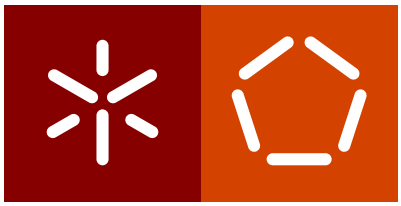


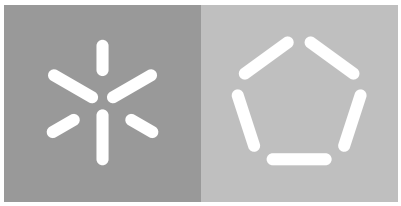
Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Carlos Lima Martins

CLAV:
API de dados e Autenticação

Relatório de Pré-Dissertação

November 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Carlos Lima Martins

CLAV:
API de dados e Autenticação

Relatório de Pré-Dissertação

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
José Carlos Leite Ramalho

November 2019

ABSTRACT

Write abstract here (en)

RESUMO

Escrever aqui resumo (pt)

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

GLOSSÁRIO

Application Programming Interface Interface ou protocolo de comunicação entre um cliente e um servidor i

ontologia Representação de conhecimento (conceitos e as relações entre estes) 2

Simplex Programa de Simplificação Administrativa e Legislativa 2

LISTA DE ACRÓNIMOS

AP Administração Pública 2
API Application Programming Interface 3, *Glossary:* Application Programming Interface

CLAV Classificação e Avaliação da Informação Pública 2, 3
CSV Comma Separated Values 3

DGLAB Direção-Geral do Livro, dos Arquivos e das Bibliotecas 2

HTTP Hypertext Transfer Protocol 7

JSON JavaScript Object Notation 3

LC Lista Consolidada 2, 3

NPM Node Package Manager 7

RDF Resource Description Framework 3
REST Representational State Transfer iii, 3, 7

UM Universidade do Minho 2

XML Extensible Markup Language 3

1

INTRODUÇÃO

Vemos atualmente a mudança de paradigma em várias organizações e governos em relação a políticas e estratégias para a disponibilização de dados abertos nos domínios das ciências e da Administração Pública. Quanto à Administração Pública portuguesa têm sido promovidas políticas para a sua transformação digital com o objetivo de otimização de processos, a modernização de procedimentos administrativos e a redução de papel. De certa forma a agilização de procedimentos da Administração Pública portuguesa. IX (2019)

De forma a alcançar estes objetivos a Administração Pública (AP) tem desmaterializado processos e tem promovido a adoção de sistemas de gestão documental eletrónica bem como da digitalização de documentos destinados a serem arquivados. IX (2019)

Por forma a continuar esta transformação da AP a Direção-Geral do Livro, dos Arquivos e das Bibliotecas (DGLAB) apresentou a iniciativa da Lista Consolidada (LC) para a classificação e avaliação da informação pública. A LC serve de referencial para a construção normalizada dos planos de classificação e tabelas de seleção das entidades que executam funções do Estado. IX (2019)

Nasce assim o projeto Classificação e Avaliação da Informação Pública (CLAV) com um dos seus objetivos primordiais a operacionalização da utilização da LC, numa colaboração entre a DGLAB e a Universidade do Minho (UM) e financiado pelo Simplex. IX (2019)

A plataforma CLAV disponibiliza em formato aberto uma ontologia com as funções e processos de negócio das entidades que exercem funções públicas (ou seja a LC) associadas a um catálogo de legislação e de organismos. Desta forma, a CLAV viabiliza a desmaterialização dos procedimentos associados à elaboração de tabelas de seleção tendo como base a LC e ao controlo de eliminação e arquivamento da informação pública através da integração das tabelas de seleção nos sistemas de informação das entidades públicas alertando-as quando determinado documento deve ser arquivado ou eliminado. Esta integração promove também a interoperabilidade através da utilização de uma linguagem comum (a LC) usada no registo, na classificação e na avaliação da informação pública. IX (2019)

1.1 OBJETIVOS

A continuação do desenvolvimento da API de dados da CLAV nesta dissertação, seguindo uma metodologia REST, permite a processos ou aplicações aceder aos dados sem a intervenção humana para além de suportar a plataforma CLAV. Um dos objetivos da API de dados é permitir futuramente a criação de novas aplicações através desta. Como tal, é extrememente essencial que a API de dados do CLAV possua uma boa documentação ajudando futuros programadores ou utilizadores a utilizar a API. Advém daí a necessidade de nesta dissertação realizar a documentação da API de dados em *Swagger*.

Apesar de o projeto ter em mente a disponibilização aberta de informação pública é necessário controlar a adição, edição e eliminação da informação presente na Lista Consolidada, bem como a informação de utilizadores, da legislação, das entidades, etc, mantendo-a consistente e correta. É, portanto, necessário controlar os acessos à API de dados com múltiplos níveis de acesso restringindo as operações que cada utilizador pode realizar consoante o seu nível. Desta forma garante-se que apenas pessoal autorizado pode realizar modificações aos dados.

Este controlo de acesso exige a existência de formas de autenticação. Como um cofre para o qual ninguém tem a chave não é útil pelo facto de que algo lá guardado ficará eternamente inacessível, também algo com controlo de acesso seria inútil caso não fosse possível ultrapassar esse controlo de alguma forma. Assim, uma das formas de autenticação usadas, Autenticação.gov, criada pelo Estado português, permite a autenticação dos cidadãos portugueses nos vários serviços públicos AMA (2019) entre os quais, a Segurança Social, o Serviço Nacional de Saúde e a Autoridade Tributária Aduaneira. Sendo este um projeto do Governo Português, pretende-se que seja possível a autenticação no CLAV através do Autenticação.gov.

Por forma a contrariar o aumento da complexidade da API de dados com a adição do controlo de acesso e da autenticação pretende-se investigar se a criação de um API Gateway simplifica a comunicação entre interface/utilizadores e a API de dados.

Resumidamente, os objetivos desta dissertação são:

- Documentação em *Swagger* da API de dados da CLAV
- Adição de formatos de exportação à API de dados da CLAV (para além do já presente JSON, adicionar CSV, XML e RDF)
- (Continuação da) Integração do Autenticação.gov na CLAV
- Proteção da API de dados da CLAV com múltiplos níveis de acesso
- Estudo da crição de um API Gateway
- Integração do CLAV no iAP

2

ESTADO DA ARTE

2.1 JAVASCRIPT E A PROGRAMAÇÃO ASSÍNCRONA

A linguagem *JavaScript* teve a sua origem no *browser Netscape Navigator* como uma forma de adicionar programas a páginas web. Haverbeke (2018) Hoje em dia é usado por grande parte dos *browsers* e tornou possível as aplicações web onde o utilizador pode interagir sem precisar de realizar *refresh* ao fim de cada ação. Contudo o *Javascript* é bastante liberal no que permite o programador escrever, facilitando a aprendizagem para novos programadores mas tornando a tarefa de resolver e encontrar problemas bem mais difícil visto não apontar aonde estão esses problemas. Ainda assim esta flexibilidade permite uma quantidade de técnicas que não são possíveis em linguagens mais rígidas e que permitem ultrapassar algumas falhas do *Javascript*. Haverbeke (2018)

2.1.1 *Programação Assíncrona*

Muitos programas em *JavaScript* precisam de por exemplo obter dados através da rede ou a partir do disco. Estes casos são muito mais lentos do que obter os dados a partir de memória para posterior processamento pelo processador. Na realização de tais pedidos o programa em *Javascript* perderia o acesso ao processador, passando este acesso para outro programa que esteja a correr. Só depois de receber o sinal de que o pedido foi efetuado e de ter de novo acesso ao processador é que o programa voltaria a continuar o seu trabalho.

Por forma a evitar perder o acesso do processador em tais casos, e assim realizar trabalho enquanto se espera pela resposta do pedido, o *Javascript* segue um modelo assíncrono. O modelo assíncrono permite que aconteçam várias coisas ao mesmo tempo; quando começa um pedido o programa continua a correr; quando o pedido termina, o programa é informado e tem acesso ao resultado. Haverbeke (2018) Para além de não se perder imediatamente o acesso ao processor, o uso de programação assíncrona permite receber e enviar dados de e para múltiplos dispositivos ao mesmo tempo sem complicar a gestão de *threads* e a sincronização necessária em tais casos. Ao usar um modelo assíncrono a expressão de

programas que não seguem um modelo de controlo linear é mais fácil enquanto que torna mais difícil aqueles que o seguem. Haverbeke (2018)

Callbacks

Uma abordagem de programação assíncrona é a adição de um argumento extra, uma função *callback*, nas funções que realizam pedidos lentos. Haverbeke (2018) Quando o pedido concluir a função *callback* será chamada tendo como argumento o resultado do pedido. Com o uso de callbacks o nível de indentação aumenta com cada pedido assíncrono o que em alguns casos pode tornar o código um pouco difícil de compreender, principalmente nos casos com múltiplos pedidos assíncronos seguidos que têm de ser sequenciais. Para além disso, qualquer função que chame uma função assíncrona tem de ser ela própria assíncrona. Não é recomendado a reestruturação de grandes quantidades de código através de *callbacks* visto ser mais propenso a erros do que retornar apenas um valor.

```
almocar("comida", function(dentes_sujos){
    dentes_limpos = lavar_dentes(dentes_sujos)
})
```

Listing 2.1: Exemplo de uma *Callback*

Promessas

Uma outra abordagem, em vez de passar uma função a ser chamada no futuro, é devolver um objeto que represente este evento futuro, uma promessa. Ou seja, uma promessa é um pedido assíncrono que pode ser concluído no futuro e produzir um valor, tendo a capacidade de notificar qualquer interessado quando o valor estiver disponível.Haverbeke (2018) O resultado de uma promessa tanto pode já estar pronto ou estar apenas daqui a algum tempo. A principal vantagem das promessas é que simplificam o uso de funções assíncronas visto que não é necessário passar uma função *callback*. Como tal estas funções são similares às restantes mas com uma pequena diferença, o resultado da função pode ainda não estar disponível.

```
almocar("comida")
    .then(dentes_sujos => dentes_limpos = lavar_dentes(dentes_sujos))
```

Listing 2.2: Exemplo de uma Promessa

Exceções

Durante a execução de um pedido assíncrono podem ocorrer exceções seja por um erro ou porque por exemplo ocorreu um *time out* do pedido, este último acontece essencialmente

quando se realiza pedidos através da rede. Estas exceções precisam de ser tratadas por forma a que o programa que estámos a desenvolver não "expluda" deixando de funcionar. Esta tratamento não é simples de se realizar quando se usa *callbacks* enquanto que no caso das promessas basta o uso de um *catch*. No caso das *callbacks* o pedido assíncrono teria de devolver dois valores em vez de um, o primeiro com o erro em caso de insucesso e o segundo com o resultado em caso de sucesso o que obrigaria a função *callback* a verificar se não recebeu uma exceção.

```
almocar("comida", function(erro, dentes_sujos){
    if(!erro){
        dentes_limpos = lavar_dentes(dentes_sujos)
    }else{
        console.log("Falta pasta de dentes")
    }
})
```

Listing 2.3: Exemplo de uma *Callback* com captura de exceções

```
almocar("comida")
    .then(dentes_sujos => dentes_limpos = lavar_dentes(dentes_sujos))
    .catch(erro => console.log("Falta pasta de dentes"))
```

Listing 2.4: Exemplo de uma Promessa com captura das exceções

Funções async

Dentro de funções *async* é possível escrever código pseudo-síncrono por forma a descrever código assíncrono com recurso ao *await* que espera pela conclusão de uma promessa antes de avançar com a execução do resto da função, ficando o código com um aspeto semelhante ao síncrono. As funções *async* retornam implicitamente uma promessa e enquanto estão à espera duma promessa (*await*) ficam congeladas. Serão resumidas mais tarde quando a promessa estiver concluída.

```
async function almoco(){
    try{
        dentes_sujos = await almocar("comida")
        dentes_limpos = lavar_dentes(dentes_sujos)
    }catch(erro){
        console.log("Falta pasta de dentes")
    }
    return "Almocei!"
}
```

Listing 2.5: Exemplo de uma função *async*

2.2 NODE.JS

Node.js é um ambiente de execução que permite usar *JavaScript* fora do contexto de um *browser* permitindo construir desde ferramentas de linha de comandos até servidores HTTP. Foi originalmente desenhado de forma a ter o papel de nodo numa rede. O *JavaScript* não tem embutido a capacidade de entrada e saída de dados sendo suprimida esta necessidade com o uso do *Node.js*.

Quando instalado num sistema, o *Node.js* permite executar ficheiros *JavaScript* através do comando `node`. Se este comando for executado sem indicar um ficheiro apresenta uma consola interativa onde se pode introduzir código *JavaScript*, executá-lo e obter o resultado.

Através do NPM, mais precisamente do comando `npm`, é possível instalar, no nosso projeto pessoal, os módulos *JavaScript* disponíveis no repositório *online*. Caso exista um ficheiro *package.json* no projeto é mantido neste os módulos já instalados e as suas versões bem como meta informação do projeto.

Em *Node.js* estão disponíveis os *bindings* globais `process` e `console` que permitem inspecionar e manipular o programa atual. Para além destes, também estão diponíveis os *bindings* padrão do *JavaScript* exceto aqueles relacionados com funcionalidades do *browser*, como o `document`.

O *Node.js* tem algums módulos embutidos, entre os quais o módulo `fs` (*file system*) que exporta funções que permitem trabalhar com ficheiros e directorias. Tem também o módulo `http` que exporta funções que permitem correr servidores HTTP ou fazer pedidos HTTP.

Todo o input e output no *Node.js* é realizado de forma assíncrona a menos que se use uma variante síncrona de uma determinada função. Haverbeke (2018)

2.3 REST

Richardson and Ruby (2007)

2.4 EXPRESS

Satheesh et al. (2015)

2.5 passport

2.6 jsonwebtoken

2.7 passport-jwt

2.8 cors

2.9 axios

2.10 http status

2.11 headers do http

2.12 autenticação.gov

AMA (2018)

2.13 exceljs

2.14 mongodb

Satheesh et al. (2015)

2.15 mongoose

2.16 web semântica

DuCharme (2011)

2.16.1 *RDF*

DuCharme (2011)

2.16.2 *SPARQL*

DuCharme (2011)

2.17 GRAPHDB

2.18 SWAGGER

2.19 SWAGGER-UI

2.20 YAML-INCLUDE

2.21 SWAGGER-UI-EXPRESS

2.22 JS-YAML

2.23 NGINX

DeJonghe (2018)

2.24 ONTOLOGIA

Arp et al. (2015)

2.25 DOCKER

Mouat (2015)

2.26 DOCKER COMPOSE

Mouat (2015)

2.27 LISTA CONSOLIDADA

2.28 TABELAS DE SELEÇÃO

2.29 CACHE E FECHO TRANSITIVO

3

O PROBLEMA E OS SEUS DESAFIOS

4

CONCLUSÃO

BIBLIOGRAFIA

AMA. *Autenticação.gov - Fornecedor de autenticação da Administração Pública Portuguesa*, 1.5.1 edition, 12 2018.

AMA. Autenticação.gov, 2019. URL https://autenticacao.gov.pt/fa/Default.aspx. Acedido a 2019-11-20.

Robert Arp, Barry Smith, and Andrew Spear. *Building Ontologies with Basic Formal Ontology*. MIT Press, 1st edition, 7 2015. ISBN 978-0-262-52781-1.

Derek DeJonghe. *NGINX Cookbook*. O'Reilly, 1st edition, 11 2018. ISBN 978-1-491-96893-2. Second Release.

Bob DuCharme. *Learning SPARQL*. O'Reilly, 1st edition, 7 2011. ISBN 978-1-449-30659-5.

Marijn Haverbeke. *Eloquent JavaScript, 3rd Edition: A Modern Introduction to Programming*. No Starch Press, 3rd edition, 12 2018. ISBN 978-1-59327-950-9.

Plataforma CLAV: contributo para a disponibilização de dados abertos da Administração Pública em Portugal, 7 2019. IX Encontro Ibérico EDICIC. URL http://hdl.handle.net/10760/38643. Acedido a 2019-11-20.

Adrian Mouat. *Using Docker*. O'Reilly, 1st edition, 12 2015. ISBN 978-1-491-91576-9.

Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 1st edition, 5 2007. ISBN 978-0-596-52926-0.

Mithun Satheesh, Mithun Satheesh, and Jason Krol. *Web Development with MongoDB and NodeJS*. Packt Publishing, 2nd edition, 10 2015. ISBN 978-1-78528-752-7.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.