Cheat-Sheet REST, HTTP e API Design

1. Fondamenti di REST

REST è uno **stile architetturale** basato su pochi principi:

1. **Client–Server**
   Separazione tra chi fornisce dati (server) e chi li usa (client).
2. **Statelessness**
   Ogni richiesta contiene tutte le informazioni necessarie.
   Nessuna "sessione" mantenuta dal server.
3. **Cache**
   Le risposte possono essere memorizzate per migliorare le prestazioni.
4. **Uniform Interface**
   Interfaccia coerente: metodi HTTP standard, URI chiari, messaggi autodescrittivi.
5. **Layered System**
   Il client non deve conoscere quante componenti esistono "nel mezzo" (proxy, gateway, CDN).
6. **Code-on-Demand** (opzionale)
   Il server può fornire codice eseguibile al client.

---

2. Risorse e URI

Principi

- Una *risorsa* rappresenta un'entità: prodotto, cliente, ordine, documento.
- Ogni risorsa ha un **URI unico e stabile**.
- Gli URI devono essere **semplici, leggibili e prevedibili**.

Pattern

- Collezioni: `/products`
- Oggetto singolo: `/products/{id}`
- Relazioni: `/products/{id}/reviews`

Buone pratiche

- usare nomi **al plurale**
- evitare verbi (i verbi sono nei metodi HTTP)
- non inserire versioni nell'URI se non strettamente necessario
- evitare strutture profondissime (massimo 2–3 livelli)

---

3. Metodi HTTP (semantica)

Metodo	Significato	Idempotenza	Note
GET	legge una risorsa	✔	non modifica nulla
POST	crea una nuova risorsa	✘	produce effetti
PUT	sostituisce completamente	✔	deve essere deterministico
PATCH	aggiorna parzialmente	✘	usato per modifiche parziali
DELETE	elimina una risorsa	✔	ripetere la richiesta non cambia l'esito

---

4. Principali status code

- **200 OK** → richiesta corretta
- **201 Created** → risorsa creata
- **204 No Content** → ok, senza corpo
- **400 Bad Request** → parametri errati
- **401 Unauthorized** → autenticazione mancante
- **403 Forbidden** → autenticazione ok, ma accesso negato
- **404 Not Found** → risorsa inesistente
- **409 Conflict** → conflitto di stato
- **500 Internal Server Error** → errore interno

---

5. Filtering, Sorting, Pagination

Filtering

- usato per restringere i risultati
  Esempi:
  `/products?category=food`
  `/customers?city=Milano`

Sorting

- ordinamento dei risultati
  `/products?sort=price,asc`

Pagination

- necessario per liste grandi
  `/products?page=0&size=20`

Buone pratiche:

- documentare tutti i parametri
- mantenere naming coerente
- usare default sensati (`size=20`)

---

6. HATEOAS (concetto)

Una risorsa può contenere link che indicano le azioni successive.

Esempio concettuale:

```
{
  "id": 12,
  "name": "Lombardia",
  "_links": {
    "self": "/regions/12",
    "provinces": "/regions/12/provinces"
  }
}
```

Serve per rendere l'API **navigabile** come un sito web.

7. Design-first

Caratteristiche

- si progetta l'API **prima** di scrivere codice
- si valida il design con i team consumer
- la documentazione nasce immediatamente
- si possono generare mock server
- riduce errori e refactoring

Code-first (per confronto)

- il codice genera l'API
- la documentazione viene dopo
- maggiore rischio di incoerenza

---

8. OpenAPI (panoramica)

Una specifica OpenAPI descrive:

- titolo, versione
- endpoint
- metodi
- parametri
- modelli e tipi dei dati
- esempi
- codici di risposta
- sicurezza

È uno **standard universale** per descrivere API.

---

9. Error design (modello errori)

Buona API = errori chiari, coerenti e documentati.

Buone pratiche

- indicare `code` (numerico)
- indicare `message` (chiaro)
- non esporre stack trace
- usare gli status code corretti
- mantenere forma consistente

Esempio:

```
{
  "error": "Customer not found",
  "code": 404
}
```

---

10. Developer Experience (DX)

Una buona API dovrebbe essere:

- **prevedibile**
- **consistente**
- **ben documentata**
- **dotata di esempi**
- **con errori chiari**
- **facile da testare**

La DX è oggi un elemento critico per strumenti, app e integrazioni.