DB2REST: Uma Solução para Integração entre Bases de Dados Legadas e Web Services RESTful

Bruno A. Oliveira[1], **Sandro S. Andrade**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA) – Salvador, BA – Brasil

{brunoaraujo,sandroandrade}@ifba.edu.br

Abstract. *The integration between computational systems implies challenges commonly related to incompatible interfaces and technologies. A particular case of this difficulty is the integration between RESTful web services and legacy databases, whose data structures are divergent. In this context, this paper aims to present DB2REST, a software solution that assists in the integration between RESTful web servers and legacy databases. To evaluate the solution, experiments were performed with and without the DB2REST, aiming to analyze the productivity, complexity and density of bugs in each scenario.*

Resumo. *A integração entre sistemas computacionais implica em desafios comumente relacionados a interfaces e tecnologias incompatíveis. Um caso particular dessa dificuldade é a integração entre web services RESTful e bases de dados legadas, cujas estruturas de dados são divergentes. Neste contexto, este trabalho apresenta o DB2REST, uma solução de software que auxilia na integração entre servidores web RESTful e bases de dados legadas. Para avaliar a solução foram realizados experimentos com e sem o DB2REST, visando analisar a produtividade, complexidade e densidade de bugs em cada cenário.*

1. Introdução

Ao longo das últimas décadas, com o advento dos Sistemas de Informação, a tecnologia passou a ser um agente catalisador de profundas mudanças nas organizações, influenciando desde a forma como são administradas e até mesmo o local de realização do trabalho [Galvão et al. 2009]. Tais soluções de software muitas vezes são desenvolvidos por diferentes fabricantes e tecnologias, seguindo uma lógica de departamentalização que está ligada a hierarquia empresarial. O principal problema decorrente dessa prática é a fragmentação dos dados gerados pelos sistemas, o que dificulta a extração de informações estratégicas que permitam criar condições para reagir a problemas e criar novas oportunidades no ambiente de negócios [Lapolli 2003]. Desta forma, faz-se necessário desenvolver mecanismos de integração entre tais sistemas.

Existem diversas situações em que a integração entre sistemas é requerida. As empresas costumam investir muito dinheiro no desenvolvimento de softwares e, para obter o retorno sobre este investimento, é necessário que o software seja utilizado por vários anos. Neste contexto, desenvolver novas soluções a cada nova demanda muitas vezes demonstra-se inviável [Pinto and Braga 2005]. Alguns casos em que comumente existea necessidade de integração são: quando têm-se bases de dados legadas, utilizadas durante muitos anos pelas organizações e que, portanto, contêm dados de valor para o negócio;

quando existe a necessidade de interoperar com clientes de múltiplas plataformas, como *desktop*, *mobile, web*, etc; quando é preciso expor serviços na web ou usar plataformas de *cloud computing*; quando têm-se B2B (*Business-to-business*), isto é, uma relação comercial entre sistemas de diferentes empresas, que por sua vez podem ter sido desenvolvidas usando diferentes tecnologias e interfaces incompatíveis.

Realizar a integração de sistemas normalmente traz vários desafios, muitas vezes relacionados a tecnologias e premissas incompatíveis. Em alguns contextos, existe perda de desempenho do sistema ou até mesmo necessidade de intervenção no código-fonte. Dentre esses desafios, podemos considerar os seguintes [Pinto and Braga 2005]: i) na maioria dos sistemas obsoletos, existe a dificuldade de manutenção de *hardwares* antigos devido ao custo ou falta de fornecedores; ii) os softwares de apoio, como o sistema operacional, ferramentas, compiladores etc, utilizados no desenvolvimento do sistema, podem estar desatualizados ou mesmo descontinuados; iii) embora um sistema possa ter começado como um sistema simples, este pode ter passado por alterações como a adição de novos programas, que compartilham dados e se comunicam com outros programas, que por sua vez, foram escritos por diferentes pessoas — que podem inclusive não estar mais disponíveis —, em diferentes linguagens de programação etc; iv) em muitos sistemas legados, os dados gerados durante o tempo de existência do sistema podem estar inconsistentes, duplicados ou em diferentes formatos de arquivos; v) as informações sobre os processos internos da organização, codificadas em uma linguagem de programação e espalhadas pelos programas que fazem parte do sistema, muitas vezes não estão satisfatoriamente documentadas, aumentando o nível de complexidade para desenvolver ou integrar novas soluções.

Um caso particular desses desafios de integração de sistemas está na adaptação entre *web service RESTful* e bases de dados legadas. Considere a existência de dois cenários: no primeiro, uma aplicação para dispositivos móveis que permite a visualização da programação de um evento. A aplicação móvel consome dados de um *web services RESTful* para exibir informações sobre os eventos, atividades realizadas e suas respectivas locações, horários, dados sobre os facilitadores etc. Este *web services RESTful* foi construído considerando uma configuração de dados específica, isto é, ela considera tabelas, colunas e relacionamentos sobre os quais os serviços são implementados. No segundo cenário, considere a existência de uma instituição que realiza eventos periodicamente, cujos dados são organizados da forma que a instituição julgou necessária. Agora, supondo que a instituição do cenário 2, que já possui uma estrutura de dados previamente definida, deseje adotar a aplicação móvel definida no cenário 1 para divulgar a ocorrência dos eventos por ela realizados, porém não deseje alterar sua forma de organizar os dados de eventos. Como adaptar essas interfaces de forma que ambas as aplicações possam manter suas características?

Algumas soluções de ORM (*Object-Relational Mapping*), como o Django *Framework* e o SQLAlchemy, permitem construir aplicações em torno de bases de dados existentes. Tais soluções possuem extensões que permitem gerar o código no padrão esperado pela ferramenta, bem como APIs RESTful, a partir de uma base legada. Entretanto, elas pressupõe a criação dessas APIs a partir da estrutura do banco de dados, não havendo suporte para adaptação nos casos em que a *API RESTful* já existe.

Neste contexto, o presente trabalho tem como objetivo a implementação e avalia-

ção de uma solução flexível para facilitar a integração entre *web services RESTful* e bases da dados legadas, o DB2REST. Este componente de software realiza a adaptação entre a camada de serviço e a camada de persistência, permitindo a integração entre esses componentes sem a necessidade de modificação da *API RESTful* existente ou do *schema* da base de dados legada, reduzindo assim os problemas de implementação e implantação destes componentes. A aplicação tem como entrada informações sobre o *backend* de banco de dados que será utilizado e um arquivo de especificação, em formato JSON (*JavaScript Object Notation*), utilizado para mapear os atributos de classes e as colunas das tabelas do bancos de dados, sem que o cliente precise efetuar outras configurações adicionais.

Para avaliar a solução proposta nesse trabalho, foram realizados experimentos em dois cenários distintos: no primeiro, um novo *web service RESTful* foi desenvolvido; no segundo, utilizou-se o DB2REST para realizar a integração entre um *web service RESTful* existente e uma base de dados legada. A partir da análise desses experimentos, foram extraídas métricas de software para avaliar a efetividade da solução.

Além desta introdução, este trabalho está organizado como se segue. A Seção 2 aborda o problema da integração entre *web services RESTful* e bases de dados legadas. A Seção 3 apresenta o material introdutório ao tema abordado nesse trabalho, tais como definição de termos utilizados e apresentação de conceitos necessários ao entendimento deste trabalho. A Seção 4 apresenta os trabalhos relacionados à solução proposta neste trabalho. A Seção 5 apresenta a solução proposta e as tecnologias utilizadas no seu desenvolvimento.

2. O problema da integração entre APIs RESTful e Bases de Dados Legada

Existem diversas soluções que permitem desenvolver sistemas a partir de bases de dados legadas. Dentre essas soluções, têm-se aquelas que utilizam a técnica de ORM (*Object-Relational Mapping*) para realizar o mapeamento de uma base de dados existente e, dessa forma, permitir o desenvolvimento de sistemas em torno desta base.

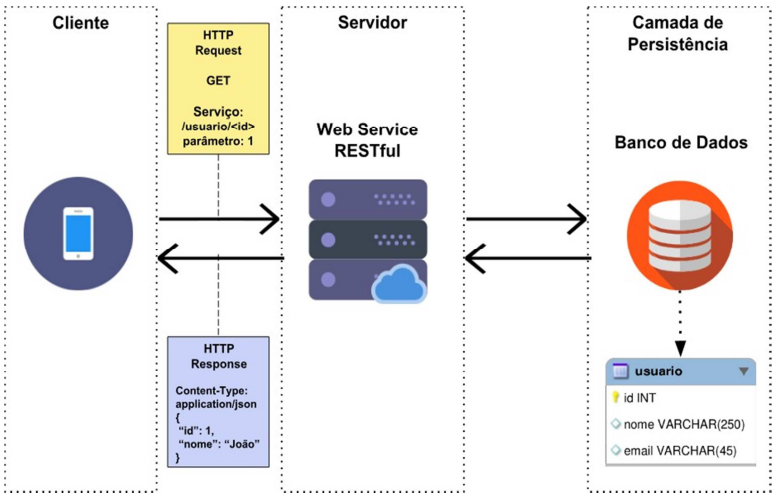

Figura 1. Cenário em que não há necessidade de integração

Existem também outras soluções que permitem criar APIs *RESTful* a partir de bases de dados legadas. No entanto, a criação facilitada dessas APIs muitas vezes está

condicionada à estrutura do banco de dados. Essa característica atende às necessidades das organizações que desejam expor serviços na web e que tem domínio sobre a API *RESTful* que foi gerada.

A Figura 1 apresenta um cenário em que um cliente de aplicativo móvel consome dados de um *Web Service Restful*. Este, por sua vez, comunica-se com a camada de persistência para realizar as operações requisitadas pelo serviço. Neste cenário, existe o domínio das regras de negócio dos serviços e da estrutura do banco de dados.

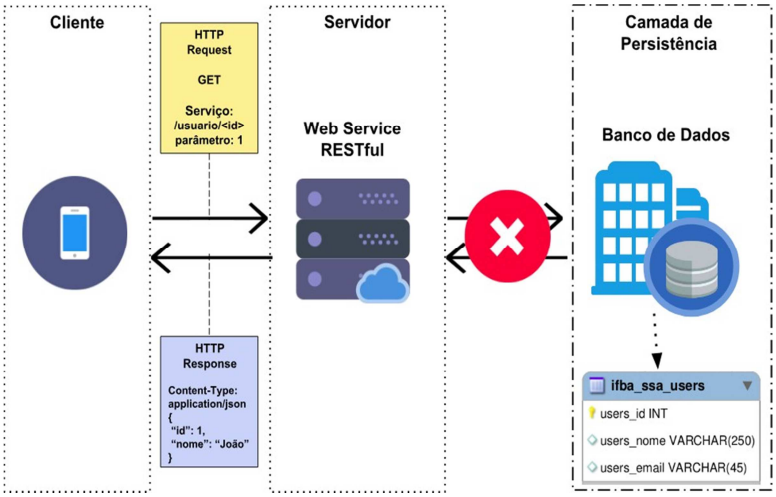

Figura 2. Cenário em que há necessidade de integração

A Figura 2 ilustra um outro cenário, similar ao apresentado na Figura 1, em que têm-se a mesma estrutura do cliente de aplicativo móvel e *Web Service Restful* porém deseja-se integrar essa estrutura a um banco de dados de outra instituição, cuja estrutura de dados é divergente daquela apresentada na Figura 1. Desta forma, tal integração não é possível devido a incompatibilidade da interface esperada pelo *Web Service Restful* e o banco de dados da instituição. Nesse contexto, o presente trabalho apresenta uma solução que visa resolver esses conflitos e viabilizar esse tipo de integração.

3. Referencial Bibliográfico

Esta seção apresenta os principais assuntos necessários para a contextualização deste trabalho. A subseção 3.1 apresenta conceitos relacionados ao estilo arquitetural REST, descrevendo suas principais características e terminologias. A subseção 3.2 apresenta algumas tecnologias utilizadas para desenvolver APIs com base na arquitetura *RESTful* e, por fim, a subseção 3.3 aborda o uso da técnica de ORM, destacando os principais benefícios obtidos a partir dos *frameworks* que implementam essa técnica.

3.1 Serviços Web RESTFul

Os *Web Services* (WS) caracterizam-se como a principal forma de utilizar os aspectos de infraestrutura da arquitetura orientada a serviços, o SOA (*Service-Oriented Architecture*), frequentemente empregada para prover estabilidade, interoperabilidade e potencial reuso dos serviços [Josuttis 2007]. Em síntese, os *Web Services* são uma tecnologia de integração de sistemas, comumente utilizados em ambientes heterogêneos, que

consiste em fornecer serviços através da Internet por meio da definição de protocolos de comunicação e das interfaces utilizadas para especificar os serviços [Josuttis 2007] [Richardson and Ruby 2007].

Tabela 1. Métodos do HTTP e suas Respectivas Operações

Metodos HTTP	Operação CRUD
GET	Obter a representação de um recurso
PUT	Atualizar informações de um recurso
POST	Criar um recurso a partir de sua representação
DELETE	Deletar um recurso
HEAD	Obter a representação de metadados de um recurso
OPTIONS	Obter a relação de métodos suportados pelo recurso

Segundo Nicolai M. Josuttis (2007,p.16), "Em essência, um serviço é uma representação de TI de algumas funcionalidades de negócios". Embora internamente os sistemas sejam técnicos na implementação das regras de negócio, as interfaces externas devem ser projetadas de forma que os desenvolvedores possam compreendê-las, sem precisar conhecer os detalhes das tecnologias envolvidas no WS [Josuttis 2007]. O SOAP (*Simple Object Access Protocol*) foi o primeiro *Web Service* real a ser desenvolvido [Josuttis 2007]. Fornece um mecanismo simples e leve de troca de informações estruturadas em um ambiente distribuído e descentralizado, usando XML (*eXtensible Markup Language*).

O REST (*Representational State Transfer*) é um estilo arquitetural que define uma série de princípios para a construção de WS, baseado no protocolo HTTP [Richardson and Ruby 2007]. Os sistemas que fornecem uma API definida pela aplicação desse estilo arquitetural recebem o nome de *RESTFul* [Richardson and Ruby 2007]. A Tabela 1 apresenta os métodos do HTTP e as respectivas operações que podem ser aplicadas a algum recurso disponível na API.

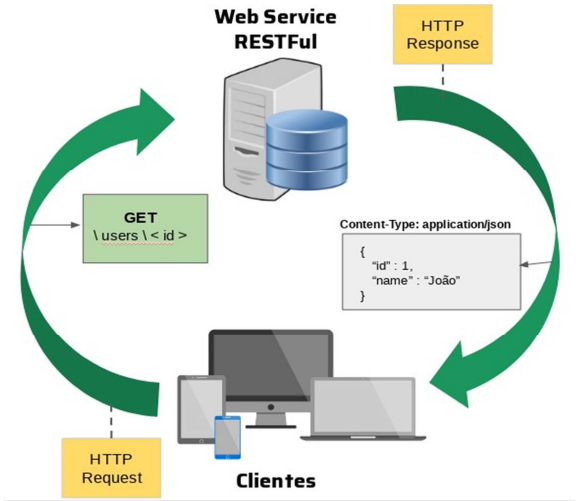

Figura 3. Ciclo de requisição em um Web Service RESTFul

A Figura 3 representa o ciclo de uma requisição definida para um *web services RESTful*. O cliente deverá enviar uma requisição (HTTP *Request*) utilizando uma das

operações do protocolo HTTP (*GET, PUT, POST, DELETE, HEAD, OPTIONS*) e o identificador do recurso. O servidor deverá receber essa requisição e executará a operação correspondente a operação HTTP sobre o recurso definido na requisição. Ao concluir o processamento, o servidor enviará uma resposta (HTTP *Response*) contendo o *status* da requisição e a representação do recurso modificado.

No contexto dos *web services RESTful*, o protocolo HTTP, antes utilizado apenas como protocolo de aplicação em outros WS, tem suas potencialidades exploradas de forma mais assertiva através do uso dos outros métodos definidos pelo protocolo. Além dos métodos do protocolo HTTP, é importante compreender os principais princípios e terminologias do REST, descritos a seguir:

- ***Resource***: a abstração chave de informação no REST é um recurso. É o nome atribuído a qualquer informação que possa receber um nome e que possa ser identificado através de uma URI (*Uniform Resource Identifier*): um documento, uma imagem, uma página HTML, uma coleção de recursos etc. [Richardson and Ruby 2007] [Tobaldini 2008];
- ***URI***: os recursos precisam ser identificados através de um *endpoint* através do qual poderão ser invocadas as operações do HTTP que serão aplicadas a um recurso [Richardson and Ruby 2007] [Tobaldini 2008];
- ***Stateless***: no REST, o servidor não mantém informações sobre o histórico de interações. Por isso, cada requisição deverá possuir todos os dados necessários para o seu processamento [Richardson and Ruby 2007] [Tobaldini 2008];
- ***Representation***: as representações são, basicamente, a forma de apresentação de um recurso, em um formato que tenha alguma utilidade para o cliente, como em JSON, XML etc. [Richardson and Ruby 2007] [Tobaldini 2008];
- ***Uniform Interface***: como o REST é baseado nos métodos do protocolo HTTP, as aplicações desenvolvidas seguindo este estilo arquitetural terão a mesma interface [Richardson and Ruby 2007] [Tobaldini 2008].

A criação de *web services RESTful* tem se tornado cada vez mais comum. Esse fenômeno ocorre graças à flexibilidade das representações de recursos, por permitirem a integração entre diferentes tipos de clientes – basicamente, qualquer cliente que tenha suporte ao protocolo HTTP poderá se comunicar com o serviço – e, sobretudo, por ser baseado em um protocolo maduro e amplamente utilizado, cujas características ajudam a prover desempenho e escalabilidade [Richardson and Ruby 2007][Hamad et al. 2010].

3.2 Tecnologias para Desenvolver APIs *RESTful*

O estilo arquitetural REST define uma série de restrições para o desenvolvimento de aplicações *RESTful*. Com a popularização do REST, diversos *frameworks* foram desenvolvidos com o objetivo de simplificar sua implementação, dispondo de funcionalidades comumente utilizadas em aplicações que adotam o estilo.

Para abordar as soluções já existentes no mercado, foram escolhidos *frameworks* desenvolvidos para linguagens de programação diversificadas, sendo selecionadas as soluções consideradas mais relevantes e utilizadas pela comunidade. Entretanto, as soluções para o desenvolvimento deste tipo de aplicação destinadas à linguagem Python, terão maior enfoque, dado que o presente trabalho tem como alvo o desenvolvimento de um mecanismo nesta linguagem.

O *Django Rest Framework* [Django REST Framework, 2017] é uma ferramenta para construção de APIs *RESTful*, em linguagem Python, utilizada de forma integrada ao *framework* de desenvolvimento Web, Django. Ele dispõe de um conjunto de funcionalidades que tornam a criação de APIs *RESTful* mais simples e intuitiva, tais como a possibilidade de criar serviços de forma quase automática por meio dos modelos de dados previamente definidos na aplicação; a fácil manipulação do modo como os modelos serão apresentados, através da definição de serializadores; possui módulos de autenticação e mecanismos de permissão bastante flexíveis; interface de API que permite realizar testes; paginação de conteúdos; filtros de conteúdo; documentação bastante detalhada e comunidade ativa [Sousa 2015].

O *Flask-RESTful* [Flask-RESTful, 2017] é uma extensão do *microframework* Flask, também desenvolvido para a linguagem Python, que torna a criação de serviços REST mais simples e fácil. O Flask-RESTful oferece uma interface simples e de fácil compreensão para construção de APIs *RESTful*, que permite formatação de respostas e definição de rotas. Apresenta como ponto forte o fato de ser independente das regras de negócio da aplicação, permitindo que os serviços possam ser implementados levando em conta as necessidades e preferências do projeto. Nesse contexto, o Flask-RESTful caracteriza-se como um ponto de partida para a construção de serviços simples, que não requerem serviços mais avançados, como módulos de autenticação, uso de ORM ou interface administrativa — embora outras extensões possam ser utilizadas em conjunto com a extensão, de forma a suprir essas necessidades [Sousa 2015].

O *Restlet* [Restlet, 2017] é um *framework* desenvolvido para a linguagem Java, que faz a aproximação dos conceitos básicos do estilo arquitetural REST, tais como recursos e representações, a artefatos equivalentes em Java. Dentre suas principais vantagens, está a flexibilidade de distribuição do módulo Java que contém a interface REST, permitindo sua integração em outras aplicações na mesma linguagem. Aliado a isto, têm-se um mecanismo de extensão que permite a inclusão de outras bibliotecas Java para suprir outros aspectos necessários para a aplicação, como módulos de segurança, bases de dados, *templates* etc [Sousa 2015].

O *Spark* [Spark, 2017], assim como o Restlet, é um *microframework* desenvolvido em Java, que tem como principal objetivo o desenvolvimento facilitado de APIs REST. Ao contrário de alguns *frameworks* deste tipo para a linguagem alvo, este visa reduzir a quantidade de configurações definidas pela especificação da linguagem para a construção de APIs REST, proporcionando assim uma experiência de desenvolvimento mais simples. O Spark dispõe das funcionalidades mínimas necessárias para este tipo de serviço, tais como: gestão das diferentes rotas disponíveis; gestão de dados de sessões; renderização de *templates* usando *Java Server Pages* (JSP) [Sousa 2015].

O *Sinatra* [Sinatra, 2017] é um *microframework*, desenvolvido para a linguagem Ruby, que segue uma abordagem leve, minimalista e, ao mesmo tempo, simples e elegante. Apesar de simples, o Sinatra é dotado de funcionalidades que auxiliam na sua adaptação às necessidades dos projetos, tais como: renderização eficiente de templates em diversos formatos; capacidade de gestão das diferentes rotas; sessões; e funcionalidades de apoio suportadas através da adição de *middlewares* ao serviço [Sousa 2015].

Existem diversos outros *frameworks* que facilitam a criação deste tipo de API.

Aqui foram apresentados aqueles considerados mais relevantes ao contexto do trabalho, em especial os *frameworks* desenvolvidos para a linguagem Python.

3.3 Object-Relational Mapping

Segundo Bauer (2005, p.23), *Object-Relational Mapping* é "uma técnica para persistir de maneira automática e transparente, os objetos de um aplicativo para tabelas em um banco de dados relacional. Em essência, transforma dados de uma representação para a outra" [Bauer and King 2005].

Object-Relational Mapping (ORM) é o conceito que possibilita o mapeamento de objetos em tabelas de bancos de dados relacionais. A técnica surgiu com o objetivo de reduzir as dificuldades de implementação de sistemas que implementam mecanismos de persistência, onde o paradigma de programação orientado a objetos está associado a um banco de dados relacional [Bauer and King 2005][Coelho and Sartorelli 2004]. O uso da técnica promove o desacoplamento entre o código da aplicação e a base de dados utilizada [Joshi and Kukreti 2014].

Entre os principais benefícios obtidos pela implementação da técnica, estão a eliminação dos comandos em SQL no código-fonte, reduzindo a complexidade das classes e, consequentemente, tempo de desenvolvimento [Bauer and King 2005]. Outro fator a ser considerado é a manutenibilidade do código, uma vez que a camada de ORM abstrai a complexidade do acesso aos dados, reduzindo assim a quantidade de linhas e desta forma, facilitando eventuais processos de refatoração do código [Joshi and Kukreti 2014][Cordeiro 2011].

A técnica de ORM é implementada por diversos *frameworks*, como SQLAlchemy e o Django ORM, desenvolvidos para a linguagem Python. Tais soluções abstraem a complexidade de implementação da técnica e permitem um excelente aproveitamento de tempo e recursos.

4. Trabalhos Correlatos

Existem diversas soluções que visam prover flexibilidade da camada de persistência em sistemas computacionais. Dentre as soluções mais relevantes está o uso da técnica de ORM [Coelho and Sartorelli 2004], atualmente implementado em vários *frameworks*, que consiste em realizar o mapeamento de objetos em memória para meios de persistência relacionais de forma transparente ao usuário [Płuciennik-Psota 2012]. Os *frameworks* considerados mais relevantes serão discutidos a seguir.

O SQLAlchemy [SQLAlchemy, 2017] é um *toolkit* e *framework* para ORM, desenvolvido para a linguagem Python. Tem como principais características sua extensibilidade e o fato de ser um ORM independente, o que torna mais fácil adequá-lo às necessidades de um projeto [Gantan 2014]. Os projetos [Sandmand2, 2017], [Flask-Sqlacoden, 2017] e [Flask-RESTful, 2017] são extensões para SQLAlchemy que possibilitam a geração automática de serviços RESTFul e/ou modelos esperados pelo SQLAlchemy. Essas extensões apresentam como ponto forte prescindir de nenhuma configuração adicional para funcionar. Entretanto, apresentam algumas desvantagens: dispõe de pouquíssimas opções de personalização da API REST (*Sandman2*), a ocorrência de *overhead* ao processar o código gerado dinamicamente (*Sandman2* e *Sqlacodegen*) e a falta de suporte para bases de dados legadas (*Flask-RESTful*).

O Django [Django, 2017] é um *framework* web *full-stack*, também escrito em Python, que possui uma implementação própria da técnica de ORM. Além disso, disponibiliza uma ferramenta chamada *inspectdb*, que facilita a integração com bases de dados legadas a partir da geração automática de modelos introspectando um banco de dados existente. O Django ORM apresenta como desvantagens o fato de funcionar apenas integrado ao Django.

Além do SQLAlchemy e Django ORM, têm-se um outro *framework* desenvolvido para a linguagem Python, chamado SQLObject [SQLObject, 2017]. Apesar de possuir baixa curva de aprendizado, possui várias limitações em relação aos *frameworks* supracitados, limitando o seu uso a projetos de pequeno porte. O Ruby on Rails (RoR) [Rails Guides, 2017] é um *framework* de desenvolvimento web que utiliza a linguagem orientada a objetos Ruby. Uma das bibliotecas do RoR, o *ActiveRecord* é uma camada de ORM que é responsável pela interoperabilidade entre os banco de dados.

O Hibernate [Hibernate ORM, 2017] é *framework* de ORM escrito em linguagem Java, que abstrai o seu código SQL, toda a camada JDBC e SQL gerado em tempo de compilação. Tem como objetivo de diminuir a complexidade entre os programas desenvolvidos em Java que precisam trabalhar com banco de dados relacional.

Embora tenham sido apresentado algumas ferramentas que proveem flexibilidade de banco de dados, existem alguns desafios inerentes a integração com bases de dados legadas, como a integração dessas bases à serviços *RESTful* já existentes, cujas interfaces são diferentes da estrutura da base de dados. Nesse contexto, este trabalho propõe uma solução de software para facilitar essa integração, de forma que seja possível realizar a introspecção de uma base existente e o mapeamento para uma interface *RESTful* existente.

5. A Solução desenvolvida

Nesta seção será apresentado o DB2REST. A subseção 5.1 apresenta as informações e o diagrama da estrutural da solução. Em seguida, a subseção 5.2 elucida, de forma detalhada, como preencher cada campo do arquivo de configuração. Por fim, a subseção 5.3 apresenta os aspectos de implementação da ferramenta.

5.1 Arquitetura do DB2REST

A ferramenta apresentada neste artigo visa adaptar a interface de um *web services RESTful* — cujos serviços implementam regras de negócio que pressupõe a existência de uma estrutura de dados específica, como entidades e relacionamentos, sobre os quais os serviços realizam algum processamento — e uma base de dados legada, utilizando para isto um arquivo de especificação em formato JSON, contendo a descrição dos modelos de dados da base legada e do *web services RESTful* e de seus respectivos atributos e relacionamentos.

A Figura 4 apresenta o fluxo de execução do DB2REST. A ferramenta possui duas etapas de execução distintas: uma parte é executada offline (1 - 4) e a outra, online (5). A seguir, têm-se uma breve descrição de cada uma dessas etapas:

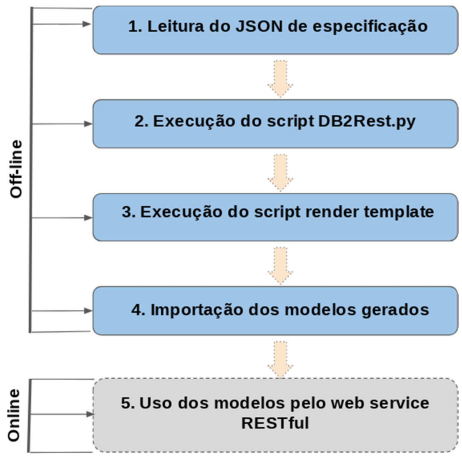

Figura 4. Fluxo de execução do DB2REST

1. **Leitura do Json de Especificação** — que será descrito detalhadamente a seguir— contendo as informações requeridas pela ferramenta;
2. **Execução do *script* db2rest.py** que é o mecanismo principal da ferramenta. Nele, são verificadas e validadas todas as informações do JSON de especificação na base de dados e a criação da estrutura do código que será gerado pela ferramenta;
3. **Execução do *script render_template***, que é um mecanismo auxiliar para geração de código a partir de um *template*;
4. **Importação dos modelos gerados**, consiste na configuração para uso dos modelos gerados pelo *web service RESTful*.
5. **Uso dos modelos gerados pelo *web service RESTful*** é a única etapa do modo de execução online. Os modelos gerados nas etapas anteriores são utilizados pelo servidor. Para cada modelo no arquivo de especificação é gerada uma classe, em linguagem Python, que é a representação daquela entidade no banco de dados e, ao mesmo tempo, realiza a adaptação entre a estrutura do banco de dados e a estrutura do *web service RESTful*.

O DB2REST é um sistema que adota o estilo arquitetural *Virtual Machines*. Este padrão é caracterizado pela separação de um sistema em subsistemas (camadas) que fornecem serviços à camada imediatamente acima ou abaixo desta [Buschmann et al. 1996].

A Figura 5 apresenta um diagrama de sequência do DB2REST, demonstrando o fluxo de controle entre os objetos durante a execução da ferramenta. O ator, que neste caso será o programador, inicia a ação ao executar o *script execute_db2rest*. O DB2REST executa a função *models_list()* que fará a leitura do arquivo de configuração e retorna uma lista de objetos do tipo *model helper*, provenientes de uma classe auxiliar criada a partir dos campos do JSON do arquivo de configuração e que permite, desta forma, tratar os dados lidos como objetos Python. Para cada objeto retornado nesta lista, serão feitas as validações de tabelas, colunas e relacionamentos na base de dados. Caso exista a definição de atributos derivados, as informações também serão verificadas na base de dados.

Após a validação dos dados informados no arquivo de configuração, caso não sejam identificadas inconsistências nos dados informados, o DB2REST cria a estrutura de cada um dos modelos a serem gerados. Em seguida, o DB2REST delega a renderização

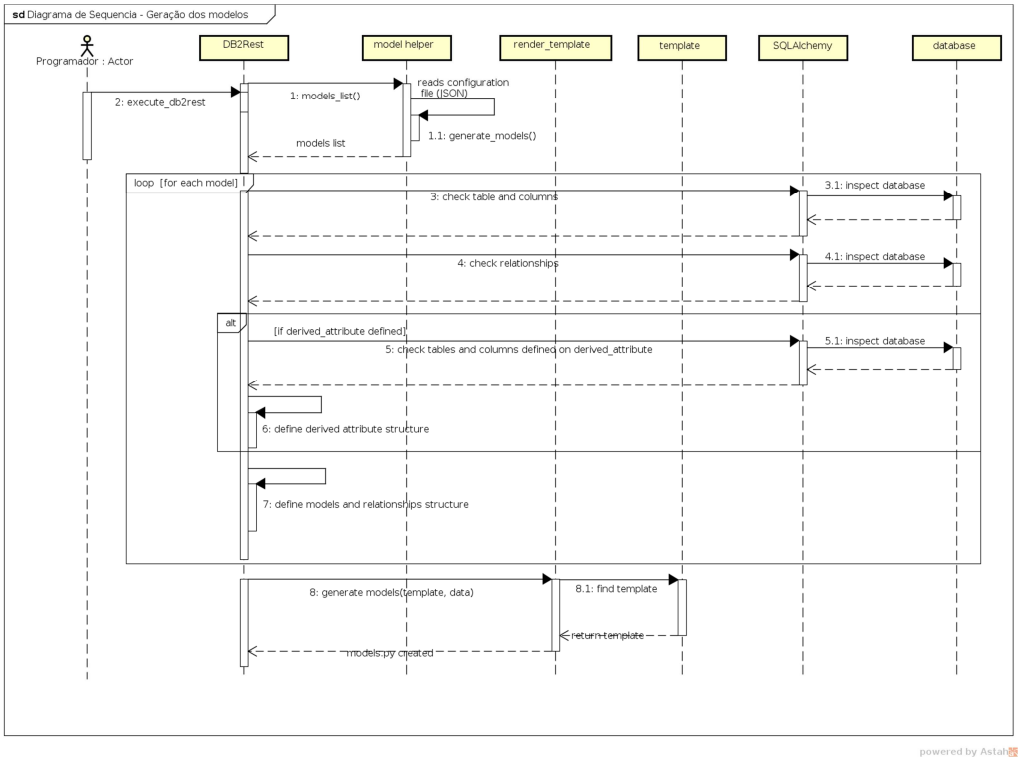

Figura 5. Diagrama de Sequência: geração dos modelos

dos modelos ao *script* auxiliar *render_template*, que invocará o *template* informado como estrutura que os modelos deverão ter. Ao final desta execução, o arquivo *models.py* será criado e estará pronto para uso, sendo necessário apenas importá-lo nos serviços.

5.2 Definições do Arquivo de Configuração

O arquivo de configuração em formato JSON deverá conter um registro para cada modelo a ser adaptado. Além disso, são necessárias informações sobre a entidade-alvo na base de dados legada, bem como outras informações sobre seus atributos, relacionamentos e outras propriedades. A seguir, serão descritas as propriedades do arquivo de configuração e uma breve descrição da sua utilidade. Os atributos com prefixo *rst* representam os parâmetros do *web service RESTful*, e aqueles iniciados com o prefixo *db* referem-se às informações da base de dados. A Listagem 1 apresenta um exemplo do arquivo de configuração esperado pelo DB2REST.

- ***__rst_model_name__***: deverá conter o nome do modelo de dados utilizado pelo *web service RESTful*, que neste caso representa um recurso.
- ***__db_table_name*** : deverá conter o nome da entidade correspondente na base de dados legada.
- ***attributes***: deverá conter uma lista de atributos a serem mapeados. Para cada atributo, têm-se um objeto JSON que contém informações sobre o respectivo atributo. Cada chave dentro desse objeto representa uma propriedade que será aplicada aos modelos gerados. A definição desta estrutura para cada atributo permite a inclusão futura de novas informações a serem mapeadas.

```
[{
"_rst_model_name_":"Postagem", "_db_table_name__":"post",
"attributes": [
      {"rst_attribute_name":"id_postagem", "db_column_table":"id","db_primary_key":"True"},
      {"rst_attribute_name":"titulo", "db_column_table":"title"},
      {"rst_attribute_name":"data_postagem", "db_column_table":"date"},
      {"rst_attribute_name":"hora_postagem", "db_column_table":"time"}],
"derived_attributes":[{
    "rst_property_name":"detalhes_categoria", "db_columns":"category.name",
    "db_clause_where":"id|1", "db_rows_many":"False"}],
"relationships": [{
    "type":"M2O", "rst_referencing_name":"categoria", "rst_referenced_model":"Categoria",
    "db_referenced_table":"category", "db_referenced_table_pk":"category.id",
    "db_referencing_table_fk":"category", "rst_referenced_backref":"postagens"}]
}]
```

Listing 1: Exemplo do arquivo de configuração em formato JSON

- ***derived_attributes***: deverá conter uma lista de objetos JSON para cada atributo derivado a ser gerado. Esta chave tem como finalidade obter informações sobre atributos presentes em outras entidades, e que serão mapeados como atributo do modelo, mesmo que não exista relacionamento entre eles na base de dados. Sua utilização é indicada quando alguma informação requerida pelo *web service RESTful* está presente em outra tabela da base de dados.
- ***relationships***: deverá conter uma lista de objetos JSON, com um registro para cada relacionamento a ser mapeado. Esta chave deverá ser preenchida considerando a perspectiva do modelo atual em relação ao modelo-alvo.

5.3 Aspectos de Implementação

O DB2REST foi desenvolvido utilizando como base o SQLAlchemy, um *framework* que implementa a técnica de ORM e tem como principais vantagens o fato de ser bastante flexível e independente. No contexto do DB2REST, o SQLAlchemy é utilizado para realizar essa introspecção na base de dados legada e, desta forma, realizar a integração com os modelos de dados do *web service RESTful*.

Após a criação do arquivo de configuração, o DB2REST realiza a leitura do arquivo e valida as informações na base de dados, checando os nomes das tabela, atributos e relacionamentos informados. Caso não sejam encontradas inconsistências no arquivo de especificação, a ferramenta gera um conjunto de dados sobre os modelos que serão delegados a um *template*. O *template* utilizado pelo DB2REST é desenvolvido usando o *Jinja2*, uma linguagem de *templates* para a linguagem Python.

As classes geradas herdam de uma classe *Base*, provida pelo DB2REST, e configurada de acordo com a especificação de API do SQLAlchemy para mapear uma base de dados existente. Cada classe é uma representação na linguagem Python, para cada entidade na base de dados legada. Estas classes são interpretadas pelo SQLAlchemy como parte da aplicação. A Listagem 2 apresenta um exemplo de classe gerada automaticamente pela ferramenta.

A Listagem 3 demonstra como os modelos gerados pelo DB2REST são utilizados em um *web service RESTful*. O serviço, disponível na URL *postagens* utiliza o modelo Postagem, apresentado na Listagem 2, a partir da importação do arquivo *models* do

```
1  fromDB2Rest.dbimport   Base
2  fromsqlalchemyimport   Column,Integer,String,ForeignKey
3  fromsqlalchemy.ormimport   relationship
4  fromsqlalchemy.ext.hybridimport   hybrid_property
5  fromDB2Restimport   queries
6
7  classPostagem (Base):
8      __tablename__="post"
9      id_postagem=Column('id',Integer,primary_key=True)
10     titulo=Column('title')
11     data_postagem=Column('date')
12     hora_postagem=Column('time')
13
14     ##Relationships##
15     categoria_id=Column('category',Integer, ForeignKey('category.id'))
16     categoria=relationship('Categoria', back_populates='postagens', lazy='joined')
17
18      @hybrid_property
19     def detalhes_categoria(self):
20         return queries.get_table_derived_attributes(table_name='category', column_name='name',
21             clause_where_attribute='id',clause_where_value=1, many=False)
```

Listing 2: Classe gerada automaticamente pelo DB2REST

módulo DB2REST (linha 5). O serviço retorna uma lista dos registros de postagens, utilizando o formato padrão da API do SQLAlchemy para fazer a consulta na base de dados. Ao acessar o modelo Postagem, os métodos assessores farão o mapeamento das entidades e atributos do modelo.

```
1  fromflaskimport   Blueprint, jsonify, request
2  importdatetime
3  fromDB2Restimport   models
4
5  postagem=Blueprint("postagem",__name__)
6
7  @postagem.route('/postagens')
8  def listar_postagens():
9      postagens=models.Postagem.query.all()
10     return jsonify(result=[dict(id=postagem.id_postagem,titulo=postagem.titulo,
11             data=postagem.data_postagem,hora=postagem.hora_postagem,
12             categoria=postagem.categoria,detalhes=postagem.detalhes_categoria)
13             for postagem in postagens])
```

Listing 3: Serviço RESTful que utiliza o mecanismo DB2REST

6. Avaliação e Resultados

Nesta seção serão apresentadas informações relativas aos experimentos realizados com usuários, com o objetivo de testar e avaliar a solução proposta nesse trabalho.

6.1 Estudo de Caso: Emile Server

O Emile é sistema *open-source* para comunicação acadêmica, desenvolvido no âmbito do GSORT (Grupo de Sistemas Distribuídos, Otimização, Redes e Tempo Real), grupo de pesquisa do IFBA (Instituto Federal da Bahia). O sistema foi projetado para permitir a aproximação entre a academia e os seus diversos atores através de um aplicativo móvel.

O sistema é composto pelo aplicativo móvel propriamente dito e de um *web service RESTful*, o *Emile Server*, que provê dados para consumo no aplicativo. No início do projeto, a equipe de desenvolvimento não obteve acesso à base de dados legada devido a questões burocráticas e procedimentos de segurança da informação adotados pela instituição. Por isso, os desenvolvedores criaram uma nova modelagem de dados, em conformidade com as entidades identificadas pela equipe naquele período.

6.2 Execução dos Experimentos

Para avaliar a solução proposta neste trabalho, foi conduzido um estudo com dois participantes (P1 e P2), ambos alunos do curso de Análise e Desenvolvimento de Sistemas do Instituto Federal da Bahia (IFBA) - Campus Salvador. A execução dos experimentos foi precedida de um mini-treinamento, com 2 horas de duração, no qual os participantes aprenderam sobre as tecnologias que seriam utilizadas no experimento, visando nivelar os conhecimentos sobre os conteúdos abordados. A execução do estudo ocorreu nos dias 7, 8 e 14 de março de 2018, no laboratório do GSORT (Grupo de Sistemas Distribuídos, Otimização, Redes e Tempo Real). Os experimentos tiveram duração mínima de 1 hora e máxima de 3 horas.

Os experimentos foram realizados utilizando o código-fonte do *Emile Server*. Para isso, um conjunto de modelos e serviços foram escolhidos como base para utilização em dois cenários diferentes. Ambos os participantes foram submetidos aos experimentos I e II, sendo que a ordem de execução destes se deu de forma inversa. Isto é, um dos participantes começou pelo experimento I e o outro pelo experimento II. No primeiro experimento, foi disponibilizada uma versão do código-fonte sem a implementação dos modelos e serviços. No segundo experimento, foi disponibilizada uma versão contendo as implementações dos modelos e serviços, porém com uma base de dados diferente. Os participantes utilizaram o DB2REST para a integração.

6.3 Resultados e Discussão

O presente estudo visa analisar e avaliar os resultados obtidos na execução dos experimentos I e II. Os códigos desenvolvidos pelos participantes dos experimentos foram enviados para o repositório de testes para análise e extração de métricas de software. A Tabela 2 apresenta as hipóteses que serão analisadas com base nos resultados obtidos.

Tabela 2. Hipóteses Avaliadas no Estudo

Hipóteses	Métricas	Resultados Esperados
H1	Produtividade: tempo	$T_{DB2REST} < T_{Flask}$
H2	Complexidade: LOC	$LOC_{DB2REST} < LOC_{Flask}$
H3	Densidade de bugs: bugs/LOC	$DB_{DB2REST} < DB_{Flask}$

Para analisar H1, o tempo de execução gasto para desenvolver cada etapa foi computado no estudo. A Figura 6 apresenta um gráfico com as informações de tempo decorrido em cada experimento. Os pontos no gráfico representam os *check points* de cada experimento, divididos em duas etapas. Observa-se que os participantes demandaram menos tempo na execução das etapas relativas ao experimento II em relação ao experimento I. Tal característica deve-se ao fato de que, no caso do experimento II, não foi necessário

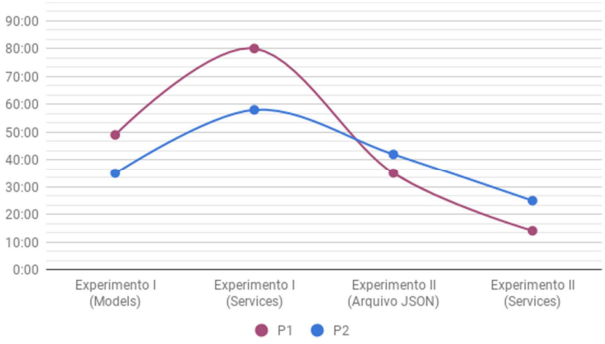

Figura 6. H1: Produtividade em função do tempo de desenvolvimento

implementar a lógica de negócio necessária para o funcionamento dos serviços, bastando apenas mudar o módulo de importação.

Considerando os dados demonstrados no gráfico da Figura 6, e com base nas análises realizadas, pode-se considerar que H1 é verdadeira. A condição $T_{DB2REST} < T_{Flask}$ é válida, uma vez que integrar um *web service RESTful* a uma base de dados legada consome menos tempo, e portanto, é mais produtivo do que desenvolver um novo serviço.

A avaliação de H2 compreende o uso da métrica *LOC* (*Lines of Code*) para analisar a complexidade envolvida em: i) reimplementar os serviços RESTful para uma nova base de dados ii) integrar serviços existentes a uma base de dados legada. Para melhor compreensão dos dados analisados, os modelos e serviços desenvolvidos nos experimentos foram separados em dois gráficos, sendo um deles apenas para os modelos e o outro apenas para os serviços escolhidos para a execução dos experimentos.

A Figura 7 apresenta os gráficos relativos ao quantitativo de linhas escritas pelos participantes para cada modelo e serviço, acompanhados de uma coluna com a quantidade total de linhas para cada experimento/participante. No primeiro gráfico da Figura 7, referente aos modelos, observa-se que a quantidade de linhas necessárias no experimento II, utilizando o DB2REST, é o dobro da quantidade de linhas requeridas no experimento I. Entretanto, um fator preponderante nesta análise refere-se à estrutura do arquivo de configuração que pela própria definição da notação JSON, acrescenta linhas ao arquivo e, por sua vez, influenciam diretamente na quantidade total do *LOC*. O segundo gráfico disponível na Figura 7 demonstra o *LOC* apenas para os serviços implementados na execução dos experimentos. Nota-se que a quantidade de linhas necessárias no experimento II foi consideravelmente menor do que a quantidade requerida no experimento I, representando a metade da quantidade de linhas utilizadas no primeiro cenário.

A partir da análise dos gráficos disponíveis nas Figura 7, conclui-se que H2 é falsa. O LOC dos experimentos indicam que $LOC_{DB2REST} < LOC_{Flask}$ não é verdadeira. Dessa forma, utilizar o DB2REST para integrar um *web service RESTful* a uma base de dados legada requer mais linhas – e portanto, gera sistemas mais complexos – do que desenvolver um novo *web service RESTful*. Contudo, é importante salientar que o presente estudo identificou que a métrica utilizada não é a mais adequada para avaliar os cenários disponíveis, devido a discrepância entre os tipos de código analisados.

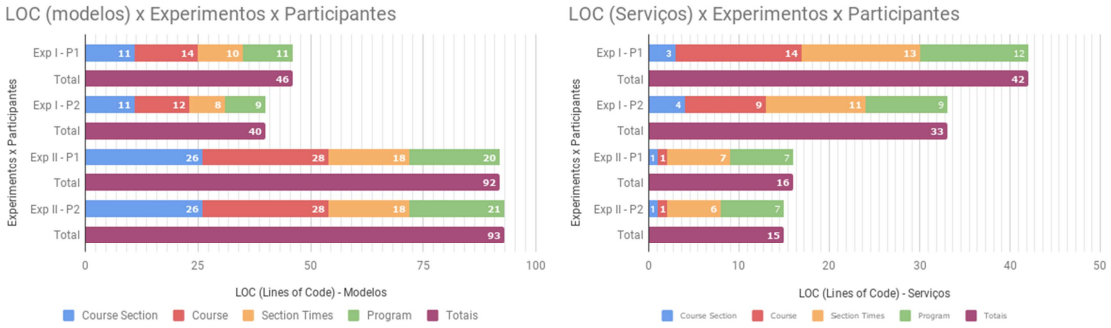

Figura 7. H2: Análise de Complexidade Modelos/Serviços

Para fins de avaliação da H3, foi realizada a extração da densidade de *bugs* (DB) dos experimentos. A Figura 8 apresenta um gráfico com os dados a DB para cada uma das etapas dos experimentos, para cada participante. Nele, observa-se que a densidade de *bugs* no experimento II é consideravelmente menor em relação a DB do experimento I. Tal característica se deve ao fato de que no experimento II foi necessário apenas alterar os módulos de importação para utilizar o DB2REST, o que implica numa densidade de *bugs* menor — ou inexistente, como foi o caso de P2 —, pois os serviços existentes já foram testados e, portanto, são menos suscetíveis a ocorrência de erros.

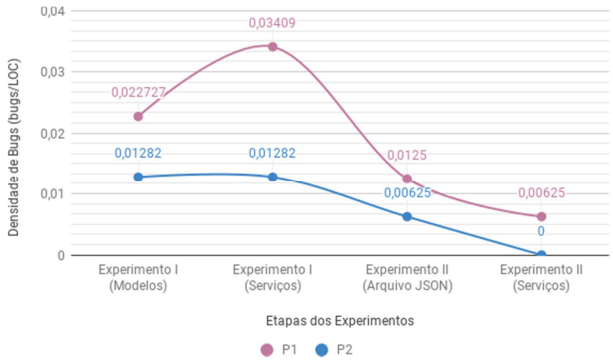

Figura 8. H3: Análise da Densidade de Bugs

De acordo com o gráfico apresentado na Figura 8, e com base no que foi analisado e exposto anteriormente, conclui-se que H3 é verdadeira. A condição $DB_{DB2REST} < DB_{Flask}$ é válida, visto que que utilizar o DB2REST para integrar um *web service RESTful* a uma base de dados legada gera sistemas com menor densidade de *bugs* do que implementar um novo *web service RESTful*.

7. Trabalhos Futuros

Durante o desenvolvimento da solução proposta, bem como durante a avaliação dos resultados, observou-se possibilidades de melhorias na implementação visando evoluir o DB2REST. Nesse contexto, destacam-se como trabalhos futuros:

- Criar uma interface de usuário, contendo as informações de tabelas e colunas da base de dados legada, bem como dos modelos utilizados no *web service RESTful*.

Os usuários poderiam apenas relacionar as tabelas-modelos e colunas-atributos com base nas informações reunidas na interface gráfica. Tal funcionalidade reduziria consideravelmente o tempo necessário e a ocorrência de erros cometidos no preenchimento do JSON de configuração de forma manual, tornando o processo mais intuitivo.

- Utilizar outra métrica para avaliar a complexidade dos experimentos (com e sem DB2REST). No processo de avaliação e resultados, constatou-se que a métrica LOC (*Lines of Code*) não é a mais adequada para avaliar a complexidade, uma vez que o JSON de configuração e os modelos em Python possuem semânticas diferentes.
- Avaliar a possibilidade de utilizar o DB2REST juntamente com outros *frameworks* de desenvolvimento de *web services RESTful* que tenham suporte para SQLAlchemy.
- Implementar mais funcionalidades ao módulo *queries*, de forma que outras cláusulas SQL sejam suportadas. Atualmente, o módulo realiza apenas consultas baseadas na cláusula *where* para atribuição de valores em atributos derivados de outras tabelas.

8. Conclusão

Existem diversas situações em que a integração entre sistemas computacionais é requerida. Entretanto, realizar essa integração perpassa por vários desafios, principalmente considerando-se o uso de sistemas obsoletos, a existência de *hardwares* antigos, falta de documentação adequada dos sistemas, inconsistência de dados etc. Um caso particular desses desafios é a integração entre *web services RESTful* e bases de dados legadas.

Nesse contexto, o presente trabalho propôs a implementação e avaliação de uma solução para facilitar a integração entre esses sistemas. O DB2REST estabelece essa integração através da adaptação entre a camada de serviço e a camada de persistência, de forma que essas interfaces diferentes passam a se comunicar sem a necessidade de modificações estruturais entre eles.

A avaliação da solução proposta foi realizada a partir da análise de dois cenários distintos: no primeiro, os participantes desenvolveram um novo *web service RESTful* em conformidade com a base de dados fornecida, a partir do seu modelo relacional. No segundo, os participantes utilizaram o DB2REST para integrar uma base de dados legada a um *web service RESTful* existente. Observou-se um ganho de produtividade significativo ao realizar a integração entre os sistemas existentes, bem como uma redução da ocorrência de *bugs* em relação ao experimento sem o DB2REST.

O desenvolvimento do presente trabalho possibilitou uma análise sobre como a integração de sistemas computacionais podem trazer resultados satisfatórios. A integração entre um *web services RESTful* e base de dados legada é uma alternativa à criação de novos softwares, medida que viabiliza o reuso de soluções e, consequentemente, a redução nos custos de produção de novos softwares dentro de um mesmo domínio de aplicação.

Referências

[Bauer and King 2005]Bauer, C. and King, G. (2005). *Hibernate in Action: Practical Object/Relational Mapping*. Manning Publications.

[Buschmann et al. 1996]Buschmann, F. et al. (1996). *Pattern-Oriented SoftwareArchitecture - Volume 1: A System of Patterns*. Wiley Publishing.

[Coelho and Sartorelli 2004]Coelho, C. and Sartorelli, R. (2004). Persistência de objetos via mapeamento objeto-relacional. *Bacharelado em Sistemas de Informação, São Paulo*.

[Cordeiro 2011]Cordeiro, J. (2011). Estudo comparativo entre os frameworks de mapeamento objeto-relacional hibernate e toplink. *Especialização em Desenvolvimento de Sistemas para Web - Universidade Estadual de Maringá*.

[Django, 2017]Django, ((Acesso em: Junho 26, 2017)). The web framework for perfectionists with deadlines. https://docs.djangoproject.com/en/1.11/.

[Django REST Framework, 2017]Django REST Framework, ((Acesso em: Julho 17, 2017)). Django rest framework is a powerful and flexible toolkit for building web apis. http://www.django-rest-framework.org/.

[Flask-RESTful, 2017]Flask-RESTful, ((Acesso em: Julho 17, 2017)). An extension for flask. https://flask-restful.readthedocs.io/en/0.3.5/.

[Flask-Sqlacoden, 2017]Flask-Sqlacoden, ((Acesso em: Julho 17, 2017)). Automatic model code generator for sqlalchemy with flask support. https://github.com/ksindi/flask-sqlacodegen.

[Galvão et al. 2009]Galvão, A. et al. (2009). Alternatives development software, without cost, for micro and small enterprises. *Revista ADMpg Gestão Estratégica*, 2(2):119–123.

[Gantan 2014]Gantan, X. (2014). Python's sqlalchemy vs other orms. Acesso em: Junho 15, 2017.

[Hamad et al. 2010]Hamad, H., Saad, M., and Abed, R. (2010). Performance evaluation of restful web services for mobile devices. *International Arab Journal of e-Technology*, 1(3).

[Hibernate ORM, 2017]Hibernate ORM, ((Acesso em: Junho 17, 2017)). More than an orm, discover the hibernate galaxy. http://hibernate.org/orm/.

[Joshi and Kukreti 2014]Joshi, A. and Kukreti, S. (2014). Object relational mapping in comparison to traditional data access techniques. *International Journal of Scientific Engineering Research*, 5.

[Josuttis 2007]Josuttis, N. (2007). *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc.

[Lapolli 2003]Lapolli, P. C. (2003). Implantação de sistemas de informações gerenciais em ambientes educacionais. *Dissertação de Mestrado, Florianópolis-SC*.

[Pinto and Braga 2005]Pinto, H. L. M. and Braga, J. L. (2005). Sistemas legados e as novas tecnologias: técnicas de integração e estudo de caso. *Informática Pública*, 7:47–69.

[Płuciennik-Psota 2012]Płuciennik-Psota, E. (2012). Object relational interfaces survey. *Studia Informatica*, 33(2A):299–310.

[Rails Guides, 2017]Rails Guides, ((Acesso em: Junho 07, 2017)). Web application framework written in ruby. http://guides.rubyonrails.org/.

[Restlet, 2017]Restlet, ((Acesso em: Julho 29, 2017)). tool to generate restful apis. https://restlet.com/documentation/.

[Richardson and Ruby 2007]Richardson, L. and Ruby, S. (2007). *Restful Web Services*. O'Reilly, first edition.

[Sandmand2, 2017]Sandmand2, ((Acesso em: Julho 17, 2017)). Automatically generate a restful api for your legacy database. http://sandman2.readthedocs.io/en/latest/.

[Sinatra, 2017]Sinatra, ((Acesso em: Julho 29, 2017)). A free and open source software web application library and domain-specific language written in ruby. http://www.sinatrarb.com/documentation.html.

[Sousa 2015]Sousa, F. (2015). Criação de framework rest/hateoas open source para desenvolvimento de apis em node.js. *Mestrado Integrado em Engenharia Informática e Computação*.

[Spark, 2017]Spark, ((Acesso em: Julho 29, 2017)). A micro framework for creating web applications in kotlin and java 8 with minimal effort. http://sparkjava.com/.

[SQLAlchemy, 2017]SQLAlchemy, ((Acesso em: Junho 06, 2017)). The database toolkit for python. https://www.sqlalchemy.org/.

[SQLObject, 2017]SQLObject, ((Acesso em: Junho 07, 2017)). An object-relational mapper for python programming language. http://sqlobject.org/SQLObject.html.

[Tobaldini 2008]Tobaldini, R. G. (2008). Aplicação do estilo arquitetural rest a um sistema de congressos. *Bacharelado em Ciência da Computação, Florianópolis-SC*.