DB2REST - Um *Middleware* para Integração entre Bases de Dados Legadas e *Web Services RESTful*

Bruno Araújo de Oliveira
Instituto Federal da Bahia
Salvador, Bahia, Brasil
brunoaraujo@ifba.edu.br

Sandro Santos Andrade
Instituto Federal da Bahia
Salvador, Bahia, Brasil
sandroandrade@ifba.edu.br

***Resumo*—Embora a integração de sistemas computacionais seja uma alternativa ao desenvolvimento de novas soluções de software, uma vez que isso pode significar redução de tempo de desenvolvimento e utilização de recursos das organizações, sua realização muitas vezes 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, que apresentam estruturas de dados divergentes. Neste contexto, este trabalho visa apresentar o DB2REST, um componente de software que auxilia na integração entre servidores web RESTful e bases de dados legadas. O *middleware* tem como entrada informações sobre o *backend* de banco de dados utilizado e um arquivo de configuração contendo os dados utilizados para realizar o mapeamentos das entidades e modelos que serão adaptados. Para avaliar a solução foram realizados experimentos com e sem o DB2REST, visando analisar a produtividade, complexidade e densidade de *bugs* em cada cenário.**

Palavras-chave*—*Legacy Database, RESTful, RESTFul-based, System integration

I. 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 [1]. 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 [2]. 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 [3]. Alguns casos em que comumente existe a 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 [3]: 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 avaliação 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 [1], 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 II aborda o problema da integração entre *web services RESTful* e bases de dados legadas. A Seção III 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 IV apresenta os trabalhos relacionados à solução proposta neste trabalho. A Seção V apresenta a solução proposta e as tecnologias utilizadas no seu desenvolvimento.

II. 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. Tal funcionalidade muitas vezes atende às necessidades das organizações que possuem dados de valor para o negócio armazenados ao longo dos anos, e portanto, não faria sentido implementar um novo modelo de dados.

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. Entretanto, tais soluções não prevêem os cenários em que já existe uma API *RESTful*, que foi construída a partir de uma configuração de dados específica, divergente da estrutura da base de dados legada. Esses cenários serão abordados a seguir.

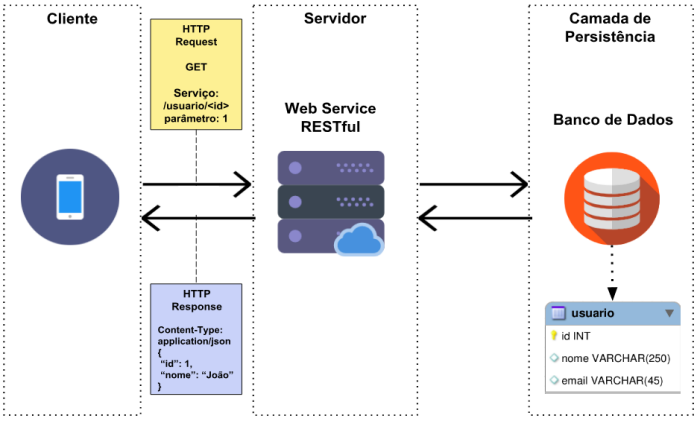


Figura 1: Cenário em que não há necessidade de integração

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.

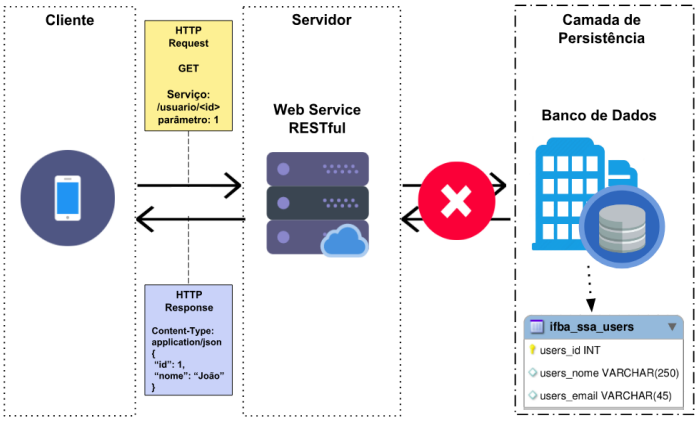


Figura 2: Cenário em que há necessidade de integração

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,

[1]JSON (*JavaScript Object Notation*) - Uma formatação leve de troca de dados, compatível com diversas linguagens de programação. Disponível em: https://www.json.org

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.

III. Referencial Bibliográfico

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

A. 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 [4]. 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 [4] [5].

Tabela I: 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 [4]. O SOAP (*Simple Object Access Protocol*) foi o primeiro *Web Service* real a ser desenvolvido [4]. Fornece um mecanismo simples e leve de troca de informações estruturadas em um ambiente distribuído e descentralizado, usando XML[2].

O REST (*Representational State Transfer*) é um estilo arquitetural que define uma série de princípios para a construção de WS, baseado no protocolo HTTP [5]. Os sistemas que fornecem uma API definida pela aplicação desse estilo arquitetural recebem o nome de *RESTFul* [5]. A Tabela I apresenta os métodos do HTTP e as respectivas operações que podem ser aplicadas a algum recurso disponível na API.

A Figura 3 representa o ciclo de uma requisição definido 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.

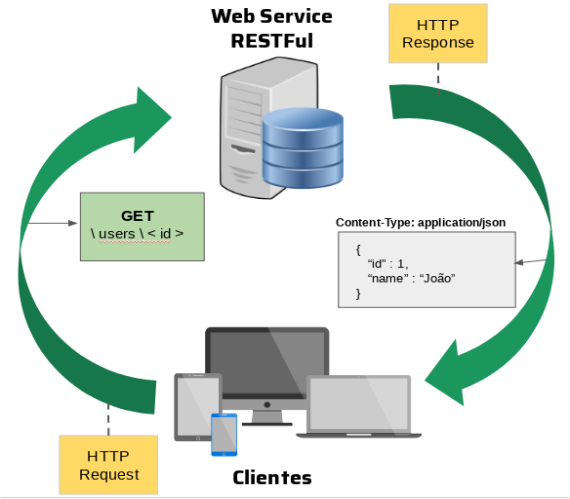


Figura 3: Ciclo de requisição em um Web Service RESTFul

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 [5][6];
- ***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 [5][6];
- ***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 [5][6];
- ***Representation***: as representações são, basicamente, a forma de apresentação de um recurso, em um formato

[2]XML (*eXtensible Markup Language* - Uma linguagem de marcação recomendada pela W3C para a criação de documentos com dados organizados hierarquicamente. Disponível em: https://www.w3.org/XML

que tenha alguma utilidade para o cliente, como em JSON, XML etc [5][6];

- ***Uniform Interface***: como o REST é baseado nos métodos do protocolo HTTP, as aplicações desenvolvidas seguindo este estilo arquitetural terão a mesma interface [5][6].

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 [5][7].

B. 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* [8] 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 [9].

O *Flask-RESTful* [10] 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 [9].

O *Restlet* [11] 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 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 [9].

O *Spark* [12] , 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) [9].

O *Sinatra* [13] 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 [9].

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.

C. 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" [14].

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 [14][15]. O uso da técnica promove o desacoplamento entre o código da aplicação e a base de dados utilizada [16].

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 [14]. 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 [16][17].

Tabela II: Posicionamento entre Frameworks para persistência.

Frameworks e Features	Mapeamento de Banco de Dados	Geração de Códigos de Modelos	Geração de API RESTful
SQLAlchemy	✓	⚠	⚠
Django ORM	✓	✓	⚠
SQLObject	✓	✗	✗
ActiveRecord	✓	✗	✗
Hibernate	✓	✓	✗

Legenda: ✓- Suportado Nativamente | ✗- Sem Suporte Nativo | ⚠ - Suportado por Extensões de Terceiros

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.

IV. 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 [15], 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 [14][18].

O SQLAlchemy [19] é um *toolkit* e *framework* para ORM, desenvolvido para a linguagem Python, que se tornou bastante popular devido a sua fácil integração com diversos bancos de dados amplamente utilizados no mercado. Tem como principais características sua extensibilidade, uma vez que é possível agregar novas funcionalidades a partir de extensões desenvolvidas pelos usuários e o fato de ser um ORM independente, o que torna mais fácil adequá-lo às necessidades de um projeto [20]. Entretanto, algumas funcionalidades que poderiam ser úteis para ampliar a flexibilidade de banco de dados em sistemas legados não possuem suporte nativo. Dentre elas, a geração automática de código (o *framework* realiza apenas geração de classes através de reflexão computacional) e criação de API RESTful, são suportadas apenas através de extensões adicionais. Estas extensões serão descritas a seguir.

O *Sandman2* [21] é uma extensão para SQLAlchemy que possibilita a geração automática de serviços RESTFul a partir do *schema* de base de dados. A aplicação recebe como parâmetro o *path* do banco de dados, usuário e senha, e gera uma interface administrativa simples, na plataforma web, contendo serviços RESTful que realizam operações de CRUD (*create, read, update e delete*) para cada tabela do banco de dados informado. A extensão apresenta como ponto forte prescindir de nenhuma configuração adicional para funcionar. Entretanto, as desvantagens estão no fato de existirem pouquíssimas opções de personalização da API REST e a ocorrência de *overhead*, pois o código é gerado dinamicamente e processado a cada requisição.

O Sqlacodegen [22] é uma biblioteca Python, que permite gerar códigos de modelos no formato utilizado pelo SQLAlchemy a partir do *schema* de um banco de dados. A extensão é bastante útil para projetos em estado inicial que pressupõe o uso de banco de dados legado, pois elimina a necessidade de codificação para uso do *framework* de ORM. Entretanto, apesar de gerar os modelos corretamente, a extensão exibe o código gerado no terminal, sendo necessário copiá-lo para utilizá-lo no projeto.

O Flask-RESTful [10] é uma extensão baseada em SQLAlchemy que visa facilitar a criação de APIs *RESTful*. A extensão visa encapsular a criação dos serviços através da definição de *endpoints* para as entidades, denominadas *resources*, sendo necessário implementar apenas as funções correspondentes aos métodos HTTP suportados por aquele recurso. Entretanto, apesar de auxiliar no processo de criação de APIs como proposto, a extensão não tem suporte para base de dados legada.

O Django [23] é um *framework* web *full-stack*, também escrito em Python, que possui uma implementação própria da técnica de ORM que chama a atenção pela sua eficiência e facilidade de uso. Além disso, possui uma ferramenta chamada *inspectdb*, que facilita a integração com bases de dado legadas a partir da geração automática de modelos introspectando um banco de dados existente [23]. O Django ORM apresenta como desvantagens o fato de funcionar apenas integrado ao Django *Framework*. Além disso, a geração de serviços *RESTful* não tem suporte nativo, sendo necessário o uso da aplicação Django REST Framework [8], que possibilita a criação de serviços *RESTful* a partir dos modelos definidos no Django *Framework*. A aplicação é de fácil uso e dispõe de várias ferramentas que auxiliam na criação de APIs. Por outro lado, por ser um *framework* que tem como dependência o Django *Framework*, fica em desvantagem em relação ao SQLAlchemy — que neste caso, é um *framework* ORM independente — e as extensões supracitadas.

Além do SQLAlchemy e Django ORM, têm-se um outro *framework* desenvolvido para a linguagem Python, chamado SQLObject [24]. Tem se tornado popular na comunidade Python devido sua semelhança com o padrão *ActiveRecord* utilizado pelo *Ruby on Rails* [25]. O mapeamento dos objetos ocorre de forma muito similar ao SQLAlchemy e Django ORM, dos quais as tabelas são mapeadas como classes, as linhas como instâncias e as colunas como atributos Python. Apesar de possuir baixa curva de aprendizado, possui uma série de limitações em relação aos *frameworks* supracitados, pois além de não possuir funcionalidade de geração de código

e serviços, não tem suporte para pacotes de terceiros, o que limita a ferramenta a projetos de pequeno porte.

O Ruby on Rails (RoR) [25] é um *framework* de desenvolvimento web que utiliza a linguagem orientada a objetos Ruby. No RoR, cada biblioteca realiza uma tarefa especializada em uma parte do padrão MVC (*active record, active view e active controller*). A biblioteca *ActiveRecord* [25] é uma camada de ORM utilizado pelo RoR que mapeia objetos entre banco de dados e a linguagem Ruby, sendo responsável pela interoperabilidade entre os banco de dados. Por ser uma implementação do padrão *Active Record*, seu funcionamento básico é de fácil compreensão. Possui sistema de migração eficiente e permite o uso da biblioteca de forma desassociada do *framework*. Como ponto negativo, podemos citar falta de documentação mais abrangente e a falta da funcionalidade de geração automática de código e de serviços.

O Hibernate [26] 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. O *framework* foi desenvolvido com o objetivo de diminuir a complexidade entre os programas desenvolvidos em Java que precisam trabalhar com banco de dados relacional. Possui uma API bastante abrangente e dispõe de uma interface para gerenciamento de sessões de banco de dados — útil nos casos em que a aplicação consulta várias bases de dados—. Apesar de gerar classes e código, o processo é feito pela interface gráfica, manualmente, através de um procedimento bastante longo [27]. Requer várias configurações manuais e apresenta menor flexibilidade em relação às outras ferramentas citadas [20].

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

V. A Solução desenvolvida

Nesta seção será apresentado o DB2REST[3], um *middleware* para integração entre bases de dados legadas e *web services RESTful*. A subseção V-A apresenta as informações e diagramas da arquitetura da ferramenta. Em seguida, a subseção V-B elucida, de forma detalhada, como preencher cada campo do arquivo de configuração. Por fim, a subseção V-C apresenta os aspectos de implementação da ferramenta.

A. 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.

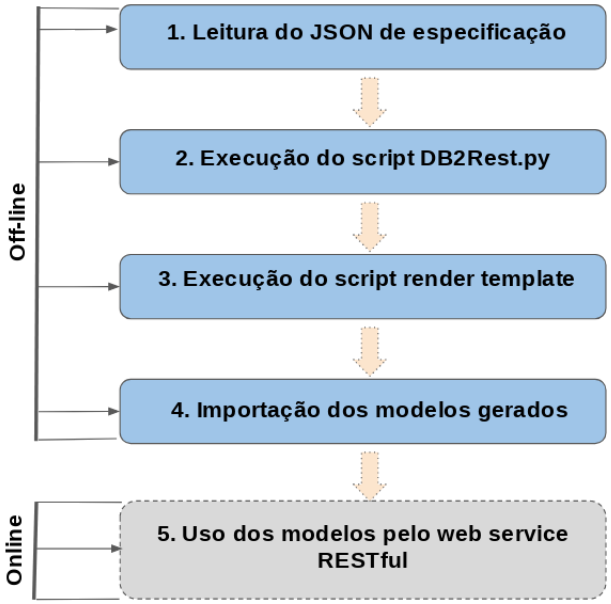


Figura 4: Fluxo de execução do DB2REST

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:

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*, contendo os dados estruturados na etapa anterior;
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

[3]DB2REST é um software livre, sob a licença GPLv3. O código-fonte e a documentação da ferramenta estão disponíveis no link: https://github.com/oliveirabrunoa/db2rest

de um sistema em subsistemas (camadas) que fornecem serviços à camada imediatamente acima ou abaixo desta [28].

A Figura 5 apresenta o diagrama de implantação do DB2REST, demonstrando as operações que ocorrem entre os principais módulos do sistema. Os clientes *Browser* e Aplicativo móvel podem enviar requisições HTTP ao invocar um serviço no servidor *web RESTful*. Este, por sua vez, realiza uma chamada de procedimento para o DB2REST por meio da invocação dos modelos gerados pela ferramenta na etapa *offiline*. Esses modelos fazem uma chamada de procedimento para o SQLAlchemy, que é responsável pela manipulação da base de dados legada. Ao final da operação associada ao serviço solicitado, o cliente recebe uma Resposta HTTP com o resultado da operação.

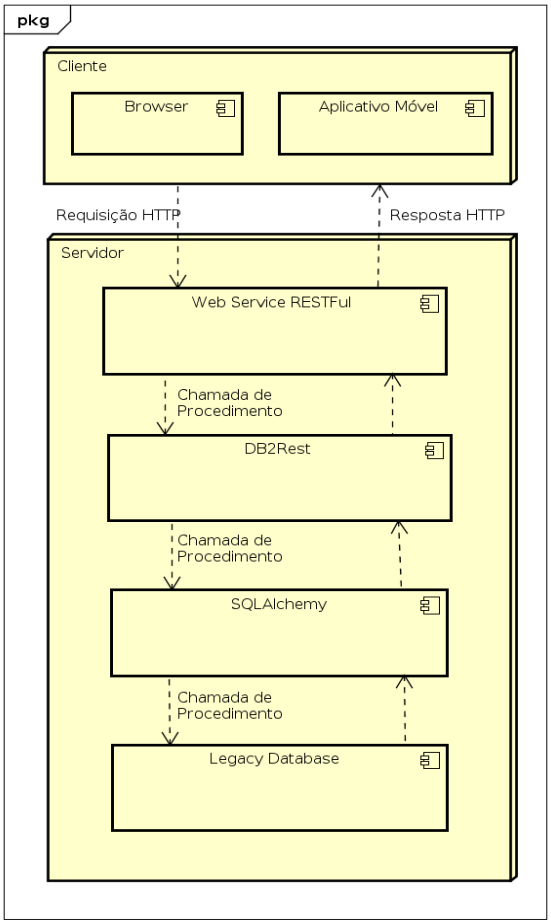


Figura 5: Diagrama de Implantação do DB2REST

A Figura 6 apresenta uma visão estrutural detalhada do DB2REST, demonstrando os componentes e conectores que compõem a ferramenta, sendo:

- **Cliente**: são os dispositivos que acessam a API do *web service RESTful*.
- ***Web Service RESTful***: o servidor que implementa o estilo arquitetural REST e fornece a API utilizada pelos clientes. É importante ressaltar que este componente foi desenvolvido considerando uma estrutura de dados — entidades, atributos e relacionamentos — que diverge da base de dados legada.

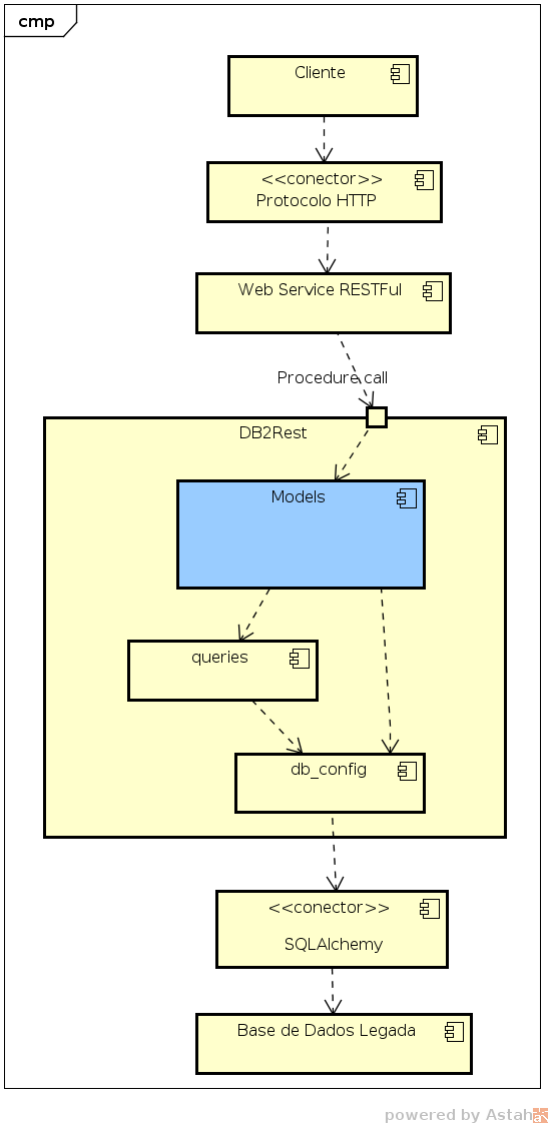


Figura 6: Visão Estrutural do DB2REST

- **DB2REST**: compreende toda a estrutura da ferramenta apresentada neste trabalho. O DB2REST atua no sentido de resolver as divergências entre a estrutura de dados do *web service RESTful*. e a base de dados legada. O componente executará o *script* de criação dos modelos a partir do arquivo de especificação JSON, fazendo todas as verificações necessárias na base de dados. Os demais componentes internos serão apresentados a seguir.
- ***models***: o models é um arquivo em Python, gerado automaticamente, que armazenará as classes geradas pela ferramenta. Esse módulo deverá ser importado e usado pelo *web service RESTful*.
- ***queries***: é um arquivo em Python onde são armazenadas as consultas que deverão retornar os dados

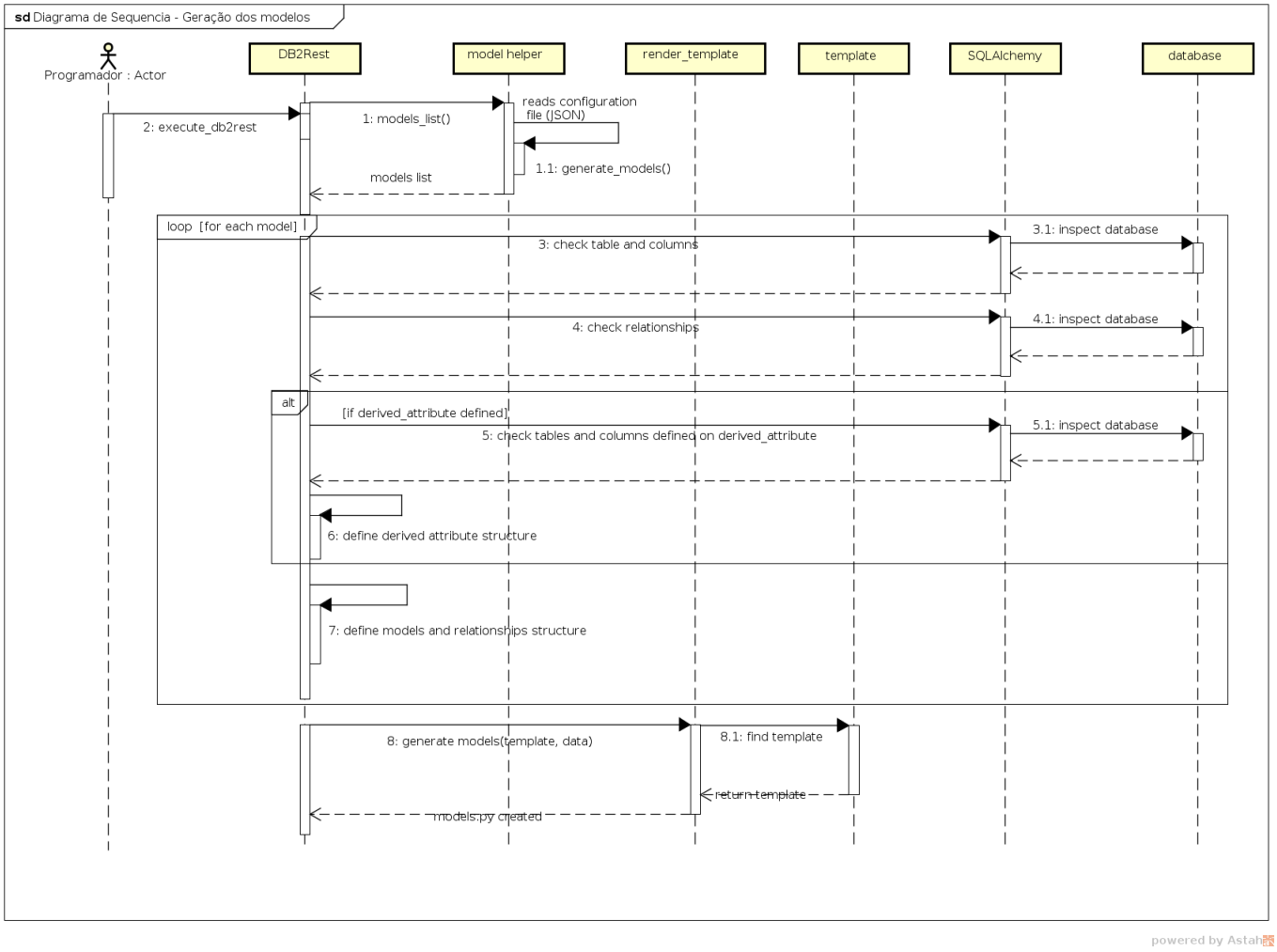


Figura 7: Diagrama de Sequência: geração dos modelos

utilizados como atributos derivados dos modelos, isto é, informações que estão disponíveis em tabelas que não possuem relacionamentos com a tabela mapeada para o modelo.

- ***db_config***: contém as configurações de API do SQLAlchemy para mapeamento da base de dados legada. Recebe como parâmetro uma variável de ambiente chamada de DATABASE_URL, contendo o caminho da base dados. Este módulo fornece uma classe Base, que é classe *Parent* dos modelos que foram gerados.

- **SQLAlchemy**: é o *framework* para a camada de persistência sobre o qual o DB2REST foi desenvolvido. Apresenta uma API bastante robusta e bem documentada, além de ser amplamente utilizado pela comunidade. No contexto do DB2REST, atua como um conector do tipo *data access*, efetuando as operações na base de dados.

- **Base de dados legada**: é a base de dados legada a ser integrada com o *web service RESTful*.

A Figura 7 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 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.

B. 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 as informações da base de dados.

- ***__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, tais como:
  - *rst_attribute_name*: contém o nome do atributo do modelo utilizado no *web service RESTful*;
  - *db_column_table*: contém o nome da tabela no banco de dados que será mapeado para este atributo;
  - *db_primary_key*: esta é uma chave especial que indica que esse atributo é chave primária da tabela no banco de dados.

  A definição desta estrutura para cada atributo permite a inclusão futura de novas informações a serem mapeadas.
- ***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. Os atributos necessários para a correta execução dessa funcionalidade são:
  - *rst_property_name*: contém o nome do atributo do modelo utilizado no *web service RESTful*;
  - *db_columns*: contém o nome do atributo que deverá ser retornado quando o *rst_property_name* for invocado. Deverá seguir sempre o formato que indica o nome da tabela e da coluna na base de dados («table».«column»);
  - *db_clause_where*: deverá conter as informações necessárias para executar a consulta e retornar o registro da base de dados. O nome da coluna na tabela e o valor a ser consultado deverão ser informados separados por uma barra, no formato «column»|«value».
  - *db_rows_many*: indica se a consulta retornará mais um registro (True) ou apenas um (False).
- ***relationships***: deverá conter uma lista de objetos JSON, com um registro para cada relacionamento a ser mapeado. É importante salientar que essa chave deverá se preenchida considerando a perspectiva do modelo atual em relação ao modelo-alvo. As seguintes chaves deverão ser definidas para cada relacionamento a ser mapeado:
  - *type*: indica o tipo de relacionamento estabelecido entre os modelos. Poderá ser dos tipos *many-to-many (M2M)*, *many-to-one (M2O)*, *one-to-many (O2M)* ou *one-to-one (O2O)*;
  - *rst_referencing_name*: define o nome do atributo através do qual essa relação será acessível no *web service RESTful*;
  - *rst_referenced_model*: define o nome do modelo com o qual este modelo se relacionará;
  - *db_referenced_table*: indica o nome da entidade na base de dados com a qual o modelo se relaciona;
  - *db_referenced_table_pk*: indica a chave primária da entidade do relacionamento na base de dados;
  - *db_referencing_table_fk*: indica o nome do atributo, presente na entidade associada a este modelo, que é chave estrangeira para outra tabela;
  - *rst_referenced_table_backref*: define um nome amigável para rastreamento dos relacionamentos de outras entidades em relação a um determinado registro. Isto é, permitirá o acesso ao relacionamento de forma reversa.

A Listagem 1 apresenta um exemplo do arquivo de configuração esperado pelo DB2REST, seguindo as definições apresentadas nesta seção. Consiste em um modelo chamado Postagem, que é utilizado no *web service RESTful*, e que será mapeado para uma entidade chamada *post* na base de dados.

A chave *attributes* possui uma lista de quatro atributos, sendo eles id_postagem, título, data_postagem e hora_postagem. Para cada um deles, existe um outro atributo chamado *column_name*, que contém o nome daquela coluna na base de dados.

A chave *derived_attributes* possui uma lista, em JSON, com apenas um item. Ele descreve uma atributo derivado que será reconhecido pelo *web service RESTful* como "detalhes_categoria"(*rst_ property_name*), e retornará o valor correspondente à tabela "*category"*, coluna "*name*"(*db_columns*), onde o "*id*"for igual ao valor 1 (*db_clause_where*). A última chave, *db_rows_ many*, cujo valor é False, indica que essa consulta retorna apenas um único registro.

```
[{
"__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

Por último, têm-se a chave *relationships*, que também é uma lista JSON, contendo apenas um item. Esse item contém a descrição de uma relacionamento do tipo *M2O* (*many-to-one*) com o modelo Categoria. Essa relação poderá ser acessada no *web service RESTful* através do nome "categoria"(*rst_referencing_name*) que retornará um objeto do modelo Categoria (*rst_referenced_model*). Os atributos *db_referenced_table* e *db_referenced_table_pk* indicam, respectivamente, a tabela *"category"* e sua chave primária "*category.id*", da entidade-alvo na base de dados. O atributo *db_referencing_table_fk* indica o atributo da entidade/modelo que está sendo mapeado, neste caso a tabela *"post"*, que é a chave estrangeira para a tabela *category*. Por fim, têm-se o atributo *rst_referenced_table_backref*, chamado neste exemplo de "postagens", que permitirá a consulta reversa dos registros do modelo Postagem através do modelo Categoria (exemplo: categoria.postagens retornaria todos os registros associados àquela categoria).

C. 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. O SQLAlchemy apresenta um suporte bastante eficiente para introspectar bases de dados legadas e dispõe de uma API robusta e bem documentada para realizar a manipulação de dados no banco de dados. 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 especificação de acordo com as orientações disponíveis na seção V-B, 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 e delega a geração do código ao script *render_template.py*. O *template* utilizado pelo DB2REST é desenvolvido usando o *Jinja2*, uma linguagem de *templates* para a linguagem Python. Nele, ocorre a definição da estrutura dos modelos que serão gerados.

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. Os serviços do *web service RESTful* precisarão apenas importar as novas classes geradas pelo DB2REST, e então utilizar a API do SQLAlchemy para realizar a manipulação dos dados. A Listagem 2 apresentada um exemplo de classe gerada automaticamente pela ferramenta.

```
from DB2Rest.db import Base
from sqlalchemy import Column,Integer,String,ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy import inspect
from DB2Rest import queries


class Postagem(Base):
    __tablename__ = "post"

    id_postagem = Column('id',Integer,primary_key=True)
    titulo = Column('title')
    data_postagem = Column('date')
    hora_postagem = Column('time')

    ##Relationships##
    categoria_id = Column('category',Integer,
                          ForeignKey('category.id'))
    categoria = relationship('Categoria',
                          back_populates='postagens',
                          lazy='joined')

     @hybrid_property
    def detalhes_categoria(self):
        return queries.get_table_derived_attributes(
            table_name='category', column_name='name',
            clause_where_attribute='id',
            clause_where_value=1,
            many=False)


    def __init__(self, **kwargs):
        for k, v in kwargs.items():
            setattr(self, k, v)
```

Listing 2: Classe gerada automaticamente pelo DB2REST

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 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.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from flask import Blueprint, jsonify, request
import datetime
from DB2Rest import models


postagem = Blueprint("postagem", __name__)


@postagem.route('/postagens')
def listar_postagens():
    postagens = models.Postagem.query.all()
    return jsonify(result=[dict(
            id=postagem.id_postagem,
            titulo=postagem.titulo,
            data=postagem.data_postagem,
            hora=postagem.hora_postagem,
            categoria=postagem.categoria,
            detalhes=postagem.detalhes_categoria)
            for postagem in postagens])
```

Listing 3: Serviço RESTful que utiliza o mecanismo DB2REST

VI. 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. A subseção VI-A apresenta o estudo de caso de um cenário real em que a solução poderia ser utilizada. A subseção VI-B introduz dados relevantes sobre o conhecimento prévio dos participantes do experimento nas tecnologias que foram utilizadas. A subseção VI-C apresenta o processo de execução dos experimentos de forma detalhada, abordando os aspectos de cada cenário analisado. Por fim, a subseção VI-D demonstra os resultados obtidos e as avaliações das hipóteses que foram investigadas.

A. 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. Dentre suas principais funcionalidades, estão o envio de mensagens de coordenadores de curso para professores ligados ao departamento e de professores para os alunos de suas turmas, consulta de cronograma de aulas dos professores, disciplinas cursadas pelos alunos, etc. O projeto teve início em 14 de Setembro de 2016, e o desenvolvimento da sua versão inicial ocorreu em cerca de 6 meses — período em que o autor do presente trabalho atuou como membro da equipe de desenvolvimento.

O aplicativo móvel foi desenvolvido em *Qt*, um *toolkit* multiplataforma que permite a criação de aplicações com interface gráfica. A aplicação móvel tem suporte para os sistemas operacionais *Android* e *IOS*. O *web service RESTful* foi desenvolvido utilizando o Flask — um *microframework* para desenvolvimento de serviços RESTful — e, para manipulação dos dados da aplicação, utilizou-se o SQLAlchemy — um *framework* que implementa a técnica de ORM (*Object-relational mapping*).

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 de desenvolvimento naquele período, tais como cursos, turmas, disciplinas, aulas, usuários etc. Essas entidades são manipuladas através de modelos de dados, isto é, cada entidade foi representada como uma classe Python que é interpretada pelo SQLAlchemy. Os serviços disponibilizados para o cliente *mobile* realizam operações e processamento de informações através dos objetos desses modelos.

Supondo que a instituição autorize o acesso ao banco de dados, utilizar o *Emile Server* nesta base de dados não seria possível. Uma vez que o *Emile Server* tem uma configuração de dados divergente da base de dados utilizada pelo IFBA (Instituto Federal da Bahia), para torná-los compatíveis os desenvolvedores teriam que reimplementar os modelos de dados e as regras de negócio dos serviços do *web service RESTful*.

O cenário supracitado ilustra um caso real em que o DB2REST poderia ser aplicado para resolver um problema de integração entre o *web service RESTful* existente e a base de dados da instituição. Os desenvolvedores precisariam criar e preencher um arquivo de configuração em JSON, descrito na subseção V-B, indicando as informações necessárias para realizar o mapeamentos entre os modelos definidos no *Emile Server* e das entidades da base de dados legada. Em seguida, o caminho do banco de dados e os dados de autenticação devem ser informados para permitir a validação dos dados inseridos no arquivo de configuração. Ao executar a ferramenta, é feita a leitura do JSON e, caso não sejam identificadas inconsistências, os novos modelos de dados que farão a adaptação entre essas interfaces são gerados automaticamente.

B. Perfil dos Participantes

Para avaliar a solução proposta neste trabalho, foi conduzido um estudo com dois participantes, ambos alunos do curso de Análise e Desenvolvimento de Sistemas do Instituto Federal da Bahia (IFBA) - Campus Salvador. Com a finalidade de conhecer mais sobre o perfil dos participantes, foram

Tabela III: Serviços e Modelos utilizados no experimento

Modelos	Serviços	Descrição dos Serviços
CourseSections	*/course_sections*	Retorna todas os registros de Disciplinas (*course_sections*)
Courses	*/course_details/<course_id>*	Retorna os detalhes de um Curso (*course*) a partir do id informado
SectionTimes	*/teachers_section_times/<teacher_id>*	Retorna o cronograma de Aulas(*section_times*) de um Professor pelo id.
Program	*/programs_course_sections/<program_id>*	Retorna todas as Disciplinas(*course_sections*) de um Curso(*program*)

recolhidas algumas informações a respeito dos conhecimentos prévios acerca das tecnologias utilizadas nos experimentos. Essas informações podem ser observadas no gráfico da Figura 8.

Os participantes escolheram, numa escala de 0 à 3, onde 0 representa nenhum conhecimento, 1 representa conhecimento básico, 2 conhecimento intermediário e 3 conhecimento avançado, o nível que melhor representa seus conhecimentos prévios em tecnologias específicas ou similares.

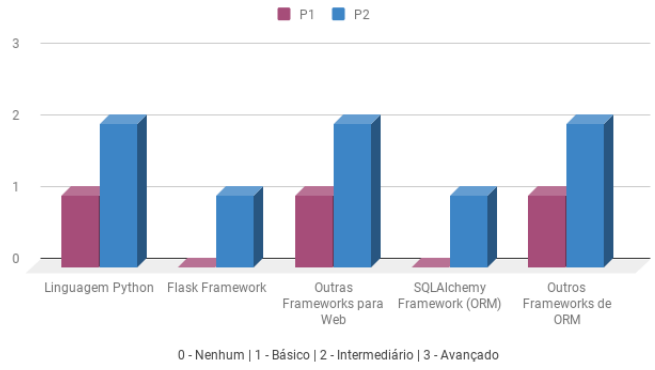


Figura 8: Conhecimento prévio dos participantes nas tecnologias utilizadas

O primeiro conjunto de colunas refere-se ao conhecimento em linguagem Python, utilizada para desenvolver a ferramenta DB2REST e também no desenvolvimento do experimento I. O segundo conjunto de colunas demonstra o conhecimento dos participantes em *Flask*, um *framework* para desenvolvimento de *web services RESTful*. O terceiro conjunto de colunas visa identificar se o usuário tem conhecimento em outros *frameworks* para desenvolvimento de *web services RESTful*. O penúltimo conjunto de colunas reporta o conhecimento em SQLAlchemy, que é a base sobre a qual foi desenvolvido o DB2REST e, portanto, tem fundamental importância para o uso da ferramenta. Por fim, o último conjunto de colunas do gráfico visa identificar se os participantes possuem conhecimento em outros *frameworks* de ORM.

O terceiro e quinto conjunto de barras, respectivamente, *Outros Frameworks para Web* e *Outros Frameworks de ORM*, referem-se ao conhecimento em tecnologias do mesmo segmento, uma vez que os princípios e terminologias podem ser similares e ajudarem na compreensão das tecnologias utilizadas.

C. Execução dos Experimentos

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. Neste treinamento, os participantes também conheceram o problema que o DB2REST busca resolver e foram orientados sobre a maneira correta de utilizar a ferramenta.

Os experimentos foram realizados utilizando o código-fonte de um sistema existente, o *Emile Server*, abordado na seção VI-A. Para isso, um conjunto de modelos e serviços, apresentados na Tabela III, foram escolhidos como base para utilização em dois cenários diferentes. Posteriormente, foram criados dois repositórios no *github* contendo uma versão do código-fonte do *Emile Server* de acordo com as configurações necessárias para avaliar os experimentos. A seguir, serão apresentadas a descrição dos repositórios e como foram utilizados em cada experimento.

1) Experimento I: foi disponibilizada uma versão do *Emile Server* sem a implementação dos modelos e serviços descritos na Tabela III. A partir de um modelo relacional, disponível no Apêndice A-1, os participantes desenvolveram os modelos no padrão do SQLAlchemy e criaram os serviços descritos na tabela utilizando o *Flask*. Ao finalizarem as implementações, eles executaram os comandos para a criação dos modelos e geração das tabelas na base de dados, bem como a realização da carga inicial de dados para verificarem a existência de *bugs* no código implementado. É importante salientar que no caso deste experimento, a base de dados foi criada durante a execução dos comandos de criação, após as implementações realizadas.

2) Experimento II: foi disponibilizada uma versão funcional do *Emile Server* contendo a implementação dos modelos e serviços descritos na Tabela III. Os participantes receberam um modelo relacional, disponível no Apêndice A-2, referente a uma base de dados diferente daquela utilizada pelo software, e utilizaram o DB2REST para integrar os serviços existentes à essa base de dados. Para isso, eles preencheram o arquivo de configuração no formato JSON, descrito na seção V-B, com as informações disponíveis nos modelos existentes e nas informações contidas no diagrama. Em seguida, executaram os comandos de execução da ferramenta e importaram os novos modelos gerados por ela no serviços existentes.

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. 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. Ao final das implementações, eles fizeram o *push* para o repositório de testes.

D. 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 IV apresenta as hipóteses que serão analisadas com base nos resultados obtidos.

Tabela IV: 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 9 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 o participante P1, no experimento I, demandou mais tempo para implementar os modelos do que o participante P2. Também é possível observar que o mesmo ocorre em relação a implementação dos serviços, que demandou muito mais tempo, pois requer maior entendimento das regras de negócio a ser implementada, bem como dos padrões de codificação definidos pelas ferramentas SQLALchemy e Flask. Tal comportamento destoante demonstrado no gráfico pode ser melhor compreendido se considerarmos o nível de conhecimento prévio dos participantes, demonstrado na Figura 8. Como P2 possui mais conhecimento nas tecnologias abordadas do que P1, este conseguiu realizar as implementações em um tempo consideravelmente menor.

Em relação ao experimento II, observa-se que para criar o arquivo de configuração, descrito na seção V-B, P1 e P2 demandaram tempo de atividade similares, pois esta etapa não requereu nenhum tipo de codificação, estando restrita a criação e preenchimento de um arquivo JSON. Entretanto, P1 obteve um desempenho significativamente melhor em relação a execução da primeira etapa do experimento I, enquanto P2 demandou pouco tempo a mais em relação ao mesmo cenário. No que tange segunda etapa, que compreende a implementação dos serviços, observa-se que o tempo de ambos os participantes é consideravelmente menor em relação a mesma etapa do experimento I. Tal característica se deve ao fato de que, no caso deste experimento, não foi necessário implementar a lógica de negócio necessária para o funcionamento correto dos serviços, bastando apenas mudar o módulo de importação.

Considerando os dados demonstrados no gráfico da Figura 9, e com base nas análises realizadas, pode-se considerar que H1 é verdadeira. A condição $T_{DB2REST} < T_{Flask}$ é válida, uma vez

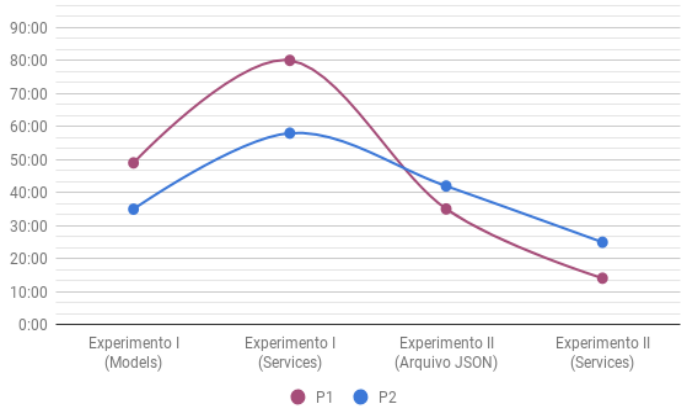


Figura 9: H1: Produtividade em função do tempo de desenvolvimento

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 *web service RESTful*.

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, descritos na Tabela III, 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.

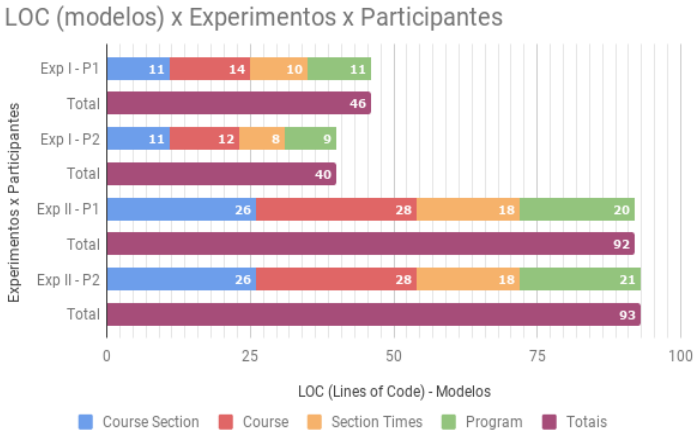


Figura 10: H2: Análise de Complexidade - Modelos

A Figura 10 apresenta um gráfico com as informações da quantidade de linhas que os participantes escreveram, demonstrando a quantitativo de linhas necessárias para cada modelo, acompanhados de uma coluna com a quantidade total de linhas para cada experimento/participante. Nele, 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*. Além disso, diferentemente da definição dos modelos do experimento I, o arquivo JSON é utilizado apenas para fornecer dados necessários para o mapeamentos das entidades/modelos.

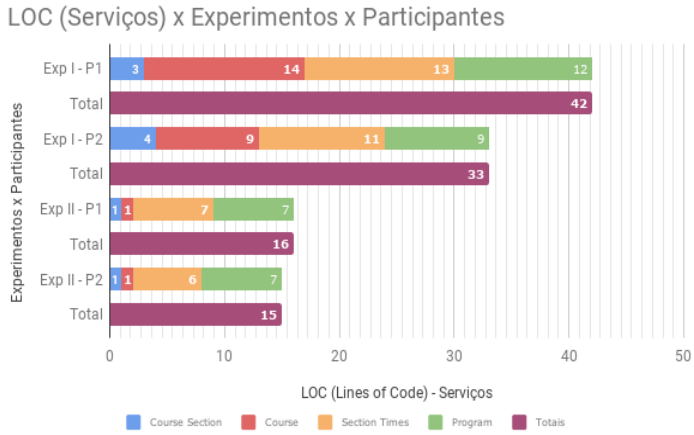


Figura 11: H2: Análise de Complexidade - Serviços

A Figura 11 apresenta um gráfico com a quantidade de linhas de código (*LOC*), apenas para os serviços implementados pelos participantes na execução dos experimentos. Assim como o gráfico de quantidade de linhas para os modelos (Figura 10), este gráfico demonstra o quantitativo de linhas necessárias para cada modelo, acompanhado de uma coluna com a quantidade de linhas para cada experimento/participante. 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 Figuras 10 e 11, conclui-se que H2 é falsa. Embora tenham sido pontuados aspectos relativos à estrutura do arquivo JSON e das diferenças semânticas entre ele e os códigos desenvolvidos no experimento I, a métrica utilizada para avaliar esta hipótese, o *LOC* (*Lines of Code*), indica 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.

Para fins de avaliação da H3, foi realizada a extração da densidade de *bugs* (DB) dos experimentos. A Figura 12 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* na primeira etapa do experimento I (P1-DB: 0,022727; P2-DB: 0,01282), cujos problemas foram relacionados a definição dos atributos de modelo, foi maior do que a constatada no experimento II (P1-DB: 0,0125; P2-DB: 0,00625), onde ocorre a criação do arquivo JSON, cujos erros ocorreram devido a inserção incorreta dos dados de mapeamento. Entretanto, ainda que tenham sido encontrados *bugs* nos dois experimentos, observa-se que a DB do experimento II foi menor em relação a mesma etapa do primeiro cenário.

No que tange a densidade de *bugs* referente a criação dos serviços, nota-se a DB identificada nos códigos do experimento I do participante P1 (DB: 0,03409) é consideravelmente maior do que a DB atribuída ao mesmo cenário para o participante P2(DB: 0,01282). Na primeira etapa do experimento I, os participantes tiveram que implementar as regras de negócio dos serviços. Os fatores que influenciaram nos resultados deste cenário estão ligados ao estilo de programação e conhecimento prévio dos participantes. Entretanto, ao analisar a DB dos serviços gerados no experimento II, nota-se que o resultados de ambos os participantes (P1-DB: 0,00625; P2-DB: 0) é muito superior em relação a mesma etapa do primeiro cenário. Tal característica se deve ao fato de que nesta etapa 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.

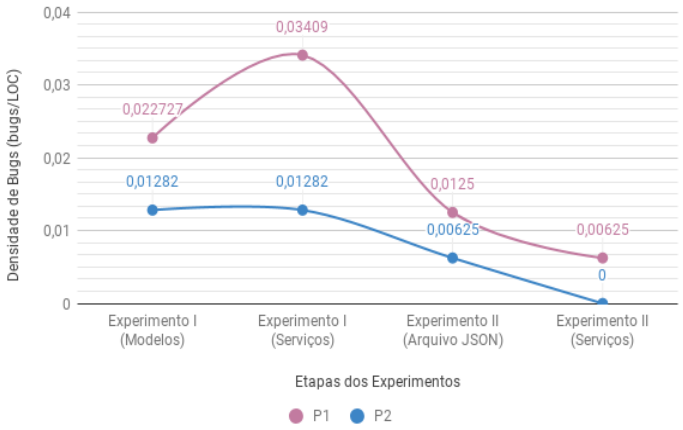


Figura 12: H3: Análise da Densidade de Bugs

De acordo com o gráfico apresentado na Figura 12, e com base no que foi analisado e exposto anteriormente, conclui-se que H3 é verdadeira. A condição $DB_{DB2REST} < DB_{Flask}$ é valida, 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*.

VII. 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.

Embora existam algumas soluções de software que auxiliem no processo de criação de APIs *RESTful* a partir de bases de dados legadas, a criação facilitada dessas APIs, na grande maioria das vezes, estão condicionadas à estrutura da base de dados legada. Entretanto, essas soluções não preveem os casos em que deseja-se integrar uma base de dados legada a um servidor *web RESTful* existente, que foi desenvolvido considerando a existência de modelos, atributos e relacionamentos sobre os quais os serviços realizam algum processamento.

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, a partir do seu modelo relacional. 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 em que viabiliza o reuso de soluções e, consequentemente, a redução nos custos de produção de novos softwares dentro de um mesmo domínio de aplicação.

VIII. 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:

- A criação de 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.

Referências

[1] A. de Oliveira Galvão; Osni Barbosa Chagas; Simone Bello Kaminski Aires and J. P. Aires, "Alternatives development software, without cost, for micro and small enterprises," *Revista ADMpg Gestão Estratégica*, vol. 2, no. 2, pp. 119–123, 2009.

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

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

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

[5] L. Richardson and S. Ruby, *Restful Web Services*, 1st ed. O'Reilly, 2007.

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

[7] H. Hamad, M. Saad, and R. Abed, "Performance evaluation of restful web services for mobile devices," *International Arab Journal of e-Technology*, vol. 1, no. 3, 01 2010.

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

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

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

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

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

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

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

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

[16] A. Joshi and S. Kukreti, "Object relational mapping in comparison to traditional data access techniques," *International Journal of Scientific Engineering Research*, vol. 5, 06 2014.

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

[18] E. Płuciennik-Psota, "Object relational interfaces survey," *Studia Informatica*, vol. 33, no. 2A, pp. 299–310, 2012.

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

[20] X. Gantan. (2014) Python's sqlalchemy vs other orms. Acesso em: Junho 15, 2017. [Online]. Available: http://pythoncentral.io/sqlalchemy-vs-orms/

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

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

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

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

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

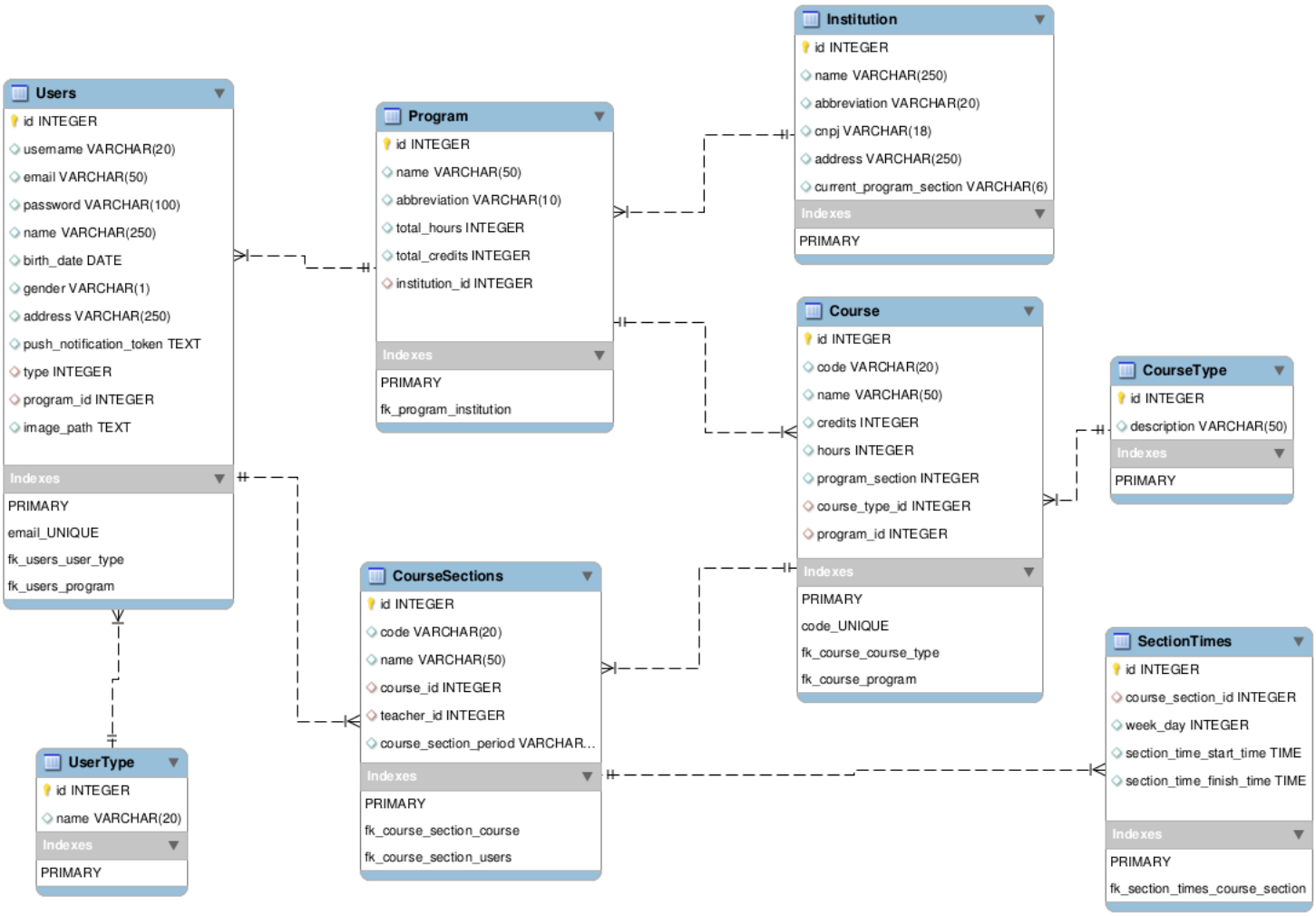
[26] Hibernate ORM Documentation, “More than an orm, discover the hibernate galaxy,” http://hibernate.org/orm/, (Acesso em: Junho 17, 2017).

[27] o7planning.org. (2016) Using hibernate tools generate entity classes from tables. Acesso em: Junho 17, 2017. [Online]. Available: http://o7planning.org/en/10125/using-hibernate-tools-generate-entity-classes-from-tables

[28] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

APÊNDICE A

1) Modelo Relacional - Experimento I:



2) Modelo Relacional - Experimento II: