REST — Representational State Transfer

Cos'è REST

REST (Representational State Transfer) è uno **stile architetturale** per la progettazione di sistemi distribuiti basati su ipermedia.
È stato introdotto nel **2000** da **Roy Fielding** nella sua tesi di dottorato presso l'Università della California, Irvine.
L'obiettivo principale di REST è:

> **Trasferire rappresentazioni di risorse tra componenti di un sistema (ad esempio tra un client e un server).**

REST non è un protocollo o una libreria: è **un insieme di principi e vincoli** che, se rispettati, rendono un sistema "RESTful".

In un'architettura REST, le **risorse** vengono identificate in modo univoco (di solito tramite URI), e la loro **rappresentazione** viene trasferita tra client e server per compiere operazioni come lettura, creazione o modifica.

Il **server** espone le risorse.
Il **client** interagisce con esse attraverso un'interfaccia uniforme, utilizzando protocolli standard come **HTTP**.

Risorsa (Resource)

Una **risorsa** è qualsiasi informazione che può essere nominata o identificata.
Esempi di risorse:

- Un documento (es. `manuale.pdf`)
- Un'immagine (es. `photo.jpg`)
- Un servizio (es. `weather-api`)
- Un oggetto fisico (es. una persona)
- Una collezione di risorse (es. tutti gli utenti)

Nota:

- Una risorsa è **un'entità logica**: può cambiare nel tempo, ma la sua identità rimane la stessa.
- Due risorse diverse possono anche rappresentare **lo stesso valore** in un dato istante.
  Entrambe possono puntare alla stessa versione del software in un certo momento, ma concettualmente sono risorse distinte.

Rappresentazione (Representation)

Una **rappresentazione** è **lo stato attuale o previsto di una risorsa**, cioè la forma con cui la risorsa viene trasferita da server a client (o viceversa).
Il client e il server non manipolano mai direttamente la risorsa, ma lavorano **sulla sua rappresentazione**

Struttura di una rappresentazione

Una rappresentazione è composta da:

1. **Dati** → il contenuto effettivo della risorsa (es. JSON, XML, HTML)
2. **Metadati** → informazioni sul formato, data di aggiornamento, cache, ecc.
   Il formato dei dati è definito dal **Media Type** (o MIME type).

Identificatori di Risorse (URI)

Per accedere o manipolare una risorsa è necessario **identificarla univocamente**.
In REST, ciò avviene tramite **Uniform Resource Identifier (URI)**.
Una **URI** è una **stringa univoca** che identifica una risorsa logica o fisica.
Esempi:

```
http://example.com/users http://api.shop.com/products/123
```

Le URI non devono descrivere azioni o operazioni, ma solo risorse (nouns, non verbs).```

Best Practices per le URI

1. **Usare sostantivi**, non verbi → le URI rappresentano risorse, non azioni.
   ❌ `/get-user`
   ✅ `/users/45`
2. **Sostantivi singolari** per risorse singole → `/users/45`
3. **Sostantivi plurali** per collezioni → `/users/`
4. **Usare "/" per indicare gerarchie**
   Esempio: `/users/45/orders`
5. **Usare trattini (-)** invece di underscore `_`
6. **Usare solo lettere minuscole**
7. **Evitare estensioni di file** (come `.json` o `.xml`) — il formato è comunicato negli header HTTP
8. **Usare query string per filtri o parametri**
   Esempio: `http://example.com/devices?region=EU&status=active`

Struttura generale di una URI

```
protocol://subdomain.domain.tld/path?query
```

Esempio:

```
https://www.google.com/maps/?hl=it
```

Vincoli di un Sistema RESTful

1. **Client-Server**
   Il principio **client-server** separa le responsabilità:
   - Il **client** gestisce l'interfaccia utente e l'esperienza dell'utente.
   - Il **server** gestisce la logica applicativa e l'archiviazione dei dati.

   ## Benefici
   - Maggiore **scalabilità**: è possibile aggiornare o sostituire il server senza modificare i client.
   - Maggiore **portabilità**: l'interfaccia utente può evolversi indipendentemente dal backend.
     Esempio:
     Un'app mobile invia richieste a un'API REST per recuperare dati → il client si occupa solo di visualizzarli.
2. **Stateless**
   Ogni richiesta HTTP deve contenere **tutte le informazioni necessarie** per essere compresa ed eseguita. Il server **non deve mantenere alcuno stato di sessione** tra richieste consecutive.
   - Il server tratta **ogni richiesta come indipendente**.
   - Il **client** è responsabile di mantenere lo stato dell'applicazione (es. token di sessione, filtri, selezioni).
3. **Cacheable**
   Le risposte del server possono essere **memorizzate (cache)** dal client, da proxy o da gateway, se contrassegnate come tali.
   Il server specifica se una risposta è cacheable e per quanto tempo.
4. **Uniform Interface**
   L'interfaccia uniforme è il vincolo **più importante** di REST.
   Consente un'interazione coerente tra componenti, **indipendentemente dall'implementazione**
   - Un'interfaccia uniforme standardizza l'interazione tra componenti.
   - Quattro principi:
     1. Identificazione delle risorse (tramite URI)
     2. Manipolazione tramite rappresentazioni
     3. Messaggi auto-descrittivi
     4. Hypermedia come motore dello stato dell'applicazione (HATEOAS)
        - Le transizioni di stato dell'applicazione vengono guidate da **link ipermediali** inclusi nelle risposte.

   ```
   - Il client conosce solo l'URI iniziale e scopre le altre risorse tramite i link.
   ```
5. **Layered System**
   - L'architettura può includere più strati (client, proxy, gateway, server).
   - Ogni componente vede solo lo **strato adiacente**.

HTTP e REST

I metodi HTTP (verbi) definiscono **quale azione** compiere sulla risorsa identificata dalla URI. In REST, **le URI identificano le risorse (sostantivi)** e **i metodi HTTP rappresentano le azioni (verbi)**.

Metodi HTTP (verbi Rest)

Metodo	Scopo	Corrispondenza CRUD	Descrizione
GET	Recupera una risorsa o collezione	**Read**	Non modifica dati. È sicuro e idempotente.
POST	Crea una nuova risorsa	**Create**	Usato sulle collezioni. Non è idempotente.
PUT	Sostituisce completamente una risorsa	**Update/Replace**	È idempotente: eseguire più volte non cambia il risultato.
PATCH	Modifica parzialmente una risorsa	**Update/Modify**	Aggiorna solo i campi specificati.
DELETE	Elimina una risorsa	**Delete**	È idempotente.

Codici di successo comuni (2xx)

Codice	Significato	Quando usarlo
200 OK	La richiesta è stata completata con successo.	Operazioni **GET**, **PUT**, **PATCH** o **DELETE** riuscite.
201 Created	È stata creata una nuova risorsa.	Dopo un **POST** riuscito. Deve includere l'URI della nuova risorsa nel campo `Location`.
204 No Content	L'azione è andata a buon fine, ma non c'è contenuto nella risposta.	Tipico per **DELETE** o **PUT** senza corpo di risposta.

Codici di errore del client (4xx)

Gli errori 4xx indicano che **il problema è nella richiesta inviata dal client**.

Codice	Significato	Causa comune
400 Bad Request	La richiesta non può essere interpretata o contiene errori di sintassi.	JSON malformato, campi mancanti, tipo errato.
401 Unauthorized	Mancano le credenziali di autenticazione o sono errate.	Nessun token o token scaduto.
403 Forbidden	L'utente è autenticato ma **non autorizzato** ad accedere alla risorsa.	Mancanza di permessi.
404 Not Found	La risorsa richiesta **non esiste**.	URI errata o ID inesistente.

Azione o Risorsa?

Uno degli errori più comuni nella progettazione di API REST è **confondere le azioni con le risorse**.
Ricorda:

> Le URI identificano *cose* (sostantivi), non *azioni* (verbi).