OpenAPI

Giuseppe Della Penna

Università degli Studi di L'Aquila
giuseppe.dellapenna@univaq.it
http://people.disim.univaq.it/dellapenna

> *Questo documento si basa sulle slide del corso di Sviluppo Web Avanzato, riorganizzate per migliorare l'esperienza di lettura. Non è un libro di testo completo o un manuale tecnico, e deve essere utilizzato insieme a tutti gli altri materiali didattici del corso. Si prega di segnalare eventuali errori o omissioni all'autore.*

1. Struttura di una Specifica OpenAPI

Una specifica OpenAPI 3.0 è composta da otto sezioni:

- `openapi`
- `info`
- `tags`
- `externalDocs`
- `servers`
- `paths`
- `components`
- `security`

2. Oggetto globale openapi

```yaml
openapi: "3.0.2"
```

L'oggetto **openapi** indica la versione della specifica OpenAPI utilizzata.

3. Oggetto globale info

```yaml
info:
 title: "API di Esempio"
 description: "...."
 version: "2.5"
 termsOfService: "https://dellapenna.univaq.it/terms"
 contact:
  name: "API di Esempio"
  url: "https://dellapenna.univaq.it/info"
  email: "giuseppe.dellapenna@univaq.it"
 license:
  name: "CC Attribution-ShareAlike4.0(CC BY-SA 4.0)"
  url: "https://dellapenna.univaq.it"
```

L'oggetto `info` contiene informazioni descrittive sulla API quali il titolo, la versione, la licenza, ecc. La maggior parte delle proprietà è opzionale.

Si veda l'esempio per i dettagli sulle informazioni definibili in questa struttura.

Nella proprietà description è possibile utilizzare il linguaggio CommonMark (https://commonmark.org/) per formattare il testo.

4. Oggetto globale tags

```yaml
tags:
 - name: Users
   description: User-related operations
 - name: Pets and other animals
   description: operations on animals
```

I tag vengono associati alle operazioni, come vedremo, e permettono di organizzarle visivamente in gruppi nelle interfacce grafiche.

Non è richiesto di pre-dichiarare i tag che verranno assegnati alle operazioni, ma facendolo è possibile chiarirne la semantica e imporre un ordine di presentazione.

Per dichiarare un tag si usa la struttura `tags`, che contiene un array di oggetti aventi proprietà

- `name`: nome del tag.
- `description`: descrizione del tag.
- `externalDocs`: permette di collegare informazioni esterne al tag (si veda la struttura externalDocs nelle prossime slides)

5. Oggetto globale externalDocs

```yaml
externalDocs:
 url: http://www.example.com
 description: Info about this specification
```

La struttura `externalDocs` permette di collegare una sorgente di informazioni esterna relativa alla specifica (o a sue parti, se la struttura è nidificata ad esempio in un tag, come visto in precedenza).

Per dichiarare tale riferimento usa un oggetto avente proprietà

- `url`: collegamento esterno.
- `description`: descrizione del documento.

6. Oggetto globale servers

```yaml
servers:
 - url: "https://dellapenna.univaq.it/api/2.5/"
   description: testing server
 - url: "https://dellapenna.univaq.it/api/beta/"
   description: beta server
 - url: "https://{hostID}.univaq.it:{port}/api"
   variables:
    hostID:
     default: dellapenna
     description: host di produzione
    port:
     default:"443"
     enum:
      - "443"
      - "8443"
```

Con l'oggetto `servers` si specificano i path di base utilizzati dalle API. I path delle operazioni saranno tutti relativi a questi path di base.

È anche possibile aggiungere una struttura server all'interno del path di un'operazione per specificare un **set di server specifico per un determinato path**, diverso dal set globale qui specificato.

Gli URL dei server possono essere **relativi**. In tal caso, l'URL viene risolto rispetto al server che ospita la specifica.

È possibile specificare più URL e fornire una descrizione di ciascuno.

È possibile usare nelle URL delle **variabili** che potranno essere assegnate a runtime. Qualsiasi parte dell'URL (schema, host o sue parti, porta, elementi del path) può essere parametrizzata utilizzando variabili. Le variabili sono indicate da parentesi graffe.

È possibile assegnare alle variabili un valore di default e/o un set di valori fissi, oltre a una descrizione.

7. Oggetto globale components

L'oggetto `components` permette di raccogliere le definizioni di varie componenti della specifica, suddivise per tipologia, riutilizzandole poi ove necessario tramite un semplice riferimento.

Questa pratica è utile sia per poter *riutilizzare* elementi comuni, come parametri o risposte, senza doverli duplicare, sia per rendere la struttura delle singole operazioni meno complessa, ad esempio spostando all'esterno lo schema dei suoi dati.

Le sotto-sezioni di components, che contengono gli oggetti definibili come componenti, sono:

- **`schemas`**
- **`parameters`**

- **`responses`**
- **`requestBodies`**
- **`securitySchemes`**
- `headers`
- `examples`
- `links`
- `callbacks`

(descriveremo qui solo le più importanti, in grassetto)

La definizione di un oggetto di un certo tipo all'interno delle sezioni di `components` è identica a quella che avrebbe se definito direttamente ove riferito, tuttavia in questo caso all'oggetto deve essere anche attribuito un **nome** univoco, che verrà poi usato per farvi riferimento.

Per far riferimento a un componente, si sostituisce alla sua definizione (ovunque sia richiesta) una **reference**, *scritta secondo la notazione JSON Reference* (anche se stiamo scrivendo in YAML!), che "punta" alla effettiva definizione specificando il relativo path, tipicamente all'interno del file corrente (#), ad esempio `$ref: '#/components/schemas/User'`

7.1. Schemi (schemas)

```yaml
UnitType:
  type: string
  enum:
    - standard
    - metrico
    - imperiale
  default: "metrico"

UUID:
  type: string
  format: uuid

GreaterThenZeroInt:
 type: integer
 minimum: 1
```

Tutte le informazioni scambiate dale operazioni della API devono essere tipate. Per definire questi tipi **si usano gli Schemi JSON**, per la cui sintassi si rimanda alla relative slides.

La specifica OpenAPI estende quella degli schemi JSON permettendo l'uso dei seguenti formati per i relative tipi:

- `integer`: `int32`, `int64`
- `number`: `float`, `double`

- `string` : `date-time` , `password` , `byte` , `binary` (utile per modellare il tipo file, ad esempio)

Altri formati possono essere usati nel caso il software li riconosca.

7.2. Parametri (parameters)

```yaml
CAP:
 name: cap
 in: query
 description: "Search by CAP"
 schema:
  type: string
  example: "67100"
Unit:
 name: unit
 in: query
 schema:
  type: string
  enum:
   - standard
   - metrico
   - imperiale
  default: "metrico"
RequestID:
 in: header
 name: X-Request-ID
 schema:
  type: string
  format: uuid
 required: true
UserID:
 in: path
 name: userID
 required: true
 schema:
  type: integer
  minimum: 1
 description: The user ID
 examples:
  oneId:
   summary: Example of ID
   value: 32032
  anotherId:
   summary: Another example of ID
   value: 128387
CRSFToken:
 in: cookie
 name: csrftoken
 schema:
  $ref: '#/components/schemas/Token'
```

Ogni parametro di una richiesta RESTful è definito tramite le seguenti proprietà:

- `name`: nome del parametro (*da non confondersi col nome del componente*: il nome del parametro è quello che dovrà effettivamente comparire nelle url, nei cookie, ecc.).
- `in`: dove compare il parametro. I valori possibili sono *header, path, query e cookie* .
- `description`: descrizione del parametro.
- `required`: indica se il parametro è obbligatorio (da specificare con valore true se il parametro è nel path, altrimento può essere omessa e ha default falso).
- `schema`: un oggetto che indica il tipo ammesso per il parametro tramite il formalismo JSON schema, *o un riferimento a una componente di tipo schema*.
- `example/examples`: un esempio di valore per il parametro/un oggetto contenente vari esempi di valori per il parametro.
- `deprecated`: indica se il parametro è deprecato.
- `allowEmptyValue`: indica se il parametro può avere un valore vuoto.
- `style`: come vengono serializzati i dati del parametro (cioè come strutture quali array o oggetti vengono convertite in stringhe parametro). Si veda più avanti.
- `explode`: parametro avanzato relativo alla serializzazione di stile *form*. Si veda più avanti.
- `allowReserved`: indica se i caratteri riservati sono consentiti.

Serializzazione parametri

Il modo in cui dovrebbero essere serializzate strutture complesse quali array e oggetti all'interno delle stringhe dei parametri dipende dai valori degli attributi `style` e `explode`:

Per gli array *(ad esempio [blue,green,red])*

- `style=simple` (solo per i parametri path e header): `blue,green,red`
- `style=form` (solo per i parametri query e cookie):
  - `color=blue,green,red` (con `explode=false`)
  - `color=blue&color=green&color=red` (con `explode=true`)
- `style=spaceDelimited` (solo per i parametri query): `color=blue%20green%20red`
- `style=pipeDelimited` (solo per i parametri query): `color=blue|green|red`

Per gli oggetti *(ad esempio {R:100, G:200, B:150})*

- `style=simple` (solo per i parametri path e header): `R=100,G=200,B=150`
- `style=form` (solo per i parametri query e cookie)
  - `color=R,100,G,200,B,150` (con *explode=false*)
  - `R=100&G=200&B=150` (con *explode=true*)
- `style=deepObject` (solo per i parametri query): `color[R]=100&color[G]=200&color[B]=150`

Il default è *form* per i parametri di query e cookie e *simple* per quelli di path e header

Esistono altri stili di serializzazione che non tratteremo qui (si veda https://swagger.io/specification/#parameter-style).

7.3. Risposte (responses)

```
TwoDataResponse:
 description: Successful response
 content:
  application/json:
   schema:
    type: object
    properties:
     data1:
      type: integer
      description: response data 1
     data2:
      type: number
      description: response data 2
 headers:
  X-RateLimit-Limit:
   schema:
    type: integer
    description: Request limit per hour.

NotFound:
  description: Not found response
  content:
   text/plain:
    schema:
     type: string
     example: Not found

ImageResponse:
 description: An image.
 content:
  image/*:
   schema:
    type: string
    format: binary

GenericError:
 description: An error occurred.
 content:
  application/json:
   schema:
    $ref: '#/components/schemas/Error'
```

Le responses rappresentano i contenuti generabili dal servizio.

- `description`: descrizione della risposta.

- `content`: il contenuto effettivo della risposta (o meglio la sua struttura). *Omesso se la risposta è vuota*.
- `headers`: elenca gli eventuali header da inserire nella risposta. *Omesso se non ci sono header da aggiungere*.

Una risposta può prevedere diversi *media types*, e per ciascuno di essi la risposta stessa può avere una struttura diversa. Per questo motivo, il `content` della risposta è un oggetto che elenca i possibili media types e, sotto ciascuno di essi, la struttura della relativa risposta

Gli header, se presenti, sono invece indipendenti dal media type e riportati nell'omonima sezione, ciascuno col suo nome e il suo schema.

- `schema`: un oggetto che indica il tipo ammesso per il contenuto/header tramite il formalismo JSON schema.

Sia per gli esempi che per gli schemi è ovviamente possibile far riferimento ad altre componenti.

7.4. Richieste (requestBodies)

```yaml
PetRequest:
 description: the name of a pet
 content:
  application/xml:
   schema:
    $ref: '#/components/schemas/PetForXml'
  application/json:
   schema:
    $ref: '#/components/schemas/Pet'
   examples
    dog:
     summary: An example of a dog
     value:
      name: Fluffy
      petType: dog
    hamster:
     $ref: '#/components/examples/hamster'

ImageBinaryRequest:
 content:
  image/png:
   schema:
    type: string
    format: binary

MultipartRequest:
 content:
  multipart/form-data:
   schema:
    type: object
    properties:
     orderId:
      type: integer
     userId:
      type: integer
     fileData:
      type: string
      format: binary
```

I request body rappresentano i dati inviati come *payload* delle richieste POST, PUT, ecc.

- `description`: descrizione del request body.
- `content`: il contenuto effettivo del request body (o meglio la sua struttura).

Una richiesta può essere codificata tramite diversi *media types*, e per ciascuno di essi la richiesta stessa può avere una struttura diversa. Per questo motivo, il `content` della richiesta è un oggetto che elenca i possibili media types e, sotto ciascuno di essi, la struttura della relativa richiesta.

- `schema`: un oggetto che indica il tipo ammesso per la richiesta/header tramite il formalismo

JSON schema.

- `examples`: un oggetto che può essere usato per fornire degli esempi concreti di contenuto per la richiesta, strutturato in base allo schema.

Sia per gli esempi che per gli schemi è ovviamente possibile far riferimento ad altre componenti.

7.5. Schemi di sicurezza (securitySchemes)

```yaml
BasicAuth:
 type: http
 scheme: basic

BearerAuth:
 type: http
 scheme: bearer
 bearerFormat: internalToken

ApiKeyAuth:
 type: apiKey
 in: header
 name: X-API-Key

OpenID:
 type: openIdConnect
 openIdConnectUrl: https://example.com/openid

OAuth2:
 type: oauth2
 flows:
  authorizationCode:
   authorizationUrl: https://example.com/oauth/auth
   tokenUrl: https://example.com/oauth/token
   scopes:
    read: Grants read access
    write: Grants write access
    admin: Grants access to admin operations
```

I security schemes definiscono le politiche di sicurezza dell'API.

- `type`: tipo di schema di sicurezza: `http`, `apiKey`, `openIdConnect`, `oauth2`.

Le altre proprietà dello schema di sicurezza dipendono dal suo tipo. Ad esempio, per il tipo `http`, è possibile usare la proprietà `scheme` per indicare se si intende usare la `basic` *http authentication* oppure la `bearer` (connessa all'header Authentication della richiesta).

Per applicare uno o più schemi di sicurezza alla API è necessario inserire la struttura security all'interno delle singole operazioni o a livello radice (per applicare lo schema a tutte le operazioni), come vedremo più avanti.

8. Oggetto globale paths

```yaml
paths:
 /operation1:
  get:
   ...
 /users/{id}:
  get:
   ...
  put:
   ...
  delete:
   ...
 /data/{pc1}/info/{pc2}:
   post:
    ...
```

L'oggetto paths dichiara i **percorsi** (*endpoint*) **delle operazioni** del servizio RESTful.

Ogni elemento della struttura paths è la **URL** (*relativa all'indirizzo base del server*) di un endpoint.

I path possono contenere **parametri**, indicati da sezioni racchiuse tra partenesi graffe. All'interno della parentesi è posto il nome del parametro, che verrà usato per definirne la struttura all'interno dell'operazione.

Ciascuna URL **avrà tanti figli quanti sono i verbi HTTP (GET, PUT, POST...) utilizzabili** con essa.

La struttura nidificata in ciascun verbo, infine, descrive in dettaglio le caratteristiche dell'endpoint tramite i seguenti elementi (descriveremo qui solo i più importanti, in grassetto)

- **`operationId`**
- **`tags`**
- **`description`**
- **`summary`**
- **`parameters`**
- **`responses`**
- **`requestBody`**
- **`security`**
- `externalDocs`
- `deprecated`
- `servers`
- `callbacks`

8.1. Elementi descrittivi

```
paths:
 /users/{id}:
  get:
   operationId: getUserById
   tags:
    - Users
   summary: Gets a user by ID.
   description: A detailed description of the operation.  Use markdown for rich text represent
   ...
```

- `operationId`: un identificatore univoco per l'endpoint. Spesso corrisponde al nome del metodo che lo implementerà nel codice.
- `tags`: utilizzati per raggruppare gli endpoint in base a delle parole chiave (nelle interfacce grafiche).
- `summary`: una breve descrizione dell'endpoint.
- `description`: una descrizione completa dell'endpoint.

8.2. Parametri (parameters)

```
paths:
 /users/{id}:
  parameters:
   - $ref: '#/components/parameters/RequestID'
  get:
   ...
   parameters:
    - name: optionalParam
      in: query
      description: "An optional query parameter"
      schema:
       type: string
    - $ref: '#/components/parameters/UserID':
    - $ref: '#/components/parameters/CRSFToken'
  post:
   ...
```

L'oggetto `parameters` contiene un array di oggetti rappresentanti i singoli parametri dell'operazione

I parametri possono essere posti sulla URL, nella query string, negli header e nei cookie.

È possibile specificare parametri a livello di path e/o a livello di singolo verbo, in modo da fattorizzare quelli comuni (ad esempio presenti nel path)

I parametri sono definiti come visto nella sezione componenti. È possibile incorporarne la definizione nell'operazione o, in modo più modulare, fare un riferimento alla componente corrispondente.

8.3. Risposte (responses)

```yaml
paths:
 /users/{id}:
  get:
   ...
   responses:
    '200':
     description: Successful operation
     content:
      application/json:
       schema:
        $ref: '#/components/schemas/User'
    '404':
      $ref: '#/components/responses/NotFound'
   ...
```

L'altra parte essenziale per ogni operazione è l'oggetto `responses`.

Vengono elencate le strutture di tutte le risposte generabili dall'operazione, **una per ciascun codice di stato http restituibile**.

Le risposte sono definite come visto nella sezione componenti. È possibile incorporarne la definizione nell'operazione o, in modo più modulare, fare un riferimento alla componente corrispondente.

8.4. Richieste (requestBodies)

```yaml
paths:
 /pets:
  post:
   ...
   requestBody:
    $ref: '#/components/requestBodies/PetRequest'
   ...
```

Quando l'operazione richiede un *payload* (POST, PUT, ecc.), la struttura `requestBody` permette di specificarne la struttura e il tipo.

Il requestBody è definito come visto nella sezione componenti. È possibile incorporarne la definizione nell'operazione o, in modo più modulare, fare un riferimento alla componente corrispondente.

8.5. Sicurezza (securitySchemes)

```yaml
paths:
 /pets:
  post:
   ...
  security:
   - BearerAuth: []
   - OAuth2: [admin]
 /ping:
  get:
   ...
   security: []   # No security
```

Per attivare uno schema di sicurezza (definito nella sezione componenti) su un singolo endpoint (o fare override di quanto dichiarato globalmente) è possibile inserire la struttura `security`.

La struttura contiene una lista di schemi (o meglio dei loro nomi così come definiti tra le componenti) che verranno usati in maniera alternativa (OR) per autorizzare l'accesso all'operazione (è possibile anche specificare una combinazione di schemi in AND, ma qui non la tratteremo).

Ogni schema ha come parametro **un array di ruoli**, che è vuoto per gli schemi il cui tipo non li preveda

Un **array vuoto** di schemi disabilita la sicurezza su un certo endpoint anche se questa è attiva a livello globale.

9. Oggetto globale security

```yaml
security:
 - BearerAuth: []
 - OAuth2: [admin]
```

Per attivare uno schema di sicurezza (definito nella sezione componenti) su tutti gli endpoint possibile inserire la struttura **security a livello globale**.

La struttura contiene una lista di schemi (o meglio dei loro nomi così come definiti tra le componenti) che verranno usati in maniera alternativa (OR) per autorizzare l'accesso all'operazione (è possibile anche specificare una combinazione di schemi in AND, ma qui non la tratteremo).

Ogni schema ha come parametro **un array di ruoli**, che è vuoto per gli schemi il cui tipo non li preveda

10. Riferimenti

Specifica OpenAPI

https://www.openapis.org/

Tutorial per OpenAPI3
https://support.smartbear.com/swaggerhub/docs/tutorials/openapi-3-tutorial.html

OpenAPI editor e generatore di documentazione
https://swagger.io/tools/swagger-editor/

Title: OpenAPI
Author: Giuseppe Della Penna*, University of L'Aquila*
Version: 20240221