FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

IPBrick - API para integração de aplicações “third party”

João Lopes Ferreira Sobreira

PREPARAÇÃO DA DISSERTAÇÃO

PREPARAÇÃO DA DISSERTAÇÃO

Orientador: João Manuel Couto das Neves

Co-orientador: Miguel Ramalhão Ribeiro

16 de Fevereiro de 2014

© João Sobreira, 2014

Resumo

Este relatório foi elaborado no âmbito da unidade curricular Preparação da Dissertação e contem uma apresentação geral da dissertação a ser desenvolvida no ano curricular 2013/2014. Incluída no projeto IPBrick da empresa *iPortalMais*, esta dissertação visa a criação de ferramentas através de uma Application Programming Interface (API) pública para integração de aplicações ”third party“ na plataforma de forma independente. Neste relatório são apresentadas as condições atuais sobre a qual vai ser criada e integrada a API e uma análise à manutenção da solução original baseada no protocolo Simple Object Access Protocol (SOAP) com definição de serviços em Web Service Description Language (WSDL) e à adaptação de uma arquitetura mais recente baseada em Representational State Transfer (REST) com mensagens transmitidas em JavaScript Object Notation (JSON). No final são analisadas as ferramentas disponiveis para elaboração e desenvolvimento do projeto, identificando as suas características e potencialidades.

Abstract

This report was elaborated in the context of the subject Preparação da Dissertação and contains a brief presentation of the thesis being developped in the curricular year 2013/2014. Included in the project IPBrick from *iPortalMais*, this thesis aims to create tools to integrate third-party applications independently via a published API. In this report it is presented the base in which the API will be developped and integrated and an analysis on the maintenance of the actual soluction based on the protocol SOAP with services being described by WSDL and the change to the architecture REST with JSON messages. Lastly, an analysis of the tools available for development of this project is made, identifying their features.

Conteúdo

1 Introdução **1**
1.1 Motivação . . . 1
1.2 Estrutura do Relatório . . . 1

2 Enquadramento **3**
2.1 IPBrick . . . 3
2.1.1 IPBrick.I . . . 3
2.1.2 IPBrick.C . . . 4
2.2 Aplicações “third-party” . . . 4
2.3 Application Programming Interface . . . 5
2.4 Objetivos . . . 5
2.5 Análise de Requisitos . . . 6

3 Estado da Arte **7**
3.1 Formatos de Mensagens . . . 7
3.1.1 XML . . . 9
3.1.2 JSON . . . 9
3.1.3 XML vs JSON . . . 9
3.2 Arquitetura . . . 10
3.2.1 SOA e Web Services . . . 10
3.2.1.1 SOAP . . . 13
3.2.1.2 REST . . . 15
3.3 API IPBrick . . . 17

4 Tecnologias **19**
4.1 SOAP UI . . . 19
4.2 Pacotes Debian . . . 19
4.3 PHP . . . 21
4.3.1 SimpleXML . . . 21
4.3.2 Recess . . . 21

5 Trabalho Efetuado e Planeado **23**
5.1 Trabalho Efetuado . . . 23
5.2 Planeamento . . . 23

A Exemplo de um ficheiro WSDL **25**

Referências **27**

Lista de Figuras

3.1 Distribuição de formatos de mensagens em aplicações 8
3.2 Estrutura de uma aplicação SOA 11
3.3 Distribuição de estilos de aplicações 12
3.4 Estrutura de uma comunicação na arquitetura REST [15] 15

4.1 Software SOAP UI 19
4.2 Estrutura de um pacote debian 20
4.3 Ficheiros de um pacote debian 20
4.4 Ficheiro *control* 21

5.1 Diagrama GANTT 24

Lista de Tabelas

2.1 Tabela de Requisitos 6

3.1 Número de aplicações por formato de mensagem 8
3.2 Vantagens e Desvantagens de XML 10
3.3 Vantagens e Desvantagens de JSON 10
3.4 Número de aplicações por estilo 12
3.5 Elementos de um ficheiro WSDL 14

4.1 Estrutura de um pacote Debian 20

Abreviaturas e Símbolos

AMF Action Message Format

API Application Programming Interface

CNAME Canonical Name

CPU Central Processing Unit

CSV Comma-Separated Values

DHCP Dynamic Host Configuration Protocol

DNS Dynamic Network Services

FTP File Transfer Protocol

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

IDS Intrusion Detection System

IM Instant Messaging

JMS Java Message Service

JSON JavaScript Object Notation

KML Keyhole Markup Language

LDAP Lightweight Directory Access Protocol

MX Mail Exchange

PBX Private Branch Exchange

PHP PHP Hypertext Preprocessor

RDF Resource Description Framework

REST Representational State Transfer

RPC Remote Procedure Call

RSS Rich Site Summary

SMS Short Message Service

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

UCoIP Unified Communications over IP

URI Uniform Resource Identifier

VoIP Voice Over IP

WSDL Web Service Description Language

XML Extensible Markup Language

XSD XML Schema

Capítulo 1

Introdução

1.1 Motivação

A conhecida expressão de Sir Isaac Newton “If I have seen a little further it is by standing on the shoulders of Giants."em 1676 é suficiente para resumir toda a evolução humana, a realidade de que os avanços são apenas possível pois outros antes de nós avançaram até onde nós começámos. Mas não foi o famoso físico e matemático o primeiro a utilizar esta analogia para referenciar a evolução do conhecimento. João de Sallsbúria em 1159 num excerto do seu livro Metalogicon escreve “Bernard of Chartres used to say that we are like dwarfs on the shoulders of giants, so that we can see more than them, and things at a greater distance (...) because we are carried high and raised up by their giant size.”
O conhecimento do homem é atualmente tão vasto e complexo que não poderia evoluir de outra maneira, daí surgir a necessidade de estruturar as bases sobre as quais trabalhamos, as camadas do desenvolvimento. Nas tecnologias da informação essa necessidade é ainda mais imperial pois tanto as máquinas como as redes de máquinas já se encontram estruturadas em camadas fixas. É então importante, para o melhor funcionamento, providenciar o mais eficaz e mais adequado método de comunicação inter-camadas.

1.2 Estrutura do Relatório

Para além da introdução, este relatório contém mais 4 capítulos. No capítulo 2 é apesentado e contextualizado o problema da dissertação e são definidos objetivos para o projeto. No capítulo 3, é descrito o estado da arte e a teoria necessária para desenvolvimento do projeto. No capítulo 4 são apresentadas e analisadas as tecnologias escolhidas para elaboração do trabalho. No capítulo 5 é descrito o trabalho já efetuado e o planeamento para o trabalho futuro.

Capítulo 2

Enquadramento

2.1 IPBrick

O sistema IPBrick é uma solução tecnológica integrada de comunicações, segurança e armazenamento de dados entre outros serviços, de fácil e rápida implementação, criada em Portugal e baseada em software open-source Linux. Graças à existência de uma interface gráfica web é possível efetuar a gestão de uma rede e dos seus servidores sem conhecimentos avançados das tecnologias utilizadas.

A plataforma é pioneira no conceito Unified Communications over IP (UCoIP) fornecendo a comodidade da utilização de uma vasta gama de serviços de comunicação em rede (e-mail, Voice Over IP (VoIP), Short Message Service (SMS), Instant Messaging (IM)) com apenas um único endereço. É estruturada em módulos, com cada módulo focado num aspecto da dinâmica de uma rede descritos nas secções 2.1.1 e 2.1.2.

2.1.1 IPBrick.I

A IPBrick.I é o módulo IPBrick que fornece ferramentas de gestão de redes intranet e dos seus servidores tais como servidor de email, de ficheiros, domínio, impressão, fax e de backup. Pode ser utilizado em três modos: Master, Slave e Client.

IPBrick Master Modo default em que todos os serviços usam o servidor Lightweight Directory Access Protocol (LDAP);

IPBrick Slave O servidor Slave deverá manter uma cópia sincronizada do servidor Master indicado. Permite a autenticação mas não a gestão do LDAP.

IPBrick Client A autenticação neste servidor é feita remotamente, no servidor LDAP, não existindo nenhuma informação guardada localmente. É comum a utilização deste modo em máquinas de relay como proxies e firewalls.

Fornece o seguinte conjunto de serviços:

Servidor de áreas de trabalho individuais e de grupo

Servidor LDAP (gestão de máquinas, utilizadores e grupos)

Servidor de imagens de estações de trabalho
Servidor de Domínio (com roaming-profiles e centralized scripting)
Servidor de Dynamic Host Configuration Protocol (DHCP)
Servidor de Dynamic Network Services (DNS)
Servidor de correio electrónico
Serviço de Anti-SPAM
Serviço de Anti-vírus para correio-electrónico e área de trabalho
Servidor de impressoras
Servidor de autenticação centralizada
Servidor de base de dados
Servidor de Groupware (Calendário e Livro de Endereços)
Servidor de Backup

Este módulo também permite a incorporação de aplicações third party como o sistema de gestão documental iPortalDoc, o software de backup Bacula, o servidor de CRM SugarCRM e o sistema de monitorização de redes Nagios entre outros.

2.1.2 IPBrick.C

Com o módulo IPBrick.C é possível equipar uma rede com mecanismos de comunicação e gestão de fluxo de e para a Internet como o serviço de e-mail, VoIP, proxy e firewall.

Fornece o seguinte conjunto de serviços:

Relay de correio electrónico
Webmail
Servidor web
Servidor Proxy (Hypertext Transfer Protocol (HTTP), HyperText Transfer Protocol Secure (HTTPS), File Transfer Protocol (FTP) com estatísticas)
Traffic shaping
Segurança baseada em pacotes
Intrusion Detection System (IDS)
Servidor de VoIP
Integração transparente com Private Branch Exchange (PBX) (ISDN E1/BRI e linhas analógicas)
Serviço Mail 2 SMS

2.2 Aplicações “third-party”

Designa-se por aplicação “third-party” o software distribuído por outra entidade que não o da(s) plataforma(s) a que se destina. Podem consistir em programas completos ou apenas plugins e extensões que adicionam funcionalidades a outro programa. Sendo o sistema IPBrick um sistema

baseado em Debian, a instalação de aplicação no mesmo procede-se com o uso de pacotes Debian. De maneira a compreender a estrutura de uma aplicação desenvolvida para IPBrick é preciso analisar a estrutura de um pacote deste tipo. Esta análise é feita na secção 4.2.

2.3 Application Programming Interface

Uma API é uma interface que define o modo como duas aplicações comunicam entre si reunindo um conjunto de instruções e padrões para o seu acesso.

A criação de uma API por parte de uma aplicação permite assim a integração de recursos e funcionalidades da mesma por uma outra sem necessidade do envolvimento da segunda nos detalhes da sua implementação, razão pela qual tem sido uma solução adotada em muitos casos quer pelas aplicações que beneficiam por desenvolvimento externo de aplicações com as suas funcionalidades, quer pela aplicações externas que acrescentam funcionalidades sem necessidade de as desenvolver.

É, na prática, criada na forma de uma ou mais bibliotecas com especificações de rotinas, estruturas de dados, classes de objetos e variáveis podendo também ser apresentada, no caso de APIs desenhadas para serviços web, como uma definição de acessos e pedidos remotos.

Sendo em programação, por definição, uma interface, ajuda a definir propriedades, métodos e eventos que as classes são capazes de implementar sendo um meio de objetos não relacionados comunicarem, um "contrato" de parâmetros que ambos devem implementar para que possa haver comunicação. Podem então descrever, sucintamente:

1. As mensagens que são compreendidas pelo objeto;
2. Os argumentos que podem ser enviados nas mensagens;
3. O tipo de resultados que as mensagens retornam.

No estado da arte serão aprofundadas duas partes que compõem uma API e que as destinguem na sua utilidade. Uma análise sobre o formato das mensagens é feita na secção 3.1 e um estudo sobre a arquitetura e soluções existentes é efetuado na secção 3.2.

2.4 Objetivos

É a intenção deste projecto a revisão e extensão da API existente no sistema IPBrick no que toca a integração de aplicações "third-party". Tendo em consideração a existência de uma API elaborada mas não completa, uma revisão ajudará a identificar a base sobre a qual as funcionalidades previstas poderão ser desenvolvidas como extensão da API atual. Atualmente a integração de uma aplicação "third-party" no sistema é feita pela equipa IPBrick isto porque os pacotes precisam de incluir comandos de acesso direto ao sistema. Esta situação levanta dois problemas:

1. Nenhuma aplicação destinada ao sistema IPBrick pode ser desenvolvida isoladamente por uma entidade externa;

RNF00	Estabelecimento de mensagens de erro
RNF01	Execução de queries com possibilidade de *commit/rollback*
RNF02	Desenvolvimento em versões PHP posteriores a 5.4
RNF10	Obtenção de variáveis da aplicação
RNF11	Verificação de dependências e conflitos
RNF12	Verificação de versões IPBrick
RNF13	Verificação de dupla instalação da aplicação
RNF14	Possibilidade de atualização da aplicação
RNF15	Criação de base-de-dados temporária para alojamento de configurações
RNF16	Identificação de instalação através da interface web para permissão de administrador
RF20	Criação de *Virtual Host*
RF21	Criação de *alias Virtual Host*
RF22	Criação de registo CNAME no servidor DNS
RF23	Criação de registo MX no servidor DNS
RF24	Configurações adicionais de e-mail
RF25	Configurações adicionais de LDAP
RF26	Configurações adicionais de firewall
RNF30	Criação de entrada na lista de *bugfix*
RNF31	Apresentação de instalação bem-sucedida

Tabela 2.1: Tabela de Requisitos

2. Decorre uma ocupação de recursos por parte da IPBrick no desenvolvimento da integração da aplicação na plataforma.

Surge então a necessidade do desenvolvimento de ferramentas para que entidades externas o possam fazer individualmente com certeza de que existe compatibilidade, comunicação e integração do seu produto e assegurando o acesso seguro e controlado ao sistema.

2.5 Análise de Requisitos

Na tabela 2.1 são apresentados os requisitos necessários para o sistema atual.

Na gama RNF0x são apresentados requisitos de implementação gerais e boas práticas para desenvolvimento do código. Na gama RNF1x estão listados requisitos do sistema para preparação durante a pré-instalação da aplicação. A gama RF2x refere-se a ferramentas disponiveis para aplicar configurações no sistema IPBrick. Por fim, a gama RNF3x identifica requisitos para a finalização da instalação da aplicação.

Capítulo 3

Estado da Arte

Dada a implementação de serviços no sistema IPBrick proceder-se por meio de serviços web é relevante e adequado projetar a API de acordo com este paradigma. Desta abordagem resulta não só uma ferramenta mais adequada mas também mais generalista à qual se poderá, posteriormente, extender o uso. É portanto importante tem em conta dois pontos importantes, abordados por Nick Gall [1]

- Neutralidade de aplicação
- Neutralidade de implementação

Segundo a analogia de Nick Gall [1], é conveniente olhar para a arquitetura de uma API como uma ampulheta. Por um lado, é necessário desenvolver uma estrutura neutra ao nível da aplicação, podendo abranger o máximo de entidades possíveis. A este ponto Gall refere-se como o topo da ampulheta pois permite uma maior abrangência, maior quantidade de recursos. Por outro, é também necessário ter em conta a neutralidade de implementação, uma abordagem flexível para todos os tipos de tecnologias disponíveis e meios de comunicação, sendo este o fundo da ampulheta de Gall. Porém, embora um protocolo de aplicação mais genérico exerça um efeito maior na rede e deva ser o foco principal, um equilíbrio entre as duas partes trará um maior beneficio do que a implementação baseada em apenas uma.

No que toca à estrutura de uma API, podemos dividi-la em duas partes fundamentais:

- o formato das mensagens na transferência;
- a arquitetura

Tendo como base os pontos referidos anteriormente analisemos então os tipos de fomatos de comunicação e arquiteturas disponíveis atualmente.

3.1 Formatos de Mensagens

Durante muitos anos, Extensible Markup Language (XML) foi o standard e considerado por muitos a solução para o problema de serialização na transferência de dados estruturados entre

aplicações, eliminando a necessidade de parsers especializados para cada formato de dados criado por uma entidade. No entanto, com o passar do tempo, novos formatos foram aparecendo que diziam resolver os problemas que o XML tinha.

Na tabela 3.1 é possível analisar o número de interfaces que utilizam ou utilizaram um dado formato de dados, com valores de Janeiro de 2014, segundo [2].

Formato	**Número de interfaces**	**Percentagem de interfaces**
XML	6006	48,69
JSON	5117	41,48
RSS	242	1,96
HTML	236	1,91
CSV	227	1,84
PHP	152	1,23
Text	138	1,12
RDF	74	0,60
YAML	63	0,51
KML	57	0,46
Outros	24	0,19
Total	12336	100

Tabela 3.1: Número de aplicações por formato de mensagem

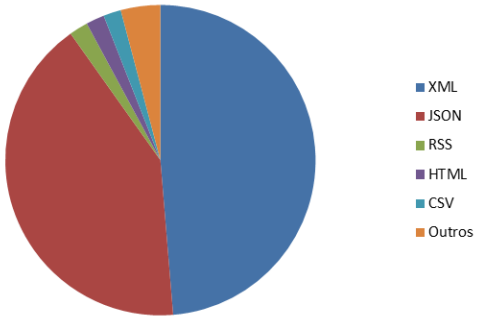

Figura 3.1: Distribuição de formatos de mensagens em aplicações

Como é possível verificar, dois valores destacam-se dos outros, sendo um XML (como referido anteriormente) e o outro JSON, um formato apresentado 3 anos após o primeiro ter sido considerado um recomendação W3C, cuja popularidade tem vindo a crescer. Emboras todos os formatos sejam válidos de serem usados, é necessário primeiro ter em conta a natureza de alguns deles.

- Resource Description Framework (RDF), Keyhole Markup Language (KML), HyperText Markup Language (HTML) e Rich Site Summary (RSS) são formatos baseados em XML;
- Os formatos Comma-Separated Values (CSV) e Text são demasiado simples para uma implementação deste género;
- RSS está desenhado para outro tipo de funcionalidades (alertas);

- PHP Hypertext Preprocessor (PHP) não é flexível o suficiente visto não ser desenhado para o efeito.

Tendo em conta as estatísticas, o suporte e ferramentas fornecidas e as observações indicadas, optou-se pela análise mais aprofundada de XML e JSON.

3.1.1 XML

XML (eXtended Markup Language) é uma linguagem de representação universal de dados com sintaxe baseada na utilização de marcadores estruturados em forma de árvore. Esta componente permite não só descrever o conteúdo de um dado objeto ou informação mas também a sua estrutura, garantindo assim uma fácil e rápida interpretação quer por humanos, quer por máquinas. É usada maioritariamente em RPCs e serialização de objetos.

Foi criado para homogenizar o formato dos dados na transferência dos mesmos entre aplicações e adotado, desde aí, como uma das principais soluções neste assunto e base de muitas outras novas linguagens.

A capacidade de utilização de esquemas e *namespaces* garante ao XML um carácter flexível, aberto, genérico e versátil, permitindo adaptação aos mais variados contextos.

Contudo, a quantidade de funcionalidades disponíveis e a sua versatilidade tem custo de largura de banda, armazenamento e poder de processamento [3]. É na tentativa de resolver estas questões que surge a linguagem JSON.

3.1.2 JSON

JSON (JavaScript Object Notation) define um formato de dados leve para transferência de objetos baseado em pares atributo-valor. Na sua sintaxe constam 4 tipos primitivos (*strings*, inteiros, *booleans* e *null*) e 2 tipos estruturados (objetos e vetores). [4] Foi desenhado com o intuito de, além de ser independente da linguagem com a qual se relaciona e ser fácil de gerar, compreender e interpretar, também aproveitar as capacidades standard dos navegadores e da ligação duplex de uma aplicação a um servidor Web.

Ao analisar código em JSON é intuitivo associar a sua aparência, estrutura e sintaxe a muitas linguagens de programação disponíveis, razão pela qual foi facilmente adotada como uma alternativa ao XML por parte de diversos programadores que viram no novo formato um modo mais natural e simples de definir dados.

3.1.3 XML vs JSON

Debates com este tema são recurrentes sem existir um consenso das partes integrantes. É, contudo, possível de se afirmar que ambos os formatos são válidos, possuem pontos fortes e pontos fracos e devem ser escolhidos consoante a situação.

Nas tabelas 3.2 e 3.3 são apresentadas algumas vantagens e desvantagens analisadas de cada linguagem tendo em conta um comparativo.

XML	
Vantagens	**Desvantagens**
Fácil leitura	Excesso de elementos (ex: marcadores de fecho)
Maior flexibilidade	Demasiado liberal tornando-se fácil de complicar
Utilização de *namespaces* evita colisões de nomes	Mais lento na transferência (mais símbolos)
Esquemas fornecem validação de tipos de dados e estruturas	Necessidade de manipulação de dados na receção

Tabela 3.2: Vantagens e Desvantagens de XML

JSON	
Vantagens	**Desvantagens**
Sintaxe mais simples	Menor flexibilidade efeito da simplicidade da sintaxe
Menor *overhead*	Validação de entrada não suportada por defeito
Integração natural com algumas linguagens, nomeadamente Javascript	
Interpretação mais rápida no recetor	

Tabela 3.3: Vantagens e Desvantagens de JSON

3.2 Arquitetura

3.2.1 SOA e Web Services

Service Oriented Architecture (SOA) é um estilo de arquitetura de software baseado na apresentação de funcionalidades a outras aplicações por meio de serviços.

Um serviço consiste numa ação executada por um prestador de serviços a pedido de um consumidor. Esta é uma metodologia que permite, do ponto de vista logístico, a separação de responsabilidades em diferentes serviços que comunicam entre si podendo ser programados em linguagens e sobre plataformas distintas.

O principal atributo que define um software baseado nesta arquitetura é o pouco ou nenhum conhecimento acerca do funcionamento de cada um dos componentes que providenciam cada serviço, apenas das funcionalidades que oferecem por meio de uma interface pública. É possível comparar um componente com uma caixa-preta, encapsulando o serviço que presta, tornado-o apenas assessível através da interface correspondente [5]. Uma representação visual da estruturação de um serviço deste género pode ser visto na figura 3.2 [6]

Aplicações desenhadas com este propósito existiram muito antes da arquitetura SOA ter sido definida mas só posteriormente foram concebidos e utilizados conjuntos de tecnologias que se tornaram padrões em aplicações deste género, sendo as mais comuns o protocolo baseado em

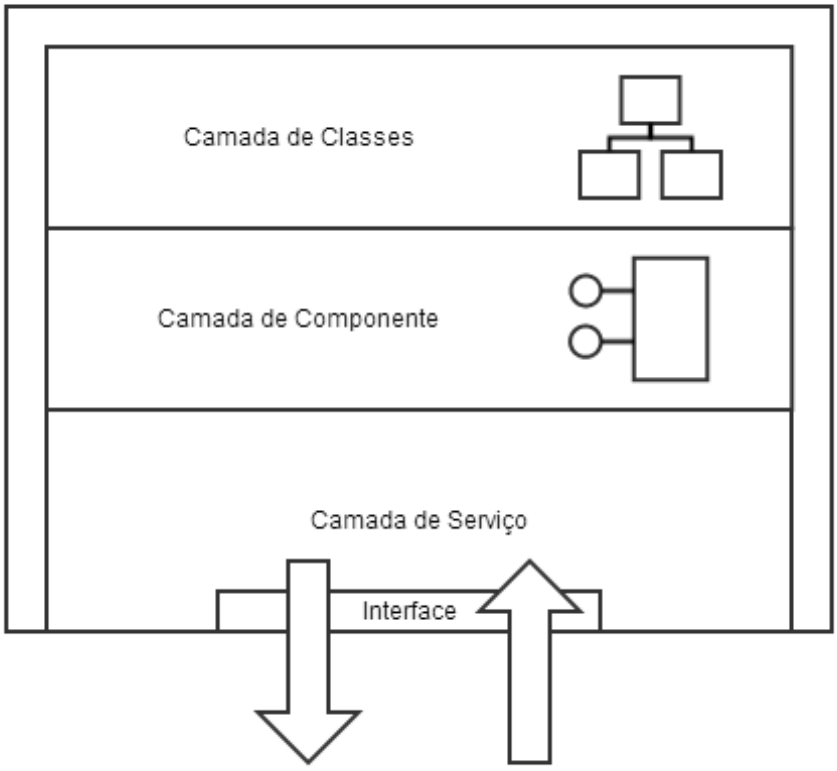

Figura 3.2: Estrutura de uma aplicação SOA

XML e HTTP SOAP, a arquitetura baseada em recursos REST, o esquema de XML XML Schema (XSD) e a norma de desrição de serviços WSDL entre outras. Estas tecnologias serão analisadas com mais detalhe posteriormente.

As características desta arquitetura levam-na a ser uma solução eficaz no planeamento de aplicações hoje em dia. É do conhecimento comum que vivemos num mundo em constante mudança e é, por isso, necessária uma constante adaptação a essa nova mudança, modificando os recursos que ontem eram adequados mas que hoje já não o são. A arquitetura SOA garante a flexibilidade necessária para alterações rápidas e eficazes como resposta a fatores internos (reestruturações, aquisições e redução de custos) ou externos (exigência de clientes e competitividade).

A sua estruturação em componentes também permite quer um investimento mais ponderado por serviços baseados em recursos existentes quer um desenvolvimento mais rápido e segmentado apenas desses serviços. O seu foco mais direccionado para a especificação do que para a integração de um componente permite também a re-utilização mais eficaz de serviços com menos duplicação de recursos o que se traduz em menos custos.

É no entanto possível identificar alguns aspetos menos positivos desta arquitetura como é o caso da necessidade de uma redefinição e reestruturação total de uma aplicação que não seja mas queira ser baseada em SOA o que leva a, dependendo do tamanho, investimentos monetários e de tempo não viáveis. É também opinião de uma maioria dos programadores que grande parte

das tecnologias em que SOA se baseia não sejam de fácil aprendizagem e especialização, sendo ainda normalmente necessárias mais do que uma para implementação. Por fim, alguns estudos indicam essas tecnologias também são menos eficazes do que outras opções em termos de latência, utilização do Central Processing Unit (CPU) e largura de banda, principalmente as baseadas em XML [7].

Web services são as implementações mais comuns de componentes de aplicações SOA. Como o próprio nome indica, são serviços fornecidos através da rede, entre aplicações, com base em mensagens de pedido e resposta.

Têm como principais atributos a utilização de protocolos *open standard* nomeadamente HTTP e XML, sendo completos, suficientes e passiveis de serem utilizados por outras aplicações [8]. Por serem baseados em SOA, têm de ser desenhados com vista a existir independência da plataforma e da linguagem de desenvolvimento, permitindo interoperabilidade entre soluções de diferentes tipos.

Estilo	**Número de interfaces**	**Percentagem de interfaces**
REST	6888	68,88
SOAP	2130	21,30
Javascript	585	5,85
XML-RPC	200	2,00
HTTP GET/POST	96	0,96
AtomPub	29	0,29
JSON-RPC	28	0,28
GData	20	0,20
RSS	14	0,14
XMPP	8	0,08
OSCAR	2	0,02
Total	10000	100

Tabela 3.4: Número de aplicações por estilo

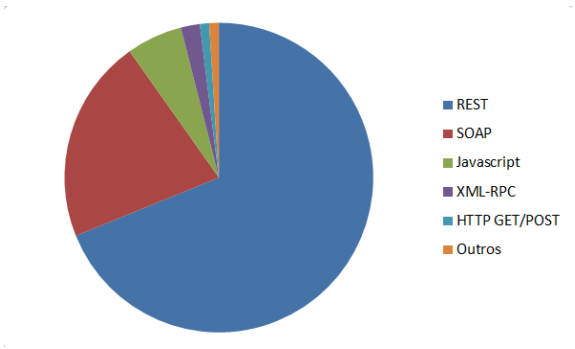

Figura 3.3: Distribuição de estilos de aplicações

Atualmente, como se pode verificar na tabela 3.4 que lista os estilos utilizados em diferentes web services segundo a Y, os estilos mais utilizados de *web services* baseiam-se ou no protocolo SOAP ou na arquitetura REST.

3.2.1.1 SOAP

SOAP (Simple Object Access Protocol) é o protocolo responsável por codificar as mensagens XML trocadas entre as aplicações de maneira a serem compreendidas pelas duas partes, independentemente das especificações de cada uma delas e da rede que as liga.

Este protocolo introduz a necessidade das mensagens serem transportadas num envelope SOAP e do serviço ser descrito segundo a norma WSDL [9], tendo sido a Microsoft a primeira empresa a apostar no seu desenvolvimento com contribuições posteriores de outras companhias como a Ariba, Compaq, HP, IBM e Lotus [10] [11].

Um envelope SOAP é o elemento raíz que encapsula a mensagem, contendo os campos de cabeçalho (opcional) e corpo da mensagem (obrigatório). Cada um destes campos contém sub-elementos nos quais a informação está estruturada. O cabeçalho poderá conter informação relacionada com a aplicação, direccionada a nós intermediários enquanto que o corpo da mensagem contém os dados da transmissão ponto-a-ponto.

Pedido SOAP

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
        <soap:Body xmlns:m="http://www.example.org/stock">
                <m:GetStockPrice>
                        <m:StockName>IBM</m:StockName>
                </m:GetStockPrice>
        </soap:Body>
</soap:Envelope>
```

Types	definição dos tipos de dados com base em esquemas
Message	definição dos dados a serem transmitidos
Operation	definição das funcionalidades do serviço
Port Type	identificação do conjunto de operações integradas nas entidades
Binding	identificação do protocolo e formato de dados utilizado por um Port Type
Port	definição de uma entidade através do binding e do endereço de rede
Service	conjunto de entidades relacionadas

Tabela 3.5: Elementos de um ficheiro WSDL

Resposta SOAP

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
        <soap:Body xmlns:m="http://www.example.org/stock">
                <m:GetStockPriceResponse>
                        <m:Price>34.5</m:Price>
                </m:GetStockPriceResponse>
        </soap:Body>
</soap:Envelope>
```

O fomato das mensagens SOAP assume uma estrutura semelhante ao formato XML, podendo mesmo dizer-se que as primeiras são especificações da segunda. Isto é necessário pois, assegurando que cada entidade participante consegue interpretar e gerar mensagens SOAP é possível assegurar a comunicação e transferência de informação estruturada entre elas.

A norma WSDL é um formato de esquema XML que permite a descrição de um serviço (as suas funções, parâmetros e retornos) para que um cliente possa obter todas as informações acerca do mesmo e gerar e interpretar as mensagens de acordo com os requisitos pedidos. [12] WSDL é versátil o suficiente para permitir a descrição de entidades e as suas mensagens independentemente dos formatos de mensagem ou protocolos de rede usados para comunicar [13]. É garantido assim não só o conhecimento das funcionalidades do serviço mas como a especificação correta e completa dos pedidos. Um ficheiro WSDL é composto por 7 elementos necessários para definir um serviço, apresentados na tabela 3.5 [13].

No anexo A é possível visualizar um exemplo de um ficheiro WSDL.

Existem no total 5 padrões possíveis de mensagens SOAP derivados de pares de estilos e métodos de serialização de dados. Uma mensagem pode ser formatada segundo o estilo Remote

Procedure Call (RPC) ou *documento*. O primeiro começou por ser o mais adotado pela sua simplicidade de compreensão, lógica e facilidade de mapeamento para as tecnologias de computação tradicionais. No entanto a liberdade de estruturação do corpo da mensagem do estilo *documento* em XML nativo e o fraco desempenho e escalabilidade do estilo RPC convenceram os programadores a mudar a sua escolha. Atualmente, na versão 1.2, a utilização do estilo RPC é opcional e o padrão é *documento*.

É também prática comum associar estes dois estilos com dois métodos de serialização de dados: *literal* e *codificado*. O primeiro baseia-se na utilização de esquemas XML padrão pra definir a estruturação do conteúdo enquanto que o segundo utiliza regras SOAP específicas para esta questão. O fato de o método *codificado* não fazer parte dos padrões WS-I (organização responsável por estabelecer boas práticas para interoperabilidade entre web services) levou a uma maior adesão da opção alternativa.

No entanto, uma das críticas apontadas ao padrão *documento/literal* residia na dificuldade de leitura e compreensão de uma mensagem por falta de identificação clara do método e parâmetros utilizados. Em ordem a solucionar este problema, foi criada uma extensão deste padrão ao qual se deu o nome de *documento/literal encapsulado* que adicionava elementos para auxiliar a leitura da mensagem. No entanto, esta adição traduz-se numa maior complexidade de código necessário para a implementar.

3.2.1.2 REST

REST (Representational State Transfer) foi o nome designado por Roy Fielding [14], um dos principais autores da especificação do protocolo HTTP, em 2000 para a arquitetura de *web services* baseada em HTTP e assente num conjunto de conceitos definidos. Esta arquitetura restringe as acções de uma interface a um conjunto de verbos do protocolo HTTP (GET, POST, PUT e DELETE) com a justificação de que estas são as ferramentas suficientes para a construção de um web service cujo acesso se tornou generalizado e que remove a necessidade de implementação de um protocolo extra por cima do protocolo HTTP tornando a sua utilização mais leve e rápida.

Figura 3.4: Estrutura de uma comunicação na arquitetura REST [15]

A proliferação e abundância de utilizadores consumidores de *web services* criou a necessidade de simplificação dos protocolos utilizados. REST associado com JSON surgiu como a solução

mais adequada para o problema. Com requisitos mais baixos de largura de banda e utilização do CPU, grandes empresas prestadoras de serviços através da Internet como a Google, Amazon, Facebook e Twitter, cujo valor de tráfego era elevado, adotaram esta arquitetura nos seus sistemas com vista a aumentar a eficácia do mesmo. Outro acontecimento que despoletou a procura por serviços baseados em REST foi o aparecimento de utilizadores de dispositivos móveis. A limitação na largura de banda e capacidade de processamento do ponto de acesso reenforçou a necessidade de desenvolver *web services* com menos exigências. Uma arquitetura que trata cada pedido de maneira independente como o REST permite também distribuir a carga por servidores de maneira a melhorar o desenpenho, reduzindo e, em alguns casos, eliminando a coordenação entre servidores [Roy Fielding Dissertation]. Alguns casos de utilização de serviços baseados nos príncipios REST incluem o serviço Google Maps, o motor de pesquisa Google e a API do serviço Twitter.

As características desta arquitetura identificadas por Roy Fielding são listadas

1. Estrutura Cliente-Servidor
2. Processamento independente de pedidos
3. *Caching*
4. Sistema estruturado por camadas
5. *Code-on-demand*
6. Interface Uniforme

Roy Fielding [14] começou por definir a primeira restrição a REST como a necessidade de uma estrutura cliente-servidor. Esta estrutura traz os benefícios da separação entre lógica e implementação, aumentando a portabilidade da aplicação e a escalabilidade da mesma, permitindo que os seus componentes possam evoluir de maneira independente.

A segunda característica, como referido anteriormente, além de garantir uma melhor escalabilidade por simplificar a gestão de recursos no decorrer de pedidos [15], também torna o serviço mais fiável e aumenta a capacidade de um terceiro componente poder gerir uma comunicação. No entanto, a existência de informação redundante em cada pedido afeta a eficácia da rede, sendo essa uma das consequências negativas cuja característica 3 procurou atenuar. A identificação da possibilidade do cliente manter um pedido em memória permite a reutilização de informação de pedidos desse tipo no processsamento de pedidos seguintes.

A característica 5 refere-se à necessidade de encapsulamento serviços em camadas, cada camada apenas tendo alcance às camadas adjacentes. Uma implementação deste género promove a simplicidade da aplicação e a independência dos seus componentes, colocando um limite na sua complexidade. No entanto estes sistemas aumentam a latência das comunicações por introdução de fronteiras de processamento algo que pode ser atenuado com utilização de memórias de *caching* em intermediários.

A utilização de *code-on-demand* é uma característica opcional de REST na qual um componente do cliente tem acesso a um conjunto de recursos mas não à maneira como os deveria processar. Neste caso, o cliente executa um pedido a um servidor pelo código que permite manipular os

recursos e executa-os localmente. Esta característica aumenta a simplicidade de implementação de um serviço e de adição de novas funcionalidade ao mesmo.

Por fim, interfaces uniformes permite a simplificação da arquitetura e separação dos serviços da implementação dos mesmos. Em REST, esta característica é conseguida por implementação de 4 restrições às interfaces:

1. Identificação de recursos
2. Manipulação de recursos através de representações
3. Mensagens que se auto-descrevem
4. Mudança de estado da aplicação através de hiperligações

1. Identificação de recursos Um recurso é um entidade (física ou conceptual) providenciada por uma aplicação com um identificador Uniform Resource Identifier (URI). Na arquitetura REST existe a necessidade de definir regras para um identificador sendo elas:

- A semântica do mapeamento de um URI para um recurso não pode mudar
- A identidade de um recurso é independente do seu valor
- O fornecedor de um recurso apenas é responsável por manter a validade da semântica de um URI
- Um URI não deve conter nenhuma referência ao tipo de recurso

2. Manipulação de recursos através de representações

Uma representação de um recurso contém dados indicadores do estado desse mesmo recurso, podendo este ser representado de várias maneiras, consoante o pedido efetuado pelo cliente. Em REST existem dois tipos de estados: o estado do recurso e o estado da aplicação que indica o caminho que o cliente percorreu através da aplicação. O primeiro é mantido no lado do servidor enquanto que o segundo é guardado do lado do cliente.

3. Mensagens auto-descriptivas

Esta condição obriga a que todos os detalhes para compreensão e processamento de uma mensagem sejam incluido na própria, garantindo uma comunicação com processamento independente de mensagens.

4. Mudança de estado da aplicação através de hiperligações

A ideia da arquitetura REST é que seja fornecido ao cliente um conjunto de hiperligações em cada representação para interação com o servidor e mudança do seu estado.

3.3 API IPBrick

A API disponibilizada atualmente pela IPBrick para aplicações thid-party está desenhada para funcionar com base no protocolo SOAP e garante funcionalidades de comunicação necessária para

o funcionamento das aplicações, como é o caso da obtenção de utilizadores VoIP, envio de fax, obtenção de impressoras entre outros.

Aquando da versão 5.3, o sistema garantia a transferência de uma única *string* entre cliente e servidor, sendo que essevalor era então interpretado localmente e dividido nos diferentes parâmetros dos métodos utilizados. Assim a descrição de cada funcionalidade do serviço era apresentada com um único parâmetro de entrada do tipo *string* no ficheiro WSDL. Ora esta abordagem não identificava os valores pretendidos e a ordem dos mesmos (havia necessidade de consulta constante do manual de web service fornecido) razão pela qual acabou por ser abandonada a favor de uma descrição mais detalhada dos parâmetros esperados, obrigando a uma reestruturação total do ficheiro WSDL do sistema.

Ficou à consideração do aluno a integração da API a elaborar na já existente ou a divisão em duas APIs distintas (uma de integração e outra de comunicação) sendo que, não havendo motivos para haver uma separação das duas, foi escolhida a primeira abordagem.

O fato de toda a lógica da API já existente e uma grande parte da lógica do sistema ter sido desenvolvida na linguagem PHP contribuiu para a escolha da mesma linguagem para elaboração deste projeto de modo a garanir uniformidade e homogenidade da plataforma.

Capítulo 4

Tecnologias

4.1 SOAP UI

SOAP UI é uma ferramenta *open-source* que permite executar uma vasta gama de testes funcionais a web services abrangendo as tecnologias SOAP, REST, WSDL, HTTP, Java Message Service (JMS) e Action Message Format (AMF) com uma interface gráfica simples e intuitiva.

Entre as funcionalidades encontram-se testes à segurança, simulação de serviços e teste à sobrecarga do serviço.

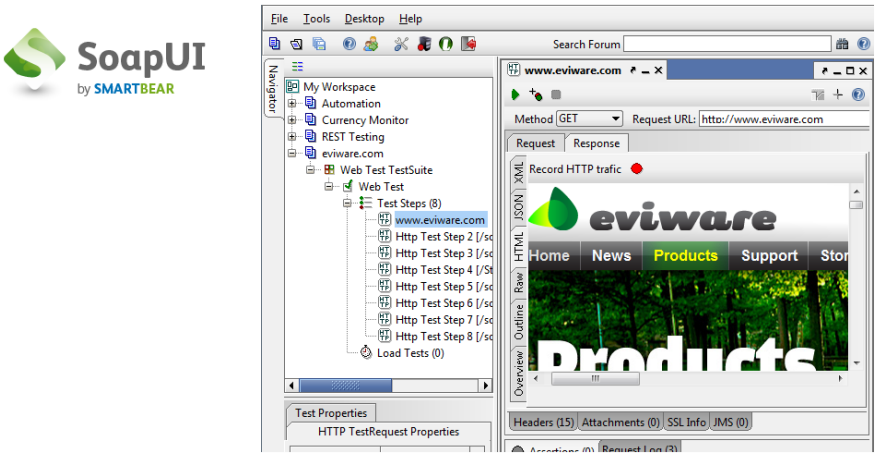

Figura 4.1: Software SOAP UI

4.2 Pacotes Debian

É por meio de pacotes debian que é possível a instalação de software num sistema Unix, sendo que cada pacote compreende 3 partes essenciais representadas na tabela 4.1. Com vista a examinar um pacote debian, foi feito o download de um ficheiro .deb do directório da Apache [1] e utilizadas as ferramentas disponíveis nos sistemas Unix. A figura 4.2 apresenta a estrutura de mais alto nível

[1] https://directory.apache.org/apacheds/download/download-linux-deb.html

debian-binary			identificador da versão do formato .deb do pacote
control.tar.gz			arquivo comprimido com ficheiros de controlo do pacote
	control		contém a informação essencial de controlo
		Source	identificador da origem do pacote
		Maintainer	nome e endereço de e-mail do responsável pelo pacote
		Uploaders	nome e endereço de e-mail de co-responsáveis pelo pacote
		Section	categoria de aplicação do pacote
		Priority	representação da importância para o utilizador da instalação deste pacote
		Build-Depends et al	indicação da dependência de outros pacotes
		Standards-Version	versão mais recente do manual de normas do pacote
	md5sum		*hash* md5 para verificação da integridade do ficheiro
	preinst		*script* para execução antes da extração do pacote
	postinst		*script* para execução após a extração do pacote
	prerm		script para execução antes da remoção dos ficheiros associados com o pacote
	postrm		script para execução após a remoção dos ficheiros associados com o pacote
data.tar.gz			ficheiros da aplicação

Tabela 4.1: Estrutura de um pacote Debian

do pacote em que é possível identificar os ficheiros debian-binary, control.tar.gz e data.tar.gz. Na figura 4.3 estão apresentados os ficheiros extraídos dos ficheiros .tar.gz apresentados na figura 4.2, onde já se pode identificar o ficheiro control, preinst e postinst. Por fim, a figura 4.4 apresenta o conteúdo do ficheiro control, com as informações relativas ao pacote em questão.

```
[redacted]$ ar tv 'apacheds-2.0.0-M15-amd64
.deb'
rw-r--r-- 0/0       4 Aug 14 20:18 2013 debian-binary
rw-r--r-- 0/0    1538 Aug 14 20:18 2013 control.tar.gz
rw-r--r-- 0/0 9837570 Aug 14 20:18 2013 data.tar.gz
```

Figura 4.2: Estrutura de um pacote debian

```
[redacted]$ ls -la
total 40
drwxr-xr-x  5 joao joao  4096 Ago 14  2013 .
drwxr-xr-x 59 joao joao 12288 Fev 15 18:45 ..
-rw-r--r--  1 joao joao   553 Ago 14  2013 control
drwxr-xr-x  3 joao joao  4096 Ago 14  2013 etc
drwxr-xr-x  3 joao joao  4096 Ago 14  2013 opt
-rwxr-xr-x  1 joao joao  1467 Ago 14  2013 postinst
-rwxr-xr-x  1 joao joao  1532 Ago 14  2013 prerm
drwxr-xr-x  3 joao joao  4096 Ago 14  2013 var
```

Figura 4.3: Ficheiros de um pacote debian

```
  GNU nano 2.2.6              Ficheiro: control

Package: apacheds
Version: 2.0.0-M15
Section: devel
Priority: optional
Architecture: amd64
Installed-Size: 11504
Maintainer: Apache Directory Project <dev@directory.apache.org>
Description: LDAP Server
 ApacheDS is an embeddable directory server entirely written in Java, which
 has been certified LDAPv3 compatible by the Open Group. Besides LDAP it
 supports Kerberos 5 and the Change Password Protocol. It has been designed
 to introduce triggers, stored procedures, queues and views to the world of
 LDAP which has lacked these rich constructs.

                              [ 13 linhas lidas ]
^G Obter Ajud^O Gravar     ^R Ler Fichei^Y Página Ant^K Cortar Tex^C Pos Cur
^X Sair      ^J Justificar^W Onde está ^V Página Seg^U DesCortar ^T Ortografia
```

Figura 4.4: Ficheiro *control*

A convenção de atribuição de nome a um pacote segue a seguinte forma: <Nome do pacote>_<Versão>-<Versão de Revisão>_<Arquitetura Debian>.deb

4.3 PHP

Como mencionado na secção 3.3, a lógica da API a implementar será desenvolvida na linguagem PHP para manutenção da homogeniedade do sistema em que será integrado. PHP é uma linguagem de programação de servidor procedural e orientada a objetos criada em 1995 e atualmente implementada em milhões de servidores web em todo o mundo. Originalmente criada para desenvolvimento de páginas web dinâmicas, atualmente é também utilizada na lógica de aplicações integradas em servidores web. Encontra-se, aquando do desenvolvimento deste projeto, na versão 5.5.8.

4.3.1 SimpleXML

SimpleXML é uma extensão PHP apresentada na versão 5 que fornece ferramentas para manipulação e obtenção de dados e objetos de mensagens XML para utilização local.

4.3.2 Recess

Recess é uma *framework open-source* PHP para desenvolvimento rápido e simples de aplicações baseadas na arquitetura REST e desenhada para as versões mais recentes de PHP.

Capítulo 5

Trabalho Efetuado e Planeado

5.1 Trabalho Efetuado

Até ao momento o tempo disponível foi utilizado para:

- estudo da plataforma IPBrick
- aprendizagem e adaptação aos métodos de trabalho da empresa
- análise e compreensão da atual forma de integração de aplicações third-party
- análise de aplicações third-party integradas
- estudo das soluções existentes no âmbito do projeto e da sua adaptação às circunstâncias da plataforma IPBrick
- depuração e caracterização das etapas do projeto

5.2 Planeamento

O primeiro passo passará por testar e escolher uma das soluções apresentadas na secção 3.2.1. Com base nessa escolha será iniciado o desenvolvimento da API de integração com as funcionalidades listadas em secção 2.5 na análise de requisitos. Por fim, terá de ser desenvolvida uma aplicação de maneira a serem efetuados testes ao produto final e proceder-se à análise da sua eficácia. A figura 5.1 apresenta o diagrama GANTT do planeamento apresentado.

ID	Nome da Tarefa	Início	Fim	Duração
1	Teste das ferramentas de desenvolvimento	17-02-2014	21-02-2014	5d
2	Teste e escolha das soluções	22-02-2014	13-03-2014	20d
3	Teste de SOAP / XML	22-02-2014	03-03-2014	10d
4	Teste de REST / JSON	04-03-2014	13-03-2014	10d
5	Desenvolvimento de requisitos de pré-instalação	14-03-2014	27-03-2014	14d
6	Desenvolvimento de requisitos de pós-instalação	28-03-2014	10-04-2014	14d
7	Desenvolvimento de requisitos de configuração	11-04-2014	08-05-2014	28d
8	Integração na plataforma	09-05-2014	22-05-2014	14d
9	Teste da API	23-05-2014	05-06-2014	14d
10	Elaboração da dissertação	17-02-2014	06-06-2014	110d

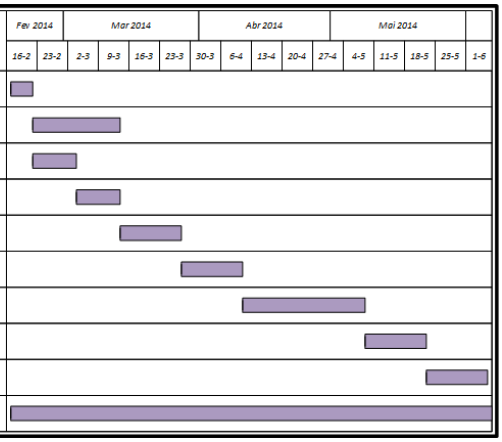

Figura 5.1: Diagrama GANTT

Anexo A

Exemplo de um ficheiro WSDL

A estrutura de um documento WSDL segundo a W3C é a seguinte:

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

    <import namespace="uri" location="uri"/>*

    <wsdl:documentation .... /> ?

    <wsdl:types> ?
        <wsdl:documentation .... />?
        <xsd:schema .... />*
        <-- extensibility element --> *
    </wsdl:types>

    <wsdl:message name="nmtoken"> *
        <wsdl:documentation .... />?
        <part name="nmtoken" element="qname"? type="qname"?/> *
    </wsdl:message>

    <wsdl:portType name="nmtoken">*
        <wsdl:documentation .... />?
        <wsdl:operation name="nmtoken">*
           <wsdl:documentation .... /> ?
           <wsdl:input name="nmtoken"? message="qname">?
               <wsdl:documentation .... /> ?
           </wsdl:input>
           <wsdl:output name="nmtoken"? message="qname">?
               <wsdl:documentation .... /> ?
```

```
            </wsdl:output>
            <wsdl:fault name="nmtoken" message="qname"> *
                <wsdl:documentation .... /> ?
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="nmtoken" type="qname">*
        <wsdl:documentation .... />?
        <-- extensibility element --> *
        <wsdl:operation name="nmtoken">*
            <wsdl:documentation .... /> ?
            <-- extensibility element --> *
            <wsdl:input> ?
                <wsdl:documentation .... /> ?
                <-- extensibility element -->
            </wsdl:input>
            <wsdl:output> ?
                <wsdl:documentation .... /> ?
                <-- extensibility element --> *
            </wsdl:output>
            <wsdl:fault name="nmtoken"> *
                <wsdl:documentation .... /> ?
                <-- extensibility element --> *
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="nmtoken"> *
        <wsdl:documentation .... />?
        <wsdl:port name="nmtoken" binding="qname"> *
            <wsdl:documentation .... /> ?
            <-- extensibility element -->
        </wsdl:port>
        <-- extensibility element -->
    </wsdl:service>

    <-- extensibility element --> *

</wsdl:definitions>
```

Referências

[1] Nick Gall. WOA: Putting the Web Back in Web Services, 2008. URL: http://blogs.gartner.com/nick_gall/2008/11/19/woa-putting-the-web-back-in-web-services/. Citado na página 7.

[2] API Directory - ProgrammableWeb. URL: http://www.programmableweb.com/apis/directory. Citado na página 8.

[3] Alex Ng, Shiping Chen, e Paul Greenfield. An evaluation of contemporary commercial SOAP implementations. *Proceedings of the 5th* . . . , 2004. URL: http://mercury.it.swin.edu.au/ctg/AWSA04/awsa04-proc.pdf#page=72. Citado na página 9.

[4] RFC 4627. URL: http://www.ietf.org/rfc/rfc4627.txt. Citado na página 9.

[5] Hao He. What Is Service-Oriented Architecture. páginas 1–5, 2003. Citado na página 10.

[6] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pœ l Krogdahl, Min Luo, e Tony Newling. Patterns: Service-Oriented Architecture and Web Services. *Contract*, 1:17–42, 2004. URL: http://www.chinagrid.net/grid/paperppt/Patterns-Services.pdf, doi:10.1109/SOSE.2005.5. Citado na página 10.

[7] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, e Clemente Izurieta. Comparison of JSON and XML Data Interchange Formats: A Case Study. *CAINE*, 2009:157–162, 2009. Citado na página 12.

[8] Web Services Architecture. URL: http://www.w3.org/TR/ws-arch/. Citado na página 12.

[9] Understanding SOAP. URL: http://msdn.microsoft.com/en-us/library/ms995800.aspx. Citado na página 13.

[10] Pavan Kumar Potti. On the Design of Web Services : SOAP vs . REST. 2011. Citado na página 13.

[11] NC Huang. A Cross Platform Web Service Implementation Using SOAP. *Unpublished MSc. Thesis, Knowledge Systems* . . . , (April), 2003. URL: http://www.ksi.edu/thesis/rhuang/rhuang.pdf. Citado na página 13.

[12] Understanding WSDL. URL: http://msdn.microsoft.com/en-us/library/ms996486.aspx. Citado na página 14.

[13] Web Service Definition Language (WSDL). URL: http://www.w3.org/TR/wsdl. Citado na página 14.

[14] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures, 2000. Citado nas páginas 15 e 16.

[15] Xinyang Feng e Ying Fan. REST: An alternative to RPC for Web services architecture. *2009 First International Conference on Future Information Networks*, páginas 7–10, Outubro 2009. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5339611, doi:10.1109/ICFIN.2009.5339611. Citado nas páginas vii, 15 e 16.