

Lab Hours - Lecture 1

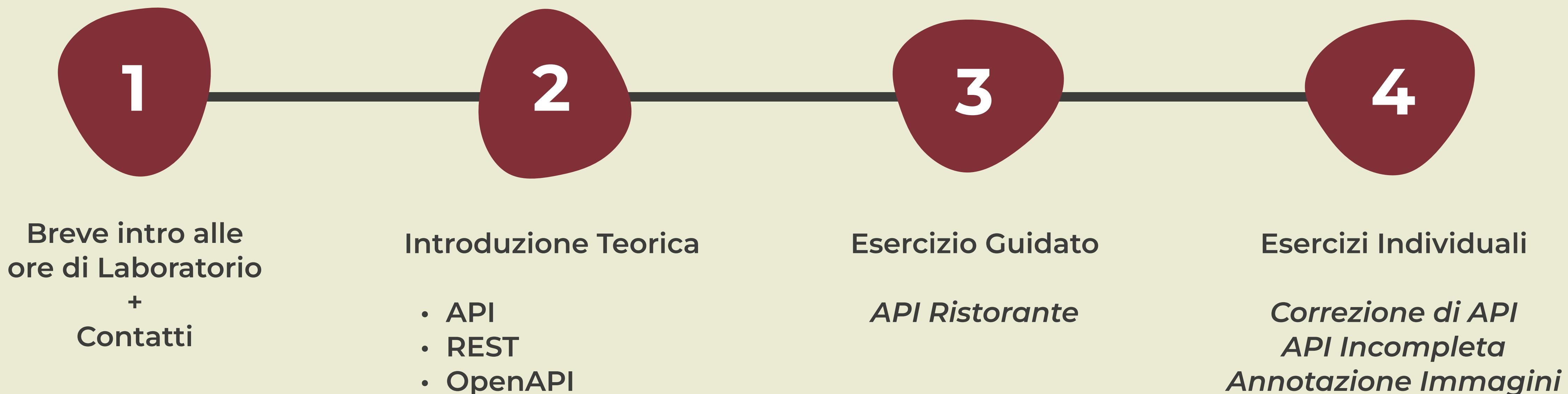
OpenAPI

Come descrivere una **RESTful API**

```
1  openapi: 3.0.3
2  info:
3    title: Swagger Petstore - OpenAPI 3.0
4    description: ""
5    termsOfService: http://swagger.io/terms/
6    contact:
7      email: apiteam@swagger.io
8    license:
9      name: Apache 2.0
10     url: http://www.apache.org/licenses/LICENSE-2.0.html
11     version: 1.0.11
12   externalDocs:
13     description: Find out more about Swagger
14     url: http://swagger.io
15   servers:
16     - url: https://petstore3.swagger.io/api/v3
17   tags:
18     - name: pet
19       description: Everything about your Pets
20   externalDocs:
21     description: Find out more
22     url: http://swagger.io
23     - name: store
24       description: Access to Petstore orders
```

Contact: ellepuntopi.98@gmail.com

Programma di oggi:



Struttura Generale

Lo scopo di queste ore di Laboratorio è fornirvi dimostrazioni pratiche riguardo l'utilizzo delle tecnologie e degli strumenti introdotti dai docenti.

Ogni lezione seguirà (se possibile) la stessa struttura di base, composta da:

- Breve recap teorico sul tema della lezione
- Esercizio guidato
- Esercizi individuali

Se dovete avere bisogno di ulteriori esercizi inviate una mail (contatti nella prossima slide).

Per domande più “articolate” preferirei aspettaste la pausa (**45min di lezione + 15min di pausa**) o la fine della lezione.

Contatti

API

Sono **Lorenzo Paolini**, studente magistrale di DHDK (ancora per una settimana - evviva), e sarò il vostro tutor per queste lezioni di Laboratorio.

Questa è la mia prima esperienza da Tutor, quindi se dovessero esserci *incomprensioni* o problemi di ogni tipo vi prego di farmelo sapere, magari a fine lezione. Proverò a rispondere a qualsiasi domanda ma, se non dovesse essere *in grado di fornire una risposta nell'immediato*, ritornerò sulla domanda durante la lezione successiva, abbiate pazienza. **Feedback di ogni tipo sono molto apprezzati.**

Al di fuori delle ore di laboratorio potete **contattarmi in queste modalità:**

1. Via mail all'indirizzo: **ellepuntopi.98@gmail.com** [preferita - generalmente rispondo entro 3 giorni]
2. Via mail all'indirizzo: **lorenzo.paolini6@studio.unibo.it** [se non dovesse rispondere alla mail inviata all'indirizzo sopra entro 3 giorni]
3. Gruppo **Telegram** (prossima slide), utile per informazioni generali [se non dovesse rispondere alle vostre email (magari finiscono nello spam), potete inviare la domanda direttamente nel gruppo]
4. **NO TEAMS**

Gruppo Telegram

Scannerizzate il QR code per entrare nel gruppo. Vi chiedo la cortesia di **non condividere il link al gruppo a persone esterne al corso.**

Il link rimarrà attivo, dunque il gruppo accessibile, solo per la durata di questa lezione. Successivamente, se qualcuno avesse bisogno di essere inserito nel gruppo può contattarmi via mail agli indirizzi forniti prima.

Per favore, siate **rispettosi** e tenete a mente che **ogni comportamento o messaggio “sbagliato” - o comunque non inherente al gruppo - verrà immediatamente ripreso e segnalato**, grazie.

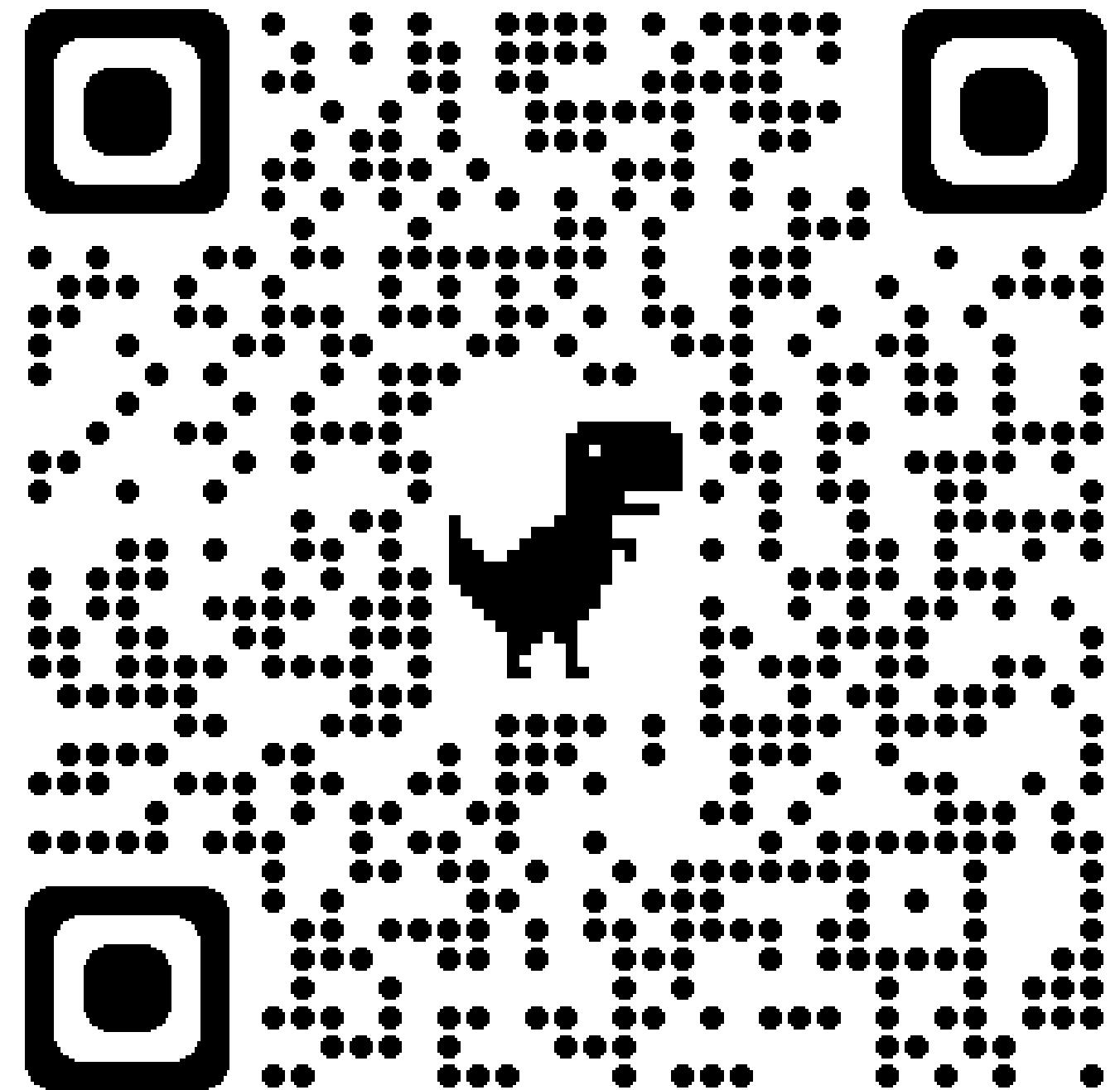


GitHub

Non so quanto conosciate **GitHub**, ma è lo strumento che, assieme a Virtuale, utilizzerò per fornirvi esercizi, slide, ed esercitazioni aggiuntive.

Vi chiedo di **creare un account se già non lo avete, e poi accedere alla repository del corso**, che contiene al momento le slide della lezione di oggi, e gli esercizi (testo e codice). Mano a mano che completeremo le varie parti, andrò ad aggiornare la repo aggiungendo soluzioni e codici.

Tendenzialmente proverò a rendere disponibile tutto il materiale il giorno stesso della lezione.



COSA SONO LE API



Pensate ad un'applicazione (o sito web) come ad un *ristorante*.

Il cliente arriva, si siede, e sfoglia il menu. Nel menu sono presenti diversi piatti, ognuno con una sua descrizione. Una volta scelto il piatto, il cliente fa il suo ordine, e la cucina prepara il cibo richiesto.

In questa analogia:

- La cucina rappresenta il server (sistema back-end) dell'applicazione
- Il cliente rappresenta il client (l'utente dell'applicazione, o meglio l'applicazione stessa)
- Il menu rappresenta invece l'API (*Application Programming Interface*)

Le API sono un modo per due sistemi informatici di comunicare tra loro. Così come il menu permette a voi (i clienti) di capire cosa si può ordinare e cosa si riceverà in cambio, un'API dice a un software come interagire con un altro software, fornendo dunque una serie di operazioni che gli sviluppatori possono utilizzare, insieme a una descrizione di cosa fanno queste operazioni.

RESTful API & CRUD

Definizione: Una RESTful API descrive un'interfaccia che permette la comunicazione tra diversi sistemi software attraverso protocolli di trasporto (HTTP), seguendo i principi del REST (Representational State Transfer), e basando su di essi l'architettura:

- Definisce "risorsa" ogni concetto rilevante dell'applicazione Web
- Associa un URI come identificatore e selettore primario
- Utilizza i verbi HTTP per esprimere ogni operazione dell'applicazione secondo il modello CRUD:
 - **Create** = metodo **POST**
 - **Read** = metodo **GET**
 - **Update** = metodo **PUT**
 - **Delete** = metodo **DELETE**
- Esprime in maniera parametrica ogni rappresentazione dello stato interno della risorsa, personalizzabile dal richiedente attraverso un Content Type preciso. Dunque, visto che una stessa risorsa potrebbe essere disponibile in più formati, come JSON, XML, YAML, o anche HTML, a seconda delle necessità del client, una RESTful API è in grado di fornire diverse rappresentazioni dello stesso stato di una risorsa.

Come descrivere una RESTful API

Swagger & OpenAPI

Per documentare un API è necessario definire:

- Un **end-point** (URI/route), separando collezioni e individui
- Dei **metodi HTTP** di accesso (GET, PUT, POST, DELETE, ecc...)
- Delle **rappresentazioni di Input e Output** (solitamente non si utilizza un linguaggio di schema, ma un esempio fittizio e sufficientemente strutturato)
- Delle **condizioni di errore**, assieme ai **messaggi** che restituiscono

Per gli esercizi utilizzeremo **Swagger** (<https://swagger.io/tools/swagger-editor/>), un editor per lo sviluppo di API REST.

N.B.: Durante l'esame non avrete accesso a Swagger, ma ad un semplice editor di testo!

Una buona risorsa per avere un'**overview generale sulle RESTful API** è consultabile al link:
<https://www.ibm.com/it-it/topics/rest-apis>

Esercizio Guidato - Ristorante e Menù

API

Progettare un'API REST (parziale) per la gestione di un *ristorante* e descriverla in [Swagger/OpenAPI](#).

Il ristorante offre **menù diversi**, ognuno caratterizzato da un **ID** e una **descrizione testuale**; ogni menù include **diversi piatti**, ognuno caratterizzato da un **ID**, una **descrizione testuale** e un **prezzo**.

-> **Tutti gli attributi sono obbligatori.**

Scrivere un file in formato **JSON** o **YAML**.

L'API permette di:

- **ottenere l'elenco di tutti i menù`**
- **ottenere le informazioni di uno specifico menù** (*ID* e *descrizione*, [senza elenco piatti](#))
- **aggiungere un nuovo piatto** ad un menù

Specificare: [URL](#) di accesso, [metodi HTTP](#), parametri e risposte con esempi.

Note:

- Gestire gli errori con status code 200 e 400 per tutte le richieste
- Non è richiesto includere le sezioni servers, tags
- Non è richiesto gestire autenticazione

Esercizio Guidato - Ristorante e Menù (1): Basi

API

Iniziamo dalla definizione di alcuni elementi base dell'API:

- **swagger**: '**2.0**': qui definiamo la versione di Swagger (o OpenAPI dal 3.0.0) che andremo ad utilizzare. **Tutti gli esercizi saranno, per conformità con esempi forniti dal professore, in swagger 2.0.** Se volete approfondire struttura e sintassi di OpenAPI siete liberi di farlo, se avete domande inviate pure una mail.
- **info**: è un oggetto che contiene informazioni di base sull'API, ed è composto da:
 - **description**: breve descrizione testuale dell'API.
 - **title**: nome dell'API.
 - **version**: La versione dell'API che state descrivendo. La notazione sulle versioni segue il **Semantic Versioning (SemVer)**, che è *uno schema di versionamento* molto usato nell'industria del software. Il **SemVer** è formato da tre parti: **MAJOR.MINOR.PATCH** (per approfondire <https://semver.org/>).

```
1  swagger: '2.0'  
2  info:  
3    description: 'API gestione ristorante e menù'  
4    title: 'Esercizio ristoranti'  
5    version: 1.0.0
```

Esercizio Guidato - Ristorante e Menù (2): Paths

Una volta definita la base dell'API, iniziamo a definire i vari **paths** richiesti. Questa sezione definisce gli *endpoint* dell'API e come essi devono essere utilizzati.

Come da specifiche, in questa fase iniziamo a definire il path più generale, ovvero quello che ci permette di accedere all'*elenco dei menù disponibili*.

La prima cosa da definire, a seguito della definizione del path, è il metodo HTTP che tale percorso supporta. In questo caso il metodo **GET**, utilizzato per richiedere dati da un'API, è quello che ci permette di ottenere l'*elenco dei menu*.

Dovremo successivamente specificare un **summary**, un **operationId**, e gestire le risposte secondo le casistiche di **successo (200)** ed **errore (400)**.

In caso di successo, dovremo definire uno schema per la risposta.

```
6  paths:
7    /menus/:
8      get:
9        summary: 'Elenco dei menu'
10       operationId: 'listamenu'
11       responses:
12         '200':
13           description: 'successo'
14           schema:
15             type: array
16             items:
17               $ref: '#/definitions/Menu'
18         '400':
19           description: 'Errore richiesta'
```

Esercizio Guidato - Ristorante e Menù (3): Schema

API

Dunque, per quanto riguarda lo **schema**, questo definisce la struttura dei dati restituiti da un'operazione API specifica, in questo caso da una richiesta GET all'endpoint `/menus/`.

Il **type** sta ad indicare il formato dati che vogliamo ottenere in output, in questo caso specifico un **array**. In altre parole, il server restituirà una lista di elementi.

La keyword **items** specifica invece il tipo di elementi contenuti nell'array, definendo la struttura (o lo schema) di ciascun elemento dell'array.

Infine, **\$ref: '#/definitions/Menu'** è particolarmente importante perché introduce il concetto di riferimenti all'interno delle specifiche OpenAPI (Swagger). Il **\$ref** è un operatore di riferimento **JSON** che indica che la definizione dell'oggetto viene riutilizzata da un'altra parte della specifica. In questo caso, si fa riferimento a una definizione denominata Menu, che andremo a specificare nella sezione **definitions** (successivamente).

```
14 |           schema:  
15 |             type: array  
16 |             items:  
17 |               $ref: '#/definitions/Menu'
```

Esercizio Guidato - Ristorante e Menù (4): Riferimenti

API

'#/definitions/Menu' è un riferimento a un percorso all'interno del documento. Il simbolo # indica l'inizio di un riferimento interno, seguito dal percorso. **Menu** è il nome della definizione specifica all'interno della sezione **definitions**.

definitions è una sezione standard in una specifica OpenAPI e serve come una specie di "libreria" di schemi riutilizzabili.

Un oggetto complesso (e.g., un Menu, che potrebbe avere diverse proprietà come id, descrizione, ecc...), deve essere definito una sola volta nella sezione **definitions** e poi, ogni volta che avrete bisogno di specificare che qualcosa rappresenta quell'oggetto, fare riferimento a tale definizione. Ciò assicura che la specifica sia più organizzata, più facile da leggere e da mantenere, e che si evitino ripetizioni.

Dunque, **\$ref: '#/definitions/Menu'** dice "vai alla sezione **definitions** di questo documento e usa lo schema definito sotto Menu".

Esercizio Guidato - Ristorante e Menù (5): Altri path

API

Andiamo ora a definire un altro **path**, quello che ritorna le informazioni di uno specifico menù.

In questo caso il path da definire è quello che accede - gerarchicamente - al menù singolo del quale vogliamo ottenere informazioni. Anche qui utilizzeremo il metodo **GET** dunque.

Rispetto al path precedente, questo snippet ha due differenze principali, riuscite a capire quali e a spiegarmi il perché?

```
20  /menus/{menuId}:
21    get:
22      description: 'Informazioni su un menù'
23      operationId: 'infomenu'
24      parameters:
25        - name: menuId
26          in: path
27          description: "ID del menu"
28          required: true
29          type: integer
30      responses:
31        '200':
32          description: 'successo'
33          schema:
34            $ref: '#/definitions/Menu'
35        '400':
36          description: 'ID non valido'
```

Esercizio Guidato - Ristorante e Menù (6): Altri path

API

Esatto (?), gli elementi differenti sono:

- La presenza della sezione **parameters**, e
- L'assenza del **type** nello **schema** della risposta positiva (**200**).

Partiamo dalla differenza più semplice da spiegare.

L'assenza del **type** nello **schema** della risposta positiva (**200**) è semplicemente dovuta al fatto che vogliamo come output un singolo menù.

Dunque, avendo parlato della sezione **definitions**, e avendo detto che ci è utile nel richiamare definizioni ed evitare ripetizioni, è facile capire che in questo caso specifico stiamo semplicemente restituendo un oggetto complesso (**Menu**) che abbiamo già definito (non ancora in realtà, ma lo definiremo a brevissimo).

Dunque, il tipo di oggetto che vogliamo ci venga restituito è richiamato e referenziato dal **\$ref**.

```
20  /menus/{menuId}:
21    get:
22      description: 'Informazioni su un menù'
23      operationId: 'infomenu'
24      parameters:
25        - name: menuId
26          in: path
27          description: "ID del menu"
28          required: true
29          type: integer
30      responses:
31        '200':
32          description: 'successo'
33          schema:
34            $ref: '#/definitions/Menu'
35        '400':
36          description: 'ID non valido'
```

Esercizio Guidato - Ristorante e Menù (7): Altri path parameters

API

Parliamo ora della sezione **parameters**. Questa sezione viene utilizzata per descrivere i parametri che possono essere passati all'API. Ogni parametro ha diverse proprietà che ne definiscono l'uso, il tipo, se è obbligatorio, e altre informazioni.

Quando si definisce un parametro con **in: path**, si sta specificando che il parametro è parte dell'URL della richiesta. **name: menuId** rappresenta quindi il nome del parametro, che dovrà corrispondere alla variabile del path, quella all'interno delle graffe **{menuId}**.

description fornisce una breve descrizione del parametro, utile per la documentazione.

required: true indica invece che il parametro è obbligatorio. Una richiesta senza questo parametro sarà considerata non valida.

type serve a definire il tipo di dato del parametro. In questo caso, **menuId** dovrà quindi essere un intero (**integer**).

24	parameters:
25	- name: menuId
26	in: path
27	description: "ID del menu"
28	required: true
29	type: integer

Esercizio Guidato - Ristorante e Menù (8): Altri path

API

Andiamo ora a definire l'ultima funzionalità richiesta, ovvero *l'aggiunta di un piatto all'interno di uno specifico menù.*

La prima differenza rispetto ai due path precedenti è rappresentata dal metodo HTTP utilizzato, ovvero **POST**, comunemente utilizzato per operazioni di tipo **Create**. In questo caso, infatti, *stiamo creando una nuova entità di tipo Piatto*.

Un altro elemento chiave dello snippet è rappresentato dalla presenza di *due parametri obbligatori*. Mentre, come prima, il parametro **menuId** è definito all'interno del path; il secondo parametro, ovvero **Piatto**, è definito all'interno del **body** della richiesta e *deve seguire lo schema specificato in #/definitions/Piatto* (che vedremo tra pochissimo).

```
37  /menus/{menuId}/piatti/:
38    post:
39      summary: 'Aggiunge un piatto ad un menu'
40      operationId: 'aggiungipiatto'
41      parameters:
42        - name: menuId
43          in: path
44          description: "ID del menu"
45          required: true
46          type: integer
47        - name: piatto
48          in: body
49          description: 'Piatto da aggiungere'
50          required: true
51          schema:
52            $ref: '#/definitions/Piatto'
53      responses:
54        '200':
55          description: 'successo'
56        '400':
57          description: 'ID non valido'
```

Esercizio Guidato - Ristorante e Menù (9): Differenza tra PUT e POST

Come abbiamo visto nella slide precedente, in questo caso utilizziamo il metodo **POST**, utile alla creazione di una nuova risorsa. Ma, qual è la logica dietro all'utilizzo di tale metodo? In che modo differisce dal **PUT**?

- **Create:** Il metodo **POST** è comunemente usato per creare nuove risorse. Quando inviate una richiesta **POST**, state chiedendo al server di accettare la nuova entità fornita e di assegnarle un identificatore univoco. Il server decide l'URI della nuova risorsa. **POST non è idempotente**, perciò ogni richiesta risulterà generalmente nella creazione di una nuova risorsa.
- **Update:** Il metodo **PUT** è usato invece per aggiornare una risorsa esistente, ma può essere impiegato anche per crearne una nuova all'URI specificato, se non esiste (altrimenti la sovrascrive). **PUT è idempotente**, il che significa che inviare la stessa richiesta **PUT** più volte non ha ulteriori effetti dopo la prima applicazione (infatti si andrà a sovrascrivere un dato con lo stesso dato).

Idempotenza: è una proprietà delle operazioni per cui inviare la stessa richiesta più volte produce lo stesso effetto di una singola richiesta.

Le best practices in ambito RESTful API impongono comunque l'utilizzo di **PUT** per le operazioni di aggiornamento, e di **POST** per quelle di creazione.

Esercizio Guidato - Ristorante e Menù (10): Definitions

API

Finora abbiamo parlato più volte della sezione **definitions**, ma non abbiamo ancora implementato nulla al suo interno. Ricordiamo che le **definitions**, nella specifica OpenAPI (Swagger), servono a descrivere strutture di dati comuni che possono essere riutilizzate in più parti della documentazione dell'API.

Partiamo dal **type** delle due entità (**Menu** e **Piatto**). Entrambe sono di tipo **object**, ovvero si definiscono in quanto strutture complesse, aventi diverse proprietà. Come vedete, tutte le proprietà in questo caso sono considerate **obbligatorie**, e questo è specificato in **required**.

Le varie proprietà, seguendo la logica di cui abbiamo parlato finora, sono quindi definite. Fate attenzione **all'indentazione, utile per una rappresentazione gerarchica del contenuto**.

Infine, potete notare l'aggiunta di **example**. Questi migliorano la documentazione, facilitano i test, e chiariscono le specifiche.

```
58  definitions:
59    Menu:
60      type: object
61      required:
62        - id
63        - descrizione
64      properties:
65        id:
66          type: integer
67          minimum: 0
68        descrizione:
69          type: string
70          example: Menu di terra
71    Piatto:
72      type: object
73      required:
74        - id
75        - descrizione
76        - prezzo
77      properties:
78        id:
79          type: integer
80          minimum: 0
81        descrizione:
82          type: string
83          example: Spaghetti alla carbonara
84        prezzo:
85          type: integer
86          minimum: 1
```

Esercizio Individuale (1) - Correzione di un'API

API

In questo codice OpenAPI ci sono alcuni problemi cruciali che devono essere corretti. Quindi, ispezionando il codice riga per riga, dovete cercare di capire cosa potrebbe essere sbagliato.

Di seguito **le specifiche originali dell'API**: Progettare un'**API REST** (parziale) per gestire una piattaforma di giochi e descriverla in Swagger/ OpenAPI.

Ogni **gioco** è caratterizzato da un **ID** (di tipo intero per semplicità), un **nome** (string) e una **categoria**, che può assumere valori *Shooter*, *Adventure*, *Puzzle*, o *Sport*, e un **numero di giocatori minimo e massimo**, entrambi valori interi.

Scrivere un file in formato JSON o YAML.

L'API permette di:

- ottenere l'elenco di tutti i giochi di una data categoria
- modificare il numero minimo e massimo di giocatori in un gioco
- aggiungere un insieme di giochi e le relative informazioni; è possibile quindi aggiungere anche più di un gioco con un'unica richiesta.

Specificare: URL di accesso, metodi HTTP, parametri e risposte con esempi. L'API restituisce un errore, con codice 400, se i parametri in input non sono corretti.

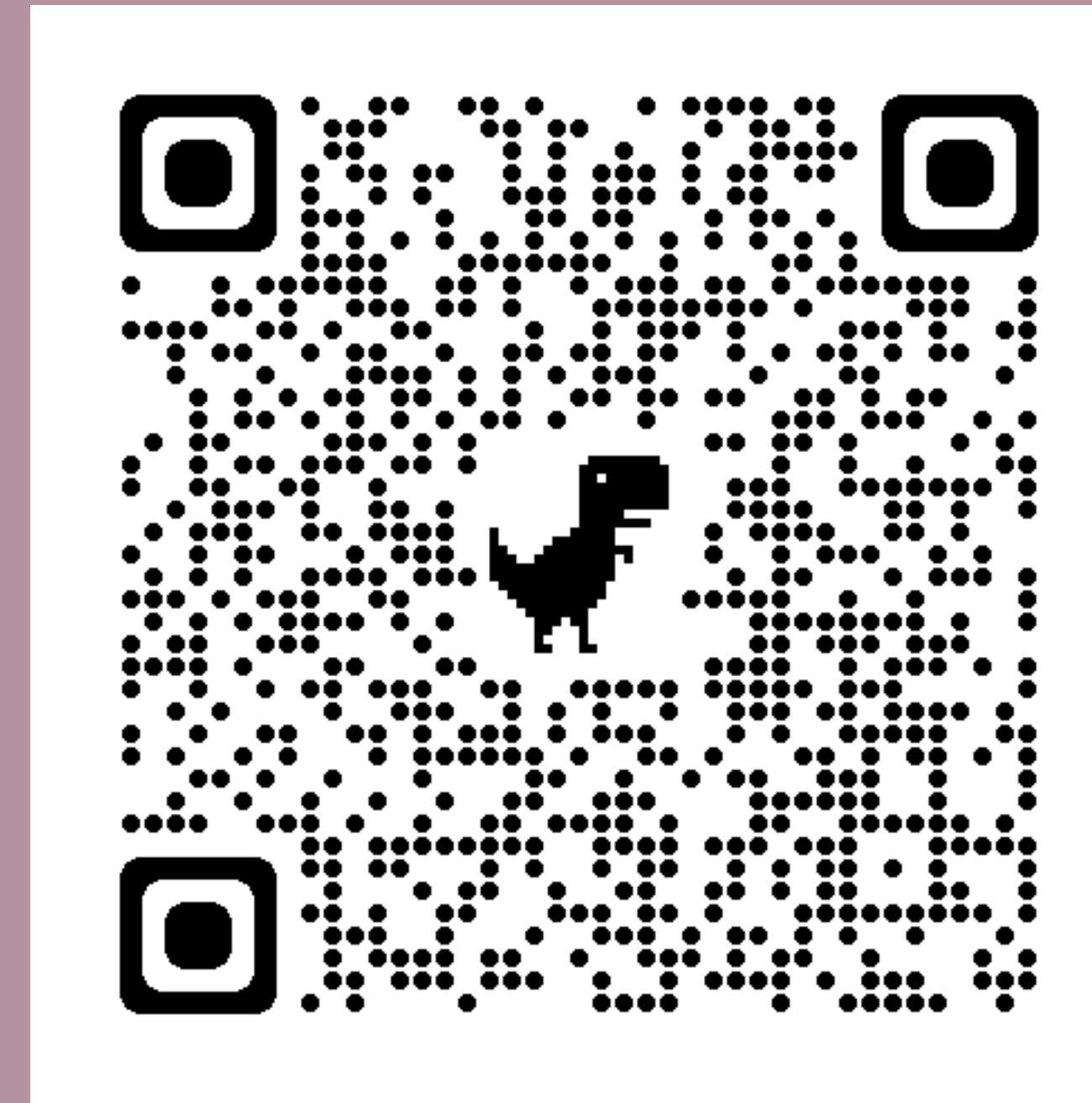
Note:

- Non è richiesto includere le sezioni servers, tags
- Non è richiesto gestire autenticazione

Esercizio Individuale (1) - Correzione di un'API (1)

API

Qui sotto il QR Code per scaricare il codice YAML errato



Ora aprete il file, copiate e incollate il codice su Swagger, e cercate di risolvere l'esercizio, poi lo correggeremo insieme.

Fate particolare attenzione alla sintassi, alle ripetizioni, e ai pezzi mancanti.

Esercizio Individuale (2) - API Incompleta

API

A uno studente sono state fornite le istruzioni seguenti per progettare un'API. Sebbene questo studente abbia svolto un buon lavoro con alcune parti, ne ha perse altre, quindi **questa documentazione API è incompleta**. Si prega di individuare le parti mancanti e aggiungerle.

Progettare un'API REST (parziale) per la gestione di un **blog di botanica**.

Il blog contiene **articoli**, ognuno individuato da un **ID**, una **data di pubblicazione**, un **titolo** e un **contenuto testuale**; un articolo inoltre può appartenere a **una o più categorie** tra ‘orto’, ‘fiori’, ‘attrezzi’, ‘prodotti’, ‘consigli’ e ‘news’.

Scrivere un file in formato JSON o YAML. L'API permette di:

- **ottenere l'elenco di tutti gli articoli di una data categoria e pubblicati dopo una certa data**
- **aggiungere un nuovo articolo**
- **aggiornare il contenuto testuale di un articolo esistente**

Specificare: **URL di accesso, metodi HTTP, parametri e risposte con esempi**.

L'API restituisce un errore, con codice 400, se i parametri in input non sono corretti.

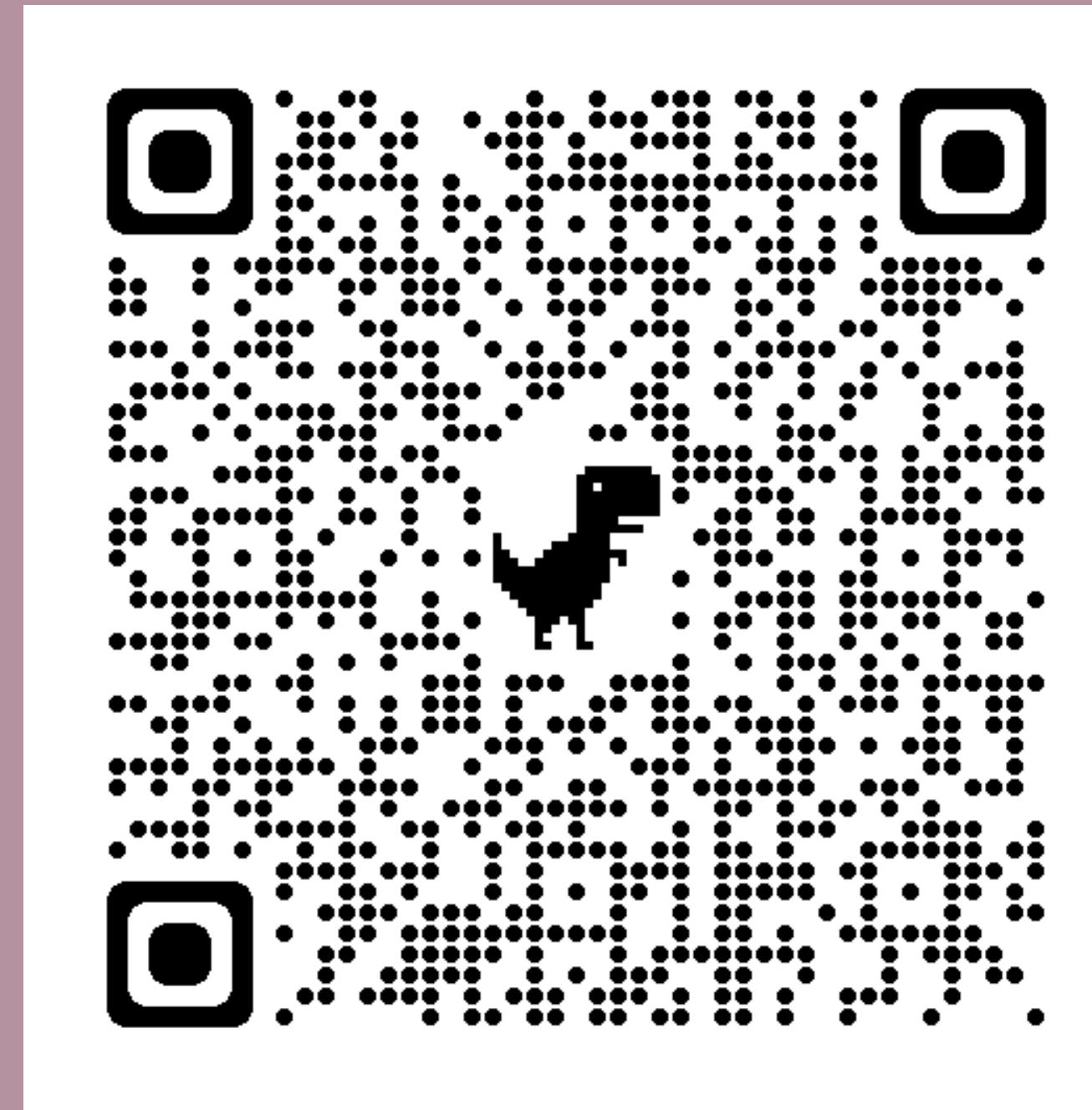
Note:

- Non è richiesto includere le sezioni servers, tags
- Non è richiesto gestire autenticazione

Esercizio Individuale (2) - API Incompleta (1)

API

Qui sotto il QR Code per scaricare il codice YAML incompleto.



Ora aprite il file, copiate e incollate il codice su Swagger, e cercate di risolvere l'esercizio, poi lo correggeremo insieme.

Fate particolare attenzione alla sintassi, alle ripetizioni, e ai pezzi mancanti.

Esercizi Individuali - Nuovi Elementi

API

Come avete visto, abbiamo introdotto dei nuovi elementi:

- **host**: Specifica **l'host** su cui l'API è disponibile (e.g., *botanica.swagger.io*).
- **schemes**: Definisce i protocolli supportati dall'API, in questo caso **http** e **https**.
- **in: query**: indica che il parametro viene inviato come parte della **query string dell'URL della richiesta**. Le query string *iniziano dopo il simbolo ?* nell'URL e sono **formate da coppie chiave-valore** separate da **&**. **Esempio**: Considerando l'URL **https://example.com/api/items?categoria=orto**, il parametro **categoria** è un **parametro di query che indica al server di filtrare gli articoli per quelli appartenenti alla categoria "orto"**.
- **enum**: è utilizzato per **definire un insieme di valori stringa predefiniti** per un parametro o una proprietà. Serve a **limitare i valori che possono essere assegnati a quel parametro o proprietà a quelli specificati nell'elenco**. **Esempio**: Se un parametro **categoria** ha un **enum** che include valori come **["orto" , "fiori" , "attrezzi"]**, significa che il parametro **categoria** può accettare solo uno di questi tre valori come input valido.

Fine

La lezione di oggi è conclusa.

Su GitHub (poi anche su virtuale) troverete un altro esercizio da svolgere da soli che richiede lo sviluppo di un'API (tipo quello del ristorante). In settimana caricherò la soluzione, provate intanto a risolverlo da soli e, se avete dubbi, contattatemi senza problemi.

Qui sotto il link all'ultimo esercizio.

