USO DE API RESTFUL PARA GESTÃO DE EVENTOS

Hitalo Cunha de Sousa[1]
Bruno Ramon de Almeida e Silva[2]
Júnior Marcos Bandeira[3]

Resumo: O seguinte trabalho tem como objetivo demonstrar a aplicação das restrições da arquitetura *REST* em um servidor utilizando o caso de uso do módulo de um sistema de gestão de eventos distribuído. Nesse módulo serão demonstrados a aplicação das principais restrições da arquitetura *REST*, mostrando como cada restrição busca criar uma estrutura de organização do servidor para otimização e organização de serviços distribuídos. Ao final da aplicação das restrições serão justificado o uso do termo *API RESTful* para se referir ao presente sistema.

Palavras Chaves: *Web Services, API RESTfull, REST*.

Abstract: The following work aims to demonstrate the application of REST architecture constraints on a server using the module use case of a distributed event management system. This module will demonstrate the application of the main constraints of the REST architecture, showing how each constraint seeks to create a server organization structure for optimization and organization of distributed services. At the end of the application of the restructurings will justify the use of the term *API RESTful* to refer to the present system.

Keywords: *Web Services, Api RESTfull, REST*.

1. INTRODUÇÃO

O grande crescimento no consumo de serviços compartilhados pela web contribuiu para o sucesso das redes de computadores, em reflexo desse crescimento as arquiteturas que organizam as redes também cresceram, foram melhoradas e até novas foram criadas. Com o crescimento na popularidade dos aplicativos mobiles, surgiram novas demandas e requisição de tipos de dados a servidores. Com o objetivo de servir dados de forma mais elaborada para

[1] Acadêmico do curso Sistemas de Informação / Unibalsas – Faculdade de Balsas / E-mail: hitalocunhadesousa@gmail.com
[2] Orientador – Professor do curso de Sistemas de informação / Unibalsas – Faculdade de Balsas / E-mail: brunoramonalmeida@gmail.com
[3] Orientador – Professor do curso de Sistemas de informação / Unibalsas – Faculdade de Balsas / E-mail: coord.sistemas@unibalsas.edu.br

os vários clientes em um sistema distribuído o cientista Roy Fielding elaborou o modelo de arquitetura REST (FIELDING, 2000).

O presente trabalho busca demonstrar as principais características da arquitetura *REST* no caso de uso de um sistema de gestão de eventos distribuídos, com o objetivo mostrar como as restrições da arquitetura *REST* foram aplicadas no modulo de eventos do sistema de gestão de eventos distribuídos. O presente trabalho contém os seguintes objetivos específicos:

- Demonstrar os conceitos relacionados a arquitetura *REST*.
- Demonstrar a aplicação de cada uma das restrições *REST*.
- Mostrar exemplos das restrições aplicadas ao sistema.

Foi implementado uma *API* utilizando os conceitos da arquitetura *REST*, para um sistema de para gestão de eventos com a utilização do framework de desenvolvimento *Web Laravel*, bem como a utilização do banco de dados relacional *MySQL* para armazenamento dos dados do sistema. É demonstrado onde foram aplicadas no sistema as restrições dessa arquitetura, os recursos para serem consumidos, a especificação tipos de dados em hipermídias, a construção de rotas para consumo dos clientes e exemplos de interações entre o cliente e o servidor sem guardar estado nas solicitações.

2. ESTUDO DA ARTE SOBRE ARQUITETURA REST E SUAS TECNOLOGIAS

No seguinte capítulo será apresentada uma visão geral sobre o ambiente no qual a arquitetura *REST* é aplicada desde os conceitos mais genéricos como redes de computadores, sistemas distribuídos, *Web Services*, até a arquitetura *REST*.

2.1 Redes de computadores

Desde o surgimento dos primeiros computadores uma das necessidades primordiais foi a troca de informações entre máquinas servidoras em locais distintos. Esse desafio foi superado graças à criação das primeiras redes de computadores.

Segundo Torres et al. (2009), “as redes de computadores surgiram da necessidade de troca de informações, já que é possível ter acesso a um dado que está fisicamente localizado distante de você”. O surgimento das redes de computadores está intimamente ligado à necessidade de compartilhamento de informações em larga escala e geograficamente

distantes. A necessidade da busca pela informação passou a fazer parte do cotidiano do ser humano.

Após o surgimento da grande rede de computadores a *World Wide Web*, e com o aumento na necessidade de consumo de documentos e arquivos pela internet e a popularização dos servidores os serviços tradicionais da web foram ganhando destaque. Com todo o fluxo de acesso a documentos e recursos, foram criados meios para prover acesso rápido e para o compartilhamento de recursos entre servidores a serem consumidos pelos clientes de forma rápida e transparente, esses meios passaram a ser classificados e estudados como sistemas distribuídos.

2.2 Sistemas distribuídos

Com o crescimento das redes os computadores passaram a ser cada vez mais responsáveis pelos processamentos e armazenamento dos dados. Desde então surgiu a necessidade do compartilhamento inteligente de recursos e armazenamento, sem que que tal compartilhamento, manutenção e atualização desses serviços fossem perceptíveis ao usuário e sendo realizados de forma transparente, iniciando-se assim o estudo dos sistemas distribuídos.

De acordo com Tanenbaum (2007) “Um Sistema Distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente”. A ideia de se trabalhar de forma distribuída na computação, surgiu em conjunto com a criação dos microprocessadores e a invenção das redes de computadores com as famosas Redes Locais, que “permitem que centenas de máquinas localizadas dentro de um edifício sejam conectadas de modo tal que pequenas quantidades de informação possam ser transferidas entre máquinas em alguns microssegundos” (Tanenbaum, 2007).

No geral um sistema distribuído funciona como componentes autônomos que precisam se comunicar e colaborar entre si. Esta comunicação deve funcionar de modo que o cliente, que pode ser um usuário ou outra máquina, acredite estar trabalhando em um único sistema.

Outra característica de um sistema distribuído é que a comunicação possa ocorrer independente do sistema utilizado pelos clientes ou as demais máquinas que se comunicam entre si. No geral para que um sistema distribuído possa ser construído devem-se seguir quatro pilares essenciais que são: oferecer fácil acesso aos recursos; ocultar o fato de que os

recursos estão distribuídos em uma rede; devem ser abertos e poderem ser distribuídos. Partindo dessas premissas podem-se destacar algumas habilidades que um sistema distribuído deve ter: Acesso a recursos, transparência da distribuição, abertura e escalabilidade.

2.3 Web Services

Um *Web Services* é um sistema de software criado para dispor serviços gerais remotamente pela grande rede sem a interações direta com os usuários" (TANENBAUM, 2007). Uma das particularidades é que obedecem padrões que permitem ser descobertos e acessados por outras aplicações desde que também implementem os padrões definidos. Um importante elemento da arquitetura dos *web services* são os serviços. Um serviço obedece a um padrão de descoberta e descrição universal, o que chamamos de *UDDI*[4]. O *UDDI* tem a função de prescrever o layout de um banco de dados para armazenar a descrição dos serviços. Os serviços também são descritos por outro padrão chamado de linguagem de definição de serviços *Web* (*Web Services Definition Language – WSDL*). As *WSDL* tem a função de armazenar as definições exatas das interfaces fornecidas por um serviço.

Por fim o elemento central de um *web service* é o protocolo simples de acesso a objeto (*Simple Object Acess Protocol – SOAP*), que tem a função de especificar o modo como ocorrem as comunicações no web service (TANENBAUM e STEEN, 2007).

Partindo desse princípio, um *web service* trabalha com padrões bem definidos que servem dados aos seus clientes cumprindo o que um sistema distribuído propõe. Porém no tocante a facilidade de implementação e nível de complexidade os *web services* implementados no modelo clássico são na maioria das vezes serviços complexos. E essa complexidade aumenta ainda mais quando a comunicação deve passar por cada uma dessas etapas e ainda se comunicar com outros *web services* de provedores diferentes (TANENBAUM e STEEN, 2007, p. 334).

Nesse contexto é que foi proposta uma solução por Fielding (2000), em sua tese de doutorado, propondo uma solução que obedece a padrões comuns e mais simples baseados no funcionamento do protocolo de rede *HTTP*[5].

2.4 O Protocolo *HTTP*

[4] UDDI: é um serviço de diretório onde empresas podem registrar e buscar por *web services*.
[5] *HTTP*: (Protocolo de Transferência de Hipertexto) é um protocolo de comunicação utilizado para sistemas de informação de hipermídia, distribuídos e colaborativos.

O Protocolo de Transferência de *Hipertexto HTTP*, é um protocolo de comunicação utilizado na Web para transferências de arquivos de hipertexto. Está localizado na camada de aplicação do modelo (*INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1994*). Fundado por Tim Berners-Lee, Henrik Frystyk Nielsen e Roy Fielding, o protocolo facilitou a comunicação entre sistemas de informação em rede em razão da sua flexibilidade.

Essa facilidade e flexibilidade se dão por conta do formato das requisições que possibilitam atender necessidades diferentes. O protocolo *HTTP* trabalha com requisições e respostas, estas requisições são realizadas através de verbos que definem a natureza da solicitação, os verbos *HTTP* estão dispostos na Tabela 1.

Verbo	**Função**
GET	Buscar dados.
POST	Adicionar dados.
PUT	Alterar dados.
PATH	Alterar parcialmente os dados.
DELETE	Excluir os dados.
HEAD	Recuperar informações do cabeçalho.

Tabela 1 - Tipos de verbos *HTTP*

Fonte: Adaptado de Saudate (*2013ª, p. 15*)

As requisições *HTTP* carregam informações sobre si, essas informações são chamadas de meta dados que estão localizados nos cabeçalhos *HTTP*. Apesar dos cabeçalhos não serem obrigatórios em uma requisição, eles são padronizados caso sejam enviados. Os cabeçalhos mais estão dispostos na Tabela 2.

NOME	**DEFINIÇÃO**
Host	Mostra qual endereço de *DNS* utilizado para se localizar o servidor
User-Agent	Mostra qual o meio utilizado para se chegar o servidor. Os *User-Agent*: geralmente são *Browsers* de internet
Accept	Especifica o tipo de conteúdo aceito
Accept-Language	Resolve o idioma a ser utilizado na resposta
Accept-Encoding	Resolve o método de codificação da resposta
Connection	Define se a conexão é persistente ou não

Tabela 2 - Tabela de Cabeçalhos *HTTP*

Fonte: Adaptado de Saudate *(*2013ª, p. 23*)*

Os *Media Types* são outro ponto importante nas requisições *HTTP*, pois definem o tipo de informação trafegada. Dessa forma os *Media Types* servem como padronizadores que descrevem o tipo de uma informação (BORENSTEIN; FREED, 1996). Os *Medias Types* mais comuns são mostrados na Tabela 3.

NOME	**FUNÇÃO**
Application	Tráfico de dados entre aplicações
Audio	Utilizado para formatos de áudio
Image	Utilizado para formatos de imagem
Text	Utilizados em formatos de texto legíveis a humanos
Video	Utilizado para formatos de vídeo
Vnd	Utilizado para tráfego de informações para softwares específicos

Tabela 3- Tipos de *Media Types*

Fonte: Adaptado de Saudate (*2013ª, p. 24*)

Os *Media Types* são passados nos cabeçalhos chamados *Accept*, no caso de uma requisição, e o *Content-Type*, no caso de uma resposta e são utilizados em larga escala em *web services*. As formas mais comuns de representar dados em uma arquitetura *REST* são utilizando dos formatos *XML* [6]e *JSON*[7]. Ambos são linguagens de marcação de texto, tendo suas finalidades nas aplicações.

De acordo com a (RFC 7159, 2017) o formato de Intercâmbio de Dados da Notação de Objeto *Javascript* ou *JSON*, é um formato baseado em texto, que se derivada da notação da linguagem de programação *ECMAScript*. *JSON* é um formato mais leve se comparado ao *XML*.

O formato *XML* por sua vez é uma linguagem de marcação extensível também utilizada para trafegar dados entre servidores em requisições *HTTP*, não se limitando a esta função (RFC 4825, 2007). Contudo o formato *XML* pode ser utilizado para representar dados em *Media Types* contanto que sua estrutura esteja legível a humanos (RFC 3023, 2017).

Toda requisição realizada a um servidor retorna uma resposta ao solicitante, essa resposta, para a melhor identificação, são classificadas em grupos de status que são denominados *Status Codes*. Cada grupo *Status Codes* representam uma categoria de resposta,

[6] XML: (Linguagem de Marcação Extensível) é linguagem de marcação extensível utilizada na web

[7] JSON: (JavaScript Object Notation) é um formato de padrão aberto que utiliza texto legível a humanos para transmitir objetos de dados consistindo de pares atributo-valor

sendo assim, uma categoria de resposta pode conter várias respostas relacionadas. Os grupos de respostas do protocolo *HTTP* podem ser observados na Tabela 04.

FAMÍLIA DE RESPOSTAS	**DEFINIÇÃO**
1xx	Informacionais
2xx	Códigos de sucesso.
3xx	Códigos de redirecionamento.
4xx	Erros causados pelo cliente.
5xx	Erros originados no servidor.

Tabela 04 - Tabela de *Status Codes*

Fonte: Adaptado de Saudate (*2013ª, p. 23 a 28*)

A Tabela 04 mostra que os grupos de respostas são aninhados como famílias de respostas comuns que são devolvidas ao cliente pelo servidor, afim de que este saiba lidar com a resposta recebia de acordo com sua necessidade. Um exemplo comum de resposta recebida por um servidor é o *status code* 200, que indica que uma operação teve sucesso, ou a 404, que indica que o recurso solicitado pelo usuário não existe, assim o classificando na família de respostas de erros causados pelo cliente, que no caso solicitou um recurso inexistente.

2.5 A Arquitetura REST (*Representational State Transfer*)

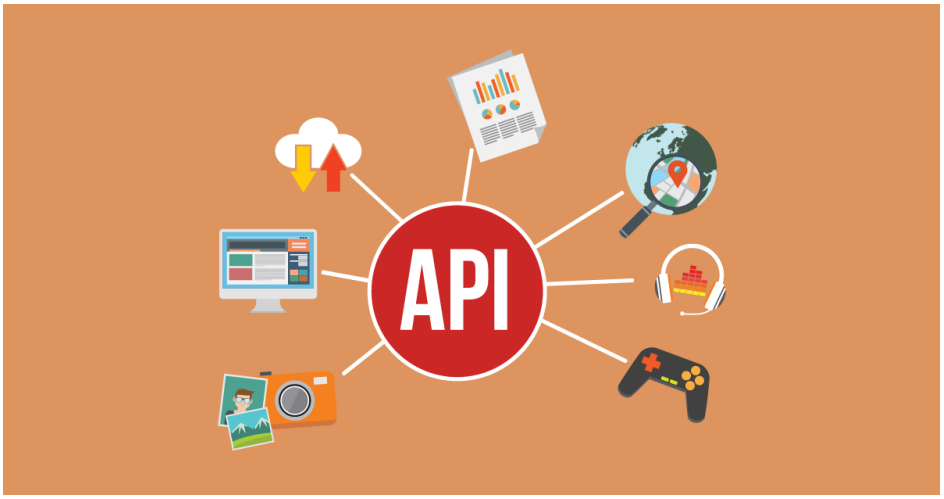
REST é uma arquitetura para se trabalhar com sistemas distribuídos e foi criada com base no protocolo *HTTP*. Criada por Roy Fielding, a arquitetura *REST* é uma arquitetura para sistemas de hipermídia distribuídos, sendo escalado como alternativa mais simples a outros sistemas comuns como o protocolo *SOAP* (*Simple Object Access Protocol*).

A REST foi criado com base em outros estilos arquitetônicos, que foram observados por Fielding (2000) e compilados em restrições conhecidas como *constraints*. As *constraints* são um conjunto de regras de estilos arquitetônicos para serem seguidos e aplicados para homologar uma arquitetura *REST*.

A primeira *constraint* da arquitetura *REST* é a *constraint Client-Server*, essa *constraint* define a separação de preocupações entre a interface do usuário e o servidor de dados. De acordo com Fielding (2000), ao se separar a interface do usuário do armazenamento dos dados melhoramos a portabilidade da interface do usuário em várias plataformas, tornando o sistema escalável, além de simplificar os componentes do servidor e permitir que os componentes

evoluam de forma independente. A Figura 1 mostra a conexão de vários dispositivos distintos a uma *API REST* que provê os dados aos dispositivos por meio de requisições *HTTP*.

Figura 1 - Exemplo arquitetura Client-Servidor em REST.



Fonte: Becode (2017).

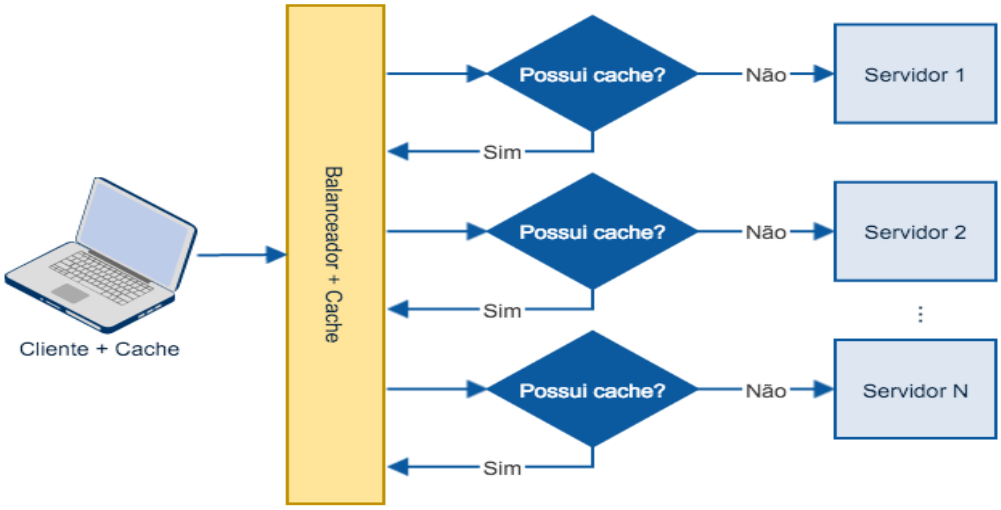
A Figura 1 mostra dispositivos distintos e com interfaces diferentes que interagem recuperando e solicitando dados a uma API RESTful. Dessa forma os clientes podem evoluir suas funcionalidades de forma independente do servidor de armazenamento e de processamento dos dados, recebendo apenas representações dos dados via *JSON* ou *XML*, e liberando do servidor a responsabilidade de oferecer uma interface visual de acesso, isso faz com que o servidor gaste recursos e se comprometa apenas com o processamento dos dados recebidos.

A segunda *constraint* da arquitetura *REST* denominada *Stateless*, define que as requisições realizadas por clientes aos servidores não devem guardar estado, ou seja, as requisições realizadas ao servidor não tem qualquer ligação com as requisições anteriores. Sendo assim cada requisição do cliente ao servidor deve conter todas as informações necessárias para o atendimento do pedido, não sendo permitido ao solicitante tirar proveito de qualquer do contexto armazenado no servidor. Portando todo o estado da sessão é de inteira responsabilidade do cliente. A principal vantagem apontada por *Fielding* (2000) no uso de requisições *stateless* é o aumento na visibilidade, confiabilidade e escalabilidade.

Na Figura 2, é ilustrado a requisição de um cliente para um servidor, onde o mesmo utiliza a estratégia da utilização de um *token* de acesso, visto que as solicitações são *stateless*.

Então o servidor responde a solicitação de um cliente enviando um *token* de autenticação para acessar os dados do servidor. Uma vez obtido um *token* de autenticação o cliente tem a possibilidade de realizar uma busca com o *token* de autenticação que fica armazenado nos cookies do cliente.

Figura 2 - Solicitação *Stateless*



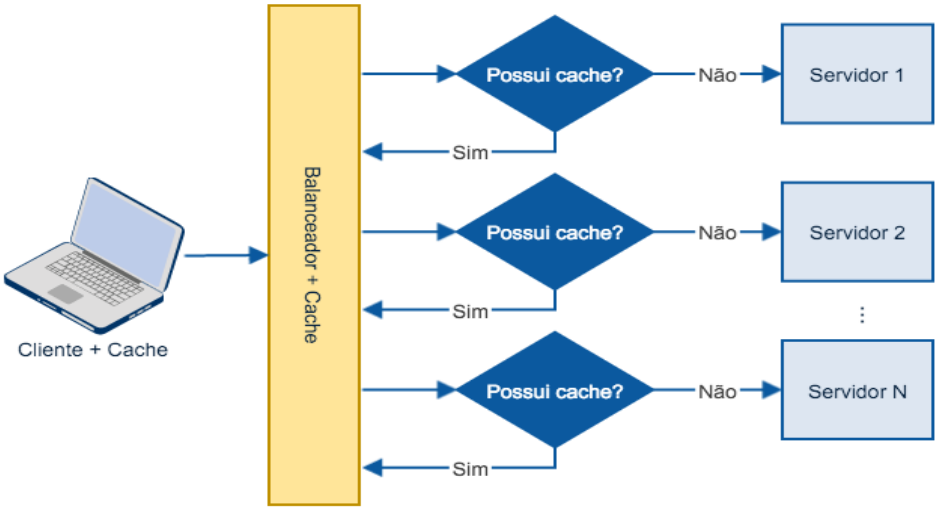
Fonte: Dias (2016, p. 14)

O grande problema ao se trabalhar com requisições *stateless* é que as requisições podem diminuir o desempenho da rede, por conta do aumento dos dados repetitivos e de sobrecarga por interação, uma vez que esses dados não podem permanecer no servidor em um contexto compartilhado (FIELDING, 2000).

Para melhorar as resposta às requisições, a arquitetura *REST* deve possibilitar políticas para que as respostas às requisições possam ser cacheadas sendo essa denominada a terceira *constraint* da arquitetura *REST*.

O cache fará o papel de balanceador de carga, a fim de tornar as aplicações mais fluidas, e evitar o processamento desnecessário em cada requisição realizada ao servidor, visto que as requisições sem estado como já vistas anteriormente podem causar repetição de dados (FIELDING, 2000). Na Figura 3, pode-se observar o papel do cache como balanceador de carga.

Figura 3 - Trabalhando com Cache



Fonte: Dias (2016, p. 15)

Nesse caso quando o cliente realiza uma requisição ao servidor e necessita novamente realizar uma outra requisição idêntica à anterior, o cliente acessa na verdade o cache. Assim o cliente tem a impressão de ser respondido mais rapidamente, quando na verdade está apenas acessando o cache que está sendo utilizado como balanceador de carga, como visto na Figura 3. A principal vantagem do uso do cache é que existe a possibilidade de eliminar parcialmente ou totalmente novas requisições que custariam ao servidor uma sobrecarga de processamento em várias requisições. Porém a desvantagem é a diminuição da confiabilidade, caso os dados obsoletos dentro do cache diferem muito dos dados que seriam recebidos caso o cliente tivesse enviado a solicitação direto ao servidor.

A *Uniform Interface* é a quarta *constraint* da arquitetura *REST*, e é a principal característica que difere a arquitetura *REST* dos demais estilos de arquiteturas baseados em rede, por defender a uniformidade entre seus componentes. Isso acontece por que é aplicado o princípio da *generalidade*[8] da engenharia de software no componente. Como resultado a arquitetura geral do sistema é simplificada melhorando a visibilidade entre os componentes, além de possibilitar a evolução independente dos componentes. A desvantagem porém é que a

[8] Generalidade - É um princípio que visa durante a resolução de um problema, descobrir se ele é uma instância de um problema mais geral, no qual a solução pode ser reutilizada em outros casos.

eficiência acaba prejudicada, uma vez que a informação é transferida de forma padronizada e pode não ser específica para as necessidades da aplicação.

Para se conseguir uma interface uniforme, é necessário orientar o comportamento dos componentes. *REST* é definido por quatro definições de interface sendo elas:

- A identificação de recursos
- Manipulação de recursos através de representações
- Mensagem auto descritivas
- E, hipermídia como motor do estado da aplicação

O ponto de partida da arquitetura *REST* são os recursos, sendo eles a principal abstração da informação na arquitetura REST. A nomeação de um recurso sempre é formado por um substantivo e nunca por um verbo. Os verbos dão a ideia de ação, porém, as ações que os recursos vão realizar são definidas pelos verbos *HTTP* como definidos na tabela 1. Dessa forma a os recursos terão todas as suas ações manipuladas pelo tipo de verbo *HTTP* passado na requisição que o cliente realiza. Os recursos são acessados e identificados por uma *URI* (*Uniform Resource Identifier*, Identificador de Recursos Uniforme) que é uma forma única para se identificar um recurso seja ele físico ou abstrato (*RFC* 3986, 2005). A Figura 4, mostra duas tabelas, uma com a implementação seguindo os padrões *REST* e outra seguindo uma implementação tradicional.

Figura 4 - Diferença entre *URI's*

Tradicional

URI	HTTP Method
/getEvent	GET
/getAllEvents	GET
/getEventsByStatus	GET
/getEventsByStatus	GET
/deleteEvent	POST
/deleteEventById	POST
/updateEventStatus	POST
/updateEventName	POST
/createEvent	POST
/createEvents	POST

REST

URI	HTTP Method
/event?tatus=ACTIVE&...	GET
/event	PUT
/event	POST
/event	DELETE
/event/1234	GET
/event/1234	PUT
/event/1234	DELETE

Fonte: O Autor

A partir da Figura 4, nota-se que nas implementações tradicionais a função dos verbos *HTTP* não são respeitadas, e isto influencia diretamente na nomenclatura das *URI's* que levam os clientes aos recursos, além de misturar o nome dos recursos com as funções. Em uma implementação seguindo os padrões *REST*, a nomenclatura se torna mais dinâmica e padronizada tendo sua função bem definida por delegar as ações aos verbos *HTTP*, simplificando a nomenclatura das *URI's*. O exemplo pode ser observado na *URI* de recuperação de todos os eventos, esta *URI* é composta pelo método *GET*, que tem a função de recuperar dados assim como disposto na Tabela 1, e é seguida pela *URI /event*, sendo assim a API entende que o cliente solicita todos os eventos da aplicação, e caso seja necessário recuperar um evento individual a *URI* de acesso é */event/1234*, adicionando somente um identificador e utilizando o mesmo verbo que tem a função de recuperação de dados.

A quinta *constraint* da arquitetura *REST* é o *Layered System*, que define que uma arquitetura deve ser composta de camadas hierárquicas ao restringir o comportamento dos componentes.

Para que um sistema seja escalável é necessário adicionar elementos intermediários que trabalhem de forma transparente ao cliente. Partindo desse princípio a arquitetura *REST* deve prover a capacidade de se trabalhar com camadas de forma transparente ao cliente, ou seja, o cliente não deve perceber as camadas do sistema. Um exemplo a se observar é o sistema de *DNS*, se toda vez que um usuário tivesse a necessidade de acessar o domínio do Google fosse obrigado a decorar o *IP* do *Google*, talvez o conceito de web como é visto hoje não existiria. Nesse contexto o *DNS* faz o papel de uma camada intermediária entre o cliente e o recurso que o cliente queira acessar. Outro bom exemplo de camada são os balanceadores, que permitem adicionar mais servidores a aplicação fazendo isso com total transparência ao cliente (DIAS, 2016).

A vantagem do uso de camadas em uma arquitetura é a extensibilidade dos códigos, o baixo acoplamento. Um código é considerado extensível quando se é possível realizar alterações e melhorias, seja manutenção, ou adicionar uma nova funcionalidade sem quebrar o código. O código sob demanda tem o mesmo objetivo, procurando adaptar o cliente de acordo com as novas funcionalidades e novos requisitos do sistema (FIELDING, 2000).

A desvantagem ao se utilizar um sistema em camadas é que eles aumentam a sobrecarga e a latência quando os dados são processados, e isso se torna perceptível ao

usuário, porém, essa sobrecarga pode ser compensada pelo armazenamento do cache compartilhado entre os intermediários.

Quando um servidor contém uma API que aplica todas as *constraints* da arquitetura REST é dito que esse servidor é uma API RESTful, pois o seguinte termo nada mais é que um termo utilizado para explicar que a API aplica todas as *constrains* da arquitetura REST, visto que podem ocorrer casos nos quais as *constraints* são todas aplicadas, ou quando são isso é feito de forma errônea ou parcial (DIAS, 2016).

2.6 O Framework Laravel

Laravel é um framework *MVC* [9]criado por Taylor Otwell para desenvolvimento de aplicações web. *Laravel* é escrito na linguagem de programação PHP, e implementa as boas práticas de programação, como padrões de projeto, além de seguir parte das especificações das *PSR's*[10].

Uma das características do *Laravel* que se adequam bem a arquitetura *REST* é o fato de que o *Laravel* trabalha com pacotes através do gerenciador de dependências do PHP, tratando as funcionalidades do framework como módulos separados por pacotes, ou mesmo pequenos componentes de um sistema principal, essa característica se assemelha e muito com uma das restrições do *REST* código sob demanda, como já tratada acima (LARAVEL, 2012).

Outro componente vindo por padrão no *Laravel* são as rotas que possibilitam trabalhar com recursos, ou quaisquer métodos *HTTP*, tornando o *Laravel* um forte candidato trabalhar com REST (TURINI, 2015).

2.7 JSON Web Tokens

JSON Web Tokens é um padrão da web que visa representar reivindicações entre duas partes de uma comunicação de forma segura, permitindo a seus utilizadores descodifica verificar e gerar novos *tokens* no padrão *JWT*. As reivindicações são representadas como um objeto *JSON*, que é utilizada como uma estrutura de carga útil, permitindo uma espécie de assinatura digital de mensagens protegidas com um código de autenticação (RFC 7519, 2015).

[9] MVC – (Model-View-Controller) Estrutura de organização de código que separa modelo, visão e controle das operações.
[10] PSR – (PHP Standards Recommendation) Conjunto de boas práticas recomendadas para desenvolvedores PHP.

JWT são comumente utilizadas na troca de mensagens entre dispositivos móveis e servidores web, buscando proteger a integridade dos dados, visto que uma das características mais importantes do *JWT* é de identificar se os dados foram alterados no meio da requisição ou se o remetente é de fato autêntico.

3. ESTUDO DE CASO UTILIZANDO API RESTFUL

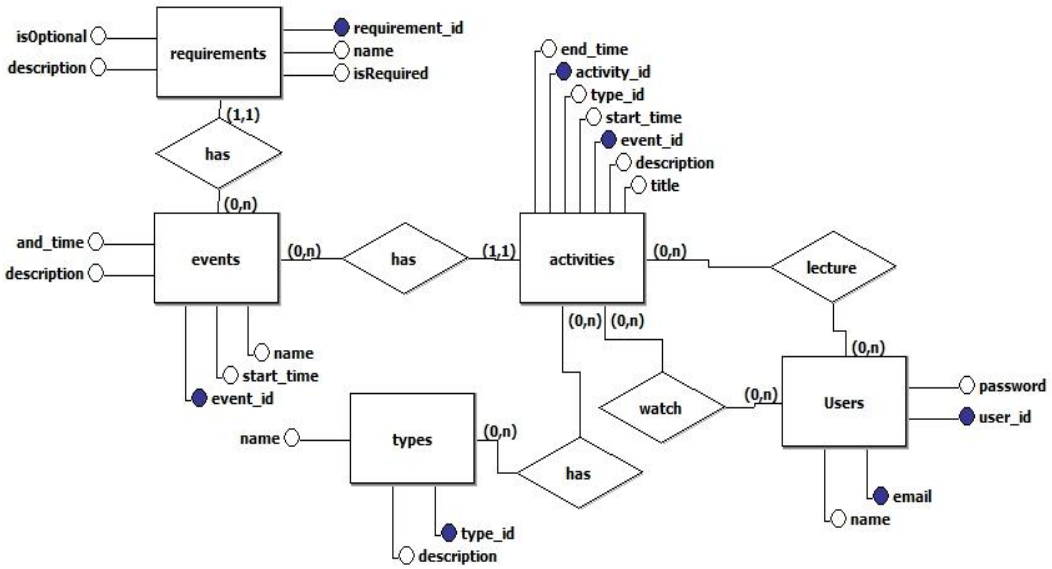
Para o melhor entendimento das técnicas e conceitos que estão reunidos na arquitetura REST, foi proposta a criação de uma *API RESTful* para gestão de eventos. As técnicas e conceitos demonstrados no presente trabalho podem ser utilizados em qualquer linguagem de programação ou framework, porém, o presente trabalho demonstra toda a codificação utilizando o *framework PHP Laravel*, também serão utilizadas outras ferramentas, que serão demonstradas no decorrer do desenvolvimento.

O caso de uso utilizado na construção da *API RESTful* do presente artigo, demonstra a aplicação dos conceitos da arquitetura *REST* na construção de um sistema de gestão de eventos distribuídos. O sistema foi criado para servir como base de informações e gerência de dados para aplicações mobile, com o objetivo de centralizar informações sobre eventos como: a disponibilidade dos eventos, cidade onde o evento vai ocorrer, cadastro, cronograma de atividades, notificação de informações e inscrição de membros.

3.1 Construção do Banco

A primeira ação a ser tomada ao iniciar um novo projeto é a criação do banco de dados. A partir do banco de dados poderão ser gerados e armazenados os recursos para serem devolvidos aos clientes da *API*. Na Figura 5, é mostrado o Modelo Conceitual do banco de dados criado para o sistema de gestão de eventos.

Figura 5 - Modelo Conceitual



Fonte: Autor

O SGBD utilizado para a criação do banco de dados foi o *MySQL*. Em linhas gerais o *MySQL* é um banco de dados otimizado e livre para *Web*, tem a capacidade de armazenar e devolver de forma rápida consultas utilizando a linguagem *SQL*. O *MySQL* atende as necessidades de armazenamento da *API*. Porém, com o crescimento da *API*, necessidades de mudanças na base de dados podem surgir. Essas mudanças muitas vezes podem ser dificultosas e até gerar erros que podem ocasionar na indisponibilidade da *API*. Algumas operações comuns que surgem nos projetos que podem ocasionar esses problemas são: A necessidade da migração para outro banco de dados, alteração das colunas das tabelas do banco ou até mesmo a forma como os dados são consultados. Todas essas atividades geram a necessidade de alteração no código das consultas, e consequentemente afetam o conjunto dos componentes relacionados.

Contudo para resolver esses problemas e começar a aplicar as *constraits* definidas pela arquitetura *REST*, é necessária a aplicação das restrições a começar por *Layered System*. Como citados em capítulos anteriores a *Layered System* define que sejam criadas camadas intermediárias com funções bem definidas para restringir o comportamento dos componentes, assim, o componente não poderá ver a camada imediata a qual está interagindo. No caso da ocorrência de um erro o componente estará isolado, não afetando os demais componentes, podendo ser manutenível e permitindo a escalabilidade e o balanceamento de carga, na medida que o componente pode evoluir de forma independente.

Porém é necessário encontrar ou desenvolver uma solução com essas características, e é nesse contexto onde utilizamos os *ORM's* (Mapeamento objeto-relacional). Os *ORM's* criam uma camada de comunicação com o banco de dados, criando objetos que são transformados em entidades no banco de dados. Para a criação do banco de dados foi utilizado o *ORM* do *Laravel* o *Eloquent ORM*. Na Figura 06, pode-se observar outro recurso do *Laravel* chamado *Migration* que são responsáveis por auxiliar na criação das tabelas do banco de dados, no exemplo da figura está mostrando a criação da tabela de eventos com suas respectivas colunas.

Figura 6 - Migração da tabela de eventos

```
14      public function up()
15      {
16          Schema::create('events', function (Blueprint $table) {
17              $table->increments('id');
18              $table->string('name', 80);
19              $table->string('description')->nullable();
20              $table->datetime('start')->nullable();
21              $table->datetime('end')->nullable();
22              $table->timestamps();
23          });
24      }
```

Fonte: Autor

O código da Figura 6 mostra a implementação do método *up* da classe de eventos herdada de *Migrations*. Dentro da classe é instanciado de forma estática a classe Schema que é responsável pela criação das tabelas do banco de dados através do método create. O método *create* recebe 2 parâmetros, um é o nome da tabela do banco de dados a ser gerada, e o outro, recebe a injeção de uma outra classe chamada *Blueprint*, responsável por seta as colunas da tabela referida do primeiro parâmetro.

Geradas as migrações de todas as tabelas mostradas na Figura 04, agora a API já tem a possibilidade de gerar as tabelas dos bancos de dados independente de qual banco foi selecionado, uma vez que todas as entidades geradas são classes do PHP, o banco de dados foi gerado baseado nas migrações. Nessa aplicação foi utilizado o banco de dados *MySQL*.

Uma vez que as migrações foram geradas, o modelo pode ser manipulado pelo *Laravel* para controlar as tabelas no banco de dados e manipular as entidades do banco sem

comprometer diretamente o as outras camadas do sistema criando a primeira *Layered System* da aplicação.

3.2 Estruturações dos Recursos

As *API's RESTful* são formadas por recursos, que são a principal abstração da informação de uma API. Os recursos são os dados retornados a quem realiza um pedido a uma API. Porém de acordo com a *constraint* de *Uniforme Interface*, esses pedidos devem seguir um padrão, que por sua vez também devolvem uma resposta também padronizada. A começar pelos recursos, os seguintes capítulos demonstrarão a padronização dos pedidos de realizados por clientes a aplicação, bem como, as respostas devolvidas e sua padronização.

A partir dos recursos as *API RESTful* pode ser representadas por meio de um formato padrão como *JSON* ou *XML*. Os recursos por sua vez podem ser localizados por meio de uma *URI* que compõe uma *URL* de acesso, pela qual um cliente pode acessar ou solicitar modificações nos recursos. Para a criação das *URI's* da aplicação de gestão de eventos foi utilizado o recurso de rotas do *Laravel* conforme disposto na Figura 7.

Figura 7 - Rotas de URI

```
14  Route::prefix('v1')->group(function () {
15      Route:get('event', 'EventController@index')
16      Route:get('event/{id}', 'EventController@show')
17      Route:post('event', 'EventController@store')
18      Route:put('event/{id}', 'EventController@update')
19      Route:delete('event/{id}', 'EventController@delete')
20  });
```

Fonte: Autor

A Figura 7, mostra o mapeamento do recurso de Eventos da aplicação, das linhas 6 a 10 estão especificados a *URI*, o verbo *HTTP*, e o controlador correspondente que dará a resposta a requisição. Assim como disposto na Figura 7, as *URI's* do sistema de gestão de eventos foram criadas seguindo a especificação da constraint *Uniform Interface*, separando no nome das *URI's* das ações a serem tomadas pelas requisições e delegando essa responsabilidade aos métodos *HTTP*. A Tabela 5, mostra o mapeamento das rotas do recursos de eventos de acordo com a função e a classe com os métodos correspondentes que responde pela ação.

Método *HTTP*	**URI**	**Método do Controlador**	**Ação**
GET	api/v1/event	Index	Lista todos os eventos.
GET	api/v1/event/{id}	Show	Seleciona um evento.
POST	api/v1/event	Store	Cria um novo evento.
PUT	api/v1/event/{id}	Update	Atualiza dados do evento.
DELETE	api/v1/event/{id}	Cestroy	Exclui um evento.

Tabela 5 - Mepeamento dos recursos por rotas

Fonte: Autor

Na Tabela 5, mostra que cada *URI* leva a um método específico da classe controladora *EventController* que é responsável por devolver as respostas aos clientes.

A *cosntraint Uniform Interface* define ainda o uso correto dos status codes, que são as respostas dadas ao servidor quando determinada ação acontece, como a criação de um novo recurso. A Figura 8, mostra a implementação do método *store* da classe *EventController*.

Figura 8 - Implementação do Método Store

```
55      public function store(Request $request)
56      {
57        try {
58          $event['data'] = Event::create($request->all());
59          return response()->json($event, 201);
60        } catch(Exeption $errors) {
61          $response['error'] = $errors;
62          return response()->json($response, 400);
63        }
64      }
```

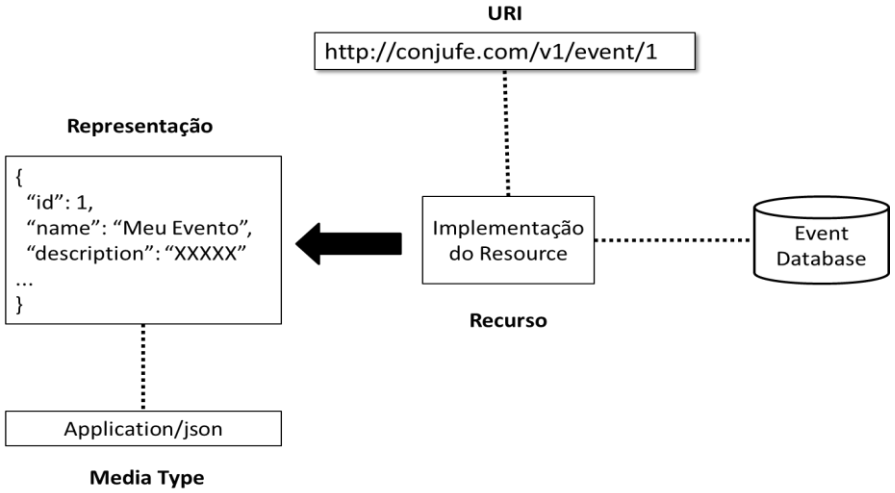
Fonte: Autor

A Figura 8 mostra a implementação do método *store*, após a criação do recurso de evento, o código retornado para o servidor é o código correspondente a criação de um recurso, que no caso é o 201. O código 201 indica que um novo recurso foi criado retornando ao servidor a mensagem *201 Created*, ou no caso de falha o código *400 Bad Request*, que indica que a requisição não foi bem feita.

As respostas devolvidas aos clientes de uma *API RESTful* são chamadas de representações. As representações são a forma como um recurso pode ser representado, sendo os tipos mais comuns *XML* e *JSON*. A constraint *Uniform Interface* define que os clientes

solicitem o tipo de representação que eles desejam obter do recurso através dos cabeçalhos de requisições do protocolo *HTTP*. A Figura 9, mostra o fluxo da informação desde a solicitação de um recurso através de uma *URL*, até a devolução da representação do recurso solicitado.

Figura 9 - Fluxo da informação representada



Fonte: Autor

Os *Media Types* são o tipo de dado que o recurso vai ser representado. A partir da Figura 9, é observado que o recurso é representacional, ou seja, o que o cliente obtém no resultado de uma requisição é uma representação do recurso que nesse caso é *JSON* e interage com ela através de uma *URL* de localização do recurso e utilizando métodos *HTTP* para indicar as ações a serem tomadas. A Figura 10 mostra o exemplo de uma requisição ao método *store* conforme disposto na Figura 9, utilizando a ferramenta de linha de comando *cURL* passando o cabeçalho de requisição *Content-Type* como valor *application/json*.

Figura 10 - Recuperação dos dados criados

```
hitalo@hitalo:~$ curl -i \
> -X POST http://localhost:8000/api/v1/event \
> -H "Content-Type: application/json" \
> -d '{"name": "Conjufe Norte", "description": "Descrição do Evento"}'
HTTP/1.1 201 Created
Host: localhost:8000
Connection: close
X-Powered-By: PHP/7.0.22-0ubuntu0.16.04.1
Cache-Control: no-cache, private
Date: Mon, 06 Nov 2017 11:35:36 GMT
Content-Type: application/json
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 59

{"data":{"name":"Conjufe Norte","description":"Descri\u00e7\u00e3o do
Evento","updated_at":"2017-11-06 11:35:36","created_at":"2017-11-06 11
:35:36","id":18}}
```

Fonte: Autor

Dessa forma uma requisição é realizada solicitando que o formato de representação dos dados seja *JSON*, modificando o *Media Type* pelo parâmetro *-H*, e enviando no corpo da requisição os dados para serem cadastrados conforme previsto no parâmetro *–d* e passando o método *HTTP* como *POST* e em seguida a URI de acesso ao recurso conforme observado no parâmetro *-X*. O resultado da requisição é o *HTTP/1.1 201 Created*, que indica que o recurso foi criado com sucesso. Também é retornado a representação em JSON do novo recurso criado e os demais cabeçalhos padrões da requisição.

A última técnica utilizada na *constraint Uniform Interface* é a *HATEOS* (*Hypermedia As The Engine Of Application State*), que nada mais é do que a *Hypermedia* como o motor do estado da aplicação. A implementação do *HATEOS* em um servidor possibilita a navegabilidade do cliente entre os recursos, tendo como benefício o desacoplamento do cliente e do servidor possibilitando que o servidor evolua de forma independente. A Figura 11, demonstra a implementação do HATEOAS na resposta pela busca do recurso *activity*.

Figura 11 - Adicionando Links HATEOAS

```
{
    "data": {
        "id": 1,
        "title": null,
        "description": "Palestra da Tarde",
        "start_time": "2017-11-06 17:33:27",
        "end_time": "2017-11-06 17:33:27",
        "created_at": null,
        "updated_at": null
    },
    "links": {
        "self": "http://localhost:8000/api/v1/activity/1",
        "event": "http://localhost:8000/api/v1/event/1",
        "type": "http://localhost:8000/api/v1/type/1"
    }
}
```


Fonte: Autor

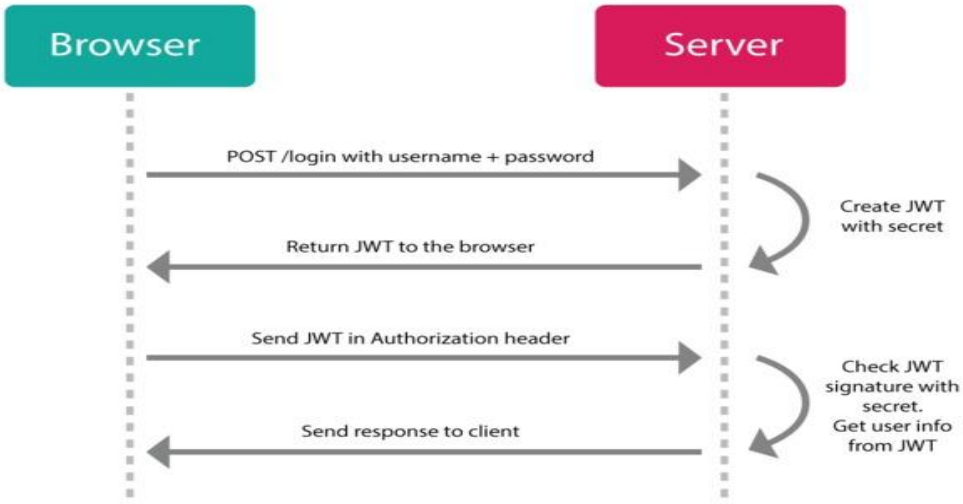
A Figura 11 mostra a resposta com a adição de um vetor de links, que possibilitam a navegação entre os recursos através da *URL*. Dessa forma caso a *URL* do evento mude de versão, os recursos relacionados a ele não sofrerão muito impacto. Também existem benefícios quanto a performance da busca e carregamento, uma vez que os dados não são completamente carregados e partes deles são referenciados.

3.3 Autenticação de clientes e Cache

Nos modelos tradicionais de desenvolvimento de aplicações é comum o uso de mecanismos de autenticação para proteger dados de usuários. Porém a arquitetura *REST* exige

pela *constraint Stateless* que os dados sejam trafegados sem guardar estado. Essa abordagem acaba anulando o uso de mecanismos de autenticação como sessões que são vastamente utilizados na *WEB*. Nesse contexto a seguir serão vistas estratégias para aplicação de autenticação sem estado *API's RESTful*, bem como, a resolução de problemas causados pela consequência do uso desse modelo de iteração. Uma opção de autenticação de clientes sem guardar estado é com o uso de *JWT (JSON Web Tokens)*. O *JWT* gera um *token* de acesso para um cliente e esse *token* se torna o passaporte de acesso aos recursos da aplicação, sendo esse passaporte o único meio de acesso aos dados do cliente. A Figura 12 mostra o esquema de autenticação de um cliente em uma aplicação.

Figura 12 - Fluxo de autenticação JWT



Fonte: JWT.io (2017)

Ainda na Figura 12, o cliente que nesse caso é um *Browser*, envia através do método *POST* o nome de usuário e a senha, o servidor por sua vez devolve ao cliente um *token* de acesso, o cliente usa esse token no cabeçalho da requisição *HTTP*, e solicita acesso a um recurso, caso o token seja válido o usuário recebe o recurso. A vantagem dessa abordagem é que toda a responsabilidade sobre os dados de acesso é passada para cliente, livrando a aplicação de guardar estado e cumprindo com a *constraint Stateless*. A Fugura 13, mostra a autenticação por meio de JWT do sistema de gestão de eventos.

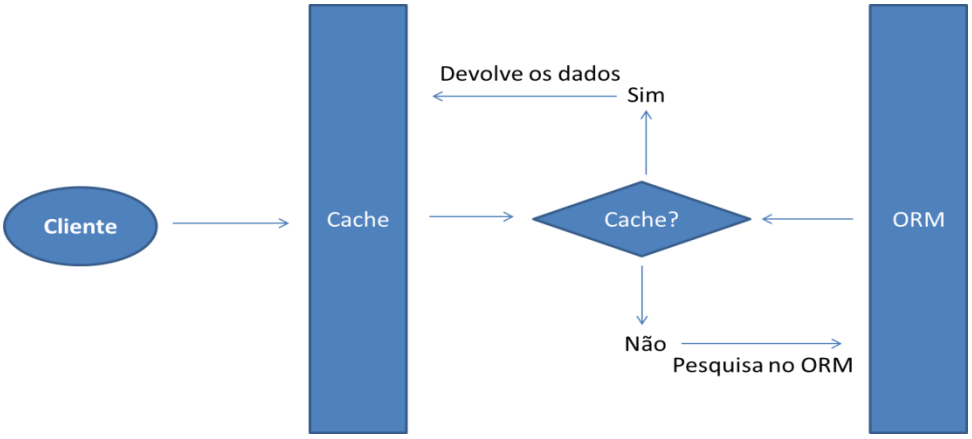
Figura 13 - Método de autenticação da aplicação

```
12      public function authenticate(Request $request) {
13        $credentials = $request->only('email', 'password');
14        try {
15              if (!$token = JWTAuth::attempt($credentials)) {
16                  return response()->json(['error' => 'invalid_credentials'], 401);
17              }
18          } catch (JWTException $e) {
19              return response()->json(['error' => 'could_not_create_token'], 500);
20          }
21          return response()->json(compact('token'), 201);
22      }
23  }
```

Fonte: Autor

A Figura 13, demostra como foi configurado o mecanismo de autenticação de clientes na *API RESTful*, na linha 15 realiza a checagem das credências do cliente, caso sejam corretas o usuário recebe um *token*, conforme previsto na linha 21, caso contrário na linha 16 recebe uma resposta de credenciais inválidas com *status code 401*. O grande problema em se trabalhar com requisições Steteless é a carga que é gerada nos processados além da perda de performance, visto que todas as vezes que houver a necessidade de um cliente solicitar dados no sistema, servidor tratar aquela requisição somo nova. Tendo em vista esse problema a última constraint abordada no presente artigo é a *constraint* de *Cache*. Essa *constraint*, define que as respostas do servidor devem ser armazenadas em cache, para que o *cache* trabalhe como um balanceador de cargas. No *Laravel* o cache é trabalhado de forma nativa bastando porém os detalhes sobre o funcionamento do cache na aplicação podem ser observador conforme a Figura 14.

Figura 14 - Cache dos Resultados



Fonte: Autor

3.3 Credenciando *RESTfull*

O presente módulo de sistema do sistema de gestão de eventos construído aplicou e implementou todas as *constraints* da arquitetura *REST* dessa forma a *API* construída utilizando essas *constraints* é chamada de *API RESTful*, uma vez que foram aplicadas todas as restrições que a arquitetura exige para ser considerada *RESTful*, que foram, a *constraint Client-Server,* para separação da interface visual do sistema com o servdor, *Stateless*, para realizar requisições sem guardar o estado, *Cache* para balancear a carga das requisições sem estado da restrição anterior, *Layered System*, uma vez que o sistema foi separado em camadas bem definidas com suas funções, *Uniform Interface*, para aplicação do *HATEOAS* e controle de representações e de hipermídias distribuídos e a *constraint* opcional da arquitetura *REST Code On Demand*, uma vez que a aplicação pode evoluir de forma gradativa criando várias funcionalidades de acordo com a necessidade.

4. CONSIDERAÇÕES FINAIS

No presente trabalho foi apresentado uma abordagem prática sobre as principais características da arquitetura de sistemas *REST*. Através do estudo e do uso de referências bibliográficas apresentadas ao longo do artigo. Nos capítulos de revisão bibliográficas foram abordadas toda a os conceitos por traz da arquitetura *REST*, como a importância do protocolo *HTTP* para a arquitetura, bem como, a correta implementação e uso dos cabeçalhos e retornos com códigos de status e o uso correto dos verbos *HTTP*, na realização de requisições a um servidor. Foram expostas técnicas e características de organização de hipermídias, e essas características foram aplicadas em cima da demonstração do módulo principal de um sistema de gestão de eventos, que foi o módulo de eventos, onde foi apresentado, através do uso de imagens que demonstravam esquemas de organização e pedaços de códigos, os benefícios e dificuldades do uso da arquitetura. No presente trabalho foram aplicadas as *constraints* da arquitetura *REST*, com a aplicação de todas as *constraints* obrigatórias que são *Client-Server*, *Stateless*, *Cache*, *Layered System* e *Uniform Interface*, pode-se concluir que aplicação é uma aplicação *RESTful* pois segue todos as *constraint*, da arquitetura *REST*. Contudo conclui-se que o uso da arquitetura *REST* na construção de Web Services para resolução de problemas na Web moderna, é uma ótima opção, visto as vantagens de escalabilidade, separação de componentes, sistema de hipermídias distribuídos. O seguinte trabalho será utilizado como base para construção do sistema de controle do evento chamado CONJUFE realizado pela Igreja Batista Filadélfia, o sistema irá servir dados a aplicativos mobile para consumo.

REFERÊNCIAS

BECODE - O que é API? REST e RESTful? Conheça as definições e diferenças!. Disponível em: <https://becode.com.br/o-que-e-api-rest-e-restful/>. Acesso em: 20.11.2017.

BENTO, Evaldo J. Desenvolvimento web com PHP e MySQL. São Paulo. Casa do Código. 2013.

BORENSTEIN, Nathaniel; FREED, Ned. Multipurpose Internetmail extensions (mime) part two: Media types. 1996. Disponível em: https://tools.ietf.org/html/rfc2046.

DIAS, Emílio. Desmistificando REST com Java. 1a Edição, 2016. Disponível em: <http://cafe.algaworks.com/livreto-desmistificando-rest-com-java/>. Acesso em: 15.05.2017.

FIELDING, Roy Thomas. Architectural styles and the design of networkbased software architectures. 2000. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>. Acesso em: 25.05.2017.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. Open systems interconnection – basic reference model: The basic model. 1994.

JWT.io - Introduction to JSON Web Tokens. Disponível em <https://jwt.io/introduction/> Acesso em: 20.11.2017.

LARAVEL. Laravel API. Disponível em: <https://laravel.com/api/5.4/>. Acesso em: 15.05.2017.

RFC 3023- XML Media Types. Disponível em: https://www.ietf.org/rfc/rfc3023.txt>. Acesso em: 09.10.2017.

RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax. Disponível em: <https://www.ietf.org/rfc/rfc3986.txt>. Acesso em: 29.05.2017.

RFC 4825 - The Extensible Markup Language (XML) Configuration Access Protocol (XCAP). Disponível em: <https://tools.ietf.org/html/rfc4825>. Acesso em: 09.10.2017.

RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format. Disponível em: <https://tools.ietf.org/html/rfc7159>. Acesso em: 09.10.2017.

RFC 7230 - Hypertext Transfer Protocol (*HTTP*/1.1): Message Syntax and Routing. Disponível em: <https://tools.ietf.org/html/rfc7230#section-1.2>. Acesso em: 29.05.2017.

RFC 7519 - JSON Web Token (JWT). Disponível em: https://tools.ietf.org/html/rfc7519#section-1>. Acesso em: 19.06.2017.

SAUDATE, Alexandre. Rest construa API’s inteligentes de forma simples. 1a ed. São Paulo: Casa do Código, 2013a.

SAUDATE, Alexandre. Soa aplicado: integrando com web services e além. Casado Código, 2013b.

TANENBAUM, Andrew S.; STEEN, Maarten V. Sistemas Distribuídos Princípios e Paradigmas. 2º edição. São Paulo - SP, Paerson Pretice Hall, 2007.

TORRES, Gabriel. Redes de Computadores: versão revisada e atualizada. 1a edição. Rio de Janeiro – RJ, NovaTerra, 2009, 832 p.

TURINI, Rodrigo. PHP e Laravel Crie aplicações web como um verdadeiro artesão. São Paulo, Casa do Código. 2015.