Estruturação de Aplicações Distribuídas com a Arquitetura REST

Willian O. Ferreira[1], Igor O. Knop[1]

[1]Centro de Ensino Superior de Juiz de Fora - Juiz de Fora - MG - Brasil
ottoniwillian@gmail.com, igorknop@pucminas.cesjf.br

Abstract. *The definition of a software architecture is becoming an essential activity to permit evolution software in order to improve development process, reducing its complexity. Furthermore, it is necessary to conduct a software process with some objectives, such as: reusability of architecture objects, communication protocols, performance, physical distribution and so on. This article describes techniques for development software based on disciplines, such as, web services and distributed systems, in order to reduce the complexity of application developed. Specifically, this article approaches a real case study using RESTful Web Services supported by Jersey Framework and, the AngularJS technology, which used in client interface. These techniques and technologies were applied using web platform as a real example.*

Resumo. *Com a complexidade dos sistemas aumentando a cada dia, estruturar um software em camadas torna-se imprescindível, pois resulta em maior facilidade de desenvolvimento e manutenção. Para estruturar um projeto é necessário analisar composição de elementos, protocolos de comunicação, desempenho, estruturas de controle, sua distribuição física e outros. O objetivo geral deste trabalho é descrever e discutir os protocolos de comunicação, desempenho e a aplicabilidade das técnicas de estruturação de aplicações distribuídas. Será apresentada a arquitetura REST que permite a outras plataformas fazerem uso das funcionalidades de web services. Um estudo de caso real foi desenvolvido, com o intuito de aumentar os recursos de um software proprietário, nativamente desktop, utilizando web services via Framework RestFul Jersey no servidor e uma aplicação web com o AngularJS no lado do cliente.*

1. Introdução

Os sistemas de controle financeiros de pequenas a grandes empresas podem estar em sistemas legados que dificilmente recebem manutenções, em algumas organizações têm-se uma maior quantidade de fornecedores de software para diferentes áreas devido a questões de preço e outros. Com os avanços da sociedade, surge a necessidade de acesso a informações online bem como: consultas de saldos; contas a pagar; contas a receber; movimentos bancários e outras [Sebrae 2016]. Os limites da empresa tradicional estão esmaecendo à medida que as organizações estão disponibilizando suas funcionalidades de aplicativos locais e dados para organizações parceiras via web, aplicativos móveis, dispositivos inteligentes e nuvem. Com a necessidade de integração de ambientes totalmente diferentes, o uso de *web services* é uma das maneiras mais utilizadas para integrar os diferentes sistema [Vigo e Oliveira 2014]. Porém, todas as decisões de projeto, seja na arquitetura ou no *software* devem ser realizadas dentro do contexto de requisitos funcionais, modelos de processos e de dados, aspectos comportamentais, e não usar somente as tendências de mercado. Com a informatização de praticamente todos os setores, uso de vários softwares em conjunto, gestores sentem a necessidade que as informações analíticas sejam concentradas em uma única solução. A

integração se torna importante ao fazer soluções de áreas diferentes e heterogêneas comunicarem gerando indicadores importantes sobre o desempenho da empresa para auxiliar as tomadas de decisão.

Vários estudos e tecnologias são levantadas no início de um projeto de software, pensando em reutilização em um cenário onde precisamos integrar clientes, plataformas diferentes ou parceiros. As empresas de *software* visam um sistema rápido de se desenvolver e simples de reaproveitar em ambientes variados. Entretanto, deve-se pensar sempre em uma futura integração, mesmo que não seja pertinente no momento. Sistemas de gestão estando distribuídos contribuem para excelência corporativa da empresa.

Um dos focos desse trabalho é a disponibilização de informações através de uma interface padronizada. Sendo assim, podemos pensar em *web services* para realizar esta padronização, permitindo a qualquer tipo de aplicação consumir os mesmos, aumentando a flexibilidade da aplicação e oferecendo possibilidades infinitas de visualização destas informações.

Para permitir a qualidade, desempenho e manutenção dos serviços remotos, várias abordagens surgiram. Uma dessas é a arquitetura REST, que vem se popularizando em relação a aplicações implementadas tradicionalmente usando SOAP (*Simple Object Access Protocol*) [Rozlog 2016].

Em sua tese "Estilos Arquitetônicos: Projeto de Arquiteturas de Software Baseadas em Rede, Roy Fielding define a Representational State Transfer (REST), ou em uma tradução livre, Transferência de Estado Representacional. Este trabalho apresenta um estudo sobre tecnologias para construção de sistemas distribuídos via web, dando ênfase na arquitetura REST. O uso do REST recebe maior foco por permitir que sistemas com modelos arquitetônicos projetados para sistemas hipermídia distribuídos sejam construídos com certa rapidez e simplicidade.

Um estudo de caso usando as tecnologias citadas foi desenvolvido com base no conhecimento e experiência profissional do autor em organizações situadas na região de Juiz de Fora. As funcionalidades implementadas permitiram a exposição dos dados na forma de serviços *web* e foram consumidas por uma aplicação Web. O inicio do projeto referente a este estudo de caso ocorreu em junho de 2015, momento no qual definiu-se o escopo e o referencial teórico dos temas abordados neste artigo.

Este trabalho está organizado em 8 seções. Além desta Introdução, a Seção 2 fala sobre o uso dos *web services*. A Seção 3 será realizada uma breve revisão das abordagens baseadas em SOAP e suas vantagens e desvantagens em relação ao REST. A Seção 4 apresenta uma introdução sobre as características do modelo arquitetural REST. Na Seção 5 será apresentada algumas características para integração de sistemas. Para evidenciar o uso da arquitetura do modelo REST, um estudo de caso com esta abordagem será apresentado na Seção 7. Por fim, a Seção 8 apresenta as considerações finais do trabalho e possíveis caminhos para sua continuação.

2. Web Services

A medida que os computadores começaram a se comunicar, foram surgindo protocolos especiais para transmissão dos dados como: SMTP, FTP, HTTP e outros. Algumas tecnologias foram surgindo para facilitar essa transmissão, uma delas é o RPC (*Remote Procedure Call*) e esta descrito pela [RFC 707 2016], Chamada de Procedimento Remoto em tradução livre. Consiste em uma tecnologia capaz de um sistema acessar um procedimento em outro computador ligado a rede. A empresa Microsoft [Microsoft Irc 2016] criou DCOM (*Distribuied* COM) uma vertente de RPC para permitir o uso de componentes COM através

da rede. O RPC Sun foi desenvolvido pela Sun [Oracle 2016] para ser interpretados por plataformas Unix e Linux. Cada organização tinha seu próprio padrão e isso tornava a integração entre plataformas diferentes muito complexa.

Os *web services* ou serviços *web* surgiram para permitir a interoperabilidade entre sistemas de plataformas diferentes. É uma forma de prover compatibilidade e comunicação entre diferentes aplicações, permitem que sistemas dispostos compartilhem informações em diferentes localidades. Dependendo do grau de distribuição, os sistemas podem estar sujeitos à diferentes políticas. Sistemas distribuídos podem utilizar diferentes plataformas e tecnologias. A Arquitetura Orientada a Serviços, ou SOA (*Service-Oriented Architecture*) é uma forma de implementar esses sistemas, esta incorpora uma série de conceitos que visam assegurar a interoperabilidade e flexibilidade exigidas por processos de negócios [Salvadori, 2015].

Podemos citar duas formas de implementação de *web services*: através do padrão SOAP (*Simple Object Access Protocol*) e implementações que seguem os princípios arquiteturais REST (*REpresentational State Transfer*).

3. Abordagens SOAP em relação a REST

O SOAP foi o primeiro padrão de *web services.* Está atualmente na versão 1.2. Está versão descontinua a abreviação (*Simple Object Access Protocol*) e define SOAP como um protocolo leve destinado à troca de informações estruturadas em um ambiente descentralizado, distribuído. Ele usa tecnologias XML para definir uma estrutura de mensagens extensível fornecendo uma construção de mensagem que pode ser trocada por uma variedade de protocolos subjacentes [W3C 2016].

As mensagens de requisição e resposta usadas nesse modelo são encapsuladas como envelopes SOAP e as interfaces de troca de mensagens dos serviços são descritos em WSDL (*Web Services Description Language*). A Tabela 1 menciona algumas vantagens e desvantagens de SOAP em relação ao REST.

Tabela 1. Vantagens e desvantagens de SOAP em relação ao REST

Vantagens	Desvantagens
Independência de transporte: Os cabeçalhos estão dentro da mensagem, ou seja, eles são independentes de protocolo para transportar a mensagem. O envelope SOAP pode ser transportado por qualquer protocolo: HTTP, SMTP, TCP e outros. Os conceitos sobre segurança são melhores especificados e difundidos nos protocolos baseados em SOAP do que no protocolo HTTP, utilizado pelo estilo REST. Os conceitos sobre transação são melhores especificados para o protocolo SOAP.	Não utilizam todos os métodos do HTTP, limitando-se ao POST para realizar múltiplas operações. SOAP é uma especificação de um protocolo estruturado em XML. Devido ao tamanho de suas mensagens, elas naturalmente ocupam mais rede, o que pode ser um problema em algumas situações onde a largura de banda é limitada. Não pode ser armazenado em cache, pois o SOAP usa o método HTTP POST, o que é considerado não seguro, pois esse método pode alterar os dados de um recurso.

Comparar as duas abordagens é complexo, pois se trata de modelos para integração de *web services* com características distintas. É possível observar na Tabela 1 que uma abordagem pode levar vantagem sobre a outra dependendo do contexto.

O protocolo SOAP é maduro e mantido pela principal organização de padronização da World Wide Web a W3C, portanto sua especificação é bem definida e completa. REST é uma opção de integração considerado leve e com baixa curva de aprendizagem. Este modelo está em plena expansão. Por ser mais leve, alguns aspectos de segurança e controles transacionais não estão presentes no REST, sendo recomendado em cenários onde o objetivo é apenas proporcionar acesso a dados para sistemas externos [Salvadori 2015 apud Josuttis 2016). Segundo Rozlog (2016), os desenvolvedores que o utilizam REST citam, como principais vantagens a facilidade no desenvolvimento, o aproveitamento da infraestrutura web existente e um esforço de aprendizado pequeno. REST funciona bem em cenários onde existem limitação de banda e recursos. Usando o modelo REST a estrutura de retorno é definido pelo desenvolvedor em diversos formatos e qualquer navegador pode ser usado. Isso acontece porque REST usa o padrão de chamadas GET, PUT, POST e DELETE. Se as operações não precisarem ser continuadas, forem totalmente sem estado, um CRUD por exemplo (Criar, Ler, Atualizar e Excluir), o REST seria a melhor alternativa. Situações em que a informação pode ser armazenada em cache, devido à natureza da *restrição* sem estado do REST, esse seria um cenário adequado para a tecnologia. REST tem simplicidade em seu entendimento e pode ser adotado por qualquer cliente ou servidor com suporte a HTTP/HTTPS.

4. REST

A Transferência de Estado Representacional ou *Representational State Transfer*, REST, é um modelo de arquitetura desenvolvido no ano de 2000 por Roy Fielding, um dos autores do protocolo mais usado no mundo, o HTTP [Fielding 2000]. Se trata de uma coleção de princípios e restrições arquiteturais para o desenvolvimento de aplicações distribuídas na Web. Ele adota o paradigma cliente-servidor, onde as requisições partem inicialmente do cliente e são respondidas pelo servidor [Fielding 2000].

A formalização de um conjunto de melhores práticas denominadas *restrições* foi o intuito geral de REST. As *restrições* tinham como objetivo determinar a forma na qual padrões como HTTP e URI (*Uniform Resource Identifier*) deveriam ser modelados, aproveitando de fato todos os recursos oferecidos pelos mesmos. O REST foi gerado numa perspectiva conhecida como *null style.* Basicamente, representa um conjunto simples e vazio de características. A partir de uma visão arquitetural, definimos a seguir as *restrições* propriamente ditas. Os *web services* desenvolvidos com o padrão arquitetural REST são chamados de Web API ou simplesmente API REST. Se o *web service* segue todos os princípios e *restrições* em sua implementação, este é considerado RESTful [Salvadori 2015].

4.1. Cliente-servidor

Esta *restrição* tem como característica dividir as responsabilidades das partes de um sistema. Essa separação pode existir quando por exemplo queremos desvincular a interface do usuário e o *back-end.* Dessa forma permite uma melhor escalabilidade e evolução de responsabilidade de forma independente. Outro objetivo é o desacoplamento entre o *back-end* e mecanismos de armazenamento de dados de uma aplicação.

Com essa *restrição* têm-se um servidor responsável por oferecer um conjunto de serviços que serão invocados por aplicações clientes, o servidor disponibiliza seus recursos

para ser consumido por algum cliente independentemente da plataforma ou da linguagem escrita [Fielding, 2000].

4.2. Sem Estado

Pode-se dizer que a característica desta *restrição* é proporcionar ao servidor uma requisição que contenha todas as informações necessárias para que ele as trate com sucesso, ou seja, não é necessário que tenha requisições antigas ou futuras. O modelo do HTTP segue esse modelo, mas é comum o uso de *cookies* para armazenamento de sessões no servidor. O uso de *cookies* podem trazer inconvenientes por isso deve ser usado com cautela, sua utilização diminui muito a escalabilidade de aplicações.

Segundo Fielding (2000), o estilo *Sem Estado* deriva de cliente-servidor com a restrição adicional de que nenhum estado de sessão é permitido no componente de servidor. Cada solicitação do cliente para o servidor deve conter todas as informações necessárias para compreender o pedido e não pode tirar vantagem de qualquer contexto armazenado no servidor. O estado da sessão é mantido inteiramente no cliente. Utilizando um modelo de comunicação *sem estado*, as APIs REST passam a ter características como visibilidade, confiabilidade e escalabilidade. Um elemento que pode ser utilizado são os *balanceadores de carga* que permitem o adicionamento de vários servidores a aplicação e o cliente não identifica, pois está ali de forma transparente.

4.3. Cache

Na arquitetura REST para se obter maior desempenho, as respostas devem permitir o uso de cache. Os navegadores são bons controladores de cache, podemos usá-los para armazenar as respostas.

O HTTP permite que essa *restrição* funcione à partir da inserção de cabeçalhos “expires” na versão 1.0 do HTTP ou como opção na versão 1.1 do HTML usando o “cache-control”. Na Figura 1, pode-se observar que as solicitações ao servidor são reduzidas devido ao uso de cache no *browser*.

Figura 1. Exemplo do funcionamento de cache-control.

A primeira resposta é armazenada no cache do *browser* com tamanho de 1024 bytes e indica ao cliente o armazenamento por até 120 segundos. Após se passar o tempo do armazenamento está mensagem é expirada e o cliente teria que realizar novo download da solicitação. Para diminuir as solicitações, *tokens* de validação foram desenvolvidos. O servidor gera e retorna um *token* arbitrário, este será a identidade do conteúdo do arquivo. Na

segunda resposta o cliente só precisará enviar esta identidade na solicitação, se ainda for a mesma, o recurso não foi alterado e o download pode ser ignorado. As resposta devem informar aos elementos que compõe o processo (balanceadores de carga, *gateways*, *proxies,* navegadores) qual o tipo de cache mais adequado para aplicação.

A melhor solicitação é uma solicitação que não precise se comunicar com o servidor: uma cópia local da resposta permite eliminar toda a latência de rede e evita cargas de dados para a transferência de dados [Grigorik 2016].

4.4. Interface Uniforme

A principal característica que diferencia REST de outras arquiteturas baseadas em rede é a sua ênfase em uma interface uniforme entre componentes [Fielding 2000]. Usar um sistema com interface uniforme propicia uma arquitetura desacoplada e simplificada. O protocolo utilizado em *web services* REST é o HTTP e uma das formas para o estabelecimento desta uniformidade da interface é considerar a semântica do protocolo. Considerar a semântica do protocolo é utilizar seus códigos de *status* e verbos (GET, POST, PUT, DELETE).

REST sugere que sejam seguidas as boas práticas do HTTP e que os códigos de *status* devem ser utilizados. Por exemplo, quando um cliente faz uma requisição ao servidor além da resposta é recebido um código de *status.* Eles estão divididos em cinco grupos distintos (1XX Informações, 2XX Sucessos, 3XX Redirecionamentos, 4XX Erros causados pelo cliente, 5XX Erros causados no servidor).

Segundo o modelo arquitetural, as seguintes características devem ser implementadas: mensagens auto descritivas, identificação de recursos e hipermídia. Essa *restrição* define uma interface uniforme entre clientes e servidores para comunicação com o objetivo de fazê-los independentes um do outro.

4.5. Sistema em Camadas

Segundo Fielding (2000) um sistema em camadas permite que a arquitetura seja composta por diversos níveis hierárquicos, restringindo os componentes participantes de tal forma que nenhum deles possa enxergar através da camada adjacente, ou seja, o conhecimento do sistema que o componente possui é restrito à sua camada apenas.

Para que grandes sistemas distribuídos atinjam a escalabilidade, a API REST deve ter a capacidade de inserir elementos intermediários não visualizados por seus clientes. Este recurso é interessante e foi o que tornou a *web* um sucesso, para acessar um site não é necessário informar o DNS nem mesmo se deseja fazer cache.

4.6. Código Sob Demanda

A capacidade de um software evoluir sem a necessidade da quebra do mesmo se dá o nome de extensibilidade. Usando um código sob demanda podemos adaptar o cliente para as novas funcionalidades e requisitos do sistema sem que este sistema seja interrompido. Com esta *restrição* o cliente consegue estender parte da lógica do servidor utilizando os mesmo serviços disponibilizados por ele. Fielding (2000) em sua tese trata esta *restrição* como opcional. É aconselhável sua implementação em caso de algum serviço do lado do cliente seja mais rápida e eficaz.

4.7. Recursos

Recursos formam a base dos princípios REST e podem ser qualquer informação que se deseje tornar acessível a clientes remotos, e que são endereçados através de um identificador único, através de URI. Uma URI endereça apenas um recurso, mas um recurso pode ser identificado por diversas URIs [Salvadori 2015]. Quando um recurso é acessado uma representação dos valores deste recurso em um determinado momento do tempo é devolvida ao solicitante. Uma URI sempre estará ligada no mínimo a uma representação, portanto um recurso só é acessado diretamente através de uma representação.

O trecho de código da figura 2 apresenta um *web service* desenvolvido com o *framework Jersey* [Jersey 2016]. Com este código é possível explicar a organização de um *web service* REST e observar as características para criação de um recurso. Este código foi retirado do *Web Service* criado como estudo de caso deste trabalho, que tinha por objetivo, fornecer uma interface padronizada aos dados de um sistema que era acessado apenas localmente.

```
1 @Path("/pessoa")
2 public class PessoaResource {
3    @GET
4    @Produces("application/json; charset=UTF-8")
5    @Path("/{id}")
6    public Pessoa listarPessoa(@PathParam("id") Long idPessoa) {
7         try {
8            EntityManagerFactory factory =Persistence.createEntityManagerFactory("NewPersistenceUnit");
9            EntityManager entityManager = factory.createEntityManager();
10            Pessoa pessoa = null;
11            pessoa= new PessoaDao(entityManager).listar(idPessoa);
12        } catch (Exception e) {
13           entityManager.close();
14           factory.close();
15           throw new WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
16        }
17        if (pessoa = = null) {
18           entityManager.close();
19           factory.close();
20           throw new WebApplicationException(Response.Status.NO_CONTENT);
21        }
22        entityManager.close();
23        factory.close();
24        return pessoa;
25    }
26 }
```

Figura 2. Trecho da classe PessoaResource método listarPessoa. Código em JAVA framework Jersey do *web service*.

O *Jersey* usa a notação `@Path` para identificar de forma única os recursos e parâmetros inseridos via URI. Na Linha 1 `@Path("/pessoa")` indica o nome do recurso, já na Linha 5 `@Path("{id}")` indica a passagem de parâmetro compondo a URI para busca de uma *Pessoa*. Na Linha 4 `Produces("application/json; charset=UTF-8")` se trata de uma informação que faz parte do cabeçalho do HTTP, neste caso foi inserida diretamente no código do *web service* e esta indicando que o retorno ou representação deste recurso será em formato JSON, mas poderia dentro do contexto ser XML, IMAGEM, PDF e outros. Nas linhas abaixo temos um código em Java criando uma estrutura de acesso as classes de Persistência.

Caso aconteça alguma falha ao acessar o banco de dados, o sistema automaticamente retornará um código de status referente a *INTERNAL_SERVER_ERROR*. O erro não será exposto ao cliente da Web API através de uma mensagem, mas sim pelo código de *status* equivalente a este erro ou informação.

O mesmo acontece caso não seja encontrado uma *Pessoa*, o retorno será *NO_CONTENT* identificando que nesta requisição nenhum registro foi encontrado. Embora o HTTP seja cheio de limitações, o protocolo precisava de uma regra geral para interpretação de novos códigos de *status* de respostas, de modo que novas respostas poderiam ser implantadas sem prejudicar significativamente os clientes mais antigos [Fielding 2000]. O desenvolvimento mais fácil para o cliente favorecem a documentação. A aplicação que irá consumir a Web API não precisará se preocupar com qual mensagem será retornada, ao analisar o código de *status* é possível identificar o desfecho da requisição.

4.8. Representações de Recursos

As representações podem ser modeladas em diversos formatos. Os mais utilizados são JSON e XML [Richardson e Amundsen 2013]. As representações do estado devem suportar a utilização de hipermídia como imagens, áudio ou vídeo. Utilizando novamente o exemplo anterior da figura 2, a figura 3 apresenta um trecho de código referente a uma representação de um recurso REST:

```
{
  "codigo":1,
  “nome":"WILLIAN OTTONI",
  "cpf":"079.354.456-70",
  "rg":"15770012",
  "sexo":"M",
  "numero":"128",
  "logradouro":"RUA TÉOFILO OTTONI"
}
```

Figura 3. Código em formato JSON, representação resumida do recurso pessoa/1 do *web service*.

Quando um recurso é solicitado por um cliente, o servidor executa uma série de atividades e retorna uma mensagem ao solicitante. Essa mensagem é uma representação de um determinado recurso. A mesma não necessariamente precisa estar ligada a alguma parte concreta de um sistema, como por exemplo, uma tabela de banco de dados. Neste caso a resposta é um JSON mas o estado poderia ser representado por uma imagem, por exemplo. Tal escolha pode ser feita usando o recurso do HTTP conhecido como negociação de conteúdos. Que é a capacidade de processar múltiplas representações para um mesmo estado através da opção “Accept” no cabeçalho HTTP. Não é necessário que exista uma URI exclusiva para cada representação. Pode haver uma única URI associada a diferentes representações. A Figura 4 ilustra esse caso.

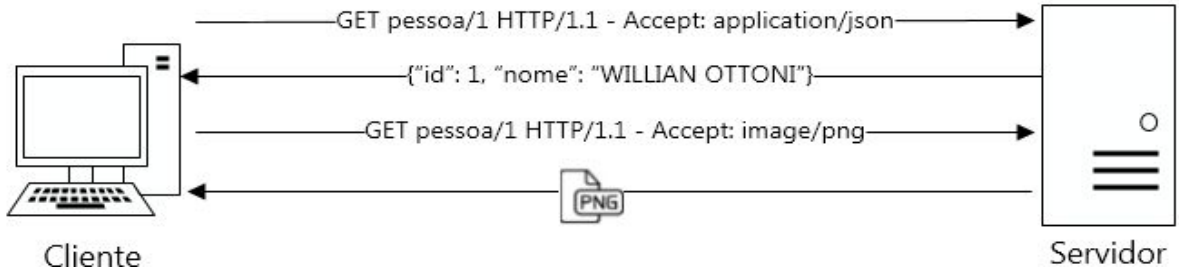

Figura 4. Acesso ao mesmo recurso com estados diferentes

Fielding (2000) menciona a técnica HATEOAS (Hypermedia as the Engine of Application State). Esse princípio define que o estado da aplicação, ou seja, o estado dos recursos desta, é transferido entre o cliente e o servidor na forma de hipermídia: hipertexto com hiperlinks.

Toda vez que uma página web é acessada, além do texto da página, diversos links para outras páginas são carregados. HATEOAS considera estes links da mesma forma, através da referência a ações que podem ser tomadas a partir do recurso atual.

5. Integração de Aplicações

As organizações necessitam que sistemas de departamentos diferentes estejam interligados. Por exemplo, o departamento financeiro precisa de dados contidos no departamento de vendas e logística. Uma solução é oferecer uma forma padronizada de acesso a informações, que, sem os *web services*, não estariam acessíveis de forma direta e segura.

A alternativa que foi escolhida para este trabalho é a integração através de uma API REST. Neste contexto, uma base de dados não é compartilhada, mas sim uma forma padronizada e estruturada de dados é exposta para que qualquer aplicação possa usufruir destas informações formando uma camada de integração, uma espécie de *wrapper* para aplicação que antes não era acessível.

A vantagem é a redução do acoplamento entre as aplicações, as alterações nos modelos de dados não irão influenciar diretamente na integração o que possibilita que as aplicações evoluam em ritmos diferentes. Existe a possibilidade de integrações para fora do domínio da organização, pelo fato da base não ser mais compartilhada os dados são disponibilizados após serem protegidos por controle de acesso, tornando o sistema mais abrangente.

Quando objetivo de um projeto é criar uma Web API para acesso externo, ou seja, liberada para que qualquer cliente a consuma, a ideia é que seja de fácil integração. A conhecida Lei de Postel, em um a tradução livre é “Seja conservador no que faz, seja liberal no que você aceita dos outros” [RFC 793 2016]. Jon Postel escreveu essa frase querendo representar a comunicação entre sistemas. Quando um sistema A falar com B, A deve falar o mais correto possível e, em contra partida, quando B falar com A ele deve ser mais flexível. Se uma requisição de B esteja faltante ou diferente do esperado, A deve saber lidar com esse problema, ele não pode ser interrompido por conta de um erro, por exemplo. Pode-se fazer uma analogia do sistema B sendo uma criança que está aprendendo a falar. O adulto seria o sistema A, que ensina a criança falar corretamente, mas caso a criança fale uma palavra errada, ele deve ter a capacidade de interpretá-la e entendê-la, caso não consiga compreender a pergunta deverá ser refeita ou sugerida uma correção, assim espera-se que seja o sistema A.

Para uma melhor interação com a Web API é necessário conhecer os recursos e como realizar requisições. As Web APIs contém suporte para WADL (*Web Application Description Language*) que é uma descrição XML de uma aplicação *web* REST. Nela estão contidos o modelo dos recursos utilizados, a sua estrutura, tipos de mídia suportados, métodos HTTP e assim por diante. O WADL tem uma similaridade com WSDL (*Web Services Description Language*), porém WADL é usado para descrever recursos da API REST [Jersey 2016].

7. Estudo de Caso

O estudo de caso pode ser dividido em duas partes: uma que funciona no servidor e age como um *wrapper* para o sistema proprietário, que por questões de confidencialidade, não pode ser citado, criando uma camada de abstração para os dados gerados por este software; a segunda

parte é uma aplicação do lado do cliente, desenvolvida em AngularJS [AngularJS 2016] que consome o serviço da primeira, mostrando exatamente a forma padronizada e estruturada de acesso aos dados que antes não estavam disponíveis devido a sua arquitetura cliente-servidor. O estudo de caso foi criado durante uma experiência profissional do autor e é utilizado para evidenciar o uso do modelo arquitetural REST.

O foco do estudo de caso são os dados do módulo financeiro do software proprietário.O *wrapper* desenvolvido permite extrair e fornecer o acesso *online* aos dados do módulo de uma aplicação existente. Outras aplicações de mesmo fim podem ser integradas a ele simplesmente realizando a adaptação do módulo *wrapper* para o modelo e servidor de dados na nova aplicação comercial, exatamente o que foi feito através da segunda parte do estudo de caso, onde uma aplicação Web foi desenvolvida para consumir os dados estruturados pelo *wrapper*.

É possível, inclusive, uma futura integração deste sistema com a plataforma móvel. Portanto, este sistema se tornará hibrido, que deverá ser escalável e interoperável. A busca pelo aproveitamento da infraestrutura da *web*, pequeno esforço de aprendizado e desenvolvimento mais facilitado, foram as premissas para o desenvolvimento da integração usando o modelo arquitetural REST. A API REST (*wrapper*) a qual expões os serviços foi implementada usando o *framework* Jersey [Jersey 2016] e a interface com o usuário (sistema *web* que consome os dados) foi desenvolvida utilizando o *framework* AngularJS.

7.1. Arquitetura de um Sistema para aplicação do RESTFul

Para que o sistema seja distribuído e escalável, o projeto foi divido em duas partes, o *wrapper* que roda do lado servidor (*back end*) e lado cliente (*front end*). No lado do servidor foi construída uma Web API REST com o *framework Jersey* o qual é responsável por acessar a camada de persistência de dados e expor rotas para consulta, criação, exclusão e alteração de recursos seguindo o estilo arquitetural REST e será identificado pelo nome genérico de *wrapper*. O servidor Web Java utilizado foi o Glassfish versão 4.1.0 [Glassfish 2016]. Para criar os relatórios foi usada a ferramenta iReport versão 5.6.0 [Jaspersoft 2016]. A arquitetura implementada no servidor permite que qualquer plataforma que tenha acesso via HTTP faça uso dos recursos disponíveis. Foi usado para mapeamento objeto relacional o *framework Hibernate* [Hibernate 2016]. O banco de dados utilizado é o *MySql*, que serve como uma ponte entre os dados do servidor proprietário, e a aplicação Web desenvolvida. A Seção 7.3 que apresentará o modelo de dados da aplicação. A Web API do sistema foi desenvolvida em três camadas, cada uma com responsabilidades bem definidas.

No lado cliente o sistema foi desenvolvido em *JavaScript* usando o *framework* AngularJS e será identificado genericamente como sistema *Web*. Ele é responsável pela manipulação e exibição dos dados para os clientes finais. O sistema cliente faz requisições à Web API exposta pelo servidor, esta devolve os dados para o cliente realizar a manipulação dos dados. A Figura 5 apresenta um diagrama de interação dos participantes do processo com o sistema.

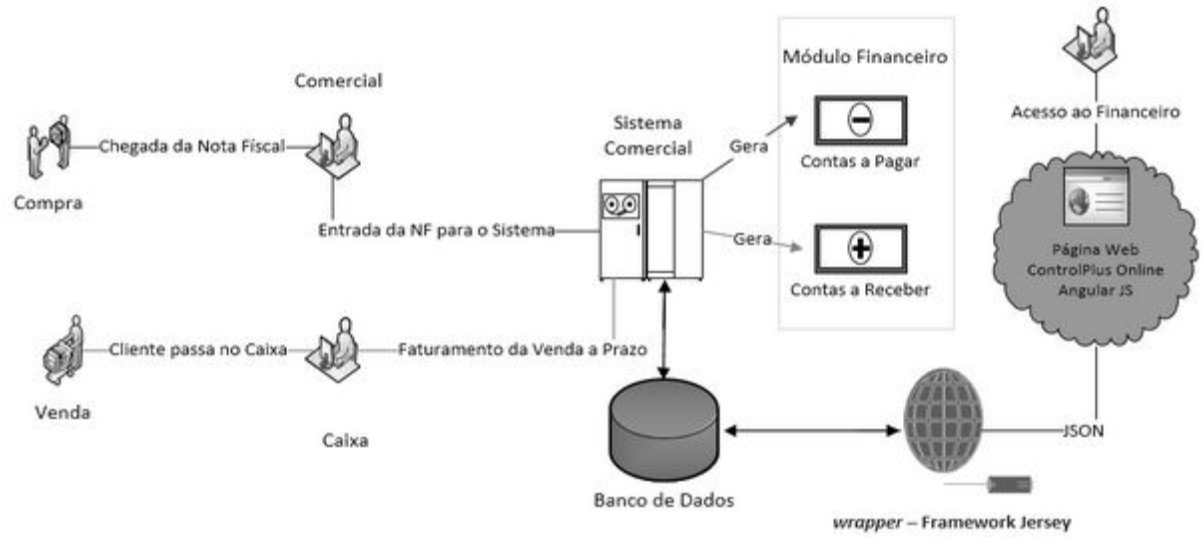

Figura 5. Diagrama de interação dos participantes do processos dos sistemas

7.2. Estruturação do Sistema

O estudo de caso foi estruturado em três camadas, visando diminuir o acoplamento. Uma delas é camada de integração a qual é responsável por receber e disponibilizar as representações dos recursos. É a única exposta aos consumidores. Ela manipula as operações de negócio e encaminha para as camadas de domínio e persistência, respectivamente. A camada de domínio contém as classes com os atributos de banco de dados. Nesta camada existem também conversores, formatadores de dados e criptografias que são solicitadas pelas camadas de persistência e integração.

Quando a aplicação cliente solicita uma representação de um recurso, imediatamente a camada de integração assume o controle. Ela valida a solicitação e aciona a camada de persistência, que realiza uma consulta no banco de dados e devolve um objeto de domínio da aplicação para a camada de integração. A camada recebe o objeto, realiza as conversões necessárias e envia para aplicação cliente. Se o cliente envia uma representação de um recurso a camada de integração realiza a conversão para um objeto de domínio da aplicação e envia para a camada de persistência realizar as operações no banco de dados de acordo com o método do recurso.

7.3. Modelo de Dados

O *wrapper* precisa extrair os dados da base proprietária para o modelo de dados exposto. Não é necessário extrair todos os dados pois a API desenvolvida atua apenas no módulo financeiro.

Futuramente, poderá ser realizada a integração com os demais módulos, bastando adicionar os recursos de acesso à Web API e ampliar o modelo de dados exposto na Figura 6.

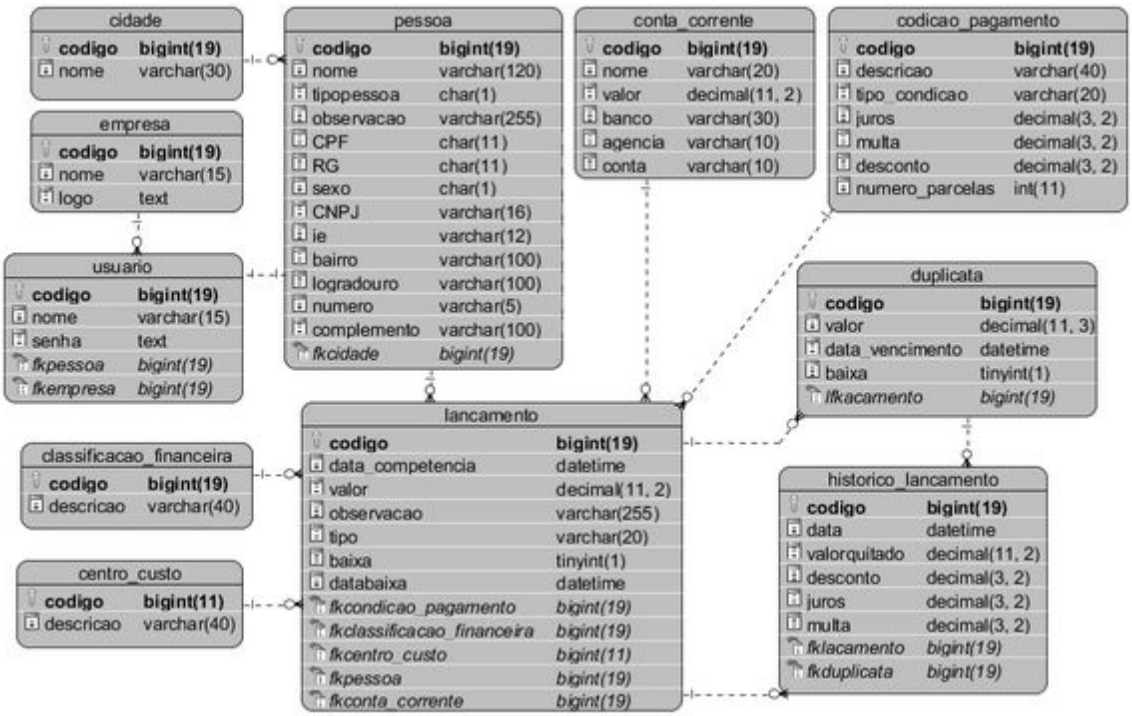

Figura 6. Diagrama de Tabelas Relacionais para o módulo financeiro que o *wrapper* utiliza.

7.4. Descrição do Wrapper

Em sua maioria os recursos da aplicação foram construídos com os métodos de inserir, alterar, consultar e excluir. O código da Figura 7 exemplifica a descrição de um recurso da *wrapper* usando WADL em uma versão resumida.

```
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.10.4 2014-08-08 15:09:00"/>
    <doc xmlns:jersey="http://jersey.java.net/" jersey:hint="This is simplified WADL with user and core
    resources only. To get full WADL with extended resources use the query parameter detail.
    Link: http://localhost:8080/wrapper/application.wadl?detail=true"/>
<grammars>
    <include href="application.wadl/xsd0.xsd">
        <doc title="Generated" xml:lang="en"/>
    </include>
</grammars>
<resources base="http://localhost:8080/wrapper/">
<resource path="/pessoa">
    <method id="inserir" name="POST">
        <request>
            <representation mediaType="application/json; charset=UTF-8"/>
        </request>
    </method>
    <method id="listarTodos" name="GET">
        <response>
            <representation mediaType="application/json; charset=UTF-8"/>
        </response>
    </method>
    <method id="atualizar" name="PUT">...
    </method>
    <resource path="/{id}">
        <method id="remover" name="DELETE">...
        </method>
        <method id="listarPessoa" name="GET">...
        </method>
    </resource>
</resources>
</application>
```

Figura 7. Descrição WADL resumida do recurso “pessoa”.

Podemos observar os detalhes do recurso `pessoa` e suas operações permitidas. A base de toda aplicação é a URL http://localhost:8080/wrapper. O recurso endereçado como `/pessoa` permite executar três operações HTTP, são eles os métodos HTTP POST, HTTP GET e HTTP PUT. O recurso contido no endereço `/pessoa/{id}` possibilita criar uma URI com campos variáveis e permite executar duas operações HTTP sobre o recurso, HTTP DELETE e HTTP GET. Em uma mesma URI não é permitido executar operações com o mesmo método HTTP, mas como podemos observar na descrição deste recurso na mesma URI são permitida várias operações com métodos distintos. No exemplo acima, *ID* é variável e identifica unicamente uma `pessoa`.

Outro recurso disponível por WADL é a descrição de dados, esta informação é formata em XSD (XML Schema) e os elementos, tipos de dados e propriedades são descritos detalhadamente.

7.5. Interface com Usuário

Para prover uma melhor interação dos usuários com o sistema foi adquirido o *Template Minovate-Angular Admin Dashboard* [Theme 2016]. Este modelo é desenvolvido em AngularJS e baseado nos padrões HTML5 e CSS3 [Theme 2016]. Sua escolha se deu pela grande disponibilidade de documentação e páginas de exemplos de uso.

Para acessar a página na *web* é necessário ter usuário e senha. As credenciais devem ser cadastradas pelo próprio usuário no primeiro acesso ao site. Os dados são validados pelo administrador que autoriza o acesso a aplicação. A Figura 8 mostra como foi projetado a página padrão do sistema.

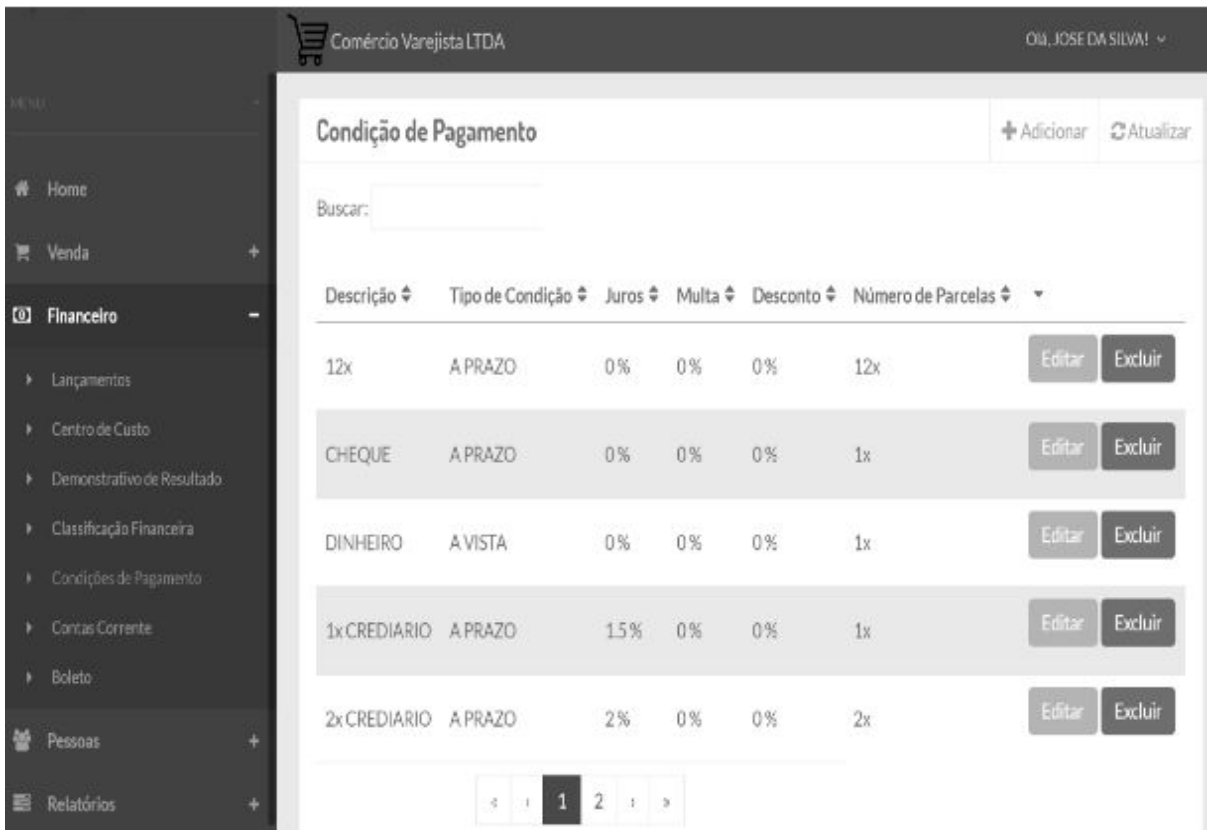

Figura 8. Tela de Manutenção de Centro de Custos.

O menu lateral contém *links* para as funções do sistema. No cabeçalho estão disponíveis o nome da empresa e sua logo. No lado direito o nome do usuário credenciado no sistema. Tanto o menu lateral quanto o cabeçalho são comuns para todas as páginas do sistema. A parte interna localizada entre o cabeçalho e o menu lateral varia de acordo com a função escolhida. Quando uma função do sistema é acessada, são consultados os cem últimos registros cadastrados no sistema. Estes dados são exibidos em formato de lista e paginados com cinco registro por página. O usuário pode realizar uma busca por registros compostos na lista em questão. Uma nova solicitação é feita ao *wrapper* somente quando o recurso não esteja contido na lista. Para realizar um cadastro, alterar e excluir algum registro, o sistema carrega uma janela modal na própria página.

Os relatórios são parametrizados pelo usuário. Na Figura 9 temos um relatório de contas a receber por período gerado no sistema. Este relatório é gerado em tela no formato de PDF.

Figura 9. Relatório de Contas a Receber por período.

8. Considerações Finais

Este trabalho apresentou uma revisão dos conceitos e tecnologias utilizadas para a construção de serviços *web*. As características e restrições de projeto do REST são apresentadas e comparadas brevemente com a abordagem SOAP. A flexibilidade é a melhor abordagem para escolha, ambas tecnologias podem realizar o trabalho necessário e trazem sua contribuição para o desenvolvimento de aplicações web. O REST foi escolhido por ser simples de entender, ter um desenvolvimento mais facilitado, aproveitamento da infraestrutura da *web* e pequeno esforço de aprendizado.

Adicionalmente, como estudo de caso, foi apresentada uma proposta de estrutura de readequação de um sistema proprietário, arquitetura cliente-servidor, através de um *wrapper* seguindo os modelos arquiteturais REST. O *wrapper* foi desenvolvido para expor as funcionalidades do módulo financeiro na forma de serviços *web* através de acesso direto ao banco de dados de uma aplicação qualquer e reimplementação da lógica de negócios.

Complementando o *wrapper,* um sistema Web cliente, foi desenvolvido em AngularJS mostrando como consumir os dados oferecidos pelo *wrapper*, dados estes, que para este estudo de caso, focaram apenas no módulo financeiro. Todos os cadastros, consultas, alterações, exclusão e alguns relatórios relacionados diretamente ao modelo de dados foram concluídos.

O estudo de caso foi implementado seguindo a restrição cliente-servidor, e suas partes *wrapper* e sistema Web cliente são capazes de evoluir separadamente por estarem desacopladas via serviços. Nenhuma requisição feita ao *wrapper* é dependente de outra, por isso é considerada sem estado e obedece à restrição *stateless* do REST.

Como resultado alcançado, podemos citar a flexibilidade de uma aplicação, que antes do estudo de caso, estava restrita apenas ao uso dentro do escritório da empresa. Agora, após a implementação, além das informações poderem estar disponíveis em tempo real, via Web, abre-se até mesmo caminhos para uso em dispositivos móveis, permitindo, no futuro até mesmo acompanhar as ações comerciais dos representantes comerciais da empresa em tempo real gerando importantes indicadores e métricas de desempenho para os gestores da empresa.

8. Referencial

AngularJS. (2016) “Superheroic JavaScript MVW Framework.” https://angularjs.org/, Abril.

Fielding, R., Thomas. (2000) “Representational State Transfer (REST)”. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, Fevereiro.

Grigorik, Ilya. (2016) “Armazemar HTTP em cache”. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching, Abril.

Glassfish. (2016) “GlassFish Server”. https://glassfish.java.net/, Abril.

Hibernate. (2016) **“**Idiomatic persistence for Java and relational databases”. http://hibernate.org/orm/, Maio.

Jaspersoft. (2016) “iReport Designer”. http://community.jaspersoft.com/project/ireport-designer, Abril.

Jersey. (2016) “RESTful Web Services in Java”. https://jersey.java.net, Abril.

Lima, R., J., Carlos. (2012) “WEB SERVICES (SOAP X REST)”. Faculdade de Tecnologia de São Paulo**.** p. 39.

Microsoft. (2016) “Visual Basic”. https://msdn.microsoft.com/pt-br/library/2x7h1hfk.aspx, Abril.

Microsoft Icr. (2016) “Microsoft Irc”. https://www.microsoft.com/pt-br/, Junho.

Mysql. (2016) “The world's most popular open source database”. https://www.mysql.com/, Abril.

Oracle. (2016) “Oracle and Sun Microsystems”. https://www.oracle.com/sun/index.html, Junho.

RFC 707. (2016) “The Goal, Resource Sharing”. https://tools.ietf.org/html/rfc707, Junho.

RFC 793 .(2016) “Tramission Control Protocol”. https://tools.ietf.org/html/rfc793, Março.

Richardson, L. e Amundsen, M. (2013) “Restful Web APIs. Oreilly & Associates Incorporated.” https://books.google.com.br/books?id=i3a7mAEACAAJ&hl=pt-BR, Março.

Rozlog, M. (2016) “REST e SOAP: Usar um dos dois ou ambos?”. http://www.infoq.com/br/articles/rest-soap-when-to-use-each, Abril.

Salvadori, L., Ivan. (2016) “Desenvolvimento de Web APIS RESTFul Semânticas Baseadas em JSON”. https://repositorio.ufsc.br/xmlui/bitstream/handle/123456789/132469/333102.pdf?sequence=1&isAllowed=y, Março.

Sebrae. (2016) “Como montar um serviço de automação comercial”. http://www.Sebra.com.br/sites/PortalSebrae/ideias/como-montar-um-servico-de-automacao-comercial, Fevereiro.

Theme Forest. (2016) “Minovate - Angular Admin Dashboard”. http://themeforest.net/item/minovate-angular-admin-dashboard/10068009, Abril..

Vigo, Leandro e Oliveira, V., Eliane. (2014) “WEB SERVICES: INTEGRAÇÃO E PADRONIZAÇÃO DE SERVIÇOS”. p. 37.

Visual Basic. (2016) “Microsoft Developer Network”. https://msdn.microsoft.com/pt-br/library/2x7h1hfk.aspx, Abril.

W3C. (2016) “Latest SOAP versions”. http://www.w3.org/TR/soap/, Fevereiro.