisa realizada, o presente artigo é composto por mais cinco seções. A segunda seção apresenta os principais conceitos relacionados à arquitetura *multi-tenancy*. A terceira, trabalhos relacionados à construção de *frameworks* que permitem a utilização da arquitetura. A quarta, o *framework* construído. A quinta, o estudo de caso. Por fim, a sexta apresenta as considerações finais sobre o trabalho e os trabalhos futuros.

2. *Multi-tenancy*

Multi-tenancy é uma arquitetura na qual uma única instância de um aplicativo de *software* é utilizada por vários consumidores. Cada consumidor é chamado de *tenant*. Os *tenants*, cada qual formado por um conjunto de usuários ligados a ele, podem personalizar algumas partes da aplicação, tais como a cor da interface gráfica ou até mesmo regras de negócio, mas estes não podem personalizar o código do aplicativo. Isto é possível porque os dados de cada *tenant* são logicamente ou fisicamente separados dos demais, ou seja, não é possível para um *tenant* ter acesso aos dados de outro *tenant* [Rouse 2011].

A arquitetura *multi-tenancy* é dividida em três componentes: autenticação, persistência (armazenamento) dos dados em banco de dados e configuração [Bezemer et al. 2010].

2.1. Autenticação

Em uma aplicação *multi-tenant*, todos os *tenants* utilizam o mesmo ambiente físico, ou seja, compartilham a mesma aplicação e a mesma instância de banco de dados. Para ser capaz de oferecer customização do ambiente e ter certeza de que os *tenants* podem acessar somente os seus próprios dados, os *tenants* devem ser autenticados [Bezemer et al. 2010].

Enquanto que a autenticação de usuários é, possivelmente, já presente na aplicação, um componente separado para a autenticação de *tenants* pode ser necessário. Geralmente é mais fácil introduzir um mecanismo de autenticação adicional, ou seja, remover ou adicionar mais uma tela de *login*, do que modificar um já existente.

2.2. Persistência

Em uma aplicação *multi-tenant* há uma grande exigência para o isolamento dos dados. Já que todos os *tenants* usam a mesma instância de um banco de dados, é necessário garantir que eles acessem somente os próprios dados [Bezemer et al. 2010].

Há três formas de isolar os dados utilizando um banco de dados: Máquina Compartilhada, Processo Compartilhado e Tabela Compartilhada [Jacobs e Aulbach 2007].

Na Máquina Compartilhada, cada *tenant* terá seu próprio banco de dados, porém, compartilhando a mesma infra-estrutura de aplicação. Esta forma é a mais segura das três porque os dados ficam totalmente separados fisicamente e, na ocorrência de um ataque a uma base de dados particular, as demais bases não são afetadas diretamente. Pelo mesmo motivo, essa separação também facilita a migração de dados de seus *tenants* [Jacobs e Aulbach 2007].

A principal limitação desta abordagem é a falta de uso de escalonamento de memória e de disco, afinal, para cada base de dados, a quantidade de memória e de disco utilizada deve ser igualitária e, dependendo do uso por *tenant*, há desperdício destes, já que um pode necessitar menos que outro [Jacobs e Aulbach 2007].

No Processo Compartilhado, os *tenants* passam a compartilhar o mesmo banco de dados, consequentemente os mesmos processos e recursos. Apesar de utilizar o mesmo banco de dados, os dados são separados por tabelas, ou seja, cada *tenant* ainda terá seus dados fisicamente separados dos demais. Além disso, com o compartilhamento de processos e de *pool* de conexões, o escalonamento de memória é mais eficiente. Sua principal limitação é com relação a flexibilidade de adição e remoção de *tenants*, porque, para determinados banco de dados, tarefas como remoção em massa de tabelas enquanto o sistema está em execução podem ser problemáticas e comprometer o desempenho [Jacobs e Aulbach 2007].

Na Tabela Compartilhada, os dados dos *tenants* são armazenados nas mesmas tabelas, porém cada qual associado com seu respectivo *tenant*. Para isso é adicionado em cada tabela uma coluna que identifica para qual *tenant* o dado está associado, ou seja, para cada consulta ao banco de dados esse valor deverá ser informado. Esta forma possui o melhor escalonamento de memória e de disco, porque qualquer operação lida somente com o identificador do *tenant* em questão [Jacobs e Aulbach 2007].

2.3. Configuração

Em uma aplicação *multi-tenant*, a customização da aplicação deve ser possível através de configuração. Para permitir que o usuário tenha uma experiência como se estivesse trabalhando em um ambiente dedicado, é necessário a implantação de pelo menos quatro tipos de configuração [Bezemer et al. 2010]:

- Estilo de *layout*: customizar a interface gráfica como temas, *logos*, *banners*, e cores.
- Configuração geral: customizar configurações específicas, como chave de criptografia e detalhes do perfil pessoal.
- Entrada e saída de arquivos: customizar a especificação de caminhos para diretório de arquivos, tais como arquivos de relatórios.
- Fluxo de trabalho (*Workflow*): customizar tarefas que cada *tenant* pode executar antes e após a realização de uma tarefa comum, como por exemplo, um cadastro de cliente. Como exemplo desta customização pode-se citar o envio de um e-mail personalizado após o cadastro de um usuário na aplicação.

3. Trabalhos Relacionados

Já existem algumas formas de implementação de *multi-tenancy*, dentre elas a fornecida pelo *framework Hibernate* na sua versão 4. Nessa abordagem, é oferecida uma conexão com o banco de dados que pode variar de acordo com o nome do *tenant* informado. Essa conexão é a responsável por mapear os *tenants* e apontar para seus respectivos dados, de forma a abstrair essa implementação do desenvolvedor. Para isso, os *tenants* ficam associados às conexões e não à lógica de negócio [Red Hat 2013].

Isso pode ser considerado um ponto negativo desta abordagem, pelo fato do *tenant* ser uma extensão de uma tabela que identifica a quem aqueles dados pertencem. Por este motivo, os *tenants* devem estar associados à quem representa as tabelas na programação orientada a objetos, ou seja, devem estar associados às classes de negócio.

[Bezemer et al. 2010] mostra a construção de um *framework* que realiza a associação dos *tenants* às classes de negócio na linguagem .NET, e sugerem a criação de um que promova, além da separação lógica e customização de interface gráfica, a possibilidade de customizar *workflows*. Para o estudo de caso, são criadas páginas ASP.NET para cada *tenant* específico para a configuração de interface gráfica. A customização de *workflows* não foi realizada no estudo de caso.

Utilizando o *plugin multi-tenant* para *Grails*, [Neto et al. 2009] associou os *tenants* à conexão com o banco de dados e também às classes de negócio. Para as classes que são *multi-tenant*, a anotação *@MultiTenant* necessita ser declarada. Nem todas as classes precisam ser anotadas, pois nem todas as tabelas de uma aplicação necessitam da separação por *tenant*, já que podem existir dados que serão compartilhados entre todos os *tenants*. Como exemplo, pode-se citar uma lista de cidades e estados de um país para a exibição em um cadastro de cliente.

Por conseguinte, com a anotação, é possível identificar quais dados serão restritos e quais serão compartilhados entre os *tenants*. Em Java, uma anotação é um metadado que fornece dados sobre um programa para serem utilizados em tempo de execução [Oracle 2013].

4. *Framework* de Persistência *Multi-tenancy*

Esta seção apresenta as características, a estrutura, composições e funcionalidades do *framework* construído.

4.1. Especificação do Protótipo

Com o intuito de explorar os conceitos de *multi-tenancy*, foi desenvolvido um protótipo de *framework* em Java capaz de realizar o mapeamento objeto-relacional de entidades de negócio, levando em consideração os *tenants* criados.

De um modo geral, o *framework* deve ser capaz de abstrair a complexidade da separação dos dados dos *tenants*, de uma forma simples, rápida e que exija o mínimo de modificações necessárias para uma possível migração de uma aplicação JDBC single-*tenant* para *multi-tenant*.

Dentre os modelos de isolamento de dados presentes na literatura sobre *multi-tenancy*, o modelo tipo Tabela Compartilhada foi escolhido para ser utilizado pelo *framework*, por sua facilidade de implementação e eficiência.

Internamente, em nível de banco de dados, a separação dos *tenants* é dada por uma coluna que irá representá-lo e que é adicionada automaticamente em todas as tabelas que desejam ter seus dados separados. Essa coluna referencia uma nova tabela, chamada de "*Tenant*", que possui código e nome, criado automaticamente pelo *framework*. A nível de aplicação, as classes responsáveis pela lógica de negócio e classes de pesquisas ao banco possuem um parâmetro extra que representa o respectivo *tenant*. Entretanto, suas informações são automaticamente recuperadas em tempo de

execução pelo *framework*, para que o desenvolvedor não tenha a necessidade de lidar com este parâmetro.

Para atingir a extensibilidade e fácil adaptação, pensou-se em criar uma camada acima da persistência, abstraída por uma classe de negócio, que realiza as modificações necessárias nas consultas ao banco de dados, facilitando a separação lógica dos dados por *tenants*. Além disso, os mesmos e as colunas da tabela a serem pesquisadas também foram abstraídas em classes, para organizar a busca dos dados.

Os *tenants* estão associados somente às classes de negócio e suas respectivas tabelas. Não há a necessidade de associá-los à conexão para realizar a separação dos dados. Na Figura 1 é ilustrada a associação entre o *framework*, o JDBC e o banco de dados. Observa-se que, para operações *multi-tenant*, o desenvolvedor não tem contato direto com o JDBC, sendo o *framework* responsável por abstrair essa comunicação e realizar as operações necessárias.

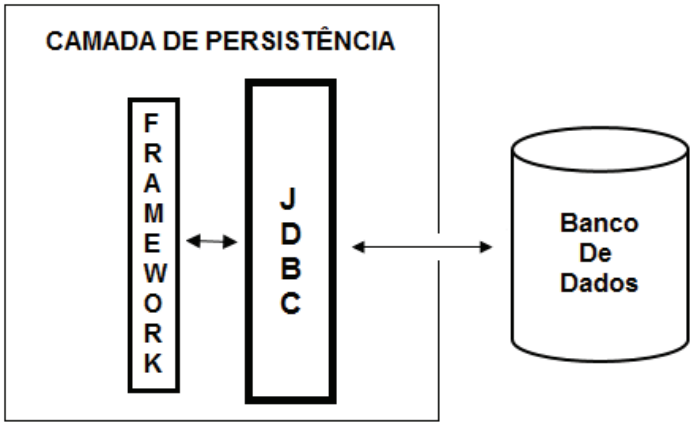

Figura 1. Associação do *framework* com a API JDBC.

4.2. Estrutura e Diagrama de Classes

O *framework* possui cinco classes, duas interfaces e um *enumerator*, contendo as operações básicas de *Structured Query Language* (SQL) sobre uma coluna de uma tabela. Um *enumerator* é uma lista enumerada de valores pré-definidos [Serson 2009]. Na Figura 2 é ilustrado o diagrama de classes do *framework*.

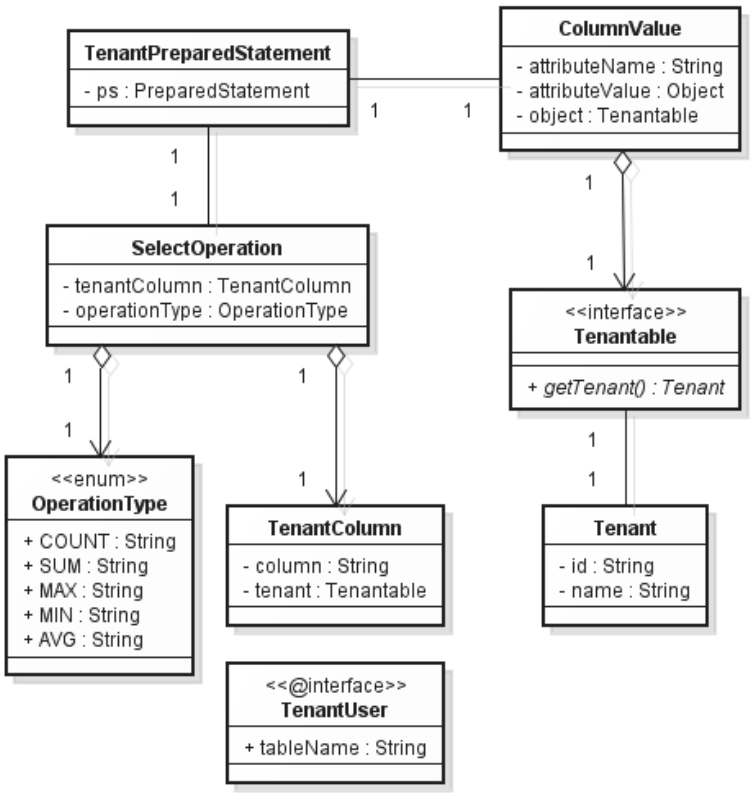

Figura 2. Diagrama de classes do *framework*.

O relacionamento entre as classes é realizado da seguinte forma:

- A classe *TenantPreparedStatement* possui relação com a interface *PreparedStatement* que, por sua vez, é nativa da API JDBC. Esta também possui associação com as classes *SelectOperation* e *ColumnValue* que são utilizadas para abstrair dados para as consultas SQL.
- A classe *SelectOperation* abstrai um *TenantColumn* e um *OperationType*, para que seja possível realizar consultas com essas operações de forma correta, simplificada e com um alto nível de abstração.
- O *enumerator OperationType* armazena o nome algumas operações SQL: *Count* (Contar elementos), *Sum* (Somar elementos), *Max* (Retornar o maior elemento), *Min* (Retornar o menor elemento) e *Avg* (Obter a média dos elementos).
- A classe *ColumnValue* possui relação com dois objetos: um *String* que representa o nome do atributo a ser utilizado na consulta e um *Object* que pode assumir qualquer valor a ser buscado para aquele atributo. Ela também possui relação com uma classe de negócio, ou seja, classe da aplicação que irá utilizar o *framework*, que implementa a interface *Tenantable* para a comparação de chaves estrangeiras, coluna que faz a associação entre duas ou mais tabelas.
- A interface *Tenantable* faz uma associação com a classe *Tenant*. Essa interface deve ser implementada por toda classe de negócio que deverá ter seus dados separados por *tenant*.
- A classe *Tenant* representa um *tenant*, com uma identificação e um nome.

- A classe *TenantColumn* possui relação com uma classe que implementa a interface *Tenantable* e um *String* que representa a coluna da tabela. É utilizada em conjunto com a classe *SelectOperation* e também é utilizada na classe *TenantPreparedStatement* para a operação *order by* do SQL, que ordena uma consulta por uma determinada coluna da tabela.
- A interface *TenantUser* é utilizada para definir o nome da tabela da classe que implementa a interface *Tenantable*. Essa é utilizada também na realização das consultas pela classe *TenantPreparedStatement* para recuperar esses nomes. Esta interface é declarada como um anotação nas classes de negócio.

A classe base do *framework* é a *TenantPreparedStatement*. A partir dela, as consultas são modificadas de forma a considerar os *tenants* informados e buscar somente os dados pertencentes para aquele *tenant*. A classe possui oito métodos (funções) públicos que podem ser utilizados, onde um é para criar um *tenant* e quatro deles se referem à operação *select* do SQL. Na Figura 3 é ilustrado a entidade *TenantPreparedStatement* (o nome *PreparedStatement* foi abreviado para PS a fim de melhorar a visualização).

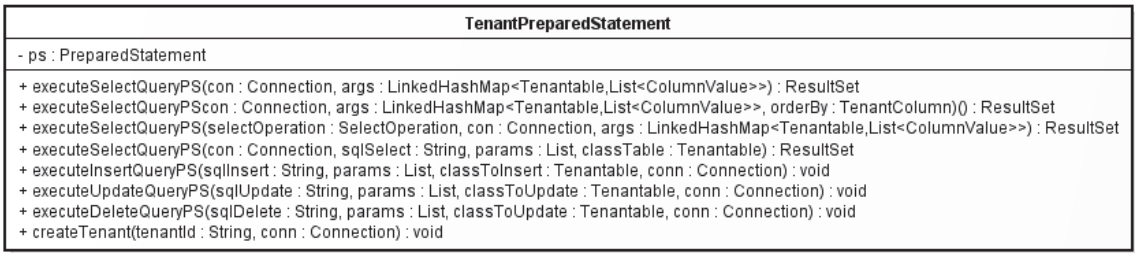

Figura 3. Métodos da classe *TenantPreparedStatement*.

5. Estudo de Caso

A aplicação escolhida para realizar o estudo de caso com o *framework* criado foi a *ONLINESHOP*, que simula um site de comércio eletrônico. Esta aplicação possui as opções de adicionar produtos ao carrinho de compras, cadastro de usuários e realização de compras. Sua interface administrativa permite o cadastro, atualização e remoção de produtos, categorias e visualização das vendas realizadas.

A aplicação manipula seis tabelas no banco de dados: a tabela produtos, pedidos, itens_pedido, categorias, usuários e *admin*, que armazenam, respectivamente, as informações sobre os produtos, tais como *id* (código de identificação) e nome, informações sobre os pedidos realizados, a associação entre produtos e pedidos, categorias, usuários da aplicação (os consumidores que realizam os pedidos) e os administradores da aplicação (que podem cadastrar os produtos e categorias), como ilustrado na Figura 4.

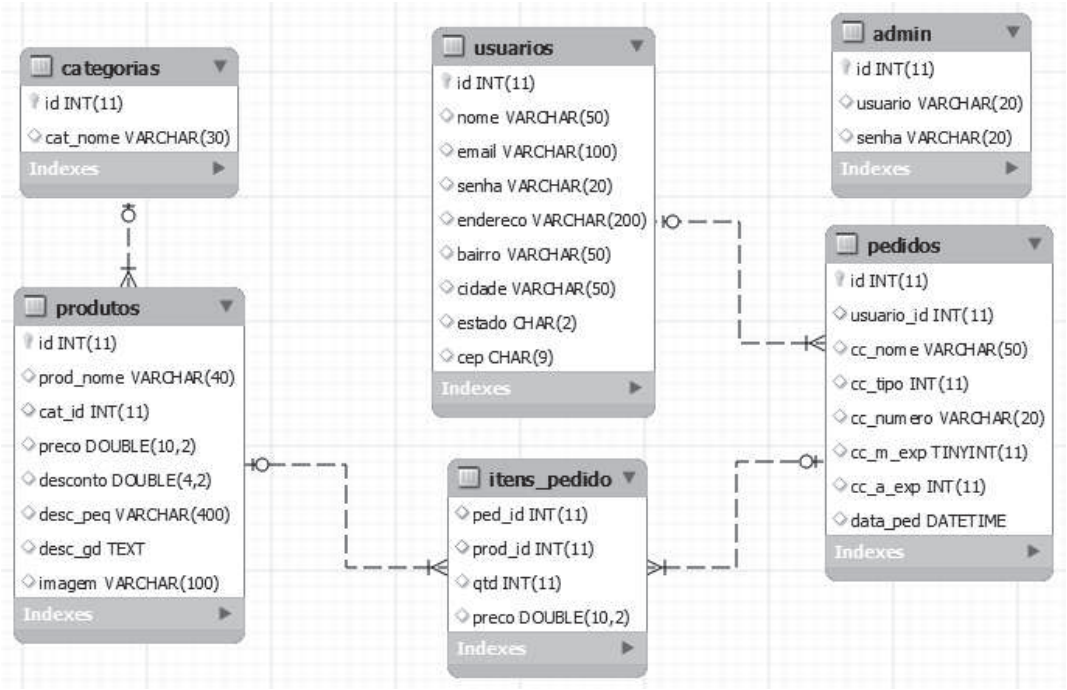

Figura 4. Tabelas manipuladas pela *ONLINESHOP*.

Após a inserção e utilização do *framework*, todas as tabelas automaticamente criaram um relacionamento com a nova tabela "*tenant*", também criada automaticamente por ele. Para complementar, uma nova tabela chamada de "*tenantadmin*" foi criada manualmente, para a adição (cadastro) de *tenants* por meio de uma interface gráfica da aplicação e em tempo de execução. O novo relacionamento entre as tabelas é ilustrado na Figura 5.

Observa-se que, para cada tabela da Figura 4, foi adicionado o campo "*TENANT_ID*", que é o responsável por representar o *tenant* de cada registro. Como consequência dessa mudança, todos os registros existentes no banco de dados da aplicação foram removidos. Essa atitude evitou que existissem registros sem *tenants* associados.

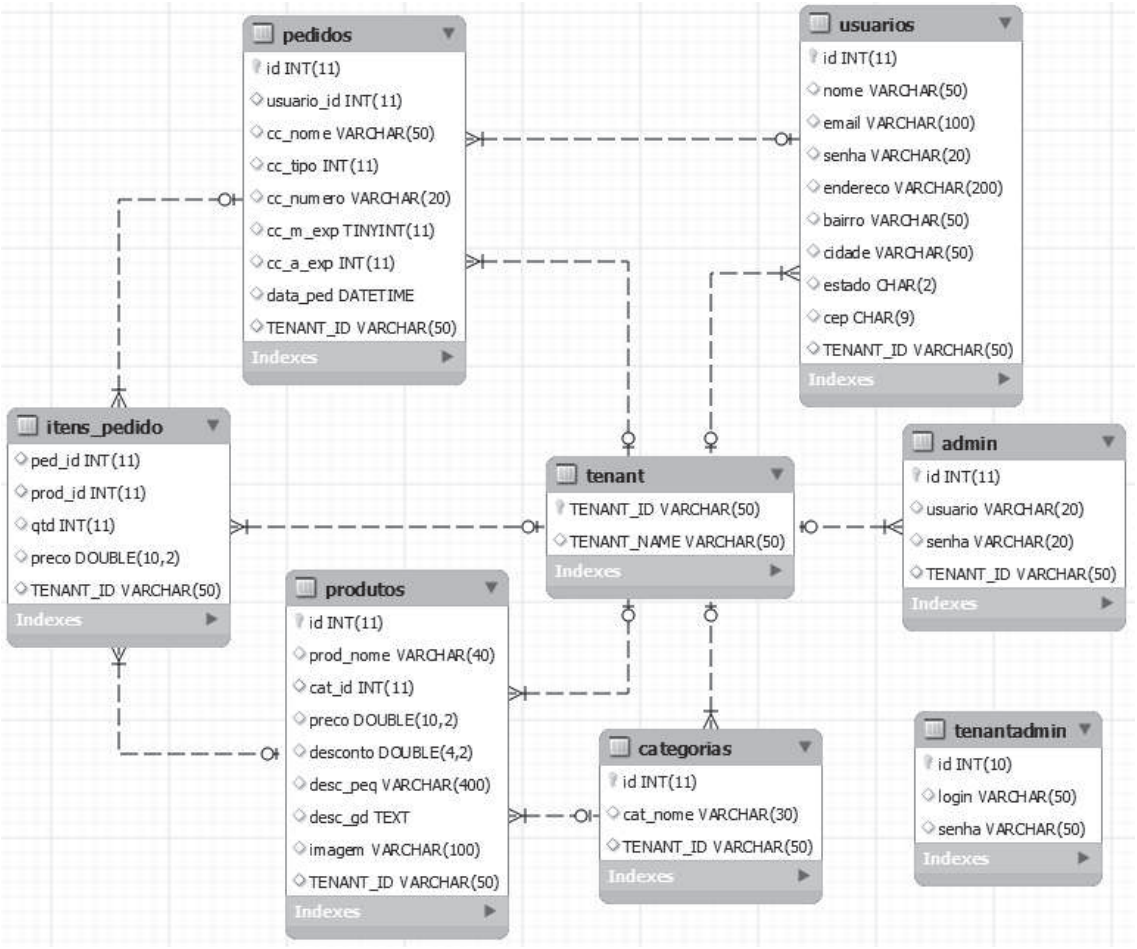

Figura 5. Tabelas da *ONLINESHOP* após a inserção do *framework*.

6. Considerações Finais

Foi apresentado neste trabalho o conceito de *multi-tenancy*, uma arquitetura emergente no contexto de desenvolvimento de *software*, que tem como objetivo permitir que uma mesma aplicação seja acessada por vários usuários diferentes, tendo seus dados, incluindo customizações de interface gráfica e *workflows*, separados dos demais, dando a impressão que são aplicações diferentes. Também foi apresentada a construção de um *framework* para a utilização de *multi-tenancy* com a API JDBC e um estudo de caso para verificar sua aplicabilidade em *softwares* reais.

O estudo de caso em questão foi uma aplicação de comércio eletrônico chamada *ONLINESHOP*, que permite exposição de produtos filtrados por categorias, cadastro de usuários que desejam comprar os produtos e a finalização do pedido propriamente dito, originalmente implementada para ser *single-tenant*. Por ser uma aplicação de comércio eletrônico, a arquitetura *multi-tenancy* se encaixou facilmente neste contexto, devido à existência de uma grande quantidade de aplicações com esta finalidade e à pouca exigência de mudanças com relação às regras de negócio de cada empresa para este nicho de aplicações.

O *framework* construído permite que desenvolvedores que ainda utilizam JDBC não necessitem migrar para *Hibernate* para obter formas mais simples e automatizadas

de separar dados por *tenants*, dado que é possível realizar as principais funcionalidades SQL para consulta a banco de dados através do mesmo, embora que seja necessário remover os registros da tabela em que o *tenant* será inserido, ou migrá-los manualmente, já que cada registro terá que ser associado a um *tenant*, sendo isto um ponto negativo da utilização do *framework*.

6.1. Trabalhos Futuros

Como trabalhos futuros, propõe-se estudos destinados ao gerenciamento automático de registros que não possuem um *tenant* associado. Dada a necessidade da remoção dos registros já existentes após a inserção do *framework*, em uma aplicação *single-tenant*, ou inclusão manual dos *tenants* para os registros, na versão atual do *framework*.

Referências

Bezemer, P., Zaidman, A., Platzbeecker, B., Hurkmans, T., Hart, A. (2010). Enabling multi-tenancy: An industrial experience report. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–8. IEEE.

Brito, E. (2012) "Esqueçam o Service Desk tradicional, agora é SaaS e na nuvem", http://www.tiinside.com.br/09/04/2012/esquecam-o-service-desk-tradicional-agora-e-saas-e-na-nuvem/os/271945/news.aspx, Abril.

Fayad, M. E., Schmidt, D. C., Johnson, R. E. (1999), Building application frameworks: object-oriented foundations of framework design, John Wiley & Sons.

Jacobs, D. e Aulbach, S. (2007). Ruminations on Multi-Tenant Databases: An industrial experience report. In *Proceedings of the Technologie und Web on Datenbanksysteme in Business*, pages 514–521. BTW.

Kabanov, J. (2012) "Java EE Productivity Report 2011", http://zeroturnaround.com/rebellabs/java-ee-productivity-report-2011, Novembro.

Neto, J. R., Garcia, V. C., Oliveira, O. S. (2009) "Desenvolvendo aplicações multi-tenancy para computação em nuvem", http://www.die.ufpi.br/ercemapi2011/minicursos/MC4.pdf, Outubro.

Oracle. (2013) "Lesson: Exceptions", http://docs.oracle.com/javase/tutorial/essential/exceptions, Abril.

Red Hat. (2013) "Chapter 16 Multi-tenancy", http://docs.jboss.org/hibernate/core/4.1/devguide/en-US/html/ch16.html, Abril.

Rouse, M. (2011) "multi-tenancy", http://whatis.techtarget.com/definition/multi-tenancy, Abril.

Serson, R. R. (2009), Certificação Java 6 - Volume 1 - Teoria, Brasport.