OpenAPI - Appunti

Recap: CRUD

- **CRUD** è un pattern per applicazioni che trattano dati, con quattro operazioni fondamentali:
  - **Create:** Inserimento di un nuovo record (es. nuovo cliente).
  - **Read:** Accesso in lettura al database.
    - Individuale: Dati di un singolo record (es. scheda cliente).
    - Contenitore: Lista di record con una proprietà specifica (es. clienti di Bologna).
  - **Update:** Modifica di un record esistente (es. numero di telefono di un cliente).
  - **Delete:** Rimozione di un record (es. eliminazione cliente).

Recap: REST

- **Architettura REST:** 4 punti chiave:

  1. Ogni concetto rilevante è una **risorsa**.
  2. Ogni risorsa ha un **URI** identificativo.
  3. **Verbi HTTP** per le operazioni CRUD:
     - `PUT` : Creazione (il client *decide* l'identificatore).
     - `GET` : Visualizzazione.
     - `POST` : Modifica o creazione (il client *non* decide l'identificatore).
     - `DELETE` : Cancellazione.
  4. **Rappresentazione** parametrica dello stato della risorsa, personalizzabile con `Content-Type` .

- **Individui e Collezioni:**

  - Concetti fondamentali: singolo elemento (es. un cliente) vs. insieme di elementi (es. tutti i clienti).
  - Entrambi hanno URI.
  - Operazioni CRUD su entrambi.
  - Il corpo di richieste/risposte è una *rappresentazione* della risorsa, non la risorsa stessa.

- **Verbi HTTP in REST (Esempi):**

Verbo	URI	Descrizione
`GET`	`/customers/`	Elenca tutti i clienti.
`GET`	`/customers/abc123`	Dati del cliente con id=abc123.
`POST`	`/customers/`	Crea un nuovo cliente (il server decide l'ID).
`PUT`	`/customers/abc123`	Crea un nuovo cliente con ID=abc123 (il client decide l'ID).
`PUT`	`/customers/abc123`	Modifica *tutti* i dati del cliente id=abc123.

`POST`	`/customers/abc123`	Modifica *alcuni* dati del cliente id=abc123.
`DELETE`	`/customers/abc123`	Cancella il cliente id=abc123.
`POST`	`/customers/abc123/telephones/`	Modifica dati di una sottorisorsa.

Descrivere API con OpenAPI

- **API RESTful:** Utilizza i principi REST.
- **Documentazione di una API REST:**
  - **End-point (URI/route):** Collezioni ed elementi singoli.
  - **Metodi HTTP:** Cosa succede con `GET`, `PUT`, `POST`, `DELETE`.
  - **Rappresentazioni:** Input e Output (spesso con esempi, senza schema formale).
  - **Errori:** Condizioni e messaggi di errore.
- **Swagger e OpenAPI:**
  - Swagger: Ecosistema di strumenti per API REST (creazione, documentazione, ecc.).
  - OpenAPI: Linguaggio (precedentemente Swagger) per la documentazione di API REST, ora di pubblico dominio.
  - Serializzazione: JSON o YAML.
  - Standard industriale per API REST.
  - Generazione automatica di documentazione, modelli e codice.
- **YAML (Ain't a Markup Language):**
  - Linearizzazione di strutture dati.
  - Sintassi simile a Python (indentazione).
  - Simile a JSON (superset).
  - Tipi: scalari (stringhe, interi, float), liste (array), array associativi (coppie chiave-valore).

OpenAPI (2.0) in YAML - Struttura

- **Informazioni Generali:**
  - `host`, `schemi supportati` (`http`, `https`), `version`, `title`, `description`.
- **Sezione `paths`:**
  - Parte centrale: descrive i percorsi (URL) delle operazioni.
  - Struttura: `<host>/<basePath>/<path>`.
  - Per ogni `path` (endpoint):
    - Sottosezioni per ogni operazione (metodo HTTP).
    - Per ogni operazione:
      - Informazioni generali.
      - Parametri di input (`parameters`) e output (`responses`).

```
<path>
    └── <Operazione (metodo HTTP)>
```

```
        ├── Formati in Output
        └── Parametri in Input
```

- **Parametri in Input (`parameters`):**
  - `in` : Dove si trova il parametro (`path` , `query` , `body` , `header` , `formData`).
  - `name` : Nome del parametro.
  - `description` : Descrizione.
  - `required` : `true` se obbligatorio, `false` altrimenti.
  - `schema` : Formato del valore (tipo scalare, oggetto, array).
    - Tipi scalari: `string` , `number` , `integer` , `boolean` , `array` , `file` .
  - Esempi
    - Parametro in path: `/pet/{petId}` .
    - Parametro in query: `/pet/?status=ready` .
    - Parametro nel body: Oggetto JSON.
- **Oggetti e Definizioni (`definitions`):**
  - Definisce i tipi di oggetti complessi, con proprietà e valori possibili.
  - Referenziabili con `$ref` in `schema` (sia in input che in output).
- **Output (`responses`):**
  - Codici di risposta HTTP (`200` , `400` , `404` , ecc.).
  - `description` : Descrizione della risposta.
  - `schema` : Tipo di output nel body (se presente). Può essere un oggetto o un array di oggetti.
  - Esempio:

```yaml
responses:
  '200':
    description: Successo
    schema:
      type: array
      items:
        $ref: '#/definitions/Pet'  # Riferimento a un oggetto definito
  '404':
    description: Non trovato
```

- **Swagger Editor:** https://editor.swagger.io/

Esercizi (Struttura Concettuale)

Gli esercizi proposti richiedono di progettare API REST parziali e descriverle in OpenAPI, specificando:

- **Risorse:** Quali concetti sono modellati (es. menù, piatti, articoli, giochi).
- **URI:** Come sono strutturati gli endpoint (es. `/menus` , `/menus/{id}` , `/articles`).
- **Metodi HTTP:** Quali verbi sono usati per quali operazioni (es. `GET /menus` , `POST /articles`).
- **Parametri:** Input necessari (es. `categoryId` , `date` , `minPlayers`).
- **Risposte:** Struttura dell'output (es. array di oggetti, oggetto singolo, messaggi di errore).

- **Definizioni di oggetti:** Se necessario, definire la struttura degli oggetti complessi.

I concetti chiave sono:

1. **Ristorante:** Gestione di menù e piatti.
2. **Blog di Botanica:** Gestione di articoli e categorie.
3. **Piattaforma di Giochi:** Gestione di giochi, categorie e numero di giocatori.

Conclusioni

- REST: Applicazione vista come un ambiente con stato modificabile tramite comandi (metodi HTTP) su risorse (URI), con rappresentazioni esplicite (`Content-Type`).
- Vantaggi di REST: Sfrutta appieno le caratteristiche del web (caching, proxying, sicurezza).
- Semantic Web: Possibilità di integrazione con tecniche del Semantic Web per funzionalità avanzate.