https://www.snap4City.org

REST API Calls

JavaScript | cURL | Swagger | Postman | REST

Paolo Nesi, paolo.nesi@unifi.it
Luciano Alessandro Ipsaro Palesi

https://www.disit.org/

SOME RIGHTS RESERVED
BY NC ND

SISTEMI DISTRIBUITI

22/04/2026

Luciano Alessandro Ipsaro Palesi

Luciano Alessandro Ipsaro Palesi

SEGUI

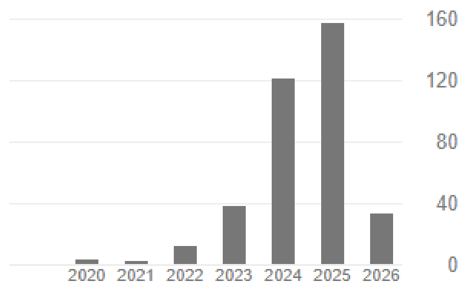
University of Florence, DINFO, DISIT Lab
Email verificata su unifi.it

Deep Learning · XAI · Intelligent Systems for Sma... · Optimization · Decision Support

TITOLO	CITATA DA	ANNO
Predicting and understanding landslide events with explainable AI E Collini, LAI Palesi, P Nesi, G Pantaleo, N Nocentini, A Rosi IEEE Access 10, 31175-31189	84	2022
Multi clustering recommendation system for fashion retail P Bellini, LAI Palesi, P Nesi, G Pantaleo Multimedia Tools and Applications 82 (7), 9989-10016	83	2023

Citata da

	Tutte	Dal 2021
Citazioni	368	365
Indice H	10	10
i10-index	10	10



Luciano Alessandro Ipsaro Palesi *lucianoalessandro.ipsaropalesi@unifi.it* *Room 231 – ex 465*
Office hours Monday 10:30 - 13:00, Via Santa Marta 3 - 50139 Firenze *Room* **231**

Deep Learning //// Explainable Artificial Intelligence (XAI)
Optimization and Simulation of Complex Systems /// Smart City, IoT/WoT e Digital Twin

Google scholar: https://scholar.google.com/citations?user=ZVSLAdgAAAAJ
Scopus: https://www.scopus.com/authid/detail.uri?authorId=57226812823
ORCID: https://orcid.org/0000-0001-8992-2084

Obiettivi e Percorso della Lezione

01

Fondamenti REST

Capire cos'è una API REST e come modellare risorse

02

Request/Response

Leggere richieste HTTP complete e comprendere le componenti

03

Metodi e Status

Scegliere metodo, URI e status code corretti per ogni operazione

04

Esempi JavaScript/Python

Implementare chiamate GET e POST nel codice

05

Strumenti Pro

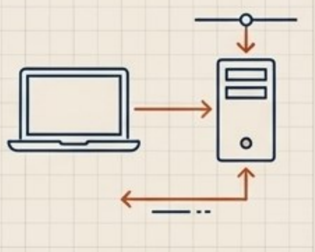
Testare API con Postman, Swagger e OpenAPI

Prerequisiti: Conoscenza di base di client/server, HTTP, JSON, URL e parametri, e lettura di JavaScript

What is a REST API?

REST	REpresentational State Transfer — an architectural style for web services.
HTTP Methods	GET · POST · PUT · PATCH · DELETE map to CRUD operations.
Stateless	Every request is self-contained; no server-side session needed.
Resources	Everything is a resource identified by a URL (e.g. /users/42).
JSON	Most APIs exchange data as JSON — lightweight and human-readable.

I vincoli fondamentali che definiscono lo stile REST

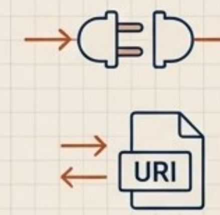


Client-Server

Separazione netta delle responsabilità. Il client gestisce l'interfaccia, il server gestisce i dati.

Stateless

Nessun contesto del client viene memorizzato sul server tra una richiesta e l'altra.



Uniform Interface

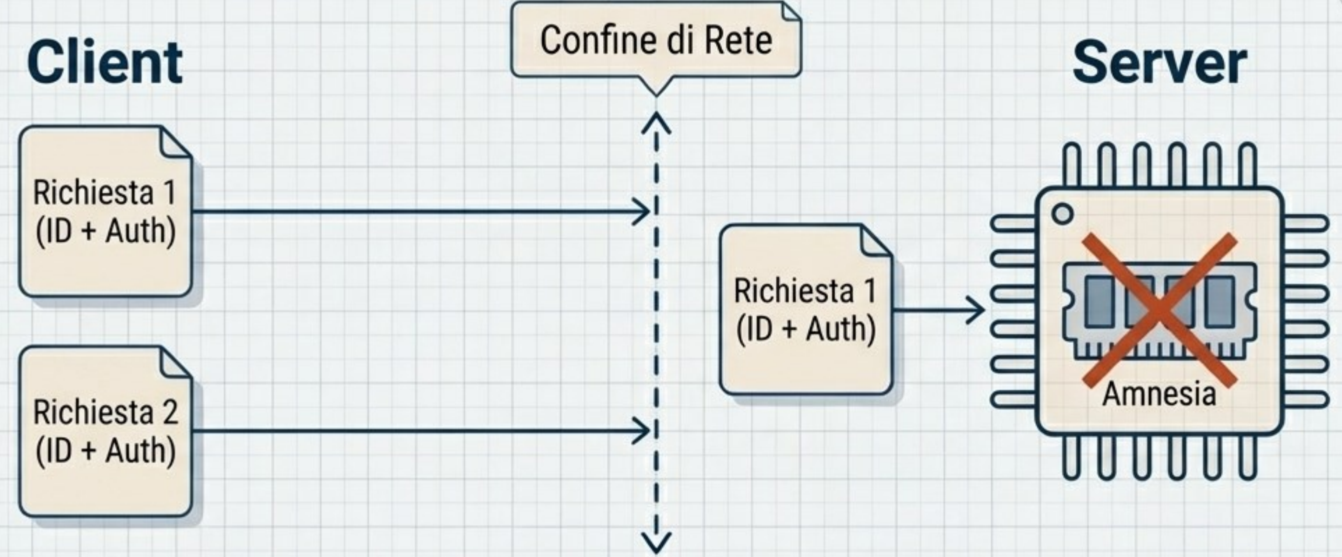
Identificazione chiara delle risorse (URI) e manipolazione tramite rappresentazioni standardizzate.

Cacheable

Le risposte devono definire esplicitamente se possono essere memorizzate nella cache per efficienza.

NotebookLM

Il Motore della "Statelessness": L'indipendenza dei pacchetti



Il server non conserva alcuna memoria delle interazioni passate. Ogni richiesta HTTP è un pacchetto isolato e autosufficiente che deve superare il confine di rete con tutto il contesto necessario per essere compreso.

NotebookLM

Anatomia di un URI: Costruire identificatori prevedibili

`https://api.store.com/v1/books/123?lang=en`

Protocollo (`https:`)
Il mezzo di trasporto sicuro.

Endpoint del Servizio (`api.store.com/v1`)
Il dominio root e il versioning.

Tipo di Risorsa (Collezione) (`books`)
Sostantivi plurali che indicano un gruppo.

ID Risorsa (`123`)
Identificatore univoco per un'entità specifica.

Query String (`lang=en`)
Parametri per filtrare o ordinare, NON per identificare.

NotebookLM

Anatomy of a REST request and response

Read a call from top to bottom: method, URL, headers, body, then status + payload.

GET | /v1/orders/123?expand=items | Headers | Body

GET usually has no body; POST and PUT often do.

REQUEST

```
GET /v1/orders/123?expand=items HTTP/1.1
Host: api.example.com
Authorization: Bearer <token>
Accept: application/json
X-Correlation-Id: 7f4a-91bd

(no request body)
```

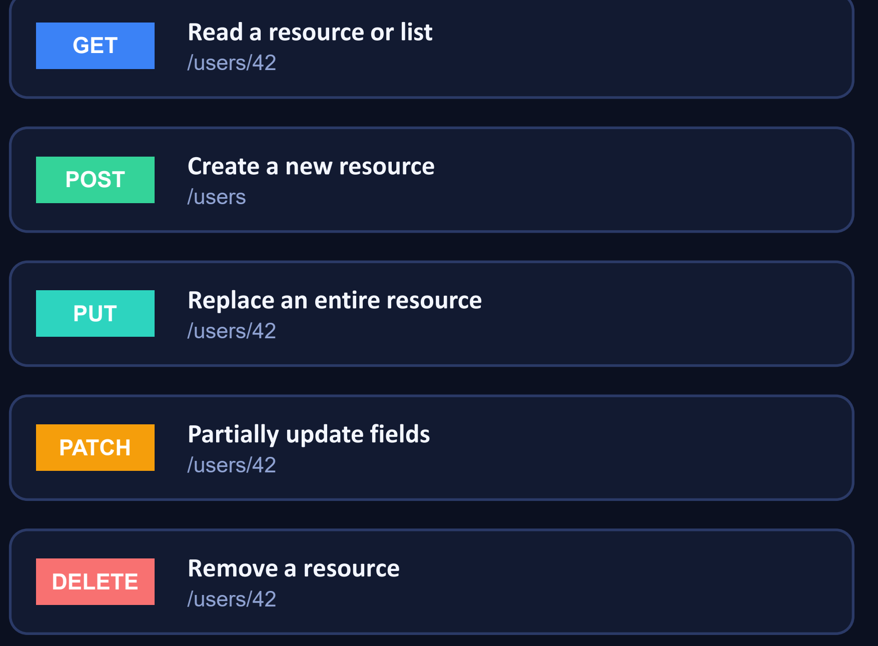
RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "id": 123,
  "status": "shipped",
  "items": 3,
  "total": 149.90
}
```

Method = intent | **URL = resource path** | **Status = outcome** | **JSON = payload**

Methods express intent; status codes express result

Use standard verbs so the API feels predictable for every caller.



200

OK — The request succeeded.

201

Created — A new resource was created.

400

Bad Request — The input failed validation.

401/403

Auth issue — The caller is missing or lacks access.

404

Not Found — The resource does not exist.

500

Server Error — The API failed internally.

Good API design lets users guess the method and status before they read the docs.

End-to-end example: GET /v1/users/42

One concrete flow makes the request/response pattern easy to explain.

1. **Client builds the call**
   Choose GET, the resource URL, and auth headers.
2. **API validates the request**
   Check the token, parse the path, and verify permissions.
3. **Service fetches the user record**
   Read the user with id 42 from storage or another service.
4. **Response returns JSON**
   Send 200 OK and the serialized user object.

REQUEST

```
GET /v1/users/42 HTTP/1.1
Host: api.example.com
Authorization: Bearer <token>
Accept: application/json
```

RESPONSE

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

{
  "id": 42,
  "name": "Ava Lee",
  "role": "admin",
  "active": true
}
```

Because the request is stateless, another server instance can answer the same call with the same contract.

La Matrice CRUD-HTTP: Mappatura delle azioni sulle risorse

	Collezione Intera (es. /clienti)	Articolo Specifico (es. /clienti/{id})
POST (Create)	Crea nuovo articolo. Ritorna 201 Created e l'intestazione Location.	Errore 404 o 409 Conflict se esiste già.
GET (Read)	Elenca risorse. Usa query per filtri. Ritorna 200 OK.	Ritorna un singolo oggetto 200 OK o 404 Not Found.
PUT (Update)	Generalmente non consentito (404).	Sostituisce l'articolo. Ritorna 200 OK o 204 No Content.
DELETE (Delete)	Cancella tutta la lista (raramente permesso).	Rimuove l'articolo. Ritorna 200 OK o 404 Not Found.

NotebookLM

Codici di Stato HTTP: Il vocabolario universale del server

2xx Successo

2xx Successo

L'azione ha funzionato.
200 OK (Completata),
201 Created (Risorsa creata).

3xx Reindirizzamento

3xx Reindirizzamento

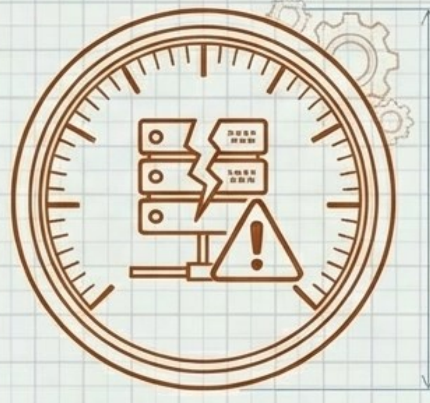
Il client deve compiere ulteriori azioni per completare la richiesta.

4xx Errore Client

4xx Errore Client

Colpa dell'utente.
400 Bad Request, 401 Unauthorized,
403 Forbidden, 404 Not Found.

5xx Errore Server



5xx Errore Server

Colpa del sistema.
500 Internal Server Error
(Anomalia del server).

NotebookLM

End-to-end example: GET /v1/users/42

One concrete flow makes the request/response pattern easy to explain.

1. **Client builds the call**
   Choose GET, the resource URL, and auth headers.
2. **API validates the request**
   Check the token, parse the path, and verify permissions.
3. **Service fetches the user record**
   Read the user with id 42 from storage or another service.
4. **Response returns JSON**
   Send 200 OK and the serialized user object.

REQUEST

```
GET /v1/users/42 HTTP/1.1
Host: api.example.com
Authorization: Bearer <token>
Accept: application/json
```

RESPONSE

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

{
  "id": 42,
  "name": "Ava Lee",
  "role": "admin",
  "active": true
}
```

Because the request is stateless, another server instance can answer the same call with the same contract.

Risorsa, Endpoint e URI: Non Sono la Stessa Cosa

Risorsa

L'oggetto del dominio che vogliamo accedere o modificare

Endpoint

Il punto tecnico di accesso all'API per quella risorsa

URI

L'identificatore univoco che localizza la risorsa

Esempi RESTful Corretti

```
/users
/users/42
/users/42/orders
/orders/123
/products/789/reviews
```

Esempi Non RESTful

- /getUsers
- /createOrder
- /deleteProductById

Principio chiave: In REST si modellano risorse, non azioni. Il verbo è nel metodo HTTP, non nell'URI.

Path param · Query param · Header

PATH PARAMETER

Identifica una risorsa specifica.

Esempio:

```
GET /users/42
```

Il valore **42** indica quale utente voglio leggere.

QUERY PARAMETER

Filtra, ordina, cerca o pagina i dati.

Esempio:

```
GET
/users?role=student&sort=name&page=1
```

role=student
→ filtra i risultati
sort=name
→ ordina i risultati
page=1
→ gestisce la paginazione

HEADER

Trasporta metadati della richiesta.

Esempi:

```
Authorization: Bearer <token>
```

```
Accept: application/json
```

```
Content-Type: application/json
```

Non identificano la risorsa: indicano come gestire la richiesta.

REGOLA PRATICA

PATH
= quale risorsa voglio

QUERY
= come voglio i dati

HEADER
= contesto e protocollo

Il path cambia la risorsa · la query cambia la vista dei dati · l'header cambia il contesto della comunicazione

Path param · Query param · Body

PATH PARAMETER

Identifica una risorsa specifica.

Esempio:

```
GET /books/10
```

Il valore **10** indica quale libro sto richiedendo.

QUERY PARAMETER

Aggiunge criteri di ricerca o visualizzazione.

Esempio:

```
GET
/books?author=Rossi&available=true&page
=2
```

Serve per:

- filtrare
- ordinare
- cercare
- paginare

BODY

Contiene i dati inviati al server.
Usato con **POST · PUT · PATCH**

Esempio POST:

```
{
  "title": "REST API Design"
  "author": "Mario Rossi"
  "available": true
}
```

REGOLA PRATICA

PATH
= quale risorsa

QUERY
= quali criteri di selezione

BODY
= quali dati sto inviando

Il path identifica · la query raffina la richiesta · il body trasporta il contenuto operativo

What is SCAPI?

Smart City API — the unified REST interface for Snap4City

IoT Data

Read and write sensor measurements, device status, and time-series values.

Services

Discover and invoke smart services registered on the Snap4City platform.

GeoData

Query POIs, routes, maps, and geographic resources with spatial filters.

Users & Auth

Manage tokens, user profiles, and role-based access control.

`Base URL: https://<your-snap4city-host>/superservicemap/api/v1/`

Authentication & Tokens

SCAPI uses OAuth2 Bearer Tokens. Obtain a token from the Snap4City dashboard or via the token endpoint.

1

Login to Snap4City

Go to your Snap4City instance and generate a Personal Access Token (PAT) from the user profile page.

2

Store the Token

Save the token securely — never expose it in client-side code or public repositories.

3

Send as Bearer Header

Include "Authorization: Bearer <token>" in every SCAPI request header.

```
const TOKEN = "eyJhbGciOiJSUz...";  // Your PAT

const headers = {
  "Authorization": "Bearer " + TOKEN,
  "Content-Type": "application/json"
};
```

Making a GET Request

Retrieve IoT device data or platform resources

```
// Fetch IoT devices near a location
async function getDevices(lat, lng, radius) {
  const BASE = "https://mysnap4city.org/superservicemap/api/v1";
  const url = `${BASE}/iot/search`
    + `?latitude=${lat}&longitude=${lng}`
    + `&radius=${radius}&format=JSON`;

  const response = await fetch(url, {
    method: "GET",
    headers: {
      "Authorization": "Bearer " + TOKEN
    }
  });

  if (!response.ok) {
    throw new Error("HTTP " + response.status);
  }

  const data = await response.json();
  return data.IoTDevices;
}

// Usage
const devices = await getDevices(43.77, 11.25, 500);
console.log(devices);
```

Key Parameters

`latitude / longitude`

WGS84 decimal coordinates

`radius`

Search radius in meters

`format`

JSON or XML

`Authorization`

Bearer token header

✓ *Always check response.ok before parsing JSON*

Making a POST Request

Push sensor values or create resources on Snap4City

```
// Send an IoT sensor observation
async function postSensorValue(deviceId, value, unit) {
  const BASE = "https://mysnap4city.org/superservicemap/api/v1";
  const url = `${BASE}/iot/${deviceId}/observation`;

  const payload = {
    timestamp: new Date().toISOString(),
    value: value,
    unit: unit
  };

  const response = await fetch(url, {
    method: "POST",
    headers: {
      "Authorization": "Bearer " + TOKEN,
      "Content-Type": "application/json"
    },
    body: JSON.stringify(payload)
  });

  if (!response.ok) {
    const err = await response.json();
    throw new Error(err.message);
  }
  return await response.json();
}

// Usage
await postSensorValue("sensor-abc-001", 23.5, "Celsius");
```

POST Checklist

- ✓ Set Content-Type: application/json
- ✓ Stringify body with JSON.stringify()
- ✓ Include ISO 8601 timestamps
- ✓ Handle non-2xx status codes
- ✓ Parse error response body
- ✓ Validate deviceId exists in platform

Asynchronous REST Calls

Non-Blocking

Why use async?

- Other code runs while waiting for the response.
- Avoids freezing the browser or server.
- Use Promises or async/await syntax.
- Ideal for multiple parallel requests.

Parallel Requests with Promise.all()

```
const [user, posts] = await Promise.all([
  fetch('/users/42'),
  fetch('/posts?userId=42')
]);
```

JavaScript — Promise chain

```
// Non-blocking Promise chain
fetch('https://api.example.com/users/42')
  .then(res => res.json())
  .then(data => {
    console.log(data);
  })
  .catch(err => {
    console.error('Error:', err);
  });

// Continues running while request is in-flight
```

SYNC — blocks until done — simple, predictable

ASYNC — non-blocking — scalable, parallel

Handling Responses & Errors

SCAPI returns standard HTTP status codes and JSON envelopes

Status Code	Meaning	Action
200 OK	Request succeeded	Parse JSON response body
201 Created	Resource created	Read Location header or body
400 Bad Request	Invalid parameters	Check your query/body params
401 Unauthorized	Missing/expired token	Refresh or re-authenticate
403 Forbidden	Insufficient permissions	Check user role and scopes
404 Not Found	Resource not found	Verify device/service ID
500 Server Error	Platform-side error	Retry with exponential backoff

```
// Robust error handling wrapper
async function scapiGet(endpoint) {
  try {
    const r = await fetch(BASE + endpoint, { headers });
    if (r.status === 401) throw new Error("Token expired — refresh needed");
    if (!r.ok) {
      const body = await r.json().catch(() => ({}));
      throw new Error(`SCAPI ${r.status}: ${body.message || r.statusText}`);
    }
    return await r.json();
  } catch (err) {
    console.error("[SCAPI Error]", err.message);
    throw err;
  }
}
```

Dominare i Dati: Paginazione, Filtri e Sorting

```
GET /items?offset=100&limit=20&sort=name:asc&filter[category]=organic
```

Paginazione

Suddivide set di dati enormi. Offset indica il punto di partenza, limit quanti record recuperare.

Sorting

Ordina i risultati. Supporta direzioni multiple, es: sort=name:asc,description:desc.

Filtering

Filtra risorse specifiche o utilizza operatori avanzati (es: gt:5 per maggiore di 5, neq per non uguale).

Pro-Tip: In alternativa a offset/limit, usa token di continuazione (es. timestamp ?since={TIMESTAMP}) per performance migliori su enormi database.

NotebookLM

Evoluzione Senza Interruzioni: Strategie di Versioning

Pianifica sempre il versioning fin dal giorno zero.

Strategia	Esempio	Analisi
RECOMMENDED URL Path	`https://api.domain.com/v1/orders`	Facile da condividere, testare nel browser e leggere nei log. La strategia più adottata e consigliata.
Subdomain	`https://v1.api.domain.com/orders`	Utile se le versioni dell'API sono ospitate su cluster di server fisicamente diversi, ma **più complesso** per la gestione dei certificati SSL.
Header / Content Negotiation	`Accept: application/json; version=1`	Mantiene l'URL pulito e semanticamente corretto per il REST purista, ma è difficile da testare senza strumenti come Postman o cURL.

NotebookLM

Good habits for designing and calling REST APIs

These practices make APIs easier to build, consume, and operate.

Use nouns in URLs
Prefer /users/42 over /getUser?id=42.

Choose the right HTTP verb
GET reads, POST creates, PUT replaces, PATCH updates, DELETE removes.

Return helpful status codes
Clear outcomes speed up debugging and client-side handling.

Design for scale
Support filtering, sorting, pagination, and idempotent retries.

Version and document the API
Stable contracts reduce breaking changes for consumers.

Secure, observable APIs are easier to trust in production.

Auth | Validation | Paging | Version | ...

cURL: Command Line REST Calls

GET

```
curl -X GET "https://api.example.com/users" \
  -H "Authorization: Bearer TOKEN"
```

POST

```
curl -X POST "https://api.example.com/users" \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice","email":"a@b.com"}'
```

PUT

```
curl -X PUT "https://api.example.com/users/42" \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice Updated"}'
```

DELETE

```
curl -X DELETE "https://api.example.com/users/42" \
  -H "Authorization: Bearer TOKEN"
```

cURL: Useful Flags & Options

`-X`	Specifies the HTTP method (GET, POST, PUT, DELETE…)
`-H`	Adds a request header (e.g. Authorization, Content-Type)
`-d`	Request body data (use with POST/PUT)
`-o file`	Save response to a file instead of printing
`-i`	Include response headers in output
`-v`	Verbose mode — shows full request & response details
`--insecure`	Skip SSL certificate verification (dev only!)

Swagger / OpenAPI

What is Swagger?

- Swagger = toolset built around the OpenAPI Specification.
- Swagger UI: auto-generated interactive docs for any REST API.
- swagger.yaml / swagger.json describe all endpoints, params, and responses.
- Try endpoints directly from the browser — no extra tools needed.
- Team standard: backend devs generate it; frontend devs consume it.

OpenAPI 3.0 snippet

```
openapi: '3.0.0'
info:
  title:
User API
  version: '1.0.0'

paths:
  /users/{id}:
    get:
      summary: Get user by ID
      parameters:
      - name:
id
        in:
path
        required: true
      responses:
        '200':
          description:
OK
```

Postman: Visual API Testing

Collections

Group related requests together and share with your team.

Environments

Switch between dev / staging / prod variables instantly.

Pre-request Scripts

Run JS before a request (e.g. generate a token).

Tests

Write JS assertions to validate response body, status, time.

Mock Servers

Fake API responses before backend is ready.

Newman CLI

Run Postman collections from CI/CD pipelines.

Tip: Import a Swagger/OpenAPI file directly into Postman to auto-generate all endpoint requests!

Swagger vs Postman — When to Use What

Swagger / OpenAPI

- API documentation (always up-to-date)
- Explore endpoints without any tool
- Generate client SDKs in any language
- Define API contract for frontend & backend
- Spec-first API design

VS

Postman

- Manual & automated API testing
- Manage environments (dev / prod)
- Team collaboration on test collections
- CI/CD integration via Newman
- Mock API responses during development

Use both together: Swagger defines the API, Postman tests and automates it.

SCAPI REST Calls

with Node-RED

Snap4City Platform — Visual Flow Tutorial

Node-RED | Visual Flows | IoT / SCAPI

snap4city.org

Flow-Based Programming

inject

http

debug

Why Node-RED for SCAPI?

Node-RED turns REST API calls into drag-and-drop visual flows — no boilerplate code required.

Visual Wiring

Connect HTTP nodes, function nodes, and outputs with wires — see data flow at a glance.

Built-in HTTP

The http request node handles GET/POST to SCAPI directly, with configurable headers and auth.

Easy Debugging

Debug nodes let you inspect msg.payload at every step of the flow in real time.

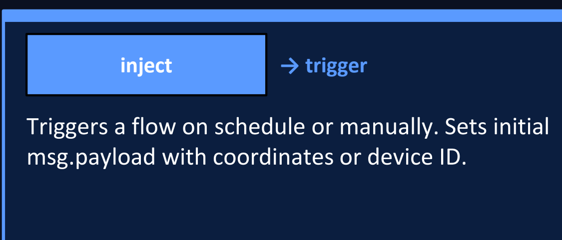
Snap4City Nodes

Dedicated snap4city Node-RED nodes wrap SCAPI calls for even simpler integration.

`Node-RED runs on Node.js — install via:  npm install -g --unsafe-perm node-red`

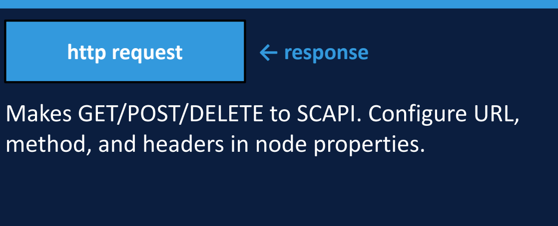
Core Node-RED Nodes for SCAPI

These six nodes cover virtually every SCAPI integration pattern



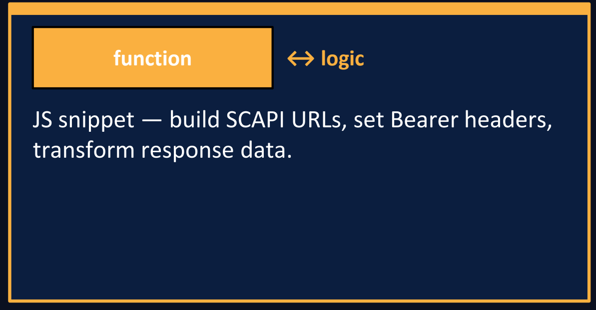
inject → trigger

Triggers a flow on schedule or manually. Sets initial msg.payload with coordinates or device ID.



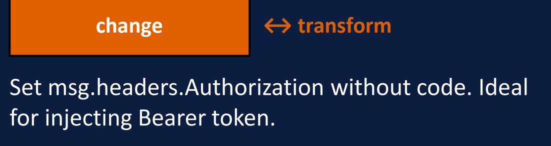
http request ← response

Makes GET/POST/DELETE to SCAPI. Configure URL, method, and headers in node properties.



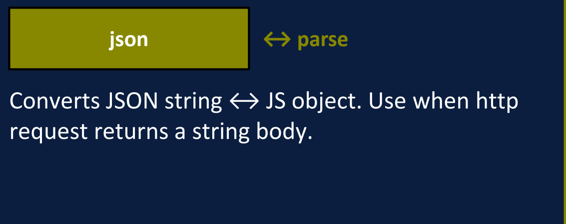
function ↔ logic

JS snippet — build SCAPI URLs, set Bearer headers, transform response data.



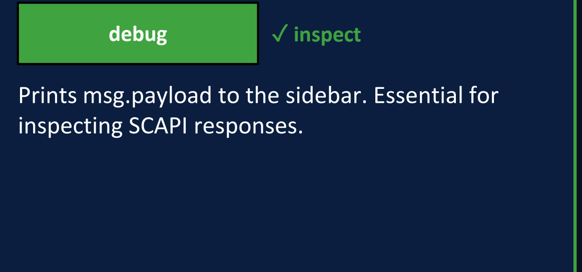
change ↔ transform

Set msg.headers.Authorization without code. Ideal for injecting Bearer token.



json ↔ parse

Converts JSON string ↔ JS object. Use when http request returns a string body.

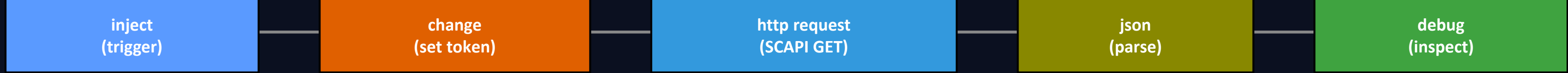


debug ✓ inspect

Prints msg.payload to the sidebar. Essential for inspecting SCAPI responses.

Authentication — Bearer Token in Node-RED

Use a change node to inject your token into msg.headers before the http request node



change node — Set Rules

```
Set  msg.headers  to  {}
Set  msg.headers.Authorization
     to  "Bearer eyJhbGciOiJSUz..."
```

http request node — Config

Method: GET

URL: {{msg.url}} (set upstream)

Return: a parsed JSON object

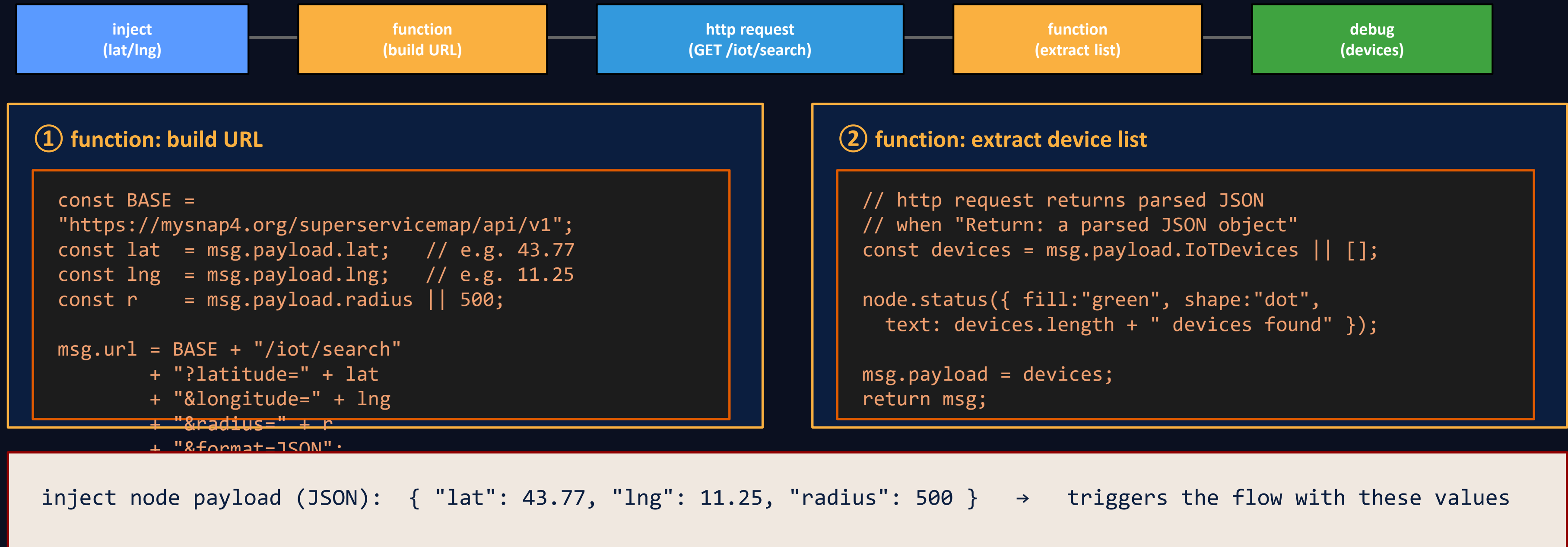
Use auth headers: ✓ from msg.headers

⚠ Never paste your token directly into a flow that is committed to source control — store it in Node-RED's global context or an environment variable: process.env.SCAPI_TOKEN

```
// In a function node — read token from environment
const token = process.env.SCAPI_TOKEN || global.get("scapi_token");
msg.headers = { "Authorization": "Bearer " + token };
return msg;
```

GET Request — Search IoT Devices

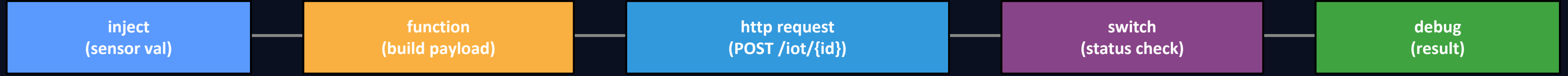
Search for nearby sensors by location using a visual flow



node.status() updates the node badge in the editor — great for live feedback

POST Request — Push Sensor Observation

Send a new sensor reading to Snap4City via a POST flow



function: build POST payload

```
const BASE  = "https://mysnap4.org/superservicemap/api/v1";
const TOKEN = process.env.SCAPI_TOKEN;
const devId = "sensor-abc-001";

msg.url    = BASE + "/iot/" + devId + "/observation";
msg.method = "POST";
msg.headers = {
  "Authorization": "Bearer " + TOKEN,
  "Content-Type":  "application/json"
};
msg.payload = JSON.stringify({
  timestamp: new Date().toISOString(),
  value:     msg.payload.value,
  unit:      msg.payload.unit || "Celsius"
});
return msg;
```

switch + error handling

```
// switch node rules:
// msg.statusCode == 200 or 201 → output 1 (success)
// otherwise                   → output 2 (error)

// Connect output 2 to a function:
if (msg.statusCode >= 400) {
  node.error("SCAPI error " + msg.statusCode,
             msg);
  msg.payload = "POST failed: " + msg.statusCode;
}
return msg;
```

http request node must have Method set to "use msg.method" and Payload to "send as request body"

`inject payload:  { "value": 23.5, "unit": "Celsius" }   — send on interval for continuous telemetry`

Handling Responses & Errors

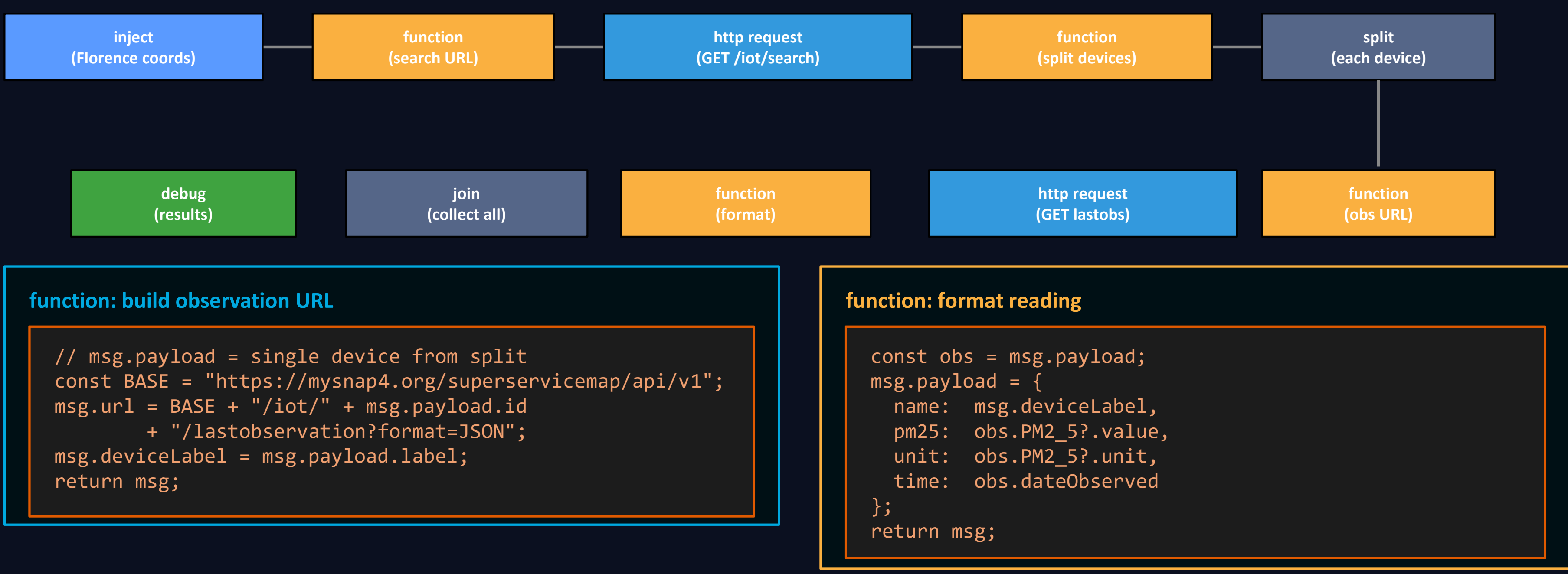
Node-RED exposes msg.statusCode from every http request — use a switch node to branch on it

statusCode	Meaning	Node-RED Action
200 / 201	Success	Pass msg.payload downstream
400	Bad request	Log with node.error(); fix URL/params
401	Unauthorized	Refresh token; re-set msg.headers
403	Forbidden	Check user role in Snap4City dashboard
404	Not found	Verify device/service ID in msg.url
429	Rate limit	Add a delay node (e.g. 1 msg/sec)
5xx	Server error	Retry with exponential backoff

```
// catch node (or function after switch) — generic error handler
msg.originalTopic = msg.topic;
node.warn("SCAPI call failed [" + msg.statusCode + "] " + msg.url);

// Optionally store error for retry:
flow.set("last_failed_url", msg.url);
msg.payload = { error: msg.statusCode, url: msg.url };
return msg;
```

Full Example — Air Quality Monitor Flow

Query nearby PM2.5 sensors and log their latest readings — mirrors the JavaScript example



split sends one msg per device → join collects all results after all http requests complete → single debug output

Best Practices in Node-RED

Environment Variables

Use process.env.SCAPI_TOKEN — set it in your Node-RED startup script or Settings.js. Never paste tokens in flow JSON.

Global Context

Store base URL and token once in global context (flow.set / global.set). Reference from every function node.

Rate Limiting

Add a delay node between the split and the http request node. Set it to rate-limit mode: e.g. 1 message per second.

Catch Nodes

Place a catch node on the same tab — it captures all unhandled errors from http request nodes automatically.

Subflows

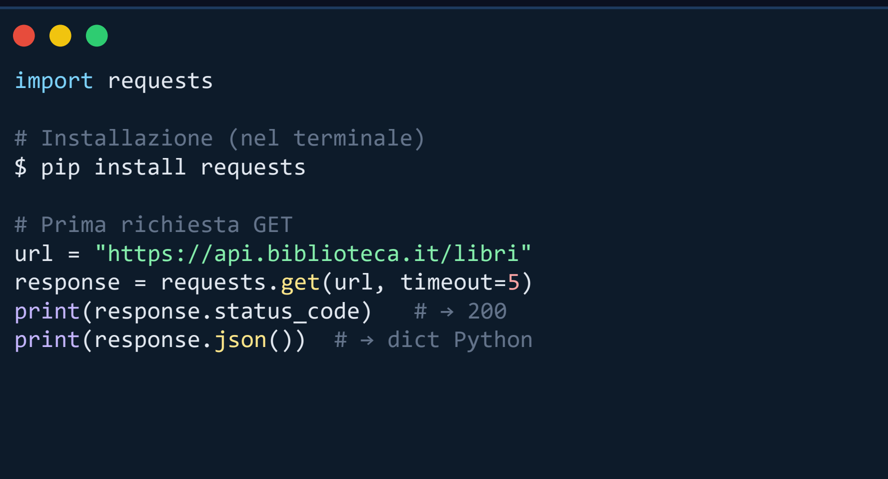
Wrap your authentication + http request pattern into a subflow. Reuse it across all SCAPI call flows.

Status Feedback

Call node.status() with color & text in every function to show live state: green=OK, red=error, yellow=waiting.

API REST & Python

Usare API REST in Python



GET · POST · PUT/PATCH · DELETE
Autenticazione · Error Handling · FastAPI

requests | json() | Bearer Token | FastAPI

Cosa serve

- Python 3.8+
- libreria requests
- Endpoint URL
- Header & JSON

Cosa faremo

- GET, POST, PUT, PATCH, DELETE
- Autenticazione Bearer Token
- Error Handling robusto
- Creare API con FastAPI

GET in Python: leggere dati da una API

requests.get() — Operazione sicura e idempotente

Recupera risorse dal server senza modificarle.
I parametri viaggiano nella query string (?chiave=valore).

`params=`

Costruisce la query string automaticamente dal dict

`headers=`

Accept, Authorization, Content-Type…

`timeout=5`

Massimo 5 secondi di attesa

`response.json()`

Converte il body JSON in un dict Python

```python
import requests

url = "https://api.biblioteca.it/libri"

response = requests.get(
    url,
    params = {
        "autore": "Calvino",
        "anno":   1972
    },
    headers = {
        "Accept": "application/json"
    },
    timeout = 5
)

if response.status_code == 200:
    libri = response.json()
    print(libri)
else:
    print("Errore", response.status_code)
```

💡 status_code 200 = OK | response.json() restituisce un dict Python

POST in Python: inviare dati e autenticarsi

`requests.post(url, json=payload, headers={...})`

Crea una nuova risorsa sul server.
Richiede autenticazione tramite Bearer Token.

Bearer Token

Header: Authorization: "Bearer " + token
Identifica e autentica il chiamante.

json = payload

Imposta Content-Type: application/json
automaticamente — nessun encoding manuale.

201 Created

Status code atteso per creazione risorsa.
La risposta include la risorsa appena creata.

```python
token = "eyJhbGciOiJSUzI1NiJ9..."

response = requests.post(
    "https://api.biblioteca.it/libri",
    headers = {
        "Authorization": "Bearer " + token,
        "Content-Type": "application/json"
    },
    json = {
        "titolo": "Il barone rampante",
        "autore": "Calvino",
        "anno":  1957
    }
)

# Status code atteso: 201 Created
print(response.status_code)   # → 201
print(response.json())
```

Aggiornare ed eliminare risorse in Python

PUT sostituisce l'intera risorsa · **PATCH** aggiorna solo i campi specificati · **DELETE** elimina la risorsa (nessun body)

PUT

```python
# Sostituzione completa
requests.put(
  ".../libri/42",
  headers = headers,
  json = {
    "titolo": "Titolo nuovo",
    "autore": "Calvino",
    "anno": 1957
  }
)

# 200 OK o 204 No Content
```

PATCH

```python
# Aggiornamento parziale
requests.patch(
  ".../libri/42",
  headers = headers,
  json = {
    "anno": 1958
  }
)

# Solo il campo 'anno'
# viene modificato
# sul server
# → 200 OK
```

DELETE

```python
# Eliminazione risorsa
response = requests.delete(
  ".../libri/42",
  headers = headers
)

# 204 No Content
# Risorsa eliminata

# Nessun body
# nella risposta
print(response.status_code)
```

Gestione corretta di errori e timeout

`timeout = 5`

Limita l'attesa a 5 s; evita blocchi infiniti

`raise_for_status()`

Lancia HTTPError se status code ≥ 400

`Timeout`

Il server non ha risposto entro il limite

`HTTPError`

Status 4xx (client) o 5xx (server)

`RequestException`

Errori di rete generici (DNS, connessione...)

Un client robusto gestisce i fallimenti con grazia, non solo i successi.

```python
import requests

try:
    response = requests.get(
        url, timeout=5
    )
    response.raise_for_status()
    data = response.json()

except requests.exceptions.Timeout:
    print("Timeout: server non risponde")

except requests.exceptions.HTTPError as e:
    print("Errore HTTP", e.response.status_code)

except requests.exceptions.RequestException as e:
    print("Errore rete:", e)

else:
    print("Dati OK:", data)
```

Query parameter e paginazione in Python

I query parameter filtrano, ordinano e paginano i risultati.

requests costruisce automaticamente la query string dal dizionario params= — non occorre concatenare manualmente le stringhe.

page

Numero pagina corrente

pageSize

Risultati per pagina

sort

Campo di ordinamento

available

Filtro booleano (True/False)

```
params = {
    "page"      : 1,        # pagina corrente
    "pageSize"  : 10,       # 10 risultati per pagina
    "sort"      : "titolo",   # ordina per titolo
    "available": True         # solo libri disponibili
}

response = requests.get("https://api.biblioteca.it/libri", params=params)

# URL costruito: /libri?page=1&pageSize=10&sort=titolo&available=True
print(response.url)   # verifica l'URL completo con query string
```

Scrivere una funzione Python riutilizzabile per le API

Centralizza headers, token e timeout in un'unica funzione.

Ogni chiamata API diventa una sola riga di codice pulita.

Riuso

Una sola funzione per tutti i metodi HTTP

Leggibilità

Il codice client è più pulito e comprensibile

Manutenzione

Cambio timeout o header in un solo posto

Testabilità

Funzione pura, semplice da mockare nei test

```python
import requests

def call_api(method, url, token=None,
             params=None, payload=None):

    headers = {"Content-Type": "application/json"}
    if token:
        headers["Authorization"] = "Bearer " + token

    response = requests.request(
        method  = method,
        url     = url,
        headers = headers,
        params  = params,
        json    = payload,
        timeout = 10
    )
    response.raise_for_status()
    return response.json()

# Utilizzo
libri = call_api("GET", url, params={"page": 1})
```

Creare una semplice API in Python con FastAPI

FastAPI — framework moderno, veloce, con documentazione interattiva automatica.

```
$ pip install fastapi uvicorn
```

```
$ uvicorn main:app --reload
```

GET /books

Restituisce la lista di tutti i libri (200 OK)

POST /books

Aggiunge un libro dal body JSON (201 Created)

Docs automatici

FastAPI genera /docs (Swagger) e /redoc gratis

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()
books = []

class Book(BaseModel):
    titolo : str
    autore : str
    anno   : int

# GET  /books → 200 OK
@app.get("/books")
def get_books():
    return books

# POST /books → 201 Created
@app.post("/books", status_code=201)
def add_book(book: Book):
    books.append(book)
    return book
```

Python può sia consumare che creare API REST.

Usare API REST in JavaScript

fetch() · async/await · REST verbs · Error handling

GET | POST | PUT/PATCH | DELETE

Stessa logica REST, linguaggio diverso.

GET in JavaScript: leggere dati da una API

GET

Passaggi chiave

1. **`fetch(url)`**
   Invia la richiesta HTTP GET
2. **`header Accept`**
   Dichiara il formato atteso
3. **`response.ok`**
   Controlla lo status HTTP
4. **`response.json()`**
   Deserializza il corpo JSON
5. **`.catch()`**
   Cattura errori di rete

script.js

```
// GET: recupera la lista dei libri
fetch('https://api.example.com/books', {
  method: 'GET',
  headers: { 'Accept': 'application/json' }
})
  .then(response => {
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    return response.json();
  })
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

GET in JavaScript con async/await

async/await

script.js

```
async function getBooks() {
  try {
    const response = await fetch('https://api.example.com/books');

    if (!response.ok) throw new Error(`Errore: ${response.status}`);

    const data = await response.json();
    console.log(data);

  } catch (err) {
    console.error('Errore di rete:', err);
  }
}

getBooks();
```

`async function`

Dichiara una funzione asincrona che restituisce implicitamente una Promise

`await fetch()`

Sospende l'esecuzione fino alla risposta del server — nessun callback nidificato

`await response.json()`

Deserializza il corpo della risposta: secondo await, seconda Promise

`try / catch`

Cattura sia errori di rete (fetch fallisce) sia errori HTTP gestiti manualmente

async/await rende il codice asincrono leggibile come codice sincrono — ideale per più chiamate sequenziali

POST in JavaScript: inviare dati e autenticarsi

POST

post-book.js

```
const token = 'eyJhbGciOiJIUzI1NiIsInR...';

const payload = { title: 'Il Signore degli Anelli', year: 1954 };

const response = await fetch('https://api.example.com/books', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': `Bearer ${token}`
  },
  body: JSON.stringify(payload)
});

// 201 Created → risorsa creata con successo
if (response.status === 201) console.log('Libro creato!');
```

JSON.stringify(payload)

Converte l'oggetto JS in stringa JSON prima di inviarlo nel body

Authorization: Bearer

Autenticazione con token JWT — inserito nell'header, non nel body

Content-Type: application/json

Informa il server che il body è in formato JSON

201 Created

Status corretto per la creazione: diverso da 200 OK usato per GET

Aggiornare ed eliminare risorse in JavaScript

PUT | PATCH | DELETE

PUT /books/:id

```javascript
// PUT: sostituzione completa
await fetch('/books/42', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ title: 'Nuovo Titolo', year: 2024 })
});
```

PATCH /books/:id

```javascript
// PATCH: aggiornamento parziale
await fetch('/books/42', {
  method: 'PATCH',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ year: 2024 })
});
```

DELETE /books/:id

```javascript
// DELETE: rimuovi la risorsa
await fetch('/books/42', {
  method: 'DELETE'
});
```

PUT vs PATCH — Differenza pratica

	PUT	PATCH
Body richiesto	Completo	Solo i campi modificati
Campi mancanti	Azzerati/null	Invariati
Uso tipico	Sostituzione totale	Aggiornamento parziale

Gestire errori e status code con fetch

error-handling.js

```
async function safeGet(url) {
  try {
    const res = await fetch(url);

    // fetch NON lancia errore su 404/500!
    if (res.status === 404) {
      throw new Error('Risorsa non trovata');
    }

    if (!res.ok) {
      throw new Error(`Errore server: ${res.status}`);
    }

    return await res.json();

  } catch (err) {
    console.error(err.message);
  }
}
```

STATUS CODE HTTP

Codice	Nome	Descrizione
200	**OK**	Richiesta riuscita
201	**Created**	Risorsa creata (POST)
204	**No Content**	Successo, nessun body
400	**Bad Request**	Dati non validi
401	**Unauthorized**	Autenticazione mancante
403	**Forbidden**	Accesso negato
404	**Not Found**	Risorsa inesistente
500	**Server Error**	Errore lato server

⚠ **fetch() NON lancia eccezione su 404 o 500 — controlla sempre response.ok o response.status manualmente**

Query parameter e paginazione in JavaScript

query-params.js

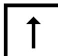
```javascript
// URLSearchParams: costruisce la query string
const params = new URLSearchParams({
  genre: 'fantasy',      // filtro
  sort: 'year',        // ordinamento
  order: 'desc',
  page: 1,          // paginazione
  limit: 10,         // elementi per pagina
  q: 'tolkien'         // ricerca testuale
});

const url = `https://api.example.com/books?${params}`;

// URL risultante:
// /books?genre=fantasy&sort=year&order=desc&page=1&limit=10&q=tolkien

const res = await fetch(url);
```

Ricerca

`q=tolkien`
Cerca per keyword

Filtro

`genre=fantasy`
Filtra per categoria



Ordinamento

`sort=year&order=desc`
Ordinamento colonna

Paginazione

`page=1&limit=10`
Carica a pagine

Scrivere una funzione riutilizzabile per chiamare API

api-client.js

```
async function callApi(url, method = 'GET', token = null, payload = null) {
  const headers = { 'Content-Type': 'application/json' };
  if (token) headers['Authorization'] = `Bearer ${token}`;

  const options = { method, headers };
  if (payload) options.body = JSON.stringify(payload);

  const res = await fetch(url, options);
  if (!res.ok) throw new Error(`HTTP ${res.status}`);
  return await res.json();
}

// Utilizzo:
const books = await callApi('/books');
await callApi('/books', 'POST', token, { title: 'Dune' });
```

Vantaggi del pattern

Riuso

Una funzione per GET, POST, PUT, DELETE — nessun codice duplicato

Manutenzione

Cambia un'API URL o header in un solo punto

Leggibilità

Il codice chiamante è pulito e espressivo

Testabilità

Facile sostituire callApi con un mock nei test

Creare una semplice API in JavaScript con Express

Express.js | Node.js

server.js

```
const express = require('express');
const app = express();

app.use(express.json());  // parse body JSON

let books = [
  { id: 1, title: 'Dune' },
  { id: 2, title: 'Neuromancer' }
];

// GET /books → restituisce tutti i libri
app.get('/books', (req, res) => {
  res.json(books);
});

// POST /books → aggiunge un nuovo libro
app.post('/books', (req, res) => {
  const newBook = { id: books.length + 1, ...req.body };
  books.push(newBook);
  res.status(201).json(newBook);
});

app.listen(3000, () => console.log('API attiva su http://localhost:3000'));
```

Concetto chiave

JavaScript può sia consumare che creare API REST — lo stesso linguaggio sui due lati della connessione.

1. **`express()`** — Crea l'istanza dell'app
2. **`express.json()`** — Middleware per body JSON
3. **`app.get/post()`** — Definisce gli endpoint REST
4. **`req.body`** — Legge il payload inviato
5. **`res.status(201)`** — Risponde con il corretto status
6. **`app.listen(3000)`** — Avvia il server HTTP

Mini laboratorio: esplora una API pubblica Snap4City

Snap4City rende disponibili API esterne documentate in Swagger; per i dati pubblici non è necessaria autenticazione. Scegli una chiamata pubblica e analizzala.

Attività

individua una GET pubblica nello Swagger
lancia la richiesta
osserva status code e body
riconosci:

- risorsa
- eventuali query parameter
- formato della risposta
- significato dei dati

Suggerimento

Puoi partire dalle API legate al **public transport**, perché Snap4City pubblica esempi di test proprio su agenzie, linee, route, fermate e posizione dei mezzi.

https://www.snap4city.org/drupal/node/70
https://www.km4city.org/swagger/external/index.html