



JEMORE
TOGETHER FOR SUCCESS

Come progettare un backend

Database

Overview

*E' un insieme organizzato di dati memorizzati in modo strutturato ed accessibili.
Un database è composto da:*

- *Database Management System (DBMS)*
- *Linguaggio di interrogazione*
- *Modello dei dati*

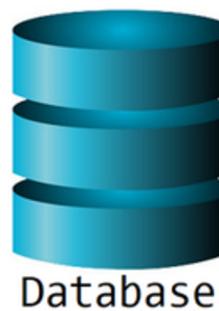
I principali 2 tipi di database sono: relazionali e non-relazionali.



Database Relazionali

Overview

Il modello di rappresentazione dei dati è a tavelle. Le varie tavelle sono tra di loro relazionate, e possono avere vari tipi di molteplicità. Il linguaggio di interrogazione è di tipo dichiarativo e si chiama: SQL



Query SQL di base



USE database

SHOW TABLES

SELECT * FROM users WHERE età=20

INSERT INTO table_name (column1, column2, column3) VALUES (value1, value2, value3);

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste'

UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;

Database relazionali

Primary Key



Una "Primary key" serve per rendere il singolo record univoco all'interno della sua tabella. La scelta della primary key è importante, e deve essere presa in considerazione secondo i seguenti punti:

- *Può essere composta da 1 o più attributi.*
- *Solo 1 record per tabella può avere un singolo valore o una combinazione di valori sui campi della primary key.*
- *Nessun attributo dentro la primary key può essere nullo.*
- *Bisogna sceglierlo in modo da minimizzare lo spazio ma anche in modo da rendere la logica sensata.*

Persona {(Constantino, Grana, 50), (Maria, Manfredini, 46), ... }

(Costantino, Grana, 49)

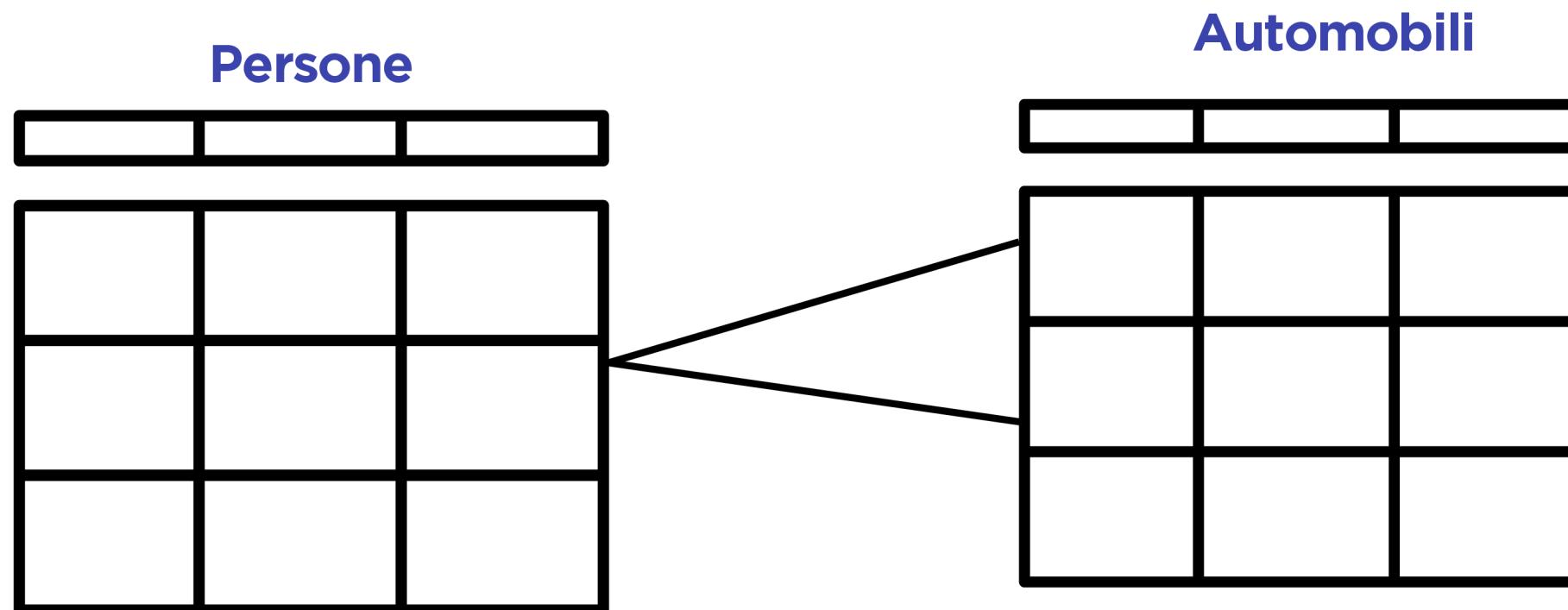


(Maria, Giuliani, 32)



Database Relazionali

Relazioni tra tabelle - 1 a m

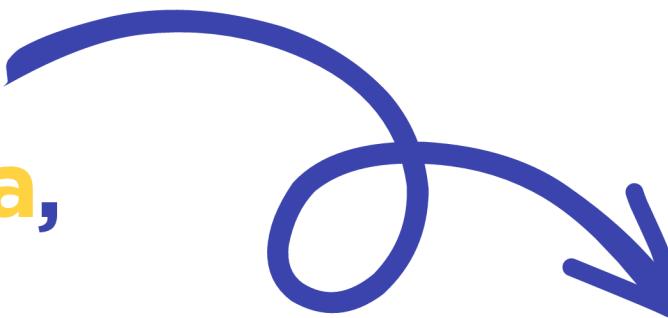


**Persona { CF,
nome, cognome,
età }**

**Automobile { Targa,
colore, modello,
marca }**

Foreign Key

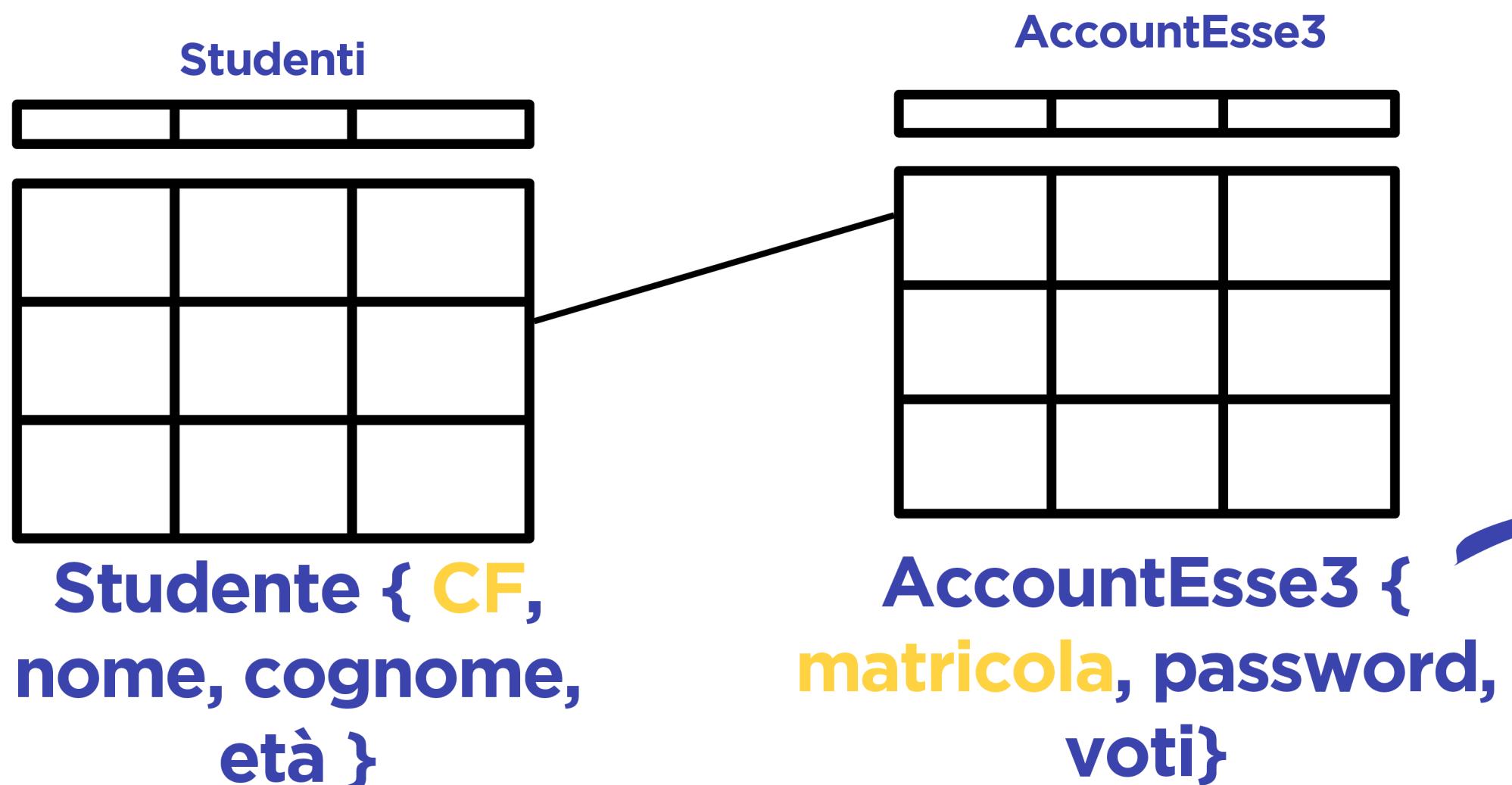
**Non è (nativamente) possibile impostare
cardinalità limitate! (Ad es. 1 a 5).
E' possibile aggiungere dei CONSTRAINT
CHECK che facciano questo controllo.**



**Automobile { Targa,
,CF_Persona, colore,
modello, marca }**

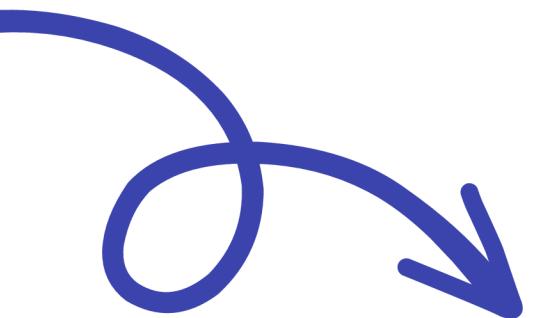
Database Relazionali

Relazioni tra tabelle - 1 a 1



Se ogni studente è OBBLIGATO ad avere un account esse3 è possibile anche fare REIFICAZIONE unendo i 2 record in un unico:

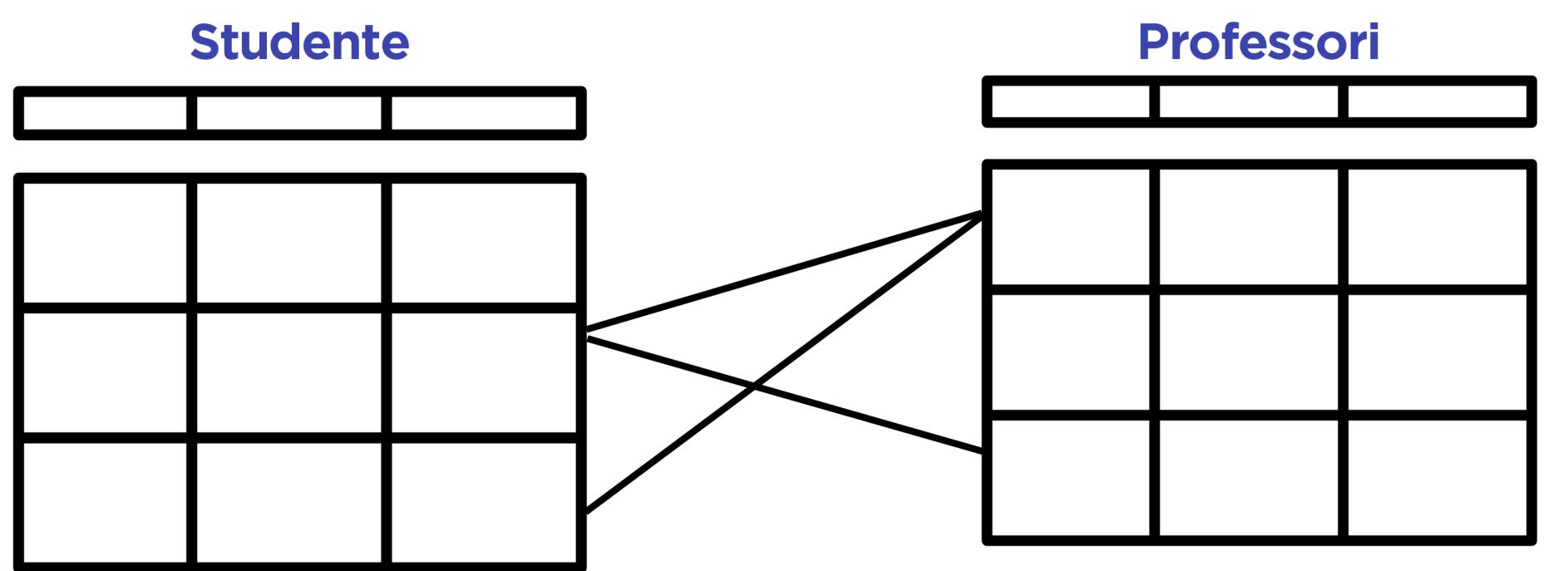
StudenteConAccount { CF, nome, cognome, età, matricola, password, voti }



**AccountEsse3 {
matricola, password,
voti, CF_Studente}**

Database Relazionali

Relazioni tra tabelle - m a m



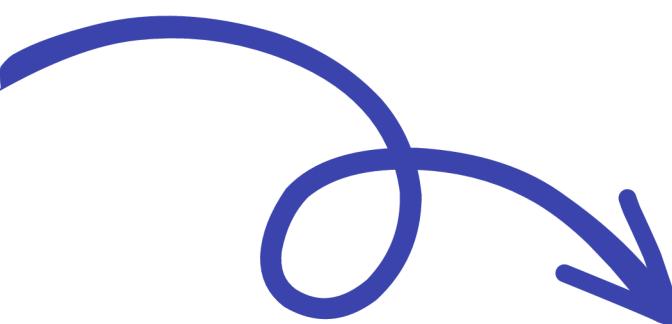
Studente {
Matricola, nome,
cognome, età }

Professori {
CF,
nome, cognome,
materia,
stipendio }

I campi **CF_Professore** e
Matricola_Studente sono sia Primary Key
che due Foreign key relative alle tabelle.

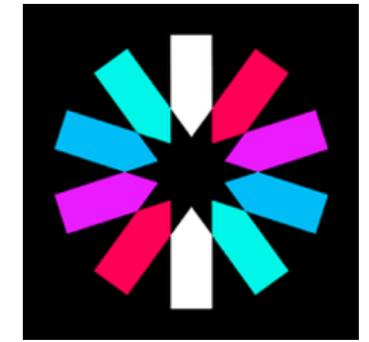
Inoltre nella nuova tabella creata
"insegnamento" è possibile inserire
ulteriori attributi relativi alla relazione.

Come fare se voglio che un professore
faccia più di un insegnamento ad uno
studente?

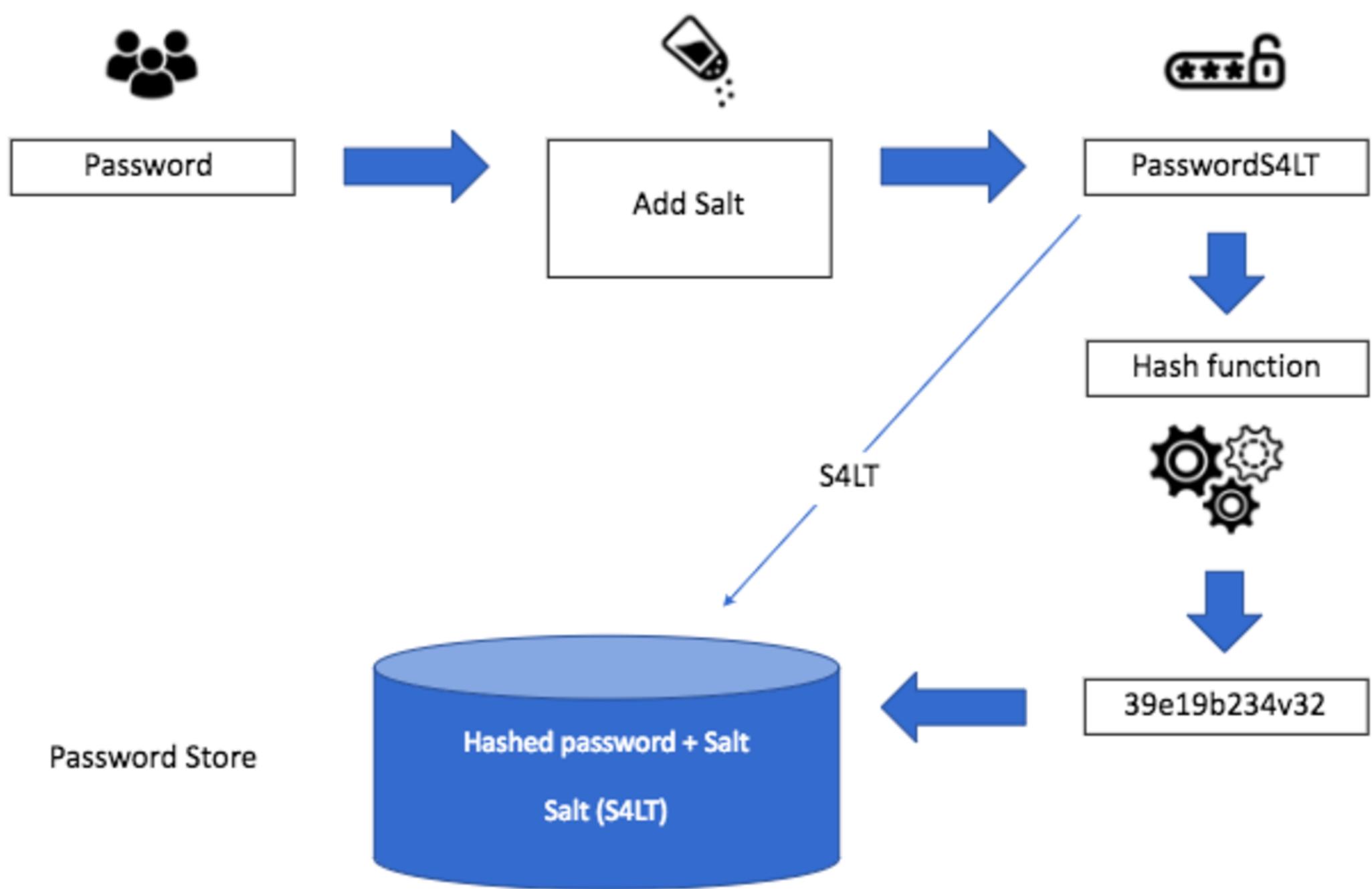


Insegnamento {
CF_Professore,
Matricola_Studente
}

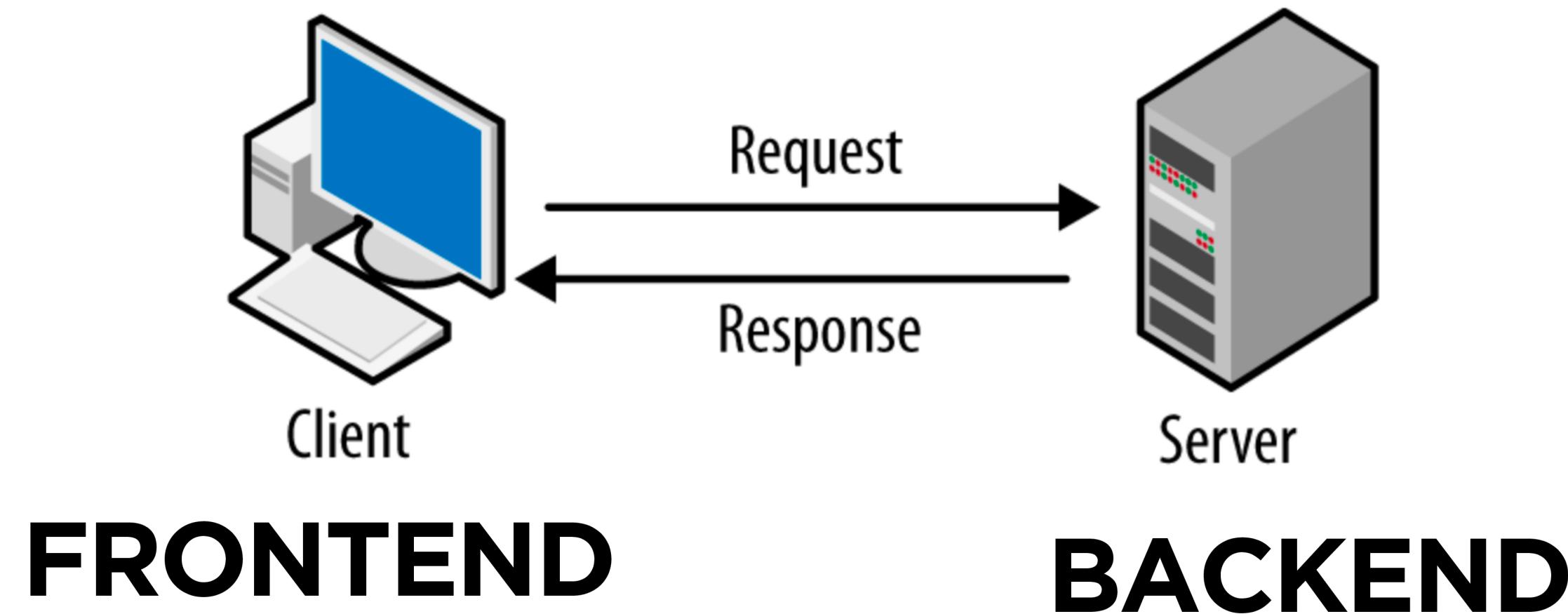
Sistemi di autenticazione



Salvare le password in chiaro è una boiata



Modello client-server

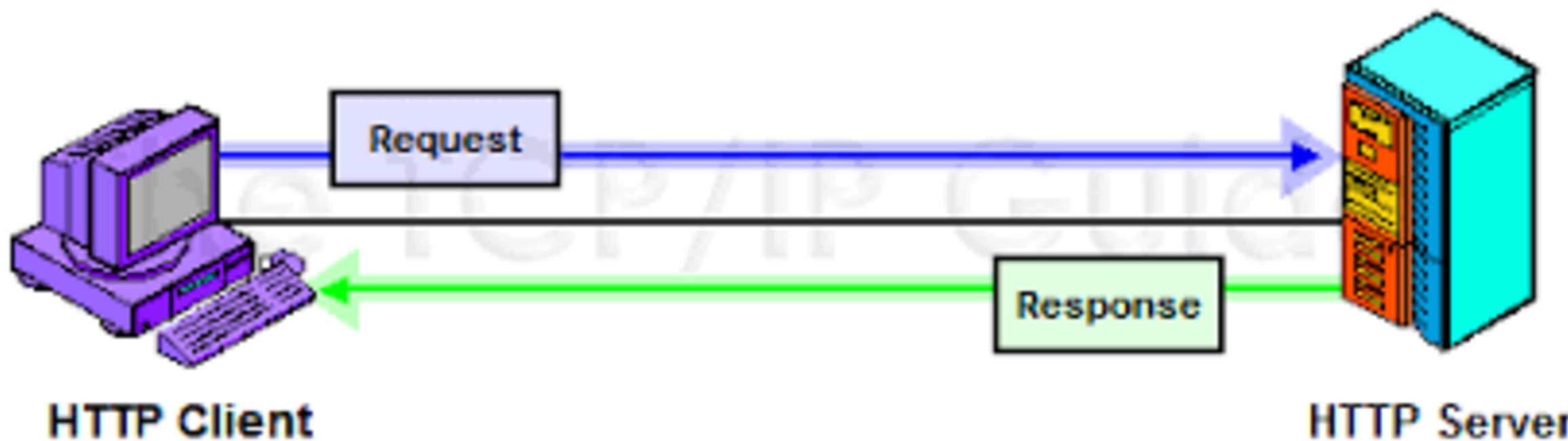


Protocollo HTTP

Hypertext Transfer Protocol

E' un protocollo di rete applicativo usato per la trasmissione di informazioni su architetture di tipo client-server.

Il client esegue delle RICHIESTE e il server fornisce delle RISPOSTE, entrambe di tipo HTTP.



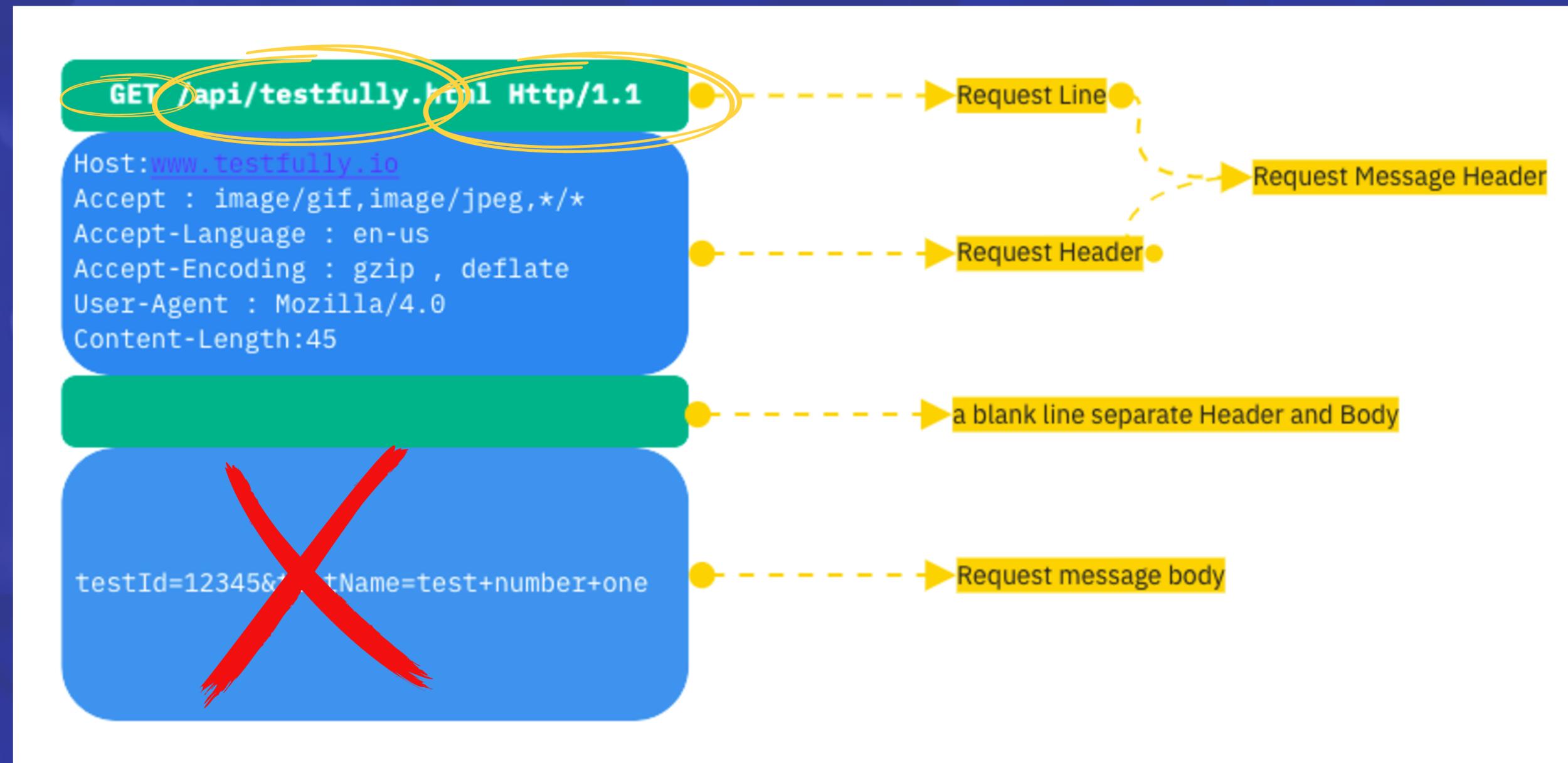
Richieste HTTP

Metodi

- **GET:** richiesta di risorse.
- **POST:** creazione di risorse.
- **PUT:** aggiornamento di risorse.
- **PATCH:** aggiornamento parziale di risorse.
- **DELETE:** eliminazione di una risorse.
- **HEAD:** per ricevere informazioni su una risorsa.
- **OPTIONS:** per ricevere informazioni riguardante funzionalità del server e i formati supportati.
- **TRACE:** per ricevere richieste inviate al server (per debug)
- **CONNECT:** per creare connessioni sicure.

Richieste HTTP

Struttura



E' possibile inserire parametri in una richiesta HTTP
in questo modo: /url?param1=A¶m2=B&...

Richieste HTTP

Alcuni header

- **Accept:** formato di dati accettati (XML, JPEG, JSON...)
- **Accept-Language:** lingua formato.
- **Cache-Control:** come il server deve gestire la cache.
- **Content-Type:** formato dei dati passati nel body (XML, JSON...)
- **If-Match:** se l'E-Tag coincide.
- **If-Modified-Since:** se la risorsa è stata modificata da quella data.
- **If-Not-Modified-Since:** se la risorsa non è stata modificata da quella data.
- **Content-Length:** grandezza delle informazioni.

Richieste HTTP

Esempio con JQuery (Ajax)

```
$ajax({
  url: 'https://jsonplaceholder.typicode.com/posts',
  params: {
    date: '2020-01-01',
    region: 'IT'
  },
  type: 'POST',
  headers: {
    'If-Not-Modified-Since': 'Mon, 26 Jul 1997 05:00:00 GMT',
  },
  data: JSON.stringify( value: {
    title: 'Titolo del post',
    content: 'Contenuto del post',
    userId: 1
  }),
  contentType: 'application/json',
  success: function (data) : void {
    console.log(data);
  },
  error: function (error) : void {
    console.error(error);
  }
});
```

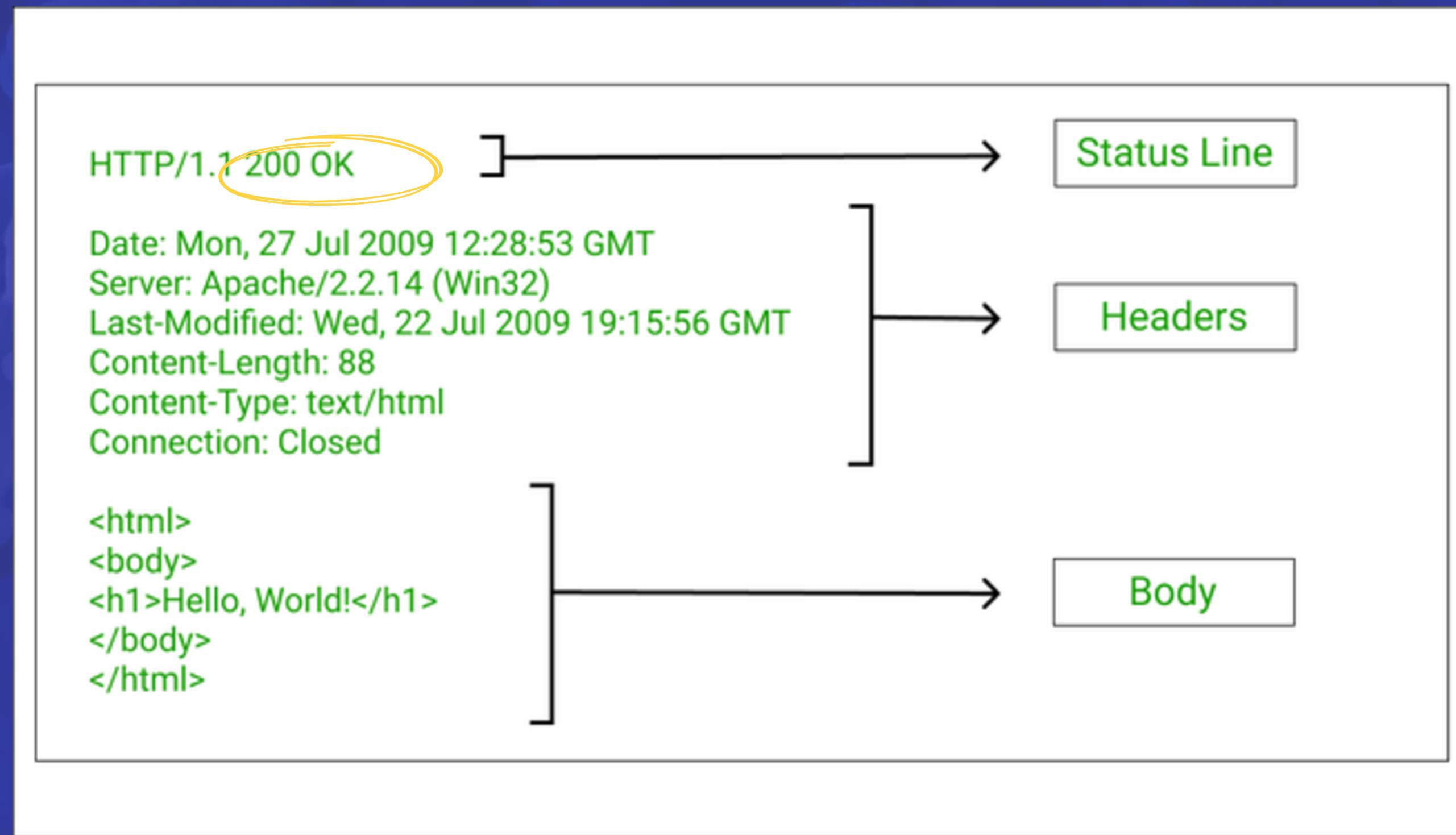
La maggior parte delle librerie comporranno le richieste HTTP al posto vostro!

/💡
POST /posts?date=2020-01-01®ion=IT HTTP/1.1
Host: jsonplaceholder.typicode.com
Content-Type: application/json
Content-Length: 57

{"title": "Titolo del post", "content": "Contenuto del post", "userId": 1}
*/

Risposte HTTP

Struttura



Risposte HTTP

Codici

- 1xx: messaggi informativi
- 2xx: successo
- 3xx: si redirige la richiesta.
- 4xx: errore nella richiesta del client
- 5xx: errori interni al client

40

- 200: OK
- 302: Found
- 400: Bad Request
- 404: Not Found
- 500: Internal error
- 502: Bad Gateway

EXAMPLE

Risposte HTTP

Esempio con Express

```
app.get('/api/users/:id', (req, res) : void => {
    // Ottenimento dell'ID dall'URL della richiesta
    const userId = req.params.id;

    // Logica per ottenere l'utente con l'ID specificato
    const user : {id: any, name: string} = { id: userId, name: 'Mario Rossi' };

    // Invio della risposta con l'utente trovato
    res.send(user);
};
```

Le risposte HTTP saranno generate e gestite da pezzi di programma chiamati "endpoint" (si vedrà dopo)

```
/*
// Richiesta
GET /api/users/123 HTTP/1.1
Host: example.com

// Risposta
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": "123",
    "name": "Mario Rossi"
}
*/
```

JSON e XML

JavaScript Object Notation

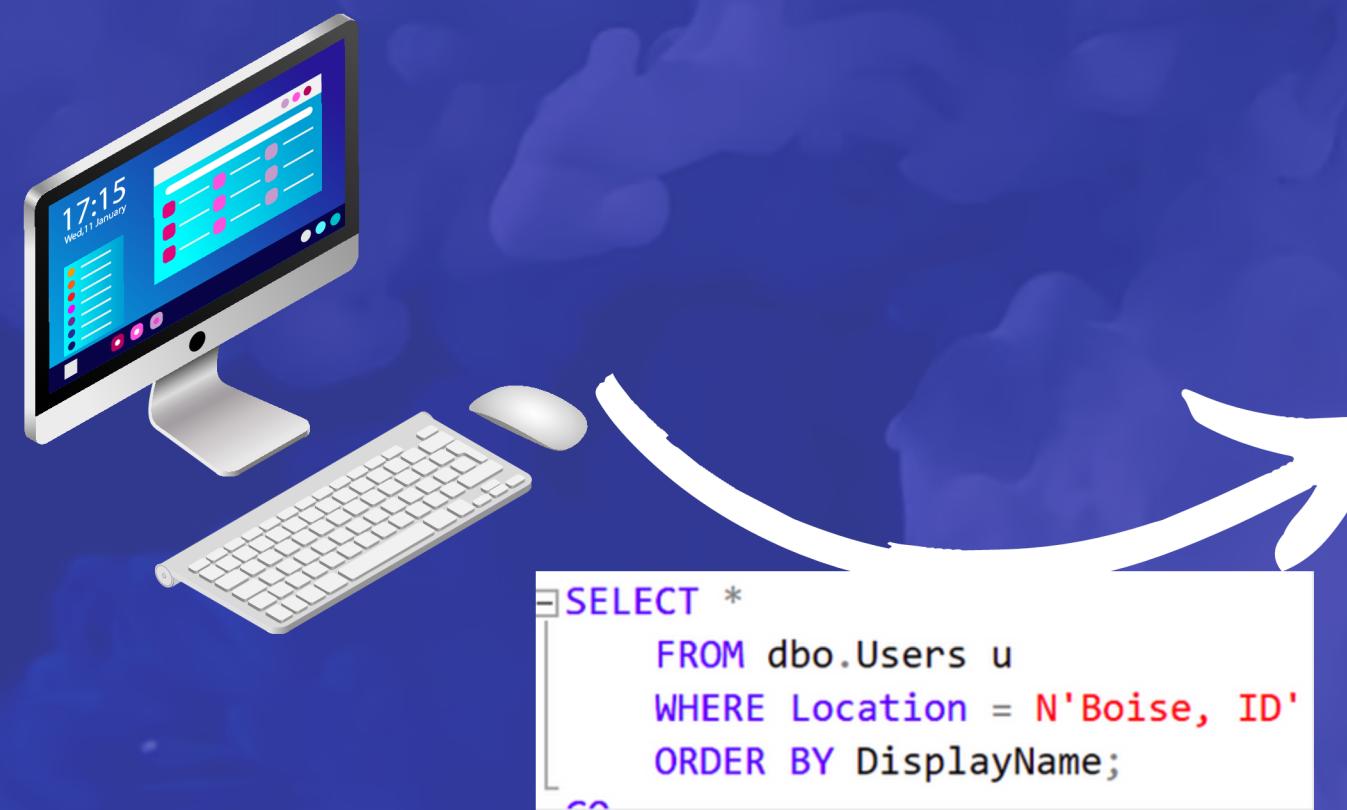
```
{  
  "nome" : "Gabriele",  
  "cognome" : "Bongiorno",  
  "età" : 20,  
  "colleghi" : [  
    { "nome" : "Davide", "roulo" : "responsabile" },  
    { "nome" : "Simone", "roulo" : "responsabile" },  
    { "nome" : "Federico", "roulo" : "consulente" },  
    { "nome" : "Claudio", "roulo" : "zavorra umana" }  
  ],  
  "odio-jetop" : true  
}
```

eXtensible Markup Language

```
<CATALOG>  
  <SPRING>  
    <TITLE>Garden Sales</TITLE>  
    <LINE>Outdoor_Tools</LINE>  
    <PAGE>  
      <CAPTION>Goodbye, Winter!</CAPTION>  
      <ITEM>Gardening Gloves</ITEM>  
      <ITEM>Potting Soil</ITEM>  
    </PAGE>  
  </SPRING>  
</CATALOG>
```

Architettura API

Motivazione



Ad un cliente serve un dato in un server. Come può prenderlo?
Una prima idea è quello di mandargli delle query SQL via internet.

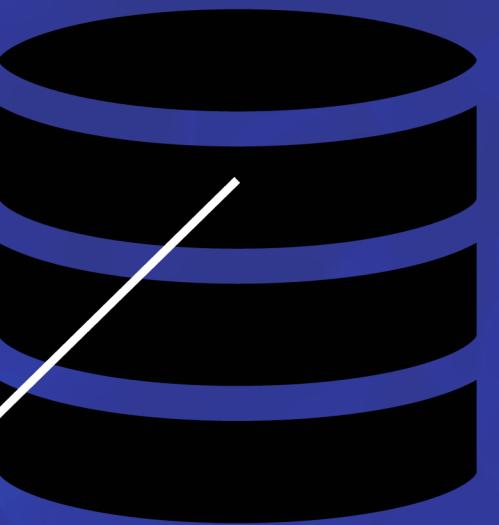
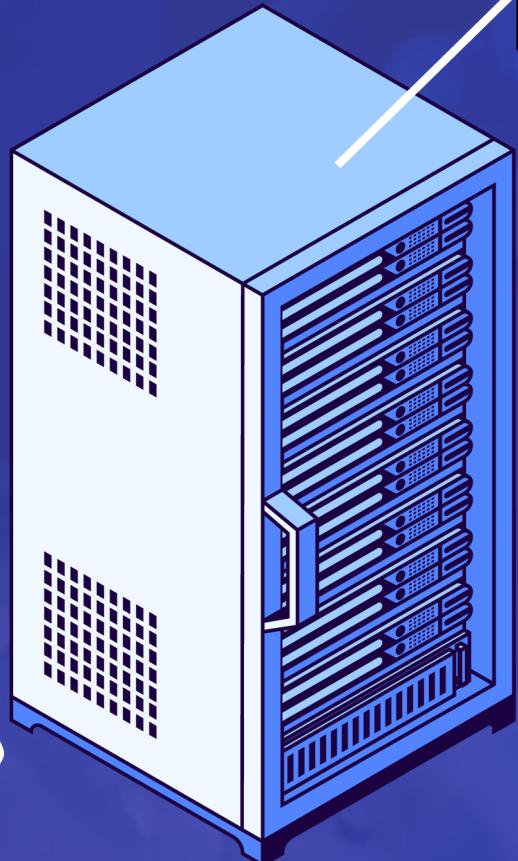
Architettura API

Motivazione



FROM dbo.Users u
WHERE Location = N'Boise, ID'
ORDER BY DisplayName;

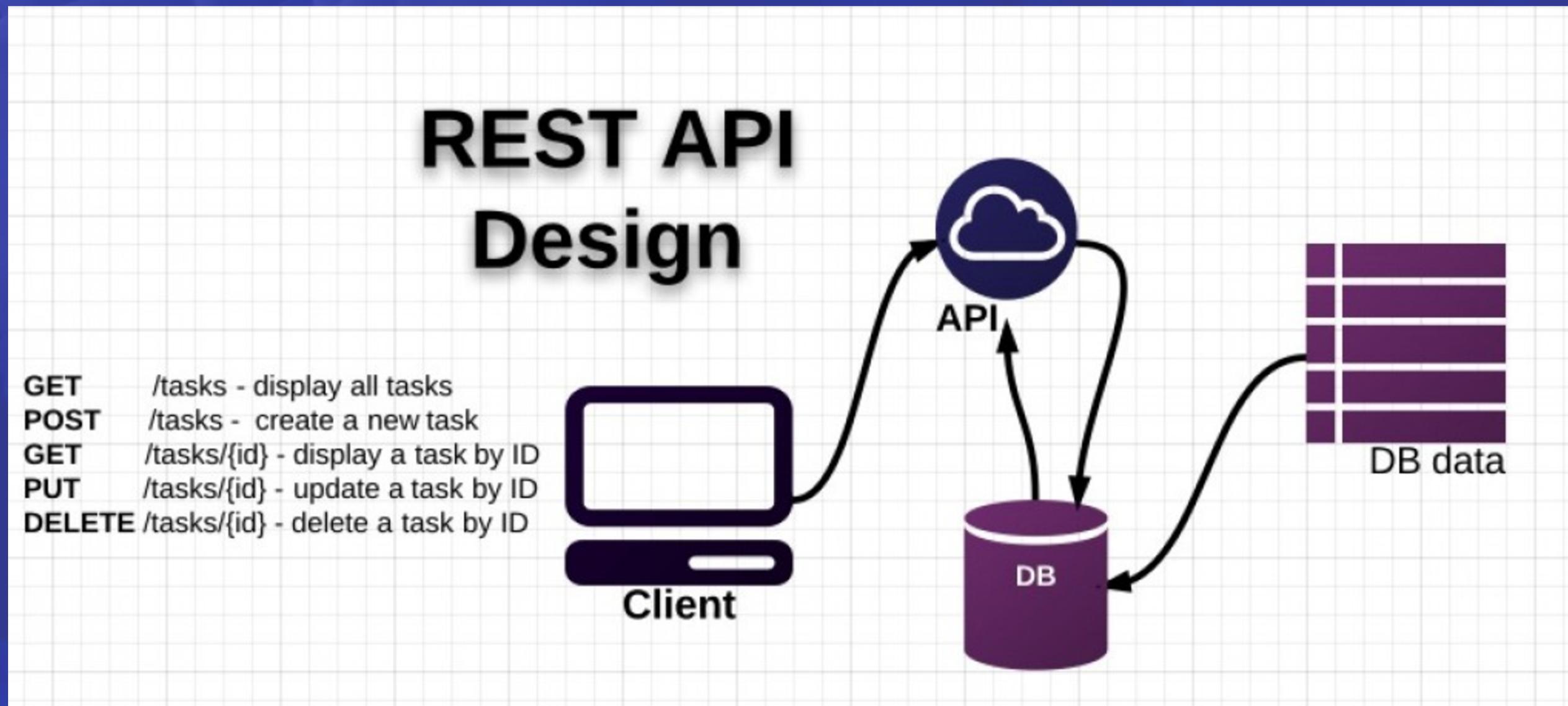
GO



- E' poco sicuro.
- E' poco flessibile.
- E' poco scalabile.
- Guerra non approva

Architettura API

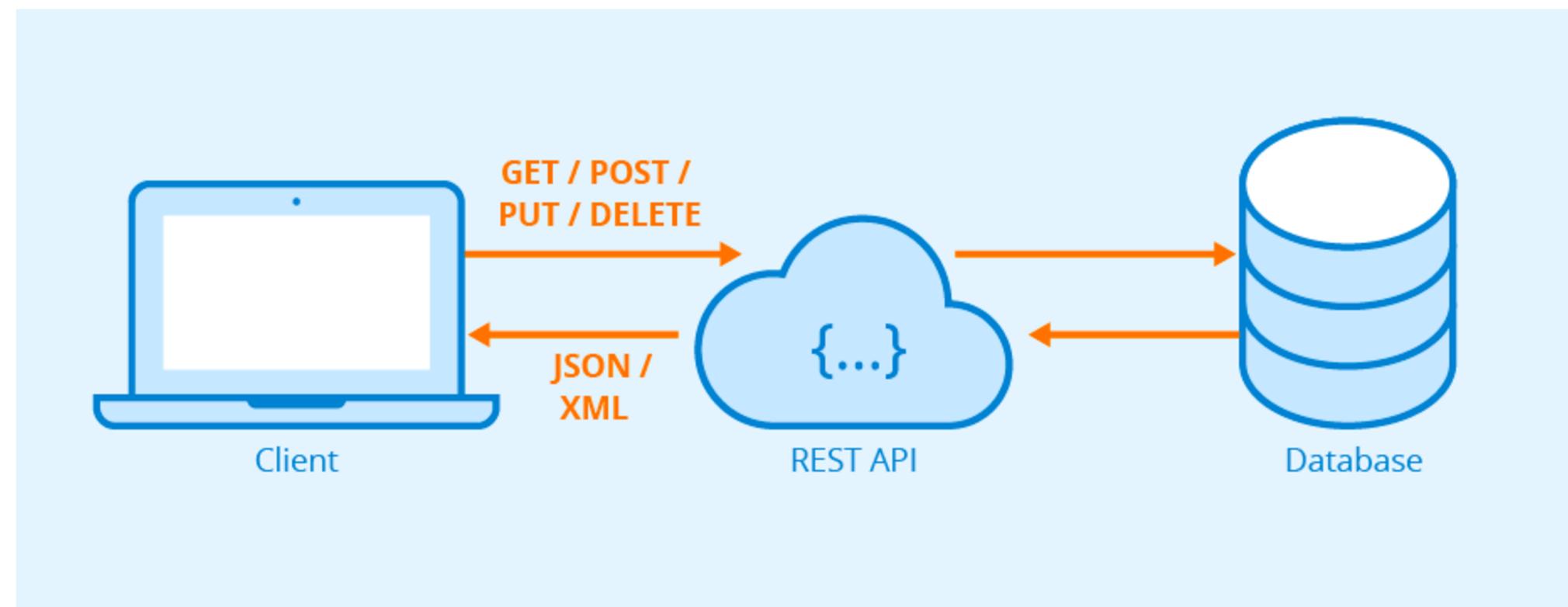
Soluzione



E' una soluzione architetturale che permette l'accesso al Database In modo indiretto, più elegante e più sicuro.

RESTful

API: APplication user Interface



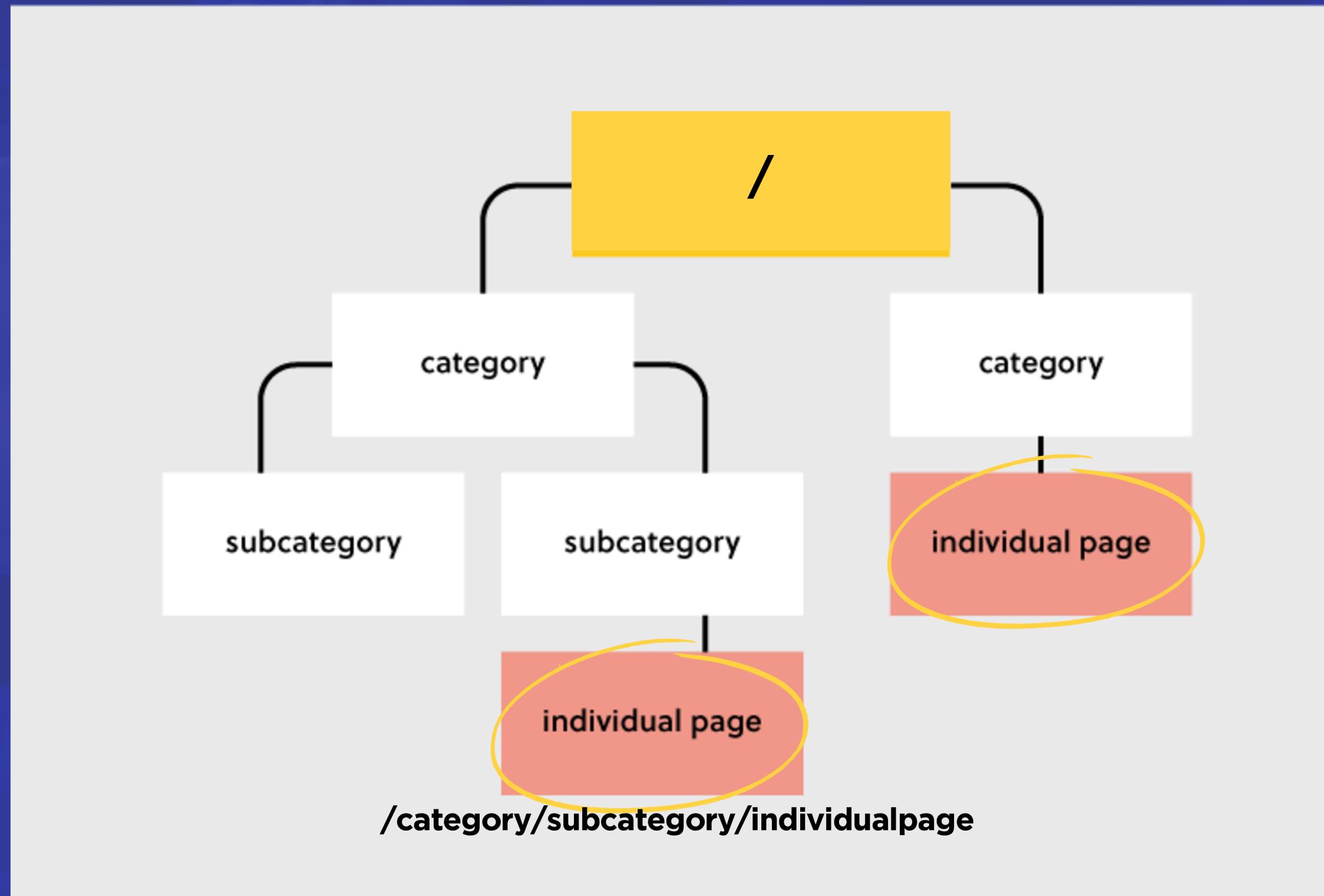
Framework e librerie



express



Endpoint API



I dati delle API sono organizzati e accessibili tramite degli URL organizzati in modo gerarchico. Gli endpoint sono dei "pezzi di programma" implementate sul server che gestiscono i metodi delle API (per esempio, REST) degli URL.

Endpoint API

Esempio con Express

```
// GET endpoint per ottenere tutti gli utenti:  
new *  
app.get('/users', (req, res) : void => {  
    // logica per ottenere tutti gli utenti  
    res.send(users);  
});  
  
// POST endpoint per creare un nuovo utente:  
new *  
app.post( path: '/users', handlers: (req, res) : void =>  
    const newUser = req.body;  
    // logica per creare un nuovo utente  
    res.send(newUser);  
});
```

```
// PUT endpoint per aggiornare un utente esistente:  
new *  
app.put( path: '/users/:id', handlers: (req, res) : void => {  
    const userId = req.params.id;  
    const updatedUser = req.body;  
    // logica per aggiornare l'utente con l'id specificato  
    res.send(updatedUser);  
});  
  
// DELETE endpoint per eliminare un utente esistente:  
new *  
app.delete( path: '/users/:id', handlers: (req, res) : void => {  
    const userId = req.params.id;  
    // logica per eliminare l'utente con l'id specificato  
    res.send( body: `User with id ${userId} has been deleted`);  
});
```

parametro di percorso dinamico

Metodo CRUD

Come standardizzare la struttura di una API

Il metodo CRUD (Create, Read, Update, Delete) è un modello di progettazione per le API RESTful che si riferisce alle quattro operazioni fondamentali che si possono eseguire su un database o su una risorsa in un'applicazione web. Queste operazioni sono:

-  • Create: Questa operazione consente di creare una nuova risorsa. In un'API RESTful, questa operazione è tipicamente eseguita tramite il metodo HTTP POST.
-  • Read: Questa operazione consente di leggere o recuperare una risorsa esistente. In un'API RESTful, questa operazione è tipicamente eseguita tramite il metodo HTTP GET.
-  • Update: Questa operazione consente di aggiornare una risorsa esistente. In un'API RESTful, questa operazione è tipicamente eseguita tramite il metodo HTTP PUT o PATCH.
-  • Delete: Questa operazione consente di eliminare una risorsa esistente. In un'API RESTful, questa operazione è tipicamente eseguita tramite il metodo HTTP DELETE.

Questi metodi HTTP corrispondono alle operazioni CRUD e permettono di interagire con le risorse in un'API RESTful. Questo modello di progettazione è fondamentale per la creazione di applicazioni web affidabili e scalabili.

Accesso ai dati

Problema

```
// Endpoint che esegue una query SQL
new *

app.get('/users', (req, res) : void => {
    // Query SQL da eseguire
    const sql : "SELECT * FROM users" = 'SELECT * FROM users';
    // Esecuzione della query
    connection.query(sql, (err, results) => {
        // Gestione dell'errore
        if (err) {
            console.error('Errore nella query SQL: ' + err.stack);
            return res.status( code: 500).json( body: { error: 'Errore nella query SQL' });
        }
        // Invio della risposta con i risultati della query SQL in formato JSON
        res.json(results);
    });
});
```

Accesso ai dati

Problema

```
// Endpoint che esegue una query SQL
new *  

app.get('/users', (req, res) : void => {
    // Query SQL da eseguire
    const sql : "SELECT * FROM users" = SELECT * FROM users';
    // Esecuzione della query
    connection.query(sql, (err, results) => {
        // Gestione dell'errore
        if (err) {
            console.error('Errore nella query SQL' + err.stack);
            return res.status(500).json({ body: { error: 'Errore nella query SQL' } });
        }
        // Invio della risposta con i risultati della query SQL in formato JSON
        res.json(results);
    });
});
```

Grana incazzato

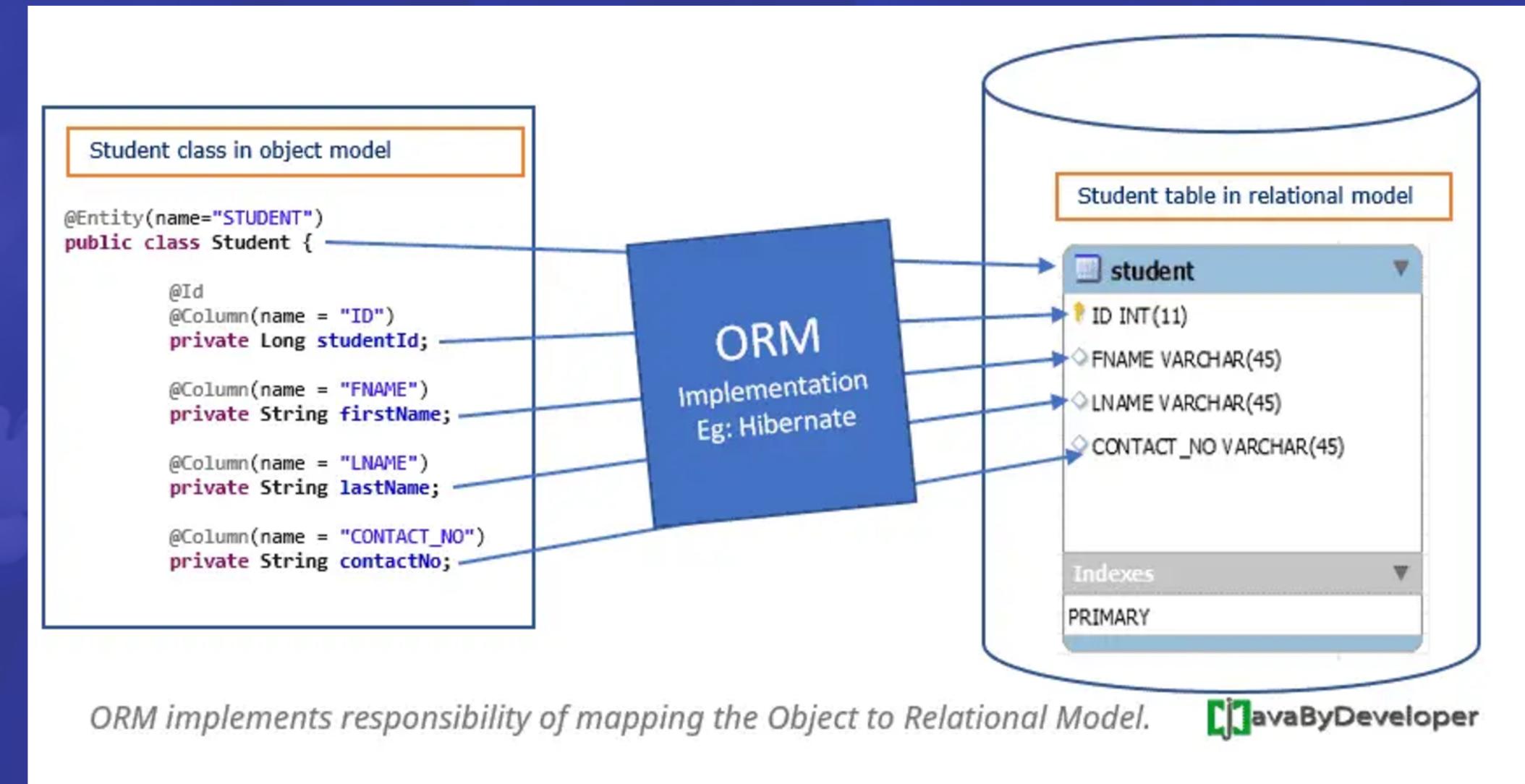


Questo approccio fa cagare!

E' codice non riutilizzabile, poco leggibile ed è un approccio per niente elastico!

ORM

Object Relational Mapping



E' un Design pattern che permette di trattare i dati di un qualsiasi database (relazionale e non) come se fossero degli oggetti. Molte librerie/framework offrono già strumenti per usare questo design, e spesso la relazione classi-tabelle SQL è circa 1-1.

ORM

Esempi con Python-Django

```
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name
```

```
entry = Entry.objects.get(pk=1)
cheese_blog = Blog.objects.get(name="Cheddar Talk")
entry.blog = cheese_blog
entry.save()
```

```
from blog.models import Blog
b = Blog(name="Beatles Blog", tagline="All the latest Beatles news.")
b.save()
```

```
Entry.objects.order_by("headline")[0:1].get()
```

Documentazione

Overview

Bisogna fare in modo che le nostre API siano facili da utilizzare e imparare!



OpenAPI

Overview

Spesso documentare le proprie API è fondamentale, soprattutto quando si lavora in gruppo. Lo standard OpenAPI (in formato YAML) permette una documentazione ordinata e coerente.

```
/users/{id}:
  get:
    summary: Get a user by ID
    operationId: getUserById
    parameters:
      - name: id
        in: path
        required: true
        description: The ID of the user to get
    schema:
      type: integer
  responses:
    '200':
      description: The requested user
      content:
        application/json:
          schema:
            type: object
            properties:
              id:
                type: integer
```

OpenAPI

Esempio con swagger-ui-express

```
const swaggerDocument = YAML.load('/home/kaeong/WebstormProjects/JEIOMBulletinBoardService/src/openapi.yaml');
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
```

Molte librerie permettono di convertire questo standard in pagine HTML grafiche - restituite in endpoint specifici - in cui è possibile visualizzare la documentazione in modo chiaro

OpenAPI

Risultato

The screenshot shows a web browser displaying an OpenAPI documentation page at `localhost:8080/api-docs/#/`. The page is titled "Responses" and lists a single endpoint detail.

| Code | Description | Links |
|------|-----------------|----------|
| 200 | A list of users | No links |

Below the table, there is a "Media type" dropdown set to "application/json". A note below it says "Controls Accept header." with a link to "Example Value" and "Schema".

```
[  
  {  
    "id": 0,  
    "name": "string",  
    "email": "string"  
  }  
]
```

At the bottom, two API operations are listed:

- POST /users** Create a new user
- GET /users/{id}** Get a user by ID

Postman

Andiamo a testare e documentare le nostre API!



POSTMAN

The screenshot shows the Postman application interface. On the left, there's a sidebar with sections for Collections, Environments, APIs, Mock Servers, Monitors, Flow, and History. The main area displays a collection titled "Notion's Public Workspace". Inside this collection, there's a "Databases" folder which contains a "GET Retrieve a database" request. The request details are as follows:

- Method: GET
- URL: https://api.notion.com/v1/databases/:id
- Params:
 - Auth
 - Headers (9)
 - Body
 - Pre-req.
 - Tests
 - Settings
 - Cookies
- Query params table:

| KEY | VALUE | DESCRIPTION | Bulk Edit |
|-----|-------|-------------|-----------|
| Key | Value | Description | |
- Path Variables table:

| KEY | VALUE | DESCRIPTION | Bulk Edit |
|-----|-----------------|--------------------------------|-----------|
| id | {{DATABASE_ID}} | Required. Enter database id... | |
- Body tab: Shows a JSON response with code highlighting. The response includes a "Publisher" object with an "id" of "%3E%24Pb", a "name" of "Publisher", a "type" of "select", and a "select" object with an "options" array containing one item: an object with an "id" of "c5ee409a-f307-4176-99ee-6e424fa89afa", a "name" of "NYT", and a "color" of "default".

On the right side of the interface, there's a "Documentation" panel for the "GET Retrieve a database" endpoint. It includes the URL (https://api.notion.com/v1/databases/:id), a description ("Retrieves a database object using the ID specified in the request path."), an "Authorization" section ("Bearer Token"), a "Request Header" section ("Notion-Version 22-02-22"), and a "Path Variables" section ("id").



JEMORE
TOGETHER FOR SUCCESS



Grazie

E SIA LODATA FOGGIA

www.jemore.it