

Introduzione

Obiettivi:

- ripassare i fondamenti REST
- capire come si modellano risorse e URI
- introdurre concetti avanzati (filtering, pagination, strumenti)
- imparare cosa significa *design-first*
- chiudere con un esercizio guidato su OpenAPI

Questo modulo collega teoria, pratica e strumenti reali.

Cos'è REST davvero

Cos'è REST davvero

REST non è:

- JSON
- URL “carini”
- CRUD applicato a caso

REST è uno **stile architetturale** basato su 6 vincoli:

1. Client–Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. (opzionale) Code-on-demand

Lo scopo: un web di risorse semplici da esplorare e integrare.

Risorse e Identificatori

Risorse e Identificatori

Le API REST si basano su **risorse**:

- regioni
- prodotti
- clienti
- ordini

Ogni risorsa ha un **identificatore univoco** (URI):

- `/regions/12`
- `/orders/987`

Principio chiave:

"Se la cosa esiste e si può recuperare, allora è una risorsa."

Rappresentazioni delle risorse

Rappresentazioni delle risorse

Una risorsa può avere più **rappresentazioni**:

- JSON
- XML
- CSV

REST non impone il formato: è una **negoziazione** tra client e server.

L'importante è che il formato sia:

- chiaro
- stabile
- ben documentato

 *Example: Esempio JSON*

```
{  
  "id": 12,  
  "name": "Lombardia",  
  "population": 10000000  
}
```

Statelessness

Statelessness

Un server REST **non mantiene stato della sessione**.

Ogni richiesta contiene tutto ciò che serve a essere processata.

Esempio intuitivo:

- ❌ "Mi ricordi chi sono?" → server mantiene sessione
- ✅ "Ecco il mio token/API-Key" → il client invia i dati sempre

Benefici:

- scalabilità
- semplicità
- fault-tolerance

Uniform Interface e HATEOAS

Uniform Interface e HATEOAS (lite)

Uniform Interface = regole coerenti per tutte le risorse.

HATEOAS = il client può scoprire cosa fare tramite link.

Concetto semplice:

- è come navigare un sito web
- ogni pagina (risorsa) contiene link verso altre operazioni

Nelle API REST:

La risposta può includere link a operazioni correlate.

Example: Risposta HATEOAS semplice

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

Metodi HTTP (panoramica)

REST sfrutta i metodi HTTP per indicare l'operazione sui dati:

- **GET** → leggo una risorsa
- **POST** → creo una risorsa
- **PUT** → aggiorno tutto
do
- **PATCH** → aggiorno una parte
- **DELETE** → cancello

Coerenza e prevedibilità rendono l'API facile da usare.

GET — recuperare informazioni

GET

Usato per:

- ottenere una risorsa
- ottenere una lista

Non modifica mai lo stato.

È **idempotente**: fare 1 o 50 GET produce lo stesso effetto.

 *Example: Esempio GET*

```
GET /products/10
```

POST — creare

POST

Usato per **creare** una nuova risorsa.

NON è idempotente:

- 1 POST → crea una risorsa
- 2 POST → ne crea 2

Example: Esempio POST

```
POST /customers
{
  "firstName": "Mario",
  "lastName": "Rossi"
}
```

PUT — aggiornare completamente

PUT

Usato per **sostituire completamente** una risorsa.

È idempotente:

- rifare lo stesso PUT lascia la risorsa invariata

Esempio: aggiornare un profilo utente completo.

PATCH — aggiornare parzialmente

PATCH

Usato per modificare **solo parte** della risorsa.

Non garantisce idempotenza.

Esempio: aggiornare solo l'email del cliente.

DELETE — cancellare

DELETE

Usato per eliminare una risorsa.

È idempotente:

- DELETE /123 → elimina
 - ripetere → risposta 204 o 404 a seconda del server
-

Status code essenziali

Status code essenziali

- 200 OK → operazione riuscita
- 201 Created → risorsa creata
- 204 No Content → ok senza risposta
- 400 Bad Request → errore client
- 401 Unauthorized → manca autenticazione

- 403 Forbidden → credenziali ok ma non permesso
 - 404 Not Found → risorsa non esiste
-

Risorse vs Collezioni

Risorse vs Collezioni

- /products → collezione
- /products/10 → risorsa singola

Pattern universale in REST:

- GET su collezione → lista
- POST su collezione → crea
- GET su risorsa → dettaglio
- PUT/PATCH/DELETE su risorsa → modifica/cancella

 *Example: Esempio*

```
/products  
/products/10
```

Relazioni nelle API REST

Relazioni nelle API REST

Due modi per rappresentare relazioni:

◆ *Attributo*

```
{"provinceId": 12}
```

◆ *Navigazione*

```
/regions/3/provinces
```

La scelta dipende dal caso d'uso:

Example: Confronto

Attributo:	/cities/5 → { "provinceId": 2 }
Navigazione:	/provinces/2/cities

Filtering — Cos'è e perché serve

Filtering — Cos'è e perché serve

Il **filtering** permette al client di chiedere solo ciò che gli serve.

Esempi tipici:

- /products?category=food
- /customers?city=Milano
- /orders?status=PAID

Perché è utile:

- riduce i dati trasferiti
 - riduce il carico sul server
 - rende l'API più "consumer-centric"
-

Filtering — Tipi di filtro

1. Filtro esatto

`/products?category=food`

2. Filtro multiplo

`/products?category=food&brand=Nike`

3. Filtro range

`/products?priceMin=10&priceMax=50`

4. Filtro per testo

`/customers?nameContains=mar`

Una buona API permette *combinazioni* di filtri, senza diventare ingestibile.



Example: Esempio combinato

`/products?category=shoes&priceMax=100&color=red`

Sorting — Ordinare i risultati

Sorting — Ordinare i risultati

Il sorting permette al client di definire l'ordine dei risultati.

Esempio:

```
/products?sort=price,asc
```

Regole comuni:

- specificare il campo
- specificare la direzione (asc, desc)

Molto utile in e-commerce, liste, report, dashboard.

Example: Esempio sorting multiplo

```
/products?sort=category,asc&sort=price,desc
```

Filtering + Sorting — Funzionano insieme

Filtering + Sorting: sì, funzionano insieme

I client reali combinano spesso filtri e sorting:

```
/products?category=food&priceMax=20&sort=price,asc
```

Buone API:

- accettano combinazioni
- documentano i campi validi
- evitano ambiguità
- definiscono default sensati

```
/customers?city=Torino&sort=lastName,asc
```

Perché la pagination è necessaria

Perché serve la Pagination

Liste enormi possono essere un problema:

- `/cities` → 8000+ record
- `/orders` → migliaia per cliente

La **pagination** evita:

- risposte troppo grandi
- lentezza
- overload del server

È come sfogliare un libro: una pagina alla volta.

Pagination — Parametri standard

Pagination — Parametri standard

I parametri più comuni:

- `page` → numero pagina (0-based o 1-based)
- `size` → numero di risultati

Esempio:

```
/products?page=2&size=20
```

Una buona API indica chiaramente:

- se page parte da 0 o 1
- qual è il default per size
- la dimensione massima consentita

Example: Risposta paginata (semplificata)

```
{  
  "content": [...],  
  "page": 2,  
  "size": 20,  
  "totalElements": 8305  
}
```

Cursor Pagination — alternativa moderna

Cursor Pagination — alternativa moderna

Usata da:

- Facebook
- Twitter
- GitHub
- Stripe

Invece di page/size:

```
/products?cursor=abc123&limit=50
```

Utile quando:

- ci sono dati in costante cambiamento
- vuoi prestazioni più elevate
- serve stabilità dei risultati

Example: Esempio risposta

```
{
  "items": [...],
  "nextCursor": "xyz789"
}
```

Errori comuni di Filtering

Errori comuni di Filtering

- ✗ Parametri non documentati
- ✗ Nome dei parametri incoerente
- ✗ Filtri nascosti che il client non può scoprire
- ✗ Filtri troppo complessi in un singolo endpoint

✓ Soluzione: documentazione chiara + esempi + consistenza

Errori comuni di Sorting

Errori comuni di Sorting

- ✗ Ordinare solo per un singolo campo
- ✗ Non validare i campi ricevuti
- ✗ Ambiguità su asc/desc

✓ Soluzione: usare un pattern coerente e documentato

Errori comuni di Pagination

Errori comuni di Pagination

- ✗ Non specificare da che numero parte page
- ✗ Restituire pagine vuote non spiegate
- ✗ Non indicare il totale
- ✗ Size predefiniti troppo grandi

✓ Soluzione: specifica chiaramente tutto nella documentazione

Filtering, Sorting e Pagination insieme

Filtering, Sorting e Pagination insieme

Le API moderne usano spesso tutti e tre:

```
/products?category=food&sort=price,asc&page=1&size=20
```

È qui che si vede la qualità del design.

Se tutto è coerente → l'API "scompare" e chi la usa non deve pensarci.

 *Example: Esempio reale*

```
/orders?customerId=10&status=PAID&sort=date,desc&page=0&size=10
```

Quando NON usare pagination

Quando NON usare pagination

- Liste molto piccole
- Risorse "singole" o quasi statiche
- Risposte dove serve l'intera collezione

Esempi:

- Lista dei metodi di pagamento
- Lista delle lingue disponibili

| Non tutto deve essere paginato.

Best practice di Filtering/Sorting/Pagination

Best Practice

- Documentare ogni parametro
- Usare nomi coerenti
- Avere valori di default sensati
- Mantenere coerenza tra endpoint
- Fornire esempi completi

| La DX (Developer Experience) nasce da queste attenzioni.

Domande chiave per un buon filtering

Domande chiave per un buon filtering

Quando si progetta:

- quali campi servono davvero?
 - quali filtri userà il client?
 - i filtri sono AND, OR, entrambi?
 - posso complicare troppo la vita del consumer?
-

Domande chiave per una buona pagination

Domande chiave per una buona pagination

- la lista può diventare enorme?
 - serve stabilità dei risultati?
 - la pagina ha un limite massimo?
 - il client deve scorrere velocemente?
 - servono link next/prev (HATEOAS)?
-

Conclusione del blocco

Abbiamo visto:

- filtering
- sorting
- pagination
- best practice

Ora passiamo agli **strumenti professionali** per progettare e documentare API.

Strumenti per API

Strumenti per API

Per progettare, testare e documentare API, oggi esistono strumenti adeguati:

- Swagger Editor
- Swagger UI
- Postman
- Stoplight Studio
- Mock server
- Linter API (Spectral)
- curl (per windows <https://curl.se/windows/>)

Non servono competenze tecniche avanzate: questi strumenti rendono le API accessibili a tutti.

Swagger Editor — Cos'è

È uno strumento web per **scrivere e modificare** specifiche OpenAPI.

Permette di:

- creare API da zero
- vedere errori in tempo reale
- validare la struttura
- esportare client e server

È come "Word per le API".



Example: Screenshot concettuale

+-----+	
Swagger Editor (colonne)	
+-----+	
Sinistra: YAML OpenAPI	
Destra: anteprima e validazione	
+-----+	

Perché usare Swagger Editor

- facile da usare
- validazione immediata
- perfetto per team misti (dev + analisti + PO)
- supporta browser: niente installazioni
- standard de facto nel settore

Swagger UI — Cos'è

Swagger UI — Cos'è

Swagger UI è una **documentazione interattiva** generata automaticamente.

Permette di:

- vedere tutti gli endpoint
- aprire i dettagli
- fare richieste direttamente dal browser
- testare esempi

È lo strumento perfetto per chi deve *provare* una API.

Example: Mock di interfaccia

GET /products	[Try it]
POST /products	[Try it]
GET /customers/{id}	[Try it]

Swagger UI — Vantaggi

Swagger UI — Vantaggi

- documentazione sempre aggiornata
- esempi integrati
- possibilità di provare subito gli endpoint
- riduce errori e incomprensioni
- ideale per la collaborazione con esterni

Postman — Cos'è

Postman — Cos'è

Postman è uno strumento per:

- testare API
- creare collezioni
- salvare richieste
- simulare scenari reali

È come un “browser” per API.

Example: Esempio concettuale

```
Collection: Demo API
- GET /products
- POST /orders
- GET /customers/12
```

Postman — Perché usarlo

Perché usare Postman

- simula utenti reali
 - permette test ripetibili
 - gestisce ambienti (dev/test/prod)
 - documentazione integrata
 - condivisione facile con i team
-

Stoplight Studio — Cos'è

Stoplight Studio è uno strumento professionale per il **design-first**.

Caratteristiche:

- editor visuale per OpenAPI
- mock server integrato
- modellazione risorse
- validazione e suggerimenti

Pensato per team enterprise.

 *Example: Screenshot concettuale*

Resources → Products → Fields
Mock server → Enabled
Documentation → Preview

Mock Server — Cos'è

Un mock server simula l'API **prima che esista davvero**.

Serve a:

- testare l'API durante il design
- far lavorare frontend e mobile senza aspettare il backend
- validare le idee con il cliente

Esempio: creare ordini finti senza database.

Example: Esempio concettuale

```
GET /products → restituisce dati di esempio  
POST /orders → simula la creazione di un ordine
```

Linting delle API — Spectral

Linting delle API — Spectral

Spectral analizza una specifica OpenAPI per:

- trovare errori
- suggerire miglioramenti
- imporre standard di qualità

È come un **correttore ortografico** per API.

Example: Messaggio Spectral (esempio)

```
Error: GET /products: missing description  
Warning: schema 'Product' missing field descriptions
```

Documentazione automatica

Documentazione automatica

Con OpenAPI puoi generare automaticamente:

- client (JavaScript, TypeScript, Java, Python...)
- server stub
- pagine di documentazione

Meno errori, più velocità di sviluppo.

 *Example: Esempio output*

```
openapi-generator generate -i api.yaml -g typescript-axios
```

API Explorer — Una visione completa

API Explorer — Una visione completa

Molti strumenti moderni offrono un'esperienza unificata:

- documentazione
- test
- monitoraggio
- collaborazione

Esempi:

- Apigee
- Postman Cloud
- Stoplight Platform

Scegliere lo strumento giusto

Scegliere lo strumento giusto

- Swagger Editor → design iniziale
- Swagger UI → documentazione
- Postman → testing e scenari
- Insomnia → testing e scenari

- Stoplight → enterprise + design-first serio
- Mock server → lavorare prima del backend
- Spectral → qualità

Il professionista API usa **più strumenti**, non uno solo.

Design-first vs Code-first

Design-first vs Code-first

Due modi opposti di creare API:

◆ *Code-first*

1. si scrive il codice
2. poi si "ricava" la documentazione

Problemi:

- documentazione spesso incompleta
- API non progettata, ma "emergente"

◆ *Design-first*

1. si disegna l'API
2. si valida col cliente
3. si genera documentazione e mock
4. poi si sviluppa il codice

È l'approccio usato da Stripe, GitHub, Twilio.

Code-first: Codice → API → Documentazione
Design-first: API → Codice & Documentazione

Vantaggi del Design-first

Vantaggi del Design-first

- API coerente e pensata in anticipo
- niente sorprese per i team consumer
- mock server disponibili subito
- documentazione immediata
- meno errori durante lo sviluppo
- facilita collaborazione tra ruoli

L'API diventa un contratto, non un “effetto collaterale del codice”.

Contract-driven Development

Contract-driven Development

Il contratto (OpenAPI) guida lo sviluppo:

- backend implementa il contratto
- frontend si integra sul contratto
- QA testa sul contratto
- mock server e client generati automaticamente

Riduzione drastica dei misunderstanding tra team.

Example: Metafora

È come costruire un ponte: prima il progetto, poi la struttura.

OpenAPI — Struttura generale

OpenAPI — Struttura generale

Una specifica OpenAPI contiene:

- informazioni generali (titolo, versione)
- elenco degli endpoint
- parametri
- modelli dei dati
- esempi
- codici di risposta
- sicurezza

È un linguaggio universale per descrivere API.

Example: Mini struttura

```
openapi: 3.0.0
info:
  title: Demo API
paths:
  /products:
    get:
      responses:
        '200':
          description: ok
```

OpenAPI — Models e Schemas

I **models** descrivono i dati dell'API:

- campi
- tipi
- obbligatorietà
- esempi
- relazioni

Servono a:

- creare documentazione leggibile
- evitare errori
- standardizzare i dati

| Sono la “forma” pubblica dei dati.

OpenAPI — Esempi (essenziali)

OpenAPI — Esempi

Gli esempi rendono la documentazione molto più chiara.

Senza esempio

Il client deve indovinare.

Con esempio

Il client capisce subito.

Example: Esempio

```
example:  
  id: 12  
  name: "Lombardia"
```

Generazione Client e Server (concetto)

Generazione automatica — Concetto

Dalla specifica OpenAPI si possono generare:

- client SDK (JS, TS, Java, Python...)
- server stub (struttura di progetto)
- documentazione HTML

L'API diventa un *produttore* di codice, non un consumatore.

Example: Esempio concettuale

```
API.yaml → Client JS  
         → Client Java  
         → Mock Server
```

Serializzazione e Deserializzazione

Serializzazione e Deserializzazione

Serializzazione

Trasformare un oggetto → JSON (o XML)

Deserializzazione

Trasformare JSON → oggetto

Il server REST usa continuamente questi processi.

Example: Esempio concettuale

Oggetto → JSON (risposta API)
JSON → Oggetto (richiesta dell'utente)

Validazione degli input

Validazione degli input

Il server deve controllare:

- campi obbligatori
- formati (es. email)
- range numerici
- consistenza dei dati

Una buona validazione previene errori e abusi.

Example: Esempio concettuale

```
email: obbligatoria  
dataNascita: deve essere nel passato  
amount: >= 0
```

Interazione con i Database

Interazione con i Database

Il server REST deve salvare e leggere dati.

Tipologie principali:

- **Database relazionali** (PostgreSQL, MySQL)
- **Database NoSQL** (MongoDB, DynamoDB)

Ogni scelta ha impatti su:

- scalabilità
- consistenza
- prestazioni
- flessibilità del modello

Relazionali vs NoSQL — differenze

Relazionali

- schema fisso
- relazioni chiare
- ottimi per dati strutturati

NoSQL

- schema flessibile
- ottimi per grandi volumi
- ottimi per dati semi-strutturati

Example: Esempio intuitivo

Relazionale: tabella clienti con colonne fisse

NoSQL: documento JSON con campi variabili

Gestione delle transazioni

Gestione delle transazioni

Le transazioni garantiscono:

- consistenza dei dati
- sicurezza delle operazioni
- nessuna “mezza scrittura”

Esempi classici:

- operazioni bancarie
- aggiornamenti multipli
- creazione ordine + pagamento

Gestione delle eccezioni

Gestione delle eccezioni

Un'API deve:

- intercettare errori
- restituire messaggi chiari
- usare i corretti status code
- non esporre dettagli interni

Una buona gestione degli errori migliora la DX.

 *Example: Messaggio concettuale*

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

Quando usare ciascuna tecnologia

Quando usare ciascuna tecnologia

- **Design-first** → quando il team è grande o multi-ruolo
- **Code-first** → quando il progetto è piccolo o prototipale
- **Relazionale** → quando il dominio è complesso
- **NoSQL** → quando i dati sono flessibili o enormi

Decidere *prima* evita refactoring costosi.

Esercizio — Introduzione

Esercizio finale

Un'attività pratica:

👉 Progettare una piccola API **design-first**, usando quello che abbiamo visto:

- risorse
- URI
- metodi HTTP
- parametri
- pagination
- esempi
- OpenAPI (livello base)

| Non serve saper programmare: è un esercizio di *design*.

Scenario: API e-commerce (ridotta)

Scenario: e-commerce minimale

I team devono progettare l'API per:

- sfogliare prodotti
- cercare prodotti
- recuperare dettagli
- creare ordini (solo concetto, non tecnicamente)

Vincoli:

- il catalogo ha 10.000 prodotti
- i clienti filtrano per categoria, nome, fascia prezzo
- i risultati vanno paginati

| Il focus è la progettazione *consumer-centric*.

Compito: definire le risorse

Compito (1): definire le risorse

Ogni gruppo deve proporre l'elenco delle risorse principali.

Esempi possibili:

- /products
- /categories
- /orders

Domande guida:

- cosa deve poter consultare il client?
- cos'è una risorsa e cosa NON lo è?



Exercise: Esercizio — Risorse

Elenca 3–5 risorse fondamentali del mini e-commerce.

Compito: definire gli URI

Compito (2): definire gli URI

Per ogni risorsa, definire gli URI:

Esempio:

- `/products`
- `/products/{id}`

Domande guida:

- la struttura è coerente?
- si distinguono collezioni e singole risorse?
- la naming convention è uniforme?



Exercise: Esercizio — URI

Definisci gli URI principali per ogni risorsa scelta.

Compito: definire i parametri

Compito (3): definire i parametri

I client filtrano spesso i prodotti.

Compito:

- decidere quali filtri offrire

- decidere quali sort offrire
- decidere quali parametri sono obbligatori

Domande guida:

- quali parametri servono davvero al consumer?
- ci sono casi d'uso reali?



Exercise: Esercizio — Parametri

Definisci 3 parametri di filtro e 1 di sorting per `/products`.

Compito: Pagination

Compito (4): Pagination

Dato che i prodotti sono 10.000:

- definire `page` e `size`
- decidere il valore di default di `size`
- decidere il massimo consentito

L'obiettivo è evitare risposte troppo grandi.



Example: Esempio

```
/products?page=0&size=20
```



Exercise: Esercizio — Pagination

Scegli un default per `size` e una dimensione massima ragionevole.

Compito: esempi (Example-driven API)

Compito (5): creare esempi

Ogni risorsa deve avere **almeno un esempio**.

Perché gli esempi sono importanti:

- riducono le ambiguità
- rendono chiaro il contratto

Richiesta:

- creare 1–2 esempi JSON per ogni risorsa



Exercise: Esercizio — Esempi

Creare un esempio JSON realistico per `/products/{id}`.

Compito: Mini-specifica OpenAPI

Compito (6): Mini-specifica OpenAPI

Usare:

- Swagger Editor
- Stoplight Studio o strumenti equivalenti

Obiettivo:

- produrre una piccola specifica con:
 - titolo
 - 2 risorse
 - 2 endpoint
 - esempi
 - parametri

- pagination

Non importa la sintassi: conta la chiarezza.



Exercise: Esercizio — Mini OpenAPI

Progetta la mini-specifica per /products e /orders.

Soluzione guida — Risorse

Soluzione guida — Step 1: Risorse

► Mostra la soluzione proposta

Risorse principali:

- /products
- /categories
- /orders

Risorse secondarie:

- /products/{id}
- /categories/{id}
- /orders/{id}



Example: Soluzione risorse

Vedi sezione dettagliata nella slide.

Soluzione guida — Step 2: URI

► Mostra la soluzione proposta

```
/products  
/products/{id}  
/categories  
/categories/{id}  
/orders  
/orders/{id}
```

 *Example: URI corretti*

Struttura pulita e RESTful.

Soluzione guida — Step 3: Filtering & Sorting

► Mostra la soluzione proposta

Filtering:

```
/products?category=food  
/products?priceMax=50  
/products?nameContains=chi
```

Sorting:

```
/products?sort=price,asc
```

Example: Filtering/Sorting proposti

Esempi minimi per chiarezza.

Soluzione guida — Pagination

Soluzione guida — Step 4: Pagination

► Mostra la soluzione proposta

Default suggerito:

```
size = 20  
max size = 100
```

Esempio:

```
/products?page=0&size=20
```

Example: Esempio pagina

```
/products?page=0&size=20
```

Soluzione guida — Mini OpenAPI

Soluzione guida — Mini OpenAPI

► Mostra la soluzione proposta

```
openapi: 3.0.0  
info:  
  title: Mini E-commerce API
```

```
paths:
  /products:
    get:
      parameters:
        - name: category
          in: query
        - name: priceMax
          in: query
        - name: sort
          in: query
        - name: page
          in: query
        - name: size
          in: query
      responses:
        '200':
          description: ok
```

Example: Mini OpenAPI

Specifica ridotta e leggibile.

Cheat-Sheet — REST, HTTP e API Design

Cheat-Sheet REST, HTTP e API Design

◆ Fondamenti REST

- risorse con URI
- stateless
- cache
- uniform interface
- layered system

◆ Metodi HTTP

- GET (legge)
- POST (crea)
- PUT (sostituisce, idempotente)
- PATCH (aggiornamento parziale)
- DELETE (idempotente)

◆ Status Code

- 200 OK
- 201 Created
- 204 No Content
- 400 / 401 / 403
- 404 Not Found
- 409 Conflict

◆ Filtering / Sorting / Pagination

- ?category=food
- ?sort=price,asc
- ?page=0&size=20

◆ HATEOAS

Risposte che includono link alle operazioni successive.

◆ Design-first

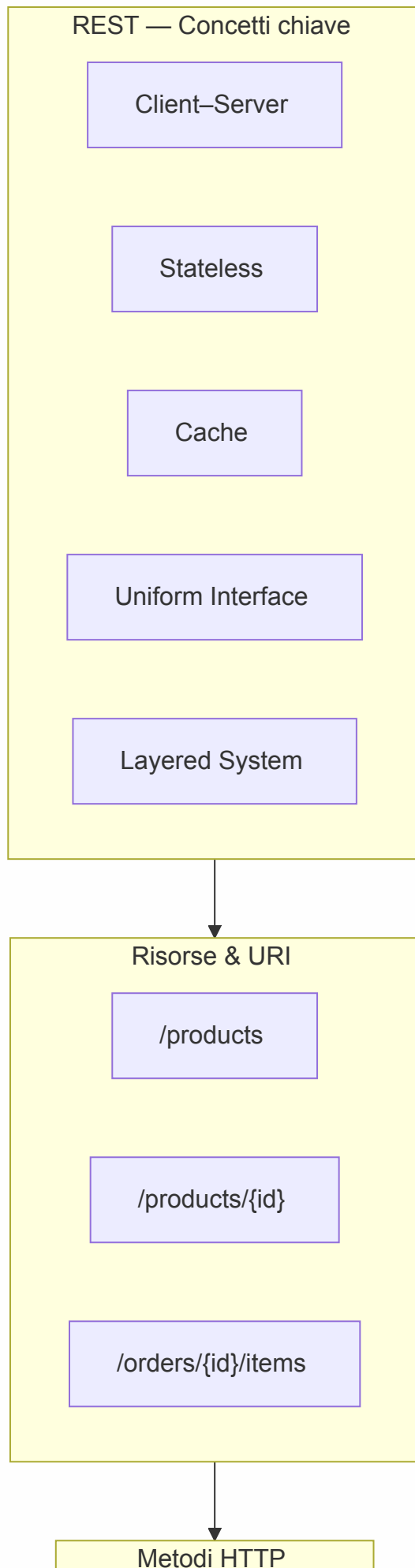
Progettare prima l'API, poi il codice.

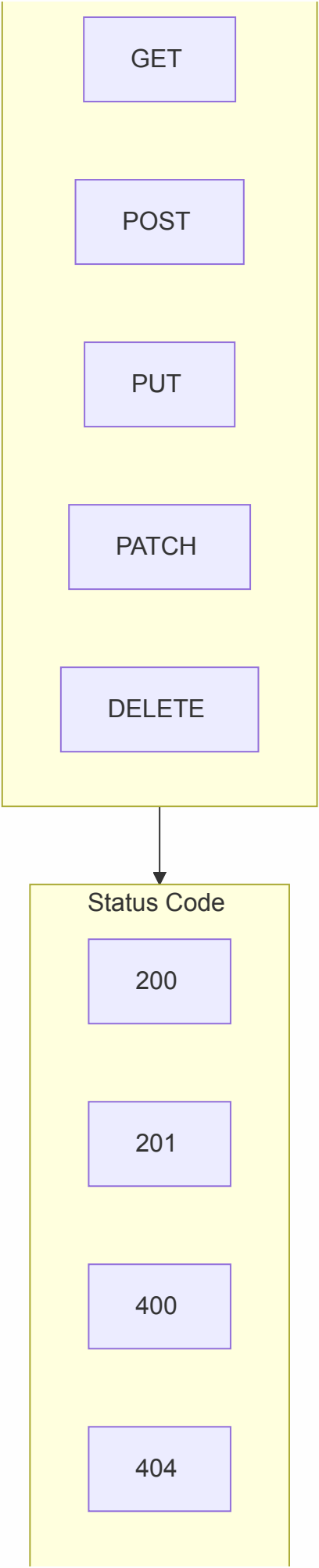
Example: Esempio HATEOAS

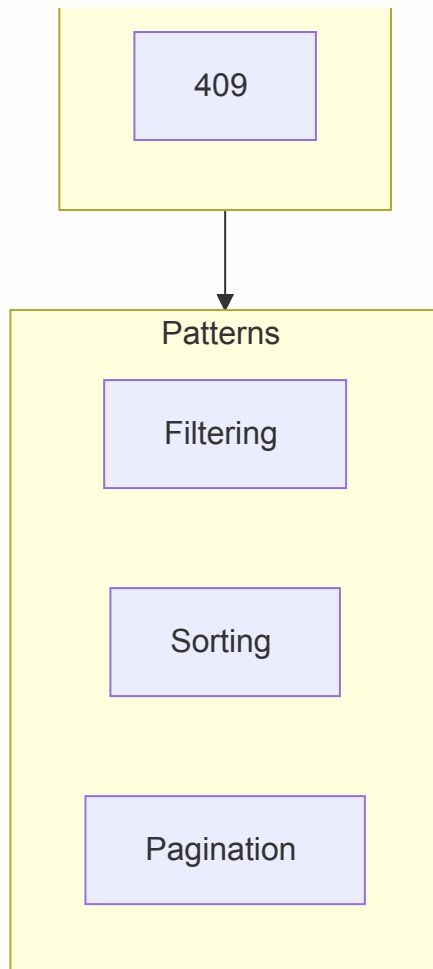
```
{
  "id": 12,
  "name": "Lombardia",
  "_links": {
    "self": "/regions/12",
    "provinces": "/regions/12/provinces"
  }
}
```

Cheat-Sheet Visuale (Schema REST)

Schema Visuale — REST & API Design







Example: Descrizione

Schema visuale completo delle relazioni tra concetti REST.
