



SAPIENZA  
UNIVERSITÀ DI ROMA

"Sapienza" Università di Roma  
Ingegneria dell'Informazione, Informatica e Statistica  
Dipartimento di Informatica

---

## Programmazione WEB

---

*Autore*  
Vincenzo Bova

A.A. 2025/2026

# Indice

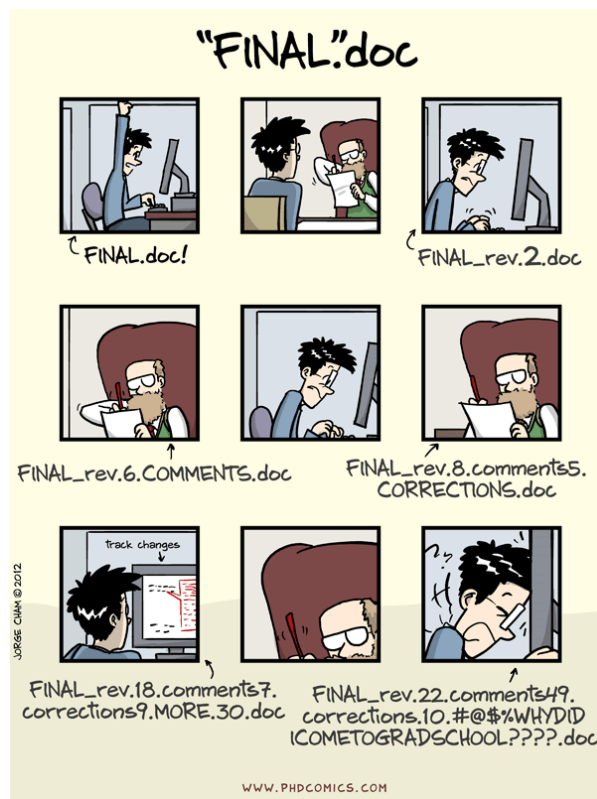
<b>1</b>	<b>Introduzione a Git</b>	<b>2</b>
1.1	Sistemi di versionamento . . . . .	2
1.2	Git . . . . .	3
1.3	Comandi Git . . . . .	6
1.4	Git flow . . . . .	7
1.5	Gestione dei conflitti . . . . .	8
1.6	Annullamento delle operazioni . . . . .	8
<b>2</b>	<b>HTTP</b>	<b>9</b>
2.1	Richieste/Risposte HTTP . . . . .	9
2.2	Intermediari . . . . .	10
2.3	Metodi HTTP . . . . .	10
2.4	Codici di stato della risposta . . . . .	12
<b>3</b>	<b>API</b>	<b>13</b>
3.1	JSON e YAML . . . . .	13
3.2	API . . . . .	14
3.3	REST . . . . .	15

# 1

## Introduzione a Git

### 1.1 Sistemi di versionamento

Durante lo sviluppo di un progetto c'è spesso la necessità di effettuare revisioni, correzioni o modifiche ai file che lo compongono.



Gestire ciò creando ogni volta nuovi file, tuttavia, comporta evidenti problemi:

- **Duplicazione del contenuto:** che rende il sistema inefficiente e aumenta la difficoltà nel mantenere integrità;
- **Assenza di Naming Convention:** che rende impossibile risalire ad uno storico delle modifiche;
- **Autori incerti;**
- ...

Per ovviare a ciò sono stati creati i **sistemi di versionamento** (git, csv, mercurial, svn...), i quali offrono vari benefici:

- **Gestione delle versioni:** il sistema si occupa automaticamente di etichettare le varie versioni in modo consistente;
- **Tracciamento delle modifiche:** è possibile accedere ad uno storico delle modifiche effettuate;
- **Presenza di metadati:** ogni modifica ha un autore, una data...;
- **Creazione di linee di sviluppo parallele:** è possibile creare una versione parallela del codice per non modificare la versione principale, e poi riunirle integrando i cambiamenti;
- **Sincronizzazione tra computer:** il sistema consente di mantenere il progetto allineato tra più computer.

## 1.2 Git

Git è un sistema di versionamento distribuito e veloce, creato nel 2005 e capace di gestire progetti di grandi dimensioni. Si basa su un design semplice e utilizza DAG (*Directed Acyclic Graph*) e Merkle trees come strutture dati.

### Definizione 1.1: Repository

È un insieme di commit, branch e tag.  
Per semplicità assumiamo che un progetto equivale ad un repository.

### Definizione 1.2: Working copy

È l'insieme dei file tracciati nella copia locale del repository.  
Quando creiamo un nuovo file non sarà ancora tracciato e bisognerà quindi aggiungerlo, quando invece modifichiamo un file già tracciato (*update*) stiamo aggiornando la working copy.

### 1.2.1 Commit

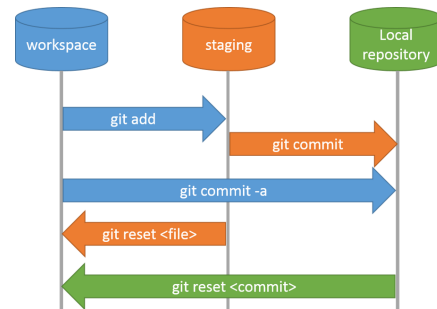
Un commit è un'istantanea del repository in un determinato momento.  
Viene identificato dallo **SHA1** del commit stesso e contiene diversi campi:

- data + autore, data + commiter
- commento **obbligatorio**
- 0,1 o più genitori
- tree: hash di tutti i file nel commit

In particolare il commit può contenere un sottoinsieme delle modifiche (anche ad un singolo file), le quali devono essere aggiunte alla staging area dei cambiamenti.

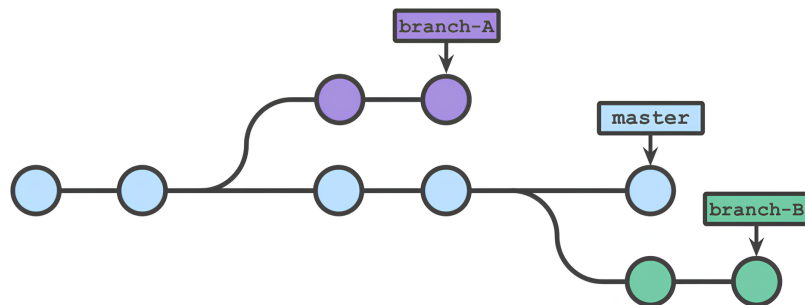
	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.



### 1.2.2 Branch

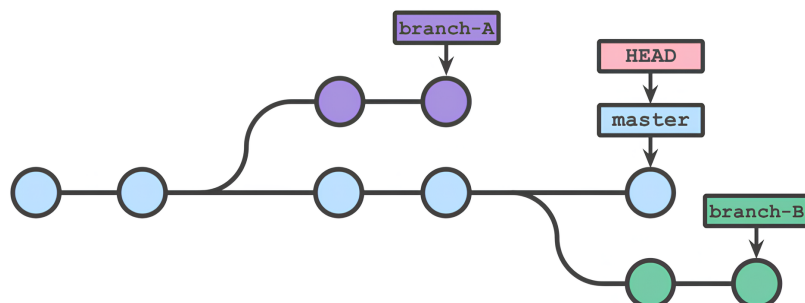
Un branch è una linea di sviluppo, composta da un insieme ordinato di commit collegati in un DAG, il quale inizia dal primo commit del repository e punta all'ultimo commit. Grazie ai branch è possibile **lavorare parallelamente** a più versioni del progetto.



### 1.2.3 HEAD

L'HEAD è un puntatore alla posizione attuale rispetto alla storia del repository e può essere aggiornato tramite il comando *checkout*.

Solitamente l'HEAD punta ad un branch o ad un tag, qualora invece puntasse ad un commit si parlerebbe di **Detached HEAD**. Quando ci si trova in questo stato i commit fatti non vengono inseriti in alcun branch, rischiando quindi di andare persi.



### 1.2.4 Tag

Un tag è un'etichetta per un commit e viene solitamente usato per segnare versioni importanti di un progetto (es. *v1.0.0*, *release-2025-09*).

### 1.2.5 Remotes

Sono dei riferimenti ai branch in repository remoti. Il nome predefinito è `origin` e vengono visualizzati nel formato `<remote>/<branch>` (es. `origin/main`).

In particolare definiamo come **tracking branch** un branch locale che tiene traccia di un branch remoto, facilitando l'uso dei comandi `git push` e `git pull`.

### 1.2.6 Merge

Il merge è un'operazione che fonde i cambiamenti apportati in due branch distinti, facendo in modo che il branch di destinazione contenga entrambi i cambiamenti e che quello di origine rimanga immutato.

Quando questa operazione viene eseguita tramite comando, Git determina in maniera autonoma quale tipo di merge sia più appropriato, basandosi sulla relazione tra i due branch e sullo storico dei loro commit.

In particolare esistono quattro tipologie di merge:

- **Fast forward:**

- *Condizione:* il branch di origine è diretto discendente di HEAD.
- *Azione:* Git sposta solo il puntatore di HEAD in avanti.
- *Risultato:* nessun nuovo merge commit.

- **Merge commit:**

- *Condizione:* i branch divergono e hanno sviluppi indipendenti;
- *Azione:* Git combina le modifiche dei due branch, creando un nuovo commit;
- *Risultato:* viene creato un merge commit con due genitori;

- **Rebase:**

- *Condizione:* si vuole aggiornare un branch basandolo su un altro, riscrivendo lo storico;
- *Azione:* Git ricrea ogni commit non in comune tra i due branch;
- *Risultato:* la storia del branch diventa lineare, senza merge commit intermedi.

- **Three way:**

- *Condizione:* storie divergenti (commit unici su entrambi i branch);
- *Punti di confronto:*
  1. Base comune (Ancestor);
  2. Versione locale (HEAD);
  3. Versione remota (Branch);
- *Azione:* Git crea un nuovo snapshot combinando le modifiche;
- *Risultato:* Viene creato un merge commit.

## 1.3 Comandi Git

Per interfacciarsi con Git vengono messi a disposizione dal sistema diversi comandi:

- `git init`: inizializza un repository creando una subdirectory `.git` all'interno della directory corrente;
- `git status`: mostra lo stato attuale del repository (file tracciati, file modificati, file nello staging, file non tracciati);
- `git diff`: mostra le differenze tra working directory, staging e commit;
- `git add <file>`: aggiunge un file alla staging area (`git add .` per aggiungere tutti i file modificati);
- `git commit -m "Messaggio"`: crea un commit, registrando le modifiche aggiunte con `git add` nella cronologia del repository;
- `git log`: mostra la lista dei commit effettuati;
- `git branch <nome>`: crea un nuovo branch con il nome indicato, ma **non ci si sposta**;
- `git checkout <nome>`: passa ad un branch esistente spostando l'HEAD;
- `git checkout -b <nome>`: crea un nuovo branch con il nome indicato, per poi **spostarsi** su quest'ultimo (`git checkout -b <nome> = git branch <nome> + git checkout <nome>`);
- `git fetch`: scarica gli aggiornamenti (commit, branch) dal repository remoto, **senza merge** col tuo branch;
- `git merge <branch>`: unisce la cronologia del branch in cui ci si trova con quella del branch specificato;
- `git pull`: scarica gli aggiornamenti (commit, branch) dal repository remoto, **facendo merge** col tuo branch (`git pull = git fetch + git merge`);
- `git push`: invia i commit locali al repository remoto, aggiornando il branch remoto corrispondente;

## 1.4 Git flow

Con il termine Git flow intendiamo un modello di branching rigido per la gestione di rilasci e cicli di sviluppo definiti.

Il suo scopo è quello di separare gli ambienti di produzione, sviluppo, funzionalità e correzioni, attraverso i seguenti branch:

- **main/master**
  - **Contenuto:** solo codice stabile, testato e rilasciato.;
  - **Checkout da:** `release-*` o `hotfix-*`;
  - **Tag:** ogni merge riceve un tag di versione (es. *v1.0*);
- **develop**
  - **Contenuto:** cronologia completa delle funzionalità di sviluppo;
  - **Checkout da:** `feature-*`;
  - **Merge in:** `release-*`
- **feature-\***
  - **Scopo:** lavoro isolato su una nuova funzionalità;
  - **Checkout da:** `develop`;
  - **Merge in:** `develop`;
  - **Regola:** non interagisce mai con `main`;
- **release-\***
  - **Scopo:** preparazione per il prossimo rilascio;
  - **Checkout da:** `develop`;
  - **Attività:** Solo bug fixing minori e aggiornamento metadata (numero di versione);
  - **Doppio merge in:** `main` per il rilascio in produzione e `develop` per preservare le correzioni;
- **hotfix-\***
  - **Scopo:** Correzione immediata di bug critici trovati nel `main`;
  - **Checkout da:** `main`
  - **Doppio merge in:** `main` per deployare subito la correzione e `develop` per garantire che il bug non riappaia in futuro;



## 1.5 Gestione dei conflitti

Immaginiamo un contesto in cui tre sviluppatori lavorano allo stesso progetto:

- **Marco** (`fix-data-leakage`): si accorge di una falla critica nel preprocessing del dataset. Ha fatto 4 commit sul suo branch;
- **Luca** (`update-rules-parser`): ha aggiornato il parser delle regole della community, modificando gli **stessi file** di preprocessing toccati da Marco. Ha fatto 3 commit sul suo branch;
- **Voi** (`main`): effettuate il merge del lavoro di Marco senza problemi e ora dovete unire il lavoro di Luca.

Nel momento in cui proverete ad effettuare il secondo merge, Git non ve lo consentirà, mettendo in pausa il merge e marcando i file in conflitto nel seguente modo:

```
1 <<<<<<< HEAD
2 # Codice sul branch main
3 =====
4 # Codice sul branch update-rules-parser
5 >>>>>>> update-rules-parser
```

A questo punto sarà necessario risolvere i conflitti tramite l'interfaccia grafica aperta dal comando `git mergetool`, oppure manualmente aprendo ogni file, rimuovendo i marcatori di conflitto e rieffettuando il commit.

## 1.6 Annullamento delle operazioni

- **Annullare in staging:** dopo aver eseguito `git add`, qualora non si volesse più committare il file, è possibile rimuoverlo dall'area di staging tramite il comando `git reset HEAD - <file>`;
- **Annullare modifiche locali:** dopo aver modificato un file, è possibile scartare le modifiche e ripristinare quest'ultimo alla versione dell'ultimo commit tramite il comando `git checkout - <file>`;
- **Annullare un commit pubblicato:** dopo aver eseguito un commit ed averlo pubblicato, è possibile annullarlo tramite il comando `git revert <hash-commit>`;
- **Annullare un commit locale:** dopo aver eseguito un commit, qualora quest'ultimo non sia ancora stato pubblicato, è possibile annullarlo tramite il comando `git reset <-soft|-hard> HEAD~1`, dove:
  - `-soft` rimuove l'ultimo commit mantenendo le modifiche nell'area di staging;
  - `-hard` rimuove l'ultimo commit cancellando completamente le modifiche;
  - `HEAD~1` indica il commit direttamente precedente ad `HEAD` (`HEAD~3` indica il 3°, etc...);
- **Riscrivere l'ultimo commit:** per rimpiazzare l'ultimo commit con uno nuovo è possibile utilizzare il comando `git commit -amend`.

# 2

## HTTP

L'**HyperText Transfer Protocol** (HTTP) è un protocollo a livello di applicazione nello stack di protocolli Internet, progettato per la trasmissione di informazioni.

La variante sicura è **HTTPS** e nel 2022 è stato pubblicato HTTP/3.

L'HTTP si basa su un'architettura **client/server**, dove:

- **Client:** detto *User Agent (UA)*, è un qualsiasi programma client che avvia una richiesta (es. browser web, app mobile...);
- **Server:** detto *Origin Server (O)*, è un programma che può originare risposte autorevoli per una data risorsa (es. sito web, telecamera per il traffico...).

### 2.1 Richieste/Risposte HTTP

L'HTTP funziona attraverso un ciclo di richieste (dal client al server) e risposte (dal server al client):

```
1  Richiesta
2  >
3  UA ===== 0
4  <
5  Risposta
```

#### 2.1.1 Richieste HTTP

Le richieste HTTP vengono inviate dal client al server e sono composte da:

- **Linea di richiesta:** contenente metodo HTTP, URI e versione del protocollo;
- **Campi di intestazione** della richiesta;
- **Corpo del messaggio** (opzionale);

```
1  GET /hello.txt HTTP/1.1 #Linea di richiesta
2  User-Agent: curl/7.64.1      # Campi
3  Host: [www.example.com](https://www.example.com) # di
4  Accept-Language: en, it      # intestazione
5  # Corpo del messaggio assente
```

### 2.1.2 Risposte HTTP

Le risposte HTTP vengono inviate dal server al client e sono composte da:

- **Stato di completamento** riguardo la richiesta;
- **Campi di intestazione** della risposta;
- **Contenuto della risposta** (opzionale).

```
1 HTTP/1.1 200 OK # Stato di completamento
2 Date: Mon, 27 Jul 2009 12:28:53 GMT # |
3 Server: Apache # |
4 Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT # |
5 ETag: "34aa387-d-1568eb00" # | Campi di
6 Accept-Ranges: bytes # | intestazione
7 Content-Length: 51 # |
8 Vary: Accept-Encoding # |
9 Content-Type: text/plain # |
10 Hello World! My content includes a trailing CRLF. # Contenuto della risposta
```

## 2.2 Intermediari

Il protocollo HTTP prevede che tra l'User Agent e l'Origin Server possano esserci uno o più intermediari, i quali possono inoltrare, filtrare o modificare le richieste e le risposte HTTP (es. proxy, gateway, tunnel...).

```
1 > > > > UA = User Agent
2 UA ===== A ===== B ===== C ===== O A, B, C = Intermediari
3 < < < < O = Origin Server
```

In particolare i proxy (a differenza dei tunnel) possono utilizzare di un sistema di **cache**, memorizzando le risposte in modo da poter servire richieste future senza contattare nuovamente il server. Per poter essere memorizzata, una risposta deve essere dichiarata dal server come **cacheable**.

```
1 > > UA = User Agent
2 UA ===== A ===== B ----- C ----- O A, B, C = Intermediari
3 < < O = Origin Server
```

## 2.3 Metodi HTTP

I metodi HTTP specificano l'azione che il client desidera eseguire su una risorsa. Ogni metodo può avere o meno le seguenti proprietà:

- **Safe** (*sicuro*): la richiesta non ha effetti collaterali sulla risorsa (sola lettura);
- **Idempotent** (*idempotente*): eseguire più volte la stessa richiesta produce lo stesso effetto di una singola richiesta;
- **Cacheable** (*memorizzabile nella cache*): la risposta può essere memorizzata nelle cache.

### 2.3.1 PUT

Il metodo PUT consente di creare una nuova risorsa, specificandola nella richiesta, o di sovrascriverla qualora l'URI esista già. Si tratta di un metodo **idempotente**.

```
1 PUT /course-descriptions/web-and-software-architecture
```

### 2.3.2 GET

Il metodo GET consente di richiedere una rappresentazione dello stato di una risorsa. Si tratta di un metodo **safe**, **idempotente** e **cacheable**.

```
1 GET /course-descriptions/web-and-software-architecture
```

### 2.3.3 POST

Il metodo POST consente di creare o modificare un subordinato della risorsa indicata nell'URI oppure di attivare un'azione. Si tratta di un metodo **cacheable**.

```
1 POST /announcements/
```

```
1 POST /announcements/{id}/comments/
```

```
1 POST /users/{id}/email
```

### 2.3.4 DELETE

Il metodo DELETE consente di richiedere al server l'eliminazione della risorsa specificata nell'URI. Si tratta di un metodo **idempotente**.

```
1 DELETE /courses/web-and-software-architecture
```

### 2.3.5 Altri metodi

Metodo	Descrizione	Safe	Idempotente	Cacheable
<b>HEAD</b>	Come GET, ma non trasferisce il contenuto della risposta.	Sì	Sì	Sì
<b>CONNECT</b>	Stabilisce un tunnel verso il server identificato dalla risorsa target.	No	No	No
<b>OPTIONS</b>	Descrive le opzioni di comunicazione per la risorsa target.	Sì	Sì	No
<b>TRACE</b>	Esegue un test di loop-back del messaggio lungo il percorso verso la risorsa target.	Sì	Sì	No
<b>PATCH</b>	Modifica parzialmente una risorsa, invece di sostituirla interamente come fa PUT.	No	No	No

## 2.4 Codici di stato della risposta

I codici di stato sono rappresentati da un numero a tre cifre nell'intervallo 100-599 e descrivono il risultato della richiesta e la semantica della risposta.

Intervallo	Descrizione
<b>1xx</b>	<i>Informazioni</i> : La richiesta è stata ricevuta, processo in corso.
<b>2xx</b>	<i>Successo</i> : La richiesta è stata ricevuta, compresa e accettata con successo.
<b>3xx</b>	<i>Reindirizzamento</i> : Sono necessarie ulteriori azioni per completare la richiesta.
<b>4xx</b>	<i>Errore del client</i> : La richiesta contiene sintassi errata o non può essere soddisfatta.
<b>5xx</b>	<i>Errore del server</i> : Il server non è riuscito a soddisfare una richiesta apparentemente valida.

In particolare i codici di stato più comuni sono:

Codice di stato	Descrizione
<b>200 OK</b>	In una richiesta GET, la risposta conterrà la risorsa richiesta; in una richiesta POST, la risposta conterrà un'entità che descrive o contiene il risultato dell'azione.
<b>201 Created</b>	La richiesta è stata soddisfatta, risultando nella creazione di una nuova risorsa.
<b>204 No Content</b>	Il server ha elaborato con successo la richiesta e non sta restituendo alcun contenuto.
<b>301 Moved Permanently</b>	Questa e tutte le richieste future dovrebbero essere indirizzate all'URI fornito.
<b>302 Found</b>	Guarda un'altra URL.
<b>400 Bad Request</b>	Apparente errore del client.
<b>401 Unauthorized</b>	È richiesta l'autenticazione.
<b>403 Forbidden</b>	La richiesta conteneva dati validi ed è stata compresa dal server, ma l'azione è proibita.
<b>404 Not Found</b>	La risorsa non è stata trovata ma potrebbe essere disponibile in futuro.
<b>405 Method Not Allowed</b>	Il metodo di richiesta non è supportato.
<b>500 Internal Server Error</b>	Condizione inaspettata riscontrata.
<b>501 Not Implemented</b>	Metodo di richiesta non riconosciuto, oppure il server non ha la capacità di soddisfare la richiesta.
<b>502 Bad Gateway</b>	Un gateway o proxy ha ricevuto una risposta non valida dal server a monte.
<b>503 Service Unavailable</b>	Server sovraccarico o inattivo per manutenzione (temporaneo).
<b>504 Gateway Timeout</b>	Il server non ha ricevuto una risposta tempestiva dal server a monte.

# 3

## API

### 3.1 JSON e YAML

#### 3.1.1 JSON

**JavaScript Object Notation** (JSON) è un formato testuale leggero per lo scambio e l'archiviazione di dati, derivato dalla sintassi JavaScript.

Si tratta di un formato facile da leggere e scrivere sia per gli umani che per le macchine e questo perchè si basa unicamente sui concetti di **oggetto** e di **array**.

In particolare gli oggetti sono una collezione non ordinata di coppie chiave:valore, racchiusa tra parentesi graffe, mentre gli array sono un elenco ordinato di valori (anche eterogenei), separati da virgole e racchiuso tra parentesi quadre.

```
1  {
2    "user": {
3      "id": 12345,
4      "username": "mario.rossi",
5      "isActive": true,
6      "lastLogin": null,
7      "hobbies": ["programmazione", "videogames", "calcio"]
8      "address": {
9        "street": "Via Roma 10",
10       "zipCode": "20100",
11       "country": "Italia"
12     }
13   }
14 }
```

#### 3.1.2 YAML

**YAML Ain't Markup Language** (YAML) è un formato testuale per la serializzazione dei dati utilizzato per file di configurazione e per archiviare o scambiare dati.

```
1  user:
2    id: 12345 #ID dell'utente
3    username: "mario.rossi"
4    isActive: true
5    lastLogin: null
6    hobbies:
7      - "programmazione"
8      - "videogames"
9      - "calcio"
10   address:
11     street: "Via Roma 10"
12     zipCode: "20100"
13     country: "Italia"
```

## 3.2 API

Un'**Application Programming Interface** (API) è la **definizione delle interazioni consentite** tra due parti di un software. Funge cioè da contratto di interazione tra il **consumer** (client) ed il **provider** (servizio), specificando:

- **Richieste** possibili;
- **Parametri** delle richieste;
- **Valori di ritorno**;
- **Formati di dato** richiesti (es. JSON, XML, YAML...).

L'adozione di un API porta vantaggi fondamentali nell'architettura software:

- **Interfaccia esplicita**: definisce chiaramente le aspettative e le modalità di interazione;
- **Contratto infrangibile**: stabilisce un insieme di regole che entrambe le parti devono rispettare;
- **Information Hiding** (occultamento delle informazioni): la logica interna del provider rimane nascosta al consumer, che deve conoscere unicamente l'interfaccia.

### 3.2.1 Categorie di API

È possibile suddividere le API in base alla loro posizione e funzione in:

- **API Locali**:
  - API per i linguaggi di programmazione (es. libreria standard di Python);
  - API del sistema operativo;
  - API delle librerie software;
  - API hardware;
- **API Remote (Web API)**:
  - Interfacce di programmazione basate su protocolli di rete (tipicamente HTTP), come le API RESTful.

Inoltre, è possibile suddividere le API sulla base dell'accessibilità in:

- **API Private**: destinate all'uso interno di un'azienda o un sistema chiuso, prevedono che l'accesso sia limitato ai componenti interni;
- **API Pubbliche**: disponibili per l'uso da parte del pubblico, prevedono che l'accesso possa essere limitato ad alcuni utenti tramite **API Tokens**.

### 3.2.2 Interfaccia e stabilità

Col tempo i cambiamenti alle API potrebbero rompere le compatibilità con i client esistenti. Si utilizzano quindi dei **marcatori di stato**:

- **beta**: indica che le parti potrebbero cambiare perché non sono ancora stabili;
- **deprecated**: indica che le parti verranno rimosse o non saranno più supportate in futuro.

### 3.2.3 Documentazione delle API

È possibile definire esplicitamente un API attraverso **documentazione** (testo, esempi, manuali), oppure attraverso un **linguaggio di descrizione** standardizzato, il quale formalizza il contratto, consentendo la generazione automatica di documentazione, codice client e validazione.

In particolare il linguaggio di descrizione leader del settore per le API moderne basate su HTTP è **OAS (OpenAPI Specification)**, un formato di descrizione **vendor-neutral** (indipendente dal fornitore).

I file OpenAPI vengono solitamente scritti in formato YAML, data la sua leggibilità.

```
1 openapi: 3.0.0
2 info:
3   title: "An example OpenAPI document"
4   description: |
5     This API allows writing down marks on a Tic Tac Toe board
6     and requesting the state of the board or of individual cells.
7   version: 0.0.1
8 paths: {} # Gli endpoint dell'API verrebbero definiti qui
```

## 3.3 REST

Il **Representational State Transfer** (REST) è uno stile architetturale per sistemi ipermediali distribuiti, il cui obiettivo è **trasferire la rappresentazione delle risorse** da un componente (es. il server) ad un altro (es. il client).

### Definizione 3.1: Risorsa

Una risorsa è qualsiasi **informazione che possa essere nominata**: un documento, un'immagine, un servizio, un oggetto non virtuale o una collezione di altre risorse.

Il contenuto di una risorsa può variare nel tempo, e due risorse diverse possono, in un determinato momento, mappare agli stessi valori.