FACULDADE DE TECNOLOGIA DE SÃO JOSÉ DOS CAMPOS
FATEC PROFESSOR JESSEN VIDAL

MARCOS VINICIUS COSTA BUSTAMANTE

MAGIC SURFACE: WEB SERVICE PARA O DESENVOLVIMENTO DE APLICATIVOS DE REALIDADE AUMENTADA

São José dos Campos
2015

MARCOS VINICIUS COSTA BUSTAMANTE

MAGIC SURFACE: WEB SERVICE PARA O DESENVOLVIMENTO DE APLICATIVOS DE REALIDADE AUMENTADA

Trabalho de Graduação apresentado à Faculdade de Tecnologia São José dos Campos como parte dos requisitos necessários para obtenção do título de tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Giuliano Araujo Bertoti

São José dos Campos
2015

BUSTAMANTE, Marcos Vinicius Costa Bustamante
Magic Surface: web service para o desenvolvimento de aplicativos de realidade aumentada.
São José dos Campos, 2015.
56f.

Trabalho de Graduação – Curso de Tecnologia em Análise e Desenvolvimento de Sistemas, FATEC de São José dos Campos: Professor Jessen Vidal, 2015.
Orientador: Msc. Giuliano Araujo Bertoti.

1. Áreas de conhecimento. I. Faculdade de Tecnologia. FATEC de São José dos Campos: Professor Jessen Vidal. Divisão de Informação e Documentação. II. Título

REFERÊNCIA BIBLIOGRÁFICA –

BUSTAMANTE, Marcos Vinicius Costa Bustamante. **Magic Surface: web service para o desenvolvimento de aplicativos de realidade aumentada.** 2015. 56f. Trabalho de Graduação - FATEC de São José dos Campos: Professor Jessen Vidal.

CESSÃO DE DIREITOS –

NOME DO AUTOR: Marcos Vinicius Costa Bustamante
TÍTULO DO TRABALHO: Magic Surface: web service para o desenvolvimento de aplicativos de realidade aumentada
TIPO DO TRABALHO/ANO: Trabalho de Graduação / 2015.

Marcos Vinicius Costa Bustamante
Praça Ana Berling de Macedo, nº 158, Bairro: Monte Castelo
CEP 12.215-530 – São José dos Campos – São Paulo

Marcos Vinicius Costa Bustamante

MAGIC SURFACE: WEB SERVICE PARA O DESENVOLVIMENTO DE APLICATIVOS DE REALIDADE AUMENTADA

Trabalho de Graduação apresentado à Faculdade de Tecnologia São José dos Campos, como parte dos requisitos necessários para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Composição da Banca

_____

Reinaldo Gen Ichiro Arakaki, Doutor, FATEC

_____

Giuliano Araujo Bertoti, Mestre, FATEC

_____

Luan Rafael Castro Pinheiro, Tecnólogo, FATEC

_____/_____/_____

DATA DA APROVAÇÃO

DEDICATÓRIA

Dedico a minha família, e a todos que fizeram parte de minha vida nessa longa jornada em busca do conhecimento.

AGRADECIMENTOS

Agradeço ao professor e orientador Giuliano Bertoti, pelo apoio e encorajamento durante a minha vida acadêmica junto a Fatec e por todo o conhecimento transmitido por ele.

Ao professor Reinaldo Arakaki e a professora Renata Ruiz que ministraram as aulas de Trabalho de Graduação e auxiliaram no desenvolvimento deste trabalho.

A todos meus amigos que me acompanharam e me auxiliaram no decorrer de todo o curso e no desenvolvimento do trabalho de graduação. Em especial aos meus amigos Erich Rodrigues Ferrares e Marcos Costa Pinto que também fizeram parte da equipe de maratona de programação Paitom, e juntos obtivemos diversas conquistas para a Fatec e um rico conhecimento em algoritmos, que me norteou na elaboração de um projeto mais eficiente.

Agradeço aos demais professores, pelo tempo e conhecimento gasto e transmitido. Aos meus pais, por terem me mostrado os caminhos corretos a seguir e apoiado durante toda a minha jornada.

Acima de tudo agradeço a Deus por ter me dado força e sabedoria para concluir este trabalho, me iluminando nas decisões a serem tomadas até a conclusão do trabalho e da graduação.

EPÍGRAFE

“Ciência da Computação está tão relacionada aos computadores quanto a Astronomia aos telescópios, Biologia aos microscópios, ou Química aos tubos de ensaio. A Ciência não estuda ferramentas. Ela estuda como nós as utilizamos, e o que descobrimos com elas.”

Edsger Dijkstra.

RESUMO

Com o constante crescimento do uso de tecnologia no cotidiano das pessoas, faz-se necessário à utilização de novas formas de inserir informações virtuais no dia a dia dos seres humanos. Uma solução pode ser o uso da realidade aumentada, mesclando o mundo real com um mundo virtual, aumentado o fluxo de informações presentes na rotina das pessoas. O objetivo deste trabalho é desenvolver um *WebService*, chamado Magic Surface, para o desenvolvimento de aplicações de realidade aumentada que se baseiam em pontos geográficos do mapa. Baseada no protocolo de comunicação HTTP, o Magic Surface oferecerá novos recursos para o desenvolvimento de aplicações de realidade aumentada, com a finalidade de diminuir o tempo de desenvolvimento destas aplicações.

Palavras-Chave: Realidade aumentada; Mágica; Tecnologia no cotidiano; Eternizar momentos; API.

ABSTRACT

With the constant increase of technology that people are using, it is important to get new ways to display virtual information into the daily lives of human beings. A solution for this case can be the use of augmented reality, mixing the real world with a virtual world, increasing the data flow present in the people routine. The objective of this work is to develop a WebService, called Magic Surface, to the real world geography augmented applications development. Inspired in the HTTP protocol, the Magic Surface offers new tools to develop the augmented reality applications, aim to decrease the development time duration of these applications.

Key-word: Augment reality; magic; day by day technology; eternalize moments; API;

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
APP	*Application*
AWS	*Amazon Web Service*
CORS	*Cross-Origin Resource Sharing*
GAE	*Google App Engine*
HTTP	*Hypertext Transfer Protocol*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
MIT	*Massachusetts Institute of Technology*
MVC	*Model View Control*
OS	*Operating System*
REST	*Representational State Transfer*
SDK	*Software Development Kit*
TCP	*Transmission Control Protocol*
URL	*Uniform Resource Locator*

SUMÁRIO

1. Introdução

1.1. Motivação

Qualquer tecnologia suficientemente avançada é indistinguível de magia (Clarke). Mágica, tal palavra é mencionada sempre que um evento que impressione os seres humanos ocorre. Muitos diriam que a possibilidade de se vivenciar um evento que já se passou é uma forma de mágica, mas para Clarke isso se trata de tecnologia.

Imagine que uma pessoa através de uma superfície tecnológica consiga, ao aponta - lá para o local de seu nascimento, ver e sentir todas as emoções que ocorreram nessa ocasião, poder ver com detalhes do ambiente e todas as decisões e atos esquecidos no tempo, rever pessoas e sentir emoções marcantes que já ocorreram há anos ou décadas. Isso para muitos seria uma mágica.

Por ser algo intrigante e ao mesmo tempo atraente, pesquisadores do MIT criaram o curso "*Indistinguishable From… Magic as Interface, Technology, and Tradition*", onde toda a mágica popular é estudada de forma tecnológica, mesclando ambos os conceitos para produzir experiências únicas e marcantes, senão emocionantes para as pessoas.

A interface de um dispositivo tecnológico pode ser um meio de proporcionar experiências mágicas, pois o mesmo oferece ao usuário o poder de armazenar momentos em um ambiente digital. Com isso, torna-se possível reviver determinadas ações e experiências de vida no futuro, seja pelo próprio usuário ou por gerações futuras de sua linhagem.

A eternidade sempre foi à busca do homem, e com a mescla de tecnologia e mágica, ela pode ser alcançada. Não de uma forma material, mas sim digital, mesclando magia com emoções.

1.2. Objetivo

Nesta sessão será abordada uma especificação dos objetivos gerais do trabalho, onde há uma descrição geral e alguns tópicos especificando o objetivo.

1.2.1. Objetivo Geral

O objetivo deste trabalho é construir um *WebService* onde seja possível persistir e consumir dados de realidade aumentada.

1.2.2. Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Criação de um *WebService.*
- Integração da aplicação com servidores dedicados.

- Criar um facilitador javascript para consumir o serviço.
- Autenticar o usuário e a aplicação para elevar a segurança do sistema.

1.3. Metodologia

Por ser um *WebService*, conceitos importantes e necessários foram abordados no trabalho e utilizados no projeto, como o conceito HTTP que é um padrão Web que nos permite trafegar dados pela rede entre clientes e servidores de forma isolada, ou seja, o cliente não precisa saber como o servidor foi implementado e o servidor também não precisa saber como o cliente foi implementado, ambos se comunicam através do protocolo HTTP.

Com o HTTP presente no projeto uma arquitetura de métodos e padrões de acesso foi proposta, com o objetivo de tornar a utilização do serviço fácil e genérico o bastante para que diversos clientes consigam usufruir das funcionalidades existentes.

Cada método do Serviço fornece um tipo diferente de tratamento de dados, com isso a eficiência e eficácia de algoritmos e estruturas de dados foi algo cuidadosamente planejado, para que a resposta do serviço seja cada vez mais rápida e precisa, evitando possíveis lentidões.

1.4. Organização do trabalho

Esse trabalho esta organizado nos seguintes capítulos:

- Capítulo 2 - Revisão Bibliográfica: Contem o estudo e conceitos importantes com o que diz respeito ao trabalho.
- Capítulo 3 - Desenvolvimento: Explica o desenvolvimento do *WebService* para o tratamento de dados.
- Capítulo 4 - Resultados: Mostra as possíveis respostas ao efetuar uma requisição para o servidor.
- Capítulo 5 - Considerações finais: Conclui e apresenta possíveis trabalhos futuros relacionados a este.

2. Revisão Bibliográfica

Este capítulo apresenta os conceitos e ferramentas estudadas para o desenvolvimento do *WebService*.

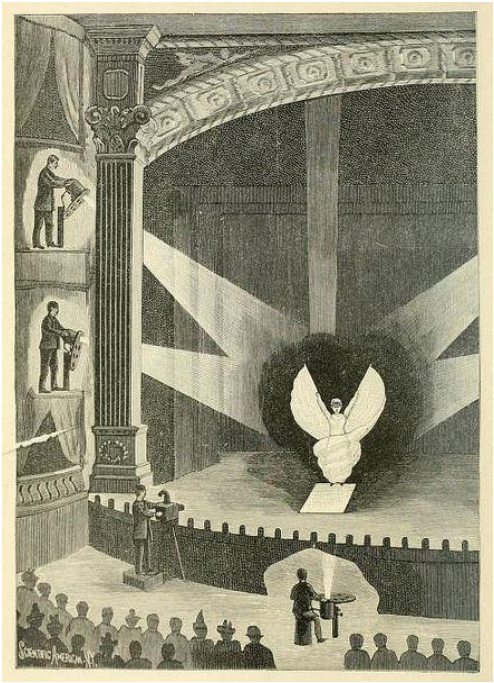
2.1. Mágica e Computação

Magia não é ciência e, portanto, não é objeto de estudo deste trabalho. Neste contexto, o termo "magia" é usado apenas como metáfora para o conceito de realidade aumentada conforme estudo do MIT Media Lab (LIN, Zhijie, 2008).

A ciência é a arte de causar mudanças no ocorrido em conformidade com a vontade (Aleister Crowley), este conceito pode ser utilizado para descrever diversas aplicações tecnológicas. Popularmente, quando uma tecnologia nova, com engenharias e conceitos novos entra no cotidiano dos seres humanos, a palavra “mágica” é mencionada diversas vezes.

Com o auxilio da tecnologia, muitos cenários diferentes já foram e estão sendo criados para impressionar as pessoas. Um exemplo clássico desse cenário é o *show* de ilusionismo, que com o auxilio da tecnologia, o ilusionista “magicamente” faz com que um ser humanóide apareça diante de diversas pessoas (Figura 1), fato que impressiona diversos espectadores do espetáculo.

Figura 1 – Ser Humanóide formado com tecnologia



Fonte: Neutral Milk Hotel Pôster, 2015

A mágica pode ser usada como uma metáfora para expressar a primeira impressão ao se experimentar uma tecnologia nova, pois ela ainda não está inscrita nas tecnologias que os usuários usufruem normalmente, seja um comando por voz que resulte em uma ação ou em um objeto virtual que imite os movimentos dos seres humanos. Tais conceitos mágicos possuem fundamentos teóricos dentro da realidade misturada.

2.2. Realidade Misturada

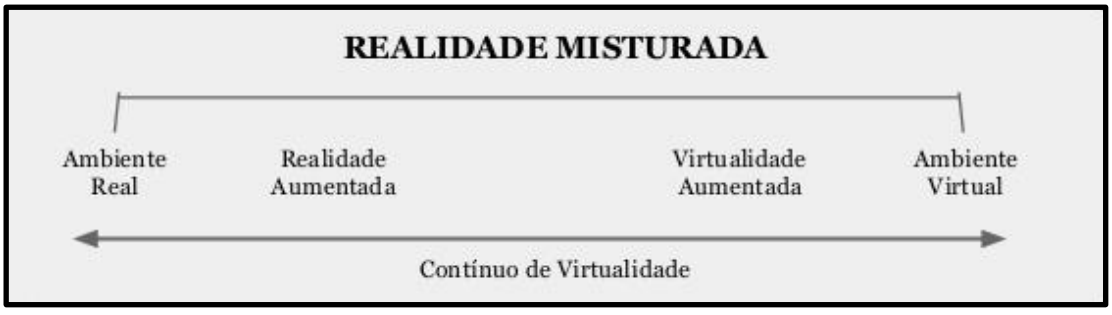
A realidade misturada pode ser definida como a sobreposição de objetos virtuais gerados por computador com o ambiente físico, mostrada ao usuário, com o apoio de algum dispositivo tecnológico, em tempo real (Kirner, 2004).

Permitindo a mistura entre dois cenários diferentes, real e virtual, a realidade misturada permite tanto mesclar elementos virtuais em um ambiente real como imergir todo um cenário real em um contexto virtual, gerando a integração entre real e virtual por meio de um usuário.

O objetivo de um dispositivo ou sistema de realidade misturada é prover um ambiente ou cenário super realista, fazendo com que o usuário não perceba a diferença entre ambos, causando a percepção de que o contexto em que está inserindo é um só, e não a junção de dois ambientes, virtual e real.

A realidade misturada pode ser dividida em dois grupos, realidade aumentada e virtualidade aumentada. O primeiro pode ser denominado quando se tem o ambiente real como o principal ou como predominante no contexto, já a virtualidade aumentada é presente quando o ambiente principal ou predominante é o virtual (Figura 2).

Figura 2 – Divisão da realidade misturada



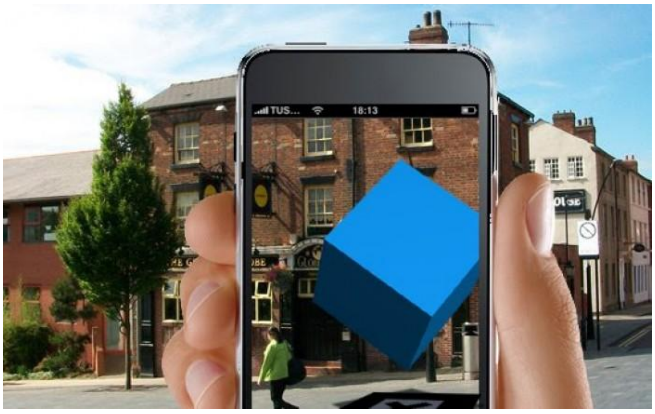
Fonte: DOMINGUES, 2007

2.2.1. Realidade Aumentada

É o enriquecimento do ambiente real com objetos virtuais, usando algum dispositivo tecnológico, funcionando em tempo real (Kirner, 2004). Porém muitos autores utilizam em seus documentos o conceito de realidade misturada como o de realidade aumentada.

Assim se torna possível inserir e manipular objetos virtuais em um contexto real, ou seja, a imagem do ambiente real deve ser capturada por um dispositivo tecnológico e processada junto aos objetos virtuais, após o processamento em tempo real ter encerrado, o resultado é mostrado através em uma superfície de saída de dados tecnológica para o usuário (Figura 3 - 4).

Figura 3 – Inserção de um objeto virtual no mundo real



Fonte: Arte com Realidade Aumentada, 2011

Figura 4 – Inserção de um ser humanóide no mundo real

Fonte: Arte com Realidade Aumentada, 2011

Após todo o processamento, fica visível para o usuário que o elemento virtual foi quem entrou no mundo real por meio de um dispositivo eletrônico, sendo o cenário real predominante no contexto da aplicação.

2.2.2. Virtualidade Aumentada

O protagonista não se vê projetado, em seu lugar um avatar ou apenas a reação de seus movimentos, o mundo da matéria e do movimento permanece real, o tempo é linear, mas o participante vivencia um espaço virtual diante de si, e costuma acoplar-se a ele, cuja existência torna-se mais sedutora do que a da realidade (ISRAEL, 2012).

Assim se torna possível a construção de objetos virtuais que expressão algum tipo de ação baseada no mundo real, um exemplo desse conceito são os jogos digitais que lêem ações humanas através de um dispositivo eletrônico, e em tempo real espelham os movimentos em humanóides digitais presentes no mundo virtual (Figura 5).

Figura 5 – Pessoas controlando seres humanóides com movimentos do corpo

Fonte: Xbox

Na realidade virtual, também se torna possível inserir um usuário inteiramente no mundo virtual, onde tudo ao seu redor, cenários e seres, são virtuais, porém o movimento de seu corpo e membros é espelhado no mundo virtual. Trazendo a sensação de que o mundo virtual é o único existente no contexto da aplicação (Figura 6 - 7)

Figura 6 – Usuário imergido em um mundo virtual utilizando um tispositivo tecnologico

Fonte: Sony

Figura 7 – Imagem visualizada pelo usuário imergido no mundo virtual

Fonte: Sony

Assim o ambiente predominante é o virtual, onde esse utiliza ações processadas com base em um usuário existente no mundo real para efetuar algum tipo de ação no mundo virtual.

2.3. Projetos de realidade aumentada

Há diversas aplicações tecnológicas que utilizam os conceitos de realidade aumentada, onde a inserção de componentes virtuais no mundo real é a principal proposta de valor. Podemos tomar como exemplo o aplicativo para a plataforma Android “Augment” e o projeto

do MIT “Second Surface”, ambos possuem características muito semelhantes, porém ambos não disponibilizam uma API pública para efetuar uma comunicação direta com o *WebService* responsável por tratar os dados de realidade aumentada.

2.3.1. Augment

Com ele é possível visualizar um objeto 3D no mundo real, tal objeto pode ser criado através de softwares dedicados a esta finalidade e inserido no aplicativo apenas para uma visualização de realidade aumentada.

Este está disponível para as plataformas Android e IOS e não possui nenhum tipo de API para auxiliar o desenvolvimento de novos projetos de realidade aumentada. Tornando-o um produto para ser comercializado e não um WebService que auxilie os desenvolvedores.

Figura 8 – Inserção de objetos virtuais no mundo real com o aplicativo Augment

Fonte: Augment, 2015

2.3.2. Second Surface

Aplicativo criado no MIT para inserir objetos virtuais criados por pessoas no mundo real, ou seja, um usuário pode inserir uma foto tirada em um espaço geográfico ou criar objetos através do movimento dos dedos na tela.

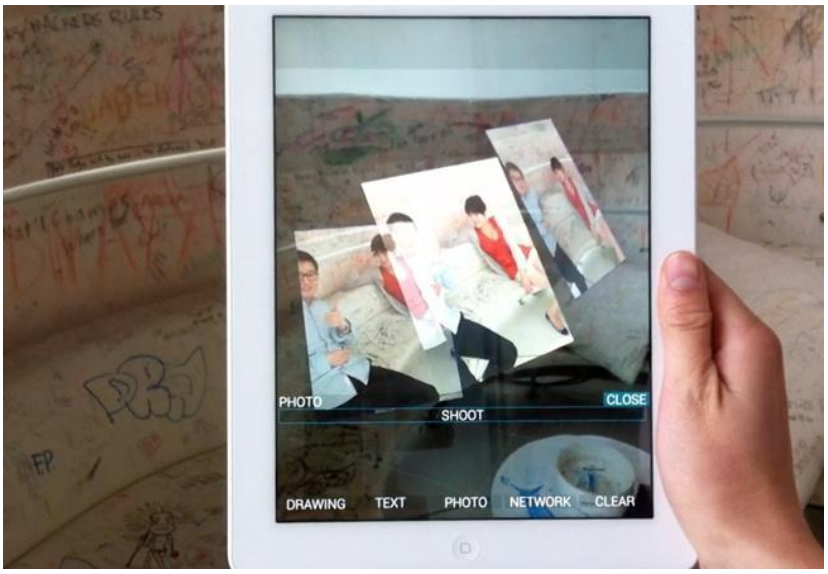
Quando um objeto é criado por um usuário, este é inserido no mundo real de uma forma que se outro usuário está assistindo a criação do objeto com outro dispositivo, tem a impressão de que o objeto está saindo do dispositivo de origem do objeto.

Porém, assim como o Augment, não há uma API para que outros usuários possam usufruir de toda a tecnologia criada, tornando as funcionalidades e código inacessíveis a outros usuários.

Figura 9 – Aplicativo Second Surface sendo usado por dois usuários ao mesmo tempo

Fonte: creativeapplications

Figura 10 – Aplicativo Second Surface salvando fotos no mundo real



Fonte: creativeapplications

Figura 11 – Second Surface sendo usado por um usuário e visualizado por outro

Fonte: creativeapplications

3. Desenvolvimento

Este capítulo apresenta as etapas do desenvolvimento do *WebService*, ilustradas com pequenos trechos de código.

3.1. Arquitetura do *WebService*

Pensando em uma API simples e poderosa, a arquitetura deste projeto foi modelada de forma "magra", com poucos métodos e com diversas funcionalidades. Por ter poucos métodos, o código se torna mais simples e menos propenso a ter redundância, ou seja, se reduz a possibilidade de ter funções diferentes com comportamentos iguais.

Com uma API de poucos métodos, há a necessidade de se ter mecanismos para a manipulação das informações baseadas nas necessidades dos usuários. Por este motivo, um objeto serializável JSON pode ser passado como parâmetro nas chamadas Ajax, esse objeto pode conter diversos filtros e opções para auxiliar a API em relação a como ela deve efetuar a manipulação e o tratamento dos dados.

Toda requisição feita pro *WebService* é processada por um *handler*, onde este se comporta da mesma maneira que um *controller,* presente no padrão de projeto MVC, tal *controller* redireciona a requisição para um arquivo que realiza o acesso ao banco de dados e manipulação dos dados, este se comporta como um *model* presente no MVC.

Após o processo ser concluído, um JSON é retornado com a resposta para o usuário. Para que toda a comunicação ocorra, temos que configurar o CORS da aplicação, onde o cabeçalho da requisição deve ser configurado para que domínios distintos possam requisitar o serviço.

3.2. Configurando o CORS

Sempre que uma requisição é feita utilizando os verbos presentes do protocolo de comunicação HTTP, o verbo *options* é chamado antes de qualquer outro, para que este verifique se há algum processo que deve ser executado antes da requisição principal.

Para configurar o CORS do *WebService*, a requisição com o método *options* é capturada pelo servidor e um processo de modificação do *header* é executado. No fragmento de código a seguir, podemos verificar como o método *options* se comporta no *handler* do projeto. Onde o *header* da requisição é processado e salvo na variável `self.response.headers` no formato de um dicionário Python. Devido ao fato de ser possível alterar os dados do dicionário criado, foram inseridos três novos dados nessa variável para configurar uma permissão de requisição para qualquer domínio.

```
class MSHandler(webapp2.RequestHandler):
    def options(self):
        self.response.headers[‘Access-Control-Allow-Origin’] = ‘*’
        self.response.headers[‘Access-Control-Allow-Methods’] =
‘get,post,options’
        self.response.headers[‘Access-Control-Allow-Headers’] =
“Origin, X-Requested-With, Content-Type, Accept, X-PINGOTHER, X-
CSRFToken”
```

Fragmento de código – 1

Ao efetuar a requisição e executar o processo solicitado, há a necessidade de se modificar o *header* da resposta antes de retorná-la ao cliente, para que o protocolo de comunicação saiba que ele pode retornar os dados para domínios diferentes, assim como foi feito na requisição.

Com o intuito de criar uma rotina centralizada e que execute o mesmo comportamento em todas as requisições, foi criado um *decorator* Python e utilizado em todas as funções dos *handlers* da API.

No fragmento de código a seguir, é mostrado as instruções do *decorator* mencionado, onde o método `callable_from_browser` recebe a função à ser processada, modifica o *header* de comunicação inserindo variáveis no dicionário Python `self.response.headers` e depois executa a função que no contexto é representada pela variável `view_func`.

```
def callable_from_browser(view_func):
    def _decorated(self):
        self.response.headers['Access-Control-Allow-Origin'] = '*'
        self.response.headers['Access-Control-Allow-Methods'] =
'get,post,options'
        self.response.headers['Access-Control-Allow-Headers'] =
'Origin, X-Requested-With, Content-Type, Content-Range, Content-
Disposition, Content-Description, Accept, X-PINGOTHER, X-CSRFToken'
```

```
        view_func(self)

    _decorated.__doc__ = view_func.__doc__
    return _decorated
```

Fragmento de código - 2

3.3. Métodos da API

Para modelar uma API “magra” foi pensando nos casos em que uma requisição é feita ao servidor, e analisado qual o comportamento que estas devem ter. Após a análise de alguns serviços REST e as respostas que estes forneciam, foram modelados os métodos *get*, *save*, *list e delete* para cada tipo de serviço:

3.3.1. *Get*

Para que uma pagina possa ser dinâmica e trazer uma melhor usabilidade para seus usuários, ela deve ser capaz de capturar informações em tempo de execução do servidor. Tais requisições feitas devem conter informações úteis para que o contexto de realidade aumentada seja rico em informações.

O método *get* é responsável por capturar informações de um e apenas um objeto, por esse motivo ele recebe como parâmetro obrigatório o identificador único do objeto, e como opção, pode receber um conjunto de parâmetros para que objetos relacionados a este sejam tratados e retornados para a tela.

Tal método deve ser chamado através de uma requisição HTTP e assinalado com o verbo GET.

3.3.2. *Save*

Como a API é responsável pela captura e tratamento de dados, esta tem que ser capaz de persistir os mesmos em uma base de dados. Para que tal ação tenha um funcionamento dinâmico e eficiente, foi criado o método *save*.

Este recebe como parâmetro um objeto JSON através do corpo da mensagem HTTP, assinalado com o verbo POST, tal JSON contém todos os campos obrigatórios e, se necessário, alguns opcionais presentes no contexto do modelo.

Com todos os dados sob o domínio da API, diversos tratamentos e verificações necessárias são executados para que o dado seja salvo de forma consistente no banco.

3.3.3. *List*

Um comportamento comum aos serviços REST, é a necessidade de se listar um número razoável de objetos, paginados ou não, com a finalidade de popular uma tela sem necessariamente conhecer algum identificador único do objeto de interesse. Com apenas uma requisição é possível listar um determinado tipo de objeto, com ou sem filtros, permitindo que a tela mostre informações dinamicamente aos usuários.

Para que o método funcione corretamente, este requer que uma requisição HTTP assinalada com o verbo GET seja enviada ao servidor REST, juntamente com um objeto JSON que contém todos os filtros necessários para tratar e listar as informações contidas na base de dados.

3.3.4. *Delete*

Com o intuito de esconder informações que não são mais relevantes para o cotidiano do usuário, o método *detele* foi analisado e projetado para o contexto da aplicação. Sempre que a API recebe uma requisição para apagar uma determinada informação do banco, está não apaga o dado de fato, mas muda o valor de uma variável de estado binária que diz se a informação está ou não apagada do banco.

Por ser a responsabilidade de uma variável de estado dizer o que foi apagado no banco, é possível recuperar facilmente um dado removido por acidente, pois é só alterar o valor de um dado no banco que o objeto removido se torna visível novamente.

Para que o método funcione corretamente, este requer que uma requisição HTTP assinalada com o verbo DELETE seja enviada ao servidor REST, juntamente com o identificador do objeto a ser excluído.

3.4. Segurança do *WebService*

Devido ao fato do WebService poder ser consumido por diversos usuários, há a necessidade de separar as informações por aplicação e gerar métricas de uso para cada uma delas. Para que tais comportamentos fossem possíveis, todo usuário antes de começar a consumir o serviço, deve obrigatoriamente se registrar no site do Magic Surface.

Com o cadastro realizado, diversas configurações podem ser efetuadas. Uma configuração importante é o registro de APPs no sistema, cada usuário ganha o direito de registrar três APPs diferentes.

Uma APP é responsável por autenticar um usuário que efetua uma determinada requisição para o servidor, e agrupar todo o conjunto de informações salvas de um projeto. Devido ao fato de que cada usuário tem a possibilidade de cadastrar três APPs diferentes, este

pode separar a manipulação de dados entre seus diversos projetos, ou seja, cada projeto pode ter uma APP diferente.

Como forma de identificação da APP, quando esta é criada, um *token* aleatório e único é gerado pelo sistema. Tal *token* é responsável por identificar qual é a APP que está efetuando a requisição em um dado momento e como os dados devem ser filtrados.

Por ser responsabilidade do servidor filtrar os dados e identificar a aplicação consumidora, o *token* de uma APP e o nome de usuário dono da aplicação, devem ser inseridos no contexto de todas as requisições efetuadas.

Caso tais informações sejam omitidas, isso irá resultar em um erro, solicitando que esses dados sejam informados corretamente para que a requisição possa ser processada de forma integra.

3.5. Tratamento de dados da API

Para verificar e autenticar os autores das requisições efetuadas ao *WebService*, foi criado um *decorator* Python que autentica o *token* e o nome de usuário passados pelas requisições. Caso alguma validação falhe, o usuário irá receberá um erro com uma mensagem adequada.

No fragmento de código abaixo é possível verificar como o *decorator* efetua tal validação, onde o *token* e o nome de usuário são procurados nas variáveis `self.request.body`, `self.request.POST` e `self.request.GET`. Caso elas não estejam presentes no contexto ou não forem dados que o servidor conheça, uma exceção é lançada a partir do método `MSException`. Após todas as validações serem feitas, o método passado para o *decorator* python na variável `method` é executado e ao final do processo um *log* é registrado de acordo com o tipo de ação passada para a variável `activity`.

```
def app_data_required(activity):
    def decorator(method):
        def wrapper(self, *args, **kwargs):
            body = json.loads(self.request.body) if
is_json(self.request.body) else {}
            token = self.request.GET.get('token') or
self.request.POST.get('token') or body.get('token')
            user_id = self.request.GET.get('user_id') or
```

```
self.request.POST.get('user_id') or body.get('user_id')

            if not token or not user_id:
                raise MSException(u'App token e user_id devem ser
enviados')
            app = App.query(App.token == token).get()
            user = User.get_by_id(user_id)
            if app and user and not app.deleted and user_id !=
app.user_id:
                self.app_data = {
                    'app': app.to_dict_json(),
                    'user': user.to_dict_json()
                }
                method(self, *args, **kwargs)
                ActivityLog.save(user_id=user_id,
app_name=app.name,token=token, activity=activity)
            else:
                raise MSException(u'App token e user_id inválidos')
        return functools.update_wrapper(wrapper, method)
    return decorator
```

Fragmento de código - 3

Validado os dados do usuário, o servidor executa o processo solicitado ao *handler*. O *handler* tem a mesma responsabilidade de um controlador presente no padrão de projeto MVC. Cada tipo de serviço do *WebService* possui quatro métodos distintos, descritos no capítulo 2.3. Cada método possui o seu próprio *handler* com todos os *decorators* de validações necessários. Tais serviços são manipulados pelas classes `Layers` e `Arquivos`.

Apesar de cada *handler* representar um método de um serviço do *WebService* estes apenas recebem uma requisição, chamam uma função de tratamento de dados para depois retornar uma resposta para o usuário.

No fragmento de código a seguir é possível verificar como os *handlers* do serviço de arquivos foram criados. É possível observar os *decorators* python, descritos anteriormente, sento usados nas três linhas que antecedem os métodos `pos` e `get`. Os *handlers* sempre tentam ler informações necessárias para o processo das variáveis do contexto de comunicação

`self.request.POST` e `self.request.GET`, chamam a função que executa o serviço necessário, os arquivos de serviço são assinalados com o sufixo '_svc', e retornam uma resposta para o cliente através da função `self.response.out.write`.

```
class FileSaveHandler(MSHandler):
    @callable_from_browser
    @ajax_error
    @app_data_required(activity='FileSave')
    def post(self):
        layer_id = self.request.POST.get('layerId')
        file_item = self.request.POST.get('file')
        app_data = self.get_app_data()
        result = save_file_svc.save(app_data, layer_id, file_item)
        self.response.out.write(json.dumps(result))

class FileGetHandler(MSHandler):
    @callable_from_browser
    @ajax_error
    @app_data_required(activity='FileGet')
    def get(self):
        file_id = self.request.GET.get('fileId')
        result = get_file_svc.get(file_id)
        self.response.out.write(json.dumps(result))

class FileListHandler(MSHandler):
    @callable_from_browser
    @ajax_error
    @app_data_required(activity='FileList')
    def get(self):
        layer_id = int(self.request.GET.get('layerId'))
        filters = json.loads(self.request.GET.get('filters'))
        result = list_file_svc.listing(layer_id, filters)
        self.response.out.write(json.dumps(result))

class FileDeleteHandler(MSHandler):
```

```
    @callable_from_browser
    @ajax_error
    @app_data_required(activity='FileDelete')
    def get(self):
        file_id = self.request.GET.get('FileId') or
self.request.GET.get('FilesId')
        result = delete_file_svc.delete_by_id(file_id)
        self.response.out.write(json.dumps(result))
```

Fragmento de código - 4

Tais funções de tratamento de dados sabem acessar a base de dados e tratar os mesmos, se comportando como um modelo do padrão de projeto MVC. Estas efetuam autenticações rigorosas para garantir a consistência e segurança dos dados, quando necessário aplicam filtros e buscam relações do dado no banco, tudo baseado nos parâmetros passados através da requisição.

No fragmento de código a seguir, é possível observar como o serviço responsável por salvar um dado se comporta. A biblioteca boto é responsável por salvar os arquivos em um banco de dados dedicado na Amazon enquanto que a classe `File` salva a referência dos arquivos no banco de dados operacional na Google. É essencial que alguns dados estejam salvos no banco de dados, para que a autenticação com a Amazon seja efetuada com sucesso, tais informações são salvas em um modelo chamado Config, e armazenadas nos seguintes atributos: `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` e `BUCKET_NAME`.

```
img_exts = ['png', 'jpg', 'jpeg']
video_exts = ['mp4']

def save(app_data, layer_id, field_storage):
    layer = Layer.get_by_id(int(layer_id))
    if layer is None:
        raise MSException(u'LayerId inválido')

    file_type = _get_file_type(field_storage)
    _s3_save(field_storage)
```

```
    instance = _get_image_instance(field_storage, file_type)

    instance.layer = layer.key
    instance.app_id = app_data['app']['id']
    instance.put()
    return instance.to_dict_json()

def _get_file_type(field_storage):
    _type = field_storage.type
    file_name = field_storage.filename

    type_ext = _type.split('/')[-1].lower()
    name_ext = file_name.split('.')[-1].lower()

    if type_ext in img_exts and name_ext in img_exts:
        return IMAGE
    if type_ext in video_exts and name_ext in video_exts:
        return VIDEO
    raise MSException(u'Tipo de arquivo invalido, verifique se ele
é: %s' % ', '.join(img_exts))

def _s3_save(field_storage):
    access_id = Config.get('AWS_ACCESS_KEY_ID')
    secret_access = Config.get('AWS_SECRET_ACCESS_KEY')
    bucket_name = Config.get('BUCKET_NAME')

    conn = boto.connect_s3(access_id, secret_access)
    bucket = conn.get_bucket(bucket_name)
    k = Key(bucket)
    k.key = field_storage.filename
    k.set_contents_from_file(field_storage.file)
    k.make_public()

def _get_image_instance(field_storage, file_type):
    bucket_name = Config.get('BUCKET_NAME')
    file_instance = File()
    file_instance.kind = file_type
```

```
    file_instance.name = field_storage.filename
    file_instance.size = field_storage.bufsize
    file_instance.link = 'https://s3-sa-east-1.amazonaws.com/%s/%s'
% (bucket_name, field_storage.filename)
    return file_instance
```

Fragmento de código – 5

3.5.1. *Layers*

Os *layers* são camadas virtuais circulares salvas com base em um mapa. Estas possuem um par de coordenadas e um raio determinado, cada camada pode ser identificada pelo usuário por um nome.

O *layer* é essencial para que outras informações possam ser salvas pelos usuários, portanto esse dado é utilizado para efetuar diversos relacionamentos entre diversas entidades do WebService.

O fragmento de código a seguir representa o modelo `Layer` que o sistema utiliza para manipular os dados no banco de dados. Tal modelo herda de `ndb.Model`, que abstrai as operações entre aplicação e banco de dados. As colunas que serão criadas na base de dados são definidas pelas instancias `ndb.IntegerProperty`, `ndb.StringProperty`, `ndb.BooleanProperty`, `ndb.FloatProperty` e `ndb.DateTimeProperty` salvas nos atributos da classe `Layer`.

```
from google.appengine.ext import ndb

class Layer(ndb.Model):
    app_id = ndb.IntegerProperty(required=True)
    name = ndb.StringProperty(required=True)
    latitude = ndb.FloatProperty(required=True)
    longitude = ndb.FloatProperty(required=True)
    radius = ndb.FloatProperty(default=20)
    deleted = ndb.BooleanProperty(default=False)
    created = ndb.DateTimeProperty(auto_now_add=True)
    last_update = ndb.DateTimeProperty(auto_now=True)
```

```
    @classmethod
    def save(cls, app_id, form):
        layer = Layer(
            name=form.get('name'),
            latitude=form.get('latitude'),
            longitude=form.get('longitude'),
            radius=form.get('radius'),
            app_id=app_id
        ).put()

        return layer
```

Fragmento de código - 6

3.5.2. Arquivos

Os arquivos são objetos virtuais que irão aparecer para o usuário no mundo real, tais arquivos podem ser vídeos ou imagens.

Para que tais objetos virtuais possam ser salvos, estes necessitam de um identificador de *layer* para que um relacionamento entre ambos os objetos seja criado. Tais arquivos não são salvos no mesmo local de suas credenciais, estes são direcionados para o *storage* da Amazon enquanto as credenciais são salvas no GAE.

O fragmento de código a seguir representa o modelo responsável por salvar as informações dos arquivos no banco de dados GAE. Assim como no modelo do Layer, a classe Arquivo herda de ndb.Model e instancia determinadas classes para criar colunas no baco de dados. As variáveis estáticas IMAGE e VIDEO foram criadas para padronizar um tipo de nomenclatura no projeto todo, assim outros scripts podem as importar e usar por padrão.

```
from google.appengine.ext import ndb

IMAGE = "image"
VIDEO = "video"

class File(ndb.Model):
```

```
    name = ndb.StringProperty(required=True)
    link = ndb.StringProperty(required=True)
    layer = ndb.KeyProperty(Layer, required=True)
    app_id = ndb.IntegerProperty(required=True)
    deleted = ndb.BooleanProperty(default=False)
    created = ndb.DateTimeProperty(auto_now_add=True)
    last_update = ndb.DateTimeProperty(auto_now=True)
    size = ndb.IntegerProperty(required=True)
    kind = ndb.StringProperty(choices=(IMAGE, VIDEO))
```

Fragmento de código - 7

3.6. Framework javascript

Com o pensamento de facilitar a maneira que as requisições são feitas para o *WebService*, foi criado um *framework* javascript capaz de efetuar todas as requisições possíveis para o serviço. Tal *framework* é um módulo AngularJs que pode ser importado no contexto de uma aplicação angular. Diversos serviços foram abstraídos e disponibilizados para serem chamados pela aplicação dos clientes, a fim de disponibilizar uma forma de acesso fácil aos serviços do *WebService.*

Antes das requisições serem disparadas, o script efetua diversas validações de dados, para que o usuário possa ter um retorno mais rápido do *status* da chamada caso algo esteja errado. Se os dados passados para o *framework* estiverem corretos, uma chamada ajax é feita para o servidor. Por ser um processo assíncrono, a requisição retorna uma promessa javascript, possibilitando que ao receber uma resposta do servidor, o código do cliente seja executado.

A promessa possui três funções distintas:

- ***success:*** é chamada sempre que a requisição ao servidor ocorre com sucesso. Ela recebe como parâmetro uma variável que irá conter um objeto javascript com a resposta do servidor respectivo ao serviço solicitado.
- ***error:*** é responsável por notificar o usuário sempre que ocorre qualquer tipo de erro no decorrer do processo. Esta recebe como parâmetro uma variável contendo um objeto javascript com informações sobre o erro que ocorreu.

- ***finally:*** é executada sempre após uma função de *error* ou *success*. Tal função é ideal para tirar componentes de requisição de tela, pois o usuário deve ser notificado sobre o resultado da mesma.

Para que o cliente não precise ficar passando o *token* e o nome de usuário da aplicação, há um serviço de configuração que salva tais informações e as disponibiliza sempre que o consumo ao servidor é efetuado.

3.7. Utilização do *framework*

Para que o tudo funcione corretamente, o AngularJs deve estar importado antes do *framework* da Magic Surface na página HTML. Com ambas as ferramentas importadas, o módulo da Magic Surface deve ser colocado como uma dependência do APP angular que as páginas do cliente irão utilizar.

O fragmento de código a seguir mostra como essa injeção de dependência é efetuada no modelo angular. Onde a referência do serviço `magicSurface` é importado no módulo.

```
angular.module('MyApp', ['magicSurface']);
```

Fragmento de código - 8

Após a dependência ter sido satisfeita, são disponibilizados diversos serviços que podem ser importados no contexto de uma aplicação angular, para facilitar o desenvolvimento de aplicações que necessitem consumir o *WebService*.

O serviço MagicSurface deve ser chamada obrigatoriamente para que todas as requisições possam ser efetuadas com sucesso, este possui a função configApp que recebe um *token* de aplicação e um nome de usuário. Com tais dados embutidos no serviço, sempre que uma requisição for efetuada os dados de autenticação também serão enviados.

O fragmento de código abaixo demonstra com dados fictícios o funcionamento do serviço, onde o módulo `MagicSurface` é importado no `controller` de uma aplicação e as variáveis `token` e `username`, com os valores do *token* e nome de usuário respectivamente, são passados para o método `configApp`.

```
angular.module('MyApp').controller('MyCtrl',
```

```
function(MagicSurface){
      var token = 'meu token';
      var username = 'meu username';
      MagicSurface.configApp(token, username);
});
```

Fragmento de código – 9

Existem outros dois serviços disponíveis, LayerApi e FileApi, para a manipulação de *Layers* e Arquivos respectivamente. Ambos os serviços possuem quatro funções, que efetuam o proposto no capítulo 3.3, tais funções possuem o nome de save, get, list e del.

O comportamento de tais serviços é mostrado no resultado deste trabalho, presente no capítulo 4.

4. Resultados

Este capítulo apresentará os resultados obtidos com o *WebService* Magic Surface. O foco será nas requisições feitas com o framework e as respostas de sucesso que estas recebem.

4.1. LayerApi

O LayerApi é um dos serviços que o framework disponibiliza, ele possui quatro métodos que nos permite efetuar algumas requisições para o servidor.

4.1.1. *Save*

Método responsável por salvar *layers*, como parâmetros obrigatórios têm que fornecer o nome, a latitude e a longitude dele. O raio é um parâmetro opcional, se este não for passado o valor 20 será tomado como padrão.

No fragmento de código 10 é possível observar a chamada de função `save` que salva um *layer* no servidor. A função recebe um objeto javascript com os dados essenciais para a existência de um *layer* e retorna uma promessa javascript que é referenciada pela variável `promise`, que possui três métodos para o registro das funções de *callback:* `success`, `error` e `finally`.

```
angular.module('MyApp').controller('MyCtrl', function(LayerApi){
    var params = {
        name: 'nome do meu Layer',
        latitude: 123,
        longitude: 321,
        radius: 30
    };

    var promise = LayerApi.save(params);

    promise.success(function(result){
        console.log('Função chamada quando ocorre sucesso');
    });
    promise.error(function(result){
        console.log('Função chamada quando ocorre erro');
    });
```

```
    promise.finally(function(result){
        console.log('Função chamada sempre');
    });
});
```

Fragmento de código – 10

No fragmento de código 11 é apresentado o JSON de retorno que a função `success` recebe. Com ele, temos todas as informações que são disponibilizadas para a apresentação do *layer* para o usuário, assim como o `id` que é utilizado para a persistência de outros dados no *WebService*.

```
{
    "created": "2015-08-10 01:02:36.435270",
    "deleted": false,
    "id": 5685265389584384,
    "last_update": "2015-08-10 01:02:36.435280",
    "latitude": 123,
    "longitude": 321,
    "name": "nome do meu Layer",
    "radius": 30
}
```

Fragmento de código – 11

4.1.2. *Get*

Método responsável por capturar dados relacionados a um e apenas um *layer*, como parâmetro obrigatório deve ser passado um identificador único para a pesquisa.

No fragmento de código 12 é possível observar a chamada de função para carregar um *layer*, a função `get` recebe como parâmetro um objeto javascript com uma chave obrigatória, `layerId`, que possui como valor um identificador único de *layer*. Esta retorna uma promessa javascript com as funções `success`, `error` e `finally`. Para que o cliente possa registrar funções a serem executadas após o termino do processo assíncrono da requisição.

```
angular.module('MyApp').controller('MyCtrl', function(LayerApi){
    var form = {
        layerId: 5685265389584384
    };

    var promise = LayerApi.get(form);

    promise.success(function(result){
        console.log('Função chamada quando ocorre sucesso');
    });
    promise.error(function(result){
        console.log('Função chamada quando ocorre erro');
    });
    promise.finally(function(result){
        console.log('Função chamada sempre');
    });
});
```

Fragmento de código - 12

No fragmento 13 é possível observar o JSON de retorno que a chamada recebe como sucesso. O objeto javascript de sucesso possui uma chave denominada `layer` com as informações do *layer*. Permitindo que o cliente utilize tais informações para notificar ou apresentar dados úteis para o usuário.

```
{
    "layer": {
        "created": "2015-08-1001: 02: 36.435270",
        "deleted": false,
        "id": 5685265389584384,
        "last_update": "2015-08-1001: 02: 36.435280",
        "latitude": 123,
        "longitude": 321,
        "name": "nome do meu Layer",
        "radius": 30
```

```
    }
}
```

Fragmento de código – 13

4.1.3. *List*

Função responsável por listar dados dos *layers*, Diferentemente das outra funções, essa não exige parâmetro obrigatório.

No fragmento de código 14 é possível observar a chamada de função para listar os *layers* da API baseado nos filtros passados, onde todos os *layers* que não foram removidos serão retornados na lista de resposta, isso se dá devido ao parâmetro `deleted` passado com o valor `false` entre os filtros. O `list` também retorna uma promessa com as três funções base: `success`, `error` e `finally`, permitindo o cliente registrar de funções de *callback* a serem chamadas no final do processo.

```
angular.module('MyApp').controller('MyCtrl', function(LayerApi){

    var params = {
        filters : {
            deleted: false
        }
    };

    var promise = LayerApi.list(params);

    promise.success(function(result){
        console.log('Função chamada quando ocorre sucesso');
    });
    promise.error(function(result){
        console.log('Função chamada quando ocorre erro');
    });
    promise.finally(function(result){
        console.log('Função chamada sempre');
    });
```

```
});
```

Fragmento de código – 14

O fragmento 15 mostra o JSON de retorno que a chamada recebe como sucesso na função de *callback* `success`. Onde há um objeto javascript com a chave `layers` possuindo uma lista de objetos como valor, cada objeto contem dados relacionados a um layer persistido no servidor.

```
{
    "layers": [
        {
            "created": "2015-08-1001: 02: 36.435270",
            "deleted": false,
            "id": 5685265389584384,
            "last_update": "2015-08-1001: 02: 36.435280",
            "latitude": 123,
            "longitude": 321,
            "name": "nome do meu Layer",
            "radius": 30
        }
    ]
}
```

Fragmento de código - 15

4.1.4. *Delete*

Método responsável por remover *layers*, como parâmetros obrigatórios têm que ser passado uma lista de identificadores único do objeto ou apenas de um.

No fragmento de código 16 é possível observar a chamada de função para remover um *layer* salvo no servidor, onde o identificador do layer está referenciado pela chave `layerId` do objeto e o `options` que deleta os dados relacionados ao *layer* chamado `delete_cascade` com o valor `true`. Essa função retorna uma promessa para que funções sejam agendadas para serem executadas ao término do processo.

```
angular.module('MyApp').controller('MyCtrl', function(LayerApi){

    var form = {
        layerId: 5685265389584384,
        options: {
            delete_cascade: true
        }
    };

    var promise = LayerApi.del(form);

    promise.success(function(result){
        console.log('Função chamada quando ocorre sucesso');
    });
    promise.error(function(result){
        console.log('Função chamada quando ocorre erro');
    });
    promise.finally(function(result){
        console.log('Função chamada sempre');
    });
});
```

Fragmento de código - 16

No fragmento 17 há o JSON de retorno que a função `success` recebe como sucesso. Onde há duas chaves: `layer` e `files`, contendo os dados respectivamente do *layer* e arquivos removidos a partir do `deleted_cascate` aplicado.

```
{
    "files": [],
    "layers": [
        {
            "created": "2015-08-1001: 02: 36.435270",
            "deleted": true,
```

```
            "id": 5685265389584384,
            "last_update": "2015-08-1001: 02: 36.435280",
            "latitude": 123,
            "longitude": 321,
            "name": "nome do meu Layer",
            "radius": 30
        }
    ]
}
```

Fragmento de código – 17

4.2. FileApi

O FileApi é um outro serviço que o framework disponibiliza, ele possui quatro métodos que nos permite efetuar algumas requisições relacionadas aos objetos virtuais para o servidor.

4.2.1. *Save*

Método responsável por salvar arquivos virtuais, como parâmetros obrigatórios este recebem um arquivo e um identificador de *layer*.

Nos fragmentos de código 18 é possível observar a chamada de função `save` para salvar o arquivo no servidor. A função recebe as variáveis `_file` e `layer_id` contendo uma instância de um File javascript e um identificador único de *layer*, e retorna pro cliente uma promessa javascript referenciada pela variável `promise` que fornece funções de *callback* para serem chamadas após o termino da execução assíncrona da função.

```
angular.module('MyApp').controller('MyCtrl', function(FileApi){
    var _file = new File()
    var layer_id = 5685265389584384

    var promise = FileApi.save(_file, layer_id);

    promise.success(function(result){
        console.log('Função chamada quando ocorre sucesso');
```

```
    });
    promise.error(function(result){
        console.log('Função chamada quando ocorre erro');
    });
    promise.finally(function(result){
        console.log('Função chamada sempre');
    });
});
```

Fragmento de código – 18

O fragmento 19 contém o JSON de retorno que a função `success` recebe como retorno, fornecendo um objeto javascript com as informações do arquivo persistido no servidor.

```
{
    "created": "2015-08-11 01:47:12.876570",
    "deleted": false,
    "id": 5666083260334080,
    "kind": "image",
    "last_update": "2015-08-11 01:47:12.876590",
    "layer_id": 5685265389584384,
    "link": "https://s3-sa-east-
1.amazonaws.com/magicsurface/spider.jpg",
    "name": "spider.jpg",
    "size": 8192
}
```

Fragmento de código – 19

4.2.2. *Get*

Método responsável por capturar dados relacionados a um e apenas um arquivo. Como parâmetro obrigatório tem que ser passado um identificador único do objeto.

No fragmento de código 20 é possível observar a chamada de função para carregar um arquivo, passando um identificador para a função `get` e esta retornando uma promessa que é

atribuída para a variável `promise`. Tal promessa possui três funções de *callback* para auxiliar o cliente executando determinadas funções quando o processo termina.

```
angular.module('MyApp').controller('MyCtrl', function(LayerApi){

    var promise = FileApi.get(5666083260334080);

    promise.success(function(result){
        console.log('Função chamada quando ocorre sucesso');
    });
    promise.error(function(result){
        console.log('Função chamada quando ocorre erro');
    });
    promise.finally(function(result){
        console.log('Função chamada sempre');
    });
});
```

Fragmento de código - 20

No fragmento 21 possui o JSON de retornado que a função `success` recebe como resposta do servidor, onde há uma chave no objeto javascript denominada `file` que possui um objeto com as informações do arquivo solicitado.

```
{
    "file": {
        "created": "2015-08-11 01:47:12.876570",
        "deleted": false,
        "id": 5666083260334080,
        "kind": "image",
        "last_update": "2015-08-11 01:47:12.876590",
        "layer_id": 5685265389584384,
        "link":                                  "https://s3-sa-east-
1.amazonaws.com/magicsurface/spider.jpg",
```

```
        "name": "spider.jpg",
        "size": 8192
    }
}
```

Fragmento de código - 21

4.2.3. *List*

Método responsável por listar dados dos arquivos, diferentemente dos outros métodos esse não exige parâmetro obrigatório.

O fragmento de código 22 apresenta a chamada de função para listar os arquivos do servidor baseado em filtros. Nele é possível observar que os arquivos serão filtrados pelos atributos `files` e `deleted` com os valores respectivos `image` e `true`, para que seja retornado apenas imagens não removidas.

```
angular.module('MyApp').controller('MyCtrl', function(LayerApi){

     var form = {
        layerId: 5685265389584384,
        filters: {
            files:'image',
            deleted: false
        }
    };

    var promise = FileApi.list(form);

    promise.success(function(result){
        console.log('Função chamada quando ocorre sucesso');
    });
    promise.error(function(result){
        console.log('Função chamada quando ocorre erro');
    });
    promise.finally(function(result){
        console.log('Função chamada sempre');
```

```
    });
});
```

Fragmento de código – 22

O JSON apresentado no fragmento de código 23 mostra o formato de retorno de sucesso da função, onde há uma chave denominada `files` contendo uma lista de objetos javascript com informações de todas os arquivos processados no lado do servidor.

```
{
    "files": [
        {
            "created": "2015-08-11 01:47:12.876570",
            "deleted": false,
            "id": 5666083260334080,
            "kind": "image",
            "last_update": "2015-08-11 01:47:12.876590",
            "layer_id": 5685265389584384,
            "link":                              "https://s3-sa-east-
1.amazonaws.com/magicsurface/spider.jpg",
            "name": "spider.jpg",
            "size": 8192
        }
    ]
}
```

Fragmento de código – 23

4.2.4. *Delete*

Método responsável por excluir um arquivo, como parâmetro obrigatório deve ser passado um identificador único do objeto arquivo.

O fragmento de código 24 demonstra a chamada de função para excluir um arquivo salvo pela API, onde o método `del` recebe um identificador e retorna uma promessa para o tratamento das respostas assíncronas efetuadas pela requisição Ajax.

```
angular.module('MyApp').controller('MyCtrl', function(LayerApi){

    var promise = FileApi.del(5666083260334080);

    promise.success(function(result){
        console.log('Função chamada quando ocorre sucesso');
    });
    promise.error(function(result){
        console.log('Função chamada quando ocorre erro');
    });
    promise.finally(function(result){
        console.log('Função chamada sempre');
    });
});
```

Fragmento de código - 24

O JSON apresentado pelo fragmento 25 mostra como o dado chega até a função `success`, permitindo que o usuário notifique a tela e retire ou informe ao usuário sobre o *status* do dado usado para a ação.

```
{
    "files": [
        {
            "created": "2015-08-11 01:47:12.876570",
            "deleted": false,
            "id": 5666083260334080,
            "kind": "image",
            "last_update": "2015-08-11 01:47:12.876590",
            "layer_id": 5685265389584384,
            "link": "https://s3-sa-east-
1.amazonaws.com/magicsurface/spider.jpg",
            "name": "spider.jpg",
            "size": 8192
        }
```

```
    ]
}
```

Fragmento de código - 25

5. Considerações finais

Este capítulo apresenta as contribuições e conclusões deste trabalho, além de apresentar sugestões para trabalhos futuros.

5.1. Contribuições

Hoje em dia, a maior parcela do mundo de tecnologia gira em torno de serviços, é possível observar tal afirmação em diversos projetos tecnológicos. Pode ser tomado como um exemplo a hospedagem de *web sites*, onde o servidor utilizado para alocar tal conteúdo e suportar o fluxo de acesso, geralmente é alocado em uma outra região da cidade, país ou mundo. Tais serviços são planejados para efetuar um trabalho dedicado ao proposto, retirando responsabilidades dos clientes que contratam o serviço.

Para efetuar a comunicação entre cliente e servidor há a necessidade da existência de uma API com padrões de comunicação entre ambas as partes. É possível verificar esse comportamento quando um determinado cliente necessita efetuar um *login* em um *software* qualquer e não quer obrigar um possível usuário efetuar um novo cadastro, para facilitar tal ação uma integração com um serviço esterno é pode ser efetuada, pode ser tomado como exemplo o Google ou Facebook, onde um sistema qualquer pode efetuar operações no servidor dessas empresas a partir de uma outra aplicação.

Há diversos sistemas que disponibilizam uma API de comunicação, para serviços que vão desde servidores até consumo de informação de bases super populadas. O MagicSurface oferece um serviço que agiliza o desenvolvimento de aplicativos de realidade aumentada, onde o cliente não precisa configurar um servidor de dados e arquivos, e sim consumir o serviço utilizando o protocolo HTTP e gastar o máximo possível de seu tempo na parte principal de sua aplicação.

Como disponibilizam um SDK para facilitar a comunicação do cliente com o servidor, o MagicSurface disponibiliza um *framework* javascript para disponibilizar essa facilidade para todos os clientes do serviço.

5.1.1. Publicações

Primariamente, foi publicado o seguinte artigo em periódico científico como resultado deste trabalho::

BUSTAMANTE, M. V. C. ; FARIA, G. G. ; ALVES, M. V. A. ; BERTOTI, Giuliano Araujo . **APLICAÇÃO DE REALIDADE AUMENTADA PARA DISPOSITIVOS MÓVEIS**. Boletim Técnico da Faculdade de Tecnologia de São Paulo, v. 40, p. 101, 2015.

Secundariamente, durante os estudos e pesquisas relacionados à realidade aumentada e o uso de APIs, foram elaborados outros dois projetos que também obtiveram artigos científicos publicados:

ALVES, M. V. A. ; FARIA, G. G. ; BUSTAMANTE, M. V. C. ; BERTOTI, Giuliano Araujo . **MAGIC POKER: JOGO PARA DISPOSITIVOS MÓVEIS UTILIZANDO INTERFACES NATURAIS**. Boletim Técnico da Faculdade de Tecnologia de São Paulo, v. 40, p. 112, 2015.

ALVES, M. V. A. ; BUSTAMANTE, M. V. C. ; MARIZ, R. N. ; BERTOTI, Giuliano Araujo . **APLICAÇÃO DE COMÉRCIO ELETRÔNICO PARA DISPOSITIVOS MÓVEIS**. Boletim Técnico da Faculdade de Tecnologia de São Paulo, v. 38, p. 97, 2014.

5.2. Trabalhos futuros

Este Trabalho não encerra o desenvolvimento em busca de aperfeiçoar o tempo de desenvolvimento de aplicações de realidade aumentada e no rico agrupamento de métricas sobre os dados persistidos, mas abre oportunidades para os seguintes trabalhos futuros:

- Criação de um *framework* com javascript puro, eliminando a dependência do AngularJs.
- Criação de um *dashboard* rico em métricas sobre os dados persistidos no *WebService*, possibilitando uma tomada de decisão mais precisa.
- Expandir os tipos de dados que a API permite, inserindo polígonos no mapa, camadas tridimensionais, persistência de vetores, entre outros.
- Integrar com redes sociais para possibilitar o compartilhamento de dados ao serem salvos, expandindo o conhecimento sobre a localização de conteúdos.
- Criar uma aplicação educacional com realidade aumentada, com o intuito de tornar o aprendizado mais dinâmico.
- Criar uma aplicação com dados públicos para que ao apontar um dispositivo para um rio ou região de uma cidade, diversas informações úteis sejam mostradas para o usuário no mundo real.
- Estudar a eficiência dos algoritmos em relação ao compartilhamento de dados em tempo real, ou seja, tempo necessário entre salvar um conjunto de dados e a distribuição dos mesmos para os clientes consumidores.

- Criar uma aplicação que eternize os momentos vividos pelas pessoas em um dado espaço geográfico do mapa mundial, tornando possível ao se apontar um dispositivo móvel para um local rever tudo o que foi vivido ali.

Referências

Angularjs.org. **AngularJS - Superheroic JavaScript MVW Framework.** Disponívels em https://angularjs.org/. Acessado em: 03/08/2015

AZUMA, Ronald T. et al. **A survey of augmented reality**. Presence, v. 6, n. 4, p. 355-385, 1997.

AZUMA, Ronald et al. **Recent advances in augmented reality. Computer Graphics and Applications**, IEEE, v. 21, n. 6, p. 34-47, 2001.

BRAY, Tim. The JavaScript Object Notation (JSON) Data Interchange Format. 2014.

DOMINGUES, Diana; VENTURELLI, **Suzete. Cibercomunicação cíbrida no continuum virtualidade aumentada e realidade aumentada: era uma vez... a realidade.** ARS (São Paulo), v. 5, n. 10, p. 108-121, 2007.

FEINER, Steven et al. **A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment**. Personal Technologies, v. 1, n. 4, p. 208-217, 1997.

FEINER, Steven; MACINTYRE, Blair; SELIGMANN, Dorée. Knowledge-based augmented reality. **Communications of the ACM**, v. 36, n. 7, p. 53-62, 1993.

FIELDING, Roy et al. **Hypertext transfer protocol--HTTP/1.1**. 1999.

FIELDING, Roy Thomas. **Architectural styles and the design of network-based software architectures**. 2000. Tese de Doutorado. University of California, Irvine.

FLANAGAN, David. **JavaScript: the definitive guide**. " O'Reilly Media, Inc.", 2006.

Gruntjs.com. **Grunt: The JavaScript Task Runner**. Disponível em http://gruntjs.com/. Acessado em 03/08/2015.

Highlightjs.org. **highlight.js**. Disponível em https://highlightjs.org/. Acesso em: 03/08/2015

ISRAEL, Karina Pinheiro. Informação e Tecnologia nos Museus Interativos do Contemporâneo. **Biblioteca Latino-Americana de Cultura e Comunicação**, v. 1, n. 1, 2012.

KASAHARA, Shunichi et al. Second surface: multi-user spatial collaboration system based on augmented reality. In: **SIGGRAPH Asia 2012 Emerging Technologies**. ACM, 2012. p. 20.

LIN, Zhijie et al. Research on web applications using ajax new technologies. In: **MultiMedia and Information Technology, 2008. MMIT'08. International Conference on**. IEEE, 2008. p. 139-142.

Mark Otto, a. **Bootstrap - The world's most popular mobile-first and responsive front-end framework**. Disponível em http://getbootstrap.com/. Acesso em: 03/08/2015

MILGRAM, Paul et al. Augmented reality: A class of displays on the reality-virtuality continuum. In: **Photonics for Industrial Applications**. International Society for Optics and Photonics, 1995. p. 282-292.

MORES, Carlos A.; CLARKE, A. C. No mês de março de 2008.

NGOLO, Márcio António Fernandes. **Arquitectura orientada a serviços REST para laboratórios remotos**. 2009.

PAUL, Perrine; FLEIG, Oliver; JANNIN, Pierre. Augmented virtuality based on stereoscopic reconstruction in multimodal image-guided neurosurgery: Methods and performance evaluation. **Medical Imaging, IEEE Transactions on**, v. 24, n. 11, p. 1500-1511, 2005.

REGENBRECHT, Holger et al. An augmented virtuality approach to 3D videoconferencing. In: **Proceedings of the 2nd IEEE/ACM international Symposium on Mixed and Augmented Reality**. IEEE Computer Society, 2003. p. 290.

TORI, Romero; KIRNER, Claudio; SISCOUTTO, Robson Augusto. **Fundamentos e tecnologia de realidade virtual e aumentada**. Editora SBC, 2006.

TWITTER. **API console Tool**. Disponível em https://dev.twitter.com/rest/reference/post/lists/create. Acesso em: 02/08/2015.

VAN KESTEREN, Anne et al. Cross-origin resource sharing. **W3C Working Draft WD-cors-20100727**, 2010.