GERENCIADOR DE WEB SERVICE E ORGANIZADOR DE REQUISIÇÕES

Matheus Heiden, Luciana Pereira de Araújo Kohler – Orientadora

Curso de Bacharel em Ciência da Computação
Departamento de Sistemas e Computação
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brazil

matheiden@furb.br, lpa@furb.br

Resumo: *Este artigo apresenta uma aplicação gerenciadora de Web Services SOAP e REST. A aplicação foi desenvolvida utilizando o interpretador de código JavaScript Node.js. Sua implementação foi feita de forma modular, para que assim seja possível expandir a aplicação através de módulos desenvolvidos dentro de um padrão definido pela aplicação. O diferencial da aplicação apresentada é a sua customização fornecida por sua modularidade, baixo consumo de memória e adaptação a escopos através de suas configurações. Teve como resultados pouca diferença no tempo de execução das requisições à API, um baixo nível de consumo de memória da aplicação. Ao utilizar configurações mais agressivas foi detectada uma diminuição efetiva no uso de processamento do servidor em que se encontra a API, graças às mitigações de requisições fornecidas pelo gerenciador de API. Também foi constatada a utilidade da API fornecida pela aplicação através da implementação de uma aplicação utilizando ReactJS.*

Palavras-chave*: Gerenciador de API. Soap. Rest. Node.js. Web Services.*

1 INTRODUÇÃO

Nos últimos anos, com o crescimento da implementação e utilização de aplicações web, criou-se também a necessidade de interfaces de programação de aplicação como Web Services (NASCIMENTO, 2014). Plataformas como Facebook (FACEBOOK, 2018) e Google (GOOGLE, 2018), oferecem Application Programming Interfaces (APIs) através de Web Services, para que desenvolvedores possam criar integrações com as suas aplicações. O interesse na implementação dessas APIs vem crescendo desde 2006 e em janeiro de 2018 foram contabilizadas cerca de 19 mil APIs de plataformas (SANTOS, 2018).

Uma API ajuda a expor um serviço ou dados a desenvolvedores de aplicativos, portanto, uma API é uma interface de software para software que define contratos para que aplicações conversem através de uma rede sem interação de um usuário (DE, 2017). Os contratos definidos durante a comunicação têm como base protocolos, podendo também incluir informações como os termos de serviço e acordos de licenciamento para utilização do serviço, bem como preços (DE, 2017).

Com o maior interesse de empresas e clientes na integração de aplicações, também cresce a demanda dos servidores web nos quais essas aplicações se encontram. Essa demanda pode causar lentidão ou até indisponibilidade da aplicação. Tarjan (2017, p1, tradução nossa) afirma que “Disponibilidade e confiabilidade são primordiais para todas aplicações web e APIs.”. Para garantir esses elementos é possível escalonar servidores para acomodar o crescimento de demanda e de usuários, mas ainda é necessário garantir que um atuador com alta demanda de requisições não afete acidentalmente ou deliberadamente a disponibilidade da API (TARJAN, 2017). Com isso, tem-se uma ligação direta entre a demanda de requisições a um Web Service e sua disponibilidade, além de que com o crescimento da demanda a necessidade de controlá-la aumenta.

Para facilitar esse controle de requisições existem gerenciadores de APIs que possibilitam a análise e gerenciamento de APIs, para conseguir isso tem como funcionalidades o serviço de gateway de APIs e o serviço de estatísticas de API (DE, 2017). A aplicação demonstrada por esse artigo tem como objetivo operar entre o cliente e o Web Service, podendo receber requisições REST e SOAP, que as prioriza, valida e encaminha ao Web Service.

Além disso, têm-se como objetivos específicos fornecer um ponto de entrada que recebe requisições e disponibilizar uma interface para o registro de Web Services que terão as requisições gerenciadas, além de disponibilizar uma interface para a definição de prioridade e regras de utilização de requisições. Outro objetivo específico é que para o gerenciamento de requisições é necessário um algoritmo de priorização de requisições e um algoritmo para definição, execução e limitação de requisições.

Assim, este artigo apresenta uma aplicação que gerencia a demanda de requisições a Web Services de aplicações, com o objetivo de garantir maior disponibilidade, suportando o protocolo SOAP e a arquitetura RESTful. A aplicação executa requisições categorizando-as e limitando seguindo regras pré-estabelecidas.

Dessa forma, o artigo está dividido nas seguintes seções. A seção 2 apresenta a fundamentação teórica. A seção 3 descreve a aplicação e aprofunda seu desenvolvimento. A seção 4 demonstra os resultados obtidos com a utilização da aplicação. Por fim, a seção 5 relata as conclusões dos resultados alcançados em relação aos objetivos definidos.

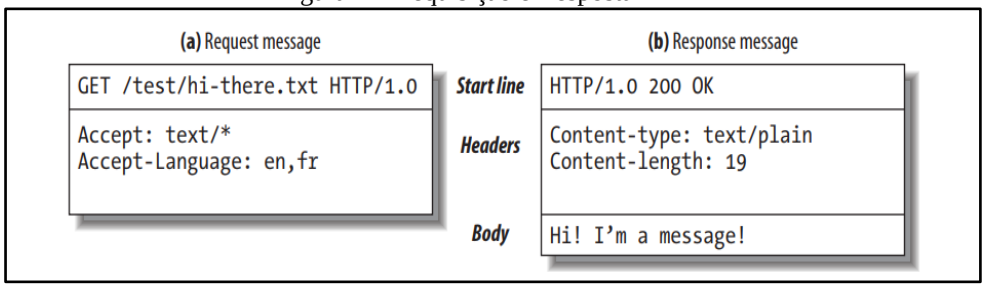
2 FUNDAMENTAÇÃO TEÓRICA

Essa seção apresenta um aprofundamento dos assuntos que fundamentam a aplicação apresentada neste artigo, sendo que a aplicação abordará o processo de gerenciamento de requisições entre um cliente e uma aplicação através de um *gateway* de APIs, utilizando o protocolo SOAP e a arquitetura RESTful e a implementação de um servidor web no qual esses protocolos serão utilizados. Dessa forma, a subseção 2.1 apresenta o conceito de Servidores Web e *gateway* de API. A subseção 2.2 apresenta as formas de comunicação com um Web Service, definindo os protocolos SOAP e a arquitetura RESTful. Por fim, a subseção 2.3 apresenta uma visão geral dos trabalhos correlatos e compara-os brevemente com a aplicação apresentada neste artigo.

2.1 SERVIDOR WEB E SUA RELAÇÃO COM GATEWAY DE API

Para receber requisições Hypertext Transfer Protocol (HTTP) é necessária a implementação de um servidor web, sendo que segundo Gourley et al. (2002) servidores web disponibilizam os dados requisitados pelo cliente HTTP. Como demonstrado na Figura 1 (a), é feita a requisição (*request message*) de um documento por um cliente através do `GET /test/hi-there.txt`. Já na Figura 1 (b) o servidor indica o sucesso ou não da requisição na linha inicial (*Start line*) da resposta e entrega o corpo (*body*) da resposta (*response message*). Dessa forma, servidores web são servidores de conteúdo, podendo entregar conteúdo previamente criado como páginas HTML, imagens e outros conteúdos dinâmicos de uma aplicação geradora de conteúdo (GOURLEY et al., 2002).

Figura 1 – Requisição e Resposta HTTP

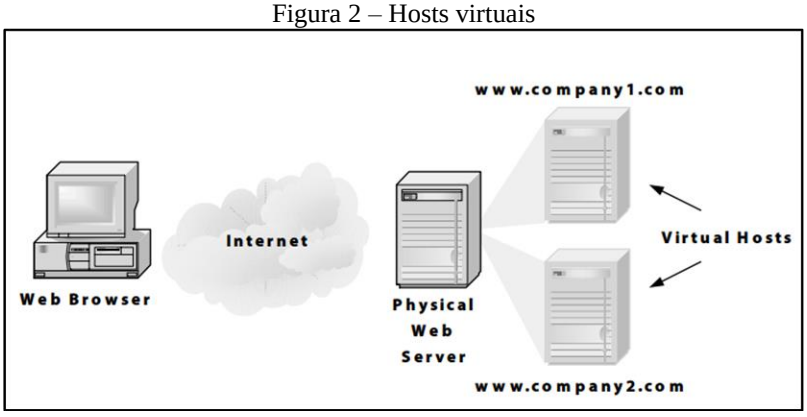


Fonte: Gourley et al. (2002).

Esse conteúdo dinâmico é entregue através de aplicações que geram conteúdo sob demanda. Há uma classe de servidores web os quais são chamados de servidores de aplicação (GOURLEY et al., 2002). Essas aplicações podem ser utilizadas para a implementação de aplicações web desde a entrega de HTML dinâmico até a implementação de Web Services.

Em um servidor web pode-se ter mais de uma aplicação web disponível, conforme representadas na Figura 2 por `www.company1.com` e `www.company2.com` (THOMAS, 2001). Disponibilizado na versão 1.1 do protocolo HTPP foi adicionado o cabeçalho host na requisição para que assim seja possível indicar ao servidor qual aplicação ele deve retornar ao cliente (THOMAS, 2001). Essa funcionalidade torna possível a disponibilidade de websites em somente um servidor, mas também é possível que serviços de hospedagem possam utilizar bancos de servidores replicados (também chamados de fazendas de servidores) para espalhar a carga sobre eles (GOURLEY et al., 2002). Em conjunto a essas ferramentas, também pode-se utilizar um gerenciador de API, que através de seu *gateway* de API pode gerenciar a carga infligida ao servidor.

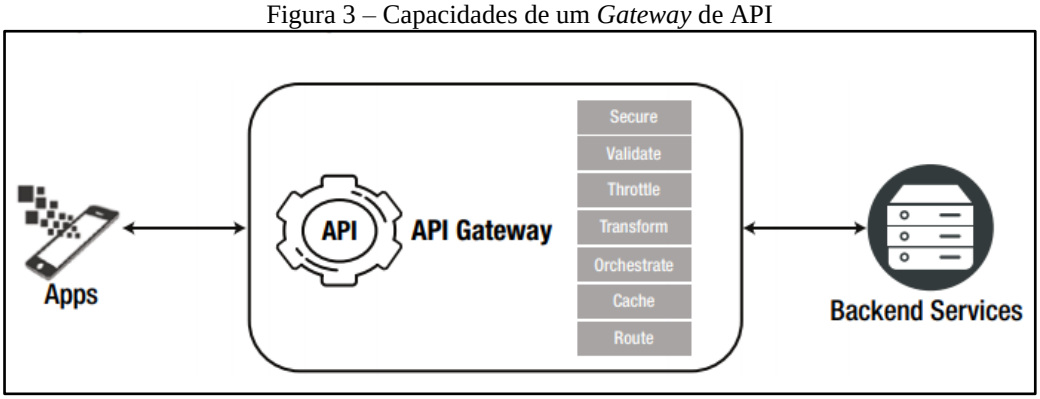
Figura 2 – Hosts virtuais



Fonte: Thomas (2001).

Um *gateway* de API é o núcleo de qualquer solução de gerenciamento de API estabelecendo comunicação entre serviços de *back-end* e aplicações, além de controlar requisições feitas à API (DE, 2017). Para fazer isso, ele disponibiliza uma interface sobre os serviços de *back-end*. Essa interface intercepta requisições para forçar segurança, validar dados, limitá-las e finalmente entregá-las ao serviço destino (DE, 2017). A Figura 3 demonstra um aplicativo se comunicando a um serviço de *back-end* através de um *gateway* de API que realiza a segurança, validação, regulação, transformação, orquestração e realiza cache para fazer o roteamento.

Figura 3 – Capacidades de um *Gateway* de API



Fonte: De (2017).

Para utilizar um gerenciador de API, é necessário entender o escopo do sistema e suas limitações (largura da banda e número de operações concorrentes que podem ser processadas) definidas (SANDOVAL, 2016), devendo levar esses fatores em consideração ao gerenciar requisições de uma aplicação. Segundo De (2017), a aplicação de gerenciamento de APIs deve fornecer uma cota de consumo, prisão de pico (do inglês, *Spike Arrest*), sufocamento de utilização e priorização de tráfego. Essas funcionalidades e suas definições estão relacionadas no Quadro 1.

Quadro 1 – Definição de Funcionalidades de Gerenciamento de Requisições

FUNCIONALIDADE	DEFINIÇÃO
Cota de consumo	Define o número de chamadas de API que uma aplicação pode fazer durante um intervalo de tempo
Prisão de pico	Identifica um crescimento inesperado de tráfego na API podendo interromper as requisições acima do pico permitido
Sufocamento de utilização	Disponibiliza um mecanismo para desacelerar requisições continuas
Priorização de tráfego	Indica ao gerenciador quais requisições devem ter maior prioridade, ou seja, requisições que devem ser processadas com antecedência a outras.

Fonte: De (2017).

Entre as funcionalidades apresentadas no Quadro 1, a priorização de tráfego será implementada na aplicação apresentada neste artigo. Ela pode ser implementada através da classificação de requisições HTTP utilizando a Uniform Resource Locator (URL) requisitada, cabeçalhos, tipo da requisição (`GET`, `PUT`, `HEAD`, `POST`, `DELETE`, `OPTIONS`, `CONNECT` e `TRACE`), servidor e porta (ANALYZER, 2011). Através dessas informações é possível indicar a prioridade (utilizando como base regras definidas na aplicação) e continuar a execução.

2.2 FORMAS DE COMUNICAÇÃO COM UM WEB SERVICE

A comunicação com Web Services é feita através de protocolos e arquiteturas. Através deles são definidos como as informações serão enviadas e como será a estrutura da requisição. Existem diversos protocolos e arquiteturas que podem ser utilizadas para comunicação com Web Services. Contudo, neste artigo serão abordados o protocolo Simple Object Access Protocol (SOAP) e a arquitetura Representational State Transfer (RESTful).

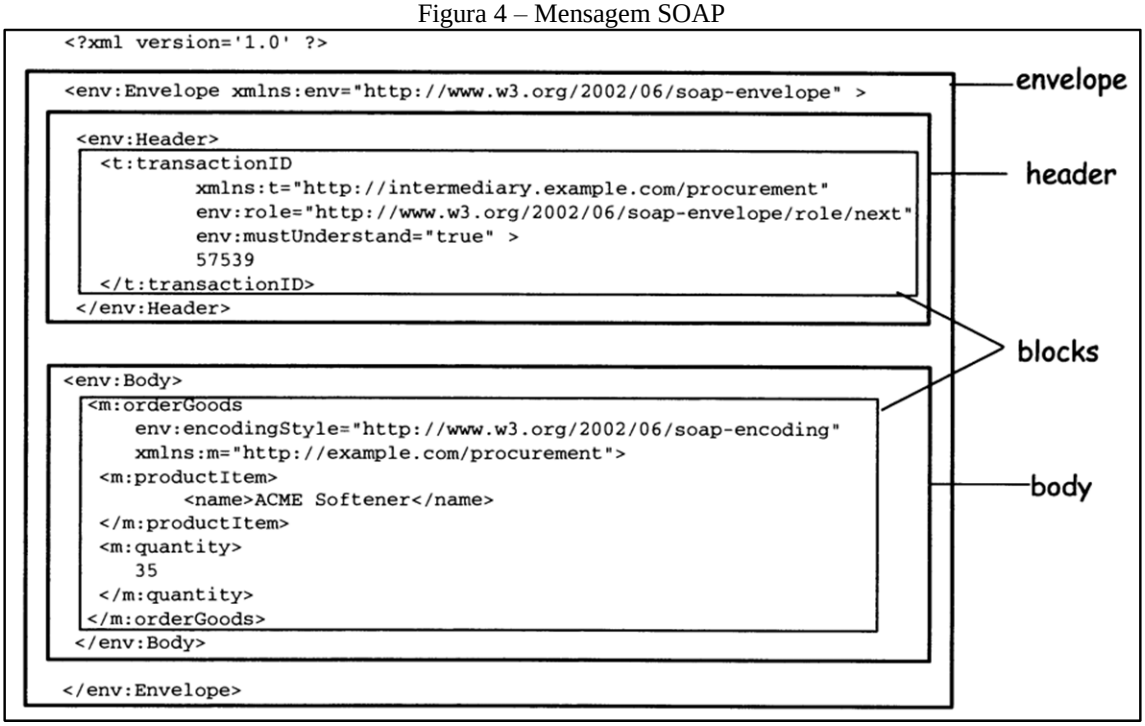
O protocolo SOAP está subjacente a todas as interações com Web Services. Esse protocolo define como organizar informações utilizando um XML de uma maneira estruturada e identificando os tipos dos dados, nomes de métodos e possíveis parâmetros (ALONSO et al., 2004). Iniciado em 1999 pelo World Wide Web Consortium (W3C), o protocolo SOAP em sua primeira versão da especificação era baseado no protocolo HTTP. Com o avanço no desenvolvimento do protocolo isso foi modificado e o SOAP se tornou apenas um transportador genérico de informações. Ao atingir a versão 1.2 foram adicionadas novas especificações semânticas para as estruturas de mensagens (ALONSO et al., 2004).

Como um protocolo de comunicação, SOAP não tem estado, ou seja, cada requisição é tratada como um evento independente não relacionado a uma requisição anterior. Ele também ignora a semântica das mensagens sendo trocadas por meio dele, sendo assim qualquer padrão de comunicação incluindo requisição-resposta deve ser implementado no sistema subjacente ao servidor SOAP (ALONSO et al., 2004). Isto é, o SOAP foi criado deliberadamente para suportar aplicações fracamente acopladas que interagem entre si utilizando mensagens assíncronas (ALONSO et al., 2004).

O protocolo SOAP troca informações utilizando mensagens. Essas mensagens são utilizadas com um envelope no qual a aplicação define qualquer informação que precisa enviar (ALONSO et al., 2004). Na Figura 4 é demonstrado um exemplo de uma mensagem SOAP e é delimitada cada parte dela (envelope, cabeçalho e corpo). Essa mensagem, segundo Alonso et al. (2004), é utilizada pelo protocolo para possibilitar a troca de informações, em que cada mensagem deve ter um envelope e cada envelope tem duas partes: um corpo e uma cabeçalho. A possível estrutura do corpo da mensagem deve estar demonstrada no Web Services Description Language (WSDL).

O WSDL é o núcleo do Web Service, fornecendo um caminho comum no qual são representados os dados que são enviados nas mensagens, as operações que podem ser efetuadas nas mensagens e o mapeamento das mensagens no transporte de rede (NEWCOMER, 2002). Segundo Newcomer (2002), o WSDL é dividido em: definição de tipos de dados; operações abstratas; e, amarrações do serviço.

Figura 4 – Mensagem SOAP



Fonte: Alonso et al. (2004).

O cabeçalho não é obrigatório, mas pode conter informações como segurança e interação transacional. No geral, o cabeçalho deve conter informações que serão processadas por uma aplicação intermediária. Caso não tenha aplicação intermediária, o cabeçalho é desnecessário (ALONSO et al., 2004). Contudo, o corpo da requisição é a verdadeira mensagem sendo enviada, a qual será repassada à aplicação destino (ALONSO et al., 2004).

Comparando o protocolo SOAP com a arquitetura REST, essa tem sua estrutura de requisição orientada ao recurso, ou seja, para qualquer requisição REST com o objetivo de interagir com um recurso é necessário indicar o seu Uniform Resource Identifier (URI). Com essa arquitetura, podem-se realizar requisições `GET`, `POST`, `PUT` e `DELETE`, por exemplo. Fazendo uma requisição `HTTP` do tipo `GET` são retornados os atributos da entidade requisitada. Fazendo uma requisição do tipo `POST` pode ser criada uma entidade passando uma relação de atributo e valor. Por sua vez, com uma requisição `PUT` é possível atualizar uma entidade existente, novamente passando uma relação de atributo e valor. Por fim, fazendo uma requisição `DELETE` uma entidade existente pode ser excluída.

O RESTful pode ser fornecido em duas arquiteturas: a RESTful orientada a recursos e a REST-RPC híbrida (RICHARDSON; RUBY, 2007). A variação RESTful que tem arquitetura orientada a recursos, do inglês Resource Oriented Architecture (ROA), tem sua estrutura de requisição padrão representada no Quadro 2 no qual é demonstrado como é definido o escopo do recurso para que assim ele seja endereçável através do princípio de interface uniforme (RICHARDSON; RUBY, 2007). Nessa arquitetura as informações do método são armazenadas no protocolo HTTP utilizado para efetuar a requisição.

Quadro 2 – Estrutura REST

IDENTIFICADOR	DESCRIÇÃO
Recurso	Alvo conceitual de uma referência hipertexto, como: `cliente/pedido`.
Identificador de recurso	A URL identificando um recurso específico, como: `cliente/:identificador`.
Metadado do recurso	Informação descrevendo o recurso, como: autor, fonte localização.
Representação	O conteúdo do recurso – Mensagem JSON, Documento HTML.
Metadado de Representação	Descrição de como processar a representação como: tipo da media, data de modificação.
Dado de Controle	Descrição de como otimizar o processamento da resposta.

Fonte: adaptado de Patni (2017).

Por sua vez, a arquitetura REST-RPC híbrida pode não seguir esses padrões conforme apresentado no Quadro 3. Isso porque pode não ter uma URI relacionada diretamente ao recurso requisitado e sim a um procedimento interno da aplicação (RICHARDSON; RUBY, 2007).

Quadro 3 - Estrutura REST-RPC

IDENTIFICADOR	DESCRIÇÃO
Procedimento	Alvo conceitual de uma referência hipertexto, como: `criar_cliente/`.
Identificador de recurso	A URL identificando um recurso específico, como: `acessar_cliente/?id=:identificador`.
Representação	O conteúdo do recurso – Mensagem JSON, Documento HTML.

Fonte: adaptado de Richardson e Ruby (2017).

Para garantir a segurança dos serviços web podem ser utilizados padrões de autenticação, sendo que os principais são a: autenticação básica e a autenticação *digest* (RICHARDSON; RUBY, 2007). A autenticação básica faz uso do cabeçalho de resposta `www-authenticate`. Esse cabeçalho indica que para que a requisição seja processada é necessário o envio de uma credencial válida pelo cabeçalho de requisição `authorization`. O valor do cabeçalho `authorization` deve ser gerado através dos valores dos campos de usuário e senha concatenados e codificados em base64 (RICHARDSON; RUBY, 2007).

A autenticação *digest* segue o mesmo padrão da autenticação básica, mas para toda requisição são retornados cabeçalhos que trazem informações sobre a autenticação e um *nonce* (uma chave gerada pela aplicação, que a cada transação é atualizada) e é responsabilidade do cliente tornar esses dados em uma credencial compreensível pela API. Essa chave somente pode ser gerada tendo o conhecimento das credenciais fornecidas inicialmente pela aplicação (RICHARDSON; RUBY, 2007).

2.3 TRABALHOS CORRELATOS

Esta seção descreve três trabalhos correlatos ao trabalho apresentado neste artigo. O Quadro 4 aborda um trabalho que implementa um gerenciador de acessos de Web Services para integração com sistemas de agentes (OVEREINDER, VERKAIK e BRAZIER, 2008). O Quadro 5 descreve o API Manager da Mulesoft (2017b) que fornece um ambiente gerente de Web Services. O Quadro 6 detalha a aplicação Azure API Management da Microsoft (2018a), sendo um *middleware* que gerencia requisições e disponibiliza um ambiente para criação e controle de APIs.

Quadro 4 - Web Service Access Management for Integration with Agent Systems

Referência	Overeinder, Verkaik e Brazier (2008)
Objetivos	Este trabalho propõe a utilização de um *gateway* para ter acesso a um Web Service controlado.
Principais funcionalidades	Suporta gerenciamento de infraestrutura integrado, utilizando Acordos de Nível de Serviço (ANS) para controlar acesso e uso do Web Service por agentes. A negociação da chave de acesso é feita através do *middleware*, ele define os requisitos para permitir acesso aos recursos e gerenciar a negociação, estabelecendo os termos de condições de um contrato especificando regras de acesso e utilização de um recurso. Uma visão geral desse processo é apresentada no Anexo A, Figura 14.
Ferramentas de desenvolvimento	AgentScape, Apache Axis.
Resultados e conclusões	O gateway de Web Services apresentado gerencia o ANS tanto para o consumidor quanto para o agente, contudo, agregadas essas requisições devem aceitar ANS forçado pelo *gateway* do serviço web. A interoperacionalidade do *gateway* com frameworks de gerenciamento de serviços web será aprofundada.

Fonte: elaborado pelo autor.

Com funcionalidades de gerenciamento de Web Service implementadas, a aplicação de Overeinder, Verkaik e Brazier (2008) gerencia a comunicação entre agentes e APIs. Este artigo tem escopo especializado ao contexto de agentes, por isso, não pode ser utilizada por outros Web Services sem devidas adaptações e melhorias.

Quadro 5 - Mulesoft API Manager

Referência	Mulesoft (2017b)
Objetivos	Atuar como um componente do ambiente Anypoint Plataform também desenvolvido pela Mulesoft e operar como um simplificador de desenvolvimento e lançamento de APIs.
Principais funcionalidades	Suporta controle de requisições, configurar limites e regras para Web Services, assim como definir regras de gerenciamento passivo, enviando notificações a usuários caso forem quebradas. Gerenciamento de um ou mais Web Services, suportando o protocolo SOAP e RESTful, possui um estúdio para desenvolvimento e lançamento de APIs.
Ferramentas de desenvolvimento	Informação não foi localizada
Resultados e conclusões	Informação não foi localizada

Fonte: elaborado pelo autor.

A aplicação oferecida pela Mulesoft (2017b) possui funcionalidades chaves no gerenciamento de APIs, mas, é paga e limitada ao ambiente Anypoint Plataform, o qual limita o seu uso a sistemas que estão encapsulados dentro da plataforma desenvolvida pela Mulesoft.

Quadro 6 - Microsoft Azure API Management

Referência	Microsoft (2018a)
Objetivos	Possibilitar a criação de um *gateway* de API e protege-lo de abusos e uso excessivo, garantindo informações sobre utilização e saúde do serviço.
Principais funcionalidades	Gerenciamento de requisições, Web Services, auto escalamento, geração automática de documentação e códigos de exemplo. Fornece diagnósticos de uso e de performance da API.
Ferramentas de desenvolvimento	Informação não foi localizada
Resultados e conclusões	Informação não foi localizada

Fonte: elaborado pelo autor.

O Azure API Management fornece ferramentas de escala empresarial com diversas funcionalidades similares a aplicação da Mulesoft. No entanto, ela tem seu uso limitado a plataforma Azure, o que acaba sendo um fator negativo para a aplicação, pois não é possível utilizar em uma infraestrutura gerenciada por outro provedor. Outra característica importante é que a aplicação tem seu código-fonte fechado, o que impede que especializações sejam implementadas.

3 GERENCIADOR DE WEB SERVICE E ORGANIZADOR DE REQUISIÇÕES

Essa seção apresenta o desenvolvimento do Gerenciador de Web Service e Organizador de Requisições, sendo este o foco do artigo. Os Requisitos Funcionais (RF) da aplicação estão apresentados no Quadro 7 e os Requisitos Não Funcionais (RNF) estão apresentados no Quadro 8.

Quadro 7 - Requisitos Funcionais da Aplicação

Requisitos Funcionais da Aplicação
RF01: possibilitar o registro de Web Services SOAP
RF02: possibilitar o registro de Web Services REST
RF03: possibilitar o registro de requisições
RF04: possibilitar o registro de classificações
RF05: possibilitar o registro de regras
RF06: classificar requisições recebidas pela aplicação (baixa prioridade, alta prioridade)
RF07: receber requisições feitas por um cliente
RF08: repassar requisição para a aplicação destino
RF09: mitigar o envio contínuo e ininterrupto de requisições, levando em consideração regras pré-definidas
RF10: permitir implementação modular possibilitando adição de funcionalidades ou remoção

Fonte: elaborado pelo autor.

Quadro 8 - Requisitos Não-Funcionais da Aplicação

Requisitos Não-Funcionais da Aplicação
RNF01: utilizar a linguagem de programação JavaScript dentro do ambiente Node.js
RNF02: implementar a persistência de dados em um banco não relacional MongoDB
RNF03: utilizar o ambiente de desenvolvimento Visual Studio Code

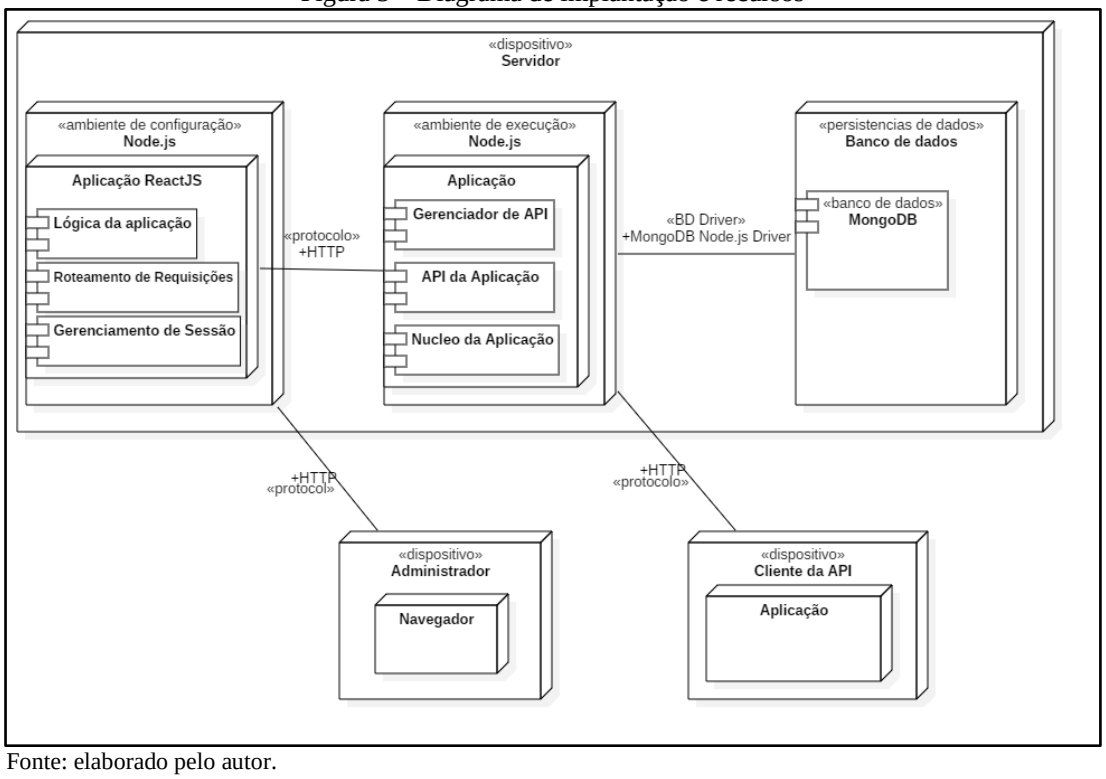
Fonte: elaborado pelo autor.

Em sua implementação base, a aplicação demonstrada nesse artigo denominada de Gerenciador de Web Service e Organizador de Requisições tem sua divisão feita em 6 módulos. O primeiro é o módulo `core` que fornece as funcionalidades básicas da aplicação. No segundo módulo, chamado `web`, é contida a base do tratamento e roteamento de requisições, o qual é estendido pelos módulos seguintes: `api` e `api_manager`, ambos estendem o módulo web para adicionar funcionalidade relativas aos seus escopos. Enquanto os módulos `soap_api` e `rest_api` adicionam o suporte de APIs que utilizam a arquitetura RESTful e o protocolo SOAP.

Para o desenvolvimento da aplicação foi utilizada a máquina virtual JavaScript Node.JS na versão 10.8.0, que tem como base o motor JavaScript V8 do navegador Google Chrome (NODEJS, 2018). Para persistência de dados foi utilizado o Banco de Dados MongoDB versão 3.6.2. O ambiente de desenvolvimento utilizado foi o Visual Studio Code com especializações para programação da linguagem JavaScript. Ainda, demonstrado na Figura 5 está toda comunicação entre os ambientes de configuração, execução e a persistência de dados fornecida pelo driver do MongoDB e como clientes se comunicam com a aplicação através do protocolo HTTP.

Com isso, na Figura 5 encontra-se o diagrama de implantação e recursos da API desenvolvida. O diagrama foi desenhado com base em componentes, de modo que possui cinco componentes macros que são subdivididos em outros componentes. Os cinco componentes macros são o ambiente de configuração (esquerda) que contém uma aplicação ReactJS que se comunica com o ambiente de execução através do protocolo HTTP. O ambiente de execução contém a aplicação de gerenciamento de API, a qual fornece o gerenciador de API, sua própria API e o núcleo da aplicação. A aplicação tem interação direta com o banco de dados (direita), utilizando o MongoDB driver para Node.js para efetuar a comunicação com o banco de dados.

Figura 5 – Diagrama de implantação e recursos



Fonte: elaborado pelo autor.

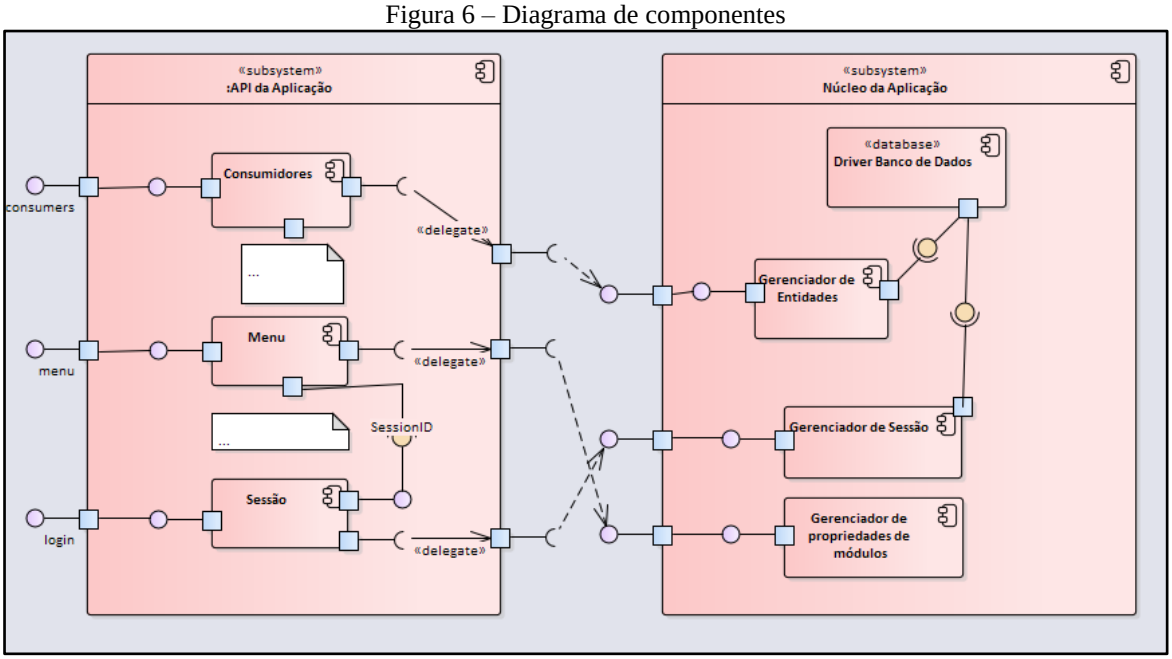
A implementação do núcleo da aplicação contém algoritmos de gerenciamento e inicialização de módulos, banco de dados, execução de tarefas, fábrica de objetos, lançamento de eventos e gerenciamento de *observers*. A aplicação tem como classe principal a classe `App`, que contém todos os métodos e objetos base da aplicação. Ela faz parte da classe `Project`. Essa classe em específico é inicializada no ponto de entrada da aplicação (`index.js`) e após isso é

disponibilizada globalmente através da variável `globals` que, segundo Galizia (2017), são variáveis que podem ser acessadas de qualquer escopo da aplicação.

Como a aplicação foi implementada com o objetivo de fornecer um ambiente modular, a inicialização dela começa buscando por módulos que estão habilitados levando em consideração dependências e os prepara para inicialização. Caso alguma dependência para a inicialização não esteja disponível, o módulo não é inicializado. Uma relação entre os módulos, sua descrição e suas dependências está demonstrada no Apêndice B, Quadro 19. O módulo que gerencia todos esses procedimentos é o módulo `core`, o qual é obrigatório para a execução da aplicação, pois ele é o orquestrador dos procedimentos necessários para a inicialização da aplicação.

Uma visão simplificada dos componentes está demonstrada na Figura 6, no qual é descrita a interação entres os componentes da API no subsistema `:API da Aplicação` (esquerda) e o subsistema `:Núcleo da Aplicação` (direita) que orquestra todas as funcionalidades do sistema. Algumas dessas funcionalidades são entidades que interagem com o banco de dados através do driver de banco de dados, o gerenciador de sessão e o gerenciador de propriedades e configurações de módulos.

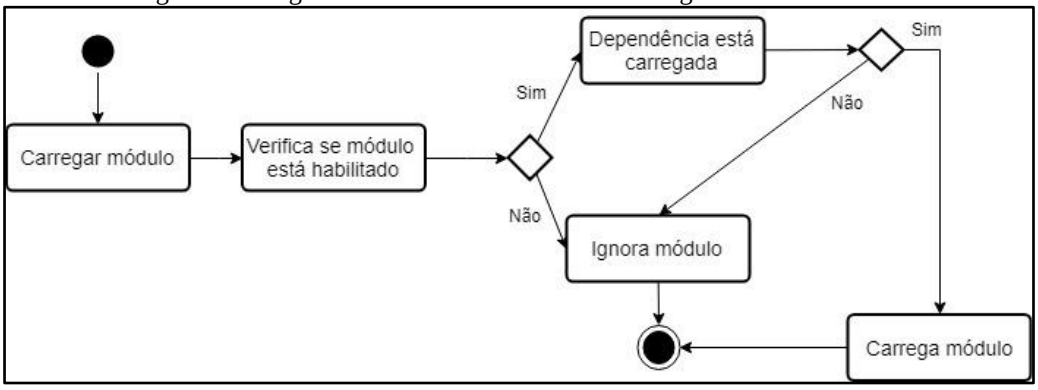
Figura 6 – Diagrama de componentes



Fonte: elaborado por autor.

Para fornecer a funcionalidade de modularização da aplicação foram estipulados padrões de desenvolvimento de módulos para aplicação, sendo que o fluxo de carregamento desses módulos está disponibilizado na Figura 7. Na figura é demonstrado que ao carregar um módulo é verificado se está habilitado e se suas dependências foram carregadas, caso o resultado de ambas verificações seja verdadeiro o módulo é carregado para a memória da aplicação.

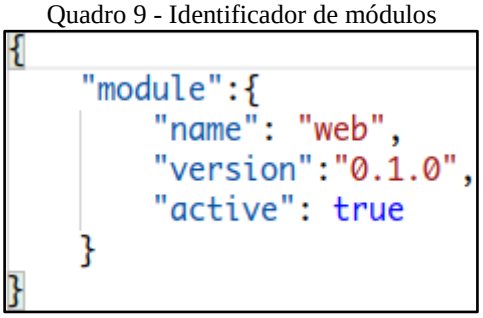
Figura 7 – Diagrama de atividade do fluxo de carregamento de módulos



Fonte: elaborado por autor.

Módulos que serão carregados devem ter seu arquivo de definição contido na pasta `app/etc/modules`, sendo que esse arquivo de definição deve ser um arquivo Java Script Object Notation (JSON) com o seu nome sendo um identificador de módulo único. Um exemplo de seu conteúdo está demonstrado no Quadro 9 sendo necessário o nome do módulo (`name`), sua versão (`version`) e se está ativo ou não (`active`).

Quadro 9 - Identificador de módulos



Fonte: elaborado por autor.

Após o carregamento do arquivo, o módulo deverá estar localizado dentro da pasta `app/code/`. Caso encontrada, é procurado pelo arquivo `config.json` que deverá estar dentro da pasta `etc/`. Um exemplo de seu conteúdo está demonstrado no Apêndice B, Quadro 22. Dentro desse arquivo são indicadas quais funcionalidades o módulo pode utilizar, a descrição das funcionalidades está disponível no Apêndice B, Quadro 21. Qualquer informação contida no arquivo de configuração dos módulos é carregada para memória e permanecerá até a finalização da execução.

Para que módulos possam adicionar funcionalidades à aplicação de forma que não seja necessária nenhuma alteração no código base da aplicação, foi utilizado o padrão de design *observer*. Esse padrão faz uso de eventos, os quais são lançados em momentos específicos da aplicação, sendo que os módulos podem observá-los e executar instruções de seu escopo. O método responsável pelo lançamento de eventos é o `dispatchEvent()`. Esse método recebe como parâmetros a identificação do evento e qualquer dado que seja relevante ao observador.

O módulo de servidor web desenvolvido utiliza a interface HTTP implementada pelo NodeJS para receber requisições e as rotear para o controlador requisitado. Para o roteamento de requisições é utilizada a classe `Router` no qual o caminho da requisição é tratado e o controlador e ação relacionada é carregada e executada. Todos os controladores têm acesso a requisição do cliente e a resposta a ser devolvida.

Tanto a requisição quanto resposta são encapsuladas pelas classes `Request` e `Response`, respectivamente. Essas classes controlam a análise das informações da requisição e a definição de dados na resposta. O controle da requisição é feito através da análise do corpo da requisição, método, caminho para que esses dados sejam tratados por roteadores, controladores ou modelos. Enquanto o controle da resposta é feito através da definição de cabeçalhos e corpo.

Após esse processo de encapsulamento da requisição e resposta o controlador que a tratará é carregado. Qualquer módulo pode ter um controlador, basta que em seu `config.json` seja indicado a existência de um. Assim, o roteador encontrará o controlador e poderá rotear a requisição a ele. Todo controlador deve estender a classe do módulo `core` chamada `Action`. Essa classe implementa métodos especiais de controladores como o `_preDispatch()` e o `_postDispatch()`, de forma que esses métodos são executados antes da ação do controlador e depois da ação do controlador, respectivamente. Esses métodos podem ser sobrescritos ou estendidos para especializações do controlador. A execução das ações é feita através roteador de requisições.

A persistência de dados na aplicação é feita através do adaptador de banco de dados que está implementado no núcleo da aplicação. Adaptadores de banco de dados podem ser implementados para aplicação, mas por padrão é utilizado o adaptador de MongoDB. Esse adaptador coordena requisições ao banco de dados executando instruções de seleção, criação e atualização. Esse adaptador pode ser utilizado por modelos de dados que estendem a classe `Abstract`.

Ao estender a classe `abstract` é necessário indicar a que coleção a entidade pertence através do atributo `_table`. Esse atributo indica aos métodos herdados da classe `Abstract` em qual coleção devem ser aplicadas as chamadas de persistência de dados. Métodos chaves que são herdados da classe `abstract` são `save()` e `load()`. Uma relação dos métodos chave herdados da classe `Abstract` está disponível no Apêndice B, Quadro 18. Entidades também podem carregar coleções de entidades através do método `collection()` no qual é possível aplicar filtros, limites e paginação a coleções de entidades.

Para configuração e gerenciamento de entidades da aplicação foram implementadas APIs REST que disponibilizam dados da aplicação serializados utilizando JSON. Para utilizar a API é necessário utilizar o recurso de `login`, que ao passar um usuário e senha é retornado um `id` de sessão. Esse `id` deve ser enviado em todas as chamadas à API por meio do cabeçalho da requisição. Essa chave é válida pelo período de 2 dias, caso invalidada será necessário um novo *login*.

Ao possuir uma chave de sessão, o acesso a todos os recursos da API é permitido. Estes recursos estão descritos e relacionados com suas funcionalidades no Apêndice B, Quadro 20. Funcionalidades da API podem ser estendidas por módulos através do `config.json` de cada módulo no qual deve ser indicada a entidade da API e os métodos HTTP suportados, sendo que um exemplo dessa extensão é demonstrado no Quadro 10.

Quadro 10 - Demonstração da configuração de API

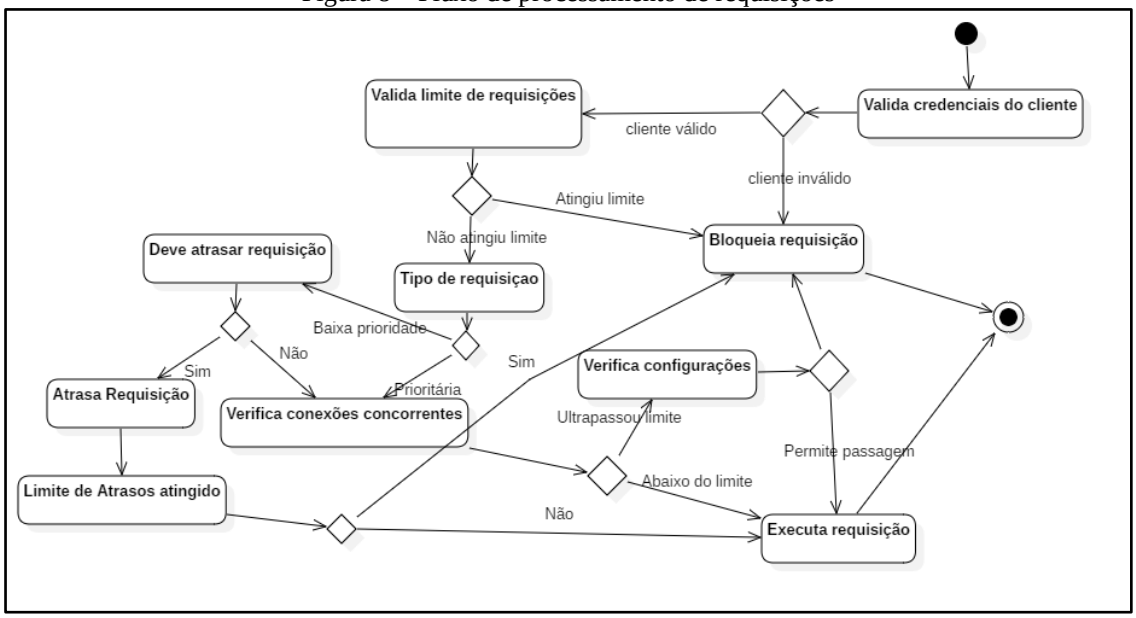
```
"global" : {
    "web" : {···
    },
    "api" : [
        {
            "entity" : "api_consumers",
            "allow" : ["get", "post", "put", "options"]
        },
        {
            "entity" : "api_applications",
            "allow" : ["get", "post", "put", "options"]
        },
        {
            "entity" : "api_statistics_daily",
            "allow" : ["get", "options"]
        }
    ],
```

Fonte: elaborado por autor.

Para o gerenciamento de API foi implementado o módulo `api_manager`, sendo que ele implementa o gerenciamento de API na aplicação. Esse módulo contém toda a base necessária para o gerenciamento de requisições. Para que esse módulo funcione como um servidor de gerenciamento de API, ele estende ao módulo de servidor web. Para o tratamento de requisições foi sobrescrito o roteador de requisições do módulo de servidor web que faz o roteamento das requisições levando em conta a aplicação requisitada. O gerenciador de API tem como base uma relação de entidades de cliente e aplicação na qual aplicações são os pontos de entrada de requisições a serem recebidas, sendo que os clientes são as aplicações e os usuários que fazem requisições.

O diagrama que demonstra o fluxo do processamento de uma requisição ao gerenciador de API está representado na Figura 8 na qual primeiro valida as credenciais do cliente, caso seja um cliente cadastrado na aplicação é validado seu limite de requisições. Após validações de limites é verificada a prioridade da requisição, caso seja uma requisição de baixa prioridade ela pode sofrer atraso pelo próprio gerenciador de API. Caso o limite de atrasos for atingido a requisição pode ser bloqueada. Durante a verificação de atraso também é verificada a quantidade de conexões concorrente na aplicação, caso atinja limites definidos por configurações, a requisição é bloqueada e caso não tenha atingido limites a requisição pode ser executada.

Figura 8 – Fluxo de processamento de requisições



Fonte: elaborado por autor.

Ao receber uma requisição, o servidor de gerenciamento de API identifica nos cabeçalhos da requisição qual o *hostname* requisitado. Através dessa informação é possível carregar a aplicação requisitada. Ao inicializar, essa aplicação é instanciada e a classe responsável pelo tratamento da requisição, a classe `Manager`, verifica qual é o protocolo suportado pela aplicação. Com isso, possibilita carregar o módulo responsável pelo tratamento de requisições utilizando o protocolo. O algoritmo utilizado para o carregar está disponível no Apêndice B, Quadro 24. Após a definição do

protocolo utilizado, a requisição é passada ao tratador de requisições do módulo. Esse tratador de requisições deve estender a classe `Handler` e estendê-la implementando tratamentos específicos ao protocolo ou arquitetura desejada.

A classe `Handler` é responsável por grande parte do gerenciamento de requisições. Ela valida as chaves enviadas por meio dos cabeçalhos da requisição e carrega o cliente que está utilizando a API. No Apêndice B no Quadro 23 e Quadro 25 é demonstrado os algoritmos utilizados pelo *handler* para o gerenciamento de requisições. Uma relação entre o significado do cabeçalho e seu nome se encontra no Apêndice B, Quadro 17. Esses cabeçalhos de resposta indicam informações relevantes ao cliente que está utilizando a API como:

a) quantidade de requisições feitas no dia;
b) limite de requisições diárias da aplicação;
c) limite de requisições diárias do cliente;
d) se a requisição está sendo gerenciada;
e) se a requisição foi atrasada;
f) se a requisição foi bloqueada.

O módulo de gerenciamento de API por si só, não suporta tipos de requisição e protocolo. Para adicionar suporte a requisições e protocolos são necessários módulos de requisições. Para que um módulo adicione suporte a um tipo de requisição é necessário ter em seu `config.json` os parâmetros `api_manager`, dentro do nó `global` e definir qual será o tipo e que classe irá gerenciar a requisição como demonstrado no Quadro 11. A implementação base da aplicação incluí módulos que adicionam suporte aos protocolos REST e SOAP.

Quadro 11 - Adição de protocolos e arquiteturas ao gerenciador de API

```
{
    "global" : {
        "api_manager" : [
            {
                "type" : "soap",
                "handler" : "request"
            }
        ]
    }
}
```

Fonte: elaborado por autor.

Para a implementação do módulo SOAP do gerenciador de API a requisição é recebida e analisada, caso seja uma requisição do WSDL ou uma requisição à API. Para isso é identificado o método da requisição recebida e os parâmetros enviados. Ao ser identificada uma requisição utilizando o método `POST` são utilizadas bibliotecas de análise de XML para identificar o corpo do XML enviado para descobrir qual método da API é chamado e esse método é repassado ao módulo de gerenciamento de API que define se a requisição deve ser executada ou atrasada.

O processamento do WSDL é feito através da interpretação do WSDL original proveniente do servidor final da aplicação. Esse WSDL tem seu endereço SOAP trocado pelo endereço configurado no gerenciador de API, para que assim o cliente da API envie suas requisições corretamente. Ao ter esse endereço trocado, todas as requisições feitas para a aplicação final transacionam através do gerenciador de API.

Para a implementação do módulo que adiciona suporte a arquitetura RESTful, foram implementados algoritmos de identificação de requisições através da URI e o método requisitado. Após a identificação do método é necessário repassar a requisição ao servidor destino. Para fazer isso é necessário manter qualquer cabeçalho da requisição feita originalmente e qualquer cabeçalho da resposta fornecida pelo servidor final, pois esses cabeçalhos podem ser necessários para a execução da requisição no servidor destino.

4 RESULTADOS

Em relação aos trabalhos correlatos da aplicação apresentada por esse artigo, no Quadro 12 tem-se uma comparação de características dessas aplicações, na qual é demonstrado que todas as aplicações apresentadas oferecem um controle de requisições. Entretanto apenas a aplicação apresentada nesse artigo e as aplicações comerciais da Microsoft (2018a) e da Mulesoft (2017b) oferecem gerenciamento de um ou mais Web Services. Também é demonstrado que todas as aplicações comerciais não são gratuitas, enquanto a aplicação de Heiden (2018), apresentada nesse artigo, e o de Overeinder, Verkaik e Brazier (2018) são código livre.

Quadro 12 – Comparação de correlatos

Correlatos / Características	Overeinder, Verkaik e Brazier (2008)	Mulesoft (2017b)	Microsoft (2018b)	Heiden (2018)
Controle de requisições	Sim	Sim	Sim	Sim
Gerenciamento de mais de um Web Service	Não	Sim	Sim	Sim
Gratuito	Sim	Não	Não (Teste por 30 dias oferecido)	Sim
Priorização de recursos	Não	Não	Não	Sim
Requer implementação	Sim	Não	Não	Não
Protocolos e arquiteturas suportadas	SOAP	SOAP, RESTful	SOAP, RESTful	SOAP, RESTful
Implementação modular	Não	Sim	Não	Sim

Fonte: elaborado pelo autor.

Outra característica comparada no Quadro 12 é se é requerido uma implementação, ou seja, se é necessário que clientes que utilizam a API necessitam alguma implementação para se adaptar ao uso do gerenciador de API. A única aplicação que tem essa necessidade é a desenvolvida por Overeinder, Verkaik e Brazier (2008). Ela também tem uma limitação em arquiteturas e protocolos suportados, pois, somente suporta o protocolo SOAP. Contudo, entre todas as aplicações as únicas que tem uma implementação modular são a Mulesoft (2017a) e a aplicação apresentada nesse artigo.

Para validar a aplicação foram executados testes com Web Services SOAP e RESTful. Para efetuar essa análise foram cadastradas cerca de 9 aplicações de Web Service ao gerenciador de API. Com essa base de aplicações foram aplicados testes de cunho comparativo (sem e com o gerenciador de API), sobre o tempo de resposta da requisição e carga no servidor da aplicação final. Também foram feitos testes da implementação, como consumo de memória e uso do processador do servidor no qual o gerenciador de API se encontra.

Com a intenção de validar o controle de demanda do gerenciador de API foram feitos testes de demanda simultâneas, verificando como a carga afetou o servidor destino, comparando-a com requisições sem o gerenciador de API. Para testar as funcionalidades da API do gerenciador de API de forma prática, foi implementada uma aplicação web que faz uso da API fornecida pelo gerenciador de API.

O ambiente para execução dos testes foi um servidor externo gerenciado por terceiros hospedado em nuvem com suas especificações apresentadas no Quadro 13. Neste servidor foi hospedada a aplicação gerenciadora de APIs, a aplicação web que é alimentada pela API do gerenciador de API e o banco de dados MongoDB utilizado pelo gerenciador de API. Todos os testes aplicados nesse servidor serão descritos e aprofundados a seguir.

Quadro 13 – Especificações do servidor

Núcleos de Processadores	4 núcleos
Processador	Intel Xeon E3-12xx
Frequência	2.5Ghz
Memória RAM	3.70GB
Distribuição	CentOS 7
Versão do Kernel	3.10.0

Fonte: elaborado pelo autor.

O benchmark comparativo do tempo de execução da requisição com o gerenciador de API foi feito através da execução de requisições iguais ao *endpoint* da aplicação destino e ao gerenciador de API. Um exemplo reduzido de uma amostra de 50 requisições desse benchmark está apresentado na Tabela 1. Nela é demonstrada uma comparação entre a execução de um login em uma API SOAP com e sem o gerenciador de API em milissegundos e a direita é demonstrada a diferença entre os valores. Com essa comparação é possível ver como o gerenciador de API afeta o tempo final da requisição com uma mediana de 123 milissegundos, ou seja, ao fazer uma requisição através do gerenciador tem-se um aumento no tempo de requisição de cerca de 123 milissegundos.

Tabela 1 – Benchmark comparativo API SOAP

Login SOAP	Login SOAP – Sem Gerenciador de API	Diferença
1332	837	495
1309	810	499
966	825	141
940	1116	-176
1071	805	266
990	845	145
917	813	104
956	833	123
956	883	73
912	824	88
1155	804	351
817	954	-137
1156	927	229
937	945	-8
850	802	48
	Média	**161.2857143**
	Mediana	**123**
	Desvio	**269.3093791**

Fonte: elaborado pelo autor.

O mesmo teste comparativo de tempo de requisição também foi aplicado sobre a arquitetura REST e está demonstrado na Tabela 2. Nela, a mediana da comparação do tempo de requisição entre o uso do gerenciador de API e sem ele ficou de 124,5 milissegundos, ou seja, pode-se esperar um acréscimo de 124,5 milissegundos em cada requisição feita através do gerenciador de API. Também a partir desses dados pode-se concluir que tanto a implementação do módulo de API RESTful e o SOAP, tem um acréscimo similar no tempo de requisição.

Tabela 2 – Benchmark comparativo API REST

Product	Product– Sem Gerenciador de API	Diferença
1044	1037	7
1100	1535	-435
1160	1004	156
1168	1004	164
1129	868	261
1038	881	157
1012	1178	-166
1181	892	289
990	885	105
1041	1507	-466
1199	924	275
1242	1057	185
1226	1027	199
958	1013	-55
1056	1736	-680
	Média	**17.34**
	Mediana	**124.5**
	Desvio Padrão	**414.4709246**

Fonte: elaborado pelo autor.

Os resultados dos testes apresentados anteriormente eram esperados durante a implementação da aplicação, pois o processo de gerenciamento de API funciona como um intermediador da comunicação do cliente com a verdadeira API. Tendo que carregar informações do cliente e validar a requisição, todo esse processo demanda acesso a banco de dados e processamento do servidor do gerenciador de API.

Para os testes de carga do servidor foram feitos testes de processamento de requisições simultâneas, escalando de 5, 10, 15, 30 requisições simultâneas. Ao aplicar esse teste também são capturadas amostras do tempo de resposta das requisições com o objetivo de verificar quanto a carga no servidor do gerenciador de API pode incrementar com a demanda. Os resultados desses testes estão presentes no Quadro 14, em que é apresentada uma relação entre a quantidade de requisições simultâneas, a carga média do servidor e o uso de memória. No quadro é possível ver um crescimento irregular na carga média e um baixo aumento no nível do uso de memória. Contudo também deve-se verificar a carga do servidor no qual a aplicação destino se encontra.

Quadro 14 - Benchmark requisições simultâneas

Requisições simultâneas	Carga média	Uso de memória
5 Clientes	0.2	630M
10 Clientes	0.4	635M
15 Clientes	0.59	648M
30 Clientes	0.71	655M

Fonte: elaborado pelo autor.

Testes de carga de comparativos foram efetuados no servidor no qual a aplicação destino se encontra, para assim verificar se existe um decremento na carga de processamento do servidor ao efetuar requisições através do gerenciador de API. Foram efetuados dois grupos de teste como demonstrado no Quadro 15: um com configurações menos agressivas (direita) no gerenciador de API, ou seja, bloqueando e atrasando menos requisições e com limites maiores de requisições simultâneas; e outro com configurações mais agressivas (centro), bloqueando e atrasando por mais tempo requisições e limites menores de requisições simultâneas.

Quadro 15 – Configurações utilizadas no gerenciador de API

Configuração	**Agressivo**	**Conservador**
Limite de requisições para Delay	20	30
Limite de Requisições Concorrentes	25	30
Limite de Delay de Requisições	5	30
Permitir Passagem de Requisições	Não	Sim
Delay de Requisições (em milissegundos)	10ms	5ms

Fonte: elaborado pelo autor.

Dados coletados desses testes com as configurações normais e agressivas demonstradas no Quadro 15 estão apresentados no Quadro 16. No Quadro 16 é demonstrado a comparação do processamento de requisições simultâneas entre execuções com o gerenciador de API agressivo (coluna 4), com o gerenciador de API mais conservador (coluna 3) e sem a utilização de gerenciador de API (coluna 2).

Nos resultados apresentados no Quadro 16 pode-se notar uma diferença na carga do servidor quando as requisições foram executadas através do gerenciador de API, tanto em configurações menos agressivas, quanto conservadoras. Também pode-se notar uma inconsistência entre alguns resultados durante a execução de 15 clientes simultâneos utilizando o gerenciador de API de forma agressiva. Essa inconsistência deve-se a execução desses testes em servidores reais de produção que podem sofrer interferência de uso real. Contudo, os testes executados demonstram uma diferença entre a execução de requisições através do gerenciador de API, sendo que através dessa diferença pode-se concluir que o gerenciador de API pode auxiliar na redução da carga de um servidor de API.

Quadro 16 – Comparativo de carga do servidor

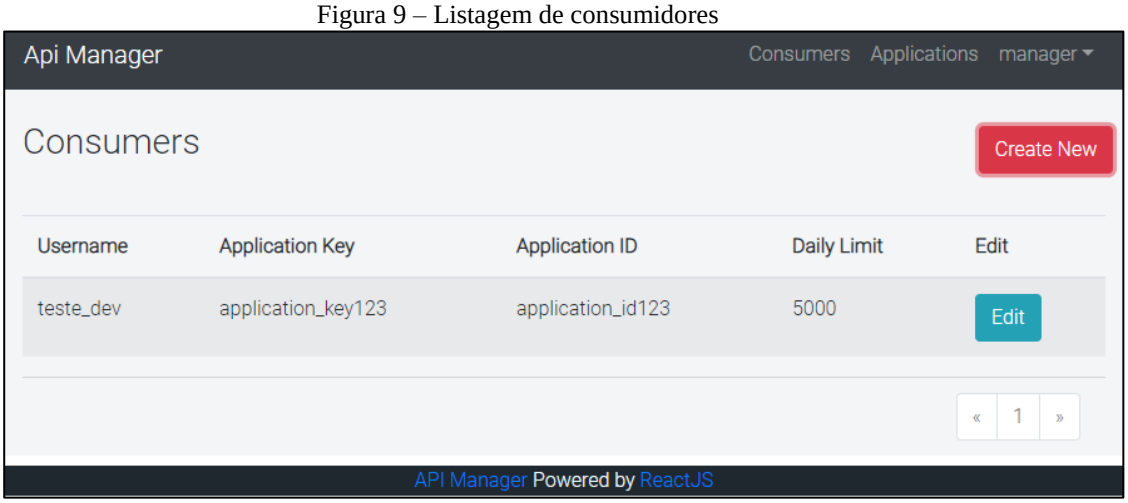
Requisições simultâneas	Sem gerenciador de API	Com gerenciador de API (conservador)	Com gerenciador de API (agressivo)
5 clientes	20%	18%	16%
10 clientes	35.65%	33.57%	29.55%
15 clientes	49.90%	41.43%	48.4%
30 clientes	74.40%	71.65%	37.2%

Fonte: elaborado pelo autor.

Para comprovar a utilidade e funcionalidade da API disponibilizada pelo gerenciador de API foi implementada uma aplicação web utilizando o *framework* ReactJS. Essa aplicação fornece um ambiente administrativo para o projeto

possibilitando através dela uma interface que realiza todas as chamadas de API de forma transparente ao usuário. Na Figura 9 é demonstrada uma tela da aplicação, na qual é possível ver a interface de listagem de consumidores que podem utilizar as APIs cadastradas no gerenciador de API.

Figura 9 – Listagem de consumidores



Fonte: elaborado pelo autor.

Outras demonstrações das interfaces de usuário implementadas pelo autor desse artigo utilizando a API disponibilizada pela aplicação se encontram no Apêndice C. Nesse apêndice é apresentada a interface de edição de entidades na Figura 11. Também é demonstrada a interface de definição de configurações do gerenciador de API na Figura 12. Ainda são demonstrados os campos de personalização da aplicação na Figura 13.

Com base nos resultados obtidos nos testes de performance comparativos, pode-se observar que o uso do API manager na comunicação com um Web Service pode diminuir o uso de processamento no servidor do Web Service. Contudo, para obter um resultado mais expressivo é necessária uma configuração mais agressiva no gerenciador de API. Ao efetuar o teste comparativo entre o tempo de requisição com e sem o gerenciador de API foi descoberto um acréscimo no tempo de execução da requisição. O acréscimo era esperado devido a procedimentos internos do gerenciador de API. Conforme resultados dos testes de carga ao servidor do gerenciador de API, pode-se concluir que para execução da aplicação, poder de processamento tem maior importância que disponibilidade de memória.

5 CONCLUSÕES

Esse artigo apresentou o desenvolvimento de uma aplicação gerenciadora de API, a qual recebe requisições a Web Services configurados nela, para assim validar as requisições (se estão obedecendo limites definidos dentro das configurações do gerenciador de API), assim podendo bloquear requisições ou enviá-las. Para desenvolvimento da aplicação foi utilizado o ambiente de desenvolvimento Visual Studio Code e a linguagem JavaScript em conjunto com o interpretador Node.JS.

A aplicação teve como objetivos específicos fornecer um ponto de entrada para receber requisições, o qual foi implementado utilizando o servidor web fornecido pelo interpretador Node.JS. Para a implementação das interfaces foi implementada uma API da aplicação utilizando a arquitetura RESTful, que fornece a criação, visualização, edição e remoção de entidades e gerenciamento de configurações da aplicação. Com o objetivo de confirmar sua funcionalidade, foi implementada uma aplicação web que faz uso dessa API.

Para o gerenciador de API outro objetivo era a implementação do algoritmo de priorização de requisições, o qual foi implementado no módulo de gerenciamento de API. Nele foi implementado um *delay* na execução de requisições classificadas pelo administrador do gerenciador de API como não prioritárias. Sendo que esse *delay* somente é aplicado durante uma faixa de tempo definida na aplicação. Outro algoritmo implementado foi o de definição, execução e limitação de requisições. A definição e execução de requisições são feitas através dos módulos especializados a arquiteturas e protocolos de API, enquanto a limitação é feita através do módulo gerenciador de API que identifica o cliente que fez a requisição e API requisitada e limita-a.

Como objetivo principal a implementação de um gerenciador de API que recebe e gerencia requisições HTTP, utilizando o protocolo SOAP ou a arquitetura RESTful, pode-se afirmar que o mesmo foi alcançado. A aplicação desenvolvida disponibiliza um gerenciador de API com suporte ao protocolo SOAP e a arquitetura RESTful, como também outras funcionalidades demonstradas.

A partir dos resultados obtidos é possível perceber que existe uma correlação entre o consumo de processamento de um servidor e o uso do gerenciador de API, conseguindo diminuir de forma significativa o uso de processamento do

servidor no qual a API se encontra. Também se tem como fatores relevantes o baixo *overhead* no tempo de requisição ao utilizar o gerenciador de API e baixos requisitos técnicos para sua execução, mas, pode-se notar o aumento no uso de processamento conforme o aumento de demanda.

Em comparação com seus correlatos, a aplicação disponibiliza funcionalidades que são fornecidas por aplicações comerciais de destaque no mercado, oferecendo estrutura modular e expansível e gerenciamento das arquiteturas e protocolos fornecidos por outras aplicações comerciais pagas. Pode-se tornar uma solução de baixo custo para gerenciamento de APIs para empresas ou pessoas que não podem arcar com os custos de utilizar uma das ferramentas comerciais correlatas. Sua implementação base feita em NodeJS pode ser utilizada para a implementação de outras aplicações com outros escopos, podendo acrescentar funcionalidades à aplicação.

Ressaltam-se como limitações do trabalho a impossibilidade de associar um consumidor exclusivamente às aplicações específicas, além de que as aplicações têm seu acesso limitado ao domínio associado. Caso uma aplicação tenha mais de um Web Service, torna-se necessária a utilização de outro subdomínio para o outro Web Service. Outra limitação é a falta da configuração de processamento para regras individuais de *passthrough* ou bloqueio.

Enfim, esse trabalho apresentou um gerenciador de API que fornece funcionalidades que podem auxiliar muitas aplicações com Web Services limitados, seja, por banda ou processamento. Ainda, fornece uma base para projetos futuros com ambiente modular, expansível, API própria para controle da aplicação e estrutura gerenciadora de requisições. Contudo, tem muita possibilidade extensora, como:

a) aumento dos protocolos e arquiteturas suportadas pelo gerenciador de API;
b) melhoria na implementação do roteamento de requisições adicionando suporte a não somente *hosts* e também caminhos;
c) otimização na utilização de recursos do gerenciador de API para suportar servidores com menos recursos;
d) implementar utilização de cache de respostas.

REFERÊNCIAS

ALONSO, Gustavo, et al. **Web Services:** Concepts, Architectures and Applications. 1. ed. Palo Alto: Editora Springer, 2004.

ANALYZER, NetFlow. **NBAR and HTTP Traffic Classification.** ManageEngine Blog. não paginado, ago. 2011. Disponível em: <https://blogs.manageengine.com/network/ netflowanalyzer/2011/08/16/nbar-and-http-traffic-classification.html>. Acesso em: 04 abr. 2018.

DE, Brajesh. **API Management:** An Architect's Guide to Developing and Managing APIs for your Organization. 1. ed. Bangalore: Apress, 2017.

FACEBOOK. **Facebook for Developers.** Facebook. não paginado, abr. 2018. Disponível em: <https://developers.facebook.com/>. Acesso em: 05 abr. 2018.

GALIZIA, Sam. **Using global variables in Node.js.** Stack abuse. não paginado, jun. 2017. Disponível em: <https://stackabuse.com/using-global-variables-in-node-js/>. Acesso em: 12 out. 2018.

GOOGLE. **Google Developers.** Google. não paginado, abr. 2018. Disponível em: <https://developers.google.com/products/>. Acesso em: 05 abr. 2018.

GOURLEY, David, et al. **HTTP:** The Definitive Guide. 1. ed. Sebastopol: O'Reilly Media, 2002.

MICROSOFT. **What is azure**. Azure. não paginado, mar. 2018a. Disponível em: <https://azure. microsoft.com/en-us/overview/what-is-azure/>. Acesso em: 28 mar. 2018.

MICROSOFT. **API Management Documentation.** Azure. não paginado, mar. 2018b. Disponível em: <https://docs.microsoft.com/en-us/azure/api-management>. Acesso em: 28 mar. 2018.

MULESOFT. **API Manager 2.x.** MuleSoft. não paginado, mar. 2017a. Disponível em: <https://docs.mulesoft.com/api-manager/v/2.x/>. Acesso em: 28 mar. 2018.

MULESOFT. **API Manager 2**. MuleSoft. não paginado, mar. 2017b. Disponível em: <https://www.mulesoft.com/platform/api/manager>. Acesso em: 28 mar. 2018.

NASCIMENTO, Lenilson. **O crescimento da Web e as Tendências de Mercado.** Tableless. não paginado, jun. 2014. Disponível em: <https://tableless.com.br/o-crescimento-da-web-e-as-tendencias-de-mercado>. Acesso em: 28 mar. 2018.

NEWCOMER. Eric. **Understanding Web Services – XML, SOAP, WSDL, SOAP and UDDI**. 1 ed. Boston: Addison-Wesley Professional, 2002.

NODEJS, **Node.js.** Node.js. não paginado, dez. 2018. Disponível em: <https://nodejs.org/en/>. Acesso em 12 nov. 2018.

OVEREINDER, B.J; VERKAIK, P.D; BRAZIER, F.M.T. **Web Service Access Management for Integration with Agent Systems.** 2008. 54 f. Artigo – Departamento de Ciência da Computação, VU University Amsterdam de Boelelaan, Amsterdam.

PATNI, Sanjay. **Pro RESTful APIs:** Design, build and Integrate with REST, JSON, XML and JAX-RS. 1. ed. Santa Clara: Apress, 2017.

RICHARDSON, Leonard; RUBY, Sam. **RESTful Web Services.** 1. ed. Sebastopol: O'Reilly Media, 2007.

SANDOVAL, Kristopher. **Stemming the Flood – How to Rate Limit an API.** Nordic APIs não paginado, fev. 2016. Disponível em: <https://nordicapis.com/stemming-the-flood-how-to-rate-limit-an-api/>. Acesso em: 28 mar. 2018.

SANTOS, Wendel. **Research Shows Interest in Providing APIs Still High.** ProgrammableWeb não paginado, fev. 2018. Disponível em: <https://www.programma bleweb.com/news/research-shows-interest-providing-apis-still-high/research/2018/02/23>. Acesso em: 28 mar. 2018.

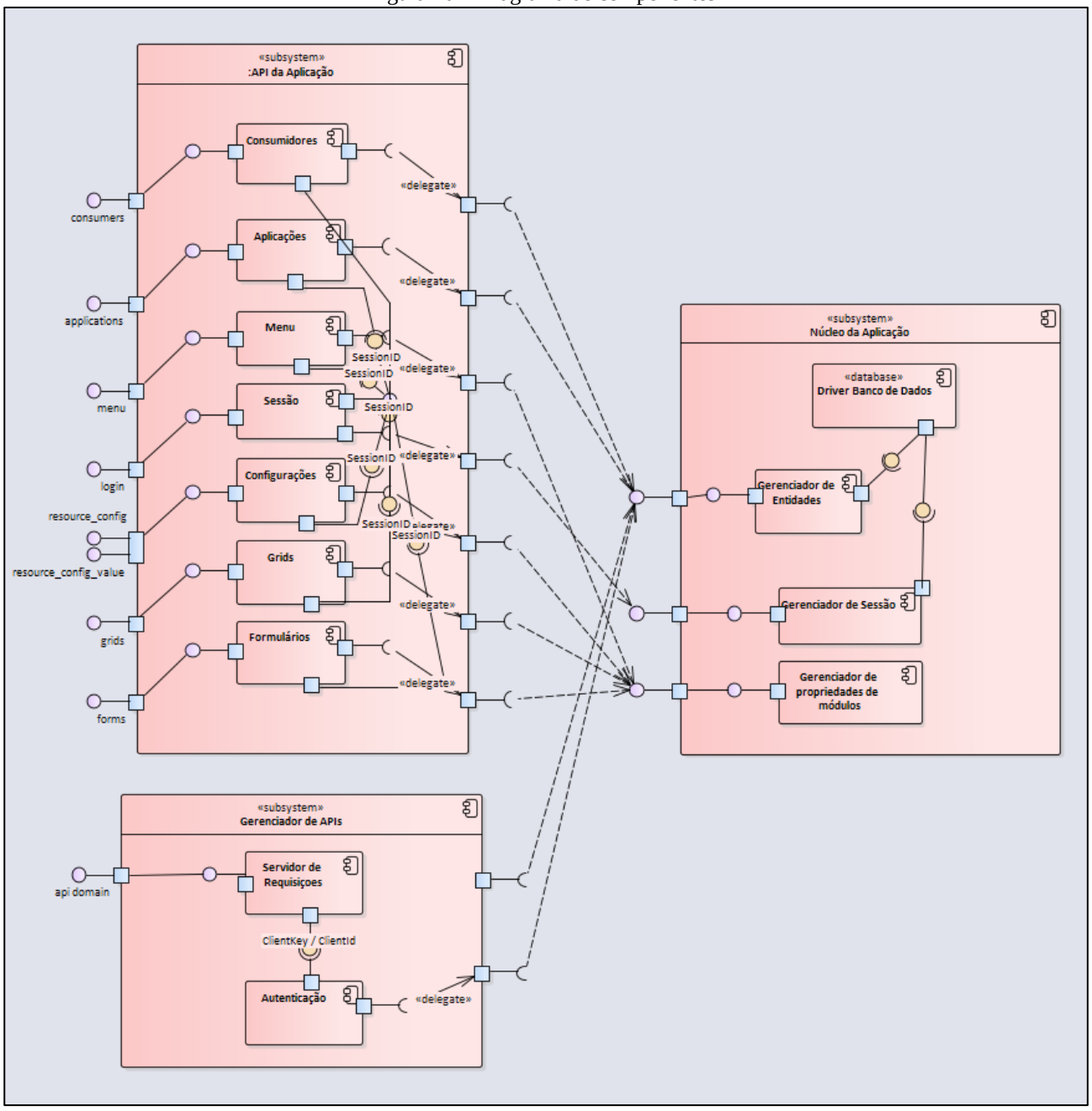
TARJAN, Paul. **Scaling your API with rate limiters.** Stripe. não paginado, mar. 2017. Disponível em: <https://stripe.com/blog/rate-limiters>. Acesso em: 28 mar. 2018.

THOMAS, Stephen. **HTTP Essentials:** Protocols for Secure, Scalable Web Sites. 1. ed. New York: John Wiley & Sons, Inc., 2001.

APÊNDICE A – DIAGRAMAS DE ESPECIFICAÇÃO

Na Figura 10 é apresentado o diagrama de componentes completo da aplicação apresentada nesse artigo.

Figura 10 – Diagrama de componentes



Fonte: elaborado pelo autor

APÊNDICE B – DETALHES DA ESPECIFICAÇÃO E IMPLEMENTAÇÃO DA APLICAÇÃO

No Quadro 17 são apresentados os cabeçalhos (à esquerda), sua descrição (centro) e o tipo do valor retornado (à direita) que são adicionados a resposta de uma requisição utilizando o gerenciador de API.

Quadro 17 – Descrição de cabeçalhos da resposta

CABEÇALHO	DESCRIÇÃO	TIPO
X-Api-Manager-Delayed	Indica se a requisição foi atrasada	boolean
X-Api-Manager-Prevented	Indica se a requisição foi bloqueada	boolean
X-Api-Manager-Managing	Indica que o gerenciador de API está trabalhando	boolean
X-Api-Manager-Request-Limit	Demonstra o número total de requisições que podem ser feitas	int
X-Api-Manager-Requests-Executed	Demonstra o número total de requisições que foram feitas	int

Fonte: elaborado por autor.

A classe abstrata de modelos tem como base os métodos descritos no Quadro 18, nele é apresentado o nome do método (à esquerda), se sua execução é assíncrona (centro) e sua descrição (à direita).

Quadro 18 – Métodos da classe Abstract

MÉTODO	ASSÍNCRONA	DESCRIÇÃO
`load`	Sim	Carrega a entidade requisitada através de seus atributos
`save`	Sim	Salva a entidade no banco de dados através de seus atributos
`loadByCondition`	Sim	Carrega a entidade através de uma condição que deve ser passada como parâmetro
`toJson`	Não	Converte valores da entidade para JSON
`collection`	Sim	Traz a coleção da entidade a partir filtros definidos por parâmetros
`isLoaded`	Não	Indica se a entidade está carregada

Fonte: elaborado por autor.

Durante o carregamento de módulos é feita uma verificação de dependências do módulo que está sendo carregado. Os módulos base da aplicação tem dependências para sua execução. Cada módulo (à esquerda), sua descrição (no centro) e suas dependências (à direita) estão relacionadas no Quadro 19.

Quadro 19 - Descrição e dependências de módulos

MÓDULO	DESCRIÇÃO	DEPENDÊNCIAS
`api_manager`	Adiciona suporte ao gerenciamento de APIs contem a base necessária para gerenciamento de API	`core, web`
`api`	Implementa uma interface de acesso a aplicação para possibilitar configuração e implementação de funcionalidade	`core, web`
`core`	Núcleo da aplicação	
`rest_api`	Adiciona suporte a arquitetura RESTful ao gerenciador de APIs	`api_manager`
`soap_api`	Adiciona suporte ao protocolo SOAP ao gerenciador de APIs	`api_manager`
`web`	Implementa o suporte ao protocolo HTTP para suportar requisições web	`core`

Fonte: elaborado por autor.

Para implementação de uma interface de comunicação com a aplicação apresentada nesse artigo foi implementada uma API. As rotas disponibilizadas por essa API (à esquerda), os métodos suportados (no centro) e uma descrição resumida da funcionalidade (à direita) estão demonstradas no Quadro 20.

Quadro 20 – Descrição de métodos da API da aplicação

URI	MÉTODOS SUPORTADOS	DESCRIÇÃO
/api_consumers	`GET, POST, PUT, OPTIONS`	Interface para a entidade de consumidores, através dela é possível editar consumidores, criar e visualizar.
/api_applications	`GET, POST, PUT, OPTIONS`	Interface para a entidade de aplicações, através dela é possível editar aplicações, criar e visualizar.
/api_statistics_daily	`GET, OPTIONS`	Interface parar estatísticas do gerenciador de API
/resource_config	`GET, OPTIONS`	Interface para trazer campos de configurações
/resource_config_value	`GET, PUT, POST, OPTIONS`	Interface para a entidade de configuração, através dela é possível criar configurações, editar e excluir.
/menu	`GET, OPTIONS`	Traz os menus da aplicação

Fonte: elaborado por autor.

Durante o carregamento de um módulo o seu arquivo de configuração é carregado nesse arquivo e podem-se ter diversos atributos que definem funcionalidades do módulo que está sendo carregado. No Quadro 21 é feita uma relação entre os atributos (coluna 1), seu escopo (coluna 2), uma descrição da funcionalidade (coluna 3) e qual módulo faz a adição dessa funcionalidade à aplicação.

Quadro 21 - Descrição de funcionalidades disponibilizadas por atributos

ATRIBUTO	ESCOPO	DESCRIÇÃO	DISPONIBILIZADO POR MÓDULO
settings	backend	Valores colocados aqui serão disponibilizados como campos de configuração	api
groups	backed	Valores disponibilizados aqui são associados com configurações para organizar elas por grupos	api
pages	backed	Define páginas que serão disponibilizadas, suporta grids e formulários	api
menu	backend	Contém item que podem estar no menu da aplicação	api
web	global	Indica rotas que serão disponibilizadas pelo o módulo de requisições web	web
api	global	Contém recursos que são disponibilizados na API, sua entidade e métodos suportados	api
async_tasks	global	Indica quais tarefas devem ser executadas após a inicialização da aplicação	core
observer	global	Indica o evento observado qual classe e qual método deve ser chamado quando o evento ocorrer	core

Fonte: elaborado por autor.

No Quadro 22 é apresentado um exemplo de um arquivo de configuração de um módulo, no qual é demonstrado como são descritas as informações na conotação JSON.

Quadro 22 – Conteúdo de configuração de um módulo

```
{
    "backend" : {
        "menu" : [
            {
                "label" : "Dashboard",
                "id" : "dashboard",
                "level" : "10",
                "path" : "/home"
            }
        ]
    },
    "global" : {
        "observer" : [
            {
                "event":"modules_loaded",
                "class": "api",
                "method": "initialize",
                "singleton": true
            }
        ],
        "async_tasks": [
            {
                "method" : "defaultTask",
                "class" : "main"
            },

            {
                "method" : "secondaryTask",
                "class" : "main"
            }
        ],
        "web" : {···
        },
        "api" : [···
        ],
        "setup" : {···
        }
    }
}
```

Fonte: elaborado pelo autor.

No Quadro 23 é apresentado o método `shouldDelay()` que é responsável pela validação da necessidade de *delay* em uma requisição. Para fazer isso é verificado se a requisição foi feita durante o período que permite a aplicação de um *delay* em sua execução. Também é apresentado o `hasReachedLimit()`, que verifica se o limite diário de requisições foi atingido pelo cliente.

Quadro 23 – Métodos de validação de delay e limite

```
shouldDelay() {
    if ( !this.delay_from || !this.delay_to ) {
        return false
    }
    if ( (this.delay_from.hour() < moment().hour() && this.delay_to.hour() > moment().hour() ) ||
    (this.delay_from.minute() < moment().minute() && this.delay_to.minute() > moment().minute() )) {
        return true
    }

    return false
}

hasReachedLimit() {
    let day = new Date()
    let currentDate = new Date(Date.UTC(day.getFullYear(),day.getMonth(), day.getDate()))
    if (!this._data.requests) {
        return false
    }
    for (let key in this._data.requests) {
        if (this._data.requests[key].date.getTime() == currentDate.getTime() && this._data.requests[key].count >= this._data.daily_limit) {
            return true
        }
    }

    return false
}
```

Fonte: elaborado pelo autor.

Ao receber uma requisição o gerenciador de API precisa carregar o *handler* que suporta o tipo da requisição recebida. Para carregar o *handler*, como demonstrado no Quadro 24, as configurações dos módulos são requisitadas e é procurado pelo atributo `api_manager` que tem todos os protocolos e arquiteturas suportadas pela aplicação, bem como a classe que deverá ser instanciada. Após ter sua instância criada através do método de *factory* `getModel()`, a instância do *handler* é retornada para ser utilizada pelo gerenciador de API.

Quadro 24 – Carregamento do handler de um protocolo ou arquitetura

```
let searchableScopes = ['global']
for (let scope of searchableScopes) {
    if (Project.app.module_data[scope].hasOwnProperty('api_manager')) {
        let apiManager = Project.app.module_data[scope]['api_manager']
        for (let handle of apiManager) {
            if (application.type == handle.type) {
                return Project.getModel(handle.moduleName+"/"+handle.handler)
            }
        }
    }
}
return null
```

Fonte: elaborado pelo autor.

No Quadro 25 é demonstrado como a aplicação opera ao receber requisições concorrentes, no qual é possível ver quando as requisições começam a ser negadas através do método `_denyExec()`. Também é apresentado como elas são atrasadas através do método `_delay()`.

Quadro 25 – Validação de requisições

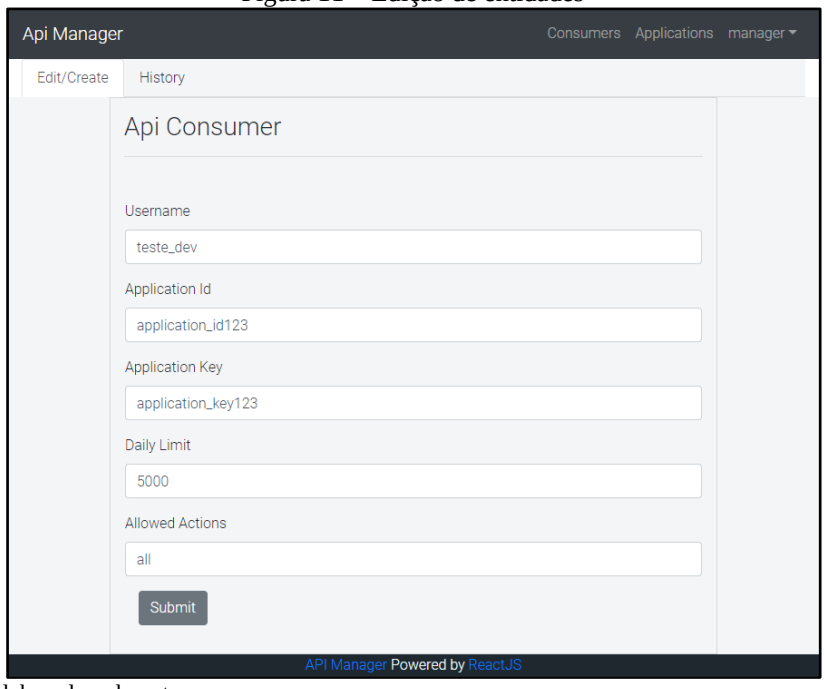
```
Project.log('Current concurrent request limit: '+this.concurrent_limit_prevent)
if (this._verifyPendingActions() > this.concurrent_limit_prevent ) {
    this._denyExec('Too many requests')
    return x._postDispatch()
}

let delayCounter = 0
while (this.flags.includes(DELAY_EXECUTION) &&  //if this action is supposed to be delayed
    this.application.shouldDelay() && //if the application has delay time configured
    this._verifyPendingActions() > this.delay_with_qty
) { // and if current actions breaks the limit of concurent action
    if (delayCounter == this.delay_limit) {
        if (!this.allow_passtrough) {
            this.flags.push(PREVENT_EXECUTION)
        }
        break
    }
    this._delay()
    x.wasDelayed = true
    delayCounter++

}
```

Fonte: elaborado pelo autor.

APÊNDICE C – DEMONSTRAÇÃO DA APLICAÇÃO WEB CONSUMIDORA DA API

Para aplicar de forma funcional a API de edição e criação de entidades foi implementado um formulário que permite esse tipo de funcionalidade. Esse formulário está apresentado na Figura 11, na qual é demonstrada a visualização e edição de um consumidor de API.

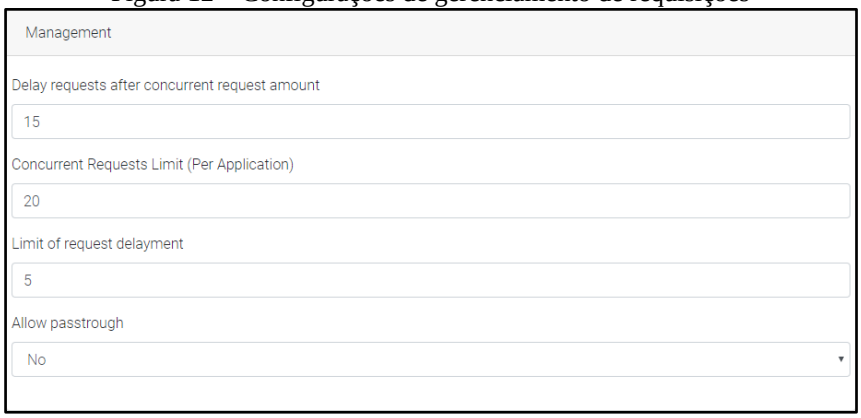
Figura 11 – Edição de entidades



Fonte: elaborado pelo autor.

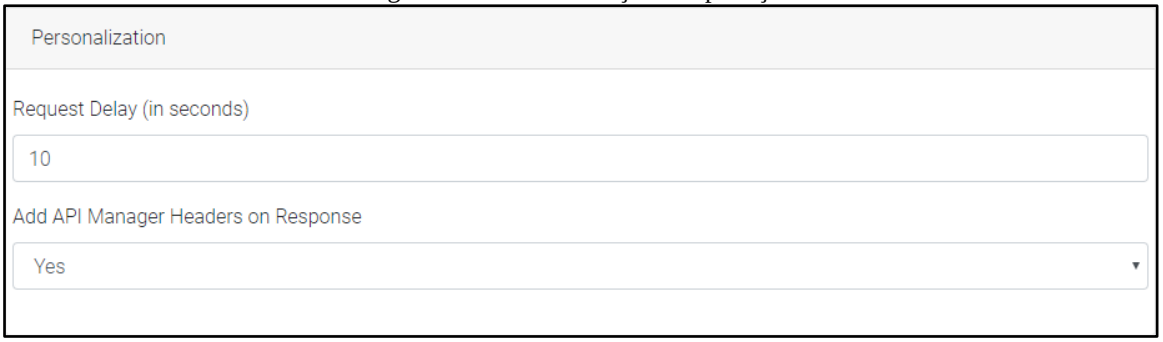
Para demonstrar a API de definição de configurações da aplicação foi implementada a tela de configurações na qual é possível definir configurações definidas pela aplicação. Alguns dos campos fornecidos na tela de configurações estão demonstrados na Figura 12 e Figura 13.

Figura 12 – Configurações de gerenciamento de requisições



Fonte: elaborado pelo autor.

Figura 13 – Personalização da aplicação

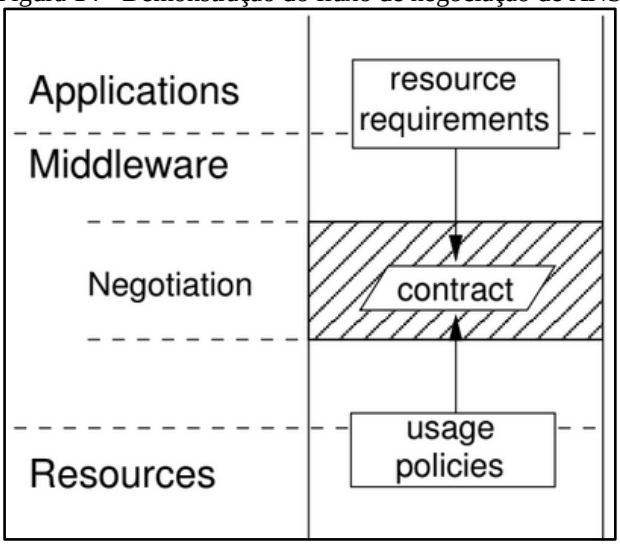


Fonte: elaborado pelo autor.

ANEXO A – DESCRIÇÃO

Na Figura 14 é demonstrado como funciona a negociação de utilização de recursos do trabalho correlato no qual as requisições de recursos (*resource requirements*) negociam com as regras de uso (*usage policies*) um contrato (*contract*) para utilização da API.

Figura 14 - Demonstração do fluxo de negociação de ANS



Fonte: Overeinder, Verkaik e Brazier (2008).