



UNICA

UNIVERSITÀ
DEGLI STUDI
DI CAGLIARI



sAifer Lab

Joint lab on Safety and Security of AI

Il Web dinamico

Maura Pintor

maura.pintor@unica.it

In questa lezione

Come progettare delle API

Standard per la documentazione

- OpenAPI

Esercizi di progettazione

Modularità dei servizi

Architetture monolitiche vs. architetture a microservizi



API e servizi

Inizialmente il web era solo un archivio di documenti *statici*

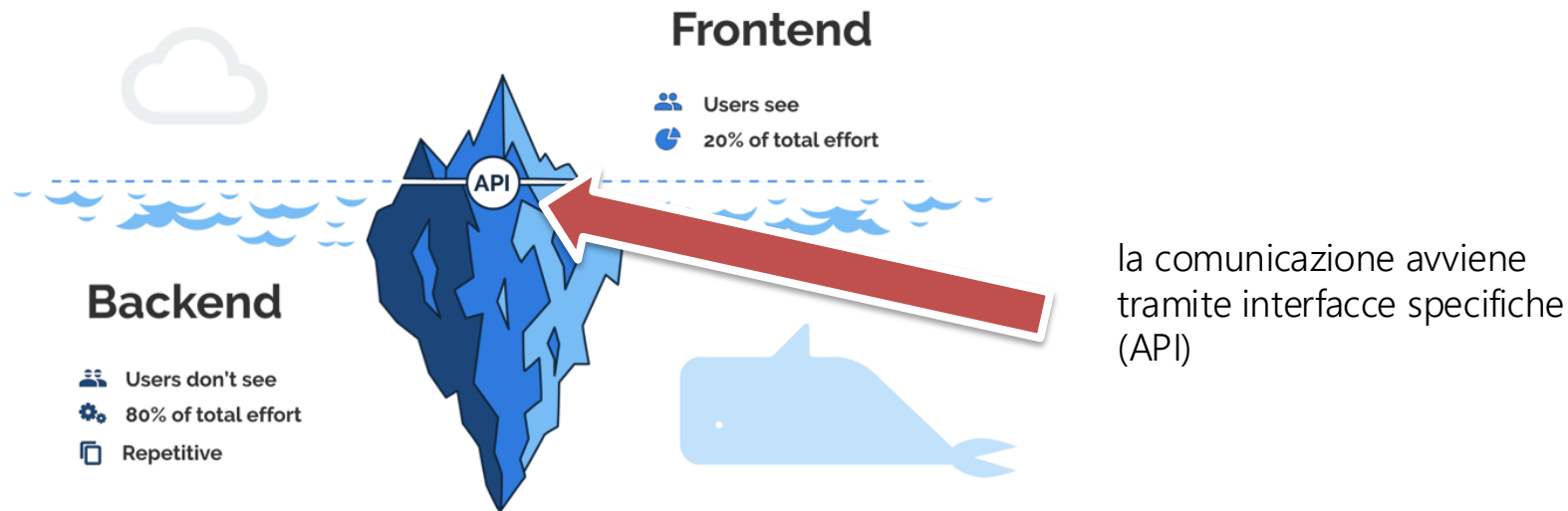
Si è poi evoluto e ha iniziato a offrire la possibilità di diversi servizi, caratterizzati da:

- Alta modularità - consente la composizione di servizi e la loro comunicazione tramite API
- Trasparenza - le operazioni di basso livello come il recupero delle informazioni e la logica di funzionamento non sono mai viste dall'utente

I web server visti dagli sviluppatori

Il server web è una macchina che risponde alle richieste inviate dagli utenti

Si divide in frontend (layer di presentazione), e backend (layer di accesso ai dati)



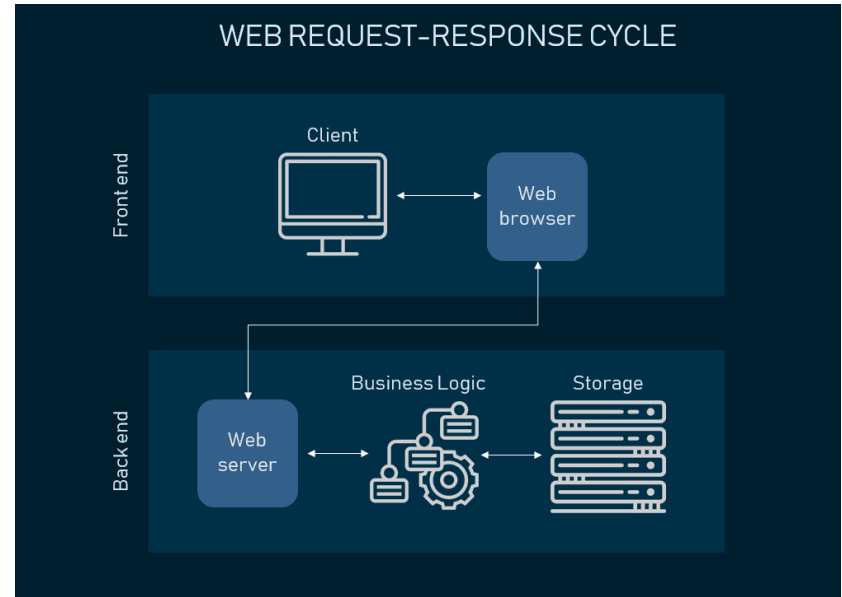
Da dove iniziare a creare un web server?

Ci sono diversi approcci

- **top-down**: partire dall'interfaccia utente e andare verso le operazioni di basso livello sui dati
- **bottom-up**: partire dai dati e arrivare all'interfaccia utente
- **middle-out**: partire dalla logica di servizio e lavorare in entrambe le direzioni

Da dove partire molte volte dipende da cosa si ha di pronto

Di solito per chi inizia è meglio partire dalla logica, ma SEMPRE iniziare anche a pianificare il resto



Prima di scrivere il codice...

La progettazione è una parte importantissima del processo di sviluppo

Se si inizia subito a scrivere il codice, si rischia di perdere tempo a riscrivere pezzi perché non erano progettati bene

Il tempo **investito** in progettazione non è mai perso! Soprattutto in casi in cui ci sono delle scadenze temporali strette, è sempre importante ottimizzare il tempo.

Questo si può fare solo tramite una solida progettazione. Immaginate di costruire una casa senza un progetto...



Come descrivere i servizi e le interazioni

Le API definiscono cosa ogni servizio è in grado di fare e come gli altri possono interagire con il servizio

Quando assumiamo il ruolo di progettisti di API, dobbiamo anticipare quali saranno le potenziali richieste dei clienti e utilizzatori e farle diventare "comode da usare"

Ecco perché si parla di ecosistema di API, non vanno viste in singolo ma vanno pensate in modo olistico, non in modo isolato



Tipi di API

Ogni API definisce:

- un **protocollo**, ovvero la struttura di controllo per la comunicazione
- un **formato**, ovvero la struttura del contenuto

Ci sono poi delle evoluzioni delle iniziali API, che erano pensate solo per comunicazioni dirette

- *Remote Procedure Calls (RPC)*, delle API che chiamano funzioni in altri processi
- *Messaging*, che mandano piccole porzioni di dati in pipeline, per esempio a seguito di eventi (es., un processo che ha completato l'esecuzione)
- *Publish-subscribe* (o *pub-sub*), un protocollo in cui uno dei due lati si iscrive (subscriber) agli aggiornamenti inviati dall'altro (publisher) su un dato argomento
- *Code (queues)*, utilizzate solitamente per gestire più richieste che richiedono tempo per essere eseguite

Tutte queste possono essere utilizzate all'interno di web server per gestire processi più complessi rispetto alla sola visualizzazione di documenti

Come descrivere le API

Possiamo pensare alle API come delle operazioni sul servizio

Ogni operazione ha un suo input e genera un output

- La documentazione è una parte importantissima delle API
- Bisogna sempre tenere traccia di cosa queste si aspettano in input e output
- Se ci sono modifiche nei servizi, sarà poi facile capire come modificare l'ecosistema perché risulta più modulare
- La documentazione permette anche di capire cosa cambia con gli update, e di generare dei test per verificare che funzioni tutto come ci si aspetta

Standard per descrivere le API

OpenAPI è una *specifica* che specifica come descrivere le API HTTP-Based, ovvero quelle che si basano sul protocollo HTTP

Tipicamente queste API sono chiamate RESTful (vedremo poi cosa significa)

Il formato OpenAPI descrive in un **file di testo** gli input e output delle API, insieme anche a informazioni sull'API, per esempio che autorizzazione è richiesta o chi fornisce il servizio

Le definizioni possono essere scritte "a mano" o tramite strumenti (ne vedremo alcuni nel corso)

Le descrizioni di OpenAPI sono di solito in formato YAML o JSON

- YAML = Yet Another Markup Language, poi cambiato in YAML Ain't Markup Language

OpenAPI fu donato da SmartBear alla Linux Foundation nel 2015

- <https://github.com/OAI/OpenAPI-Specification>



Esempio di definizione OpenAPI

Sembra una descrizione lunga, ma contiene molte informazioni utili sul servizio

Soprattutto, questo formato è facilmente leggibile da persone (*human-readable*) e macchine (*machine-readable*)

Per leggerlo in modo più rapido, possiamo usare un servizio chiamato Swagger

<https://editor.swagger.io>

```
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description
  version: 0.1.9

servers:
  - url: http://api.example.com/v1
    description: Optional server description
  - url: http://staging-api.example.com
    description: Optional server description

paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        "200": # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Perché YAML e non JSON?

Osservando la descrizione, ci si potrebbe chiedere se sia possibile descrivere la stessa API usando il JSON

La risposta è ovviamente sì, ma lo YAML risulta più compatto e semplificato

- le stringhe non richiedono virgolette
- supporta commenti
- usa l'indentazione invece che le parentesi
- YAML è un superset di JSON, quindi supporta anche il JSON al suo interno
- Consente l'utilizzo di *funzioni avanzate*, per esempio i tipi di dato definiti dall'utente

<https://yaml.org>

```
%YAML 1.2
---
YAML: YAML Ain't Markup Language™

What It Is:
YAML is a human-friendly data serialization
language for all programming languages.

YAML Resources:
YAML Specifications:
- YAML 1.2:
  - Revision 1.2.2      # Oct 1, 2021 *New*
  - Revision 1.2.1      # Oct 1, 2009
  - Revision 1.2.0      # Jul 21, 2009
- YAML 1.1
- YAML 1.0

YAML Matrix Chat: '#chat:yaml.io'      # Our New Group Chat Room!
YAML IRC Channel: libera.chat#yaml    # The old chat
YAML News: twitter.com/yamlnews
YAML Mailing List: yaml-core          # Obsolete, but historical

YAML on GitHub:                                # github.com/yaml/
  YAML Specs: yaml-spec/
  YAML 1.2 Grammar: yaml-grammar/
  YAML Test Suite: yaml-test-suite/
  YAML Issues: issues/

YAML Reference Parsers:
- Generated Reference Parsers
- YPaste Interactive Parser

YAML Test Matrix: matrix.yaml.io
```

A cosa servono le definizioni di API?

Una volta create le nostre API, possiamo creare uno schema della nostra applicazione
Possiamo creare un flusso di lavoro (workflow) che connette i vari servizi

- Per esempio, possiamo creare dei test automatici per le nostre API
- Possiamo avere un feedback (per esempio nel nostro team di lavoro)
- Possiamo verificare la consistenza con quello che ci serve
- Possiamo tenere traccia di quello che cambia tra le varie versioni (e comunicarlo, di nuovo, al team di lavoro)

Representational State Transfer (REST)

REST è una collezione di idee su come progettare un sistema interconnesso (in particolare server/client)

Si considera un'architettura software per sistemi distribuiti

Principi base:

- Client-server separation
- Stateless: ogni richiesta è indipendente
- Cacheable: risposte etichettabili come "riutilizzabili"
- Interfaccia uniforme

Operazioni CRUD attraverso metodi HTTP per interagire con le **risorse**:

- GET (Read)
- POST (Create)
- PUT/PATCH (Update)
- DELETE (Delete)

Implementazione pratica dei principi REST: le RESTful API

"Where REST starts and HTTP ends is a tricky question to answer, but the rule of thumb is that HTTP is the protocol and REST is a way of designing APIs."

Excerpt From
Designing APIs with Swagger and OpenAPI
Joshua S. Poneiat, Lukas L. Rosenstock

Best Practices:

- Nomi risorse al plurale (/users)
- Struttura gerarchica (/users/123/posts)
- Versionamento API (/v1/users)
- Documentazione chiara

Che utilità ha OpenAPI per gli utenti?

Dalla prospettiva degli utenti sviluppatori, avere della documentazione chiara sulle API è molto utile

Per comunicare con i servizi di cui le API sono le interfacce, gli sviluppatori dovranno solo scrivere dei comandi per dare gli input alle API e ricevere gli output

- Anche questo processo può essere automatizzato, per esempio creando degli SDK (Software Development Kits) direttamente da servizi come Swagger (o anche tramite servizi di AI Generativa come ChatGPT!)

Che utilità ha OpenAPI per chi fornisce il servizio?

Per i produttori di API, scrivere il codice che fornisce il servizio diventa molto più facile se si ha già in mente il progetto generale

- velocità di scrittura del codice
- consistenza di tutto il servizio a livello globale (per esempio, riutilizzo di nomi per le risorse)
- questo è ancora più importante quando le API sono tante










Esempio di documentazione di API

Proviamo, per esempio a navigare queste API

Possiamo usare:

- direttamente il browser
- script, per esempio in Python
- il terminale, tramite l'applicazione CURL
<https://curl.se>
- strumenti di terze parti (es:
<https://www.postman.com>)

 GET	/users/{user_id}	200
Retrieve a user's detail. You get a unique user object everytime. ID remains same as sent in URL.		
 GET	/users	200
Generate fake users (names, address, profile photo, phone, etc). Each time you get new user objects.		
 GET	/companies/{company_id}	200
Retrieve details about a company by passing company ID. (company-name, market capital, domain, etc.)		
 GET	/companies	200
Each time you call this API, you get unique ten company objects.		
 GET	/customers	200
Get a list of customers with username, password, full address, created date for an e-comm example.		
 GET	/notifications	200
Get a list of notifications to show to a user. You can see find title, message, read status, type.		
 GET	/	200
Default response. Responds with the meta information about this endpoint.		

Documentazione: <https://app.beeceptor.com/mock-server/fake-json-api>
Server: <https://fake-json-api.mock.beeceptor.com>



Esercizio in Python

Proviamo a usare una di queste API scrivendo del codice Python

Useremo la libreria requests, che fornisce strumenti di alto livello per gestire le richieste (e risposte)

Cosa succede se sbagliamo l'URL?

```
import requests

def get_users():
    url = "url_here"
    response = requests.get(url)
    print("Status code:",
          response.status_code)
    return response.json()

users = get_users()
for user in users:
    print(user)
```

Esercizio in Python

Gestione delle eccezioni

- La gestione deve essere fatta lato sviluppatori
- Gli utenti non vogliono vedere "errori rossi" nel loro output
- Nota: Non è una buona pratica "stampare gli errori"... Per ora serve solo a noi per fare la prova
 - Potrebbe dare delle informazioni a potenziali hacker

```
import requests

def get_users():
    url = "url_here"
    try:
        response = requests.get(url)
        response.raise_for_status()
        print("Status code:", response.status_code)
        return response.json()
    except Exception as e:
        print("Error:", e)
        return None

users = get_users()
if users:
    for user in users:
        print(user)
```

E le richieste POST?

Proviamo a capire come funzionano le API di <https://funtranslations.com/api/pirate>

POST /pirate

Translate from English to pirate.

Parameters

Parameter Name	Parameter Type	Description
text	string	Text to translate.

output

The result is a json object with the converted text.



... E scriviamo il codice

```
url = "https://api.funtranslations.com/translate/pirate.json"

def request_pirate_translation(text):
    try:
        response = requests.post(url, json={"text": text})
        response.raise_for_status()
        print("Status code:", response.status_code)
        return response.json()
    except Exception as e:
        print("Error:", e)
        return None

text = "Hello and welcome to the Web Development course!"
translation = request_pirate_translation(text)
if translation:
    print(translation)
```

Ora di scrivere la nostra prima API

Vogliamo definire un servizio di biblioteca in cui possiamo listare dei libri e mettere delle recensioni

Iniziamo a pensare a cosa potrà fare l'utente

- Ottenere la lista di libri
- Ottenere il dettaglio di un singolo libro
- Inserire una recensione per il libro scelto

Quindi ci serviranno tre API (almeno per l'inizio)

La seconda è gerarchicamente legata alla prima

- Possiamo pensare a una struttura **gerarchica** delle risorse

Inoltre, mentre le prime due sono API di richiesta informazioni, la terza è un inserimento

- Le prime due saranno **GET**, la terza sarà una **POST**



Ripasso: Metodi HTTP più usati

GET

- serve per chiedere risorse
- passaggio di parametri tramite URL <query>
- la lunghezza dell'URL è limitata

POST

- i dettagli della risorsa sono contenuti nel body del messaggio
- non ha limiti di lunghezza per i parametri
- il message body può essere *cifrato*

Ripasso: Status code HTTP

Lo status code è composto da 3 cifre, la prima indica la classe di risposta e le altre due l'errore specifico

- **1xx: informational.** Risposta temporanea alla richiesta, comunque usata poco dopo HTTP/1.0.
- **2xx: success.** La richiesta è stata processata correttamente.
 - esempio: 200 OK
- **3xx: redirect.** Il server ha ricevuto la richiesta, ma il client deve eseguire altre azioni per portarla a termine.
 - esempio: 301 Moved permanently (ma il server conosce la nuova posizione)
- **4xx: client error.** La richiesta non è formattata correttamente o non è autorizzata
 - esempio: 404 Not found (url errato)
- **5xx: server error.** La richiesta può essere corretta, ma il server ha un problema interno per cui non è in grado di rispondere.
 - esempio: 500 Internal server error

Definiamo prima il formato di un libro

Per ogni libro, in questo caso d'uso specifico, ci serviranno tre attributi

- Titolo
- Autore
- Recensione

Che tipo di dato usare?

- Per titolo e autore possiamo pensare a delle stringhe
- Per la recensione possiamo scegliere una recensione testuale, ma per ora utilizziamo un numero da 1 a 5



Formato del libro

Iniziamo a definire che attributi dovrà avere ogni libro

Questo ci consente di avere un oggetto riutilizzabile e non riscrivere ogni volta tutta la struttura del libro

- si può vedere come un tipo di dato astratto
- si possono mettere dei constraint su che forma avranno i dati (es. recensione numero intero da 1 a 5)

```
components:
  schemas:
    Libro:
      type: object
      properties:
        id:
          type: integer
          example: 1
        titolo:
          type: string
          example: "Il nome della rosa"
        autore:
          type: string
          example: "Umberto Eco"
        recensione:
          type: integer
          minimum: 1
          maximum: 5
          example: 5
```

Passiamo alla prima API: la lista di libri

Iniziamo con l'API per recuperare la lista di libri

- La richiesta avvenuta con successo risponderà 200 e restituirà un JSON con la lista dei libri
- Ogni elemento della lista è di tipo libro, come definito sopra in "components"
- Definiamo anche un caso di errore in cui il server risponderà 500 (es. la lista di libri è troppo grande per essere gestita)

```
paths:
  /books:
    get:
      summary: Recupera la lista dei libri
      description: Restituisce un elenco di libri.
      responses:
        '200':
          description: Lista recuperata.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Libro'
        '500':
          description: Errore interno del server
```

Come identificare un singolo libro?

Abbiamo due scelte: possiamo usare i parametri di path o i parametri di query

- path: solitamente usati per identificare una singola risorsa (come un libro!)
- query: solitamente usati per filtrare o riordinare i risultati (per esempio di una query su un database)

Nel nostro caso il path parameter sembra fare al caso nostro

Possiamo vederlo come il numero civico di un indirizzo (l'URL è un Locator alla fine)

Feature	Path Variable	Query Parameter
Location	Part of the URL	Attached to the end of the URL
Purpose	Identify a specific resource	Filter or sort results, or provide additional info

Recupero di un singolo libro

Per prendere un singolo libro, visto che abbiamo definito i libri come delle risorse, possiamo specificare un ID di libro come richiesta gerarchica annidata tramite **path parameter**

Definiamo sotto il path /books un altro path con /books/{id}

Anche questa API restituisce un libro, per cui riusiamo quanto definito prima

In questo caso dobbiamo anche prenderci cura di possibili ID inesistenti (errore 404: not found)

```
/books/{id}:  
get:  
  summary: Recupera un libro specifico  
  description: Restituisce i dettagli di un libro specifico dato il suo ID.  
  parameters:  
    - name: id  
      in: path  
      required: true  
      description: ID del libro da recuperare  
      schema:  
        type: integer  
  responses:  
    '200':  
      description: Dettagli del libro recuperati con successo  
      content:  
        application/json:  
          schema:  
            $ref: '#/components/schemas/Libro'  
    '404':  
      description: Libro non trovato  
    '500':  
      description: Errore interno del server
```

Infine, la richiesta POST per la recensione

Nel caso della richiesta POST, dobbiamo definire anche il formato del body della richiesta

Inoltre, dobbiamo gestire anche casi in cui la richiesta è malformata (per esempio se l'utente inserisce un numero maggiore di 5 o una stringa)

```
/books/{id}/recensione:
post:
  summary: Aggiungi una recensione a un libro
  description: Permette di aggiungere una recensione a un libro dato il suo ID.
  parameters:
    - name: id
      in: path
      required: true
      description: ID del libro a cui aggiungere la recensione
      schema:
        type: integer
  requestBody:
    required: true
    content:
      application/json:
        schema:
          type: object
          properties:
            recensione:
              type: integer
              minimum: 1
              maximum: 5
              example: 5
  responses:
    '200':
      description: Recensione aggiunta con successo
    '400':
      description: Richiesta non valida
    '404':
      description: Libro non trovato
    '500':
      description: Errore interno del server
```

Ora che la nostra API è definita...

Possiamo provare a
ispezionare la
documentazione generata da
Swagger

<https://editor.swagger.io>

Ovviamente la logica non è
ancora stata scritta, per cui
non è possibile provare
effettivamente le API

Per scrivere la logica si userà
il framework FastAPI

API Libri 1.0.0 OAS 3.0

API per la gestione dei libri

default ^

GET

/books Recupera la lista dei libri

▼

GET

/books/{id} Recupera un libro specifico

▼

POST

/books/{id}/recensione Aggiungi una recensione a un libro

▼

Schemas ^

Libro ▾ {

id > [...]

titolo > [...]

autore > [...]

recensione > [...]

}

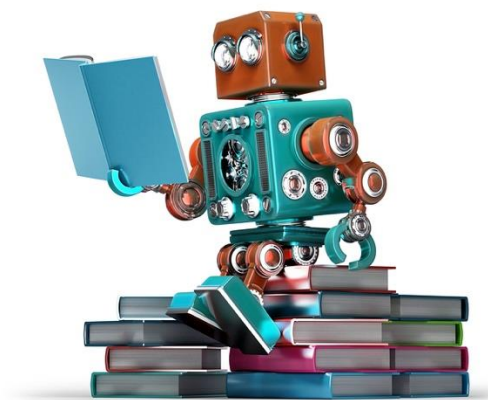
↶

Perché tutto questo lavoro?

OpenAPI è una **specifica formale** per descrivere API basate su HTTP, usando il formato JSON o YAML

Una descrizione formale può essere più facilmente interpretata da un software, mentre una informale no

- Cosa significa in pratica? Per esempio, si può generare del codice già “pronto” con tutte le interfacce specificate (mock server)
- Anche scrivere i test diventa più facile perché si conosce il formato della risposta



Il passaggio inverso: dal codice alla documentazione

Nel corso useremo FastAPI, che consente il passaggio inverso: si inizia a scrivere il codice e la documentazione sarà generata (e aggiornata) automaticamente

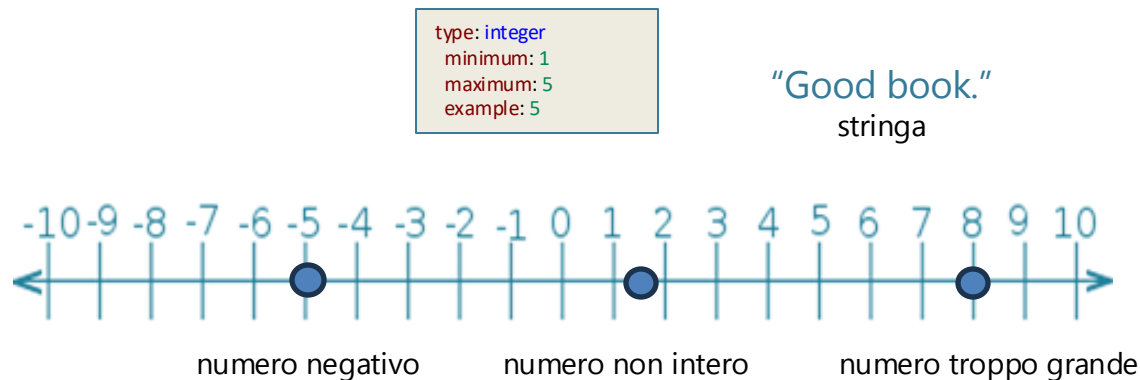
- Forte utilizzo del typing esplicito in Python (tramite la libreria Pydantic <https://docs.pydantic.dev/latest/>)
- Gestione facilitata data dall'autocompletamento dell'IDE

Questo non significa che le API non vadano comunque progettate!

Iniziamo a pensare a dei possibili test...

Una volta implementate le nostre API, dovremo fare dei test per verificare che si comportino come pianificato in fase di progettazione

- risposta corretta in caso di successo
- risposta corretta in caso di errori
 - per esempio, cosa potrebbe generare errori nella specifica appena definita?



Componenti di un servizio web

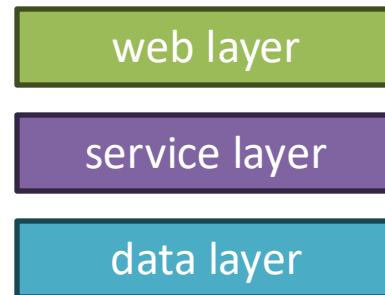
Divideremo concettualmente gli elementi che compongono un sito isolando tre strati (layers)

- Livello web
 - Livello della logica di servizio
 - Livello dei dati
- } frontend
} spesso raggruppati col termine **backend**

E aggiungeremo i componenti aggiuntivi

- Modelli e definizioni dei dati (sarà oggetto di lezioni successive)
- Test (sarà oggetto di lezioni successive)

Come prima cosa, andiamo a vedere cosa si trova nei diversi livelli

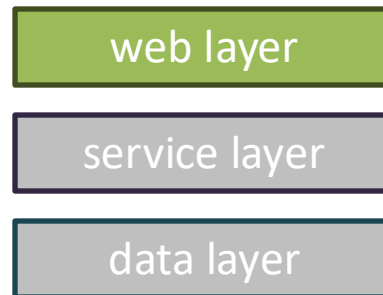


Livello Web

Il livello concettualmente più alto, quello del Web, fa da interfaccia tra l'utente e i livelli di servizio e di dati

Definisce cosa l'utente può fare, di solito in termini di operazioni CRUD (Create, Read, Update, Delete), e spesso prevede un'**interfaccia utente interattiva**

L'utente comunica con il web layer tramite applicazioni dedicate (client) o API (nei casi di utenti più esperti)

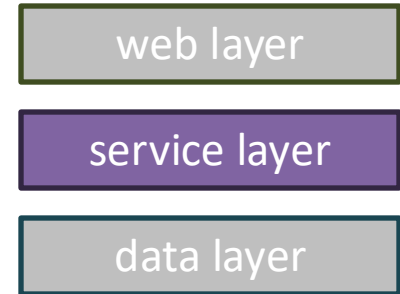


Livello della logica di servizio

Il livello di servizio contiene quella che viene chiamata anche *business logic*, ovvero tutti i dettagli di ciò che il servizio web fornisce agli utenti

Il livello di servizio controlla anche come gli utenti possono accedere ai dati, contenuti e gestiti nel layer sottostante

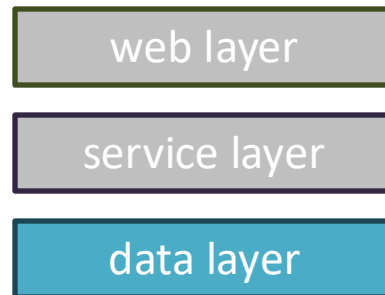
Per esempio, vedremo nel corso l'autenticazione (chi accede) e l'autorizzazione (con quali livello di accesso), che sono implementate nel service layer



Livello dei dati

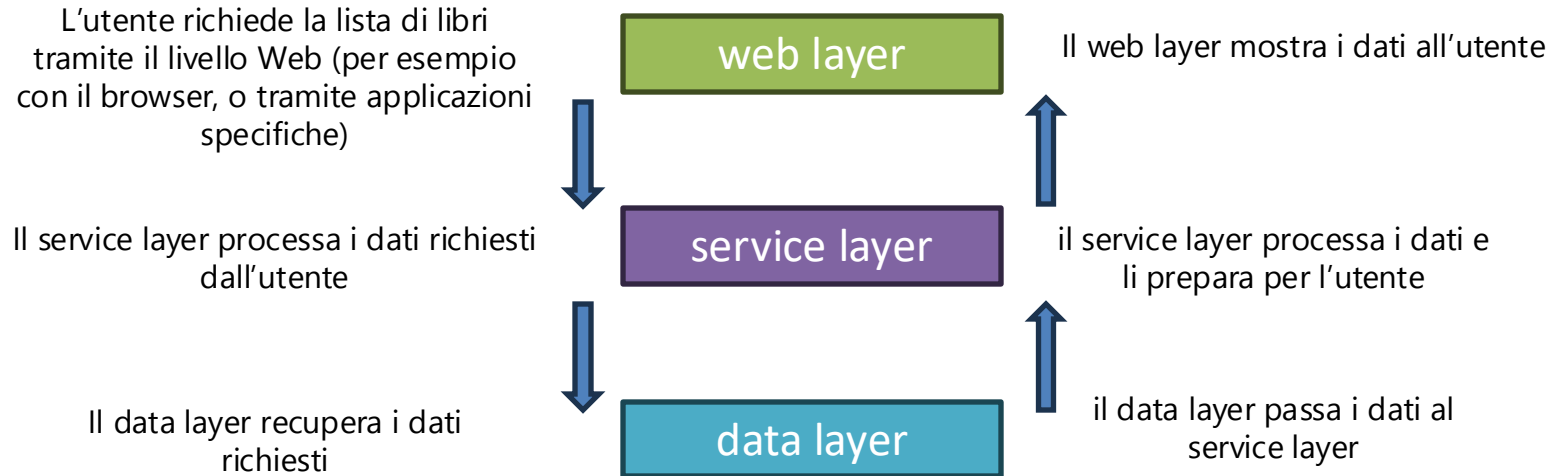
Il livello dei dati fornisce al livello di servizio accesso ai dati, attraverso file o funzioni di accesso e modifica di altri servizi che forniscono accesso ai dati (per esempio, servizi di database)

Spesso, i servizi web gestiscono operazioni di basso livello su database e in generale su risorse (file, hardware, ...) accessibili all'utente tramite il livello di web, e il cui accesso e gestione passano attraverso il livello di servizio



Esempio di gestione di una richiesta

Per esempio, torniamo alla richiesta della lista di libri vista in precedenza



A cosa serve definire una struttura a strati e avere modularità?

Ci sono diversi vantaggi derivanti dall'isolamento concettuale dei diversi livelli

- ogni layer può essere progettato da diversi specialisti
- si possono eseguire test in isolamento
- si possono integrare nuovi servizi e rimpiazzare servizi esistenti senza compromettere il resto dello stack, purché si rispettino le interfacce

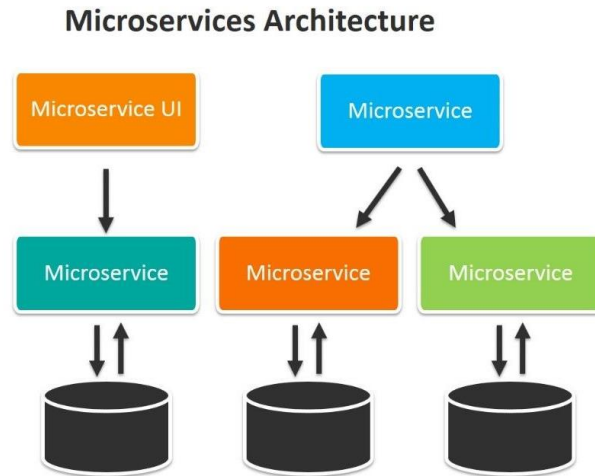
La separazione va imposta all'inizio, altrimenti è molto più difficile districare i diversi aspetti dopo che il codice è stato scritto (molte volte è imposto dal framework stesso)

Bisogna immaginare che le uniche vie di comunicazione tra diversi strati siano regolate da opportune API

Architetture a microservizi

I microservizi sono semplicemente dei servizi indipendenti, autonomi, e compatti, che collaborano

- sono applicazioni che girano in modo indipendente
- comunicano tramite API
- l'idea di essere "compatti" non deve essere confusa con l'essere "piccoli", ma che sono applicazioni con uno scopo preciso e verticale
 - principio di "Single Responsibility", ovvero essere responsabili soltanto di un compito, ma farlo bene
 - per esempio, inviare mail, invio di pagamenti, trasferimento dei file, possono essere implementati come microservizi isolati



Architetture a microservizi

Vantaggi:

- i microservizi possono essere distribuiti anche su più macchine
- isolamento per prevenire guasti all'intero sistema
- spesso gestiti da team dedicati
- possono essere parallelizzati
- più facile fare i test individuali dei singoli microservizi

Svantaggi:

- interazione di tanti servizi che possono essere disomogenei
- l'architettura si complica
- di solito approccio più costoso
- più alto rischio di sicurezza (possono esserci dei servizi meno affidabili o poco protetti)
- più complicato fare i test di integrazione di diversi microservizi

Le architetture monolitiche

Le architetture a microservizi si contrappongono alle architetture **monolitiche**

- nelle monolitiche, tutte le funzionalità devono essere sviluppate in combinazione e fatte funzionare insieme
- in questo caso, si perde la modularità e si può sfruttare meno la distribuzione dei servizi
- il controllo e la gestione centralizzati offrono comunque altri vantaggi (per esempio, consentono migliore ottimizzazione delle risorse)

Architetture monolitiche

Vantaggi:

- più facile coordinare i vari elementi
- più facile accedere a dati tramite diverse applicazioni
- a volte più facile da debuggare, si trova tutto in un solo punto
- più semplice fare i test del sistema intero

Svantaggi:

- con progetti grandi, diventano difficili da gestire
- difficile sostituire elementi quando sono accoppiati e annidati dentro l'applicazione
- più complicato isolare i singoli servizi per fare test

Quindi, quale scegliere?

In generale, la scelta è data dal framework e dai servizi che si vogliono usare

- molti framework (per esempio Django in Python) sono pensati per applicazioni monolitiche
- di solito si scelgono vie di mezzo (alcuni servizi sono gestiti come microservizi, altri sono integrati nell'applicazione)

