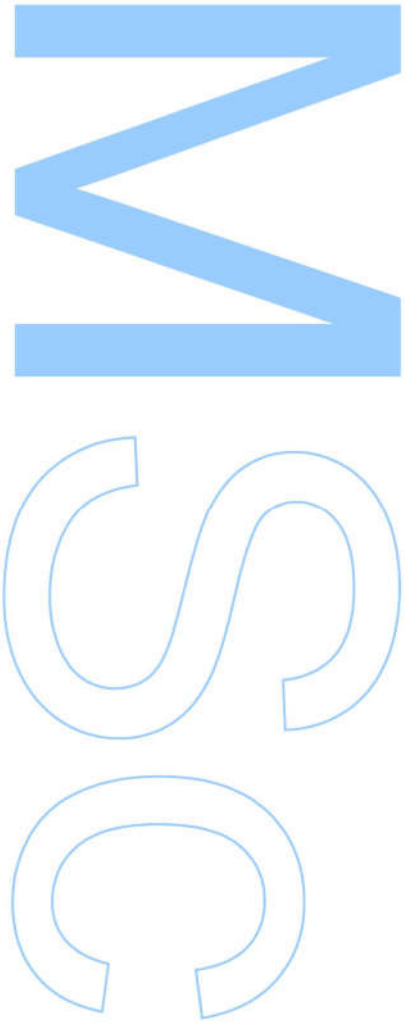
Uma abordagem para evoluir sistemas web legados para *web services*

Revailton de Souza Castro Junior
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2017
Orientador
José Paulo Leal

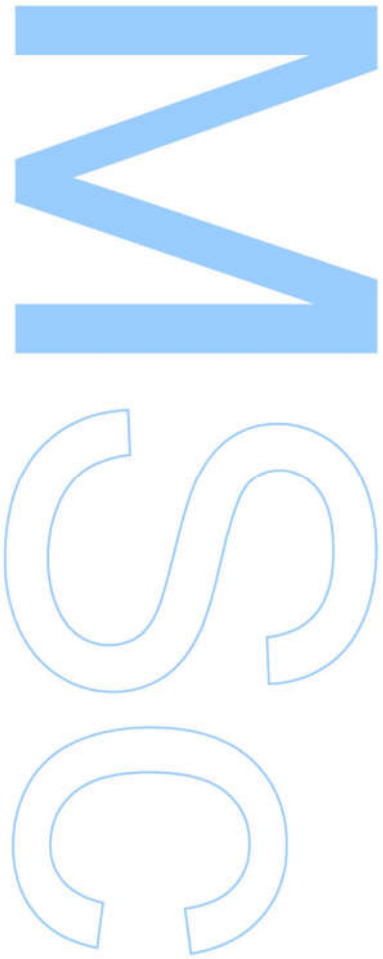


U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.
O Presidente do Júri,

Porto, _____/_____/_____



Agradecimentos

Agradeço aos meus pais por tudo que me ensinaram, pelo exemplo de seres humanos que são; à minha esposa pelo apoio incondicional e todo o seu carinho; agradeço de igual modo aos demais familiares pelos valores que carregam dentro de si e dão a entender o verdadeiro significado da palavra família.

Agradeço aos novos amigos, aos amigos de longa data, estejam eles a (l)este ou a oeste do atlântico, pela motivação e toda ajuda que me deram.

Agradeço a todos os professores que tive a oportunidade de conhecer no Departamento de Ciência de Computadores da FCUP, em especial ao meu orientador José Paulo Leal por ter me guiado ao longo deste trabalho sempre de modo solícito.

Resumo

A reutilização de sistemas de informação é um tema que desperta o interesse de muitas pesquisas e de empresas. Isso ocorre por causa dos benefícios que ela traz, visto que tipicamente o desenvolvimento de sistemas possui elevados custos e demanda tempo.

O objetivo deste trabalho é desenvolver uma abordagem para reutilizar sistemas *web* legados, através da criação de uma semântica de *web service*. Através dessa abordagem será possível criar uma interface de comunicação, de forma menos custosa, tornando os sistemas *web* legados interoperáveis.

Para além disto, fez parte deste trabalho a codificação de um protótipo de sistema. Com ele é possível criar *web services* de acordo com a abordagem desenvolvida. E, a fim de avaliar a satisfação dos utilizadores com relação a usabilidade do protótipo, uma pesquisa de satisfação foi feita. Os resultados demonstram que, apesar de haver alguma fragilidade na usabilidade, o sistema é confiável e consistente, demonstrando ainda que a abordagem e o sistema são factíveis para o fim proposto.

Palavras-chave: sistemas web, legado, reutilização, web services, RESTful.

Abstract

The software reuse is a topic that arouses the interest of many researches and companies. This is because of the benefits it brings, since typically the development of systems has high costs and requires time.

The goal of this work is to develop an approach to reuse legacy web systems by creating a web service semantics. Through this approach, it will be possible to create a communication interface, less costly, making legacy web systems interoperable.

In addition, the coding of a prototype system was part of this work. With it you can create web services according to a developed approach. And, in order to assessing users' satisfaction about the usability of the prototype, a satisfaction survey was performed. The results show that, although there is some fragility regarding the usability, the system is reliable and consistent, demonstrating that the approach and the system are feasible for the proposed objective.

Key words: web systems, legacy, reuse, web services, RESTful.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API - Application Programming Interface

GUI - Graphical user interface

GWT - Google Web Toolkit

HTTP - Hypertext Transfer Protocol

MITM - Man-in-the-middle

REST - Representational state transfer

RPC - Remote Procedure Call

URL - Uniform Resource Locator

W3C - World Wide Web Consortium

Capítulo 1

Introdução

1.1 Motivação

Com o passar dos anos, alguns sistemas de informação tornam-se obsoletos por diversas razões como, por exemplo: evolução tecnológica ou simplesmente por terem atingido o fim do seu ciclo de vida. Entretanto, são indispensáveis e permanecem suportando funções vitais do negócio. É comum nestes sistemas mais antigos um projeto arquitetónico não expansível, código confuso, documentação pobre ou inexistente e um histórico de modificações mal administrado [5]. Ainda assim, esses sistemas dão suporte a funções vitais do negócio e são imprescindíveis.

Entretanto, esses sistemas frequentemente evoluem por algumas razões, por exemplo, a necessidade de torná-lo interoperável com outros bancos de dados ou com sistemas mais modernos. Quando isso ocorre, o sistema deve passar por uma reengenharia para que permaneça viável no futuro [5]. Quando se trata um sistema web, o desafio de manter estas aplicações operacionais converge para a adoção de uma arquitetura orientada a serviços (SOA) que permita a interoperabilidade entre diferentes sistemas.

Considerando o custo de desenvolvimento de um novo sistema, criar uma camada que o torne interoperável resulta em menos custos e um maior ganho. Ou seja, será mais racional e econômico reutilizar e adaptar estes trabalhos para gerar uma semântica de *web services* em vez de construí-los do zero [3]. Atualmente os modelos mais utilizados para construir *web services* são baseados na especificação SOAP ou no modelo *RESTful*.

No momento de criar essa camada, é necessário definir uma abordagem que melhor se adeque às características do sistema para gerá-la. E, considerando esta motivação, este trabalho objetiva definir uma abordagem para gerar uma semântica de *web services*. Para isto, serão consideradas no âmbito deste trabalho as seguintes características para os sistemas-alvo: sistemas *web* baseados no protocolo HTTP v1.1, ou HTTPS; sistemas em que o *front-end* recebe HTML; que utilizem linguagens de formatação para gerar conteúdo no servidor como por exemplo JSP, ASP ou PHP. Dito isto, não farão parte do âmbito deste trabalho sistemas baseados em *Applets* ou *Adobe Flash Player*, por exemplo. Para além disto, não será de interesse deste trabalho a análise ou manipulação de eventuais informações contidas em mecanismos de gerência de sessão do usuário, JavaScript e folhas de estilho.

Faz-se aqui uma observação aos leitores deste trabalho: as características de sistemas citadas acima coincidem com as características de muitos sistemas legados. Portanto, ao longo deste trabalho quando for lido o termo sistema *web* legado, tenha em mente as características mencionadas.

1.2 Objetivos

O principal objetivo deste trabalho é expor funcionalidades de um sistema *web* legado como *web services*. Para além deste objetivo principal, listamos a seguir os objetivos específicos estabelecidos:

- Suporte aos principais modelos arquitetónicos utilizados no SOA, nomeadamente SOAP e RESTful;
- Suporte aos sistemas *web* que utilizem linguagem que geram formatação no servidor, por exemplo ASP, JSP ou PHP;
- Suporte aos sistemas *web* que utilizam o protocolo de transporte HTTPS - conexão criptografada sobre o HTTP.

1.3 Resumo da abordagem utilizada

A abordagem utilizada neste trabalho consistirá na utilização de um *proxy* HTTP para capturar o tráfego de um determinado sistema alvo. O intuito é obter os parâmetros dos pedidos e as respetivas informações de resposta. Após a captura, será possível visualizar os dados trafegados e selecionar as informações de interesse para em seguida gerar uma semântica de *web-services* baseada nos esquemas RESTful ou SOAP.

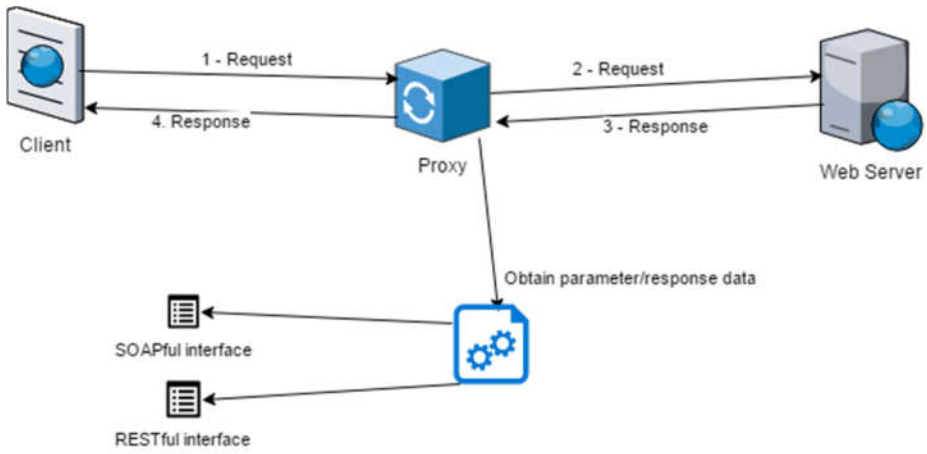


Figura 1 - Proxy

1.4 Estrutura da dissertação

Esta dissertação encontra-se dividida em seis capítulos. Este primeiro capítulo descreve a motivação do trabalho, bem como os objetivos propostos, além da estrutura da dissertação. O segundo capítulo, correspondente ao estado da arte. Diz respeito às tecnologias envolvidas no âmbito dessa dissertação, além de abordar os trabalhos relacionados.

O terceiro capítulo descreve a composição da abordagem utilizada, onde é possível ter uma ideia de forma conceitual dos elementos e passos que compõem este projeto. O quarto capítulo detalha os componentes que fazem parte da implementação do sistema, como a interface gráfica do usuário, o *proxy* HTTP e a geração de serviços *RESTful*.

No quinto capítulo é apresentado o resultado de alguns testes feitos com sistemas reais para avaliação dos objetivos definidos. Em adição, é apresentado o resultado de uma pesquisa de satisfação feita com seis utilizadores, no intuito de avaliar a usabilidade. Por fim, no sexto capítulo há um resumo das principais conclusões e sugestões para trabalhos futuros.

Capítulo 2

Estado da Arte

2.1 Tecnologias Envolvidas

Nesta subsecção será abordado o conceito de *web services* e suas formas de implementação mais conhecidas atualmente: SOAP e RESTful. Uma descrição será feita de cada uma e ao fim uma breve análise comparativa é apresentada.

2.1.1 *Web Services*

Com a evolução da Internet e das tecnologias, os sistemas de informação precisaram lidar com o problema da interoperabilidade. Desde então este tema tem sido de interesse de várias empresas que em conjunto começaram a desenvolver tecnologias para padronizar uma solução. Umas das primeiras iniciativas foi realizada por um consórcio de grandes empresas da época como a Microsoft e IBM por volta do ano 2000, a *World Wide Web Consortium* (W3C), em que foi proposto o *Simple Object Access Protocol* (SOAP) [8].

Futuramente, a W3C definiu a *Web Services Architecture* e conceituou assim um *Web Service: "*é um sistema de software projetado para suportar a interação interoperável máquina-a-máquina sobre uma rede". *Web Service* é uma noção abstrata que deve ser implementado por um agente concreto [4].

Para ilustrar o modelo proposto pela W3C, os principais passos e responsabilidades são ilustrados na Figura 2.

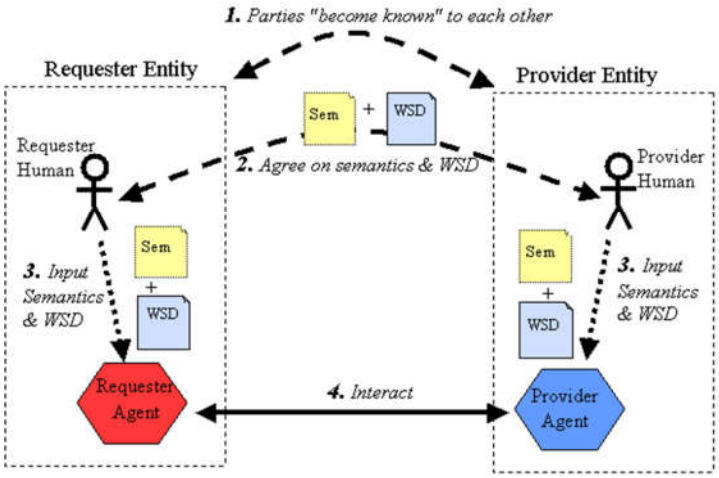


Figura 2 - WS Roles
Fonte: Uma Arquitectura Web para Serviços Web[1]

1- Entidades *Requester* e *Provider* se conhecem (ou ao menos um conhece o outro);
2- Ambos, de alguma forma, concordam com a descrição do serviço e a semântica que irão reger a interação entre eles;
3- A descrição do serviço e a semântica são compreendidas pelos agentes *Requester* e *Provider*;
4- Agentes *Requester* e *Provider* trocam mensagens, realizando assim tarefas em nome de suas respetivas entidades.

Pelo modelo da W3C, uma arquitetura *web service* deve ser implementada através de um conjunto de tecnologias, entre as principais: *Web Service Description Language* (WSDL), SOAP e XML. Entretanto, um outro estilo arquitetónico para se implementar *Web Services* vem sendo muito utilizado, o *Representational State Transfer* (REST), abordagem proposta por Roy Fielding, em sua tese de doutoramento [9].

Portanto, neste capítulo falaremos a respeito das duas abordagens, vantagens e desvantagens de cada uma, e uma comparação das principais características de cada uma.

2.1.2 SOAP

Trata-se de um protocolo de comunicação baseado em XML para troca de mensagens entre agentes em ambientes distribuídos. De acordo com Marzullo [10], em sua versão 1.2, a especificação SOAP descreve quatro componentes principais necessários para a composição de uma mensagem:

[1] Disponível em: https://repositorio-aberto.up.pt/handle/10216/281

- Convenções de formatação para o encapsulamento e distribuição dos dados em forma de um envelope: define as convenções para descrição do conteúdo das mensagens e impõe restrições de como a mensagem deve ser processada;
- Um protocolo de transporte: provê um mecanismo genérico para enviar mensagens SOAP via protocolos de baixo nível como o HTTP;
- Regras de codificação (*encoding*): proveem padrões de mapeamento para diferentes tipos de dados de aplicações em uma representação no formato XML;
- Mecanismo RPC: provê um meio de representar chamadas de procedimentos remotos e seus valores de retorno. SOAP é capaz de descrever uma chamada remota de procedimento (RPC) ou invocar um método remoto pela troca de mensagens XML.

Uma mensagem SOAP comum apresenta uma estrutura conforme a Figura 3, e contém os seguintes elementos:

- SOAP *Envelope*: elemento obrigatório, determina como o documento XML é transformado em uma mensagem SOAP e como deve ser traduzida pelo *Web Service* no serviço real [10];
- SOAP *Header*: elemento opcional. Trata-se de um mecanismo de extensão que fornece uma forma de passar informações em mensagens SOAP que não fazem parte da "carga útil" da aplicação [11];
- SOAP *Body*: elemento obrigatório. Contém as principais informações, que serão transmitidas fim-a-fim.

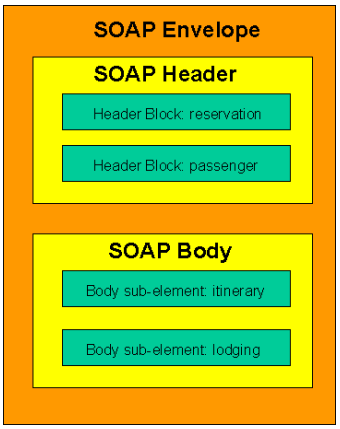


Figura 3 - SOAP envelope
Fonte: SOAP Messages Example[2]

Algumas regras de sintaxe são importantes para utilizar uma mensagem SOAP: deve ser codificada usando XML; deve-se usar o SOAP Envelope *namespace*; deve-se usar o SOAP

[2] Disponível em: https://www.w3.org/TR/2003/REC-soap12-part0-20030624/#Example

Encoding namespace; não deve conter uma referência DTD e não deve conter instruções de processamento [12].

O elemento SOAP *Envelope* é o elemento raiz, e deve possuir um *namespace*: http://www.w3.org/2003/05/soap-envelope/ (o URI varia de acordo com a versão) e mais um atributo que determina o *encoding* dos dados que serão transmitidos.

No SOAP *Header*, se presente, há três atributos que são definidos no *default namespace*:

- *mustUnderstand*: indica se uma determinada entrada do *header* deve ser processada.
- *actor* (SOAP 1.1) ou *role* (SOAP 1.2): especifica se um determinado nó deverá processar o bloco *header*.
- *encodingStyle*: define o tipo de codificação para o bloco de mensagem.

O elemento *Body* e seus elementos filhos são utilizados para a troca de informação entre o remetente SOAP inicial e o recetor SOAP final. Os elementos imediatamente filhos do elemento SOAP-Body devem ser qualificados com um *namespace.* Estes elementos do *Body* são definidos de acordo com o domínio do negócio. Ademais, o SOAP define um elemento filho no elemento *Body*, o elemento FAULT. Trata-se de um elemento para reportar erros.

Na Figura 4 podemos ver um exemplo de um pedido SOAP. E na Figura 5 a respetiva resposta:

```
<?xml version="1.0"?>
<soap:Envelope
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
 soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<soap:Body>
  <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
    <m:Item>Apples</m:Item>
  </m:GetPrice>
</soap:Body>
</soap:Envelope>
```

Figura 4 - SOAP request

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<soap:Body>
  <m:GetPriceResponse xmlns:m="http://www.w3schools.com/prices">
    <m:Price>1.90</m:Price>
  </m:GetPriceResponse>
</soap:Body>
</soap:Envelope>
```

Figura 5 - SOAP response

2.1.3 *REST*

O *Representational State Transfer* (REST) consiste em um termo apresentado por Roy Fielding em sua tese de doutoramento [9] no ano de 2000. O estilo REST é uma abstração de elementos arquiteturais dentro de um sistema de hipermédia distribuído. O *REST* ignora os detalhes de implementação do componente e sintaxe de protocolo e foca nos papéis dos componentes, nas restrições sobre sua interação com outros componentes, e na sua interpretação de elementos de dados significativos. Com isso, tenta-se minimizar a latência e comunicação de rede enquanto, ao mesmo tempo, tenta maximizar a independência e escalabilidade de implementações de componentes [13].

A primeira edição do REST foi desenvolvida entre outubro de 1994 e agosto de 1995, originalmente como um meio de comunicação de conceitos da web durante o desenvolvimento da especificação do HTTP/1.0 e da proposta inicial do HTTP/1.1. Nos anos subsequentes o REST foi iterativamente melhorado e foi originalmente referido como o "*HTTP object model*", mas esse nome muitas vezes levou a sua má interpretação como o modelo de implementação de um servidor HTTP [13].

Como o próprio Fielding afirma [13]: "*The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves: a network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user*".

Um serviço baseado em REST é conhecido como um serviço *RESTful*. Apesar do estilo arquitetural REST ser independente de qualquer protocolo, quase todas as implementações de serviços *RESTful* usam HTTP como protocolo intrínseco. Em geral, os serviços *RESTful* possuem as seguintes características e propriedades:

- Representação: o foco dos serviços *RESTful* está nos recursos e em como prover acesso a estes recursos. Usualmente o acesso é provido através de JSON ou XML.
- Mensagens: o cliente comunica com o servidor através de mensagens. O cliente envia um pedido e o servidor devolve uma resposta. Esta comunicação é realizada através do protocolo HTTP. É importante conhecer a estrutura do protocolo HTTP para se trabalhar com *RESTful*, por isso, demonstra-se a seguir, através das figuras 6 e 7 o formato de um pedido e de uma resposta:

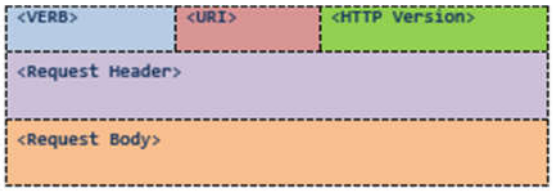


Figura 6 - HTTP Request

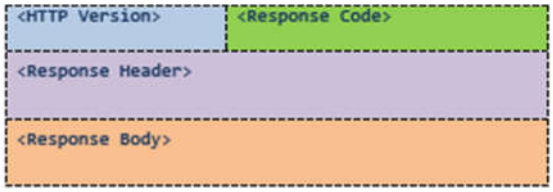


Figura 7 - HTTP Response

- URIs: o REST requer que cada recurso seja identificável, portanto, deve possuir ao menos uma URI com o propósito de identificar um recurso ou uma coleção. A operação realizada neste recurso é determinada pelo verbo utilizado no HTTP *request*, como, por exemplo: GET, POST e DELETE. Isso permite chamar o mesmo recurso com diferentes verbos HTTP e realizar diferentes operações.

- *Interface* uniforme: serviços *RESTful* devem possuir uma interface uniforme e isso é possível com o HTTP/1.1, em que são definidos os métodos de acordo com a Tabela 1.

Método	Operação realizada no servidor
GET	Lê um recurso
PUT	Insere um novo recurso ou atualiza se já existir. Método idempotente. O cliente sabe a URL completa.
POST	Envia dados no corpo do pedido para processamento no servidor.
DELETE	Apaga um recurso.
OPTIONS	Lista as operações permitidas para um recurso.
HEAD	Responde apenas com o <Response Header>, sem o <Response Body>

Tabela 1 - HTTP methods

- *Stateless*: um serviço *RESTful* não mantém o estado da aplicação para o cliente, ou seja, um pedido não pode ser dependente de uma outra do passado. Cada pedido é tratado independentemente.
- Links entre recursos: a representação de um recurso pode conter *links* para outros recursos, assim como uma página HTML pode conter *links* para outras páginas. Vamos supor que um cliente faz um pedido a um recurso que contém múltiplos recursos. Em vez

de retornar todos estes recursos, pode-se responder com uma lista de *links* que referencia cada recurso.

- *Caching*: consiste em armazenar o resultado gerado e usá-lo futuramente em um pedido idêntico, em vez de gerá-lo novamente. Isso pode ser feito no cliente, no servidor, ou em qualquer componente entre eles como, por exemplo, um servidor proxy.

2.1.4 Análise comparativa

A ideia de um *web-service* consiste em um *software* projetado para integração de sistemas distribuídos. Tanto a especificação SOAP quanto o modelo REST conseguem implementar esse software, porém, de maneiras diferentes. São muitas as diferenças técnicas entre *web-services* SOAP e *RESTful* e, sempre que é necessário implementar um *web-service* surge a dúvida: qual é o melhor modelo para se adotar? A resposta não pode ser dada de imediato. É preciso conhecer bem as características de cada modelo, assim como as restrições e requisitos para a aplicação em questão.

Nesta secção não se pretende fazer um estudo comparativo profundo sobre a discussão de qual seria a melhor implementação. Esse tipo de pesquisa pode ser encontrado em outros trabalhos [14] [15] [16]. Falaremos aqui apenas a respeito das principais características de cada uma, que são fundamentais para o processo de escolha. A Tabela 2 apresenta estas principais características e algumas comparações:

SOAP	RESTful
Usa o HTTP como protocolo de transporte, sendo possível utilizar outros.	Utiliza o poder inerente do HTTP como protocolo de aplicação, utilizando-se dos verbos (POST, GET, etc.)
Fornece maior segurança através da especificação WS-*security*	Menos seguro, provê apenas a utilização do HTTPS
Retorna XML	Pode trabalho com JSON, XML
Não permite *caching*	É possível fazer *caching* das operações GET
É possível manter sessões	É *stateless*, cada pedido é independente do anterior.
Manutenção de código mais complexa em relação a implementação RESTful	Manutenção de código mais simples em relação ao SOAP
Possui a carga útil mais pesada, podendo utilizar apenas o protocolo XML.	Possui a carga útil mais leve, e pode utilizar: XML, JSON, MIME.
Maior consumo de largura de banda	Menor consumo de largura de banda

A identificação do serviço pode ser feita através da URI ou conforme a especificação *WS-Addressing*	A identificação do serviço é feita através da URI apenas.

Tabela 2 - Comparação SOAP vs RESTful

É importante salientar que ambos os modelos de implementação de *web-service* possuem pontos fortes e pontos fracos, e que a escolha do qual utilizar vai depender das restrições e requisitos de cada situação em específico.

SOAP é mais indicado quando é necessário ter operações com um maior formalismo entre as partes, quando for necessário manter sessões ou quando for necessária uma maior segurança fim-a-fim. Em contrapartida, quando as operações exigirem a utilização de *caching*, quando a largura de banda for limitada e não for necessário manter sessões a utilização do modelo *RESTful* é mais indicado.

2.2 Trabalhos relacionados

O estudo para evoluir sistemas legados não é algo novo. Existem diversas técnicas e abordagens que variam de acordo com a tecnologia do sistema e a forma de análise. Rochimah e Sankoh fizeram uma pesquisa [1] a fim de avaliar diversos trabalhos publicados com diferentes técnicas de migração de sistemas legados em sistemas baseados na arquitetura *Web Services.* Esse estudo considerou publicações entre os anos de 2005 e 2015 publicadas na IEEExplore ou ScienceDirect.

Em um dos artigos [17] foi realizado um estudo e apresentado um *framework* para migrar aplicações web monolíticas para um sistema com Arquitetura Orientada a Serviços (SOA). A abordagem consiste em analisar sistemas desenvolvidos com linguagem de script, como PHP e Python. Nesta abordagem foram definidos quatro passos principais:

i. Procurar e identificar em toda a base de código os potenciais serviços de negócios;
ii. Separar os serviços da aplicação web, quebrando a lógica da aplicação em serviços separados. Considerando a uma aplicação baseada em PHP, esse passo é feito através das tecnologias SCA (*Service Component Architecture*) e SDO (Service Data Object);
iii. De acordo com um conjunto de critérios, selecionar dentre os serviços independentes gerados no passo anterior os que devem ser migrados para componentes SOA;
iv. O passo final consiste em reintegrar os serviços selecionados em uma nova aplicação orientada a serviços.

Um outro trabalho [18] apresenta o projeto MIGRARIA, em que consiste na utilização de um processo de modernização de sistemas legados, baseando-se no padrão arquitetural MVC para gerar uma nova camada de acesso à aplicação feita com a API REST. O processo começa através da utilização da técnica de engenharia reversa, para que seja possível gerar uma representação conceitual do sistema legado, chamada de MIGRARIA MVC *meta-model*. Este *meta-model* fornece uma representação de objetos de dados, seus atributos, relacionamentos e operações sobre eles. Então, com estas informações é gerada toda a API REST necessária para a nova aplicação.

Outro trabalho [24] com o foco em transformar aplicações *web* em *web services* apresenta uma abordagem baseada em um processo de migração que utiliza técnicas de engenharia reversa do tipo *black-box.* Para isso, utiliza-se um *wrapper*, um componente capaz de gerar um modelo das funcionalidades que devem ser expostas como *web services*. Esse modelo trata-se de um autómato finito não determinístico, em que cada estado está associado a uma página, a seus campos e ao tipo da página (*initial*, *input*, *output*, *exception*, *final*).

Nos trabalhos relacionados são apresentadas abordagens em que são necessários conhecimentos técnicos como, por exemplo, diagramação UML ou até mesmo de alguma linguagem de programação para realizar a engenharia reversa do sistema alvo.

Capítulo 3

Composição da abordagem

A abordagem desenvolvida foi concebida de tal maneira que um utilizador que não possua qualquer conhecimento técnico de linguagens de programação consiga criar *web-services*. Basta seguir o processo definido no sistema web criado para esta finalidade, o qual foi batizado de ***Tupi***.

O sistema Tupi consiste em um sistema de plataforma *web* que possui internamente um servidor *proxy* HTTP. Através dele é possível obter os dados de navegação do utilizador em um sistema específico, o qual chamaremos de sistema **alvo**. Com estes dados é criado um repositório que servirá em seguida para a geração de classes Java e expressões XPath. Estas expressões referenciam campos do HTML e as classes Java constituirão os *web-services*. Na Figura 8 é possível observamos um diagrama que inclui os principais componentes do sistema Tupi.

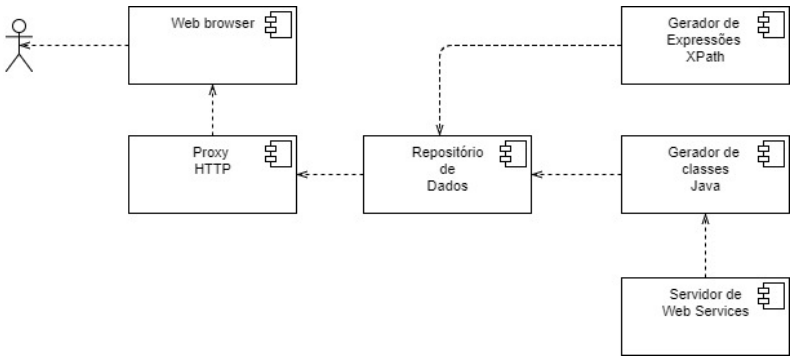


Figura 8 - Diagrama de Componentes

Os principais passos que o utilizador deve seguir através do Tupi para gerar um *web service* estão representados no seguinte diagrama de atividades:

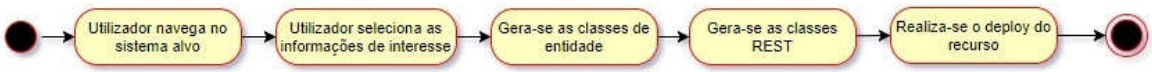


Figura 9 - Principais atividades

Apresentamos nas subsecções seguintes os elementos de maior relevância que fazem parte deste projeto.

3.1 O Servidor Proxy

O processo de geração de *web-services* inicia-se com o utilizador configurando como servidor proxy o computador onde está a executar o Tupi. Isto é necessário pois internamente ao Tupi funciona um servidor *proxy* que tem como função obter os dados da navegação do utilizador no sistema alvo, nomeadamente os dados de *request:* o método HTTP *(GET, POST, DELETE etc)*, o *Header* (*host, encoding, user-agent*, etc), e seus parâmetros.

Tão importante quanto as informações do *request* é o HTML de resposta. De posse dele poderemos construir uma árvore DOM e navegar pelos campos de interesse. De posse destes dois conjuntos de informações: HTTP-R*equest* e o HTML de resposta, podemos modelar um *web service,* fazendo uma analogia com os parâmetros e valores de resposta para um determinado recurso.

O trecho de código da Figura 10 demonstra a ideia principal do funcionamento do Proxy.

```
// Método que filtra todas as conexões que passam pelo Proxy
public HttpFilters filterRequest(HttpRequest originalRequest, ChannelHandlerContext ctx) {

    return new HttpFiltersAdapter(originalRequest) {

        @Override
        public HttpResponse clientToProxyRequest(HttpObject httpObject){

            // verifica se o pedido é para o sistema alvo
            if ( isTargetSystem(originalRequest) ) {
                // armazena os dados do pedido no repositório
                armazenarParametros(httpObject);
                ...
            }
        }

        @Override
        public HttpObject serverToProxyResponse(HttpObject httpObject){

            // verifica se o pedido é para o sistema alvo
            if ( isTargetSystem(originalRequest) ) {
                // armazena os dados de resposta no repositório
                armazenarHtmlResposta(httpObject);
                ...
            }
        }
    }
}
```

Figura 10 - Trecho do proxy

Vamos supor que interessa ao utilizador gerar um *web-service* de uma página que contém informações de departamento de uma empresa. Nesse momento o utilizador deve navegar até a página, por exemplo www.company.pt/departamento.asp?cod=123 e, através do método *clientToProxyRequest,* o *proxy* vai obter os parâmetros do pedido e, através do método *serverToProxyResponse,* o HTML de resposta. Em seguida salva estes dados no repositório localizado no servidor.

3.2 Selecionando a informação de interesse do HTML

No pedido HTTP é fácil localizar exatamente quais são e onde estão os dados de interesse, pois trata-se dos parâmetros enviados. No pedido do tipo POST, os parâmetros estão localizados no corpo da mensagem. E para o tipo GET estão na própria URL do pedido. No trecho de código a seguir podemos perceber como as informações serão obtidas.

```
// armazena os dados do pedido no repositório
public void armazenarParametros (HttpObject httpObject) {

        if(httpObject.getMethod == "GET"){
                parametros = extrairParametrosDaURL(httpObject);
        }else if(httpObject.getMethod == "POST"){
                parametros = extrairParametrosDoCorpoDaMensagem(httpObject);
        }
        // salva os parametros para uma determinada pagina: <key,value>
        salvarParametros(httpObject.getPagina(), parametros);
}
```

Figura 11 - Armazenar parâmetros

No entanto, na resposta HTTP, os dados de interesse estão dispersos na formatação HTML. Para isso, no Tupi será possível visualizar cópias das páginas do sistema alvo que foram navegadas. Quando solicitado pelo utilizador, estas páginas serão renderizadas no Tupi e, através de cliques, o utilizador poderá selecionar os campos de interesse e a localização de seus respetivos valores. Por meio deste mecanismo, geraremos um mapeamento dos campos e valores através de expressões XPath [19]. Por exemplo, imaginemos que o HTML de resposta para o pedido tenha sido o da Figura 12.

```
<html>
 <head></head>
 <body>
     <table width="100%">
       <tr>
        <td> Sigla do Departamento:</td>
        <td> Dep.C </td>
       </tr>
       <tr>
        <td> Responsável: </td>
        <td> Pedro </td>
       </tr>
       <tr>
        <td> Email: </td>
        <td> dep.C@company.pt </td>
       </tr>
       <tr>
        <td> Localidade: </td>
        <td> Porto </td>
       </tr>
     </table>
 </body>
</html>
```

Figura 12 - HTML exemplo de resposta

Vamos supor que interessa ao utilizador gerar um serviço que retorne apenas as informações da Sigla do Departamento e o Email. Após a página ser renderizada no Tupi, o utilizador deve clicar no campo "Sigla do Departamento" com um clique simples, e o sistema interpretará como um campo de interesse e salvará a descrição do campo no repositório. Quando o utilizador clicar no campo "Dep.C" com um clique duplo, o sistema interpretará como o local onde estará a informação desejada e, com isso, vai gerar uma expressão XPath para referenciar a localização da informação e da mesma forma salvará no repositório. O mesmo procedimento deve ser feito para o campo "Email".

Portanto, neste ponto temos uma relação de informações de interesse associadas a uma página HTML.

3.3 Construindo classes de Entidade

No âmbito da engenharia de software uma entidade consiste em um objeto que possui valor para um domínio do negócio. Uma entidade possui características e comportamentos, representa algo que o sistema de informação está interessado em rastrear[3].

É uma boa prática da programação orientada a objetos a criação de classes de entidade para nos referirmos aos objetos que vamos manipular. Por isso haverá no *Tupi* uma funcionalidade em que será possível a criação de *entity classes* através das informações de interesse previamente selecionadas.

A criação dessas classes de entidade, para além de seguir uma padronização da orientação a objetos, será útil para relacionar os campos de um determinado objeto com as respetivas expressões XPath. As classes de entidade que forem criadas no Tupi devem estender a classe TupiEntity e possuirão o formato representado de forma simplificada na Figura 13.

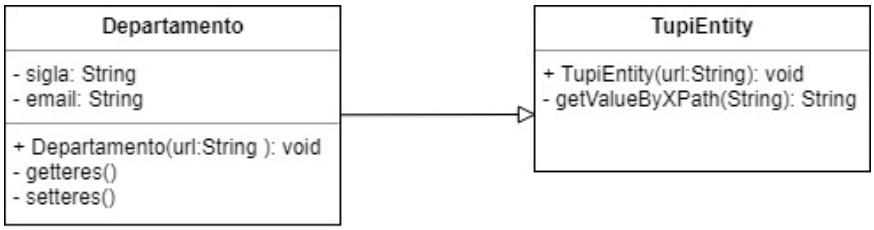


Figura 13 - Exemplo de Classe de Entidade

[3] Inspirado em https://martinfowler.com/bliki/EvansClassification.html

3.4 Construindo serviços *RESTful*

Após a criação da classe de entidade podemos então passar para o próximo passo: a geração de uma classe REST, chamada de recurso. Sem o auxílio do *Tupi*, seria necessário codificar uma classe *Java* semelhante à da Figura 14. Entretanto, através do *Tupi*, esse passo é realizado de forma quase transparente para o utilizador. Através de uma interface gráfica do sistema, o utilizador deve informar um nome para a classe, o nome do método, selecionar o método HTTP adequado (*GET, POST, DELETE* etc) e o objeto de retorno (uma entidade criada previamente) e, com isso, será criado internamente um serviço.

Seguindo o exemplo, criaremos um serviço que representará o recurso Departamento. Através da interface gráfica do Tupi o utilizador deve informar os dados adequados para a correspondente página:

- Nome da classe: DepartamentoResource
- Nome do método: consultar
- Método HTTP: GET
- Tipo de retorno: Departamento

```
...
@Path("departamento")
public class DepartamentoResource {

    public DepartamentoResource() {  }

    @GET
    @Produces("application/json")
    @Path("consulta")
    public Object consulta( @QueryParam("cod") String cod){
        String url = "www.company.pt/departamento.asp";
        String httpMethod = "GET";
        Map<String, String> params = new HashMap<String, String>();
        params.put("cod", cod);

        Departamento obj = new Departamento(url, httpMethod, params);
        return obj;
    }
}
```

Figura 14 - Exemplo recurso Departamento

Com isso, será criado internamente uma classe *Java* que possuirá o método *consulta*(), que retorna a entidade *Departamento* e que, por sua vez, possui expressões XPath que referenciam as informações desejadas em um HTML.

3.5 O servidor de serviços *RESTful*

Após a geração das classes REST, estamos aptos a fazer o *deploy* e disponibilizar o serviço. E para isso utilizaremos um servidor *web* à parte, chamado ***Grizzly***. Optou-se por isolar os recursos em um servidor distinto para não comprometer os dados da aplicação Tupi, já que eventualmente o servidor que hospedará os serviços precisará ser reiniciado quando um novo serviço for criado.

Capítulo 4

Implementação

Este capítulo apresenta os aspetos relevantes da implementação do protótipo. Começa por apresentar a interface gráfica do utilizador, a arquitetura utlizada para a criação do sistema Tupi. Nas subsecções seguintes, descrevem-se os principais passos do processo de criação de *web services* e detalham-se os componentes envolvidos: a captura de dados da navegação do utilizador, a seleção de campos de interesse, a geração das classes Java e a disponibilização dos serviços em um servidor web isolado.

4.1 A interface gráfica do utilizador

A interface gráfica do utilizador será composta basicamente por painéis redimensionáveis e um *menu.* Neste *menu* haverá uma opção para o utilizador definir qual será o sistema alvo, haverá também uma opção para inicializar o servidor *Grizzly* [23] e uma opção com um *Help*. Todas estas opções estarão na área 1 da Figura 15.

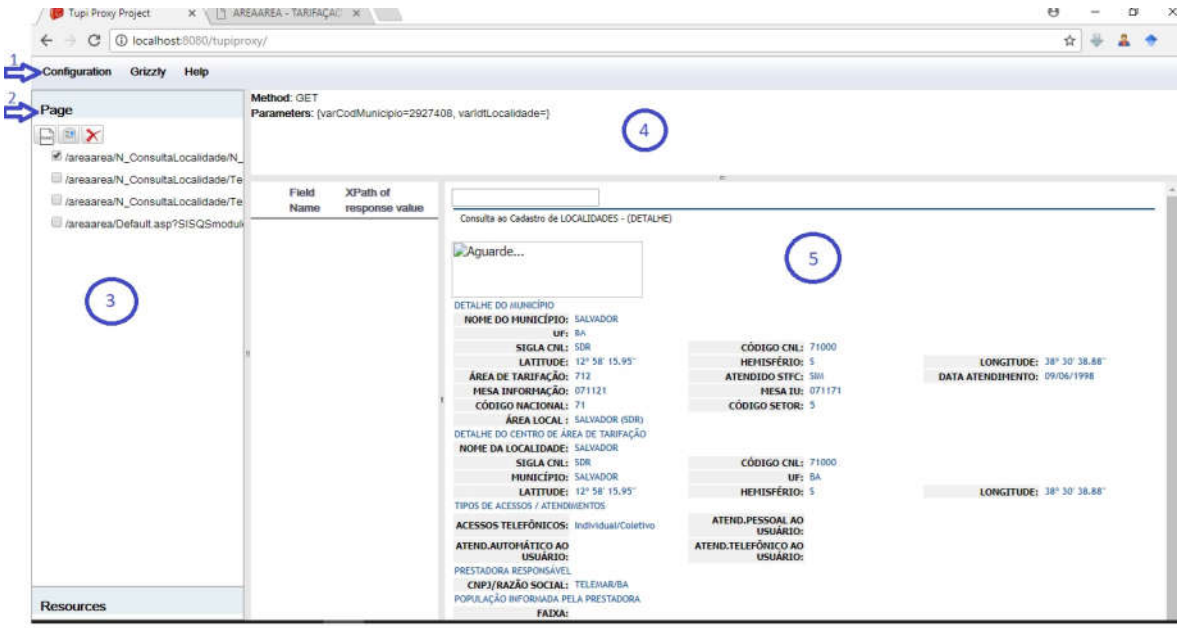


Figura 15 - Tupi GUI

Na área 2 há alguns botões que servirão para criar uma classe de entidade; para criar uma classe REST; e, para apagar todas as páginas listadas, ou seja, todas informações capturadas pelo *Proxy* HTTP.

Quando o utilizador selecionar alguma página na lista de páginas (área 3), as informações do pedido HTTP aparecerão na área 4 e o HTML de resposta aparecerá na área 5.

4.2 A arquitetura do sistema *Tupi*

O sistema Tupi foi desenvolvido de acordo com o modelo cliente-servidor. Neste modelo uma camada do sistema reside no cliente e a outra reside no servidor, podendo estar em computadores distintos ou não. Para este projeto, o utilizador vai aceder ao sistema através de um navegador, fará requisições (baseadas no protocolo HTTP) ao servidor, que estará a executar o servidor de aplicações *Tomcat*, servidor para executar aplicações *web* em *Java*.

A fim de auxiliar no desenvolvimento deste projeto, foi utilizado o *Google Web Toolkit* (GWT). Trata-se de um conjunto de ferramentas que auxiliam o desenvolvimento de sistemas baseados em navegadores. Ele contém uma API que permite a codificação da aplicação do lado do cliente em Java. Em seguida esse código é convertido de modo que sua disponibilização em produção seja no formato de uma aplicação *web* em uma combinação de HTML, CSS e JavaScript.

Outra facilidade do *GWT* é a disponibilização de *frameworks* de comunicação entre o cliente e o servidor. Para este projeto, foi utilizado o GWT-RPC*,* um mecanismo de invocação remota baseado em HTTP que implementa serviços *web* usados na GUI do GWT.

Quando se utiliza o GWT-RPC, há três elementos envolvidos nas chamadas de procedimentos remotos [20]:

- o serviço que corre no servidor (o método que está a ser chamado);
- o código do lado do cliente que invoca o serviço;
- os objetos de dados Java que são passados do cliente para o servidor através dos argumentos dos métodos invocados e o respetivo valor de retorno do método que é transferido para o cliente.

Para implementar a comunicação GWT-RPC é preciso codificar três componentes:

- uma *interface* que estende *RemoteService* e define todos os métodos RPC;
- criar uma classe que estende *RemoteServiceServlet* e implementar a interface criada;
- definir uma *interface* assíncrona para que seja possível o código do lado do cliente chamar o serviço.

A Figura 16 apresenta um diagrama de classes que demonstra a relação entre as classes citadas acima. Mais informações acerca da classe *TupiServiceImpl* são encontradas em anexo, no item 8.1.

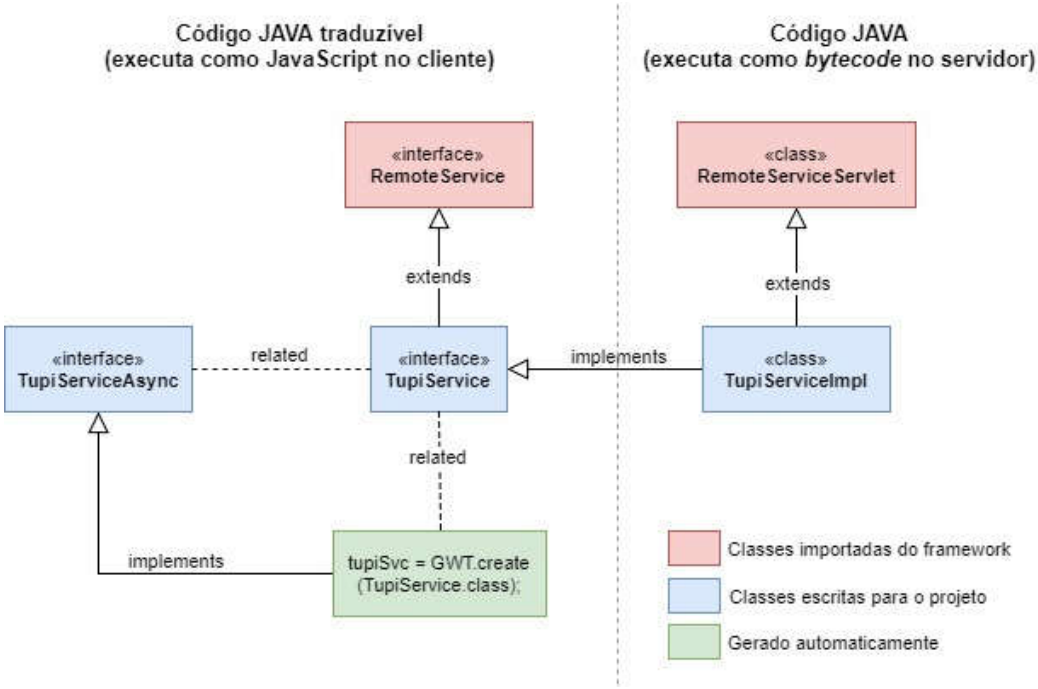


Figura 16 - GWT RPC TupiService
Inspirado no GWT RPC Tutorial [4]

4.3 O repositório de informações

O repositório de informações consiste no local onde ficarão armazenadas as informações capturadas pelo *Proxy* HTTP e que serão úteis no momento da criação das classes *Java* que vão compor os *web-services*.

Este repositório poderia ser implementado de diferentes maneiras, como por exemplo: o armazenamento dos objetos em memória volátil ou, então, o armazenamento dos dados em ficheiros. Optou-se aqui nesse protótipo pela primeira opção, por questões de performance na manipulação dos dados.

Na prática, o repositório de informações consiste em um objeto chamado *Storage* que pode possuir um conjunto de objetos do tipo *ReqRespRegistry,* conforme ilustrado na Figura 17.

[4] Disponível em: http://www.gwtproject.org/doc/latest/tutorial/RPC.html

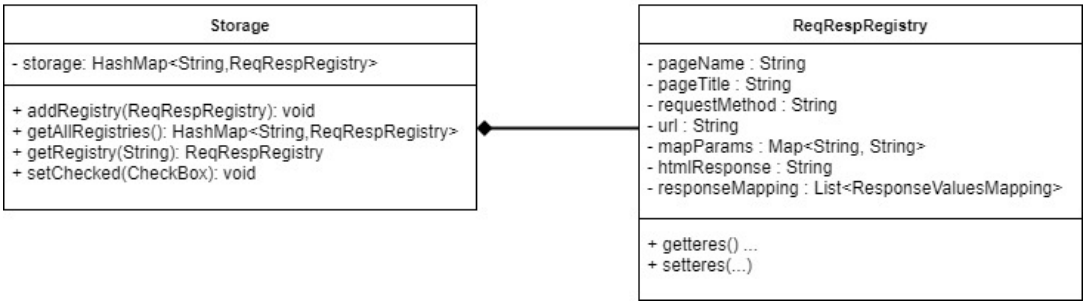


Figura 17 - Diagrama de classe do storage

4.4 O *proxy* HTTP

Um proxy consiste em uma entidade intermediária entre um cliente e um servidor. Essa entidade pode ser um sistema de computador ou uma aplicação que age para intermediar os pedidos de clientes que estão a solicitar recursos a outros servidores.

Com o objetivo de capturar os dados de navegação do utilizador, foi utilizado neste projeto o Littleproxy [21]. Trata-se de um *proxy* HTTP *opensource* escrito em *Java* e mantido por um grupo de programadores voluntários. Em adição, ele possui uma extensão que permite capturar o tráfego HTTPS de forma simples. O *Littleproxy* foi integrado ao Tupi e sua inicialização ocorre logo após a inicialização do Tupi no Tomcat.

Para implementar o *proxy* e filtrar o tráfego capturado, é necessário essencialmente trabalhar com estes dois métodos: *clientToProxyRequest()* e *proxyToClienteResponse(),* como ilustrado na Figura 10. Grande parte das informações necessárias para a criação do *web service* são obtidas através destes dois métodos, onde a cada *request/response* capturado, um objeto *ReqRespRegistry* é criado e adicionado ao repositório de informações.

4.5 Seleção de campos

Para gerar um serviço é preciso obter informações do pedido HTTP e da respetiva resposta. As informações do pedido são facilmente distinguíveis como, por exemplo, os parâmetros de um pedido do tipo GET encontram-se na URL e, para o tipo POST, no corpo da mensagem. Entretanto, os dados da resposta estão dispersos na formatação HTML. É necessário então que o utilizador indique quais são as informações de interesse.

Quando o utilizador seleciona uma página na lista de páginas, a respetiva cópia do HTML será apresentada no painel 5 e, ao clicar uma vez na descrição de um campo, o respetivo texto será guardado. Ao realizar o clique duplo no seu valor de resposta, uma expressão XPath será criada para referenciar a localização da informação. O mapeamento entre os campos

selecionados e as respetivas expressões serão apresentadas no painel 6, conforme ilustrado na Figura 18.

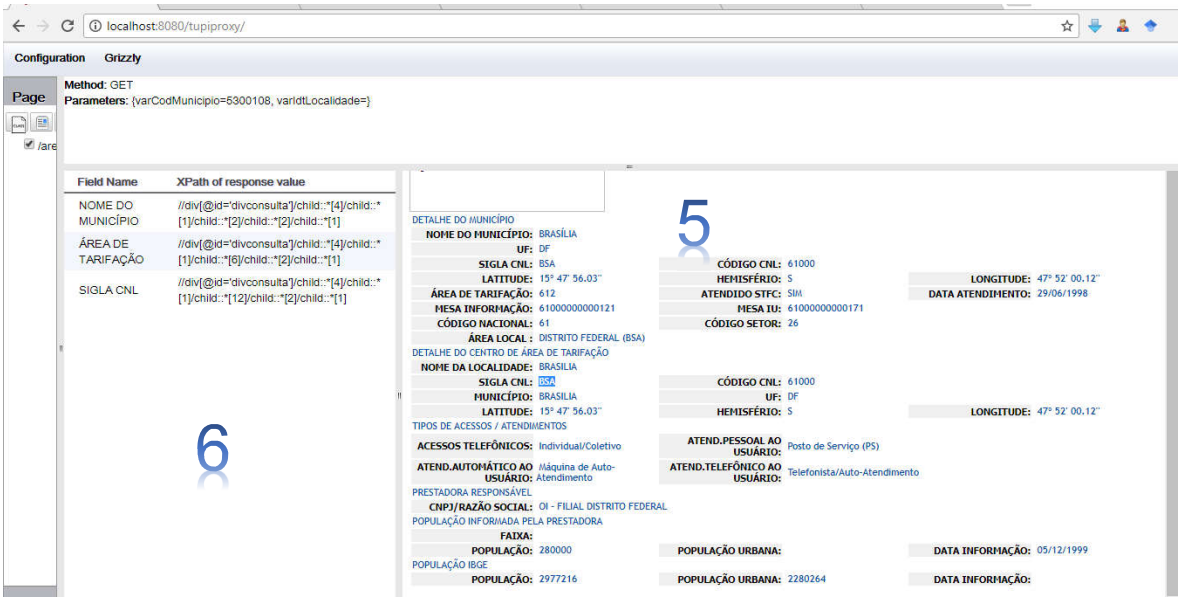


Figura 18 - Mapeamento de campos

Na codificação deste protótipo, foi criado um objeto para que fosse possível manipular e armazenar o mapeamento entre os campos selecionados e as expressões XPath no repositório de informações do Tupi. Trata-se da classe *ResponseValuesMapping*, a qual pode ser vista com mais detalhes em anexo, item 8.2.

Uma questão importante neste ponto que deve ser mencionada é a dificuldade que há para gerar expressões XPath de forma correta, ou seja, de tal maneira que a informação referenciada neste ponto seja exatamente a mesma no momento em que o serviço gerado for invocado. Tal dificuldade origina-se no fato de que frequentemente as páginas HTML não são bem formadas. Para contornar este problema, utilizou-se um HTML *Parser* para limpar o código, tornando-o válido e bem formado.

Um mecanismo para criar as expressões XPath foi desenvolvido para o TUPI. Sua principal ideia consiste em analisar o nó de interesse em relação ao seu progenitor e adicionar a expressão: `child::*[nº de referência em relação ao progenitor]`. Essa análise é feita de forma ascendente até o nó raiz, criando uma expressão de caminho absoluto. Por exemplo, para a página citada na Figura 12, a seguinte expressão é criada ao selecionarmos a localização da palavra Porto: `/child::*[1]/child::*[2]/child::*[1]/child::*[4]/child::*[2]`. Se nessa análise ascendente houver algum nó do tipo div então é criada uma expressão de caminho relativo, por exemplo: `//div[@id='somediv']/child::*[1]/child::*[4]`.

4.6 HTML *Parser*

Para assegurar que manipularemos um HTML bem formado e que as expressões XPath serão eficazes, utilizou-se o JSOUP, uma API Java para manipulação e extração de dados através do HTML DOM. Umas das principais funções deste *parser* é a aplicação de um filtro o qual limpará o HTML. Na prática essa limpeza ocorre da seguinte maneira:

- Define-se um conjunto de *tags* HTML que serão permitidas (chamamos esse conjunto de *whitelist*);
- Define-se, para cada *tag*, quais serão os atributos permitidos;
- Aplica-se esta *whitelist* a um determinado HTML.

Com isso, o HTML resultante desta limpeza possuirá apenas as tags contidas na *whitelist*, além de fazer algumas correções como, por exemplo, fechar *tags* que eventualmente estejam a faltar. O trecho de código em anexo, item 8.3, refere-se ao que foi dito acima.

Vale destacar que a aplicação deste filtro é feita em dois momentos. O primeiro é quando o *proxy* captura o HTML de resposta. E o segundo momento é quando o serviço gerado recebe o HTML de resposta e aplica as expressões XPath dos campos de interesse em um documento HTML filtrado pela mesma *whitelist*. O componente responsável por fazer essa filtragem no segundo momento é a classe *TupiEntity*, que será descrita na subsecção seguinte.

4.7 Criação de classes *Java*

A criação de classes *Java* em tempo de execução é necessária para a composição do *web service* que está a ser criado. Para isso, serão disponibilizados ao utilizador dois botões:  e  no painel 2 conforme Figura 15. A finalidade deles será para, respetivamente, criar a classe de entidade e a classe que corresponderá ao recurso *RESTful*.

A classe de entidade a ser criada possuirá como atributos os campos de interesse que foram previamente selecionados conforme subsecção 4.5. O construtor dessa classe deve invocar o método construtor da superclasse (*TupiEntity*), onde será feita a conexão HTTP com a devida URL e parâmetros. Na Figura 19 podemos ver um trecho de código da classe *Departamento* criada automaticamente pelo sistema em tempo de execução.

```
public class Departamento extends TupiEntity{

    private String sigla;
    private String responsavel;
    private String email;
    private String localidade;

    public Loc(String url, String httpMethod, Map<String,String> map) {
      super(url,httpMethod,map);

      this.nome = getValueByXPath("//div[@id='divconsulta']/child::*[2]");
      this.UF = getValueByXPath("//div[@id='divconsulta']/child::*[4]");
      this.email = getValueByXPath("//div[@id='divconsulta']/child::*[6]”);
      this.localidade = getValueByXPath(("//div[@id='divconsulta']/...);

    }

    getteres(){..}
    setteres(...){ }
}
```

Figura 19 - Trecho código Entidade Departamento

A superclasse *TupiEntity,* que deve ser estendida por todas as entidades criadas, já estará criada e funcionará como um componente. Ela será responsável por realizar a conexão HTTP, a aplicação da *whitelist* para limpar o HTML e por obter os campos de interesse através de expressões XPath. O trecho de código ilustrado na Figura 20 demonstra a ideia do componente *TupiEntity*: um método construtor para realizar a conexão, de acordo com o tipo do método HTTP; e um método para obter o valor de um determinado campo através de uma expressão XPath.

```
public class TupiEntity {

public TupiEntity(String url, String httpMethod, Map<String,String>
mapParams){

    if( httpMethod.equalsIgnoreCase("GET") ){
        doc = Jsoup.connect(url)
            .data(mapParams)
            .validateTLSCertificates(false).get();
    }else if( httpMethod.equalsIgnoreCase("POST") ) {
      // just to get a cookie
      Connection.Response form = Jsoup.connect("https://"+url)
          .method(Connection.Method.GET)
          .validateTLSCertificates(false)
          .execute();

      doc = Jsoup.connect(url)
            .data(mapParams)
            .validateTLSCertificates(false).post();

    }

    Document saneHtml = new Cleaner(myWhiteList).clean(doc);
}


public String getValueByXPath(String xpathExp){
    XPath xPath = XPathFactory.newInstance().newXPath();
    XPathExpression expr = xPath.compile(xpathExp);
    return (String) expr.evaluate(saneHtml, XPathConstants.STRING);
}
```

Figura 20 - Trecho de código TupiEntity

Para criar as classes de entidade e recurso REST em tempo de execução, foi utilizada a biblioteca *CodeModel* [22], uma API consolidada para gerar código *Java.*

Logo após a criação da classe que representará um recurso REST, o nome desta classe será escrito em um arquivo de configuração de recursos que servirá de controle para o servidor *Grizzly*, que hospedará os serviços. Uma visão comportamental da criação destes dois componentes (a classe de entidade e a classe REST) estão ilustrados em diagramas de sequência no anexo, itens 8.4 e 8.5 respetivamente.

4.8 O servidor dos serviços

Na implementação deste protótipo há internamente um servidor (*Grizzly*) que será responsável por hospedar de forma isolada do Tupi os serviços que foram criados. Ele terá como referência o arquivo de configuração de recursos mencionado na subsecção anterior. Trata-se de um

arquivo simples de texto, com o nome de uma classe por linha, que listará quais são os recursos disponíveis. Quando o utilizador inicializar o *Grizzly,* ele estará inicializando uma *Thread* que vai ler a lista de recursos existentes no arquivo de configuração; em seguida vai compilar as classes correspondentes e carregá-las no *ClassLoader e, por fim, reiniciar* o contentor que hospeda os serviços. A Figura 21 ilustra a sequência de atividades realizadas na *Thread*.

Esta *Thread* será responsável também por monitorar o diretório em que será definido pelo utilizador como destino das classes que estão a ser criadas. Desta maneira, todas as vezes que um novo serviço for criado e seu nome adicionado ao arquivo de configuração, logo em seguida o Tupi compilará a classe correspondente, carregará no *ClassLoader* e reiniciará o contentor.

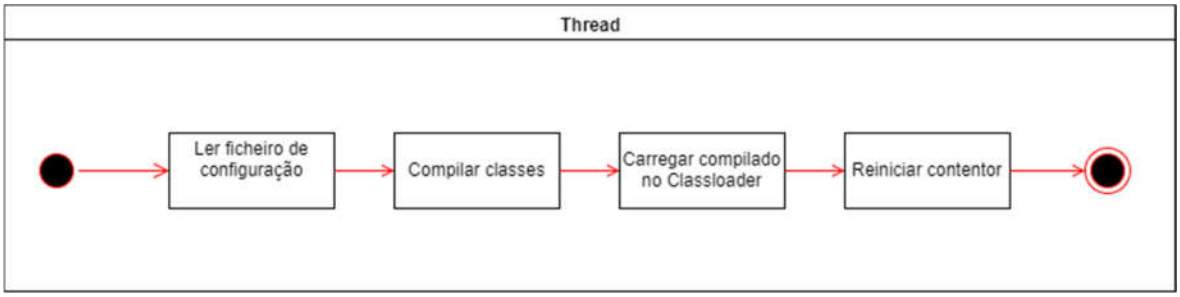


Figura 21 - Grizzly Thread

Optou-se por utilizar o *Grizzly* [23] neste protótipo por se tratar de um servidor leve, que permite a criação e a manipulação de contentores de serviços de forma simples.

Capítulo 5

Validação

Este capítulo tem como objetivo avaliar o protótipo do sistema Tupi através da criação de *web services*. São apresentados inicialmente testes realizados com sistemas reais, para os diferentes métodos HTTP: GET e POST. Em seguida é apresentado o resultado de uma pesquisa de satisfação de usabilidade em que utilizadores testaram o sistema e em seguida preencheram um questionário.

5.1 Validando a criação de serviços

Dividiremos a validação em duas subsecções, a primeira será uma página HTML que faz o pedido através do método *GET*, e a segunda será uma página que faz o pedido através do método *POST*.

Ao acedermos o Tupi, o primeiro passo necessário é configurar o *host* alvo. Como foram utilizados os sistemas da *Agência Nacional de Telecomunicações* (ANATEL), Agência Brasileira de Normas de Telecomunicações, utilizaremos o seguinte *host* através do *menu* Configuration/*Target Host*: anatel.gov.br, conforme a Figura 22.

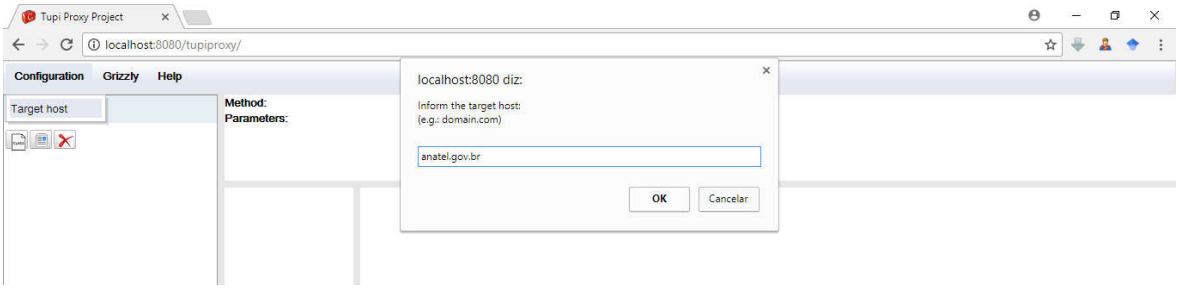


Figura 22 - Configuração do Target Host

Ao concluirmos esse passo, terá sido iniciado um servidor proxy HTTP que será responsável por obter as informações necessárias para geração do *web service*. Em seguida, faz-se necessário que o utilizador configure em seu computador o proxy HTTP e o proxy seguro (HTTPS) para o endereço utilizado pelo Tupi (localhost:8088).

5.1.1 Criação de serviço baseado em pedidos do tipo *GET*

Agora que já configuramos o *target host* e configuramos no computador do utilizador o proxy, devemos navegar até a página do sistema a qual desejamos disponibilizá-la como um

serviço. Para isso, utilizaremos uma página do sistema https://sistemas.anatel.gov.br/areaarea, opção Consultas/Localidades, com o filtro "UF" preenchido com o valor DF, e o filtro "Nome da Localidade" preenchido com o valor Brasília. A pesquisa vai listar o seguinte resultado, conforme Figura 23:

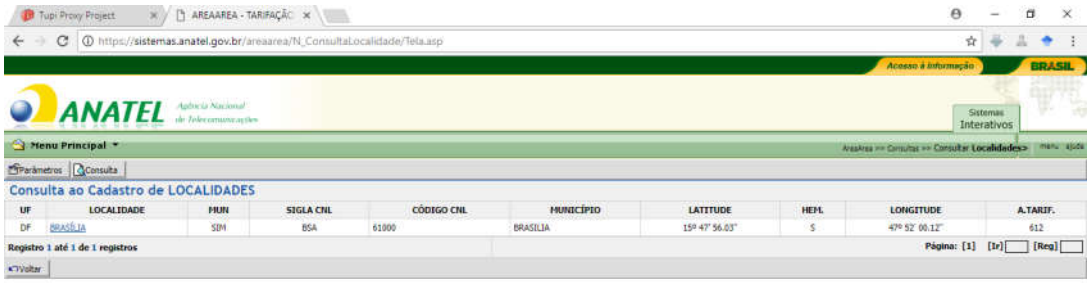


Figura 23 - Sistema AreaArea Lista de Localidades

Ao clicarmos no *link* existente no nome da localidade, neste caso: Brasília, será aberta a página de cadastro de uma localidade, a qual possui informações que desejamos disponibilizar como um serviço, conforme Figura 24.

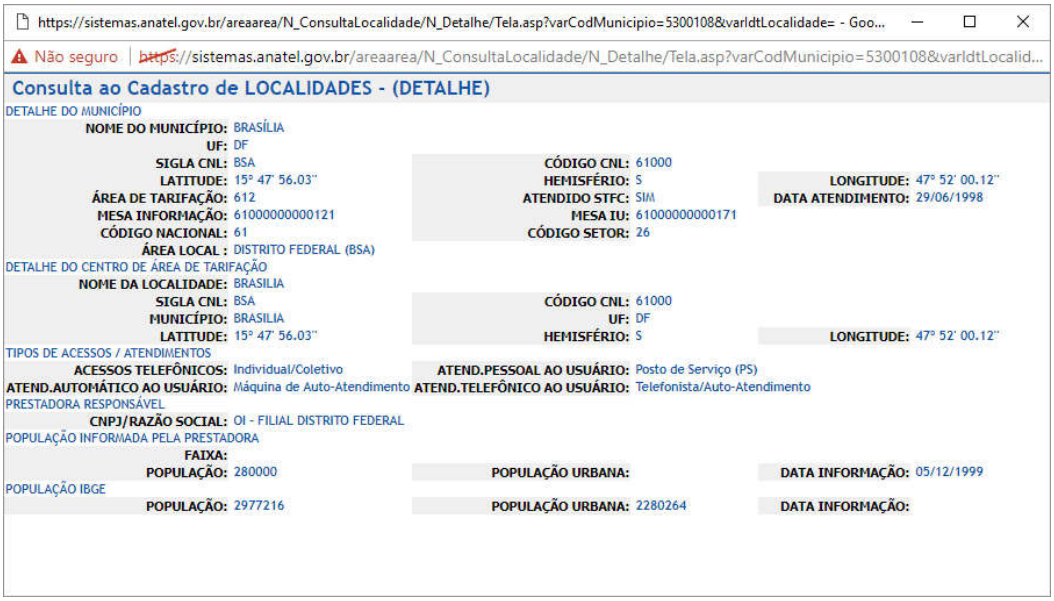


Figura 24 - Cadastro de Localidade

A partir deste ponto já teremos capturado as informações que desejamos pelo Tupi. A Figura 25 ilustra o que será visualizado pelo utilizador quando selecionar, na lista à esquerda, a cópia da pagina. Sabe-se neste ponto que se trata de um pedido HTTP do tipo *GET*, e que possui dois parâmetros: *varCodMunicipio* e *varIdtLocalidade*. Agora, o utilizador deve selecionar, através de cliques, os campos pertencentes ao HTML de resposta que deseja disponibilizar na forma de um serviço. Para isso, deve-se utilizar um clique para selecionar o rótulo de um campo qualquer, e, um duplo clique para selecionar o valor do resultado deste campo, conforme Figura 25.

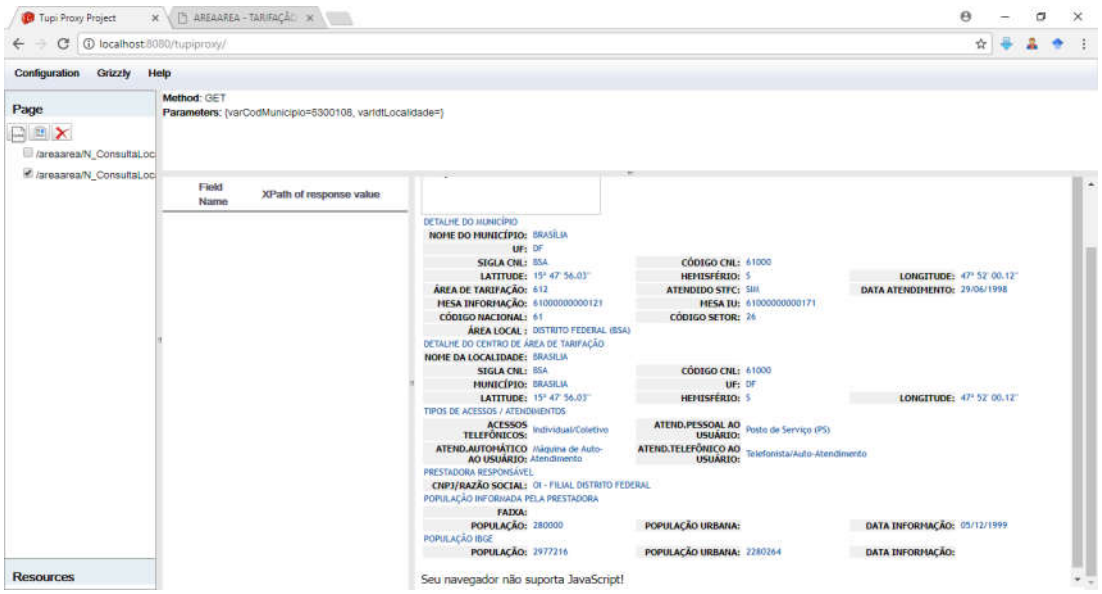


Figura 25 - Visualizando cópia de uma página no Tupi

Quando houver o duplo clique será armazenada uma expressão XPath, ou seja, será guardado uma referência à localização onde o valor de resposta do campo estará. Após a seleção dos campos desejados, a interface do utilizador possuirá informações semelhantes à da Figura 26.

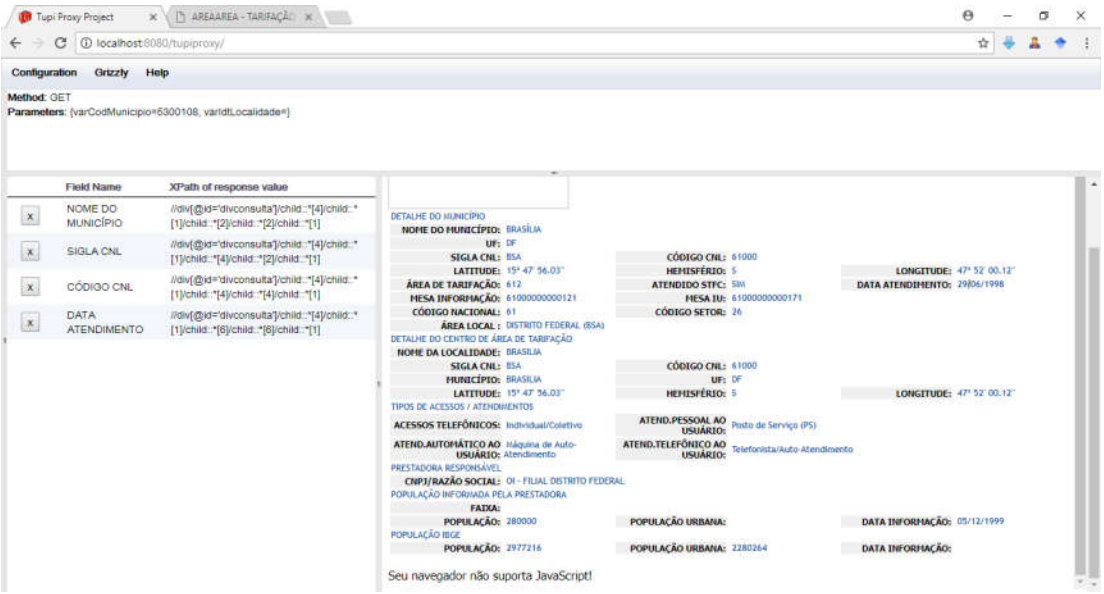


Figura 26 - Seleção de campos de interesse

Após selecionar todos os campos de interesse, passamos para o próximo passo: gerar uma entidade de domínio do negócio. Para isso, selecionamos a página correspondente, clicamos no primeiro botão da barra de botões, e então abrirá uma tela para preenchermos o nome da classe

que representará uma entidade, conforme ilustrado na Figura 27. Para esse exemplo, chamaremos a entidade de Localidade. Automaticamente aparecerão campos com o mesmo nome dos rótulos selecionados anteriormente, em minúsculo.

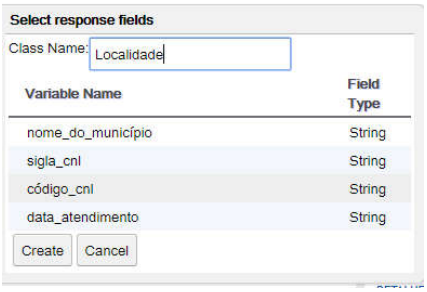


Figura 27 - Tela para criação de uma entidade

Após a criação da entidade, devemos criar o serviço propriamente dito. Para isso, selecionamos a página correspondente e clicamos no segundo botão da barra de botões. Aparecerá uma tela (Figura 28) para informarmos o nome da classe *Java* (que será criada internamente) que representará o respetivo serviço. Neste exemplo usaremos o nome *LocProvider*. Outras informações necessárias são: o nome do método, o tipo de método HTTP, e o tipo de retorno (de acordo com uma entidade criada anteriormente).

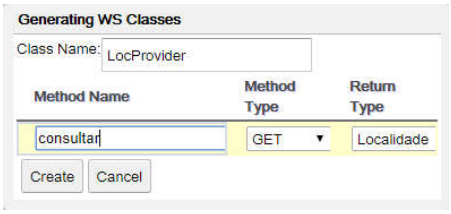


Figura 28 - Tela para criação de um serviço

Após a criação da classe que representará um serviço, é possível visualizar a lista de serviços (ou recursos, quando se trata de serviços RESTful) na opção *Resources.* E, ao clicarmos no serviço *LocProviderResource*, é possível visualizarmos a URL do serviço REST que foi criado.

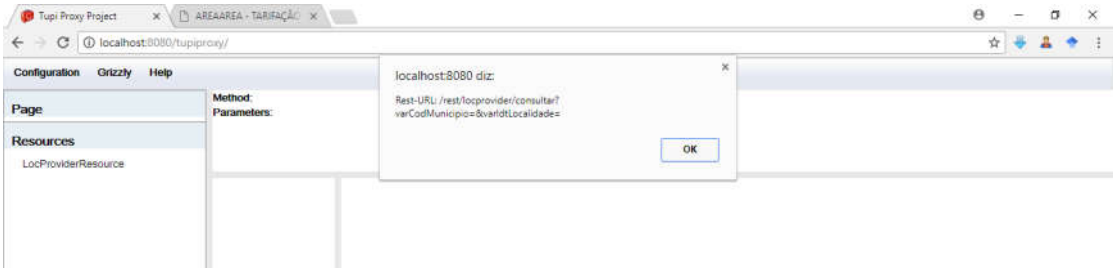


Figura 29 - Lista de Resources

Com essa URL é possível, portanto, testarmos o serviço criado. E, para isso, inicializamos o *Grizzly* (servidor que hospedará os serviços) através do menu: Grizzly / Start. Em seguida

podemos, através do próprio navegador, invocar o serviço, conforme Figura 30. É, então, possível visualizarmos a resposta do serviço *RESTful* com uma formação JSON.

Figura 30 - Chamando o serviço REST

5.1.2 Criação de serviço baseado em pedidos do tipo *POST*

O segundo teste de validação será feito com o seguinte sistema: http://sistemas.anatel.gov.br/satva, menu: SIPTV, opção "Por empresa". Será apresentada uma tela, conforme Figura 31, para informarmos um filtro: empresa. Selecionamos qualquer empresa, apenas para capturarmos as informações que precisamos.

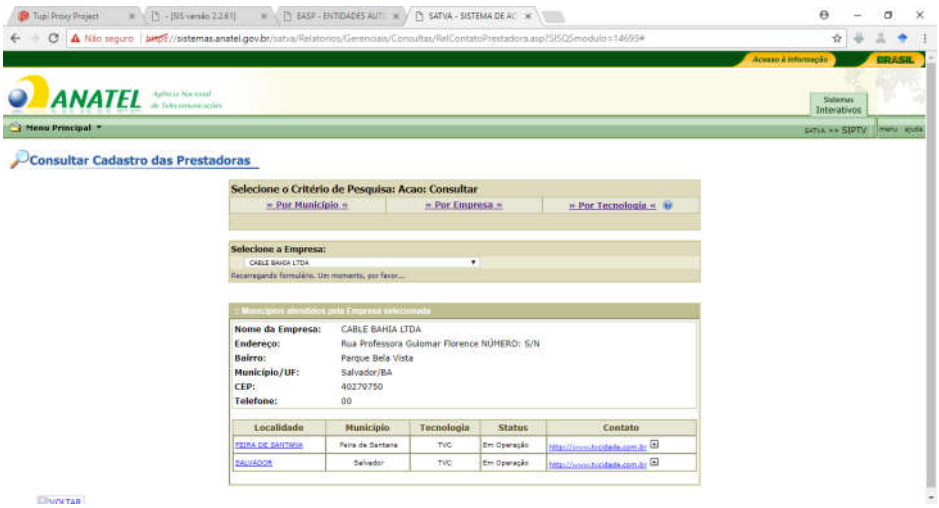


Figura 31 - SATVA Infomações por empresa

No Tupi, selecionamos os campos que desejamos disponibilizar no serviço: nome da empresa, endereço, município, CEP e Telefone, como mostra a Figura 32.

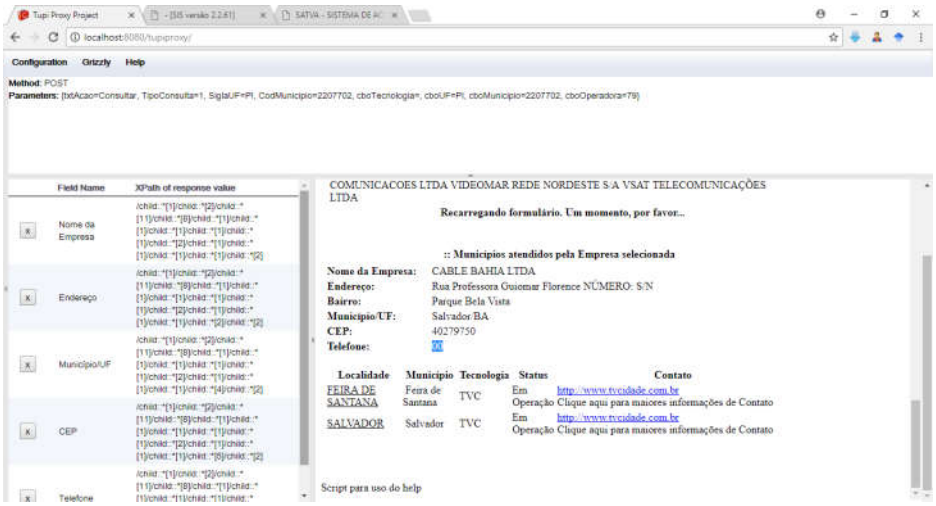


Figura 32 - SATVA Seleção de campos de interesse

Em seguida criamos uma entidade com o nome Empresa e, logo após, uma classe que representará o serviço `EmpresaProvider` (Figura 33).

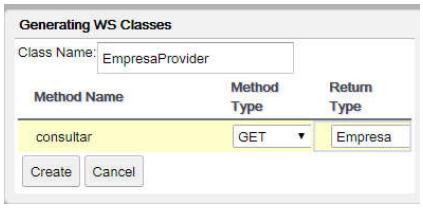


Figura 33 - Serviço EmpresaProvider

Na lista de *Resources* visualiza-se agora o respetivo serviço, como mostra a Figura 34.

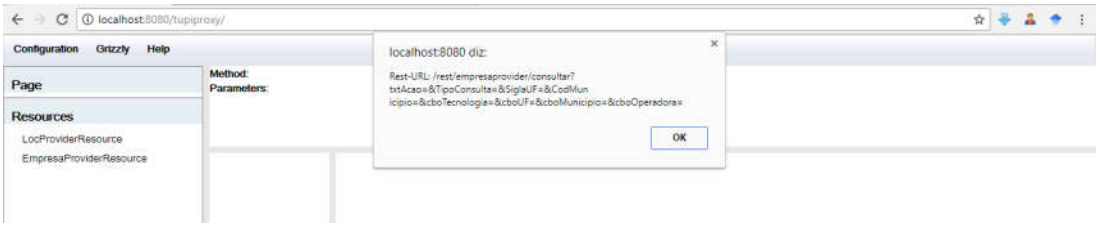


Figura 34 - EmpresaProvider na Lista de recursos

Ao invocar o serviço pelo navegador, podemos testar a sua disponibilidade e a conformidade das informações com a resposta em um formato JSON (Figura 35).

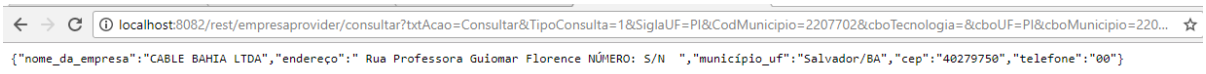


Figura 35 - Chamando o serviço REST do tipo POST

5.2 Usabilidade e satisfação

Com o intuito de avaliar a usabilidade e satisfação da primeira versão do sistema, foi realizada uma pesquisa com seis utilizadores. A quantidade de utilizadores para este teste foi definida de acordo com trabalho de *Nielsen* [25] onde ele conclui que, acima de cinco utilizadores, o custo-benefício já não compensa.

Estes seis utilizadores participantes têm idade por volta dos trinta anos, são analistas de negócio da ANATEL e possuem, portanto, conhecimento do domínio do negócio. Entretanto, não possuem qualquer conhecimento a respeito da arquitetura ou da implementação dos sistemas alvo. Foi enviado a estes utilizadores um manual, com orientações passo-a-passo, sobre como gerar um serviço através do TUPI. Para além disto, foi enviado também uma lista de consultas de sistemas da ANATEL, com acesso público, que serviriam para o teste.

Depois de testar o TUPI, os utilizadores responderam a um questionário de satisfação feito no *Google Forms.* O questionário consistiu em quarenta e sete perguntas, divididas em quatorze grupos. Eram cinco as possíveis respostas às perguntas: nunca, quase nunca, regular, quase sempre e sempre. As perguntas foram agrupadas e o resultado está contido na Figura 36.

O resultado demonstra algumas qualidades como confiabilidade, velocidade e compatibilidade. Entretanto, o resultado da pesquisa demonstra também deficiências nessa primeira versão, nomeadamente: flexibilidade, ajuda aos usuários, feedback, facilidade, prevenção e visibilidade.

Alguns utilizadores, por exemplo, reclamaram que tiveram dificuldades para configurar o *proxy* do navegador. Outros utilizadores reclamaram da visualização da cópia das páginas no Tupi pois, em algumas situações, a página era renderizada sem a formatação original.

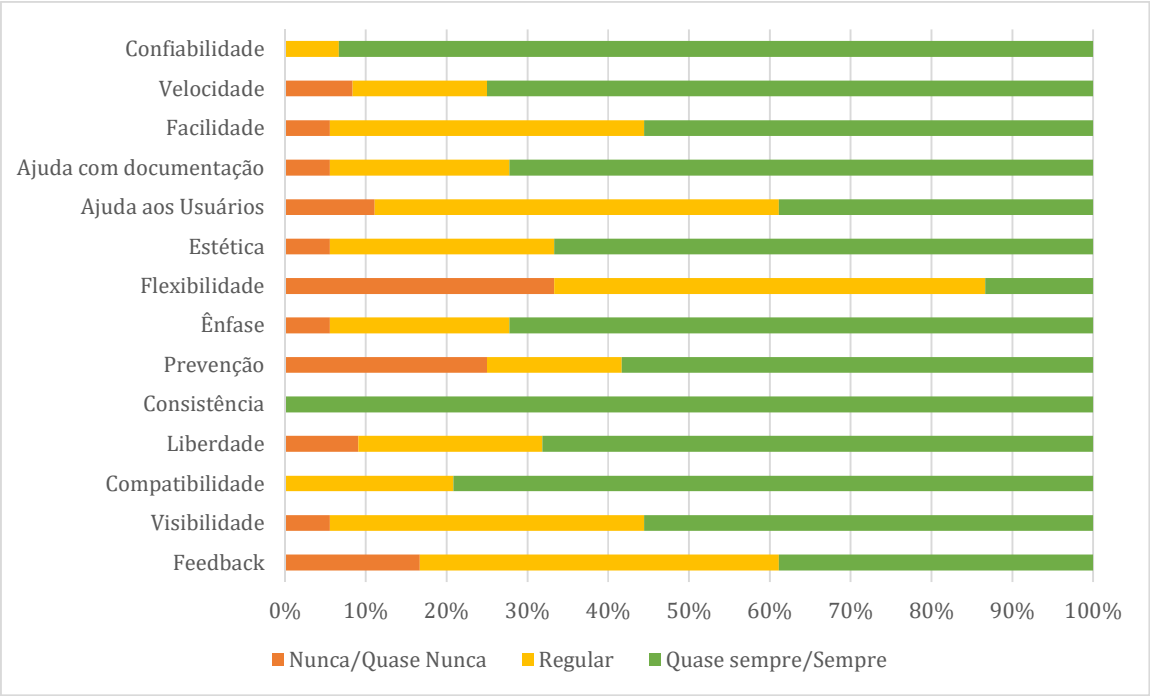


Figura 36 - Resultado da pesquisa

Capítulo 6

Conclusão

6.1 Síntese

Neste trabalho procurou-se viabilizar uma estratégia para reutilizar os sistemas web legados através da criação de uma semântica de *web services*, tornando-os interoperáveis. Desenvolver uma abordagem menos dispendiosa foi o fator motivador deste trabalho.

Com o intuito de demonstrar a viabilidade da abordagem proposta, foi desenvolvido um protótipo de um sistema proposto. O sistema consiste, resumidamente, em uma ferramenta que captura os dados de navegação em um sistema alvo através de um *proxy* HTTP, que disponibiliza uma interface gráfica para o utilizador visualizar e selecionar os dados de interesse que vão compor o *web service*, além de viabilizar a criação e hospedagem dos serviços de forma transparente.

Internamente, o sistema *Tupi* cria algumas classes *Java* que são responsáveis pelo serviço criado. Estes serviços são hospedados e disponibilizados em um servidor (*Grizzly*) que está embutido no Tupi.

Portanto, a contribuição deste trabalho consiste na disponibilização de um sistema que permite criação de uma semântica de *web services* para sistemas *web* legados de forma rápida e menos custosa que outras abordagens como, por exemplo, o desenvolvimento a partir da engenharia reversa de sistemas.

6.2 Trabalhos Futuros

Foi abordado inicialmente neste trabalho somente a conversão de funcionalidades do tipo consultas (pedidos HTTP do tipo GET ou POST) para *web services*. É possível estender esse trabalho para que sejam englobados os outros tipos de funcionalidades de sistemas como, por exemplo, funcionalidades para cadastrar ou atualizar informações (através dos métodos POST ou PUT do HTTP); ou ainda, funcionalidades de excluir informações (através do método DELETE do HTTP).

Questões de segurança também podem fazer parte de evoluções deste trabalho como, por exemplo:

- o tratamento de certificados nas conexões HTTPS, evitando o MITM (*man-in-the-middle*) feito pelo *proxy*;
- um mecanismo que permita criar *web services* para lidar com sistemas de informação que possuem autenticação do utilizador.

Alguns utilizadores testaram o *Tupi* e puderam criar serviços para funcionalidades de alguns sistemas. Depois disso, responderam a um questionário que avaliou sua satisfação. Com os resultados foi possível perceber que o protótipo do sistema possui algumas deficiências em termos de flexibilidade, feedback e ajuda aos utilizadores. Portanto, melhorias no Tupi podem ser feitas para o tornar mais intuitivo.

7 Referências

[1] Siti Rochimah and Alhaji Sheku Sankoh, *"Migration of Existing or Legacy Software Systems into Web Service-based Architectures (Reengineering Process): A Systematic Literature Review*", International Journal of Computer Applications, 2016.

[2] Giusy Di Lorenzo, Anna Rita Fasolino, Lorenzo Melcarne, Porfirio Tramontana and Valeria Vittorini, "*Turning Web Applications into Web Services by Wrapping Techniques*", IEEE, 2007, DOI: 10.1109/WCRE.2007.51

[3] Djelloul Bouchiha and Mimoun Malki, "*Towards Re-engineering Web Applications into Semantic Web Services*", IEEE, 2010. DOI: 10.1109/ICMWI.2010.5648057

[4] W3C Working Group. *Web Services Architecture*. (Feveveiro de 2004). Em 20 de Novembro de 2016, de https://www.w3.org/TR/ws-arch/

[5] Pressman, Roger S. (2011). *Software Engineering: a Practitioner's Approach*, 7th Edition, McGraw-Hill.

[6] XML Web Services. (n.d.). Em 20 de Novembro de 2016, de http://www.w3schools.com/xml/xml_services.asp

[7] Sérgio Nunes, Gabriel David. (2005) *Uma Arquitectura Web para Serviços Web*. Em 15 de Fevereiro de 2017, de https://repositorio-aberto.up.pt/handle/10216/281

[8] W3C Working Group. (2000). *Simple Object Access Protocol (SOAP) 1.1*. Em 25 de Fevereiro de 2017, de https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[9] Fielding, R. T. *Architectural styles and the design of networked-based software architectures*. Dissertação de Doutoramento: Dept. of Information and Computer Science, University of California, Irvine (2000)

[10] Marzullo, Fabio Perez (2009). *SOA na prática: inovando seu negócio por meio de soluções orientadas a serviços*. Novatec Editora.

[11] W3C Working Group. (Abril de 2007). *SOAP Version 1.2 Part 0: Primer (Second Edition)*. Em 25 de Fevereiro de 2017, de https://www.w3.org/TR/soap12-part0

[12] *XML SOAP*. (n.d.). Em 20 de Fevereiro de 2017, de http://www.w3schools.com/xml/xml_soap.asp

[13] Fielding, R. T., Taylor, R. N.: *Principle Design of the Modern Web Architecture ACM Transactions on Internet Technology*. (2002)

[14] Cesare Pautasso, Olaf Zimmermann, Frank Leymann, *RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision*, Proc. of the 17th International World Wide Web Conference (WWW2008), Bejing, China, April 2008

[15] S. Mumbaikar, P. Padiya, et al., "*Web services based on SOAP and REST principles*", International Journal of Scientic and Research Publications, vol. 3, no. 5, 2013.

[16] Smita Patil, Nilesh Marathe, Puja Padiya, "*Use of RESTful Web Services in Distributed Computing*", International Journal of Computer Science and Information Technology & Security, vol.6, no.2, Mar-April 2016.

[17] ALMONAIES, Asil A. et al. "*Towards a framework for migrating web applications to web services*". In: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp., 2011. p. 229-241.

[18] Rodríguez-Echeverría, Roberto, et al. "*Model-driven generation of a REST API from a legacy web application*". International Conference on Web Engineering. Springer International Publishing, 2013. DOI: 10.1007/978-3-319-04244-2_13

[19] W3C Working Group. (Novembro de 1999). *XML Path Language (XPath)*. Em 15 de Março de 2017, de https://www.w3.org/TR/xpath/

[20] Google Web Toolkit Project. (n.d.). *Using GWT RPC*. Em 20 de Março de 2017, de http://www.gwtproject.org/doc/latest/tutorial/RPC.html

[21] Little Proxy Project. (n.d.). Em 20 de Março de 2017, de https://github.com/adamfisk/LittleProxy

[22] Code Model Project. (n.d.). *A Java library for code generators*. Em 25 de Março de 2017, de https://javaee.github.io/jaxb-codemodel/

[23] Project Grizzly. (n.d.). Em 25 de Março de 2017, de https://javaee.github.io/grizzly/

[24] Di Lorenzo, Giusy, et al. *"Turning web applications into web services by wrapping techniques."* Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on. IEEE, 2007.

[25] Jakob Nielsen and Thomas K. Landauer. *A mathematical model of the finding of usability problems*. In Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems, CHI '93, pages 206{213, New York, NY, USA, 1993. ACM. DOI:10.1145/169059.169166.

8 Anexos

8.1 Classe *TupiServiceImpl*

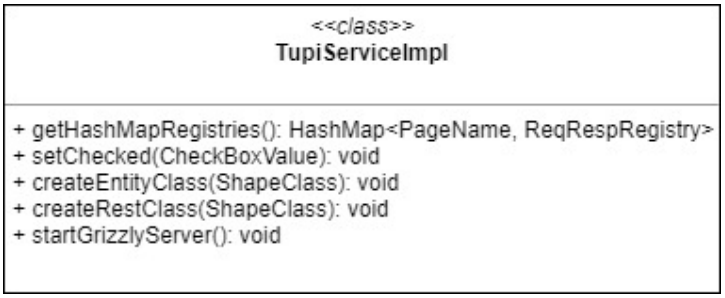


Figura 37 - TupiServiceImpl Class Diagram

O método *getHashMapRegistries* é responsável por retornar para o lado cliente os registros capturados pelo Proxy HTTP no formato de um *HashMap,* possuindo o nome de uma página como chave, e um objeto do tipo *ReqRespRegistry* como valor. Este método será chamado a cada 5 segundos para preencher a lista de páginas existentes no painel 3 da Figura 15.

O método *setChecked* será utilizado para indicar que uma página foi selecionada. Os métodos *createEntityClass* e o *createRestClass* serão responsáveis pela criação das classes *Java* que vão compor o *web service* e são acionados, respetivamente, através dos botões  e  localizados na área 2 da Figura 15. O método *startGrizzlyServer* iniciará o servidor onde ficarão hospedados os serviços.

8.2 Classe *ResponseValuesMapping*

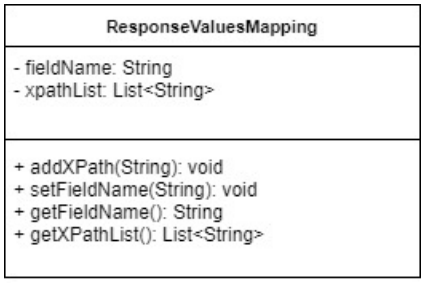


Figura 38 - ResponseValuesMapping class diagram

Cada objeto desta classe possuirá duas propriedades: o nome de um campo e a respetiva expressão XPath que indicará a localização no HTML do valor da resposta.

8.3 Trecho de código do HTML *Parser*

```
…
// documento Html original
Document doc = Jsoup.parse(inputHTML);

// Adicionando novas tags à whitelist relaxed
Whitelist myWhiteList =
Whitelist.relaxed().addTags("input","link","form","label");

// informando quais atributos são permitidos
myWhiteList.addAttributes("div", "id", "name", "style");
myWhiteList.addAttributes("link", "rel", "type", "href");
myWhiteList.addAttributes("td", "class");

// Aplica-se a whitelist ao documento Html
Document saneHtml = new Cleaner(myWhiteList).clean(doc);
…
```

Figura 39 - HTML Parser

O trecho de código acima mostra como foi feita a *whitelist* utilizada para fazer a limpeza do código HTML.

8.4 Diagrama de sequência: criar classe de entidade

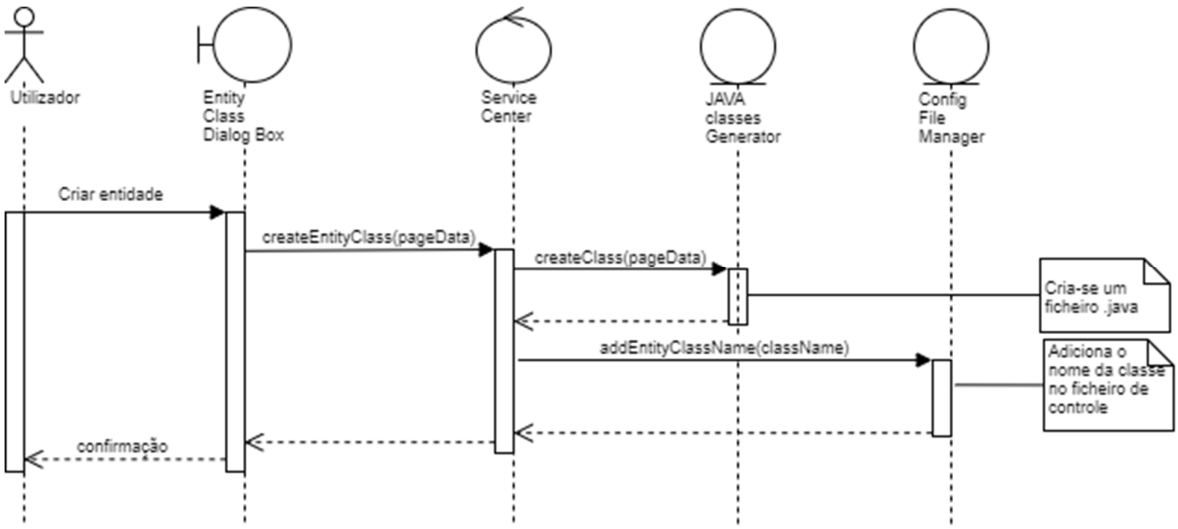


Figura 40 - Diagrama de Sequência: criar classe de entidade

8.5 Diagrama de sequência: criar classe REST

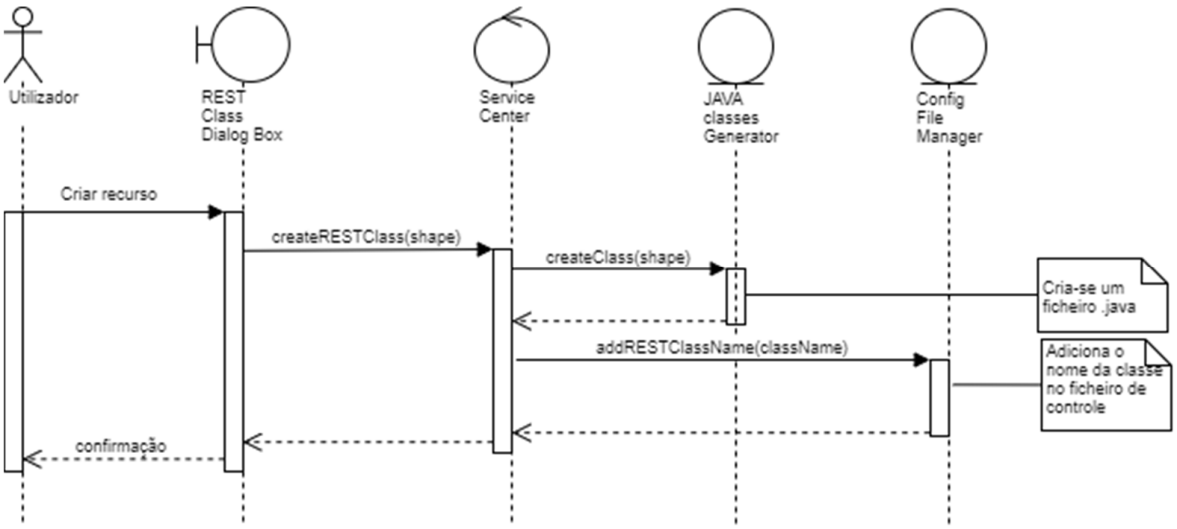


Figura 41 – Diagrama de Sequência: criar classe REST