UMA ARQUITETURA PARA REDUÇÃO DA COMPLEXIDADE NO DESENVOLVIMENTO DE NOVAS APLICAÇÕES MULTI-TENANT COM BANCO DE DADOS E ESQUEMAS COMPARTILHADOS

Mário Santos Sousa[1], Karlos Kelvin Santos[1], David Duarte[2], João Abreu[2], Guilherme Esmeraldo[2]

[1] Universidade Federal do Cariri

[2] Laboratório de Sistemas Embarcados e Distribuídos (LEDS)
Instituto Federal de Educação, Ciência e Tecnologia do Ceará – *campus* Crato

{*mario.santos,kelvin.santos*}*@ufca.edu.br, {davidduarte,joaoalberto,guilhermealvaro}@ifce.edu.br*

RESUMO: *O desenvolvimento de aplicações multi-tenant requer um cuidado peculiar, sobretudo quando o modelo de dados escolhido é uma abordagem de total compartilhamento, ou seja, banco de dados e esquemas compartilhados. Essa abordagem não possui isolamento físico dos dados, dessa forma o provedor do serviço precisa implementar mecanismos para garantir a separação dos dados, criando assim um isolamento lógico. O isolamento dos dados deve ser forte o suficiente a fim de evitar que os dados de um determinado inquilino sejam expostos a outros. Desenvolver tal mecanismo não é uma tarefa simples, porém é possível reduzir a complexidade desse trabalho por meio da seleção de ferramentas adequadas. Esse trabalho tem um caráter experimental e aborda o isolamento de dados de uma aplicação multi-tenant utilizando uma arquitetura de software implementada com tecnologias Java. A arquitetura proposta propicia uma forma de redução do esforço de desenvolvimento e do tempo necessário para implementar a separação de dados, garantindo maior viabilidade técnica para a construção e oferta de software como serviço.*

Palavras-chave:*.Multi-Tenant. Arquitetura de Software. Abordagem Compartilhada. Tecnologias Java. Software como Serviço.*

ABSTRACT: *The development of multi-tenant applications requires a peculiar care, mainly when the model of database organization includes a sharing approach, in other words, database and schema shared. This approach does not have a physical isolation model, thereby the service providers have to implement the mechanisms to ensure the data separation creating a logical isolation. The data isolation model must ensure that tenants just can access their own data. To develop this mechanism is not a simple task, however it is possible reduce the complexity of this task by selecting proper tools. This paper shows an experimental research and demonstrate the development of dada isolation in a multi-tenant application using a software architecture developed by using Java technologies. The proposed architecture provides a way to reduce the programming effort and project time needed to implement the data separation, ensuring more technical viability to develop and offer software as a service.*

Keywords: *Multi-Tenant. Software Architecture. Shared Approach. Java Technologies. Software as a Service.*

1. INTRODUÇÃO

Software como serviço, ou *Software as a Service* (SaaS), é uma modalidade de serviço de nuvem onde os aplicativos são oferecidos como um serviço, ou seja, os clientes pagam apenas pelo acesso e quantidade de funcionalidades utilizadas através da internet (VERAS, 2012). Quando um aplicativo é ofertado como serviço, o provedor da solução obtém uma economia significativa por meio do compartilhamento dos recursos para servir mais clientes, o que é conhecido como economia de escala (BEZEMER e ZAIDMAN, 2010).

Porém, ainda segundo Bezemer e Zaidman (2010), para que os provedores de soluções dessa natureza possam obter uma maior economia de escala é necessário que eles utilizem um modelo específico de aplicação SaaS denominado *multi-tenant,* o qual propicia a oportunidade de maior utilização do *hardware* onde a solução está sendo executada. Existem muitas definições para aplicações *multi-tenant*, porém elas continuam vagas (BEZEMER e ZAIDMAN, 2010). Para Neto at al. (2012 apud HARRIS e AHMED, 2011), *multi-tenant* é um tipo de aplicação SaaS onde uma única instância da solução é compartilhada por todos os inquilinos.

Aplicações *multi-tenant* são soluções que permitem que os clientes compartilhem os mesmos recursos de *hardware* por meio de uma única instancia da aplicação, enquanto permite também a eles a configuração da aplicação de acordo com a suas necessidades, como se estivessem utilizando um ambiente

dedicado (BEZEMER e ZAIDMAN, 2010). Com essa definição é possível perceber que os clientes necessitam que a aplicação se comporte de acordo com suas regras de negócios, geralmente traduzidas pelas configurações realizadas no sistema SaaS, e que possua um bom desempenho durante o compartilhamento de recursos. O isolamento de dados em aplicações *multi-tenant* é talvez a característica mais importante desse tipo solução. É necessário garantir que nenhum dado seja exposto para clientes que não sejam os respectivos proprietários.

Em soluções *multi-tenant* é possível realizar esse trabalho de diversas formas, uma vez que existe mais de uma abordagem para separação e organização de dados dos clientes no banco de dados. Essas abordagens são: banco de dados separados; banco de dados compartilhado e esquemas separados; e, por último, banco de dados e esquema compartilhados (VERAS, 2012).

Na abordagem de banco de dados e esquema compartilhados, os dados de todos os clientes ocupam as mesmas tabelas. Dessa forma, a separação de dados deve ser realizada de forma lógica, no código da aplicação, o que é apontado, por Veras (2012) e outros autores, como fator para aumento na complexidade da aplicação, tornando o esforço inicial de desenvolvimento muitas vezes maior que o esperado e, consequentemente, encarecendo o projeto.

O presente trabalho aborda, de forma experimental, o desenvolvimento de uma arquitetura para aplicações *multi-tenant* com banco de dados e esquemas compartilhados. Com a arquitetura proposta busca-se comprovar que é possível minimizar a complexidade e o esforço de programação necessários para realizar o isolamento de dados de aplicações dessa natureza, escolhendo componentes apropriados.

O restante do artigo está dividido da seguinte maneira: Na próxima seção, é realizada uma revisão teórica acerca do tema deste trabalho; A Seção 3 apresenta a arquitetura proposta para aplicações *multi-tenant*; As Seções 4 e 5 apresentam um estudo de caso e as considerações finais, respectivamente.

2. REVISÃO TEÓRICA

Para Chong, Carraro e Wolter (2006), uma das mais altas prioridades para sistemas SaaS é a criação de uma arquitetura de dados robusta e segura, pois os dados são os bens mais importantes para qualquer negócio. Os autores ainda evidenciam essa necessidade apontando a falta de confiança como principal fator para a não adoção de sistemas SaaS.

Ao se considerar o modelo *multi-tenant*, há a necessidade de se desenvolver um modelo de dados adequado para a aplicação, uma vez que a mesma deve lidar com dados de diversos clientes ao mesmo tempo. Chong e Carraro (2006) apontam como um desafio a criação de um modelo de dados onde os clientes possam estender dentro de um ambiente de vários inquilinos. Sem dúvida a escolha de uma arquitetura adequada é desafiante, pois cada cliente possui necessidades diferentes, as quais um modelo de dados padrão e rígido poderá não atender.

Existem três abordagens para resolver esse problema e, segundo Veras (2012), cada uma delas apresenta vantagens e desvantagens. São elas: banco de dados separado, esquema separado e esquema compartilhado.

Utilizar a abordagem de armazenamento em bancos de dados separados é a forma mais simples de se obter um modelo de dados isolado (VERAS, 2012). Para Chong e Carraro (2006), alguns clientes possuem requisitos de isolamento tão rígidos que nem mesmo considerarão um aplicativo que não forneça seu próprio banco de dados. Chong, Carraro e Wolter (2006) recomendam essa arquitetura, pois ela simplifica a extensão do modelo de dados e operações, tais como cópias de segurança e restauração de informação. Assim, em caso de falhas é possível restaurar os dados dos inquilinos de forma fácil. Nessa abordagem, os recursos da máquina são compartilhados por todos os inquilinos alocados em um mesmo servidor, o que ocasiona a diminuição de clientes por servidor, pois o mesmo pode não lidar com muitas instâncias de banco de dados (VERAS, 2012). Chong e Carraro (2006) expõem como desvantagem da arquitetura o custo alto de infraestrutura, pois será possível dar suporte a um número limitado de bancos de dados por servidor.

Na abordagem de banco de dados compartilhado e esquemas separadas os inquilinos utilizam o mesmo banco de dados, onde cada um deles possui um conjunto de tabelas que são organizadas em um esquema específico. Para Veras (2012), essa abordagem é relativamente fácil de implementar, embora ofereça um grau moderado de lógica de isolamento. Quando o cliente se inscreve no serviço, um conjunto de tabelas padrões é criado para o mesmo. Após serem criadas, as tabelas não necessitam estar em conformidade com o padrão e os inquilinos podem adicionar e modificar colunas, garantindo a extensibilidade do modelo. Chong, Carraro e Wolter (2006) apontam que essa abordagem acomoda mais inquilinos por servidor, em relação a abordagem com banco de dados separados. Além disso, o uso dessa arquitetura pode influenciar em uma aplicação mais barata, porém algumas operações, como as de restauração de dados dos inquilinos, são mais difíceis, pois enquanto na abordagem com bancos separados a restauração se resume a restaurar o banco de dados a partir do último backup, na arquitetura com esquemas separados a

restauração de um banco completo pode significar a sobreposição de dados de alguns inquilinos.

Por fim, a terceira abordagem, que trata de banco de dados e esquemas compartilhados, utiliza o mesmo banco de dados para alocar os inquilinos. Cada tabela do banco possui uma coluna com o identificador do cliente para associar a informação com o inquilino apropriado. Segundo Chong, Carraro e Wolter (2006), das três abordagens essa é a que oferece menor custo de hardware, pois permite servir um maior número de inquilinos por servidor. Porém, ela implica em maior esforço no isolamento e segurança para garantir que nenhum inquilino possa visualizar dados que não o pertençam. A restauração de dados de um único cliente pode se mostrar ineficiente pelo fato de que as linhas do banco de dados que dizem respeito ao inquilino devem ser apagadas e reinseridas, causando redução de desempenho, afetando todos os demais clientes, (CHONG, CARRARO e WOLTER, 2006). Veras (2012) afirma que essa abordagem é apropriada em situações em que se deve servir um grande número de clientes por servidor e quando os clientes estão dispostos a abrir mão de isolamento físico de dados por um custo menor na aplicação.

A seção a seguir apresenta a arquitetura proposta para o modelo de redução de complexidade do isolamento lógico em bancos de dados e esquemas compartilhados.

3. DESCRIÇÃO DA ARQUITETURA PROPOSTA PARA APLICAÇÕES MULTI-TENANT

A arquitetura proposta neste artigo é dividida em camadas. Segundo Larman (2007), uma camada é o agrupamento de classes, pacotes ou subsistemas que possuem responsabilidades coesivas sobre um tópico importante do sistema. As camadas são organizadas de modo que as "mais altas" solicitem serviços das "mais baixas". A Figura 1 apresenta a organização das camadas da arquitetura proposta e as tecnologias utilizadas em cada uma delas.

A primeira camada, denominada camada de apresentação, é a responsável por apresentar aos clientes os dados da aplicação. Abaixo da camada de apresentação, a camada de controle de acesso inclui funções que permitem gerenciar a segurança da aplicação, como atividades de controle de acesso e de permissões. Na camada seguinte, a de controle, as classes representam uma fachada ou geralmente casos de uso para tratar as requisições provindas da camada de apresentação. A camada de regras de negócio representa todas as classes e subsistemas que possuem e/ou representam conceitos e regras do domínio da aplicação. Por fim, a camada de acesso a dados (DAO) é composta por classes cuja responsabilidades estão diretamente vinculadas a atividades de persistência em banco de dados.

Para implementação da arquitetura apresentada na Figura 1, utilizou-se, na camada de apresentação, páginas web desenvolvidas com XHTML e componentes JSF; o controle de acesso é realizado pelo Spring Security - um *framework* baseado na especificação JAAS do Java EE -; a camada de controle é realizada por *Beans* do JSF (essas classes são responsáveis por tratar eventos nas páginas web e acessam a camada que representa as regras de negócio da aplicação, que por sua vez acessa a camada de acesso a dados responsável por qualquer ação que envolva o banco de dados). A camada de acesso a dados, que é o foco deste artigo, pois é nela que há a separação dos dados, será detalhada a seguir.

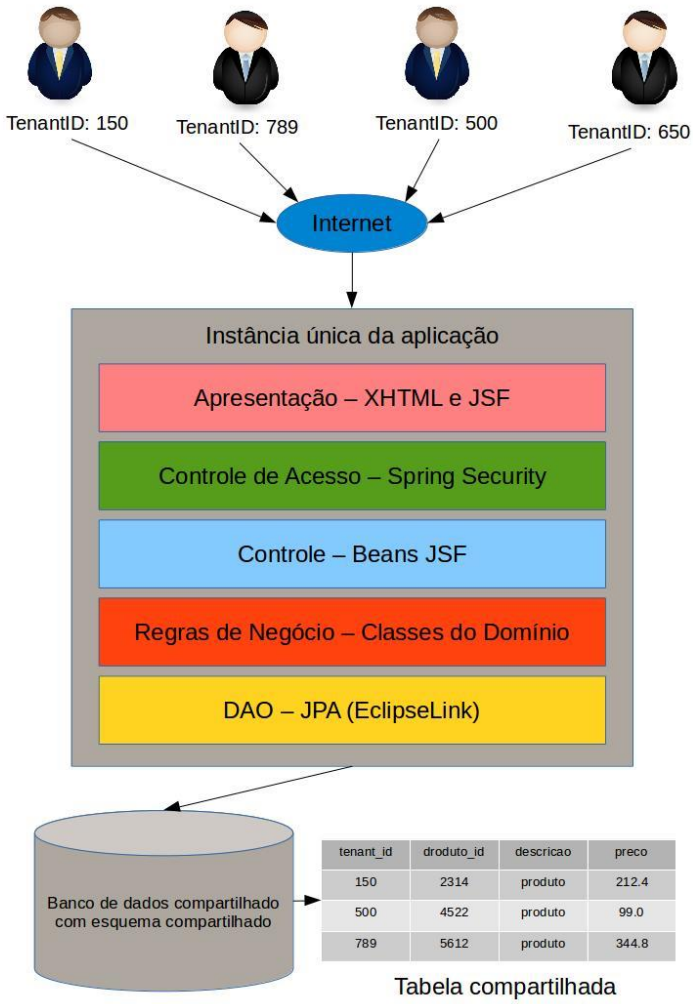
3.1. Acesso e Isolamento dos Dados

A abordagem de banco de dados compartilhados com esquemas compartilhados, embora apresente a capacidade de tornar um projeto mais barato aproveitando melhor a capacidade do *hardware* envolvido, também apresenta uma característica que pode ocasionar o aumento nos custos iniciais do projeto, a complexidade no isolamento dos dados.

Tendo como base que os dados dos clientes estão misturados em uma mesma tabela é necessário que a aplicação seja capaz de separá-los, garantindo que as informações não sejam expostas para todos os inquilinos (*tenants*). Para implementar esse requisito, um provedor de soluções SaaS precisa alocar mais tempo, no início do projeto, a fim de garantir o sucesso de tal funcionalidade. Durante esse tempo, são desenvolvidas tarefas de codificação e testes para assegurar a confiabilidade da aplicação.

Para isolar os dados o sistema, é necessário tratar inicialmente cada requisição do usuário, antes de acessar o banco de dados. Isso é necessário para identificar a que usuário a requisição está direcionada e, em seguida, identificar os dados pertencentes a ele. Imaginando um cenário onde o programador lide com cada requisição ao banco de dados manualmente, em bancos de dados relacionais, cada acesso deveria indicar qual cliente disparou tal requisição a fim de associar os dados da mesma ao *tenant*. Por exemplo, toda instrução SQL, seja ela de consulta ou alteração, deve possuir uma cláusula *where* e um parâmetro que represente o cliente.

Figura 1. Arquitetura proposta para aplicações *multi-tenant*.



No exemplo abaixo, o código SQL inclui a coluna TENANTID, o qual faz referência ao código de um determinado cliente:

```
SELECT * FROM PROD WHERE TENANTID = 300;
```

Utilizar instruções SQL diretamente no código da aplicação pode gerar uma série de problemas. Dentre eles, pode-se citar a redução da legibilidade do código e a dificuldade para se escrever instruções complexas pois podem levar a erros. Para Lascano (2008), incluir instruções SQL diretamente no código pode ocasionar o aumento da complexidade de programação.

Nas linguagens orientadas a objeto é comum a existência de componentes que encapsulam essas dificuldades e preocupações. A linguagem Java possui uma especificação própria que regulamenta esse tipo de componente, que é a *Java Persistence API* (JPA). Para Lascano (2008), o uso da JPA alivia o trabalho do programador, uma vez que não será necessário lidar manualmente com dados persistentes e sim delegar essa função a um componente específico. A especificação JPA é utilizada na camada de acesso a dados da arquitetura aqui proposta e será abordada na próxima subseção.

3.2. Java Persistence API

JPA é uma especificação Java que provê um modelo de persistência para o mapeamento objeto-relacional, ou seja, o mapeamento de objetos de *software* para o modelo de banco de dados relacional. Essa especificação determina como escrever objetos Java para que eles possam

representar o mapeamento e fornece as interfaces que realizam o trabalho entre a aplicação e o banco de dados.

O JPA é dado apenas como a especificação, e não a implementação. Dessa forma, existem vários *frameworks* que seguem essa especificação. *Frameworks* ORMs (*Object-Relacional Mapping*) garantem à equipe de desenvolvimento a agilidade e comodidade para lidar com objetos de sistema, encapsulando os detalhes e especificidades do banco de dados, assim garantindo maior produtividade ao projeto.

A arquitetura aqui proposta utiliza um *framework* ORM baseado na JPA com a finalidade de reduzir a complexidade do desenvolvimento do isolamento lógico da aplicação encapsulando todo trabalho em componentes reutilizáveis. Dessa forma, a responsabilidade para a criação de instruções SQL são reduzidas.

3.2.1. Padronização da Abordagem Multi-Tenant pelo JPA

O fato da abordagem *multi-tenant* não ter sido prevista na especificação JPA conduz a um problema de acoplamento já que cada *framework* busca a forma mais adequada para implementar as abordagens. Assim, o projeto fica acoplado a uma implementação específica por falta de compatibilidade das demais e é de difícil modificação e extensão. Esse fato contribuiu para a escolha dos *frameworks* a serem analisados e por isso foi decidido optar pelos mais usados, para que a possibilidade de descontinuação dos mesmos seja remota.

Neste trabalho, isso implicou em um estudo de dois *frameworks* para identificar o mais apropriado para o uso, tendo em vista as especificidades do projeto e principalmente a arquitetura de dados escolhida para o mesmo. Foram analisados os dois principais *frameworks* presentes no mercado, EclipseLink e o Hibernate (ambos são de uso gratuito).

3.2.2. Escolha da Implementação JPA

Como não existe a padronização de implementação, cada *framework* apresentou uma forma distinta de solucionar o problema. Realizou-se então uma análise dos *frameworks Hibernate* e *EclipseLink*, considerando apenas a implementação de abordagem *multi-tenant*. A Tabela 1 demonstra quais as arquiteturas suportadas pelos *frameworks* analisados.

Na Tabela 1, pode ser visto que o Hibernate não suporta a abordagem com banco de dados e esquemas compartilhados, ao passo que o EclipseLink suporta a abordagem escolhida. Esse fator determinou a escolha do *framework* EclipseLink para compor a camada de acesso aos dados e realizar o mapeamento objeto-relacional do projeto.

Contudo, mesmo tendo escolhido o EclipseLink por fornecer suporte a abordagem de acesso aos dados escolhida, durante o desenvolvimento da arquitetura proposta foi possível constatar a simplicidade do uso dele, visto que a configuração e codificação necessária para tratar vários inquilinos se mostrou simples, reduzindo a complexidade envolvida.

Tabela 1. Principais frameworks e abordagens de dados suportadas.

	Banco de Dados Separados	**Banco Compart. e Esquema Separado**	**Banco e Esquema Compart.**
EclipseLink	-	X	X
Hibernate	X	X	-

3.2.3. O Framework EclipseLink

Utilizar um framework ORM reduz a necessidade do desenvolvedor se preocupar com o design da interface de acesso ao banco de dados, seja escrevendo consultas SQL ou aprendendo as especificidades de cada SGBD.

A implementação do JPA Eclipselink se encarrega de fazer a separação lógica dos dados nas operações com o banco de dados. Durante a execução da aplicação, é necessário criar estruturas que serão responsáveis pela separação lógica dos dados, onde as mesmas encapsulam toda a complexidade da tarefa, restando ao desenvolvedor criar as classes e configurá-las com anotações específicas do EclipseLink..

A separação dos dados fica disponível após o usuário autenticar-se com sucesso na aplicação, através da configuração do *EntityManagerFactory* que é uma classe da JPA responsável por criar *EntityManagers* que são objetos capazes de realizar operações no banco de dados.

A configuração para isolamento de dados é feita no momento da criação do *EntityManagerFactory* que recebe um identificador (id) do cliente como parâmetro, dessa forma todo objeto criado pela *EntityManagerFactory* terá essa configuração de forma que cada acesso ao banco de dados seja ajustado para um determinado cliente. Todo esse processo é realizado automaticamente pelo EplispeLink, garantindo assim mais segurança. A Figura 2 ilustra a troca de informação entre a camada que representa as regras de negócio e a camada de acesso a dados. Após receber uma solicitação, a camada de acesso a dados realizará um breve processamento para adicionar o identificador do cliente à operação, sendo que este processamento é realizado pelo EclipseLink e o

mesmo é transparente tanto para o usuário como ao programador.

Figura 2. Relacionamento entre camada de negócio e camada de acesso a dados.

Logo a complexidade e o esforço no desenvolvimento de aplicações *multi-tenant* com essa abordagem são reduzidos beneficiando diretamente a arquitetura proposta neste artigo, reduzindo os custos iniciais inerentes ao tempo e ao esforço de programação. Além disso, *frameworks* são componentes de *software* já testados e contribuem para o aumento da qualidade do projeto. O EclipseLink é um *framework* open-source gratuito mantido por uma grande comunidade de desenvolvedores e usado largamente no mercado.

A seção a seguir apresenta um estudo de caso onde foi possível avaliar o uso da arquitetura proposta na redução da complexidade de projeto.

4. ESTUDO DE CASO E RESULTADOS PRELIMINARES

Para a avaliação da arquitetura proposta, o estudo de caso consistiu no desenvolvimento de um sistema *multi-tenant* para gestão comercial de empresas inquilinas, utilizando a abordagem de bancos de dados e esquemas compartilhados.

A aplicação proposta consiste de um sistema web, em que cada usuário está associado a uma empresa diferente. O acesso à aplicação é dado após a autenticação do usuário, nesse momento a aplicação executará a rotina de separação de dados.

Existe uma opção que possibilita o cadastro de novas empresas que desejem utilizar a aplicação proposta. O cadastro representa a aquisição do serviço prestado pela aplicação. Para o estudo de caso, funcionalidades mais complexas, como a de pagamento e termos de contrato, não foram implementadas, pois buscou-se avaliar a arquitetura sob a ótica de separação de dados. Quando um novo inquilino é cadastrado, é criado um novo registro na tabela empresa, essa tabela armazena todos os inquilinos do sistema. A Figura 3 demonstra a respectiva tabela populada com diferentes inquilinos.

Para demonstrar o isolamento de dados do sistema, foi desenvolvido um módulo de gerenciamento de estoque (cadastro de produtos). Como exposto anteriormente, a aplicação faz uso de um único banco de dados e um único esquema, sendo assim, os dados dos clientes estão contidos fisicamente no mesmo local, e é necessário separá-los logicamente, por meio de filtros na programação. A Figura 4 ilustra a tabela produto a qual é usada para armazenar os produtos de todos os inquilinos. Nessa figura, é importante notar que cada produto está associado a um único cliente por meio da coluna *empresa_id*, onde o seu valor é utilizado pelo EclipseLink como parâmetro para qualquer ação desenvolvida no sistema, uma vez que qualquer tabela compartilhada possui uma coluna para identificação do proprietário da informação.

Figura 3. Tabela empresa.

	id [PK] serial	fantasia character var	razao_social character varying(255)
1	1	MKL Develop	MKL Desenvolvimento de
2	8	EAF	IFCE crato
3	9	Apresentaçã	Empresa Teste TCC
4	10	Talles	Talles Inc
5	11	Teste	Empresa Teste
*			

Figura 4. Tabela produto.

	id integer	empresa_id character varying(31)	descricao character varying(255)	preco double precision
1	20	8	soda	3.8
2	19	8	coca-cola	3
3	23	9	mouse	12
4	24	10	notebook	1200
5	25	10	monitor	700
6	27	11	Produto Teste	2000
7	28	11	Produto Teste 001	10.54

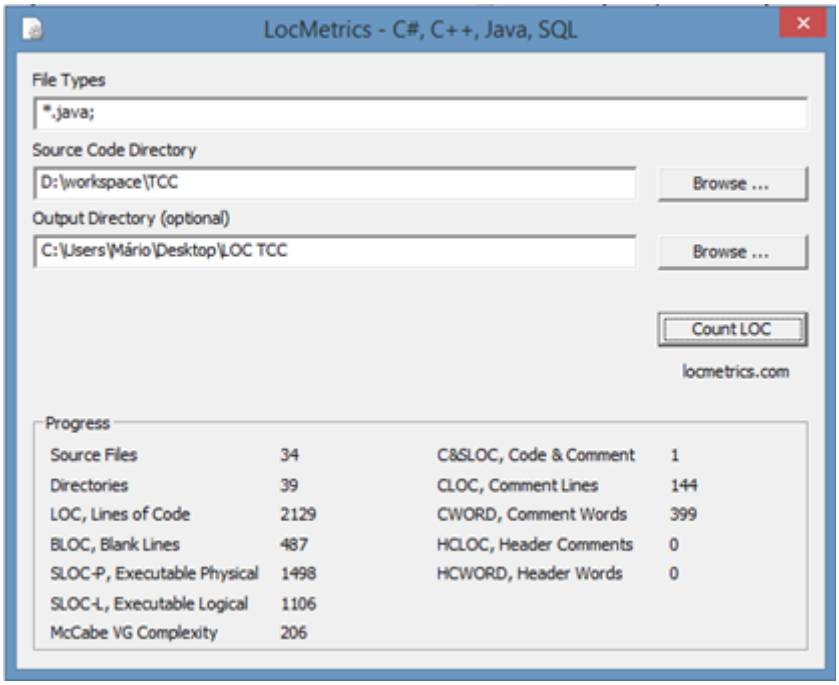
Quando um usuário acessa a interface de gerenciamento de produtos, o sistema apresenta apenas aqueles que estão associados à mesma empresa do usuário logado. A Figura 5 ilustra a visualização dos produtos de um determinado usuário (associado a empresa de código 11).

Para avaliar a complexidade da aplicação proposta, utilizou-se o *software LocMetrics*. Esse

utilitário contabiliza o total de linhas de código (LOC), linhas em branco, linhas comentadas do código, de arquivos de código e de diretórios, entre outras informações. Na Figura 6 é possível visualizar a interface do *LocMetrics*, onde para utilizá-lo é necessário informar apenas as extensões dos arquivos a serem analisados, a pasta em que o projeto se encontra e a pasta de saída para os artefatos gerados.

Figura 5. Interface para gerenciamento de produtos.

Figura 6. Tela do LocMetrics.



A Tabela 2 apresenta o resultado do uso do *LocMetrics* na aplicação de automação comercial *multi-tenant* proposta. Na Tabela 2, é possível perceber que ao todo foram utilizados 34 arquivos de código fonte para a aplicação do estudo de caso. Entre as métricas expostas na Tabela 2, destaca-se a SLOC-P, que representa as linhas de código uteis (total de linhas de código menos linhas em branco e comentadas), totalizando 1498 linhas para a implementação do isolamento de dados e todas as funcionalidades necessárias para a aplicação como login, cadastro de empresas, cadastro, listagem e edição de produtos, etc.

Outra métrica importante utilizada para medir a complexidade de uma aplicação, é a Média de Complexidade Ciclomática de McCabe (MCCABE, 1976).

Tabela 2. Métricas e respectivos resultados para a aplicação *multi-tenant* proposta.

Métrica	Total
Arquivos de Código (source files)	34
Diretórios	39
Linhas de Código (LOC)	2129
Linhas em branco de Código (BLOC)	487
Linhas de Código Físico e Executável (SLOC-P)	1498
Linhas de Código Lógico e Executável (SLOC-L)	1106
Média de Complexidade McCabe (MVG)	206
Linhas de Código e de Comentários (C&SLOC)	1
Linhas de Código Comentado (CLOC)	144
Palavras Comentadas (CWORD)	399

Essa métrica expõe o número de possíveis caminhos que o fluxo de execução da aplicação pode percorrer através da sequência de instruções do código fonte. Na Tabela 2, o valor dessa média, calculado para a aplicação do estudo de caso, é de 206. De acordo com Charney (2005), qualquer aplicação cujo código obtém a média superior a 51, pode-se ser considerada de alta complexidade.

Percebe-se então que, ao se utilizar a abordagem proposta neste artigo, pode-se obter a implementação de diferentes funcionalidades de uma aplicação SaaS *multi-tenant* para bancos de dados e esquemas compartilhados, com alta complexidade, a partir de poucas instruções de código. Isso induz a uma possível redução do esforço na codificação da aplicação e, consequentemente, a redução no tempo de projeto.

5. CONCLUSÕES

O presente trabalho abordou a proposta de uma implementação de uma arquitetura para aplicações *multi-tenant* com banco de dados e esquemas compartilhados. Os objetivos incluíram a redução da complexidade e do esforço de programação necessários para realizar o isolamento de dados de aplicações *multi-tenant*.

Os resultados mostraram que o tempo de desenvolvimento para uma abordagem compartilhada pode ser reduzido quando escolhidas as tecnologias e abordagens corretas para o desenvolvimento. A diminuição no esforço e, consequentemente, no tempo e no custo do projeto deve-se também à utilização de ferramentas de alta produtividade. É válido citar como fundamental no desenvolvimento do sistema, e que contribuiu no desempenho do projeto, o *framework* EclipseLink, onde o mesmo permitiu agilizar o desenvolvimento do isolamento de dados, encapsulando a complexidade envolvida neste processo, que é exatamente o ponto em que estava previsto o maior esforço de programação.

É fundamental para o desenvolvimento de qualquer software a preocupação com sua arquitetura e o requisito de escalabilidade. Desenvolver uma aplicação do tipo SaaS não seria diferente, visto que *software* como serviço é fornecido pela internet e possui naturalmente a expectativa de muitos acessos, uma vez que estão acessíveis em escala global.

Com a arquitetura proposta, pode-se reduzir os custos no desenvolvimento e na comercialização de software como um serviço, além da possibilidade de se obter soluções de alta qualidade e comercializá-las a custos menores.

O uso de software como serviço abre a possibilidade de alterar o contexto dos mercados para aquisição de software, havendo, em um primeiro cenário, a redução nos valores de custo de soluções preexistentes, e que não são vendidas como serviço; ou a contratação de novas soluções SaaS com valores mais acessíveis.

REFERÊNCIAS BIBLIOGRÁFICAS

ANTHONY T. V.; TOBY J. V.; ELSENPETER R. **Cloud Computing:** Computação em Nuvem Uma abordagem Prática. 2. ed. Rio de Janeiro: Alta Books. 2012. p. 173 – 191. cap. 9.

BEZEMER, C.-P.; ZAIDMAN, A. Multi-tenant SaaS applications: maintenance dream or nightmare?. **Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution** (IWPSE). ACM, 2010. p. 88-92.

CHARNEY, R. **Programming Tools: Code Complexity Metrics**. 2005. Disponível em <http://www.linuxjournal.com/article/8035>. Acesso em: 15 mai. 2015.

CHONG F.; CARRARO G.; WOLTER R. **Arquitetura de dados para múltiplos inquilinos.** Microsoft Corporation. 2006. Disponível em

<http://msdn.microsoft.com/pt-br/library/aa479086.aspx>. Acesso em: 15 mai. 2015.

CHONG F.; CARRARO G. **Estratégias de arquitetura para cauda longa (Long Tail).** Microsoft Corporation. 2006. Disponível em <http://msdn.microsoft.com/pt-br/library/aa479069.aspx>. Acesso em: 15 mai. 2015.

CHONG F.; CARRARO G. **Software como Serviço (SaaS) - Uma perspectiva corporativa.** Microsoft Corporation. 2007. Disponível em <http://msdn.microsoft.com/pt-br/library/aa905332.aspx>. Acesso em: 15 mai. 2015.

LARMAN, C. **Utilizando UML e padrões:** uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo / Craig Larman. 3. ed. Porto Alegre : Bookman. 2007.

LASCANO, J. E. **JPA implementations versus pure JDBC.** 2008. Disponível em: <http://www.espe.edu.ec/portal/files/sitiocongreso/congreso/c_computacion/PaperJPAversusJDBC_edisonlascano.pdf>. Acesso em: 29 jun. 2015.

MCCABE, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering.** pp. 308–320, 1976.

NETO, J. R.; GARCIA, V. C.; ALENCAR, A. L.; DAMASCENO, J. C.; ASSAD, R. E.; TRINTA, F. Software as a Service: Desenvolvendo Aplicações Multi-tenancy com Alto Grau de Reuso. **XVIII Simpósio Brasileiro de Sistemas Multimídia e Web**, 2012.

SOMMERVILLE I. **Engenharia de Software.** 9 ed. São Paulo: Pearson Prentice Hall, 2011. p. 103 - 123. cap 6.

VERAS, M. **Cloud Computing:** nova Arquitetura da TI / Manoel Veras. 1. ed. Rio de Janeiro: Brasport, 2012. p. 196 – 211. cap. 10.