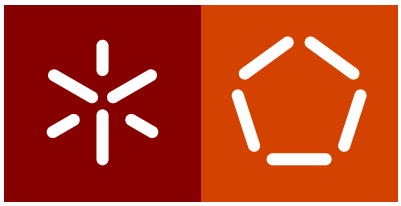


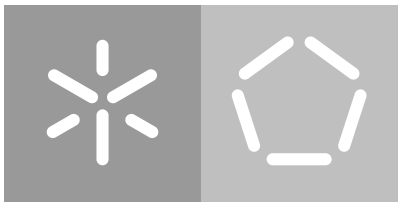
**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

José Carlos Lima Martins

# CLAV:
# API de dados e Autenticação

## Relatório de Pré-Dissertação

Janeiro 2020



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

José Carlos Lima Martins

# CLAV:
# API de dados e Autenticação

## Relatório de Pré-Dissertação

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação realizada sob a orientação do Professor
**José Carlos Leite Ramalho**

Janeiro 2020

A B S T R A C T

Write abstract here (en)

# RESUMO

Escrever aqui resumo (pt)

# CONTEÚDO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE EXEMPLOS

# GLOSSÁRIO

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

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

**Simplex** Programa de Simplificação Administrativa e Legislativa 2

# LISTA DE ACRÓNIMOS

**AEAD** Authenticated Encryption with Additional Data 14
**AMA** Agência para a Modernização Administrativa 22
**AP** Administração Pública 2, 20
**API** Application Programming Interface i, iii–vi, 2–8, 14–20, 23–25, 29–35, 40, 41, *Glossary:* Application Programming Interface

**BD** Base de Dados 40

**CC** Cartão de Cidadão 5, 7, 20, 21
**CLAV** Classificação e Avaliação da Informação Pública iii, iv, 2–7, 9, 14, 20–23, 32–35, 40, 41
**CMD** Chave Móvel Digital 20, 21
**CSS** Cascading Style Sheets 4, 23
**CSV** Comma Separated Values vi, 3, 35, 39

**DGLAB** Direção-Geral do Livro, dos Arquivos e das Bibliotecas 2, 6, 7

**HMAC** Hash-based Message Authentication Code 9, 10, 13, 14
**HTML** Hypertext Markup Language vi, 4, 23, 28
**HTTP** Hypertext Transfer Protocol 15, 27–29, 34, 40

**IETF** Internet Engineering Task Force 14

**JOSE** JSON Object Signing and Encryption 9, 10
**JSON** JavaScript Object Notation iii, vi, viii, 3, 8–14, 24, 30, 33, 35, 37–40
**JSON-LD** JavaScript Object Notation (JSON) for Linked Data 35, 40
**JWE** JSON Web Encryption 8, 9, 14
**JWS** JSON Web Signature iii, iv, 8, 9, 11–14
**JWT** JSON Web Token iii, iv, vi, 7–15

**LC** Lista Consolidada 2, 3

**MIME** Multipurpose Internet Mail Extensions 40

**N3** Notation3 40
**NIC** Número de Identificação Civil 6, 21
**NSA** National Security Agency 10

**OAS** OpenAPI Specification 23
**OASIS** Organization for the Advancement of Structured Information Standards 22

**OCSP** Online Certificate Status Protocol 21

**PDF** Portable Document Format 4
**PIN** Personal Identification Number 20, 21
**PKI** Public Key Infrastucture 20, 21
**POSIX** Portable Operating System Interface 11

**RAML** RESTful API Modeling Language 29–31
**RDF** Resource Description Framework i, 3, 35, 40, 41
**REST** Representational State Transfer i, 2, 23, 24, 30, 32, 40
**RSA** Rivest–Shamir–Adleman 9, 10

**SAIL** Storage And Inference Layer 40
**SAML** Security Assertion Markup Language iii, 13, 14, 22
**SDK** Software Development Kit 23
**SHA-2** Secure Hash Algorithm 2 10
**SMS** Short Message Service 21
**SPARQL** SPARQL Protocol and RDF Query Language i, 40
**SSO** Single Sign On 22
**SWT** Simple Web Token 13

**Turtle** Terse RDF Triple Language 35, 40

**UI** User Interface iii, iv, 23, 24, 30–33, 35
**UM** Universidade do Minho 2

**W3C** World Wide Web Consortium 40

**XML** Extensible Markup Language vi, 3, 13, 14, 22, 28, 35, 37–40

**YAML** YAML Ain't Markup Language i, 23, 24, 30, 33, 35

# 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. [6]

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. [6]

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. [6]

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. [6]

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. [6]

## 1.1 MOTIVAÇÃO

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. Além disso, uma API sem uma boa documentação de como a usar é inútil. 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 [2] entre os quais, a Segurança Social, o Serviço Nacional de Saúde e a Autoridade Tributária Aduaneira. Sendo este um projeto do Governo Português, a autenticação no CLAV através do Autenticação.gov é um requisito.

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.

## 1.2 OBJETIVOS

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 ESTADO DA ARTE DO CLAV

Quando esta dissertação teve início o projeto CLAV já tinha cerca de 2 anos de desenvolvimento. Assim nesta secção será apresentado o estado da arte do CLAV quando esta dissertação iniciou aprofundando principalmente os pontos mais importantes sobre o tema desta dissertação.

### 2.1.1 *Estrutura*

O CLAV está dividido em duas partes:

- interface (*front-end*) presente em `http://clav.dglab.gov.pt`
- API de dados (*back-end* que inclui também duas bases de dados, *GraphDB* e *MongoDB*) presente em `http://clav-api.dglab.gov.pt`.

Cada parte encontra-se numa máquina diferente.

Através da figura 1 é possível ver o possível fluxo tanto de um utilizador a aceder à interface como a de um utilizador a aceder diretamente à API de dados. No primeiro caso, quando um utilizador acede o servidor da interface do CLAV é descarregado para o lado do utilizador o ficheiro HTML (*index*) e os vários ficheiros *JavaScript*, CSS e *assets* (como imagens, PDFs, etc) quando necessários. O servidor da interface é nada mais que um servidor *web* com recurso ao *Nginx* que hospeda estes ficheiros, os quais representam a interface construída com o *Vue* e o *Vuetify*. Como tal o código apresenta-se todo do lado do utilizador e os pedidos à API serão feitos do computador do utilizador para o servidor da API de dados e não do servidor da interface para o servidor da API de dados. Ou seja, o fluxo de cada um desses pedidos será igual ao fluxo no caso em que se acede diretamente a API sem uso de qualquer interface.

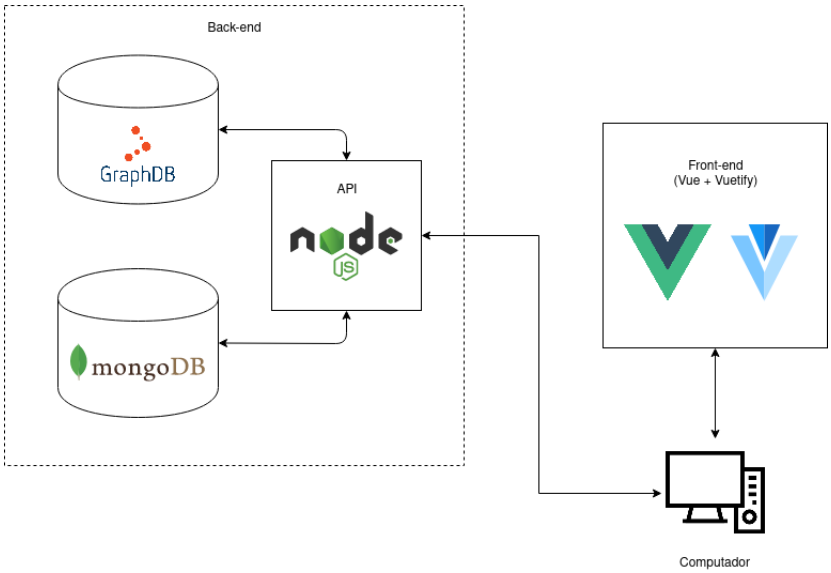



Figura 1: Estrutura do CLAV incluíndo a interação de um utilizador com a mesma

### 2.1.2 *Formas de autenticação*

A API de dados e a interface estavam inicialmente "juntas" (aplicação monolítica) onde as rotas eram protegidas contudo, com a separação da aplicação em duas partes, ambas partes deixaram de estar protegidas. Devido à plataforma já ter estado protegida esta já possui duas formas de autenticação, através de chaves API ou através de utilizadores registados. Ou seja, tanto o registo de utilizadores e de chaves API já se encontra implementado bem como o *login* de utilizadores.

As chaves API existem por forma a dar acesso a certas rotas da API a aplicações que interajam com a mesma (por exemplo sistemas de informação) sem a necessidade de interação humana.

Já os utilizadores possuem múltiplos níveis de acesso sendo que consoante o seu nível podem ou não aceder a uma rota da interface ou da API. Os utilizadores podem autenticarem-se através de *email* e *password* ou com recurso ao Cartão de Cidadão (CC) através do Autenticação.gov, este último apenas disponível através da interface do CLAV.

A hierarquia dos níveis de acesso, do nível que permite menor para o maior acesso, é a seguinte:

- Nível 0: Chaves API
- Nível 1: Representante Entidade
- Nível 2: Utilizador Simples
- Nível 3: Utilizador Avançado
- Nível 3.5: Utilizador Validador (AD)

- Nível 4: Utilizador Validador
- Nível 5: Utilizador Decisor
- Nível 6: Administrador de Perfil Funcional
- Nível 7: Administrador de Perfil Tecnológico

As chaves API poderão aceder a algumas rotas com método `GET`. Já os utilizadores poderão realizar todos os pedidos que as chaves API podem realizar mas quanto maior o seu nível de acesso mais rotas poderão aceder.

A proteção da API terá de ter esta hierarquia em conta.

*Registo*

Como já referido tanto o registo de chaves API como de utilizadores já se encontra implementado.

Para o registo de uma chave API é necessário providenciar um nome, um email e a entidade a que pertence. Após o registo da chave a informação desta chave API é mantida numa base de dados *MongoDB*.

Um utilizador pode se registar através de `email + password` ou através do Autenticação.gov. No primeiro caso, ao se registar necessita obviamente de indicar o seu email, a *password*, o seu nome, a entidade a que pertence e o nível de acesso que pretende. Já no caso do Autenticação.gov para o registo do utilizador é necessário todos os campos anteriores exceto a *password* (pode ser depois definida), sendo também necessário o campo Número de Identificação Civil (NIC) do utilizador. Caso o registo seja efetuado com recurso à interface do Autenticação.gov apenas será necessário indicar o email, a entidade a que pertence e o nível de acesso que pretende visto que os restantes campos são fornecidos pela Autenticação.gov quando o utilizador se autentica e autoriza nesta a partilha dessa informação com a plataforma do CLAV. A *password* é armazenada não na sua forma literal mas sim a sua *hash* ao aplicar a função criptográfica `bcrypt`. A utilização de funções de *hash* criptográficas ao armazenar *passwords* impede que as *passwords* originais se saibam caso a base de dados seja comprometida. Para além disso, como o `bcrypt` combina um valor aleatório (`salt`) com a *password* do utilizador, é impossível pré-computar a *password* que deu origem ao *hash* sem saber o `salt`[1].

Durante esta tese com a proteção da API ficará apenas possível o registo de utilizadores através de utilizadores que já estejam registados e possuam um nível de acesso suficiente para registar utilizadores. Estes utilizadores registados e autorizados pertencem à entidade DGLAB. Portanto por forma a utilizadores representantes de outras entidades se registarem na plataforma terão de: [4]

- Preencher o formulário disponibilizado para o efeito, para cada representante designado pela entidade;

[1] Para mais informação veja *rainbow table attack*

- O formulário deverá ser assinado por um dirigente superior da Entidade e autenticado com assinatura digital, se o envio for feito por via eletrónica (NB: não serão aceites assinaturas do formulário por dirigentes intermédios). Esta autorização autenticada pelo dirigente superior é o equivalente a uma delegação de competências, uma vez que o representante da entidade passa a ter capacidade para, em nome da entidade, submeter autos de eliminação, propostas de tabelas de seleção e novas classes para a Lista Consolidada;
- O formulário deverá ser remetido à DGLAB por via postal ou eletrónica, respetivamente, para:
  - DGLAB, Edifício da Torre do Tombo, Alameda da Universidade, 1649–010 Lisboa (formulário assinado manualmente) ou
  - clav@dglab.gov.pt (formulário com assinatura digital).
- Após receção do formulário, a DGLAB efetuará o(s) respetivo(s) registo(s) até 48 horas úteis;
- Findo esse prazo, o utilizador poderá aceder à plataforma, selecionando a opção "Autenticação";
- A autenticação, no primeiro acesso, deve ser efetuada com o Cartão de Cidadão.

*Login*

O *login* apenas está presente para o caso dos utilizadores visto que assim que uma chave API é registada é enviado por email um JWT com a duração de 30 dias a ser usado nos pedidos a realizar à API. O utilizador poderá ao fim dos 30 dias renovar a sua chave API, onde é gerado um novo JWT.

Portanto do lado dos utilizadores é possível como já referido realizar o *login* de duas formas através de uma estratégia local ou através do Autenticação.gov.

A estratégia local (`email + password`) é conseguida através do uso do *middleware Passport*. O *Passport* é um middleware de autenticação para *Node.js* que tem como objetivo autenticar pedidos. [10] Tem como única preocupação a autenticação delegando qualquer outra funcionalidade para a aplicação que a usa. Este *middleware* possui muitas estratégias de autenticação entre as quais a local (`email/username + password`), JWT, *OAuth*[2], *Facebook* ou *Twitter*. Cada estratégia está num módulo independente. Assim as aplicações que usam o *Passport* não terão um peso adicional devido a estratégias que nem sequer usam.

No caso do *login* através do Autenticação.gov, o utilizador tem de se autenticar na interface do Autenticação.gov (a partir do botão disponível na área de autenticação da interface do CLAV). O fluxo do *login* neste caso é:

[2]Protocolo *open-source* com o objetivo de permitir a autenticação simples, segura e padrão entre aplicações móveis, *web* e *desktop*

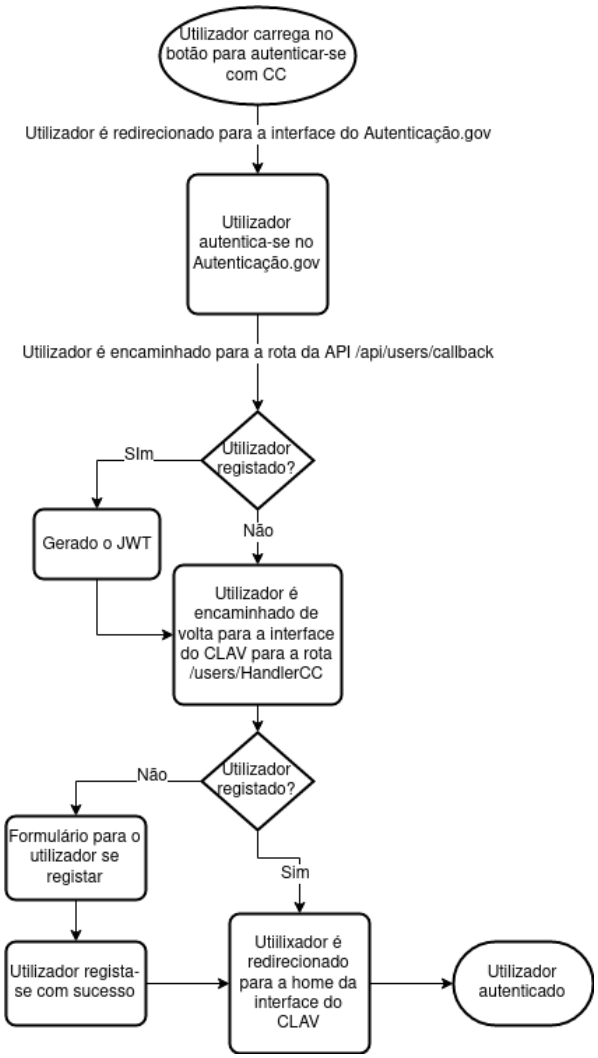



Figura 2: Fluxo do *login* de um utilizador através do Autenticação.gov

No *login* do utilizador é gerado um JWT com a duração de 8 horas que deve ser usado nos pedidos a realizar à API. No fim das 8 horas o utilizador necessita de se autenticar de novo.

## 2.2 JSON Web Token (JWT)

O JWT é um *open standard*[3] que define uma forma compacta e independente de transmitir com segurança informação entre partes com um objeto JSON. [3] O JWT pode ser assinado digitalmente (JWS), encriptado (JWE), assinado e depois encriptado (JWS encriptado, ou seja, um JWE, ordem recomendada[4]) ou encriptado e depois assinado (JWE assinado, ou seja, um JWS).

[3]Mais informação em `https://tools.ietf.org/html/rfc7519`
[4]Mais informação em `https://tools.ietf.org/html/rfc7519#section-11.2`

Caso seja assinado digitalmente é possível verificar a integridade da informação mas não é garantida a sua privacidade contudo podemos confiar na informação do JWT. A assinatura pode ser efetuada através de um segredo usando por exemplo o algoritmo HMAC ou através de pares de chaves pública/privada usando por exemplo o algoritmo RSA. No caso de se usar pares de chaves pública/privada a assinatura também garante que a parte envolvida que tem a chave privada é aquela que assinou o JWT.

Por outro lado, os JWTs podem ser encriptados garantindo a privacidade destes, escondendo a informação das partes não envolvidas. Nesta secção apenas se falará sobre JWTs e JWSs (JWT assinado). Se pretender saber mais sobre JWEs pode ler o capítulo 5 do livro *The JWT Handbook* por *Sebastián E. Peyrott*.

Sendo assim em que casos é útil o uso de JWTs? Dois dos casos são os seguintes:

- Autorização: Este será o caso para o qual o JWT será usado na CLAV. Quando o utilizador realiza o *login* gera-se um JWT por forma a que os restantes pedidos desse utilizador sejam realizados com esse JWT (*Single Sign On*). O uso de JWTs para estes casos permitem um *overhead* pequeno e a flexibilidade de serem usados em diferentes domínios.

- Troca de informação: No caso de troca de informação entre duas partes os JWTs assinados são de bastante utilidade visto que permitem verificar se o conteúdo não foi violado e, no caso de se usar pares de chaves pública/privada para assinar, permitem ter a certeza que o remetente é quem diz ser.

### 2.2.1 *Estrutura do JWT*

Os JWTs são construídos a partir de três elementos, o *header* (objeto JSON também conhecido por JOSE *header*), o *payload* (objeto JSON) e os dados de assinatura/encriptação (depende do algoritmo usado). Estes elementos são depois codificados em representações compactas (`Base64 URL-safe`[5]). As codificações `Base64 URL-safe` de cada elemento são depois concatenadas através de pontos dando origem a uma representação final compacta do JWT (*JWS/JWE Compact Serialization*). Na secção 2.2.2 está presente dois diagramas referentes à construção de dois JWTs sendo um deles assinado.

**eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.**
**eyJuYW1lIjoiSm9zw6kgTWFydGlucyIsIm51bSI6ImE3ODgyMSJ9.**
**tRPSYVsFI-nziRPuAjdGZLN2tUez5MtLML_aAnPplgM**

Figura 3: Exemplo de representação compacta de JWT (quebra de linhas por forma a melhorar leitura)

De seguida vamos aprofundar cada elemento referido:

[5]Variante da codificação `Base64` onde a codificação gerada é segura para ser usada em *URL*s. Basicamente para a codificação `Base64` gerada substitui os caracteres '+' e '/' pelos caracteres '-' e '_' respetivamente. Além disso, remove o caracter de *padding* e proibe separadores de linha

***Header*:** O cabeçalho (a vermelho na figura 3) consiste nos seguintes atributos:

- O atributo obrigatório (único campo obrigatório para o caso de um JWT não encriptado) `alg` (algorítmo) onde é indicado que algoritmo é usado para assinar e/ou desencriptar. O seu valor pode ser por exemplo HS256 (HMAC com o auxilio do SHA-256[6]) ou RSA.
- O atributo opcional `typ` (tipo do *token*) em que o seu valor é "JWT". Serve apenas para distinguir os JWTs de outros objetos que têm um JOSE *header*.
- O atributo opcional `cty` (tipo do conteúdo (*payload*)). Se o *payload* conter atributos arbitrários este atributo não deve ser colocado. Caso o *payload* for um JWT[7] então este atributo deve ter o valor de "JWT".

O cabeçalho é de grande importância visto que permite saber se o JWT é assinado ou encriptado e de que forma o resto do JWT deve ser interpretado.

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

Exemplo 2.1: *Header* usado para construir o JWT da figura 3

***Payload*:** O *payload* (a roxo na figura 3) contém a informação/dados que pretendemos transmitir com o JWT. Não há atributos obrigatórios contudo existem certos atributos que têm um significado definido (atributos registados).

Existem 7 atributos registados (*registered claims*): [12]

- `iss` (*issuer*): Identificador único (*case-sensitive string*) que identifica unicamente quem emitiu o JWT. A sua interpretação é específica a cada aplicação visto que não há uma autoridade central que gere os emissores.
- `sub` (*subject*): Identificador único (*case-sensitive string*) que identifica unicamente de quem é a informação que o JWT transporta. Este atributo deve ser único no contexto do emissor, ou se tal não for possível, globalmente único. O tratamento do atributo é específico a cada aplicação.
- `aud` (*audience*): Identificador único (*case-sensitive string*) ou *array* destes identificadores únicos que identificam unicamente os destinatários pretendidos do JWT. Ou seja, quem lê o JWT se não estiver no atributo `aud` não deve considerar os dados contidos no JWT. O tratamento deste atributo também é específico a cada aplicação.

[6] Função pertencente ao conjunto de funções *hash* criptográficas Secure Hash Algorithm 2 (SHA-2) desenhadas pela NSA

[7] JWT aninhado (*nested JWT*)

- `exp` (*expiration (time)*): Um número inteiro que representa uma data e hora específica no formato *seconds since epoch* definido pela POSIX[8], a partir da qual o JWT é considerado inválido (expira).
- `nbf` (*not before (time)*): Representa o inverso do atributo `exp` visto que é um número inteiro que representa uma data e hora específica no mesmo formato do atributo `exp`, mas que a partir da qual o JWT é considerado válido.
- `iat` (*issued at (time)*): Um número inteiro que representa uma data e hora especifica no mesmo formato dos atributos `exp` e `nbf` na qual o JWT foi emitido.
- `jti` (*JWT ID*): Identificador único (*string*) do JWT que permite distinguir JWTs com conteúdo semelhante. A implementação tem de garantir a unicidade deste identificador.

Estes atributos registados têm todos 3 caracteres visto que um dos requisitos do JWT é ser o mais pequeno/compacto possível.

Existem depois mais dois tipos de atributos, públicos e privados. Os atributos públicos podem ser definidos à vontade pelos utilizadores de JWTs mas têm de ser registados em *IANA JSON Web Token Claims registry* ou definidos por um espaço de nomes resistente a colisões de forma a evitar a colisão de atributos. Já os atributos privados são aqueles que não são nem registados nem públicos e podem ser definidos à vontade pelos utilizadores de JWTs. Os dois atributos usados no exemplo 2.2 (`name` e `num`) são atributos privados.

```
{
    "name": "José Martins",
    "num": "a78821"
}
```

Exemplo 2.2: *Payload* usado para construir o JWT da figura 3

***Signature*:** A assinatura (a azul da figura 3) é criada ao usar o algoritmo indicado na *header* no atributo `alg` tendo como um dos argumentos os elementos codificados da *header* e do *payload* juntos por um ponto e como outro argumento um segredo. O resultado do algoritmo é depois codificado em `Base64 URL-safe`. Esta assinatura no caso dos JWSs é usada para verificar a integridade do JWT e caso seja assinado com uma chave privada permite também verificar se o remetente é quem diz ser. No caso de o atributo `alg` for `none` a assinatura é uma `string` vazia.

```
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    segredo1.-uminho!clav
```

[8]Mais informação em https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16

```
)
```

Exemplo 2.3: *Signature* usado para construir o JWT da figura 3

### 2.2.2 *Criação de JWT/JWS*

Na figura 4 é apresentada a construção de um JWT em que o atributo `alg` (algorítmo) tem o seu valor igual a `none`, ou seja, o JWT não é assinado nem encriptado.

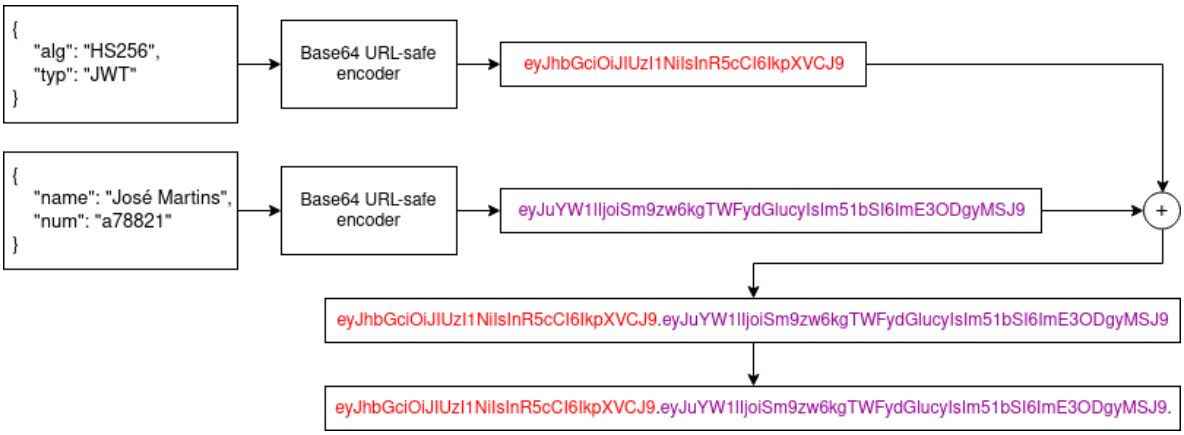



Figura 4: Criação de um JWT

Já na figura 5 é demonstrada a construção de um JWT assinado, ou seja, um JWS.

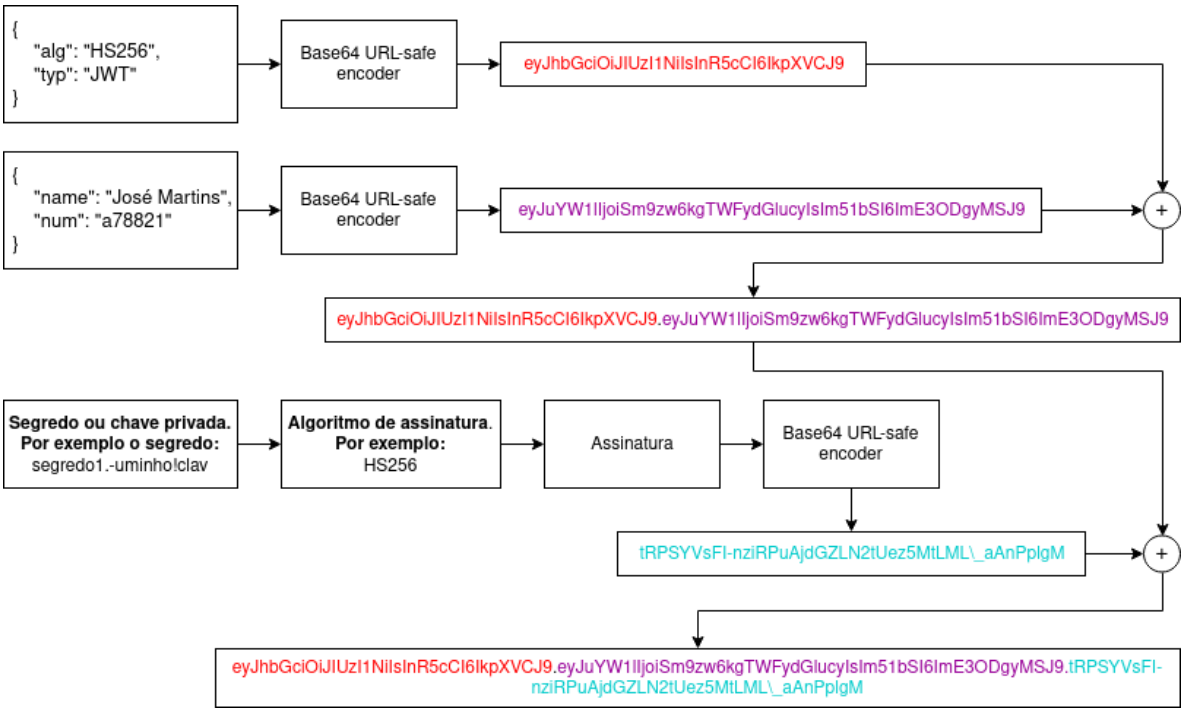



Figura 5: Criação de um JWS

### 2.2.3 *Alternativas ao JWT*

Algumas alternativas ao JWT passam pelo uso de Simple Web Token (SWT) ou Security Assertion Markup Language (SAML). Se compararmos o JWT ao SAML, o JSON é menos verboso que o XML e mesmo quando codificado o seu tamanho é menor.

De um ponto de vista de segurança o SWT apenas pode ser assinado simetricamente por um segredo partilhado usando o algoritmo HMAC. Já o JWT e o SAML podem usar pares de chaves pública/privada para assinar. Contudo assinar XML com *XML Digital Signature* sem introduzir buracos de segurança é mais dificil quando comparado com a simplicidade de assinar JSON. [3]

Houve contudo algumas bibliotecas de JWT com vulnerabilidades devido ao atributo `alg` da *header* do JWT. Havia duas situações de vulnerabilidade:

- As bibliotecas ao fazer a verificação (recebe um JWT e um segredo/chave pública como argumentos) de um JWT com `alg` igual a `none` assumiam logo que o JWT era válido mesmo que o segredo/chave pública fosse diferente de vazio. Ou seja, com a simples alteração do atributo `alg` e com a remoção da *signature* podia-se alterar o *payload* do JWT que o servidor iria continuar a considerar que a integridade do JWT não foi colocada em causa mesmo que os JWTs gerados pelo servidor tivessem sido com um algoritmo e com recurso a um segredo/chave privada.

- As bibliotecas ao fazer a verificação seja um algoritmo simétrico ou assimétrico apenas tinham como parâmetros o JWT e o segredo/chave pública. Isto gera uma segunda vulnerabilidade, se o servidor estiver à espera de um JWT assinado com pares de chaves pública/privada mas recebe um JWT assinado com HMAC vai assumir que a chave pública é o segredo a usar no algoritmo HMAC. Ou seja, se se criar um JWT com o atributo `alg` igual a HMAC e a assinatura for gerada usando o algoritmo HMAC com o segredo a ser a chave pública, podemos alterar o *payload* (antes de assinar) que o servidor vai considerar que o JWT não foi maliciosamente alterado.

Portanto a flexibilidade de algoritmos dada pelo JWT coloca em causa a segurança pelo que da parte das bibliotecas o atributo `alg` não deve ser considerado [7] bem como deve ser *deprecated* e deixar de ser incluído nos JWTs[9].

A biblioteca que será usada na CLAV, `jsonwebtoken`, já endereçou estes problemas[10] pelo que estas vulnerabilidades não estarão presentes na CLAV.

Por outro os *parsers* de JSON são mais comuns em grande parte das linguagens de programação visto que os JSONs mapeiam diretamentepara objetos ao contrário do XML que não tem um mapeamento natural de documento para objeto. [3] Portanto isto torna mais fácil trabalhar com JWT do que com SAML.

Já quando comparamos os JWT a *cookie sessions*, o JWT tem a vantagem de as sessões puderem ser *stateless* enquanto que as *cookies* são *statefull*. Contudo, ser *stateless* não permite por exemplo que a qualquer altura se possa revogar um JWT. Para endereçar esse problema é necessário, por exemplo, guardar (*statefull*) os JWTs numa base de dados associando cada JWT ao identificador único de quem é a informação contida no JWT (o uso de uma *whitelist*). Assim para revogar um JWT bastaria removê-lo da base de dados.

Outra alternativa ao JWT seria *sessionIDs*. As *sessionIDs* são *strings* longas, únicas e aleatórias. É possível revogar um *sessionID*, ao contrário do JWT, bastando para isso remover o *sessionID* da base de dados. Mais à frente na secção **??** veremos outra possível alternativa com recurso a API *gateways* em que uma possível abordagem é dentro da API *gateway* ser através de JWTs, fora ser usado *sessionIDs* e na "entrada" da API *gateway* ter uma base de dados que associa os *sessionsIDs* aos JWTs.

Por fim, uma outra alternativa bastante semelhante ao JWT é *Branca*. *Branca* usa o algoritmo simétrico *IETF XChaCha20-Poly1305 AEAD* que permite criar *tokens* encriptados e que garantam integridade. Tem também uma região de *payload* como JWT com a única diferença é que este *payload* não tem um estrutura definida. Não necessita da *header* visto que o algoritmo usado não varia. Em vez de usar codificação em `Base64 URL-safe` usa `Base62` que também é *URL-safe*. Para além disso o *token* gerado é geralmente de menor dimensão do que o gerado pelo JWT sendo como tal mais compacto que o JWT. [16] Visto que o *Branca* encripta e garante integridade de uma forma mais simples que o JWT permite (para isso era necessário recorrer a um JWE que tem no seu *payload* um JWS), sendo como tal propenso

[9] Ver `https://gist.github.com/paragonie-scott/c88290347c2589b0cd38d8bb6ac27c03`
[10] Ver `https://github.com/auth0/node-jsonwebtoken/commit/1bb584bc382295eeb7ee8c4452a673a77a68b687`

a menos erros de programação. Contudo, o *Branca* ainda não é muito conhecido nem um *standard* da indústria, ao contrário do JWT, mas não deixa de ser algo a ter em conta para o futuro.

## 2.3 Autorização de pedidos à API

Quanto à forma como os pedidos serão feitos à API poderão ser feitos de duas formas, através da *header* HTTP *Authorization* ou através da *query string* do pedido em um dos seguintes campos:

**`token`** caso seja o `token` de um utilizador:
`http://example.com/path/page?token=<token>`

**`apikey`** caso seja uma Chave API:
`http://example.com/path/page?apikey=<Chave API>`

Na *header Authorization* irá se usar o esquema de autenticação *Bearer*[11] com umas pequenas alterações. Portanto o conteúdo da *header Authorization*:

- Caso seja o `token` de um utilizador:
  `token <token>`
- Caso seja uma Chave API:
  `apikey <Chave API>`

ao invés do esquema de autenticação predefenido do *Bearer*: `Bearer <token/Chave API>`

Convém referir que a Chave API é também um *token*. A divisão entre utilizadores e chaves API permite uma mais fácil gestão dos *tokens* recebidos pela API bem como usar duas formas diferentes de os gerar/verificar com o possível benefício de melhorar a segurança da API.

Os *token*s gerados pela API serão JWTs. Contudo poderiam ser outro tipo de *tokens* (por exemplo uma *string* aleatória e única) que o processo de envio dos *tokens* para a API manter-se-ia igual.

Após descrito como poderão ser feitos os pedidos à API, irá ser apresentado possíveis fluxos de interação entre utilizadores (*browser*, *app*, etc) e o servidor da API.

O fluxo de autenticação de um utilizador na API a ser implementado será o seguinte:

[11] Mais informação em `https://tools.ietf.org/html/rfc6750`

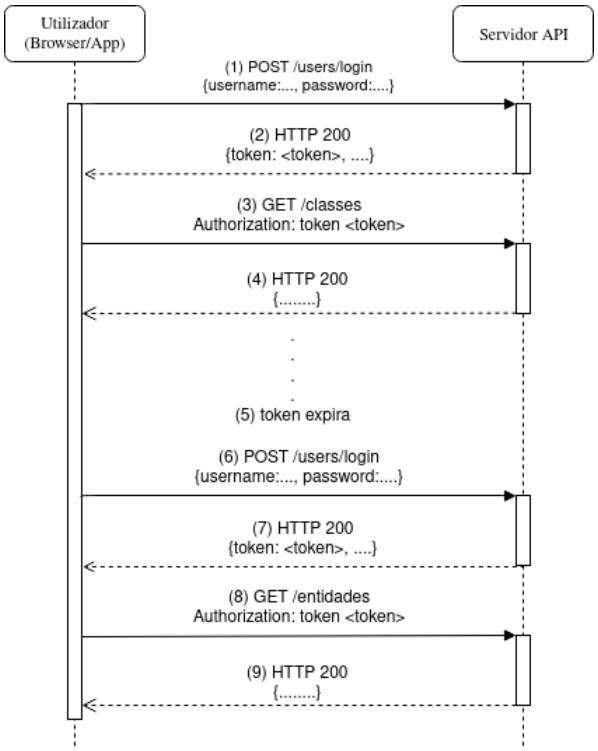



Figura 6: Fluxo de autenticação e posteriores pedidos de um utilizador

1. Utilizador autentica-se ao providenciar o seu *email* e a sua *password*
2. Caso o utilizador se autentique com sucesso é devolvido um *token* que deve ser usado nos restantes pedidos até expirar
3. Utilizador realiza um pedido para obter as classes, colocando o token na *header Authorization*
4. Caso o *token* enviado seja válido e não tenha expirado são devolvidas as classes
5. *Token* expirou após o tempo definido
6. Utilizador realiza uma nova autenticação por forma a obter um novo *token*
7. Caso o utilizador se autentique com sucesso é devolvido um *token* que deve ser usado nos restantes pedidos até expirar
8. Utilizador realiza um pedido para obter as entidades, colocando o token na *header Authorization*
9. Caso o *token* enviado seja válido e não tenha expirado são devolvidas as entidades

O fluxo de autenticação e renovação de uma Chave API na API a ser implementado será o seguinte:

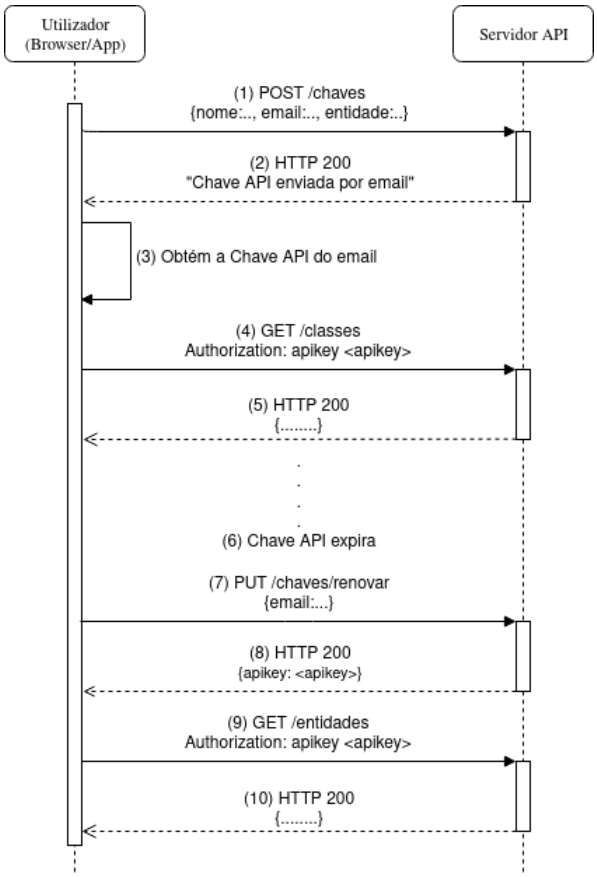



Figura 7: Fluxo de autenticação e posteriores pedidos de uma chave API

1. Utilizador cria uma chave API ao providenciar o nome, email e entidade
2. A Chave API é enviada para o email fornecido pelo utilizador com o objetivo de ser usada nos próximos pedidos
3. O utilizador obtém a chave API do email enviado
4. Utilizador realiza um pedido para obter as classes, colocando a chave API na *header Authorization*
5. Caso a Chave API enviada seja válida e não tenha expirado são devolvidas as classes
6. Chave API expirou após o tempo definido
7. Utilizador renova a Chave API ao providenciar o email usado para criar a Chave API
8. A nova (renovada) Chave API é devolvida para ser usada nos restantes pedidos
9. Utilizador realiza um pedido para obter as entidades, colocando a Chave API na *header Authorization*
10. Caso a Chave API enviada seja válida e não tenha expirado são devolvidas as entidades

### 2.3.1 *Verificação dos tokens no servidor API*

Para proteger as rotas da API é necessário haver métodos de verificação dos *tokens* com o objetivo de decidir se o utilizador/Chave API pode aceder a uma determinada rota. De seguida será apresentado o pseudo-código de verificação dos *tokens* tendo em conta que os utilizadores registados conseguem aceder a todas as rotas que as Chaves API conseguem mas que o inverso não acontece. Ou seja, um utilizador registado até o de nível mais baixo por exemplo, consegue aceder a todas as rotas que as Chaves API tem acesso e mais algumas nas quais as Chaves API não têm permissões de acesso.

Por forma a validar se uma Chave API pode aceder a uma determinada rota pode ser executada a seguinte função em *middleware*:

```
function isLoggedInKey(req, res, next)
    key = getJWTfromHeaderOrQueryString('apikey')

    if key then
        keyBD = getKeyFromMongoDB(key)
        if keyBD then
            res = jwt.verify(key, secretForAPIkey)
            if res != expired then
                if keyBD.active == True then
                    return next()
                else
                    return err
            else
                return err
        else
            return err
    else
        return isLoggedInUser(req, res, next)
```

Exemplo 2.4: Verificação se um pedido com uma determinada Chave API pode ser efetuado

É importante destacar a chamada da função `isLoggedInUser` que é executada no caso de não ser detetado uma Chave API no pedido (na *header Authorization* ou na *query string* `apikey`) e como tal, com essa chamada, tenta-se perceber se afinal foi passado um *token* de um utilizador já que todos os utilizadores conseguem aceder às rotas que as Chaves API conseguem como já referido.

No seguimento, para validar se um determinado *token* de um utilizador registado pode aceder a uma determinada rota pode ser executada a seguinte função em *middleware*:

```
function isLoggedInKey(req, res, next)
    key = getJWTfromHeaderOrQueryString('token')

    if key then
        res = jwt.verify(key, secretForToken)
        if res != expired then
```

```
            if keyBD.active == True then
                return next()
            else
                return err
        else
            return err
    else
        return err
```

Exemplo 2.5: Verificação se um pedido com um determinado *token* de um utilizador registado pode ser efetuado

A obtenção do *token* bem como a verificação deste *token* pode ser obtido através da utilização da estratégia do `passport` chamada `passport-jwt`.

Além disso as obtenções dos *tokens* tanto das Chaves API como de *tokens* de utilizadores registados podem ser obtidos através da utilização de extratores presentes na estratégia `passport-jwt`. Assim para extrair o *token* da *query string* basta:

```
var ExtractJWT = require("passport-jwt").ExtractJwt
token = ExtractJWT.fromUrlQueryParameter("<nome do campo, 'token' ou 'apikey' no
    caso da CLAV>")
```

Exemplo 2.6: Extração do *token* da *query string*

Já para extrair o *token* da *header Authorization* bastaria:

```
var ExtractJWT = require("passport-jwt").ExtractJwt
token = ExtractJWT.fromAuthHeaderWithScheme("<palavra antes do token, 'Bearer' no
     caso dum bearer token, 'token' ou 'apikey' no caso da CLAV>")
```

Exemplo 2.7: Extração do *token* da *heaer Authorization*

Para verificar se o utilizador registado tem um nível suficiente para aceder a uma rota, depois de se verificar que o utilizador está autenticado (`isLoggedInUser`), deve-se executar também em *middleware* a seguinte função:

```
function checkLevel(clearance)
    return function(req, res, next)
        havePermissions = False

        if clearance is Array then
            if req.user.level in clearance then
                havePermissions = True
        else
            if req.user.level >= clearance then
                havePermissions = True

        if havePermissions then
            return next()
```

```
        else
            return err
```

Exemplo 2.8: Verificação se um utilizador registado tem permissões suficientes para aceder a uma determinada rota

Ou seja, a variável `clearance` poderá ser uma lista de números ou apenas um número. No primeiro caso verifica-se que o nível do utilizador está presente na lista, em caso afirmativo então o utilizador tem permissões para aceder. Já no segundo caso, o utilizador só terá permissões para aceder se o seu nível foi igual ao superior ao `clearance`.

Com estas três funções (`isLoggedInKey`, `isLoggedInUser` e `checkLevel`) é possível proceder à proteção da API da CLAV garantindo que utilizadores com diferentes níveis de acesso apenas conseguem aceder ao que lhes é permitido.

## 2.4 Autenticação.gov

O Autenticação.gov surgiu da necessidade de identificação unívoca de um utilizador perante sítios na Web. [1] Será esta quem realiza o processo de autenticação do utilizador e que fornecerá os atributos do utilizador necessários para identificar o utilizador numa entidade (*website*/portal).

O CC em conjunto com o Autenticação.gov permite obter os identificadores dos utilizadores junto das entidades participantes da iniciativa do CC (funcionalidade de Federação de Identidades da Plataforma de Interoperabilidade da Administração Pública). Além disso, o Autenticação.gov gere os vários fornecedores de atributos disponíveis bem como possui uma estreita ligação com a infraestrutura de chave pública do Cartão de Cidadão (Public Key Infrastucture (PKI)), com o intuito de manter os elevados níveis de segurança e privacidade no processo de autenticação e identificação. [1]

O Autenticação.gov permite também a criação de credencias comuns a todos os sites da AP, ou seja, o utilizador apenas necessita de se autenticar uma vez que poderá aceder aos vários portais (Portal do Cidadão, etc) com a mesma autenticação.

Para além disso o utilizador pode autenticar-se utilizando outros certificados digitais que não o CC (por exemplo Chave Móvel Digital (CMD), *user+password* ou redes sociais, estes dois últimos quando o *website*/portal necessita apenas de conhecer do utilizador o *email*).

No projeto CLAV irá ser implementado a autenticação com recurso ao Autenticação.gov através de dois certificados digitais diferentes:

- Cartão de Cidadão (CC): Já se encontra implementado como referido na secção 2.1.2. A autenticação é realizada através da leitura do CC (através de um leitor de cartões sendo necessário a instalação de *software* do Autenticação.gov para proceder à leitura do CC) e posterior inserção do PIN de autenticação recebido quando se cria/renova o CC.

- Chave Móvel Digital (CMD): Um dos objetivos desta tese é a implementação da autenticação com recurso a este certificado digital. Com o CMD, após o utilizador associar um número de telemóvel ao NIC, o utilizador pode autenticar-se com o número de telemóvel, o código PIN da CMD e o código de segurança temporário enviado por SMS.

De forma a completar a figura 2 apresenta-se de seguida o fluxo de pedidos efetuado entre o CLAV e o Autenticação.gov de forma a autenticar um utilizador na CLAV: [1]

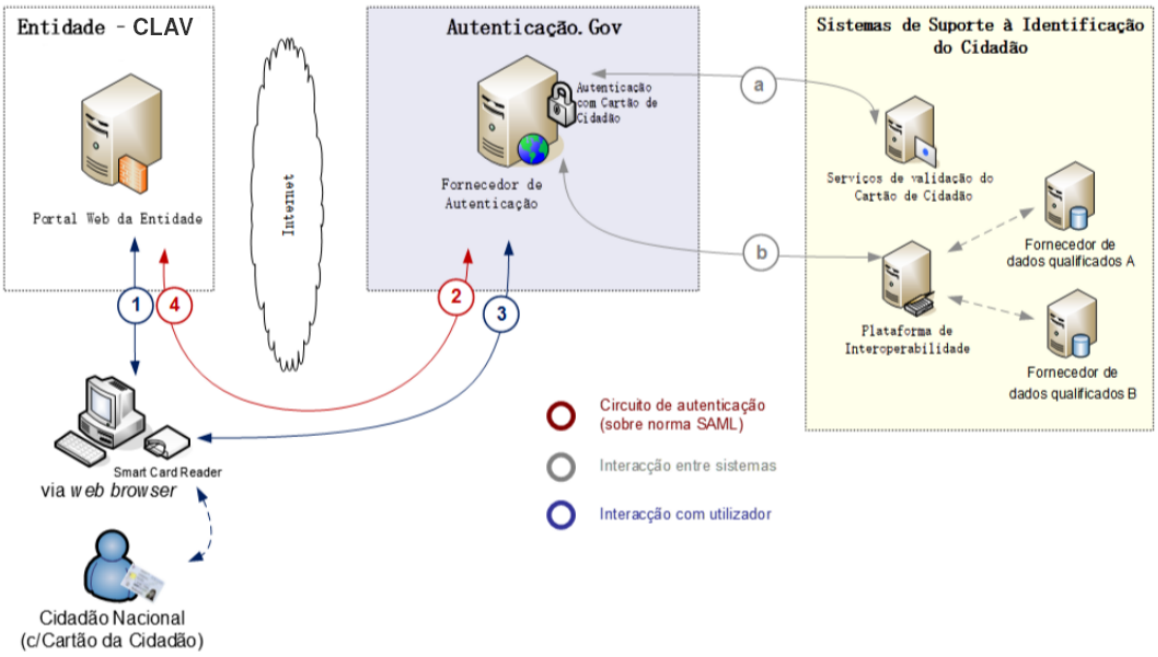



Figura 8: Fluxo de pedidos entre CLAV e o Autenticação.gov de forma a autenticar um utilizador na CLAV. Fonte: [1]

1. O utilizador pretende aceder à área privada do portal de uma entidade (do CLAV), na qual é necessário que comprove a sua identidade;

2. O portal da entidade (CLAV) delega a autenticação e redireciona o utilizador para o Autenticação.gov, juntamente com um pedido de autenticação assinado digitalmente;

3. O Autenticação.gov valida o pedido de autenticação recebido e solicita a autenticação do utilizador com recurso ao seu CC pedindo a inserção do seu PIN de autenticação. Durante este processo, o Autenticação.gov efetua as seguintes operações internas:

   a) Valida as credenciais do utilizador com recurso à PKI do CC via OCSP

   b) Obtém atributos que sejam solicitados pelo portal da entidade (CLAV) junto dos vários fornecedores de atributos qualificados. Esta operação é efetuada via Plataforma de Interoperabilidade. Este processo pode incluir a obtenção de dados da Federação de Identidades ou de outras Entidades.

4. A identificação e atributos do utilizador são autenticadas e assinados digitalmente pelo Autenticação.gov, após o que redireciona o utilizador de volta ao portal da entidade original (CLAV). Cabe à entidade (CLAV) a validação das credenciais do Autenticação.gov e utilização dos atributos do cidadão.

A troca de pedidos entre o CLAV e o Autenticação.gov é feita através de SAML 2.0 (com as extensões que a AMA considera obrigatórias). De seguida será feita uma pequena introdução ao SAML 2.0.

### 2.4.1 *SAML 2.0*

O Security Assertion Markup Language (SAML) define uma *framework standard* em XML. [5] Foi aprovado pela OASIS e permite a troca segura de informação de autenticação e autorização entre diferentes entidades. Através do SAML é possível através de uma credencial (*login* de um utilizador) aceder autenticado a um conjunto de *websites*. Esta funcionalidade é conhecida por Single Sign On (SSO).

Existem três tipos de papéis em SAML: [11]

- Utilizador
- *Identity Provider*: Realiza a autenticação de que o utilizador é quem diz ser e envia essa informação ao *Service Provider* junta com as permissões de acesso do utilizador para o serviço
- *Service Provider*: Precisa de autenticação do *Identity Provider* para poder dar autorização ao utilizador

O documento XML enviado pelo *Identity Provider* para o *Service Provider* é conhecida por *SAML Assertion*. Existem três tipos de *SAML Assertion*: [11]

- *Authentication Assertion*: Prova a identificação de um utilizador e fornece a hora em que o utilizador se autenticou e o método de autenticação usado
- *Attribution Assertion*: Envia *SAML attributes* (formato de dados que contém informação acerca do utilizador) para o *Service Provider*
- *Authorization Assertion*: Indica se o utilizador está autorizado a usar o serviço ou se o *Identity Provider* recusou o pedido à inserção de uma password errada ou por falta de permissões para usar o serviço

No projeto CLAV o utilizador final é o utilizador da CLAV, o *Identity Provider* é representado pelo Autenticação.gov e o *Service Provider* é representado pela CLAV.

## 2.5 SWAGGER

O *Swagger* é um ecossistema de ferramentas para desenvolver APIs com a OpenAPI Specification (OAS).

Até 2015 o *Swagger* consistia numa especificação e num ecossistema de ferramentas para implementar a especificação. Em 2015 a fundadora do *Swagger*, *SmartBear Software*, doou a especificação *Swagger* para a *Linux Foundation* e renomeou a especificação para OpenAPI Specification. [15]

A especificação *OpenAPI* é agora desenvolvida pela *OpenAPI Initiative* que envolve várias empresas tecnológias entre as quais *Microsoft*, *Google*, *IBM* e a fundadora *Smartbear Software*.

Já o conjunto de ferramentas *Swagger* inclui ferramentas *open-source*, gratuitas e comerciais que podem ser usadas em diferentes estágios do ciclo de vida de uma API, que inclui documentação, desenho, testes e *deployment*. Algumas das ferramentas são: [13]

- ***Swagger Editor***: Permite editar especificações *OpenAPI* em YAML no *browser*[12], validar as especificações em relação às regras do OAS bem como pré-visualizar a documentação em tempo real. Facilita o desenho e a documentação de APIs REST
- ***Swagger UI***: Coleção de *assets HTML*, *JavaScript* e *CSS* que geram dinamicamente documentação a partir de uma especificação *OpenAPI* de uma API
- ***Swagger Codegen***: Permite a geração de bibliotecas cliente (geração de SDK), *server stubs* e documentação automática a partir de especificações *OpenAPI*
- ***Swagger Inspector*** (gratuita): Ferramenta de testes de APIs que permite validar as APIs e gerar definições *OpenAPI* de APIs existentes
- ***SwaggerHub*** (gratuita e comercial): Desenho e documentação de APIs, construido para equipas que trabalham com *OpenAPI*

O *Swagger* possui duas abordagens: [17]

- *top-down*: Uso do *Swagger Editor* para criar a especificação *OpenAPI* e depois usar o *Swagger Codegen* por forma a gerar o código do cliente e do servidor. Ou seja, primeiro desenha-se a API antes de escrever código
- *bottom-up*: Utilizador já possui uma API REST e o *Swagger* irá ser usado apenas para documentar a API existente

Visto que o CLAV já possui grande parte da API construída vai ser usada uma abordagem *bottom-up*. Portanto, o *Swagger* vai ser usado apenas para uma das últimas fases da construção da API, a documentação da API. De forma a produzir a documentação, do portfólio de ferramentas do *Swagger* apenas precisaremos de utilizar o *Swagger UI* e o *Swagger Editor*. O primeiro permitirá apresentar aos utilizadores a documentação gerada e o segundo permitirá validar a especificação *OpenAPI* (documentação) criada, verificando se não possui erros.

[12] Aceder `https://editor.swagger.io/`

### 2.5.1 *Especificação OpenAPI*

A especificação *OpenAPI* providencia um conjunto de propriedades que podem ser usadas para descrever uma API REST. Com um documento de especificação válido é possível usá-lo para criar uma documentação interativa, por exemplo, através do *Swagger UI*.

De seguida será apresentado o que é possível documentar com a especificação *OpenAPI* e como. É possível usar YAML como JSON para a especificar. Esta parte será demonstrada usando YAML.[13]

#### *Metadata*

O primeiro passo é escolher a versão da especificação *OpenAPI* que irá ser usada para documentar:

```yaml
openapi: 3.0.0
```

Exemplo 2.9: Exemplo de indicação da versão da especificação *OpenAPI*

Depois na secção `info` é possivel descrever um pouco sobre a API que estamos a documentar, indicando o título, a descrição e a versão da API. As propriedades `title` e `version` são obrigatórias. É possível também colocar informação sobre os contactos disponíveis, termos de uso e a licença:[14]

```yaml
info:
  title: CLAV API
  description: Esta é a API do projeto CLAV...
  version: 1.0.0
```

Exemplo 2.10: Exemplo de secção `info` indicando título, descrição e versão da API na especificação *OpenAPI*

#### *Servidores*

Há depois uma secção com o nome de `servers` para indicar os *URL*s da API que se pode aceder. Podem ser indicados mais do que um *URL*:[15]

```yaml
servers:
  - url: http://clav-api.dglab.gov.pt/api
    description: Official API server
  - url: http://clav-test.di.uminho.pt/api
    description: Testing server
```

[13] A especificação completa do *OpenAPI* com versão igual a 3.0.0 pode ser vista em https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md

[14] Ver mais em https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#infoObject

[15] Para mais detalhes sobre esta secção veja https://swagger.io/docs/specification/api-host-and-base-path/

```
  - url: http://localhost:7779/api
    description: Local server
```

Exemplo 2.11: Exemplo de secção `servers` indicando os *URL*s e a descrição de cada na especificação *OpenAPI*

*Caminhos/Rotas*

De seguida apresenta-se uma das secções mais importantes da especificação, a secção `paths`. Aqui são definidas as rotas que a API disponibiliza. Para definir cada rota basta indicar o caminho relativo aos definidos na secção `servers` (`<server-url>/<caminho relativo>`). Nesta secção é definido tudo que envolve as rotas, desde os parâmetros necessários, as respostas que devolve, os métodos *HTTP* disponíveis, etc:[16,17]

```
paths:
  /users/{id}:
    get:
      summary: Resumo do que faz a rota
      description: >
        Descrição detalhada, pode ser usado Markdown para enriquecer o texto
      parameters:
        - name: id
          in: path
          description: Id do utilizador
          required: true
          schema:
            type: string
      responses:
        200:
          description: Descrição da resposta, p.e: Sucesso
          content:
            application/json:
              schema:
                #A estrutura do JSON devolvido pode ser definido logo aqui ou num
                     componente à parte, fazendo referência desse. Iremos aplicar
                    o segundo caso para demonstrar que estas funcionalidades
                    tornam a documentação mais fácil de manter
                $ref: '#/components/schemas/User'
    post:
      ...
    delete:
      ...
  /users:
    ...

components:
```

[16]Mais detalhes em https://swagger.io/docs/specification/paths-and-operations/
[17]mais detalhes sobre a funcionalidade `$ref` em https://swagger.io/docs/specification/using-ref/

```yaml
  schemas:
    User:
      type: object
      properties:
        id:
          type: string
        ...
      required:
        - id
        ...
```

Exemplo 2.12: Exemplo de secção `paths` indicando os detalhes de cada rota na especificação *OpenAPI*

Outro ponto importante a referir é que é possível agrupar as rotas em grupos através do uso de *tags*. As *tags* tem de ser definidas numa secção chamada *tags*:

```yaml
tags:
  - name: users
    description: Descrição
  - name: classes
    description: Outra descrição
```

Exemplo 2.13: Exemplo de secção `tags` defininfo tags na especificação *OpenAPI*

Depois em cada rota é necessário indicar a que *tag* (grupo) pertence:

```yaml
paths:
  /users/{id}:
    get:
      summary: Resumo do que faz a rota
      tags:
        - users
      ...
```

Exemplo 2.14: Exemplo de uso de *tags* numa rota na especificação *OpenAPI*

*Parâmetros*

Como já exemplificado no exemplo 2.12 os parâmetros de uma rota são definidos na secção `parameters` de cada rota. Existem quatro tipo de parâmetros que variam de acordo com o local onde se encontram. O tipo de um parâmetro é definido na propriedade `in` de um parâmetro e pode ser um dos seguintes:

- Parâmetros no caminho: Servem normalmente para apontar para um recurso específico. Estes parâmetros são sempre obrigatórios como tal a propriedade `required` com o valor igual a verdadeiro deve ser sempre adicionado. Para além disso o `name` tem de ser igual ao que está no caminho. A propriedade `in` tem o valor de *path*.

- Parâmetros na *query string*: A propriedade `in` tem o valor de *query*. No caso de tokens passados em parâmetros da *query string* deve-se usar esquemas de segurança, veja a secção 2.5.1 Autenticação.
- Parâmetros no cabeçalho: A propriedade `in` tem o valor de *header*. Contudo os cabeçalhos *Accept*, *Content-Type* e *Authorization* não são aqui definidos.
- Parâmetros no cabeçalho da *Cookie*: A propriedade `in` tem o valor de *cookie*.

Cada parâmetro tem várias propriedades que permitem definí-lo:[18]

- `required`: Indica se o parâmetro é obrigatório ou opcional. Possíveis valores são *true* ou *false*.
- Na propriedade `schema`:
  - `default`: Valor padrão de um parâmetro opcional
  - `type`: O tipo do parâmetro. Possíveis valores: *string*, *integer*, etc
  - `enum`: Indica os possíveis valores para o parâmetro
  - `nullable`: Indica se o parâmetro pode ser *null*. Possíveis valores são *true* ou *false*.
- `allowEmptyValue`: Indica se o parâmetro pode ser vazio. Apenas aplicável no caso de um parâmetro na *query string*. Possíveis valores são *true* ou *false*.
- `example`: Um exemplo do valor
- `examples`: Múltiplos exemplos
- `deprecated`: Indica se o parâmetro é ou não *deprecated*. Possíveis valores são *true* ou *false*.

*Request Body*

O *request body* é definido em cada rota na secção `requestBody` sendo usado essencialmente em rotas com o método HTTP igual a POST ou a PUT, ou seja, em casos que há necessidade de criar ou alterar um objeto de acordo com a informação fornecida no pedido. As propriedades que podem ser definidas no `requestBody` são as seguintes:[19,20]

- `description`: Opcionalmente pode ser adicionada uma descrição
- `required`: Indica se o *request body* é obrigatório ou opcional. Possíveis valores são *true* ou *false*. Por padrão o *request body* é opcional.
- `content`: Obrigatório. Lista os *media types* consumidos pela rota e especifica o `schema` para cada *media type*

[18] Mais detalhes em https://swagger.io/docs/specification/describing-parameters/

[19] Para mais detalhes, desde *upload* de ficheiros, a *Form Data*s, veja em https://swagger.io/docs/specification/describing-request-body/

[20] Para mais informação sobre os *media types* veja https://swagger.io/docs/specification/media-types/

*Respostas*

Nesta seccção, propriedade `responses` de cada rota, é descrita as possíveis respostas de cada rota. Na propriedade será definido as várias respostas, sendo uma resposta por cada HTTP *status code* possível de ser devolvido. Cada resposta pode possuir as seguintes propriedades:[21]

- `description`: Obrigatório, descrição da resposta
- `content`: Opcional, semelhante ao `content` do *request body* e define o contéudo que é devolvido.
- `headers`: Opcional, define as *headers* que são devolvidas na resposta

*Adição de Exemplos*

Na secção 2.5.1 Parâmetros já se referiu como é possível adicionar exemplos aos parâmetros. De forma semelhante o mesmo pode ser realizado tanto no *request body* como nas respostas através da propriedade `example` (um exemplo) ou `examples` (múltiplos exemplos) aninhado na propriedade `schema` ou aninhado no *media type* no caso do `schema` ser apenas uma referência para um modelo presente na secção `components`. A propriedade `example` pode também ser usada em objetos ou propriedades de um `schema`. Por fim, para adicionar exemplos de XML ou HTML os exemplos devem ser exemplificados como *strings*:[22]

```
content:
  application/xml:
    schema:
      $ref: '#/components/schemas/xml'
    examples:
      xml:
        summary: A sample XML response
        value: '<objects><object><id>1</id><name>new</name></object><object><id>
            2</id></object></objects>'
  text/html:
    schema:
      type: string
      examples:
        html:
          summary: A list containing two items
          value: '<html><body><ul><li>item 1</li><li>item 2</li></ul></body></html
              >'
```

Exemplo 2.15: Exemplo de adição de exemplos para XML e HTML na especificação *OpenAPI*

[21] Mais detalhes em https://swagger.io/docs/specification/describing-responses/
[22] Mais detalhes em https://swagger.io/docs/specification/adding-examples/

*Modelos*

A secção `schemas` presente na secção `components` permite definit estruturas de dados (modelo) a serem usados na API. Estes modelos podem ser referenciados usando a funcionalidade `$ref`.[23]

*Autenticação e Autorização*

Nesta secção será demonstrada como se pode adicionar a autenticação e autorização à especificação *OpenAPI*. Para tal é necessário criar *security schemes*. Os esquemas são definidos na secção `securitySchemes` dentro da secção `components`. Para cada esquema de segurança é necessário definir a propriedade `type`. Na especificação é possível descrever os seguintes esquemas de segurança:

- Esquemas de autenticação HTTP (usam o cabeçalho *Authorization*) (`type` = `http`):
  - *Basic* (propriedade `scheme` = `basic`)
  - *Bearer* (propriedade `scheme` = `bearer` e pode também ser definido o formato do *Bearer* (a palavra usada antes de indicar o *token*) através da propriedade `bearerFormat`)
  - Outros esquemas HTTP definidos pelo RFC 7235 e pelo registo de esquemas de autenticação HTTP
- Chaves API no cabeçalho, na *query string* ou em *cookies* (`type` = `apiKey` e na propriedade `in` indicar em que local se encontra, se no cabeçalho (`header`), se na *query string* (`query`) ou se nas *cookies* (`cookie`))
- *OAuth 2* (`type` = `oauth2`)
- *OpenID Connect Discovery* (`type` = `openIdConnect`)

Após definir os esquemas de segurança é necessário aplicá-los nas rotas que devem estar protegidas por esses esquemas. Paral tal em cada rota pode ser definida a propriedade `security` e indicar os esquemas de segurança que essa rota suporta.[24]

*Alternativas*

Em termos de alternativas à especificação *OpenAPI* existem duas concorrentes: *RAML*[25] e *API Blueprint*[26].

Comparemos as três hipóteses: [14]

[23] Mais detalhes em https://swagger.io/docs/specification/data-models/

[24] Mais detalhes em https://swagger.io/docs/specification/authentication/

[25] Ver https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/

[26] Ver https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md

| Especificação | Vantagens | Desvantagens |
|---|---|---|
| *OpenAPI* | • Grande adoção<br>• Grande comunidade de utilizadores<br>• Bom suporte<br>• Suporte para várias linguagens | • Falta de construtores avançados para metadados |
| RAML | • Suporta construções avançadas<br>• Adoção decente<br>• *Human readable format*<br>• Grande apoio da indústria | • Falta de ferramentas ao nível do código<br>• Ainda não comprovado a longo prazo |
| *API Blueprint* | • Fácil de entender<br>• Simples de escrever | • Pouca adoção<br>• Falta de construtores avançados<br>• Instalação complexa |

Tabela 1: Comparação entre especificações de documentação de APIs

Além das vantages apresentadas as três são *open-source*. Para além disso, o *OpenAPI* pode ser escrito em JSON ou YAML, o RAML é escrito em YAML e o *API Blueprint* é escrito em *Markdown*. Escolheu-se a especificação *OpenAPI* devido à sua grande adoção e por permitir usar o ecossistema de ferramentas *Swagger*.

### 2.5.2 *Swagger UI*

O *Swagger UI* permite a qualquer um visualizar uma API REST. A partir de um documento JSON ou YAML (especificação *OpenAPI*) é automaticamente gerado uma documentação interativa.

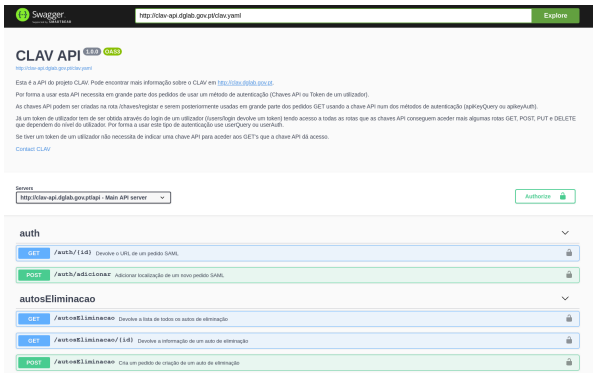



Figura 9: *Swagger UI* exemplo

*Alternativas*

Existem várias alternativas ao *Swagger UI*:

| Ferramenta | Vantagens | Desvantagens |
|---|---|---|
| *Swagger UI* | • Suporta a especificação *OpenAPI*<br>• *Open-source*<br>• Amplamente usado | |
| *Apiary*[27] | • Suporta a especificação *API Blueprint* e a especificação *OpenAPI* | • Necessário pagar de forma a puder integrar a documentação da API num domínio próprio<br>• *Closed-source* |
| *API Console*[28] | • Suporta a especificação RAML e a especificação *OpenAPI*<br>• *Open-source* | |
| *Slate*[29] | • *Open-source*<br>• API definida em *Markdown* | • Não suporta nenhuma especificação |
| *apiDoc*[30] | • Documentação criada a partir das anotações nos comentários do código<br>• *Open-source* | • Não suporta nenhuma especificação |
| *ReDoc*[31] | • Suporta a especificação *OpenAPI*<br>• *Open-source*<br>• Fácil de integrar | |

Tabela 2: Comparação entre ferramentas de APIs

De forma a escolher a ferramenta apropriada é necessário ter em conta que:

- Não há financiamento
- Já existe uma API desenvolvida

[27] Ver https://apiary.io/
[28] Ver https://github.com/mulesoft/api-console
[29] Ver https://github.com/slatedocs/slate
[30] Ver https://apidocjs.com/
[31] Ver https://github.com/Redocly/redoc

- A documentação deve estar acessível de um domínio próprio
- A documentação deve ser fácil de criar, de editar e de manter
- Será usada a especificação *OpenAPI*

As escolhas ficam como tal reduzidas ao *Swagger UI* e ao *ReDoc*. Optou-se por escolher o *Swagger UI* visto ser a ferramenta mais amplamente usada para além de que é possível obter também uma fácil integração no *Swagger UI* com recurso à package `swagger-ui-express`.

## 2.6 Documentação da API do CLAV

Agora sabendo que será usada a especificação *OpenAPI* e o *Swagger* é importante perceber que bibliotecas devem ser usadas para a produção da documentação.

Existem duas *packages* que podem ser usados para criar documentação interativa para uma API REST criada com *Node.js* e *Express.js*: [17]

- `swagger-node-express`
  - Vantagens
    - Módulo oficial suportado pelo *Swagger*
    - É *open-source* e como tal é possível contribuir para a correção de problemas
    - A solução contém *Swagger Editor* e *Swagger Codegen* e como tal tanto podemos usar uma abordagem *top-down* como *bottom-up*
  - Desvantagens
    - Instalação manual do *Swagger UI*. O código do *Swagger UI* tem de ser copiado manualmente para o projeto e sempre que há uma atualização é necessário copiar manualmente de novo
    - Instalação complexa. Por forma a aplicação hospedar a documentação é necessário adicionar algumas rotas ao servidor para além das já definidas na especificação *OpenAPI*
    - Fraca documentação
- `swagger-ui-express`
  - Vantagens
    - É *open-source* e como tal é possível contribuir para a correção de problemas
    - Não é necessário copiar manualmente o *Swagger UI*
    - De fácil instalação, apenas é necessário adicionar uma rota aonde estará hospedada a documentação
    - Boa documentação

- Desvantagens
  - Não é o módulo oficial suportado pelo *Swagger*

Das duas foi escolhida a `swagger-ui-express` visto ser de mais simples implementação e de mais fácil manutenção.

```
var swaggerUI = require('swagger-ui-express')
//JSON
var swaggerDocument = require('./swagger.json')
//ou YAML
var yaml = require('js-yaml')
var fs = require('fs')
var swaggerDocument = yaml.load(fs.readFileSync('./swagger.yaml'))

app.use('/doc', swaggerUI.serve, swaggerUI.setup(swaggerDocument));
```

Exemplo 2.16: Exemplo de uso do `swagger-ui-express`

No exemplo 2.16 a documentação da API está presente na rota `'/doc'`. Neste exemplo é exemplificado como carregar uma especificação *OpenAPI* em JSON bem como em YAML. Quanto ao *middleware* `serve` retorna os ficheiros estáticos necessários para hospedar o *Swagger UI*. Já o segundo *middleware* `setup` para além de puder receber o documento com a especificação *OpenAPI* pode também receber um outro parâmetro de opções que o utilizador pode definir para a apresentação interativa da documentação com o *Swagger UI*[32].

Agora há duas abordagens possíveis de realizar a documentação:

- Documentação de cada rota nos comentários da rota através da utilização da *package* `swagger-jsdoc`[33]
- Documentação à parte do código

A abordagem escolhida foi a da documentação à parte do código por forma a modularizar a documentação. A modularização da documentação foi realizada através do uso da *package* `yaml-include`[34]. Esta *package* permite que o documento YAML da especificação *OpenAPI* possa ser dividida por vários ficheiros. Ela permite a inclusão de arquivos YAML externos ou a inclusão de pastas de ficheiros YAMLs. Esta é funcionalidade é desaprovada pela equipa d desenvolvimento do YAML contudo é de grande ajuda e de simplificação da construção do ficheiro de especificação *OpenAPI*.

```
openapi: 3.0.0
info:
  description: Esta é a API do projeto CLAV. Pode encontrar mais informação sobre
      o CLAV em [http://clav.dglab.gov.pt](http://clav.dglab.gov.pt).
```

[32] As opções possíveis estão presentes em https://github.com/scottie1984/swagger-ui-express. Para o atributo (opção) `swaggerOptions` as opções possíveis estão presentes em https://github.com/swagger-api/swagger-ui/blob/master/docs/usage/configuration.md

[33] Ver https://github.com/Surnet/swagger-jsdoc

[34] Ver https://github.com/claylo/yaml-include

```
  version: 1.0.0
  title: CLAV API
  contact:
    name: CLAV
    email: clav@dglab.gov.pt
servers:
  - url: http://localhost:7779/api
    description: Local API server
paths: !!inc/dir [ 'paths' ]
components:
  schemas: !!inc/dir [ 'schemas', excludeTopLevelDirSeparator: true ]
```

Exemplo 2.17: Exemplo de uso do `yaml-include` no documento de especificação *OpenAPI*(*index.yaml*)

O ficheiro *index.yaml* será a raiz do documento de especificação *OpenAPI* a ser gerado com a *package* `yaml-include`. A estrutura dos ficheiros para gerar o documento de especificação *OpenAPI* final exemplifica como se pode dividir a documentação por vários ficheiros com esta *package*:

```
* index.yaml
* paths/
    * classes/
        * get.yaml
        * ~id/
            * get.yaml
    * users/
        * ~id/
            * post.yaml
            * delete.yaml
* schemas/
    * User.yaml
```

Exemplo 2.18: Exemplo de estrutura dos ficheiros para gerar o documento de especificação *OpenAPI*

Assim, o `!!inc/dir` fará que no ficheiro *index.yaml* na *tag paths* sejam incluídos todos os ficheiros que estão na pasta *paths*. Cada ficheiro corresponderá a uma determinada rota com um determinado método HTTP. O método HTTP é definido a partir do nome do ficheiro e o caminho da rota é determinado pelo nome das pastas e do aninhamento destas. Quando o nome da pasta é iniciado por "~" no caminho será colocado o nome da pasta sem o til entre chavetas ("{") por forma a indicar um parâmetro que é colocado no caminho do pedido.

Já no caso do `!!inc/dir` dos *schemas* a opção `excludeTopLevelDirSeparator` permite que os ficheiros que estejam dentro da pasta *schemas* (mas não aninhados dentro de outras pastas) sejam incluídos sem qualquer aninhamento, assumindo o nome do ficheiro como o atributo a colocar.

O documento de especificação *OpenAPI* final gerado será:

```
openapi: 3.0.0
```

```yaml
info:
  description: Esta é a API do projeto CLAV. Pode encontrar mais informação sobre
      o CLAV em [http://clav.dglab.gov.pt](http://clav.dglab.gov.pt).
  version: 1.0.0
  title: CLAV API
  contact:
    name: CLAV
    email: clav@dglab.gov.pt
servers:
  - url: http://localhost:7779/api
    description: Local API server
paths:
  /classes:
    get:
      <conteúdo do ficheiro paths/classes/get.yaml>
  /users/{id}:
    post:
      <conteúdo do ficheiro paths/~id/post.yaml>
    delete:
      <conteúdo do ficheiro paths/~id/delete.yaml>
components:
  schemas:
    User:
      <conteúdo do ficheiro schemas/User.yaml>
```

Exemplo 2.19: Documento de especificação *OpenAPI* gerado a partir do ficheiro *index.yaml* com o uso da *package* `yaml-include`

No final teremos um ficheiro no formato YAML com toda a documentação da API que puderá então ser usado para alimentar a documentação dinâmica *Swagger UI*.

## 2.7 Importação de dados

## 2.8 Exportação de dados

Um dos requisitos da API da CLAV é permitir a exportação de Classes, Entidades, Tipologias e Legislações em formato JSON, XML e CSV. Deve também permitir exportar toda a ontologia do projeto nos formatos Turtle, JSON-LD e RDF/XML.

Para a primeira parte foi necessário desenvolver dois conversores, de JSON para XML e de JSON para CSV visto que o JSON já é por predefinição devolvido.

O conversor de JSON para XML foi contruído sem recorrer a nenhuma biblioteca através do seguinte algoritmo:

```
sizeTab = 4

function protectForXml(string)
    string = replace '<' by '&lt;' in string
```

```
    string = replace '>' by '&gt;' in string
    string = replace '&' by '&amp;' in string
    string = replace "'" by '&apos;' in string
    string = replace '"' by '&quot;' in string
    return string

function protectKey(string)
    string = replace '<' by '' in string
    string = replace '>' by '' in string
    string = replace '&' by '_' in string
    string = replace "'" by '' in string
    string = replace '"' by '' in string
    string = replace '\s+' by '_' in string
    return string

function json2xmlArray(array, nTabs)
    xml = ''
    len = length(array)

    for i=0; i < len; i++
        type = type of array[i]
        xml += repeat(' ', nTabs * sizeTab) + '<item index="' + i + '" type="' +
            type + '">'

        if type == 'object' or type == 'string' or type == 'boolean' or type == '
            number' then
            xml += json2xmlRec(array[i], nTabs + 1)

        if type == 'object' then
            xml += repeat(' ', nTabs * sizeTab)

        xml += '</item>\n'

    return xml

function json2xmlRec(json, nTabs)
    xml = ''
    type = type of json

    if type == 'object' then
        xml = '\n'

        if json is an Array then
            xml = json2xmlArray(json, nTabs)
        else
            for key in json
                aux = ''
                type = type of json[key]

                if type === 'object' then
```

```
                    if json[key] is an Array then
                        aux = '\n' + json2xmlArray(json[key], nTabs + 1)
                        type = 'array'
                    else
                        aux += json2xmlRec(json[key], nTabs + 1)

                    aux += repeat(' ', nTabs * sizeTab)
                else if type == 'string' then
                    aux = protectForXml(json[key])
                else if type == 'boolean' or type == 'number' then
                    aux = json[key]

                xml += repeat(' ', nTabs * sizeTab)
                xml += '<' + protectKey(key) + ' type="' + type + '">'
                xml += aux + '</' + protectKey(key) + '>\n'
    else if type == 'string' then
        xml = protectForXml(json)
    else if type == 'boolean' or type == 'number' then
        xml = json

    return xml

function json2xml(json)
    xml = '<?xml version="1.0" encoding="utf-8"?>\n'
    xml += '<root>'
    xml += json2xmlRec(json, 1)
    xml += '</root>'
    return xml
```

Exemplo 2.20: Algorítmo de conversão de JSON para XML

Portanto, os dados exportados estarão sempre encapsolados na *tag* `root` por forma a garantir que só existe um elemento *root* no documento XML gerado. Cada tipo de dados do JSON é convertido da seguinte forma:

- *string*: Mantém-se igual tirando os caracteres "`<`", "`>`", "`&`", "`'`" e "`"`" que são convertidos para a *Entity Reference*[35] correspondente
- *number*: Mantém-se igual
- *boolean*: Mantém-se igual
- *null*: Origina uma *string* vazia
- *array*: Cada item é encapsulado numa *tag* `item` que possui um atributo `index` que indica a posição do elemento no *array* e um atributo `type` que indica o tipo do elemento do *array*. O tipo pode ser *number*, *boolean*, *string*, *array* ou *object*.

[35] "`<`" para "`<`", "`>`" para "`>`", "`&`" para "`&`", "`'`" para "`&apos;`" e "`"`" para "`&quot;`"

- *object*: Para cada propriedade será criado uma *tag* com valor igual à chave da propriedade e ao valor da propriedade será aplicado recursivamente uma das transformações desta lista. Esta *tag* terá um atributo `type` em que o seu valor, tal como nos *arrays*, pode ser *number*, *boolean*, *string*, *array* ou *object*.

Apresenta-se de seguida uma conversão exemplo. Para o seguinte JSON:

```
{
    "Actors": [
        {
            "name": "Tom Cruise",
            "age": 56,
            "Born At": "Syracuse, NY",
            "Birthdate": "July 3, 1962",
            "wife": null,
            "weight": 67.5,
            "hasChildren": true,
            "hasGreyHair": false,
            "children": [
                "Suri",
                "Isabella Jane",
                "Connor"
            ]
        },
        {
            "name": "Robert Downey Jr.",
            "age": 53,
            "Born At": "New York City, NY",
            "Birthdate": "April 4, 1965",
            "wife": "Susan Downey",
            "weight": 77.1,
            "hasChildren": true,
            "hasGreyHair": false,
            "children": [
                "Indio Falconer",
                "Avri Roel",
                "Exton Elias"
            ]
        }
    ]
}
```

Exemplo 2.21: JSON exemplo a converter

é gerado o seguinte XML:

```
<?xml version="1.0" encoding="utf-8"?>
```

```xml
<root>
   <Actors type="array">
      <item index="0" type="object">
         <name type="string">Tom Cruise</name>
         <age type="number">56</age>
         <Born_At type="string">Syracuse, NY</Born_At>
         <Birthdate type="string">July 3, 1962</Birthdate>
         <wife type="object">
         </wife>
         <weight type="number">67.5</weight>
         <hasChildren type="boolean">true</hasChildren>
         <hasGreyHair type="boolean">false</hasGreyHair>
         <children type="array">
            <item index="0" type="string">Suri</item>
            <item index="1" type="string">Isabella Jane</item>
            <item index="2" type="string">Connor</item>
         </children>
      </item>
      <item index="1" type="object">
         <name type="string">Robert Downey Jr.</name>
         <age type="number">53</age>
         <Born_At type="string">New York City, NY</Born_At>
         <Birthdate type="string">April 4, 1965</Birthdate>
         <wife type="string">Susan Downey</wife>
         <weight type="number">77.1</weight>
         <hasChildren type="boolean">true</hasChildren>
         <hasGreyHair type="boolean">false</hasGreyHair>
         <children type="array">
            <item index="0" type="string">Indio Falconer</item>
            <item index="1" type="string">Avri Roel</item>
            <item index="2" type="string">Exton Elias</item>
         </children>
      </item>
   </Actors>
</root>
```

Exemplo 2.22: XML resultante da conversão do JSON presente em 2.21

Da mesma forma que o XML, o CSV é convertido sem recurso a qualquer biblioteca visto que a conversão a realizar é muito específica a cada objeto JSON. Ao contrário do conversor desenvolvido para XML, o conversor para CSV não converte qualquer objeto para CSV mas apenas um conjunto restrito de objetos JSON.

Exemplo 2.23: Algorítmo de conversão de JSON para CSV

Por fim quanto à exportação da ontologia, das três é a mais simples visto que o *GraphDB*[36] possui funcionalidades de exportação dos triplos presentes numa BD armazenada no *GraphDB*.

Para um fácil uso e compatibilidade com os *standards* da indústria, o *GraphDB* implementou as interfaces da *framework RDF4J*, a especificação do protocolo W3C SPARQL[37] e suporta vários formatos de serialização RDF[38]. [9]

O *GraphDB* é um *plugin* SAIL para a *framework RDF4J* fazendo uso extensivo dos recursos e infraestrutura do *RDF4J* especialmente do modelo RDF, dos *parsers* RDF e os motores de pesquisa. [8]

Assim, o *GraphDB* possui uma REST API do servidor RDF4J[39] a partir da qual é possível obter todos os triplos através da rota `<url do GraphDB>/repositories/<id do repositório>/statements` indicando no cabeçalho HTTP *Accept* o formato de serialização RDF[38] de saída (*MIME type*[40]) dos triplos. Dos vários formatos de serialização RDF serão apenas suportados (acessíveis) na CLAV, como já indicado, o Turtle (`text/turtle`), o JSON-LD (`application/ld+json`) e o RDF/XML (`application/rdf+xml`).

Apesar da facilidade de exportação da ontologia estes pedidos de exportação originam um grande consumo de recursos de *hardware* por parte do *GraphDB* visto que cada pedido devolve todos os triplos de uma BD (a atual BD da CLAV possui já cerca de 150 000 triplos explicitos e cerca de 85 000 triplos implicitos) para além da conversão necessária desses triplos para o formato de serialização RDF de saída. Deve-se então limitar o número de pedidos de exportação realizados ao *GraphDB*. Para tal irá ser usado o seguinte mecanismo de controlo/*cache*:

- Os ficheiros exportados são mantidos pela API da CLAV
- Mantém-se dois ficheiros por cada serialização RDF, um com os triplos explicitos e outro com os triplos explicitos e implicitos.
- Se o ficheiro pretendido não existe na API da CLAV realiza-se o pedido de exportação ao *GraphDB*
- Se o ficheiro pretendido existe na API da CLAV mas não é atualizado há sete dias realiza-se o pedido de exportação ao *GraphDB*
- Se o ficheiro pretendido existe na API da CLAV e foi atualizado há menos de sete dias devolve-se ao utilizador o ficheiro guardado na API da CLAV
- Mantém-se na API da CLAV apenas o ficheiro mais recente para cada versão de cada serialização RDF

[36] Base de Dados (BD) Semântica baseada em grafos compatível com os padrões W3C. Suporta RDF e SPARQL

[37] Ver `https://www.w3.org/TR/sparql11-protocol/`

[38] *TriG*, *BinaryRDF*, *TriX*, *N-Triples*, *N-Quads*, N3, RDF/XML, RDF/JSON, JSON-LD e Turtle

[39] Ver `https://rdf4j.org/documentation/rest-api/`

[40] *Standard* que indica a natureza e o formato de um documento, ficheiro ou conjunto de *bytes*. Ver RFC 6838

- Cada ficheiro é apenas atualizado (removendo o antigo) quando é feito um pedido por um utilizador desse ficheiro

Assim, respeitando todas estas restrições, são mantidas pela API da CLAV no máximo seis ficheiros, dois por cada serialização RDF. Para além disso estes ficheiros são atualizados no melhor caso de sete em ete dias e no pior caso nunca se o ficheiro nunca for requisitado pelos utilizadores.

# 3

# CONCLUSÃO

# BIBLIOGRAFIA


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

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

[3] Auth0. Introduction to JSON Web Tokens, 2019. URL https://jwt.io/introduction/. Acedido a 2019-12-19.

[4] DGLAB. CLAV - Classificação e Avaliação da Informação Pública, 2019. URL http://clav.dglab.gov.pt. Acedido a 2019-12-15.

[5] Hal Lockhart, Thomas Wisniewski, Prateek Mishra, and Nick Ragouzis. *Security Assertion Markup Language(SAML) V2.0 Technical Overview*. OASIS, 7 2005.

[6] Alexandra Lourenço, José Carlos Ramalho, Maria Rita Gago, and Pedro Penteado. Plataforma CLAV: contributo para a disponibilização de dados abertos da Administração Pública em Portugal. Acedido a 2019-11-20, 7 2019. URL http://hdl.handle.net/10760/38643.

[7] Tim McLean. Critical vulnerabilities in JSON Web Token libraries, 3 2015. URL https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/. Acedido a 2019-12-22.

[8] Ontotext. Architecture & Components, 12 2019. URL http://graphdb.ontotext.com/documentation/free/architecture-components.html. Acedido a 2020-01-09.

[9] Ontotext. About GraphDB, 1 2020. URL http://graphdb.ontotext.com/documentation/free/about-graphdb.html. Acedido a 2020-01-09.

[10] Passport.js. Overview, 2019. URL http://www.passportjs.org/docs/. Acedido a 2019-12-17.

[11] Jeff Petters. What is SAML and How Does it Work?, 8 2018. URL https://www.varonis.com/blog/what-is-saml/. Acedido a 2019-12-26.

[12] Sebastián E. Peyrott. *The JWT Handbook*. 0.14.1 edition, 2018.

[13] Ryan Pinkham. What Is the Difference Between Swagger and OpenAPI?, 10 2017. URL https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/. Acedido a 2019-12-27.

[14] Kristopher Sandoval. Top Specification Formats for REST APIs, 3 2016. URL https://nordicapis.com/top-specification-formats-for-rest-apis/. Acedido a 2019-12-31.

[15] Swagger. What is Swagger?, 2019. URL https://swagger.io/tools/open-source/getting-started/. Acedido a 2019-12-27.

[16] Mika Tuupola. Branca as an Alternative to JWT?, 8 2017. URL https://appelsiini.net/2017/branca-alternative-to-jwt/. Acedido a 2019-12-22.

[17] Ivan Vasiljevic. Adding Swagger To Existing Node.js Project, 8 2017. URL https://blog.cloudboost.io/adding-swagger-to-existing-node-js-project-92a6624b855b. Acedido a 2019-12-28.