REST API Design

Progettazione e Implementazione di API RESTful

15 novembre 2025

Indice

Prefazione

A chi si rivolge questo manuale

Questi appunti sono stati pensati per sviluppatori, studenti e professionisti IT che desiderano approfondire la progettazione e implementazione di REST API moderne e scalabili. Il manuale copre dai principi fondamentali del REST alle best practices professionali per la costruzione di API robuste, sicure e ben documentate.

Struttura del corso

Il corso è organizzato in 12 capitoli che coprono l'intero ciclo di progettazione e sviluppo di REST API:

Parte I - Fondamenti REST (Capitoli 1-3)

- Principi architetturali REST e Richardson Maturity Model
- Confronto REST vs SOAP vs GraphQL
- Metodi HTTP: GET, POST, PUT, PATCH, DELETE
- Proprietà di idempotenza e safety
- Status codes HTTP completi (1xx-5xx) e loro utilizzo appropriato

Parte II - Design e Struttura (Capitoli 4-6)

- Resource design: URI patterns, naming conventions
- Nesting, filtering, sorting e searching
- Formati dati: JSON, HAL, JSON:API
- Content negotiation e hypermedia
- Strategie di versioning: URI, header, semantic versioning

Parte III - Sicurezza e Performance (Capitoli 7-9)

- Autenticazione: Basic Auth, API Keys, OAuth 2.0, JWT
- Rate limiting e throttling
- Pagination, filtering avanzato, caching
- CORS e security headers

Parte IV - Professionalizzazione (Capitoli 10-12)

- Error handling standardizzato (Problem Details RFC 7807)

- Documentazione con OpenAPI/Swagger
- Best practices: HATEOAS, testing, monitoring
- Casi di studio reali

Prerequisiti

Per affrontare questo corso è consigliabile avere:

- Conoscenza di base del protocollo HTTP
- Familiarità con JSON e formati di dati strutturati
- Esperienza con almeno un linguaggio di programmazione (Python, Java, JavaScript, PHP)
- Comprensione dei concetti di client-server e web services
- (Opzionale) Nozioni di database relazionali

Strumenti necessari

Tool per testing e sviluppo:

- **Postman**: Client API completo per testing e documentazione
- **Insomnia**: Alternative a Postman, focus su GraphQL e REST
- **curl**: Command-line tool per richieste HTTP
- **HTTPie**: CLI user-friendly per API testing
- **jq**: Processore JSON da command line

Framework e runtime:

- **Node.js + Express**: Framework JavaScript/TypeScript
- **Python + FastAPI/Flask**: Framework Python moderni
- **Java + Spring Boot**: Enterprise Java framework
- **PHP + Laravel/Symfony**: Framework PHP per API
- **ASP.NET Core**: Framework Microsoft .NET

Documentazione e design:

- **Swagger Editor**: Editor OpenAPI online
- **Swagger UI**: Interfaccia interattiva per API docs
- **Stoplight Studio**: Design-first API development
- **Redoc**: Generatore documentazione da OpenAPI

Testing e monitoring:

- **Newman**: Test runner per Postman collections
- **REST Assured**: Testing library Java
- **Gatling/K6**: Load testing e performance
- **Datadog/New Relic**: APM e monitoring

Come studiare

Per ottenere il massimo da questi appunti:

1. **Comprendi i principi**: Il REST è prima di tutto un'architettura
2. **Testa ogni esempio**: Usa Postman/curl per provare le richieste
3. **Analizza gli status codes**: Impara quando usare 200 vs 201 vs 204
4. **Progetta prima di implementare**: Disegna la struttura delle risorse
5. **Leggi le specifiche**: RFC HTTP, OpenAPI spec, JSON:API
6. **Studia API reali**: GitHub, Stripe, Twitter API come reference
7. **Implementa progetti**: Crea una API completa end-to-end
8. **Documenta sempre**: API senza documentazione è API inutilizzabile

Nota importante

Questo manuale segue gli standard **HTTP/1.1 RFC 7231-7235**, **REST RFC 6570**, **OpenAPI 3.1**, e **JSON:API 1.1**. Gli esempi sono language-agnostic ma includono snippet in Python, JavaScript e PHP quando necessario per chiarezza.

Convenzioni tipografiche

Nel testo vengono utilizzate le seguenti convenzioni:

- `Codice HTTP/JSON`: Request, response e payload in monospace
- **Metodi HTTP**: In grassetto (GET, POST, PUT, DELETE)
- *Header HTTP*: In corsivo (Content-Type, Authorization)
- `/api/resources`: URI endpoints e path parameters
- Box colorati: Best practices, Errori comuni, Attenzioni, Esempi

Format esempio HTTP request/response:

```
GET /api/v1/users/123 HTTP/1.1
Host: api.example.com
Accept: application/json
Authorization: Bearer eyJhbGc...

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

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

API di riferimento studiate

Vengono analizzate come case study le seguenti API pubbliche:

- **GitHub REST API v3**: Design classico e maturo
- **Stripe API**: Eccellente developer experience
- **Twitter API v2**: Filtering e pagination avanzata
- **OpenWeatherMap API**: Esempio di API semplice e chiara
- **JSONPlaceholder**: Mock API per testing e prototyping
- **REST Countries**: Dataset geografici RESTful

Sito web e risorse

Materiale aggiuntivo disponibile su:

- Repository GitHub: https://github.com/campionluca/Appunti
- Collezioni Postman scaricabili
- Specifica OpenAPI 3.1 completa
- Mock server per testing
- Esempi di implementazione in vari linguaggi

Risorse online consigliate:

- https://restfulapi.net/ - Tutorial e best practices
- https://www.ietf.org/rfc/rfc7231.txt - HTTP/1.1 Specification
- https://jsonapi.org/ - JSON:API specification
- https://swagger.io/specification/ - OpenAPI Specification
- https://developer.mozilla.org/en-US/docs/Web/HTTP - MDN HTTP docs

Ringraziamenti

Si ringrazia l'Istituto Tecnico Antonio Scarpa per il supporto nella realizzazione di questo materiale didattico. Un ringraziamento speciale alla community open source e agli autori delle specifiche REST, HTTP e OpenAPI.

Prof. Luca Campion
Novembre 2025

Note sulla versione

Versione 1.0 - Novembre 2025

- Prima release completa
- 12 capitoli + 2 appendici
- Coverage: REST principles, HTTP methods, status codes, resource design
- Versioning, authentication, rate limiting, pagination
- Error handling, OpenAPI documentation, best practices
- Esempi completi con JSON request/response
- Diagrammi HTTP flow con TikZ
- Case study: GitHub, Stripe, Twitter API

Aggiornamenti previsti:

- GraphQL comparison approfondita
- WebSocket integration pattern
- gRPC e HTTP/2 considerations
- API Gateway patterns
- Microservices e distributed tracing

Feedback e contributi

Questo è un documento in evoluzione. Segnalazioni di errori, suggerimenti e contributi sono benvenuti:

- Issue tracker GitHub: `https://github.com/campionluca/Appunti/issues`
- Email: `luca.campion@example.com`
- Pull requests per correzioni e miglioramenti

Licenza

Buono studio e buon design di API RESTful!

Capitolo 1

Introduzione a REST

Mappa del capitolo

Definizione REST, Principi architetturali, Vincoli REST, Richardson Maturity Model, REST vs SOAP vs GraphQL, Architettura client-server, Stateless communication, Cacheability, Layered system, Uniform interface, Code on demand, Esempi pratici, Diagrammi, Best practices.

Obiettivi di apprendimento

- Comprendere i principi fondamentali dell'architettura REST
- Conoscere i sei vincoli architetturali di REST
- Applicare il Richardson Maturity Model per valutare API
- Confrontare REST, SOAP e GraphQL: vantaggi e svantaggi
- Progettare API RESTful seguendo le best practices
- Riconoscere API veramente RESTful da quelle HTTP-based

1.1 Cos'è REST

1.1.1 Definizione

REST (Representational State Transfer) è uno **stile architetturale** per sistemi distribuiti, introdotto da Roy Fielding nella sua tesi di dottorato del 2000. Non è un protocollo né uno standard, ma un insieme di **vincoli architetturali** che, se applicati, rendono un sistema scalabile, performante e manutenibile.

Concetto chiave

REST non significa "usare HTTP con JSON". È un'architettura basata su:

- **Risorse** identificate da URI
- **Rappresentazioni** (JSON, XML, HTML) delle risorse
- **Operazioni standard** (GET, POST, PUT, DELETE) su risorse
- **Stateless** communication tra client e server
- **Hypermedia** per navigare tra risorse correlate

1.1.2 REST vs HTTP API

Molte API sono chiamate "REST API" ma in realtà sono solo **HTTP API**:

- **HTTP API**: Usa HTTP per trasporto, ma non rispetta vincoli REST
- **REST API**: Rispetta tutti i vincoli REST (client-server, stateless, cacheable, layered, uniform interface, code on demand)

Listing 1.1: Esempio: HTTP API (NON RESTful)

```
POST /getUserById HTTP/1.1
Content-Type: application/json

{
  "userId": 123
}
```

Problema: usa POST per operazione di lettura, nome operation-based invece di resource-based.

Listing 1.2: Esempio: REST API (RESTful)

```
GET /api/v1/users/123 HTTP/1.1
Accept: application/json

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

{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario.rossi@example.com",
  "_links": {
    "self": {"href": "/api/v1/users/123"},
    "orders": {"href": "/api/v1/users/123/orders"}
  }
}
```

Corretto: GET per lettura, URI resource-based, hypermedia links.

1.2 I Sei Vincoli Architetturali REST

Roy Fielding definisce sei vincoli che caratterizzano REST:

1.2.1 1. Client-Server

Principio: Separazione delle responsabilità tra client (UI) e server (dati/logica).
Vantaggi:

- Client e server evolvono indipendentemente
- Miglior portabilità UI
- Scalabilità server separata da client

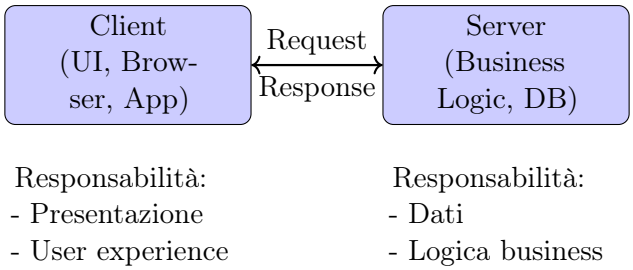


Figura 1.1: Architettura Client-Server

1.2.2 2. Stateless

Principio: Ogni richiesta deve contenere tutte le informazioni necessarie. Il server non mantiene stato della sessione client.

Implicazioni:

- Nessuna sessione server-side
- Autenticazione in ogni richiesta (token, API key)
- Client mantiene stato applicazione

Listing 1.3: Esempio Stateless: token in ogni richiesta

```
GET /api/v1/orders HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Accept: application/json

# Server non ha sessione, autentica via token
```

Vantaggi:

- Scalabilità: qualsiasi server può gestire qualsiasi richiesta
- Affidabilità: nessuna sincronizzazione stato tra server
- Visibilità: request completa per monitoring/debugging

Svantaggi:

- Overhead: ripetere dati in ogni richiesta
- Performance: nessuna ottimizzazione basata su stato precedente

1.2.3 3. Cacheable

Principio: Le risposte devono esplicitamente indicare se sono cacheable o meno.

HTTP Headers per caching:

- `Cache-Control`: Direttive di caching
- `ETag`: Identificatore versione risorsa
- `Last-Modified`: Data ultima modifica
- `Expires`: Data scadenza cache

Listing 1.4: Response cacheable con Cache-Control

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=3600, public
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
Last-Modified: Wed, 15 Nov 2025 10:00:00 GMT

{
  "id": 123,
  "name": "Product ABC",
  "price": 99.99
}
```

Richiesta successiva con ETag:

```
GET /api/v1/products/123 HTTP/1.1
If-None-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4"

HTTP/1.1 304 Not Modified
# Server risponde 304, client usa cache locale
```

1.2.4 4. Layered System

Principio: Client non sa se è connesso direttamente al server finale o a un intermediario.
Intermediari possibili:

- Load balancer
- Proxy cache (Varnish, Cloudflare)
- API Gateway
- Firewall

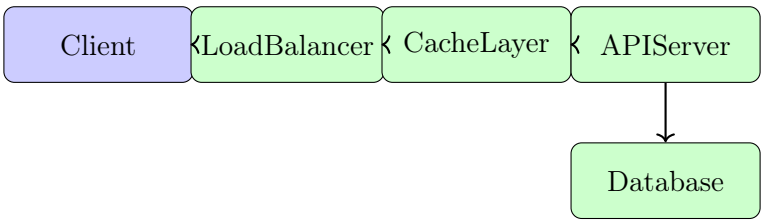


Figura 1.2: Layered System: client ignora intermediari

Vantaggi:

- Scalabilità: aggiungere layer senza modificare client
- Security: firewall e gateway nascondono architettura interna
- Caching centralizzato

1.2.5 5. Uniform Interface

Principio: Interfaccia uniforme semplifica e disaccoppia architettura.
Quattro sotto-vincoli:
a) Identificazione risorse: URI univoci

```
/api/v1/users/123
/api/v1/orders/456
/api/v1/products?category=electronics
```

b) Manipolazione tramite rappresentazioni: Client manipola rappresentazione (JSON), non risorsa diretta

```
PUT /api/v1/users/123 HTTP/1.1
Content-Type: application/json

{
  "name": "Mario Rossi Updated",
  "email": "mario.updated@example.com"
}
```

c) Messaggi auto-descrittivi: Ogni messaggio include metadati per processarlo

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 142
Cache-Control: max-age=300

{...}
```

d) Hypermedia as the Engine of Application State (HATEOAS): Response include link a risorse correlate

```
{
  "id": 123,
  "status": "processing",
  "_links": {
    "self": {"href": "/api/v1/orders/123"},
    "cancel": {"href": "/api/v1/orders/123/cancel", "method": "POST"},
    "customer": {"href": "/api/v1/customers/789"}
  }
}
```

1.2.6 6. Code on Demand (opzionale)

Principio: Server può estendere funzionalità client inviando codice eseguibile (JavaScript, applet).

Esempio:

- API invia JavaScript per validazione form
- Single Page Application che carica dinamicamente UI component

Nota: Unico vincolo opzionale. Raramente implementato in REST API moderne.

1.3 Richardson Maturity Model

Leonard Richardson propone modello a 4 livelli per valutare "quanto REST" è una API:

1.3.1 Level 0: The Swamp of POX

POX = Plain Old XML (o JSON).

Single URI, tutto via POST, payload contiene operazione:

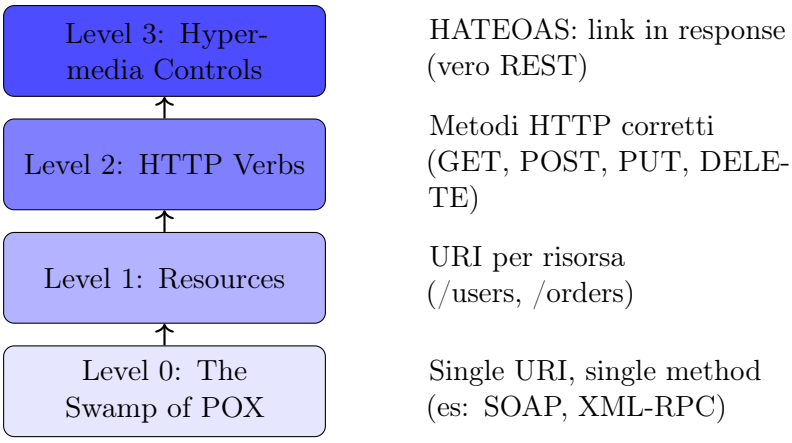


Figura 1.3: Richardson Maturity Model

Listing 1.5: Level 0 Example (SOAP-like)

```
POST /api HTTP/1.1
Content-Type: application/json

{
  "operation": "getUser",
  "parameters": {
    "userId": 123
  }
}
```

Problema: HTTP usato solo come tunnel. Nessun vantaggio REST.

1.3.2 Level 1: Resources

URI per ogni risorsa:

Listing 1.6: Level 1: URI per risorse

```
POST /api/users/123 HTTP/1.1
# Ottiene user 123

POST /api/orders/456 HTTP/1.1
# Ottiene order 456
```

Miglioramento: risorse identificate, ma metodi HTTP non usati correttamente (tutto POST).

1.3.3 Level 2: HTTP Verbs

Usa metodi HTTP semanticamente corretti:

Listing 1.7: Level 2: HTTP Verbs corretti

```
GET /api/users/123 HTTP/1.1
# Legge user

POST /api/users HTTP/1.1
Content-Type: application/json
{"name": "Mario", "email": "mario@example.com"}
# Crea nuovo user

PUT /api/users/123 HTTP/1.1
```

```
Content-Type: application/json
{"name": "Mario Rossi"}
# Aggiorna user

DELETE /api/users/123 HTTP/1.1
# Elimina user
```

Status codes HTTP semantici:

```
HTTP/1.1 200 OK          # GET successo
HTTP/1.1 201 Created     # POST creazione
HTTP/1.1 204 No Content  # DELETE successo
HTTP/1.1 404 Not Found   # Risorsa non trovata
```

1.3.4 Level 3: Hypermedia Controls (HATEOAS)

Response include link navigabili:

Listing 1.8: Level 3: HATEOAS full REST

```
GET /api/orders/123 HTTP/1.1

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

{
  "id": 123,
  "status": "shipped",
  "total": 99.99,
  "items": [
    {
      "productId": 789,
      "quantity": 2,
      "price": 49.99
    }
  ],
  "_links": {
    "self": {
      "href": "/api/orders/123"
    },
    "customer": {
      "href": "/api/customers/456"
    },
    "invoice": {
      "href": "/api/orders/123/invoice"
    },
    "track": {
      "href": "/api/orders/123/tracking"
    }
  }
}
```

Client può navigare senza conoscere URI structure a priori.

Nota pratica

La maggior parte delle API moderne si ferma al **Level 2**. Level 3 (HATEOAS) è raro perché:

- Complessità implementativa
- Client spesso preferiscono URI hardcoded
- Overhead payload con link

Tuttavia, HATEOAS è fondamentale per API veramente RESTful e auto-documentanti.

1.4 REST vs SOAP vs GraphQL

1.4.1 SOAP (Simple Object Access Protocol)

Caratteristiche:

- Protocollo XML-based
- WSDL per definizione interfaccia
- WS-Security per sicurezza enterprise
- Stateful o stateless
- Overhead elevato

Esempio SOAP request:

```
POST /api/soap HTTP/1.1
Content-Type: application/soap+xml

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <auth:credentials>token123</auth:credentials>
  </soap:Header>
  <soap:Body>
    <m:GetUser xmlns:m="http://example.com/users">
      <m:UserId>123</m:UserId>
    </m:GetUser>
  </soap:Body>
</soap:Envelope>
```

Quando usare SOAP:

- Ambiente enterprise con WS-* standards
- Transazioni ACID distribuite
- Sicurezza complessa (WS-Security)
- Legacy integration

1.4.2 GraphQL

Caratteristiche:

- Query language per API
- Client richiede esattamente dati necessari

- Single endpoint
- Strongly typed schema
- Introspection

Esempio GraphQL query:

Listing 1.9: GraphQL: client specifica campi

```
POST /graphql HTTP/1.1
Content-Type: application/json

{
  "query": "{ user(id: 123) { name email orders { id total } } }"
}

# Response include solo campi richiesti
{
  "data": {
    "user": {
      "name": "Mario Rossi",
      "email": "mario@example.com",
      "orders": [
        {"id": 1, "total": 99.99},
        {"id": 2, "total": 149.99}
      ]
    }
  }
}
```

Quando usare GraphQL:

- Client con necessità dati variabili (mobile, web, desktop)
- Evitare under-fetching/over-fetching
- Real-time subscriptions
- Rapid prototyping frontend

1.4.3 Confronto REST vs SOAP vs GraphQL

1.5 Quando usare REST

REST è ideale per:

- **API pubbliche**: GitHub, Stripe, Twitter
- **Microservices**: Comunicazione inter-service
- **CRUD operations**: Create, Read, Update, Delete risorse
- **Web applications**: Backend per frontend
- **Mobile backend**: App iOS/Android
- **IoT**: Device communication

REST NON è ideale per:

Aspetto	REST	SOAP	GraphQL
Tipo	Architettura	Protocollo	Query language
Formato	JSON, XML, HTML	XML obbligatorio	JSON
Endpoint	Multipli (/users, /orders)	Singolo	Singolo
Metodi HTTP	GET, POST, PUT, DELETE	POST (SOAP), GET (REST-like)	POST
Schema	Opzionale (OpenAPI)	WSDL obbligatorio	Schema obbligatorio
Caching	HTTP nativo	Complesso	Richiede libreria
Curva apprendimento	Bassa	Alta	Media
Over-fetching	Sì (fisso)	Sì	No (client decide)
Versioning	URI o header	Namespace	Schema evolution
Performance	Buona	Overhead XML	Ottima per client
Use case	Web API pubbliche, microservices	Enterprise, legacy	Mobile, dashboard complesse

Tabella 1.1: Confronto REST, SOAP, GraphQL

- Operazioni complesse non mappabili a risorse
- Real-time bi-direzionale (usare WebSocket)
- Query complesse con join (considerare GraphQL)
- Transazioni distribuite ACID (considerare SOAP o gRPC)

1.6 Best Practices REST

Best Practices

1. **Nouns, not verbs**: `/users` non `/getUsers`
2. **Plural names**: `/products` non `/product`
3. **HTTP methods semantici**: GET per lettura, POST per creazione
4. **Status codes corretti**: 201 per creazione, 404 per non trovato
5. **Versioning**: `/api/v1/users`
6. **Filtering**: `/users?status=active&role=admin`
7. **Pagination**: `/users?page=2&limit=50`
8. **HATEOAS**: Include link per navigazione
9. **Stateless**: Ogni richiesta self-contained
10. **Security**: HTTPS, authentication, rate limiting

Errori comuni

- Usare GET per operazioni che modificano stato
- URI con verbs: `/createUser`, `/deleteOrder`
- Ignorare status codes: tutto 200 OK
- Session-based auth invece di token

- Mancanza di versioning (breaking changes)
- Over-nesting: `/users/123/orders/456/items/789/reviews`
- Esporre implementazione interna in URI

1.7 Caso di studio: GitHub REST API v3

GitHub API è esempio eccellente di REST Level 2:

Listing 1.10: GitHub API: Get repository

```
GET /repos/octocat/Hello-World HTTP/1.1
Host: api.github.com
Accept: application/vnd.github.v3+json
Authorization: token ghp_abc123...

HTTP/1.1 200 OK
Content-Type: application/json
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4999

{
  "id": 1296269,
  "name": "Hello-World",
  "full_name": "octocat/Hello-World",
  "owner": {
    "login": "octocat",
    "id": 1,
    "url": "https://api.github.com/users/octocat"
  },
  "private": false,
  "description": "My first repository",
  "fork": false,
  "url": "https://api.github.com/repos/octocat/Hello-World",
  "forks_url": "https://api.github.com/repos/octocat/Hello-World/forks",
  "stargazers_count": 80,
  "watchers_count": 80,
  "language": "Python",
  "forks": 9,
  "open_issues": 0
}
```

Caratteristiche notevoli:

- URI resource-based: `/repos/:owner/:repo`
- Versioning in Accept header: `application/vnd.github.v3+json`
- Rate limiting headers: `X-RateLimit-*`
- Hypermedia partial: URL reference in response
- Pagination via Link header (RFC 5988)
- Authentication: token-based

1.8 Riepilogo

- REST è stile architetturale con 6 vincoli: client-server, stateless, cacheable, layered, uniform interface, code on demand
- Richardson Maturity Model valuta API da Level 0 (POX) a Level 3 (HATEOAS)
- SOAP: protocollo enterprise XML-based, verbose
- GraphQL: query language, client-driven, single endpoint
- REST: ideale per API pubbliche, CRUD, microservices
- Best practices: nouns not verbs, HTTP semantic, status codes, versioning, stateless

Esercizi

1. Valuta la tua API aziendale: a che livello Richardson si trova?
2. Progetta URI structure per blog API: posts, comments, users
3. Confronta overhead payload SOAP vs REST vs GraphQL per stessa operazione
4. Implementa client che consuma GitHub API v3
5. Identifica violazioni REST in API pubbliche (Spotify, Facebook)
6. Progetta HATEOAS completo per e-commerce API

Riferimenti

- Roy Fielding PhD dissertation: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- Richardson Maturity Model: https://martinfowler.com/articles/richardsonMaturityModel.html
- RFC 7231 HTTP/1.1 Semantics: https://tools.ietf.org/html/rfc7231
- GitHub REST API: https://docs.github.com/en/rest
- REST API Tutorial: https://restfulapi.net/

Capitolo 2

Metodi HTTP

Mappa del capitolo

Metodi HTTP principali, GET, POST, PUT, PATCH, DELETE, OPTIONS, HEAD, Idempotenza, Safety, Request/Response completi, Esempi pratici, Diagrammi flow, When to use, Best practices, Errori comuni.

Obiettivi di apprendimento

- Comprendere semantica di ogni metodo HTTP
- Applicare correttamente GET, POST, PUT, PATCH, DELETE
- Distinguere tra safe e unsafe methods
- Comprendere idempotenza e sue implicazioni
- Scegliere metodo appropriato per ogni operazione
- Evitare errori comuni (GET per modifiche, POST per tutto)

2.1 Panoramica Metodi HTTP

I metodi HTTP (o **verbs**) definiscono l'**azione** da eseguire su una risorsa. RFC 7231 definisce 9 metodi, di cui 5 fondamentali per REST:

Metodo	Operazione	Safe	Idempotent	Uso REST
GET	Leggi	Sì	Sì	Recupera risorsa
POST	Crea	No	No	Crea nuova risorsa
PUT	Sostituisci	No	Sì	Aggiorna/sostituisce
PATCH	Modifica	No	No*	Modifica parziale
DELETE	Elimina	No	Sì	Rimuovi risorsa
HEAD	Metadata	Sì	Sì	Come GET senza body
OPTIONS	Capacità	Sì	Sì	Descrive opzioni

Tabella 2.1: Metodi HTTP principali

** PATCH può essere idempotente se implementato correttamente*

2.2 Proprietà dei Metodi

2.2.1 Safe Methods

Un metodo è **safe** se non modifica lo stato del server. Operazioni di sola lettura.

Safe methods: GET, HEAD, OPTIONS, TRACE

Definizione Safe

Un metodo è safe se la sua semantica è **essenzialmente read-only**. Il client non richiede né si aspetta modifiche di stato sul server.

Implicazioni:

- Crawler e bot possono chiamare safe methods senza preoccupazioni
- Prefetching e caching sicuri
- Nessun side effect atteso

Nota: "Safe" è semantico, non garantisce che server non cambi (es: logging, analytics).

2.2.2 Idempotenza

Un metodo è **idempotente** se **N richieste identiche** hanno lo **stesso effetto** di una singola richiesta.

Idempotent methods: GET, PUT, DELETE, HEAD, OPTIONS

Definizione Idempotenza

Formalmente: $f(f(x)) = f(x)$
Applicando operazione idempotente multiple volte, lo stato finale è uguale ad applicarla una volta.

Esempio idempotenza:

Listing 2.1: PUT è idempotente

```
# Prima richiesta
PUT /api/users/123 HTTP/1.1
Content-Type: application/json

{"name": "Mario Rossi", "email": "mario@example.com"}

# HTTP/1.1 200 OK
# User 123 ora ha name="Mario Rossi"

# Seconda richiesta identica
PUT /api/users/123 HTTP/1.1
Content-Type: application/json

{"name": "Mario Rossi", "email": "mario@example.com"}

# HTTP/1.1 200 OK
# User 123 ha ANCORA name="Mario Rossi" (stesso stato)
```

Esempio NON idempotente:

Listing 2.2: POST NON è idempotente

```
# Prima richiesta
POST /api/users HTTP/1.1
Content-Type: application/json

{"name": "Mario Rossi"}

# HTTP/1.1 201 Created
# Location: /api/users/123
# Creato user con ID 123

# Seconda richiesta identica
POST /api/users HTTP/1.1
Content-Type: application/json

{"name": "Mario Rossi"}

# HTTP/1.1 201 Created
# Location: /api/users/124
# Creato ALTRO user con ID 124 (stato diverso!)
```

Perché idempotenza è importante:

- **Retry safety**: Se richiesta fallisce per network error, client può ritentare
- **Distributed systems**: Garantisce consistency in caso di duplicazione messaggi
- **Predictability**: Comportamento prevedibile per client

2.3 GET - Read

2.3.1 Semantica GET

Scopo: Recuperare rappresentazione di una risorsa.

Caratteristiche:

- Safe: sola lettura
- Idempotent: multiple GET → stesso risultato
- Cacheable: response può essere cached
- NO request body (parametri in query string)

2.3.2 Esempio GET - Singola Risorsa

Listing 2.3: GET singolo user

```
GET /api/v1/users/123 HTTP/1.1
Host: api.example.com
Accept: application/json
Authorization: Bearer eyJhbGc...

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=300
ETag: "abc123"

```

```
{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario.rossi@example.com",
  "role": "admin",
  "created_at": "2025-01-15T10:30:00Z"
}
```

2.3.3 Esempio GET - Collection

Listing 2.4: GET lista users con filtering

```
GET /api/v1/users?role=admin&status=active&page=1&limit=10 HTTP/1.1
Host: api.example.com
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json
X-Total-Count: 42
Link: </api/v1/users?page=2&limit=10>; rel="next"

{
  "data": [
    {
      "id": 123,
      "name": "Mario Rossi",
      "email": "mario@example.com"
    },
    {
      "id": 124,
      "name": "Luigi Verdi",
      "email": "luigi@example.com"
    }
  ],
  "pagination": {
    "page": 1,
    "limit": 10,
    "total": 42,
    "pages": 5
  }
}
```

2.3.4 GET con Conditional Request

Listing 2.5: GET con ETag per evitare transfer non necessari

```
# Prima richiesta
GET /api/v1/users/123 HTTP/1.1

HTTP/1.1 200 OK
ETag: "33a64df551425fcc"
Content-Type: application/json

{"id": 123, "name": "Mario"}

# Seconda richiesta con If-None-Match
```

```
GET /api/v1/users/123 HTTP/1.1
If-None-Match: "33a64df551425fcc"

HTTP/1.1 304 Not Modified
# Nessun body, client usa cache locale
```

2.3.5 Quando usare GET

Usare GET per:

- Recuperare singola risorsa: `GET /users/123`
- Recuperare collection: `GET /users`
- Search/filtering: `GET /products?category=electronics`
- Operazioni read-only

NON usare GET per:

- Modificare stato server (usare POST/PUT/PATCH/DELETE)
- Operazioni sensibili in URL (password, token visibili in logs)
- Request body (usare POST se dati complessi)

Errore comune

MAI fare:

```
GET /api/users/123/delete  # NO! Violazione safe property
GET /api/users/create?name=Mario  # NO! GET non crea risorse
```

Crawler e proxy possono pre-fetch GET, causando eliminazioni/creazioni accidentali!

2.4 POST - Create

2.4.1 Semantica POST

Scopo: Creare nuova risorsa o eseguire operazione complessa.

Caratteristiche:

- NOT safe: modifica stato
- NOT idempotent: multiple POST → risorse multiple
- Request body obbligatorio (payload)
- Server assegna ID alla nuova risorsa

2.4.2 Esempio POST - Creazione Risorsa

Listing 2.6: POST crea nuovo user

```
POST /api/v1/users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer eyJhbGc...

```

```
{
  "name": "Mario Rossi",
  "email": "mario.rossi@example.com",
  "role": "user"
}

HTTP/1.1 201 Created
Location: /api/v1/users/125
Content-Type: application/json

{
  "id": 125,
  "name": "Mario Rossi",
  "email": "mario.rossi@example.com",
  "role": "user",
  "created_at": "2025-11-15T14:22:00Z"
}
```

Elementi chiave:

- Status: **201 Created**
- Header **Location**: URI della risorsa creata
- Response body: rappresentazione completa risorsa creata

2.4.3 POST per Operazioni Complesse

POST può essere usato per operazioni non mappabili a CRUD:

Listing 2.7: POST per operazione custom (search complessa)

```
POST /api/v1/users/search HTTP/1.1
Content-Type: application/json

{
  "filters": {
    "age": {"min": 18, "max": 65},
    "roles": ["admin", "moderator"],
    "location": {
      "country": "IT",
      "city": "Milano"
    }
  },
  "sort": [
    {"field": "created_at", "order": "desc"}
  ],
  "pagination": {
    "page": 1,
    "limit": 20
  }
}

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

{
  "results": [...],
  "total": 152
}
```

Nota: Query complessa in POST body permette strutture che non stanno in query string.

2.4.4 POST per Sub-risorse

Listing 2.8: POST crea comment su post

```
POST /api/v1/posts/456/comments HTTP/1.1
Content-Type: application/json

{
  "author": "Mario",
  "text": "Great article!"
}

HTTP/1.1 201 Created
Location: /api/v1/posts/456/comments/789
Content-Type: application/json

{
  "id": 789,
  "post_id": 456,
  "author": "Mario",
  "text": "Great article!",
  "created_at": "2025-11-15T14:25:00Z"
}
```

2.4.5 Quando usare POST

Usare POST per:

- Creare nuova risorsa: `POST /users`
- Creare sub-risorsa: `POST /users/123/orders`
- Operazioni complesse: `POST /users/search`
- Triggering actions: `POST /orders/123/cancel`
- File upload

2.5 PUT - Update/Replace

2.5.1 Semantica PUT

Scopo: Sostituire interamente una risorsa esistente.
Caratteristiche:

- NOT safe: modifica stato
- Idempotent: PUT ripetuto → stesso stato
- Request body contiene rappresentazione completa
- Client specifica URI della risorsa

2.5.2 PUT vs POST

Differenza chiave: Chi decide l'URI?

- **POST**: Server decide URI (assegna ID)
- **PUT**: Client specifica URI completo

2.5.3 Esempio PUT - Replace completo

Listing 2.9: PUT sostituisce user completamente

```
PUT /api/v1/users/123 HTTP/1.1
Content-Type: application/json
Authorization: Bearer eyJhbGc...

{
  "name": "Mario Rossi Updated",
  "email": "mario.updated@example.com",
  "role": "admin",
  "phone": "+39 123 456 7890"
}

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

{
  "id": 123,
  "name": "Mario Rossi Updated",
  "email": "mario.updated@example.com",
  "role": "admin",
  "phone": "+39 123 456 7890",
  "updated_at": "2025-11-15T14:30:00Z"
}
```

Importante: PUT sostituisce **intera** risorsa. Campi omessi vengono rimossi/resettati.

2.5.4 PUT Idempotente - Esempio

Listing 2.10: PUT ripetuto ha stesso effetto

```
# Prima richiesta
PUT /api/v1/users/123 HTTP/1.1
Content-Type: application/json

{"name": "Mario", "email": "mario@example.com"}

# User 123: name=Mario, email=mario@example.com

# Seconda richiesta IDENTICA (network retry)
PUT /api/v1/users/123 HTTP/1.1
Content-Type: application/json

{"name": "Mario", "email": "mario@example.com"}

# User 123: ANCORA name=Mario, email=mario@example.com
# Stato identico, nessuna duplicazione
```

2.5.5 PUT per Upsert

PUT può creare risorsa se non esiste (upsert pattern):

Listing 2.11: PUT upsert - crea se non esiste

```
PUT /api/v1/users/999 HTTP/1.1
Content-Type: application/json

{"name": "Nuovo User", "email": "nuovo@example.com"}

# Se user 999 non esiste
HTTP/1.1 201 Created
Location: /api/v1/users/999

# Se user 999 già esiste
HTTP/1.1 200 OK
```

Nota: Pattern upsert non sempre consigliato (preferire POST per create, PUT per update).

2.6 PATCH - Partial Update

2.6.1 Semantica PATCH

Scopo: Modificare **parzialmente** una risorsa esistente.
Caratteristiche:

- NOT safe: modifica stato
- Può essere idempotent (dipende da implementazione)
- Request body: solo campi da modificare
- Standard: JSON Patch (RFC 6902) o JSON Merge Patch (RFC 7396)

2.6.2 PATCH vs PUT

- **PUT**: Sostituisce intera risorsa (campi omessi → rimossi)
- **PATCH**: Modifica solo campi specificati (altri invariati)

2.6.3 Esempio PATCH - Simple Merge

Listing 2.12: PATCH modifica solo email

```
PATCH /api/v1/users/123 HTTP/1.1
Content-Type: application/json

{
  "email": "mario.new@example.com"
}

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

{
  "id": 123,
  "name": "Mario Rossi",  // Invariato
  "email": "mario.new@example.com",  // Modificato
```

```
  "role": "admin",  // Invariato
  "updated_at": "2025-11-15T14:35:00Z"
}
```

2.6.4 JSON Patch (RFC 6902)

Formato standardizzato per patch complesse:

Listing 2.13: PATCH con JSON Patch RFC 6902

```
PATCH /api/v1/users/123 HTTP/1.1
Content-Type: application/json-patch+json

[
  { "op": "replace", "path": "/email", "value": "new@example.com" },
  { "op": "add", "path": "/phone", "value": "+39 123 456" },
  { "op": "remove", "path": "/temporary_field" }
]

HTTP/1.1 200 OK
```

Operazioni JSON Patch:

- `add`: Aggiungi campo
- `remove`: Rimuovi campo
- `replace`: Sostituisci valore
- `move`: Sposta valore
- `copy`: Copia valore
- `test`: Verifica valore (condizionale)

2.6.5 Quando usare PATCH

Usare PATCH per:

- Modificare pochi campi di risorsa grande
- Bandwidth optimization (mobile)
- Partial updates atomiche

Usare PUT invece:

- Update completo più semplice da implementare
- Tutti i campi sempre inviati (form completo)

2.7 DELETE - Remove

2.7.1 Semantica DELETE

Scopo: Eliminare una risorsa.

Caratteristiche:

- NOT safe: modifica stato

- Idempotent: DELETE ripetuto → stesso stato (risorsa assente)
- NO request body (solitamente)
- Response può essere empty (204) o con dettagli (200)

2.7.2 Esempio DELETE - Success

Listing 2.14: DELETE risorsa con successo

```
DELETE /api/v1/users/123 HTTP/1.1
Authorization: Bearer eyJhbGc...

HTTP/1.1 204 No Content
# Nessun body, risorsa eliminata
```

Alternative response:

Listing 2.15: DELETE con response body

```
DELETE /api/v1/users/123 HTTP/1.1

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

{
  "message": "User deleted successfully",
  "deleted_id": 123,
  "deleted_at": "2025-11-15T14:40:00Z"
}
```

2.7.3 DELETE Idempotente

Listing 2.16: DELETE ripetuto è idempotente

```
# Prima richiesta
DELETE /api/v1/users/123 HTTP/1.1

HTTP/1.1 204 No Content
# Risorsa eliminata

# Seconda richiesta (retry)
DELETE /api/v1/users/123 HTTP/1.1

# Opzione 1: Considerare successo (idempotenza)
HTTP/1.1 204 No Content

# Opzione 2: Indicare risorsa già assente
HTTP/1.1 404 Not Found
{
  "error": "User not found"
}
```

Debate: 204 o 404 per DELETE su risorsa già assente?

- **204 (idempotente)**: Stato finale identico (risorsa assente)
- **404 (preciso)**: Risorsa non esiste, impossibile eliminarla

Best practice: **204** per mantenere idempotenza.

2.7.4 Soft Delete vs Hard Delete

Hard Delete: Risorsa fisicamente rimossa da database

Listing 2.17: Hard delete - risorsa rimossa permanentemente

```
DELETE /api/v1/users/123 HTTP/1.1

HTTP/1.1 204 No Content
# User completamente rimosso dal database
```

Soft Delete: Risorsa marcata come deleted (flag)

Listing 2.18: Soft delete - risorsa marcata come eliminata

```
DELETE /api/v1/users/123 HTTP/1.1

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

{
  "id": 123,
  "name": "Mario",
  "deleted": true,
  "deleted_at": "2025-11-15T14:45:00Z"
}

# GET successivo
GET /api/v1/users/123 HTTP/1.1

HTTP/1.1 410 Gone
# Risorsa esiste ma è stata eliminata (soft delete)
```

2.8 HEAD - Metadata Only

Scopo: Come GET ma solo headers, senza body.

Uso:

- Verificare esistenza risorsa
- Controllare dimensione (Content-Length)
- Validare cache (ETag, Last-Modified)

Listing 2.19: HEAD per verificare risorsa

```
HEAD /api/v1/users/123 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 245
ETag: "abc123"
Last-Modified: Wed, 15 Nov 2025 10:00:00 GMT
# Nessun body
```

2.9 OPTIONS - Capabilities

Scopo: Descrivere metodi HTTP supportati dalla risorsa.
Uso principale: CORS preflight requests

Listing 2.20: OPTIONS per CORS

```
OPTIONS /api/v1/users/123 HTTP/1.1
Origin: https://example.com
Access-Control-Request-Method: DELETE

HTTP/1.1 204 No Content
Allow: GET, PUT, PATCH, DELETE, OPTIONS
Access-Control-Allow-Origin: https://example.com
Access-Control-Allow-Methods: GET, PUT, PATCH, DELETE
Access-Control-Max-Age: 86400
```

2.10 Diagramma HTTP Request/Response Flow

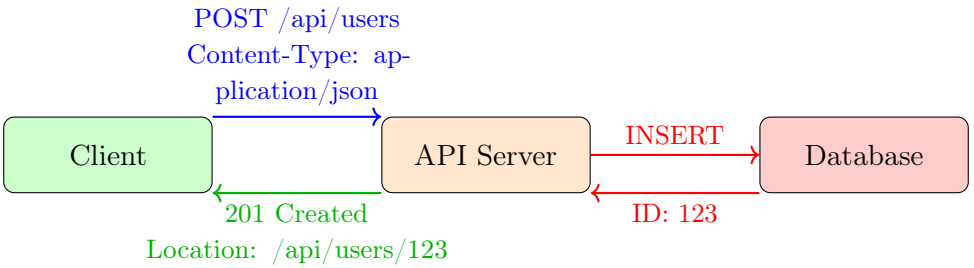


Figura 2.1: HTTP Flow: POST create user

2.11 Best Practices

Best Practices HTTP Methods

1. **GET per read-only**: Mai modificare stato con GET
2. **POST per create**: Lascia server assegnare ID
3. **PUT per replace**: Invia rappresentazione completa
4. **PATCH per partial**: Modifica solo campi necessari
5. **DELETE è idempotente**: 204 anche se già eliminato
6. **Status codes corretti**: 201 per POST, 204 per DELETE
7. **Location header**: Sempre in 201 Created
8. **Safe methods cacheable**: Implementa ETag per GET
9. **Idempotenza per retry**: PUT/DELETE sicuri da ritentare
10. **Validation**: Valida input prima di processare

Errori comuni

- **GET per delete**: `GET /users/123/delete` (PERICOLOSO!)
- **POST per tutto**: Ignora semantica altri metodi
- **PUT senza idempotenza**: Incrementi in PUT violano idempotenza
- **PATCH completo**: Se modifichi tutto, usa PUT
- **DELETE con body**: Semanticamente scorretto
- **Status codes sbagliati**: 200 OK per DELETE invece di 204
- **Mancanza Location**: POST 201 senza Location header

2.12 Riepilogo

Metodo	CRUD	Safe	Idempotent	Status Success
GET	Read	Sì	Sì	200 OK
POST	Create	No	No	201 Created
PUT	Update/Replace	No	Sì	200 OK / 204 No Content
PATCH	Update Partial	No	No*	200 OK
DELETE	Delete	No	Sì	204 No Content / 200 OK
HEAD	Read metadata	Sì	Sì	200 OK (no body)
OPTIONS	Metadata	Sì	Sì	204 No Content

Tabella 2.2: Riepilogo metodi HTTP

Esercizi

1. Implementa endpoint CRUD completo per risorsa `/products`
2. Progetta API per blog: POST create, PATCH update, DELETE
3. Implementa idempotenza: gestisci retry duplicati con idempotency-key
4. Test ETag caching: verifica 304 Not Modified
5. Implementa JSON Patch RFC 6902 per update complessi
6. Confronta bandwidth: PUT vs PATCH per update singolo campo

Riferimenti

- RFC 7231 - HTTP/1.1 Semantics and Content: https://tools.ietf.org/html/rfc7231
- RFC 5789 - PATCH Method: https://tools.ietf.org/html/rfc5789
- RFC 6902 - JSON Patch: https://tools.ietf.org/html/rfc6902
- RFC 7396 - JSON Merge Patch: https://tools.ietf.org/html/rfc7396

Capitolo 3

HTTP Status Codes

Mappa del capitolo

Status codes overview, 1xx Informational, 2xx Success, 3xx Redirection, 4xx Client Errors, 5xx Server Errors, Quando usare ogni codice, Best practices, Esempi pratici completi, Error response structure, Diagrammi decision tree.

Obiettivi di apprendimento

- Comprendere semantica di ogni categoria status code
- Scegliere status code appropriato per ogni scenario
- Implementare error handling standardizzato
- Distinguere 4xx (client error) da 5xx (server error)
- Costruire response body informativi per errori
- Evitare anti-pattern (tutto 200 OK, generic 500)

3.1 Panoramica Status Codes

HTTP status codes comunicano l'**esito** di una richiesta. Definiti in RFC 7231, sono divisi in 5 categorie:

Range	Categoria	Significato
1xx	Informational	Richiesta ricevuta, processamento in corso
2xx	Success	Richiesta ricevuta, compresa ed accettata
3xx	Redirection	Client deve fare azione aggiuntiva
4xx	Client Error	Richiesta contiene errore del client
5xx	Server Error	Server ha fallito richiesta valida

Tabella 3.1: Categorie HTTP Status Codes

Principio chiave

Status code giusto migliora:

- **Debugging**: Client sa subito se errore è suo (4xx) o del server (5xx)
- **Retry logic**: Client può ritentare 5xx, non 4xx
- **Caching**: 2xx/3xx cacheable, 4xx/5xx no
- **Monitoring**: Alert diversi per 4xx vs 5xx

3.2 1xx - Informational

Status codes informativi indicano che richiesta è stata ricevuta e processamento continua. **Raramente usati in REST API**.

3.2.1 100 Continue

Uso: Client chiede conferma prima di inviare request body grande.

Listing 3.1: 100 Continue - Upload file grande

```
# Client invia headers
POST /api/v1/uploads HTTP/1.1
Host: api.example.com
Content-Length: 104857600
Expect: 100-continue

# Server risponde
HTTP/1.1 100 Continue

# Client invia body (100MB)
[...binary data...]

# Server risponde dopo processamento
HTTP/1.1 201 Created
Location: /api/v1/uploads/abc123
```

Quando usare:

- Upload file molto grandi
- Evitare invio body se server rifiuterà (es: auth fail)

3.2.2 101 Switching Protocols

Uso: Server accetta richiesta client di cambiare protocollo (es: HTTP → WebSocket).

Listing 3.2: 101 - WebSocket upgrade

```
GET /chat HTTP/1.1
Host: api.example.com
Upgrade: websocket
Connection: Upgrade

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

Quando usare: WebSocket handshake, HTTP/2 upgrade.

3.2.3 102 Processing (WebDAV)

Uso: Server ha ricevuto richiesta e sta processando, ma nessuna response ancora disponibile.
Nota: Rarissimo in REST API moderne.

3.3 2xx - Success

Richiesta ricevuta, compresa e processata con successo.

3.3.1 200 OK

Uso: Richiesta generica di successo.
Quando usare:

- GET: risorsa trovata e ritornata
- PUT: risorsa aggiornata
- PATCH: risorsa modificata parzialmente
- POST: operazione custom completata

Listing 3.3: 200 OK - GET success

```
GET /api/v1/users/123 HTTP/1.1

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

{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com"
}
```

Listing 3.4: 200 OK - PUT success con body

```
PUT /api/v1/users/123 HTTP/1.1
Content-Type: application/json

{"name": "Mario Updated"}

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

{
  "id": 123,
  "name": "Mario Updated",
  "updated_at": "2025-11-15T15:00:00Z"
}
```

3.3.2 201 Created

Uso: Nuova risorsa creata con successo.
Elementi obbligatori:

- Header `Location`: URI della risorsa creata

- Response body: rappresentazione risorsa creata (opzionale ma consigliato)

Listing 3.5: 201 Created - POST crea user

```
POST /api/v1/users HTTP/1.1
Content-Type: application/json

{
  "name": "Luigi Verdi",
  "email": "luigi@example.com"
}

HTTP/1.1 201 Created
Location: /api/v1/users/456
Content-Type: application/json

{
  "id": 456,
  "name": "Luigi Verdi",
  "email": "luigi@example.com",
  "created_at": "2025-11-15T15:05:00Z"
}
```

Quando usare:

- POST crea nuova risorsa
- PUT crea risorsa (upsert pattern)

3.3.3 202 Accepted

Uso: Richiesta accettata per processamento, ma non completata.
Scenario tipico: Operazioni asincrone/long-running.

Listing 3.6: 202 Accepted - Job asincrono

```
POST /api/v1/exports HTTP/1.1
Content-Type: application/json

{
  "format": "csv",
  "filters": {...}
}

HTTP/1.1 202 Accepted
Location: /api/v1/jobs/789
Content-Type: application/json

{
  "job_id": 789,
  "status": "pending",
  "created_at": "2025-11-15T15:10:00Z",
  "_links": {
    "self": "/api/v1/jobs/789",
    "status": "/api/v1/jobs/789/status"
  }
}
```

Client poi polling su `/api/v1/jobs/789/status` per verificare completamento.
Quando usare:

- Processamento batch
- Export/report generation
- Video transcoding
- Email sending in background

3.3.4 204 No Content

Uso: Richiesta processata con successo, ma nessun contenuto da ritornare.
Caratteristiche:

- **NO response body**
- Headers presenti (es: metadata)
- Bandwidth optimization

Listing 3.7: 204 No Content - DELETE success

```
DELETE /api/v1/users/123 HTTP/1.1

HTTP/1.1 204 No Content
```

Listing 3.8: 204 No Content - PUT success senza body

```
PUT /api/v1/users/123 HTTP/1.1
Content-Type: application/json

{"name": "Mario Updated"}

HTTP/1.1 204 No Content
# Nessun body, ma update avvenuto
```

Quando usare:

- DELETE: risorsa eliminata
- PUT/PATCH: aggiornamento senza necessità di ritornare risorsa
- POST: operazione completata senza output

3.3.5 206 Partial Content

Uso: Server ritorna solo parte del contenuto richiesto (Range requests).

Listing 3.9: 206 Partial Content - Range request

```
GET /api/v1/files/video.mp4 HTTP/1.1
Range: bytes=0-1023

HTTP/1.1 206 Partial Content
Content-Range: bytes 0-1023/10485760
Content-Length: 1024
Content-Type: video/mp4

[...primi 1024 bytes del file...]
```

Quando usare:

- Download resumable
- Video streaming
- Large file transfer

3.4 3xx - Redirection

Client deve fare azione aggiuntiva per completare richiesta.

3.4.1 301 Moved Permanently

Uso: Risorsa spostata permanentemente a nuovo URI.

Listing 3.10: 301 - Risorsa spostata permanentemente

```
GET /api/v1/users/123 HTTP/1.1

HTTP/1.1 301 Moved Permanently
Location: /api/v2/users/123
Content-Type: application/json

{
  "message": "Resource moved to /api/v2/users/123"
}
```

Implicazioni:

- Browser/client aggiornano bookmark
- Search engine aggiornano indice
- Cache permanente del redirect

Quando usare: API versioning, ristrutturazione URI permanente.

3.4.2 302 Found (Temporary Redirect)

Uso: Risorsa temporaneamente disponibile a diverso URI.

Listing 3.11: 302 - Redirect temporaneo

```
GET /api/v1/users/current HTTP/1.1

HTTP/1.1 302 Found
Location: /api/v1/users/123
```

Quando usare: Maintenance mode, load balancing, temporary routing.

3.4.3 303 See Other

Uso: Response disponibile a diverso URI via GET.
Pattern PRG (Post-Redirect-Get):

Listing 3.12: 303 - POST-Redirect-GET pattern

```
POST /api/v1/users HTTP/1.1
Content-Type: application/json

{"name": "Mario"}
```

```
5
6  HTTP/1.1 303 See Other
7  Location: /api/v1/users/789
8
9  # Client fa GET automatico
10 GET /api/v1/users/789 HTTP/1.1
11
12 HTTP/1.1 200 OK
13 Content-Type: application/json
14
15 {"id": 789, "name": "Mario"}
```

3.4.4 304 Not Modified

Uso: Risorsa non modificata da ultima richiesta (cache validation).

Listing 3.13: 304 - Cache validation con ETag

```
1  # Prima richiesta
2  GET /api/v1/users/123 HTTP/1.1
3
4  HTTP/1.1 200 OK
5  ETag: "abc123"
6  Content-Type: application/json
7
8  {"id": 123, "name": "Mario"}
9
10 # Seconda richiesta con ETag
11 GET /api/v1/users/123 HTTP/1.1
12 If-None-Match: "abc123"
13
14 HTTP/1.1 304 Not Modified
15 ETag: "abc123"
16 # NESSUN body, client usa cache
```

Benefici:

- Bandwidth saving
- Faster response (no body)
- Cache optimization

3.4.5 307 Temporary Redirect

Uso: Come 302, ma garantisce che metodo HTTP non cambi.

Listing 3.14: 307 - Method preserved

```
1  POST /api/v1/process HTTP/1.1
2
3  HTTP/1.1 307 Temporary Redirect
4  Location: /api/v1/process-new
5
6  # Client ripete POST (non GET) a nuovo URI
```

3.4.6 308 Permanent Redirect

Uso: Come 301, ma garantisce che metodo HTTP non cambi.

3.5 4xx - Client Errors

Richiesta contiene errore del client (sintassi, autorizzazione, input invalido).

3.5.1 400 Bad Request

Uso: Richiesta malformata, sintassi errata.

Listing 3.15: 400 - JSON malformato

```
POST /api/v1/users HTTP/1.1
Content-Type: application/json

{
  "name": "Mario",
  "email": "invalid-json-missing-brace"

HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": "bad_request",
  "message": "Invalid JSON syntax",
  "details": "Expected '}' at line 3, column 45"
}
```

Listing 3.16: 400 - Validation error

```
POST /api/v1/users HTTP/1.1
Content-Type: application/json

{
  "name": "",
  "email": "not-an-email"
}

HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": "validation_failed",
  "message": "Request validation failed",
  "errors": [
    {
      "field": "name",
      "message": "Name is required"
    },
    {
      "field": "email",
      "message": "Invalid email format"
    }
  ]
}
```

Quando usare:

- JSON/XML malformato
- Validation errors

- Parametri obbligatori mancanti
- Input invalido (tipo errato, range fuori)

3.5.2 401 Unauthorized

Uso: Autenticazione richiesta ma non fornita o invalida.

Nota naming confusa: Dovrebbe chiamarsi "Unauthenticated" (manca auth, non "non autorizzato").

Listing 3.17: 401 - Missing authentication

```
GET /api/v1/users HTTP/1.1

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="api"
Content-Type: application/json

{
  "error": "unauthorized",
  "message": "Authentication required",
  "details": "Missing Authorization header"
}
```

Listing 3.18: 401 - Invalid token

```
GET /api/v1/users HTTP/1.1
Authorization: Bearer invalid-token-xyz

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="api", error="invalid_token"
Content-Type: application/json

{
  "error": "invalid_token",
  "message": "Access token is invalid or expired"
}
```

Quando usare:

- Manca header Authorization
- Token scaduto/invalido
- Credenziali errate

3.5.3 403 Forbidden

Uso: Client autenticato ma NON autorizzato per questa risorsa.

Listing 3.19: 403 - Insufficient permissions

```
DELETE /api/v1/users/999 HTTP/1.1
Authorization: Bearer valid-token-user-role

HTTP/1.1 403 Forbidden
Content-Type: application/json

{
  "error": "forbidden",
```

```
  "message": "Insufficient permissions",
  "details": "Only admin users can delete users",
  "required_role": "admin",
  "current_role": "user"
}
```

Differenza 401 vs 403:

- **401**: "Chi sei?" (authentication)
- **403**: "Ti conosco, ma non puoi fare questo" (authorization)

Listing 3.20: 401 vs 403 - Flow decisionale

```
# Nessun token
GET /api/users/123
-> 401 Unauthorized (manca autenticazione)

# Token valido, ma risorsa di altro user
GET /api/users/999
Authorization: Bearer user-123-token
-> 403 Forbidden (autenticato ma non autorizzato)
```

3.5.4 404 Not Found

Uso: Risorsa non esiste.

Listing 3.21: 404 - Risorsa non trovata

```
GET /api/v1/users/99999 HTTP/1.1

HTTP/1.1 404 Not Found
Content-Type: application/json

{
  "error": "not_found",
  "message": "User not found",
  "resource_type": "User",
  "resource_id": 99999
}
```

Quando usare:

- GET su risorsa inesistente
- PUT/PATCH/DELETE su risorsa inesistente (se non si supporta upsert)
- Endpoint stesso non esiste: `/api/v1/nonexistent`

Security consideration: A volte 404 usato per nascondere esistenza risorsa protetta invece di 403.

3.5.5 405 Method Not Allowed

Uso: Metodo HTTP non supportato per questa risorsa.

Listing 3.22: 405 - Metodo non supportato

```
DELETE /api/v1/system/config HTTP/1.1

```

```
HTTP/1.1 405 Method Not Allowed
Allow: GET, PUT
Content-Type: application/json

{
  "error": "method_not_allowed",
  "message": "DELETE not allowed on this resource",
  "allowed_methods": ["GET", "PUT"]
}
```

Quando usare:

- Risorsa esiste ma metodo non supportato
- Es: DELETE su risorsa read-only

3.5.6 406 Not Acceptable

Uso: Server non può produrre response nel formato richiesto da Accept header.

Listing 3.23: 406 - Formato non supportato

```
GET /api/v1/users/123 HTTP/1.1
Accept: application/xml

HTTP/1.1 406 Not Acceptable
Content-Type: application/json

{
  "error": "not_acceptable",
  "message": "Requested format not supported",
  "requested": "application/xml",
  "available": ["application/json"]
}
```

3.5.7 408 Request Timeout

Uso: Client non ha completato richiesta entro timeout server.

Listing 3.24: 408 - Timeout upload

```
POST /api/v1/uploads HTTP/1.1
Content-Length: 104857600

# Client inizia upload ma si interrompe

HTTP/1.1 408 Request Timeout
Content-Type: application/json

{
  "error": "request_timeout",
  "message": "Request took too long to complete",
  "timeout_seconds": 30
}
```

3.5.8 409 Conflict

Uso: Conflitto con stato corrente risorsa.

Listing 3.25: 409 - Conflict email duplicata

```
POST /api/v1/users HTTP/1.1
Content-Type: application/json

{
  "name": "Mario",
  "email": "existing@example.com"
}

HTTP/1.1 409 Conflict
Content-Type: application/json

{
  "error": "conflict",
  "message": "User with this email already exists",
  "conflicting_field": "email",
  "conflicting_value": "existing@example.com"
}
```

Listing 3.26: 409 - Optimistic locking conflict

```
PUT /api/v1/users/123 HTTP/1.1
If-Match: "old-etag"
Content-Type: application/json

{"name": "Updated"}

HTTP/1.1 409 Conflict
ETag: "current-etag-different"
Content-Type: application/json

{
  "error": "conflict",
  "message": "Resource was modified by another request",
  "current_version": "current-etag-different",
  "your_version": "old-etag"
}
```

Quando usare:

- Unique constraint violation (email, username duplicato)
- Optimistic locking conflict
- Business logic conflict (es: cancellare order già spedito)

3.5.9 410 Gone

Uso: Risorsa esisteva ma è stata rimossa permanentemente.

Listing 3.27: 410 - Risorsa eliminata permanentemente

```
GET /api/v1/users/123 HTTP/1.1

HTTP/1.1 410 Gone
Content-Type: application/json

{
  "error": "gone",
```

```
  "message": "User was permanently deleted",
  "deleted_at": "2025-10-01T10:00:00Z"
}
```

Differenza 404 vs 410:

- **404**: Non esiste / mai esistita
- **410**: Esisteva ma ora eliminata permanentemente

3.5.10 413 Payload Too Large

Uso: Request body troppo grande.

Listing 3.28: 413 - File upload troppo grande

```
POST /api/v1/uploads HTTP/1.1
Content-Length: 104857600

[...100MB file...]

HTTP/1.1 413 Payload Too Large
Content-Type: application/json

{
  "error": "payload_too_large",
  "message": "File size exceeds limit",
  "max_size_bytes": 10485760,
  "your_size_bytes": 104857600
}
```

3.5.11 415 Unsupported Media Type

Uso: Content-Type non supportato.

Listing 3.29: 415 - Content-Type non supportato

```
POST /api/v1/users HTTP/1.1
Content-Type: application/xml

<user><name>Mario</name></user>

HTTP/1.1 415 Unsupported Media Type
Content-Type: application/json

{
  "error": "unsupported_media_type",
  "message": "Server does not support this Content-Type",
  "provided": "application/xml",
  "supported": ["application/json", "application/x-www-form-urlencoded"]
}
```

3.5.12 422 Unprocessable Entity

Uso: Request sintatticamente corretto ma semanticamente invalido.
Differenza 400 vs 422:

- **400**: Sintassi errata (JSON malformato)

- **422**: Sintassi OK ma logica invalida

Listing 3.30: 422 - Business logic validation error

```
POST /api/v1/orders HTTP/1.1
Content-Type: application/json

{
  "product_id": 123,
  "quantity": -5
}

HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
  "error": "unprocessable_entity",
  "message": "Request contains semantic errors",
  "errors": [
    {
      "field": "quantity",
      "value": -5,
      "message": "Quantity must be positive integer"
    }
  ]
}
```

3.5.13 429 Too Many Requests

Uso: Client ha superato rate limit.

Listing 3.31: 429 - Rate limit exceeded

```
GET /api/v1/users HTTP/1.1

HTTP/1.1 429 Too Many Requests
Retry-After: 60
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1700058000
Content-Type: application/json

{
  "error": "rate_limit_exceeded",
  "message": "Too many requests, slow down",
  "retry_after_seconds": 60,
  "limit": 1000,
  "window_seconds": 3600
}
```

3.6 5xx - Server Errors

Server ha fallito processare richiesta valida.

3.6.1 500 Internal Server Error

Uso: Errore generico server.

Listing 3.32: 500 - Errore generico server

```
GET /api/v1/users/123 HTTP/1.1

HTTP/1.1 500 Internal Server Error
Content-Type: application/json

{
  "error": "internal_server_error",
  "message": "An unexpected error occurred",
  "error_id": "err_abc123xyz",
  "timestamp": "2025-11-15T15:30:00Z"
}
```

Best practice:

- NON esporre stack trace in produzione
- Loggare dettagli server-side
- Fornire error_id per support tracking
- Monitorare e alertare su 500

3.6.2 501 Not Implemented

Uso: Server non supporta funzionalità richiesta.

Listing 3.33: 501 - Feature non implementata

```
PATCH /api/v1/users/123 HTTP/1.1

HTTP/1.1 501 Not Implemented
Content-Type: application/json

{
  "error": "not_implemented",
  "message": "PATCH method not yet implemented for this resource"
}
```

3.6.3 502 Bad Gateway

Uso: Server gateway/proxy ha ricevuto response invalida da upstream server.

Listing 3.34: 502 - Gateway error

```
GET /api/v1/users HTTP/1.1

HTTP/1.1 502 Bad Gateway
Content-Type: application/json

{
  "error": "bad_gateway",
  "message": "Upstream service returned invalid response",
  "upstream_service": "user-service"
}
```

Quando capita:

- API Gateway → Backend service down

- Load balancer → Application server crashed
- Microservice communication failure

3.6.4 503 Service Unavailable

Uso: Server temporaneamente non disponibile.

Listing 3.35: 503 - Maintenance mode

```
GET /api/v1/users HTTP/1.1

HTTP/1.1 503 Service Unavailable
Retry-After: 3600
Content-Type: application/json

{
  "error": "service_unavailable",
  "message": "API is temporarily unavailable for maintenance",
  "retry_after_seconds": 3600,
  "maintenance_end": "2025-11-15T18:00:00Z"
}
```

Quando usare:

- Maintenance programmata
- Server overload
- Database connection pool esaurito
- Circuit breaker open

3.6.5 504 Gateway Timeout

Uso: Gateway non ha ricevuto response da upstream entro timeout.

Listing 3.36: 504 - Gateway timeout

```
GET /api/v1/reports/heavy HTTP/1.1

HTTP/1.1 504 Gateway Timeout
Content-Type: application/json

{
  "error": "gateway_timeout",
  "message": "Upstream service did not respond in time",
  "timeout_seconds": 30,
  "upstream_service": "report-generator"
}
```

3.7 Diagramma Decision Tree

3.8 Error Response Structure

3.8.1 RFC 7807 - Problem Details

Standard per error response strutturato:

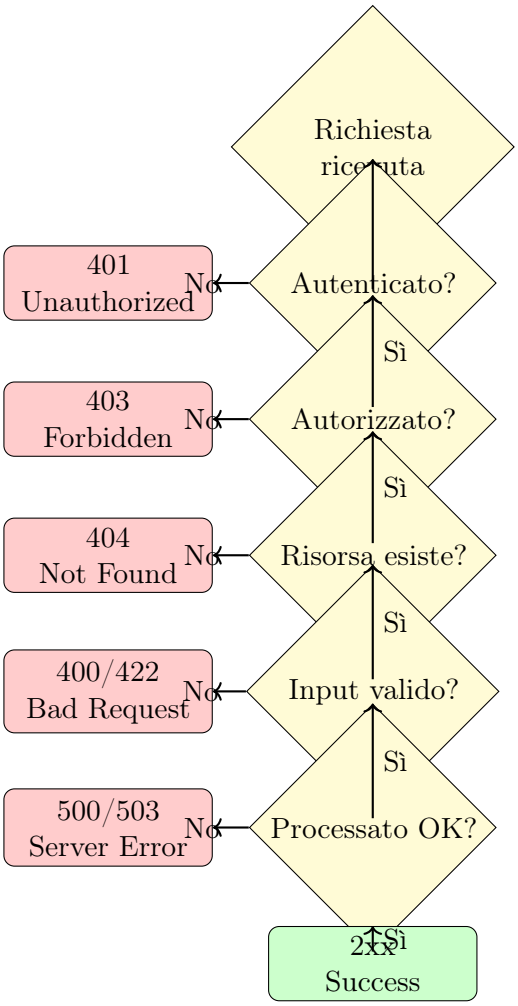


Figura 3.1: Decision tree per status code

Listing 3.37: RFC 7807 Problem Details

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

{
  "type": "https://api.example.com/errors/validation-failed",
  "title": "Validation Failed",
  "status": 400,
  "detail": "Request contains invalid fields",
  "instance": "/api/v1/users",
  "errors": [
    {
      "field": "email",
      "message": "Invalid email format"
    }
  ],
  "trace_id": "abc123xyz"
}
```

Campi standard:

- `type`: URI che identifica tipo errore
- `title`: Breve descrizione human-readable

- `status`: HTTP status code
- `detail`: Spiegazione specifica errore
- `instance`: URI richiesta che ha generato errore

3.9 Best Practices

Best Practices Status Codes

1. **Usa status code semantico**: Non tutto 200 OK
2. **4xx per client error**: Input validation, auth, not found
3. **5xx per server error**: Database down, exception
4. **201 + Location per create**: POST → 201 con header Location
5. **204 per success senza body**: DELETE, PUT senza response
6. **401 vs 403 correttamente**: Auth vs Authorization
7. **409 per conflict**: Unique constraint, optimistic locking
8. **422 per semantic error**: Logica invalida (non sintassi)
9. **429 per rate limit**: Include Retry-After header
10. **503 per maintenance**: Usa Retry-After header
11. **Error body strutturato**: RFC 7807 Problem Details
12. **Mai esporre stack trace**: Security risk
13. **Error ID per tracking**: Correlazione con logs

Errori comuni

- **Tutto 200 OK + error in body**: Anti-pattern
- **Generic 500 sempre**: Usa 502, 503, 504 appropriatamente
- **404 per auth failure**: Usare 401/403
- **200 per validation error**: Usare 400/422
- **POST → 200 invece di 201**: Manca semantica creazione
- **Stack trace in response**: Security issue
- **Error senza dettagli**: "Error occurred" inutile

3.10 Riepilogo

Esercizi

1. Implementa error handling completo con RFC 7807 Problem Details

Code	Nome	Quando usare
200	OK	GET/PUT/PATCH success con body
201	Created	POST crea risorsa (+ Location header)
202	Accepted	Async operation accettata
204	No Content	DELETE/PUT success senza body
301	Moved Permanently	Risorsa spostata permanentemente
304	Not Modified	Cache validation, risorsa invariata
400	Bad Request	Sintassi errata, validation error
401	Unauthorized	Manca/invalida autenticazione
403	Forbidden	Autenticato ma non autorizzato
404	Not Found	Risorsa non esiste
405	Method Not Allowed	Metodo HTTP non supportato
409	Conflict	Unique constraint, optimistic lock
410	Gone	Risorsa eliminata permanentemente
422	Unprocessable Entity	Logica business invalida
429	Too Many Requests	Rate limit superato
500	Internal Server Error	Errore generico server
502	Bad Gateway	Upstream service error
503	Service Unavailable	Maintenance, overload
504	Gateway Timeout	Upstream timeout

Tabella 3.2: Status codes principali - Quick reference

2. Crea middleware per catturare exception e mappare a status code corretto
3. Test suite: verifica ogni endpoint ritorna status code appropriato
4. Implementa rate limiting con 429 e Retry-After header
5. Progetta error response structure consistente per tutta API
6. Analizza API pubblica (GitHub, Stripe): catalogare uso status codes

Riferimenti

- RFC 7231 - HTTP/1.1 Status Codes: https://tools.ietf.org/html/rfc7231#section-6
- RFC 7807 - Problem Details: https://tools.ietf.org/html/rfc7807
- HTTP Status Dogs: https://httpstatusdogs.com/
- HTTP Cats: https://http.cat/
- MDN Status Codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Capitolo 4

Resource Design

Mappa del capitolo

URI design principles, Naming conventions, Resource modeling, Collection vs singular, Nesting resources, Filtering, Sorting, Searching, Pagination, Query parameters, Path parameters, Best practices, Anti-patterns, Esempi pratici completi.

Obiettivi di apprendimento

- Progettare URI structure chiara e consistente
- Applicare naming conventions REST standard
- Modellare risorse e relazioni
- Implementare filtering, sorting, pagination
- Scegliere tra nesting e flat structure
- Evitare anti-pattern comuni
- Costruire API navigabile e intuitiva

4.1 Principi URI Design

4.1.1 Cosa è una Risorsa

In REST, tutto è una **risorsa**: qualsiasi cosa identificabile e accessibile via URI.

Esempi di risorse:

- **Entity**: `/users/123`, `/products/456`
- **Collection**: `/users`, `/products`
- **Relationship**: `/users/123/orders`
- **Action result**: `/search`, `/reports`
- **Virtual resource**: `/users/current`, `/users/me`

4.1.2 Caratteristiche URI RESTful

URI ben progettato deve essere:

- **Intuitivo**: Developer capisce cosa rappresenta
- **Leggibile**: Human-readable, lowercase
- **Consistente**: Pattern uniforme in tutta API
- **Stable**: Non cambia (versioning per breaking changes)
- **Hierarchical**: Riflette relazioni tra risorse

URI non è implementazione

URI deve riflettere **modello logico** risorse, NON implementazione tecnica.
Cattivo: `/api/get_users.php?id=123`
Buono: `/api/users/123`
URI nasconde:

- Linguaggio backend (PHP, Java, Python)
- Struttura database (tabelle, query)
- File system structure

4.2 Naming Conventions

4.2.1 Regola 1: Nouns, Not Verbs

URI identifica **risorsa** (noun), non azione (verb). Azione è metodo HTTP.

Listing 4.1: BAD: Verbs in URI

```
GET /api/getUsers
GET /api/users/getUserById/123
POST /api/createUser
DELETE /api/deleteUser/123
```

Listing 4.2: GOOD: Nouns + HTTP methods

```
GET /api/users
GET /api/users/123
POST /api/users
DELETE /api/users/123
```

Mapping azione → metodo HTTP:

- `getUsers` → `GET /users`
- `createUser` → `POST /users`
- `updateUser` → `PUT /users/123`
- `deleteUser` → `DELETE /users/123`

4.2.2 Regola 2: Plural Names per Collections

Usa plural per collections:

Listing 4.3: Plural names consistency

```
GET /api/users           # Collection di users
GET /api/users/123       # Singolo user dalla collection

GET /api/products        # Collection di products
GET /api/products/456    # Singolo product
```

NON mixare singular/plural:

Listing 4.4: BAD: Mixed singular/plural

```
GET /api/user           # NO Inconsistente
GET /api/user/123

GET /api/users          # OK Consistente
GET /api/users/123
```

4.2.3 Regola 3: Lowercase con Hyphens

Standard: lowercase con hyphens (-) per multi-word.

Listing 4.5: Lowercase + hyphens

```
# GOOD
GET /api/order-items
GET /api/user-profiles
GET /api/shipping-addresses

# BAD
GET /api/OrderItems          # PascalCase
GET /api/order_items         # snake_case (usato in query params)
GET /api/orderItems          # camelCase
```

Nota: Query parameters usano snake_case per convenzione:

```
GET /api/users?created_after=2025-01-01&sort_by=name
```

4.2.4 Regola 4: Forward Slash per Gerarchia

/ indica relazione gerarchica:

Listing 4.6: Hierarchical structure

```
/organizations/123                      # Organization 123
/organizations/123/departments          # Departments of org 123
/organizations/123/departments/456      # Department 456 of org 123
/organizations/123/departments/456/employees  # Employees di dept 456
```

NO trailing slash:

```
GET /api/users/     # NO BAD
GET /api/users      # OK GOOD
```

4.2.5 Regola 5: File Extensions NOT Needed

Content negotiation via headers, NON extension:

Listing 4.7: Content negotiation

```
# BAD
GET /api/users/123.json
GET /api/users/123.xml

# GOOD
GET /api/users/123
Accept: application/json

GET /api/users/123
Accept: application/xml
```

4.3 Resource Modeling

4.3.1 Collection Resource

Collection: Insieme di risorse dello stesso tipo.

Listing 4.8: Collection resource

```
GET /api/v1/users HTTP/1.1

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

{
  "data": [
    {
      "id": 1,
      "name": "Mario Rossi",
      "email": "mario@example.com"
    },
    {
      "id": 2,
      "name": "Luigi Verdi",
      "email": "luigi@example.com"
    }
  ],
  "total": 2
}
```

Operazioni tipiche:

- `GET /users`: Lista tutti users
- `POST /users`: Crea nuovo user

4.3.2 Singular Resource

Singular: Risorsa singola identificata da ID.

Listing 4.9: Singular resource

```
GET /api/v1/users/123 HTTP/1.1

```

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

{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com",
  "role": "admin",
  "created_at": "2025-01-15T10:00:00Z"
}
```

Operazioni tipiche:

- `GET /users/123`: Recupera user
- `PUT /users/123`: Aggiorna user
- `PATCH /users/123`: Modifica parziale
- `DELETE /users/123`: Elimina user

4.3.3 Singleton Resource

Singleton: Risorsa unica, senza collection.

Listing 4.10: Singleton resource - current user

```
GET /api/v1/me HTTP/1.1
Authorization: Bearer token123

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

{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com"
}
```

Altri esempi singleton:

- `/api/status`: API health status
- `/api/config`: Configurazione corrente
- `/api/users/me`: Current authenticated user

4.4 Nested Resources

4.4.1 Quando Usare Nesting

Nesting modella relazione parent-child.

Usa nesting quando:

- Child esiste solo in contesto parent
- Child dipende logicamente da parent
- Operazioni su child necessitano parent context

Listing 4.11: Nested resource - Comments di un Post

```
# Get all comments di post 123
GET /api/posts/123/comments HTTP/1.1

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

{
  "data": [
    {
      "id": 1,
      "post_id": 123,
      "author": "Mario",
      "text": "Great post!",
      "created_at": "2025-11-15T10:00:00Z"
    },
    {
      "id": 2,
      "post_id": 123,
      "author": "Luigi",
      "text": "Thanks for sharing",
      "created_at": "2025-11-15T10:05:00Z"
    }
  ]
}

# Create comment su post 123
POST /api/posts/123/comments HTTP/1.1
Content-Type: application/json

{
  "author": "Anna",
  "text": "Nice article!"
}

HTTP/1.1 201 Created
Location: /api/posts/123/comments/3
```

4.4.2 Quando Evitare Nesting

NON usare nesting quando:

- Child esiste indipendentemente (es: users → orders)
- Nesting diventa troppo profondo (>2 livelli)
- Operazioni su child non necessitano parent

Listing 4.12: Nesting vs Flat - Orders

```
# BAD: Troppo nested
GET /api/users/123/orders/456/items/789/reviews/12

# GOOD: Flat con filtering
GET /api/orders?user_id=123
GET /api/order-items?order_id=456
GET /api/reviews?item_id=789
```

4.4.3 Pattern Ibrido

Supporta sia nested che flat:

Listing 4.13: Hybrid approach

```
# Nested: comments di post specifico
GET /api/posts/123/comments

# Flat: tutti comments con filtering
GET /api/comments?post_id=123

# Direct access: comment specifico
GET /api/comments/789
```

4.5 Filtering

4.5.1 Query Parameters per Filtering

Filtering via query string:

Listing 4.14: Basic filtering

```
# Single filter
GET /api/users?role=admin

# Multiple filters (AND logic)
GET /api/users?role=admin&status=active

# Exact match
GET /api/products?category=electronics

# Multiple values (OR logic)
GET /api/products?category=electronics,books,toys
```

4.5.2 Advanced Filtering Operators

Listing 4.15: Filter operators

```
# Comparisons
GET /api/products?price_gt=100          # price > 100
GET /api/products?price_gte=100         # price >= 100
GET /api/products?price_lt=500          # price < 500
GET /api/products?price_lte=500         # price <= 500

# Range
GET /api/products?price_min=100&price_max=500

# Date filtering
GET /api/orders?created_after=2025-01-01
GET /api/orders?created_before=2025-12-31
GET /api/orders?created_between=2025-01-01,2025-12-31

# Text search
GET /api/users?name_contains=mario
GET /api/users?email_starts_with=admin
GET /api/products?name_like=%laptop%

```

```
# Negation
GET /api/users?status_not=deleted
GET /api/products?category_not_in=adult,gambling
```

4.5.3 Complex Filtering

Filtering con JSON in POST body per query complesse:

Listing 4.16: Complex filter con POST

```
POST /api/users/search HTTP/1.1
Content-Type: application/json

{
  "filters": {
    "and": [
      {
        "field": "age",
        "operator": "gte",
        "value": 18
      },
      {
        "or": [
          {
            "field": "role",
            "operator": "eq",
            "value": "admin"
          },
          {
            "field": "role",
            "operator": "eq",
            "value": "moderator"
          }
        ]
      },
      {
        "field": "status",
        "operator": "in",
        "value": ["active", "pending"]
      }
    ]
  }
}

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

{
  "data": [...],
  "total": 42
}
```

4.6 Sorting

4.6.1 Basic Sorting

Listing 4.17: Sorting con query parameter

```
# Sort ascending (default)
GET /api/users?sort=name

# Sort descending (prefix -)
GET /api/users?sort=-created_at

# Multiple sort fields
GET /api/users?sort=role,-created_at,name
# Ordina per: 1) role ASC, 2) created_at DESC, 3) name ASC
```

4.6.2 Alternative Syntax

Listing 4.18: Alternative sorting syntax

```
# Explicit direction
GET /api/users?sort_by=name&order=asc
GET /api/users?sort_by=created_at&order=desc

# Array notation
GET /api/users?sort[]=name:asc&sort[]=age:desc
```

4.6.3 Response Metadata

Listing 4.19: Sorting in response metadata

```
GET /api/users?sort=-created_at HTTP/1.1

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

{
  "data": [...],
  "meta": {
    "sort": {
      "field": "created_at",
      "direction": "desc"
    }
  }
}
```

4.7 Pagination

4.7.1 Offset-based Pagination

Pattern più comune: offset + limit

Listing 4.20: Offset pagination

```
# Page 1: primi 20 items
GET /api/users?offset=0&limit=20

# Page 2: items 21-40
GET /api/users?offset=20&limit=20

# Page 3: items 41-60
```

```
GET /api/users?offset=40&limit=20

HTTP/1.1 200 OK
Content-Type: application/json
X-Total-Count: 150
Link: </api/users?offset=20&limit=20>; rel="next",
      </api/users?offset=0&limit=20>; rel="first",
      </api/users?offset=140&limit=20>; rel="last"

{
  "data": [
    {"id": 1, "name": "User 1"},
    {"id": 2, "name": "User 2"}
  ],
  "pagination": {
    "offset": 0,
    "limit": 20,
    "total": 150
  }
}
```

4.7.2 Page-based Pagination

Alternative: page number + page size

Listing 4.21: Page-based pagination

```
# Page 1
GET /api/users?page=1&per_page=20

# Page 2
GET /api/users?page=2&per_page=20

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

{
  "data": [...],
  "pagination": {
    "current_page": 1,
    "per_page": 20,
    "total_pages": 8,
    "total_items": 150
  },
  "links": {
    "first": "/api/users?page=1&per_page=20",
    "last": "/api/users?page=8&per_page=20",
    "prev": null,
    "next": "/api/users?page=2&per_page=20"
  }
}
```

4.7.3 Cursor-based Pagination

Migliore per large datasets: cursor opaco

Listing 4.22: Cursor pagination

```
# Prima richiesta
GET /api/users?limit=20 HTTP/1.1

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

{
  "data": [
    {"id": 1, "name": "User 1"},
    {"id": 2, "name": "User 2"}
  ],
  "pagination": {
    "next_cursor": "eyJpZCI6MjAsImNyZWF0ZWRfYXQiOiIyMDI1LTExLTE1In0=",
    "has_more": true
  }
}

# Richiesta successiva con cursor
GET /api/users?cursor=eyJpZCI6MjAsImNyZWF0ZWRfYXQiOiIyMDI1LTExLTE1In0=&
   limit=20

HTTP/1.1 200 OK
{
  "data": [...],
  "pagination": {
    "next_cursor": "eyJpZCI6NDAsImNyZWF0ZWRfYXQiOiIyMDI1LTExLTE0In0=",
    "prev_cursor": "eyJpZCI6MjAsImNyZWF0ZWRfYXQiOiIyMDI1LTExLTE1In0=",
    "has_more": true
  }
}
```

Vantaggi cursor pagination:

- Consistent results (anche se data cambia durante pagination)
- Performance migliore su large datasets
- No "page drift" problem

4.7.4 Link Header (RFC 5988)

Standard per pagination links:

Listing 4.23: RFC 5988 Link header

```
GET /api/users?page=3 HTTP/1.1

HTTP/1.1 200 OK
Link: </api/users?page=1>; rel="first",
      </api/users?page=2>; rel="prev",
      </api/users?page=4>; rel="next",
      </api/users?page=10>; rel="last"
Content-Type: application/json

{
  "data": [...]
}
```

4.8 Searching

4.8.1 Simple Search

Listing 4.24: Simple text search

```
# Search in default fields
GET /api/users?q=mario

# Search in specific field
GET /api/users?search=mario&search_fields=name,email

HTTP/1.1 200 OK
{
  "data": [
    {"id": 1, "name": "Mario Rossi", "email": "mario@example.com"},
    {"id": 2, "name": "Mario Bianchi", "email": "mario.b@example.com"}
  ],
  "meta": {
    "query": "mario",
    "total_results": 2
  }
}
```

4.8.2 Full-Text Search

Listing 4.25: Full-text search endpoint

```
GET /api/search?q=REST+API+design&type=posts,articles HTTP/1.1

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

{
  "results": {
    "posts": [
      {
        "id": 123,
        "title": "REST API Design Best Practices",
        "excerpt": "Learn how to design RESTful APIs...",
        "score": 0.95
      }
    ],
    "articles": [
      {
        "id": 456,
        "title": "API Design Patterns",
        "excerpt": "REST and GraphQL comparison...",
        "score": 0.78
      }
    ]
  },
  "meta": {
    "query": "REST API design",
    "total": 2,
    "execution_time_ms": 45
  }
}
```

4.9 Field Selection (Sparse Fieldsets)

Client seleziona solo campi necessari:

Listing 4.26: Field selection

```
# Default: tutti campi
GET /api/users/123 HTTP/1.1
{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com",
  "phone": "+39 123 456",
  "address": {...},
  "created_at": "2025-01-15T10:00:00Z",
  "updated_at": "2025-11-15T10:00:00Z"
}

# Sparse: solo campi specificati
GET /api/users/123?fields=id,name,email HTTP/1.1
{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com"
}

# Multiple resources
GET /api/users?fields=id,name&limit=100
{
  "data": [
    {"id": 1, "name": "Mario"},
    {"id": 2, "name": "Luigi"}
  ]
}
```

Benefici:

- Bandwidth optimization (mobile)
- Faster response
- Privacy (escludere campi sensibili)

4.10 Actions on Resources

4.10.1 Quando URI con Verbs è Accettabile

Operazioni NON mappabili a CRUD possono usare verbs:

Listing 4.27: Actions - verbs accettabili

```
# Actions su risorsa specifica
POST /api/orders/123/cancel
POST /api/orders/123/refund
POST /api/orders/123/ship

POST /api/users/123/activate
POST /api/users/123/deactivate
POST /api/users/123/reset-password

```

```
# Operazioni globali
POST /api/search
POST /api/export
POST /api/import
```

Alternative RESTful:

Listing 4.28: Alternative: State as resource

```
# Invece di POST /orders/123/cancel
PATCH /api/orders/123 HTTP/1.1
Content-Type: application/json

{
  "status": "cancelled",
  "cancelled_reason": "Customer request"
}

# Invece di POST /users/123/activate
PATCH /api/users/123 HTTP/1.1
{
  "active": true
}
```

4.11 Best Practices

Best Practices Resource Design

1. **Nouns not verbs**: `/users` non `/getUsers`
2. **Plural collections**: `/users` consistente
3. **Lowercase + hyphens**: `/order-items`
4. **No trailing slash**: `/users` non `/users/`
5. **Versioning in URI**: `/api/v1/users`
6. **Nesting max 2 livelli**: Evitare `/a/b/c/d/e`
7. **Filtering via query**: `?status=active&role=admin`
8. **Sorting standard**: `?sort=-created_at`
9. **Pagination**: Offset, page, o cursor
10. **Sparse fields**: `?fields=id,name`
11. **Search endpoint**: `/search?q=term`
12. **Link header**: RFC 5988 per pagination
13. **Metadata in response**: total, pagination info

Anti-patterns

- **Verbs everywhere**: `/createUser`, `/deletePost`
- **Mixed singular/plural**: `/user` vs `/products`

- **Implementation leak**: `/api/users.php`, `/db/table`
- **Over-nesting**: `/users/123/posts/456/comments/789/likes`
- **CamelCase URIs**: `/orderItems`
- **Extensions**: `/users/123.json`
- **Filtering in path**: `/users/active` (usare `?status=active`)
- **Actions as resources**: `/users/123/delete` (usare DELETE method)

4.12 Esempi Pratici

4.12.1 E-commerce API

Listing 4.29: E-commerce resource structure

```
# Products
GET    /api/v1/products
POST   /api/v1/products
GET    /api/v1/products/123
PUT    /api/v1/products/123
DELETE /api/v1/products/123

# Product categories
GET /api/v1/categories
GET /api/v1/products?category_id=5

# Cart
GET    /api/v1/cart
POST   /api/v1/cart/items
DELETE /api/v1/cart/items/789

# Orders
GET  /api/v1/orders
POST /api/v1/orders
GET  /api/v1/orders/456
GET  /api/v1/orders?status=pending

# Order actions
POST /api/v1/orders/456/cancel
POST /api/v1/orders/456/refund

# Reviews
GET  /api/v1/products/123/reviews
POST /api/v1/products/123/reviews
GET  /api/v1/reviews/789

# Search
GET /api/v1/search?q=laptop&category=electronics&price_max=1000
```

4.12.2 Social Media API

Listing 4.30: Social media resource structure

```
# Users
GET /api/v1/users/me
GET /api/v1/users/123
GET /api/v1/users?search=mario

# Posts
GET  /api/v1/posts
POST /api/v1/posts
GET  /api/v1/posts/456
GET  /api/v1/users/123/posts

# Comments
GET  /api/v1/posts/456/comments
POST /api/v1/posts/456/comments
GET  /api/v1/comments/789

# Likes
POST   /api/v1/posts/456/likes
DELETE /api/v1/posts/456/likes

# Following
POST   /api/v1/users/123/follow
DELETE /api/v1/users/123/follow
GET    /api/v1/users/123/followers
GET    /api/v1/users/123/following

# Feed
GET /api/v1/feed?cursor=xyz&limit=20
```

4.13 Riepilogo

- URI identifica risorsa (noun), metodo HTTP è azione (verb)
- Naming: plural, lowercase, hyphens
- Nesting per relazioni parent-child (max 2 livelli)
- Filtering, sorting, pagination via query parameters
- Cursor pagination per large datasets
- Sparse fieldsets per bandwidth optimization
- Actions eccezionalmente accettabili se non mappabili a CRUD
- Consistency è fondamentale: pattern uniforme in tutta API

Esercizi

1. Progetta URI structure per sistema biblioteca (books, authors, loans)
2. Implementa filtering avanzato con operatori (gt, lt, contains, in)
3. Implementa cursor-based pagination con encoding opaco
4. Progetta search endpoint con multi-type results

5. Confronta offset vs cursor pagination: pro/cons, performance
6. Refactoring: trasforma API con verbs in nouns RESTful

Riferimenti

- RFC 3986 - URI Generic Syntax: https://tools.ietf.org/html/rfc3986
- RFC 5988 - Web Linking: https://tools.ietf.org/html/rfc5988
- REST API Tutorial: https://restfulapi.net/resource-naming/
- GitHub API Docs: https://docs.github.com/en/rest
- Stripe API Design: https://stripe.com/docs/api

Capitolo 5

JSON e Formati Dati

Mappa del capitolo

JSON fundamentals, Response structure, Envelope vs no-envelope, HAL (Hypertext Application Language), JSON:API specification, Siren, Collection+JSON, Content negotiation, Media types, Accept header, Esempi pratici completi, Best practices.

Obiettivi di apprendimento

- Strutturare JSON response consistenti e leggibili
- Comprendere HAL per hypermedia
- Applicare JSON:API specification
- Implementare content negotiation
- Scegliere formato appropriato per use case
- Costruire API auto-documentanti con hypermedia

5.1 JSON Fundamentals

5.1.1 Perché JSON

JSON (JavaScript Object Notation) è formato standard per REST API moderne.

Vantaggi:

- **Leggibile**: Human-readable
- **Compatto**: Meno verbose di XML
- **Type-safe**: String, number, boolean, null, array, object
- **Universale**: Supporto nativo in tutti linguaggi
- **Parsing veloce**: Performance migliori di XML

Tipi JSON:

Listing 5.1: JSON data types

```
{
  "string": "Hello World",
```

```
  "number": 42,
  "float": 3.14159,
  "boolean": true,
  "null_value": null,
  "array": [1, 2, 3],
  "object": {
    "nested": "value"
  }
}
```

5.1.2 JSON vs XML

Listing 5.2: Same data: JSON vs XML

```
# JSON (conciso)
{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com"
}

# XML (verbose)
<?xml version="1.0"?>
<user>
  <id>123</id>
  <name>Mario Rossi</name>
  <email>mario@example.com</email>
</user>
```

JSON: 87 bytes, **XML**: 135 bytes (55% più grande)

5.2 Response Structure

5.2.1 Envelope Pattern

Envelope: Wrapping data in container object.

Listing 5.3: Envelope pattern

```
GET /api/users/123 HTTP/1.1

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

{
  "status": "success",
  "data": {
    "id": 123,
    "name": "Mario Rossi",
    "email": "mario@example.com"
  },
  "meta": {
    "timestamp": "2025-11-15T15:00:00Z",
    "version": "1.0"
  }
}
```

Pro envelope:

- Metadata separato da data
- Consistenza structure anche per errori
- Extensibility

Contro envelope:

- Overhead extra nesting
- Ridondanza (status già in HTTP status code)

5.2.2 No Envelope (Naked Response)

No envelope: Data direttamente nel root.

Listing 5.4: No envelope - naked response

```
GET /api/users/123 HTTP/1.1

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

{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com"
}
```

Pro no-envelope:

- Più conciso
- HTTP headers per metadata
- Meno parsing client-side

Raccomandazione

Modern REST API: Preferire **no envelope** per singole risorse, **envelope leggero** per collections.
Singola risorsa: Data in root
Collection: Wrapper minimo per metadata pagination

5.2.3 Collection Response Structure

Listing 5.5: Collection con metadata

```
GET /api/users?page=1&limit=10 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/json
X-Total-Count: 150

{
  "data": [
    {
      "id": 1,
      "name": "Mario Rossi",
      "email": "mario@example.com"
```

```
    },
    {
      "id": 2,
      "name": "Luigi Verdi",
      "email": "luigi@example.com"
    }
  ],
  "pagination": {
    "page": 1,
    "per_page": 10,
    "total": 150,
    "total_pages": 15
  },
  "links": {
    "self": "/api/users?page=1&limit=10",
    "first": "/api/users?page=1&limit=10",
    "last": "/api/users?page=15&limit=10",
    "next": "/api/users?page=2&limit=10"
  }
}
```

5.3 HAL - Hypertext Application Language

5.3.1 Cos'è HAL

HAL (RFC 8288) è formato hypermedia che aggiunge link navigabili a JSON/XML.
Media type: `application/hal+json`
Elementi chiave:

- `_links`: Collegamenti ipertestuali ad altre risorse
- `_embedded`: Risorse correlate embedded nella response
- Resource properties: Dati effettivi risorsa

5.3.2 HAL Structure

Listing 5.6: HAL - Single resource

```
GET /api/users/123 HTTP/1.1
Accept: application/hal+json

HTTP/1.1 200 OK
Content-Type: application/hal+json

{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com",
  "role": "admin",
  "_links": {
    "self": {
      "href": "/api/users/123"
    },
    "orders": {
      "href": "/api/users/123/orders"
    },
```

```
    "profile": {
      "href": "/api/users/123/profile"
    }
  }
}
```

5.3.3 HAL Embedded Resources

Listing 5.7: HAL - Embedded resources

```
GET /api/orders/456 HTTP/1.1
Accept: application/hal+json

HTTP/1.1 200 OK
Content-Type: application/hal+json

{
  "id": 456,
  "status": "shipped",
  "total": 99.99,
  "_links": {
    "self": {
      "href": "/api/orders/456"
    },
    "customer": {
      "href": "/api/customers/789"
    },
    "invoice": {
      "href": "/api/orders/456/invoice"
    }
  },
  "_embedded": {
    "customer": {
      "id": 789,
      "name": "Mario Rossi",
      "email": "mario@example.com",
      "_links": {
        "self": {
          "href": "/api/customers/789"
        }
      }
    },
    "items": [
      {
        "id": 1,
        "product_id": 100,
        "quantity": 2,
        "price": 49.99,
        "_links": {
          "self": {
            "href": "/api/order-items/1"
          },
          "product": {
            "href": "/api/products/100"
          }
        }
      }
```

```
    ]
  }
}
```

5.3.4 HAL Collection

Listing 5.8: HAL - Collection

```
GET /api/users HTTP/1.1
Accept: application/hal+json

HTTP/1.1 200 OK
Content-Type: application/hal+json

{
  "_links": {
    "self": {
      "href": "/api/users?page=1"
    },
    "first": {
      "href": "/api/users?page=1"
    },
    "last": {
      "href": "/api/users?page=10"
    },
    "next": {
      "href": "/api/users?page=2"
    }
  },
  "_embedded": {
    "users": [
      {
        "id": 1,
        "name": "Mario Rossi",
        "_links": {
          "self": {
            "href": "/api/users/1"
          }
        }
      },
      {
        "id": 2,
        "name": "Luigi Verdi",
        "_links": {
          "self": {
            "href": "/api/users/2"
          }
        }
      }
    ]
  },
  "page": 1,
  "total": 100
}
```

5.3.5 HAL Link Relations

Standard IANA link relations:

- `self`: URI della risorsa corrente
- `first/last/next/prev`: Pagination
- `up`: Parent resource
- `related`: Risorsa correlata
- `alternate`: Rappresentazione alternativa
- `edit`: URI per modificare risorsa
- `delete`: URI per eliminare

Listing 5.9: HAL - Rich link relations

```
{
  "id": 123,
  "name": "Mario Rossi",
  "_links": {
    "self": {
      "href": "/api/users/123"
    },
    "edit": {
      "href": "/api/users/123",
      "title": "Edit this user"
    },
    "delete": {
      "href": "/api/users/123",
      "title": "Delete this user",
      "method": "DELETE"
    },
    "avatar": {
      "href": "/api/users/123/avatar",
      "type": "image/jpeg"
    }
  }
}
```

5.4 JSON:API Specification

5.4.1 Cos'è JSON:API

JSON:API è specification completa per costruire API JSON consistenti.
Media type: `application/vnd.api+json`
Obiettivi:

- Struttura standardizzata
- Ridurre roundtrips (compound documents)
- Sparse fieldsets
- Filtering, sorting, pagination built-in

5.4.2 JSON:API Document Structure

Listing 5.10: JSON:API - Single resource

```
GET /api/users/123 HTTP/1.1
Accept: application/vnd.api+json

HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "users",
    "id": "123",
    "attributes": {
      "name": "Mario Rossi",
      "email": "mario@example.com",
      "role": "admin"
    },
    "relationships": {
      "orders": {
        "links": {
          "self": "/api/users/123/relationships/orders",
          "related": "/api/users/123/orders"
        }
      },
      "company": {
        "data": {
          "type": "companies",
          "id": "456"
        }
      }
    },
    "links": {
      "self": "/api/users/123"
    }
  }
}
```

Componenti:

- `data`: Resource object principale
- `type`: Tipo risorsa (required)
- `id`: Identificatore (required per existing resources)
- `attributes`: Attributi risorsa
- `relationships`: Relazioni con altre risorse
- `links`: Hypermedia links

5.4.3 JSON:API Collection

Listing 5.11: JSON:API - Collection

```
GET /api/users HTTP/1.1
Accept: application/vnd.api+json
```

```
3
4  HTTP/1.1 200 OK
5  Content-Type: application/vnd.api+json
6
7  {
8    "data": [
9      {
10       "type": "users",
11       "id": "1",
12       "attributes": {
13         "name": "Mario Rossi",
14         "email": "mario@example.com"
15       },
16       "links": {
17         "self": "/api/users/1"
18       }
19     },
20     {
21       "type": "users",
22       "id": "2",
23       "attributes": {
24         "name": "Luigi Verdi",
25         "email": "luigi@example.com"
26       },
27       "links": {
28         "self": "/api/users/2"
29       }
30     }
31   ],
32   "links": {
33     "self": "/api/users?page[number]=1",
34     "first": "/api/users?page[number]=1",
35     "last": "/api/users?page[number]=10",
36     "next": "/api/users?page[number]=2"
37   },
38   "meta": {
39     "total": 100
40   }
41 }
```

5.4.4 JSON:API Compound Documents (Included)

Riduce N+1 queries: Include risorse correlate.

Listing 5.12: JSON:API - Compound document

```
1  GET /api/orders/456?include=customer,items HTTP/1.1
2  Accept: application/vnd.api+json
3
4  HTTP/1.1 200 OK
5  Content-Type: application/vnd.api+json
6
7  {
8    "data": {
9      "type": "orders",
10     "id": "456",
11     "attributes": {
12       "status": "shipped",
```

```
      "total": 99.99
    },
    "relationships": {
      "customer": {
        "data": {
          "type": "customers",
          "id": "789"
        }
      },
      "items": {
        "data": [
          {
            "type": "order-items",
            "id": "1"
          },
          {
            "type": "order-items",
            "id": "2"
          }
        ]
      }
    }
  },
  "included": [
    {
      "type": "customers",
      "id": "789",
      "attributes": {
        "name": "Mario Rossi",
        "email": "mario@example.com"
      }
    },
    {
      "type": "order-items",
      "id": "1",
      "attributes": {
        "product_id": 100,
        "quantity": 2,
        "price": 49.99
      }
    },
    {
      "type": "order-items",
      "id": "2",
      "attributes": {
        "product_id": 200,
        "quantity": 1,
        "price": 49.99
      }
    }
  ]
}
```

5.4.5 JSON:API Sparse Fieldsets

Listing 5.13: JSON:API - Sparse fieldsets

```
GET /api/users?fields[users]=name,email HTTP/1.1
Accept: application/vnd.api+json

HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [
    {
      "type": "users",
      "id": "1",
      "attributes": {
        "name": "Mario Rossi",
        "email": "mario@example.com"
      }
    }
  ]
}
```

5.4.6 JSON:API Errors

Listing 5.14: JSON:API - Error format

```
POST /api/users HTTP/1.1
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "users",
    "attributes": {
      "name": "",
      "email": "invalid-email"
    }
  }
}

HTTP/1.1 422 Unprocessable Entity
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "status": "422",
      "source": {
        "pointer": "/data/attributes/name"
      },
      "title": "Invalid Attribute",
      "detail": "Name cannot be blank"
    },
    {
      "status": "422",
      "source": {
        "pointer": "/data/attributes/email"
      },
      "title": "Invalid Attribute",
      "detail": "Email is not a valid email address"
    }
```

```
  ]
}
```

5.5 Altri Formati Hypermedia

5.5.1 Siren

Siren: Hypermedia specification con focus su actions.

Listing 5.15: Siren format

```
GET /api/orders/456 HTTP/1.1
Accept: application/vnd.siren+json

HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json

{
  "class": ["order"],
  "properties": {
    "orderNumber": 456,
    "status": "pending",
    "total": 99.99
  },
  "entities": [
    {
      "class": ["customer"],
      "rel": ["customer"],
      "properties": {
        "customerId": 789,
        "name": "Mario Rossi"
      },
      "links": [
        {
          "rel": ["self"],
          "href": "/api/customers/789"
        }
      ]
    }
  ],
  "actions": [
    {
      "name": "cancel-order",
      "title": "Cancel Order",
      "method": "POST",
      "href": "/api/orders/456/cancel"
    },
    {
      "name": "update-order",
      "title": "Update Order",
      "method": "PUT",
      "href": "/api/orders/456",
      "type": "application/json",
      "fields": [
        {
          "name": "status",
          "type": "text"
        }
```

```
      ]
    }
  ],
  "links": [
    {
      "rel": ["self"],
      "href": "/api/orders/456"
    }
  ]
}
```

5.5.2 Collection+JSON

Collection+JSON: Focus su collections e forms.

Listing 5.16: Collection+JSON format

```
GET /api/users HTTP/1.1
Accept: application/vnd.collection+json

HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json

{
  "collection": {
    "version": "1.0",
    "href": "/api/users",
    "items": [
      {
        "href": "/api/users/1",
        "data": [
          {
            "name": "name",
            "value": "Mario Rossi"
          },
          {
            "name": "email",
            "value": "mario@example.com"
          }
        ]
      }
    ],
    "template": {
      "data": [
        {
          "name": "name",
          "value": "",
          "prompt": "Full name"
        },
        {
          "name": "email",
          "value": "",
          "prompt": "Email address"
        }
      ]
    }
  }
}
```

5.6 Content Negotiation

5.6.1 Accept Header

Client specifica formato desiderato via Accept header.

Listing 5.17: Content negotiation - Accept header

```
# Request JSON
GET /api/users/123 HTTP/1.1
Accept: application/json

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

{"id": 123, "name": "Mario"}

# Request HAL
GET /api/users/123 HTTP/1.1
Accept: application/hal+json

HTTP/1.1 200 OK
Content-Type: application/hal+json

{
  "id": 123,
  "name": "Mario",
  "_links": {...}
}

# Request JSON:API
GET /api/users/123 HTTP/1.1
Accept: application/vnd.api+json

HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "users",
    "id": "123",
    "attributes": {...}
  }
}
```

5.6.2 Multiple Accept Types

Client può specificare preferenze con quality values:

Listing 5.18: Accept with quality values

```
GET /api/users/123 HTTP/1.1
Accept: application/hal+json;q=1.0, application/json;q=0.8, */*;q=0.5

# Server sceglie in ordine di preferenza:
# 1. application/hal+json (q=1.0)
# 2. application/json (q=0.8)
# 3. Qualsiasi altro formato (q=0.5)

```

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

{...}
```

5.6.3 406 Not Acceptable

Server non supporta formato richiesto:

Listing 5.19: 406 quando formato non supportato

```
GET /api/users/123 HTTP/1.1
Accept: application/xml

HTTP/1.1 406 Not Acceptable
Content-Type: application/json

{
  "error": "not_acceptable",
  "message": "Requested format not supported",
  "supported_formats": [
    "application/json",
    "application/hal+json"
  ]
}
```

5.6.4 Content-Type Header

Client specifica formato payload in POST/PUT:

Listing 5.20: Content-Type in POST

```
POST /api/users HTTP/1.1
Content-Type: application/json
Accept: application/hal+json

{
  "name": "Mario Rossi",
  "email": "mario@example.com"
}

HTTP/1.1 201 Created
Location: /api/users/123
Content-Type: application/hal+json

{
  "id": 123,
  "name": "Mario Rossi",
  "_links": {...}
}
```

Aspetto	Plain JSON	HAL
Semplicità	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet extbullet rc
Hypermedia	rc rc rc rc rc	extbullet extbullet extbullet extbullet extbullet
Spec rigida	rc rc rc rc rc	extbullet extbullet rc rc rc
Tooling	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet rc rc
Learning curve	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet extbullet rc
Compattezza	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet rc rc
Adoption	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet rc rc

Tabella 5.1: Confronto formati JSON

5.7 Confronto Formati

5.8 Best Practices

Best Practices JSON e Formati

1. **Plain JSON per default**: Semplicità prima di tutto
2. **HAL per hypermedia**: Se serve navigabilità
3. **JSON:API per complex API**: Spec completa, ecosystem
4. **Consistenza naming**: camelCase o snake_case (scegliere uno)
5. **ISO 8601 per date**: `2025-11-15T15:00:00Z`
6. **null vs omissione**: Null per valore assente, ometti per non applicabile
7. **Array sempre array**: Anche singolo elemento `[item]`
8. **Evita deep nesting**: Max 2-3 livelli
9. **Pagination metadata**: In response body o headers
10. **Content negotiation**: Accept e Content-Type corretti
11. **Pretty print in dev**: Readability, minify in prod
12. **UTF-8 encoding**: `Content-Type: application/json; charset=utf-8`

Anti-patterns

- **Stringified JSON**: `{"data": "{\"id\": 123}"}`
- **Inconsistent naming**: Mix camelCase/snake_case
- **String numbers**: `{"age": "25"}` invece di `{"age": 25}`
- **Date as string non-ISO**: `"15/11/2025"` invece di `"2025-11-15"`
- **Array singolo come object**: `{"user": {...}}` quando potrebbe essere `[{...}]`
- **Envelope inutile**: Wrapping quando HTTP headers sufficienti
- **BOM in UTF-8**: Causa parsing errors
- **Trailing commas**: Non valido in JSON strict

5.9 Esempi Pratici

5.9.1 E-commerce API - Multiple Formats

Listing 5.21: Plain JSON

```
GET /api/products/123 HTTP/1.1
Accept: application/json

{
  "id": 123,
  "name": "Laptop XYZ",
  "price": 999.99,
  "category": "electronics",
  "in_stock": true
}
```

Listing 5.22: HAL

```
GET /api/products/123 HTTP/1.1
Accept: application/hal+json

{
  "id": 123,
  "name": "Laptop XYZ",
  "price": 999.99,
  "_links": {
    "self": {"href": "/api/products/123"},
    "category": {"href": "/api/categories/electronics"},
    "reviews": {"href": "/api/products/123/reviews"},
    "add-to-cart": {"href": "/api/cart/items", "method": "POST"}
  }
}
```

Listing 5.23: JSON:API

```
GET /api/products/123 HTTP/1.1
Accept: application/vnd.api+json

{
  "data": {
    "type": "products",
    "id": "123",
    "attributes": {
      "name": "Laptop XYZ",
      "price": 999.99,
      "in_stock": true
    },
    "relationships": {
      "category": {
        "data": {"type": "categories", "id": "electronics"},
        "links": {"related": "/api/categories/electronics"}
      }
    }
  }
}
```

5.10 Riepilogo

- JSON è standard de facto per REST API moderne
- Plain JSON: semplice, universale, ottimo default
- HAL: aggiunge hypermedia con `_links` e `_embedded`
- JSON:API: spec completa con compound documents, sparse fields
- Siren: focus su actions e forms
- Content negotiation via Accept e Content-Type headers
- Consistenza naming e structure fondamentale
- ISO 8601 per date/time

Esercizi

1. Implementa HAL per API blog (posts, comments, authors)
2. Converti API esistente a JSON:API spec
3. Implementa content negotiation: JSON, HAL, JSON:API
4. Progetta error response standardizzato RFC 7807
5. Benchmark: Plain JSON vs HAL vs JSON:API (payload size)
6. Crea client che consuma HAL hypermedia links

Riferimenti

- JSON Specification: https://www.json.org/
- HAL Specification: http://stateless.co/hal_specification.html
- JSON:API Specification: https://jsonapi.org/
- Siren Specification: https://github.com/kevinswiber/siren
- RFC 7159 - JSON: https://tools.ietf.org/html/rfc7159
- RFC 8288 - Web Linking: https://tools.ietf.org/html/rfc8288

Capitolo 6

API Versioning

Mappa del capitolo

Perché versioning, Breaking changes vs backward compatible, Semantic versioning, URI versioning, Header versioning, Query parameter versioning, Content negotiation versioning, Deprecation strategy, Migration path, Best practices, Case studies.

Obiettivi di apprendimento

- Comprendere necessità di versioning API
- Distinguere breaking vs non-breaking changes
- Applicare semantic versioning
- Implementare URI versioning
- Implementare header versioning
- Pianificare deprecation strategy
- Gestire multiple versioni simultanee
- Comunicare changes a developers

6.1 Perché Versioning

6.1.1 Il Problema

API pubbliche hanno client multipli fuori dal tuo controllo:

- Mobile apps (iOS, Android) con release cycles diversi
- Third-party integrations
- Partner commerciali
- Legacy systems

Senza versioning:

Listing 6.1: Breaking change rompe client

```
# API v1 (iniziale)
GET /api/users/123
{
  "id": 123,
  "name": "Mario Rossi"
}

# Breaking change: rename field
GET /api/users/123
{
  "id": 123,
  "full_name": "Mario Rossi"  # Client che si aspetta "name" si rompe!
}
```

Client esistenti crashano o malfunzionano.

6.1.2 Quando Serve Versioning

Necessario per:

- Breaking changes inevitabili
- Redesign architetturale
- Security fixes che cambiano behavior
- Performance optimization con side effects

NON necessario per:

- Bug fixes senza cambio behavior
- Nuovi endpoint (additive)
- Nuovi optional fields (backward compatible)
- Performance improvements trasparenti

6.2 Breaking vs Non-Breaking Changes

6.2.1 Breaking Changes

Breaking: Client esistenti smettono di funzionare.

Esempi breaking:

- Rimuovere campo da response
- Rinominare campo
- Cambiare tipo campo (`string` → `number`)
- Rimuovere endpoint
- Cambiare semantica esistente
- Rendere campo obbligatorio (era opzionale)
- Cambiare status code default

- Cambiare error structure

Listing 6.2: Breaking changes - Esempi

```
# BREAKING: Campo rimosso
# Prima
{"id": 123, "name": "Mario", "email": "mario@example.com"}
# Dopo
{"id": 123, "name": "Mario"}  # email rimosso!

# BREAKING: Tipo cambiato
# Prima
{"price": "99.99"}  # string
# Dopo
{"price": 99.99}    # number

# BREAKING: Semantica cambiata
# Prima: GET /users ritorna tutti users
# Dopo: GET /users ritorna solo active users
```

6.2.2 Non-Breaking Changes

Non-breaking: Client esistenti continuano a funzionare.

Esempi non-breaking:

- Aggiungere nuovo campo (client ignora)
- Aggiungere nuovo endpoint
- Aggiungere optional parameter
- Rendere campo opzionale (era obbligatorio)
- Fix bug senza cambio semantica
- Aggiungere valore a enum esistente

Listing 6.3: Non-breaking changes - OK

```
# NON-BREAKING: Nuovo campo aggiunto
# Prima
{"id": 123, "name": "Mario"}
# Dopo
{"id": 123, "name": "Mario", "avatar_url": "https://..."}
# Client vecchio ignora avatar_url

# NON-BREAKING: Nuovo endpoint
POST /api/v1/users/bulk-import  # Nuovo, non impatta esistenti

# NON-BREAKING: Optional parameter
GET /api/users?include_deleted=true  # Default false, backward
    compatible
```

Robustness Principle (Postel's Law)

Be conservative in what you send, be liberal in what you accept.

API: Mantieni response contract stabile

Client: Ignora campi unknown, non assumere field order

6.3 Semantic Versioning

6.3.1 SemVer Schema

Semantic Versioning: `MAJOR.MINOR.PATCH`

- **MAJOR**: Breaking changes incompatibili
- **MINOR**: Nuove feature backward compatible
- **PATCH**: Bug fixes backward compatible

Esempi:

- `1.0.0` → `1.0.1`: Bug fix (PATCH)
- `1.0.1` → `1.1.0`: Nuova feature (MINOR)
- `1.1.0` → `2.0.0`: Breaking change (MAJOR)

Listing 6.4: Semantic versioning changelog

```
# Version 1.0.0 - Initial release
- GET /users
- POST /users
- GET /users/:id

# Version 1.1.0 - Minor (additive)
- GET /users/:id/orders (nuovo endpoint)
- GET /users?role=admin (nuovo filter)

# Version 1.1.1 - Patch (bug fix)
- Fix: GET /users pagination count incorrect

# Version 2.0.0 - Major (breaking)
- BREAKING: "name" field renamed to "full_name"
- BREAKING: DELETE /users/:id requires admin role
- New: Support for filtering by multiple roles
```

6.3.2 SemVer in API

Strategia comune: Solo MAJOR version in URI.

Listing 6.5: SemVer in API URI

```
/api/v1/users      # Version 1.x.x
/api/v2/users      # Version 2.x.x
/api/v3/users      # Version 3.x.x

# MINOR/PATCH changes dentro stesso MAJOR
# v1.0.0 -> v1.5.3 tutti su /api/v1/users
# Breaking change -> v2.0.0 -> /api/v2/users
```

6.4 URI Versioning

6.4.1 Pattern

Versione in URI path: Approccio più comune.

Listing 6.6: URI versioning

```
GET /api/v1/users/123 HTTP/1.1

HTTP/1.1 200 OK
{
  "id": 123,
  "name": "Mario Rossi"
}

GET /api/v2/users/123 HTTP/1.1

HTTP/1.1 200 OK
{
  "id": 123,
  "full_name": "Mario Rossi",  # Breaking: name -> full_name
  "avatar": "https://..."
}
```

6.4.2 Vantaggi URI Versioning

- **Esplicito**: Versione visibile in URL
- **Semplice**: No header custom
- **Cacheable**: Proxy/CDN vedono versione
- **Browser-friendly**: Testabile direttamente
- **Developer-friendly**: Intuitive

6.4.3 Svantaggi URI Versioning

- Viola principio REST (risorsa = URI unico)
- Duplicazione codice per multiple versioni
- URI pollution

6.4.4 Varianti URI Versioning

Listing 6.7: URI versioning - Varianti

```
# Version in path prefix (common)
/api/v1/users
/api/v2/users

# Version in subdomain
https://v1.api.example.com/users
https://v2.api.example.com/users

# Version in path segment
/api/users/v1
/api/users/v2
```

Raccomandato: `/api/v{major}/resource`

6.5 Header Versioning

6.5.1 Custom Header

Versione in custom HTTP header:

Listing 6.8: Custom header versioning

```
GET /api/users/123 HTTP/1.1
API-Version: 1

HTTP/1.1 200 OK
API-Version: 1
{
  "id": 123,
  "name": "Mario Rossi"
}

GET /api/users/123 HTTP/1.1
API-Version: 2

HTTP/1.1 200 OK
API-Version: 2
{
  "id": 123,
  "full_name": "Mario Rossi"
}
```

6.5.2 Accept Header (Content Negotiation)

Versione in Accept header via vendor media type:

Listing 6.9: Accept header versioning

```
GET /api/users/123 HTTP/1.1
Accept: application/vnd.example.v1+json

HTTP/1.1 200 OK
Content-Type: application/vnd.example.v1+json
{
  "id": 123,
  "name": "Mario Rossi"
}

GET /api/users/123 HTTP/1.1
Accept: application/vnd.example.v2+json

HTTP/1.1 200 OK
Content-Type: application/vnd.example.v2+json
{
  "id": 123,
  "full_name": "Mario Rossi"
}
```

Esempi real-world:

- GitHub: `Accept: application/vnd.github.v3+json`

- Twilio: `Accept: application/vnd.twilio.v1+json`

6.5.3 Vantaggi Header Versioning

- **RESTful puro**: URI identifica risorsa, non versione
- **Single URI**: `/api/users/123` per tutte versioni
- **Content negotiation**: Allineato con HTTP standard

6.5.4 Svantaggi Header Versioning

- **Meno intuitivo**: Developer deve conoscere header
- **Caching complesso**: Proxy/CDN devono considerare header
- **Testing difficile**: Non testabile direttamente in browser
- **Debugging**: Versione nascosta, non in URL

6.6 Query Parameter Versioning

Listing 6.10: Query parameter versioning

```
GET /api/users/123?version=1 HTTP/1.1

HTTP/1.1 200 OK
{
  "id": 123,
  "name": "Mario Rossi"
}

GET /api/users/123?version=2 HTTP/1.1

HTTP/1.1 200 OK
{
  "id": 123,
  "full_name": "Mario Rossi"
}
```

Vantaggi:

- Semplice da implementare
- Visibile in URL
- Default version possibile (`?version=1` default se omesso)

Svantaggi:

- Mescola versioning con filtering
- URL pollution
- Poco usato in practice

6.7 Confronto Strategie

Aspetto	URI	Header
Semplicità	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet rc rc
RESTful	extbullet extbullet rc rc rc	extbullet extbullet extbullet extbullet extbullet
Caching	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet rc rc
Testabilità	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet rc rc rc
Adoption	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet rc rc
Documentation	extbullet extbullet extbullet extbullet extbullet	extbullet extbullet extbullet rc rc

Tabella 6.1: Confronto strategie versioning

Raccomandazione

Per API pubbliche: **URI versioning** (`/api/v1/`)
Motivi:

- Massima semplicità per developer
- Testabile in browser
- Documentazione chiara
- Caching straightforward
- Industry standard (Stripe, Twitter, GitHub v3)

Per API interne o advanced: Header/Accept versioning se team ha expertise REST.

6.8 Version Lifecycle

6.8.1 Supportare Multiple Versioni

Strategia common: Supporta N versioni simultanee.

Listing 6.11: Multiple versions deployment

```
# Production deployment
/api/v1/users  -> v1 codebase (maintenance mode)
/api/v2/users  -> v2 codebase (stable)
/api/v3/users  -> v3 codebase (current)

# Client migration
- Legacy apps: v1
- Majority apps: v2
- New apps: v3
```

6.8.2 Deprecation Strategy

Processo graduale:

1. **Announce**: Comunicare deprecation con anticipo (6-12 mesi)
2. **Sunset header**: Indicare data fine supporto
3. **Warning header**: Alert in response
4. **Migration guide**: Documentare breaking changes
5. **Grace period**: Overlap versioni

6. **Shutdown**: Disabilitare old version

Listing 6.12: Deprecation headers

```
GET /api/v1/users/123 HTTP/1.1

HTTP/1.1 200 OK
Sunset: Sat, 31 Dec 2025 23:59:59 GMT
Warning: 299 - "API v1 is deprecated. Migrate to v2 by 2025-12-31"
Link: <https://docs.example.com/migration-v1-to-v2>; rel="deprecation"
Content-Type: application/json

{
  "id": 123,
  "name": "Mario Rossi",
  "_deprecation": {
    "sunset": "2025-12-31T23:59:59Z",
    "migration_guide": "https://docs.example.com/migration-v1-to-v2",
    "current_version": "v2"
  }
}
```

6.8.3 Shutdown Sequence

Listing 6.13: Graceful shutdown v1

```
# Phase 1: Announce (12 months before)
- Blog post
- Email to registered developers
- In-app notifications

# Phase 2: Warning (6 months before)
- Sunset header in every response
- Dashboard warning for API key holders
- Deprecation notice in docs

# Phase 3: Restricted (3 months before)
- Rate limit reduced for deprecated version
- 429 Too Many Requests con upgrade suggestion

# Phase 4: Read-only (1 month before)
- Only GET requests allowed
- POST/PUT/DELETE -> 410 Gone

# Phase 5: Shutdown
GET /api/v1/users/123 HTTP/1.1

HTTP/1.1 410 Gone
Content-Type: application/json

{
  "error": "version_sunset",
  "message": "API v1 has been shut down",
  "sunset_date": "2025-12-31",
  "current_version": {
    "version": "v2",
    "endpoint": "/api/v2/users/123"
```

```
  },
  "migration_guide": "https://docs.example.com/migration"
}
```

6.9 Migration Best Practices

6.9.1 Documentation

Migration guide must include:

- **Breaking changes list**: Dettagliato con esempi
- **Side-by-side comparison**: v1 vs v2 request/response
- **Code examples**: Snippet in linguaggi popolari
- **Timeline**: Date chiave (announce, sunset, shutdown)
- **Support**: Canale per domande migration

Listing 6.14: Migration guide example

```
# Migration Guide: v1 -> v2

## Breaking Changes

### 1. User name field renamed

**v1**
GET /api/v1/users/123
{
  "id": 123,
  "name": "Mario Rossi"
}

**v2**
GET /api/v2/users/123
{
  "id": 123,
  "full_name": "Mario Rossi"
}

**Migration**: Update client code to use "full_name" instead of "name"

### 2. Authentication changed

**v1**: API key in query parameter
GET /api/v1/users?api_key=abc123

**v2**: Bearer token in Authorization header
GET /api/v2/users
Authorization: Bearer eyJhbGc...

**Migration**: Upgrade to OAuth 2.0, obtain access token

## Timeline

```

```
- 2025-06-01: v2 released, v1 deprecated
- 2025-09-01: v1 rate limits reduced
- 2025-11-01: v1 read-only
- 2025-12-31: v1 shutdown

## Support

- Migration questions: api-support@example.com
- Slack channel: #api-v2-migration
```

6.9.2 Versioning Client Libraries

Listing 6.15: SDK versioning

```
# Python SDK
pip install example-api-client==1.x  # v1 API
pip install example-api-client==2.x  # v2 API

# JavaScript SDK
npm install example-api@1.x
npm install example-api@2.x

# Code example
# v1 SDK
from example_api import Client
client = Client(api_key="abc123")
user = client.users.get(123)
print(user.name)  # v1 field

# v2 SDK
from example_api import ClientV2
client = ClientV2(access_token="Bearer ...")
user = client.users.get(123)
print(user.full_name)  # v2 field
```

6.10 Case Studies

6.10.1 Stripe API

Strategia: Date-based versioning in header

Listing 6.16: Stripe versioning

```
GET /v1/customers/cus_123 HTTP/1.1
Stripe-Version: 2025-01-15
Authorization: Bearer sk_test_...

# Breaking changes rilasciati con date version
# Client può rimanere su old version indefinitamente
```

Caratteristiche:

- Single URI per risorsa
- Versione = date (YYYY-MM-DD)
- Backward compatibility garantita
- Client sceglie quando migrare

6.10.2 GitHub API

v3: URI versioning + Accept header

Listing 6.17: GitHub API v3

```
GET /repos/octocat/Hello-World HTTP/1.1
Accept: application/vnd.github.v3+json

# URI: /repos (no /v3/ prefix)
# Version in Accept header
```

v4: GraphQL (breaking redesign)

```
POST /graphql HTTP/1.1
# Completamente diverso da v3 REST
```

6.10.3 Twitter API

Strategia: URI versioning major rewrites

Listing 6.18: Twitter API evolution

```
# v1.1 (deprecated 2013)
GET /1.1/statuses/user_timeline.json

# v2 (released 2020, redesign completo)
GET /2/tweets
GET /2/users/:id

# Multiple versioni per transition period
```

6.11 Best Practices

Best Practices Versioning

1. **Versiona da subito**: Include v1 anche per prima release
2. **Solo MAJOR in URI**: `/api/v1/`, `/api/v2/`
3. **Semantic versioning**: MAJOR.MINOR.PATCH internamente
4. **Default version**: Redirect `/api/` → `/api/v2/`
5. **Breaking changes solo in MAJOR**: v1 → v2
6. **Supporta N versioni**: Almeno current + previous
7. **Sunset header**: Comunica deprecation
8. **Migration guide**: Documentazione dettagliata
9. **Long sunset period**: 6-12 mesi minimo
10. **Monitor usage**: Analytics per versione
11. **Changelog pubblico**: Tutte le modifiche documentate
12. **Never break existing**: Se possibile, evita breaking changes

Errori comuni

- **No versioning**: Rompere client esistenti
- **Breaking in MINOR/PATCH**: Viola semantic versioning
- **Too many versions**: Supportare v1-v10 contemporaneamente
- **Sunset improvviso**: Shutdown senza preavviso
- **Poor documentation**: Migration guide incompleto
- **Versioning inconsistente**: Mix URI + header + query
- **MINOR/PATCH in URI**: `/api/v1.2.3/` troppo granulare

6.12 Evitare Breaking Changes

6.12.1 Strategie per Backward Compatibility

Listing 6.19: Additive changes invece di breaking

```
# BREAKING: Rimuovere campo
# BAD
# v1: {"id": 123, "name": "Mario", "email": "mario@example.com"}
# v2: {"id": 123, "name": "Mario"}  # email rimosso

# GOOD: Deprecare ma mantenere
# v2: {"id": 123, "name": "Mario", "email": "mario@example.com"}
# Response header: Warning: 299 - "email field deprecated"

# BREAKING: Rinominare campo
# BAD
# v1: {"name": "Mario"}
# v2: {"full_name": "Mario"}

# GOOD: Aggiungere nuovo, mantenere vecchio
# v2: {"name": "Mario", "full_name": "Mario"}
# Deprecare "name", clients migrano gradualmente

# BREAKING: Cambiare tipo
# BAD
# v1: {"price": "99.99"}
# v2: {"price": 99.99}

# GOOD: Aggiungere campo nuovo tipo
# v2: {"price": "99.99", "price_numeric": 99.99}
```

6.12.2 Expand/Contract Pattern

Three-phase migration:

1. **Expand**: Aggiungere nuovo campo/endpoint (backward compatible)
2. **Migrate**: Client migrano gradualmente
3. **Contract**: Rimuovere vecchio campo (new major version)

Listing 6.20: Expand/Contract example

```
# Phase 1: Expand (v1.1.0 - backward compatible)
{
  "name": "Mario",         # Old field (deprecato)
  "full_name": "Mario"     # New field (consigliato)
}

# Phase 2: Clients migrate over 6 months

# Phase 3: Contract (v2.0.0 - breaking)
{
  "full_name": "Mario"     # Solo nuovo campo
}
```

6.13 Riepilogo

- Versioning è essenziale per API pubbliche con client multipli
- Breaking changes richiedono new major version
- URI versioning: più semplice e comune (`/api/v1/`)
- Header versioning: più RESTful ma complesso
- Semantic versioning: MAJOR.MINOR.PATCH
- Supportare N versioni simultanee durante transition
- Deprecation graduale con Sunset header
- Migration guide dettagliato obbligatorio
- Sunset period: 6-12 mesi minimo
- Preferire additive changes quando possibile

Esercizi

1. Progetta versioning strategy per API e-commerce
2. Implementa URI versioning con routing multiplo v1/v2
3. Implementa header versioning con content negotiation
4. Scrivi migration guide da v1 a v2 (almeno 5 breaking changes)
5. Implementa Sunset header e deprecation warnings
6. Analizza GitHub/Stripe versioning: pro/cons

Riferimenti

- Semantic Versioning: https://semver.org/
- RFC 8594 - Sunset Header: https://tools.ietf.org/html/rfc8594
- Stripe API Versioning: https://stripe.com/docs/api/versioning
- GitHub API Versioning: https://docs.github.com/en/rest/overview/api-versions
- Roy Fielding on Versioning: https://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm

Capitolo 7

Autenticazione e Autorizzazione

L'autenticazione è il processo di verifica dell'identità di un client che effettua una richiesta API, mentre l'autorizzazione determina quali risorse e operazioni l'utente autenticato può accedere. In questo capitolo esploreremo i principali meccanismi di autenticazione per REST API.

7.1 Principi Fondamentali

7.1.1 Stateless Authentication

Le REST API seguono il principio di statelessness: ogni richiesta deve contenere tutte le informazioni necessarie per l'autenticazione, senza che il server mantenga lo stato della sessione.

Vantaggi dell'Autenticazione Stateless

- **Scalabilità**: Nessuna necessità di sincronizzare sessioni tra server
- **Semplicità**: Ogni richiesta è indipendente
- **Cache-friendly**: Le risposte possono essere facilmente cachate
- **Load balancing**: Qualsiasi server può gestire qualsiasi richiesta

7.1.2 Differenza tra Autenticazione e Autorizzazione

- **Autenticazione**: "Chi sei?" - Verifica l'identità del client
- **Autorizzazione**: "Cosa puoi fare?" - Verifica i permessi dell'utente

7.2 HTTP Basic Authentication

7.2.1 Concetto e Funzionamento

HTTP Basic Authentication è il meccanismo più semplice definito nello standard HTTP. Le credenziali (username e password) vengono codificate in Base64 e inviate nell'header `Authorization`.

Listing 7.1: Formato Header Basic Auth

```
1 Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

7.2.2 Implementazione

Listing 7.2: Esempio cURL con Basic Auth

```
# Metodo 1: Credenziali inline
curl -X GET https://api.example.com/users \
  -H "Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ="

# Metodo 2: Flag -u (piu' comodo)
curl -X GET https://api.example.com/users \
  -u username:password

# Metodo 3: Credenziali da variabili
USER="admin"
PASS="secret123"
curl -X GET https://api.example.com/users \
  -u "$USER:$PASS"
```

7.2.3 Codifica e Decodifica

Listing 7.3: Codifica Base64 delle Credenziali

```
# Codifica
echo -n "username:password" | base64
# Output: dXNlcm5hbWU6cGFzc3dvcmQ=

# Decodifica
echo "dXNlcm5hbWU6cGFzc3dvcmQ=" | base64 -d
# Output: username:password
```

7.2.4 Vantaggi e Svantaggi

Vantaggi

- Semplicità di implementazione
- Supporto universale
- Nessuna configurazione server complessa

Svantaggi

- Le credenziali viaggiano con ogni richiesta
- Base64 non è crittografia (facilmente decodificabile)
- **Richiede HTTPS obbligatorio**
- Nessuna scadenza token
- Impossibile revocare l'accesso senza cambiare password

7.2.5 Implementazione Server-side

Listing 7.4: Verifica Basic Auth in Python

```python
import base64
from functools import wraps
from flask import request, jsonify

def require_basic_auth(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        auth = request.headers.get('Authorization')

        if not auth or not auth.startswith('Basic '):
            return jsonify({'error': 'Missing credentials'}), 401

        try:
            # Decodifica Base64
            credentials = base64.b64decode(
                auth[6:]
            ).decode('utf-8')
            username, password = credentials.split(':', 1)

            # Verifica credenziali (esempio semplificato)
            if not verify_credentials(username, password):
                return jsonify({'error': 'Invalid credentials'}), 401

            # Aggiungi username al contesto
            request.current_user = username
            return f(*args, **kwargs)

        except Exception as e:
            return jsonify({'error': 'Invalid auth format'}), 401

    return decorated_function

@app.route('/api/protected')
@require_basic_auth
def protected_resource():
    return jsonify({'message': f'Hello {request.current_user}'})
```

7.3 Bearer Token Authentication

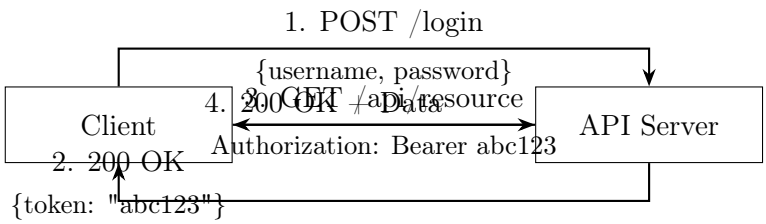
7.3.1 Concetto

Bearer Token è un meccanismo dove il client presenta un token (una stringa opaca) nell'header Authorization. Il token viene ottenuto tramite un endpoint di login separato.

Listing 7.5: Formato Bearer Token

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

7.3.2 Flow di Autenticazione



7.3.3 Esempio Completo

Listing 7.6: Login e Richieste con Bearer Token

```
# 1. Login e ottenimento token
TOKEN=$(curl -X POST https://api.example.com/auth/login \
  -H "Content-Type: application/json" \
  -d '{
    "username": "user@example.com",
    "password": "SecurePass123!"
  }' | jq -r '.token')

echo "Token obtained: $TOKEN"

# 2. Utilizzo del token per richieste API
curl -X GET https://api.example.com/api/users/me \
  -H "Authorization: Bearer $TOKEN"

# 3. Refresh del token (se supportato)
NEW_TOKEN=$(curl -X POST https://api.example.com/auth/refresh \
  -H "Authorization: Bearer $TOKEN" \
  | jq -r '.token')

# 4. Logout (revoca token)
curl -X POST https://api.example.com/auth/logout \
  -H "Authorization: Bearer $TOKEN"
```

7.3.4 Struttura Response Login

Listing 7.7: Response di Login con Token

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "def50200a1b2c3d4e5f6...",
  "user": {
    "id": 12345,
    "email": "user@example.com",
    "name": "Mario Rossi"
  }
}
```

7.4 JSON Web Tokens (JWT)

7.4.1 Struttura JWT

Un JWT è composto da tre parti separate da punti:

```
HEADER.PAYLOAD.SIGNATURE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
    .
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

7.4.2 Componenti JWT

Header

Listing 7.8: JWT Header (Base64URL decoded)

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload

Listing 7.9: JWT Payload (Claims)

```
{
  "sub": "1234567890",
  "name": "Mario Rossi",
  "email": "mario.rossi@example.com",
  "role": "admin",
  "iat": 1516239022,
  "exp": 1516242622,
  "nbf": 1516239022,
  "iss": "https://api.example.com",
  "aud": "https://example.com"
}
```

7.4.3 Standard Claims

Claim	Nome	Descrizione
`iss`	Issuer	Identifica chi ha emesso il token
`sub`	Subject	Identifica il soggetto del token (user ID)
`aud`	Audience	Destinatari del token
`exp`	Expiration	Timestamp di scadenza (Unix time)
`nbf`	Not Before	Il token non è valido prima di questo timestamp
`iat`	Issued At	Timestamp di emissione
`jti`	JWT ID	Identificatore univoco del token

Tabella 7.1: Standard JWT Claims (RFC 7519)

7.4.4 Algoritmi di Firma

- **HS256** (HMAC SHA-256): Firma simmetrica, secret condiviso
- **RS256** (RSA SHA-256): Firma asimmetrica, chiave privata/pubblica
- **ES256** (ECDSA SHA-256): Firma asimmetrica con curve ellittiche

7.4.5 Generazione e Verifica JWT

Listing 7.10: Creazione JWT in Python

```python
import jwt
import datetime

def generate_jwt(user_id, email, role):
    """Genera un JWT per un utente"""

    # Payload con claims
    payload = {
        'sub': str(user_id),
        'email': email,
        'role': role,
        'iat': datetime.datetime.utcnow(),
        'exp': datetime.datetime.utcnow() + datetime.timedelta(hours=1),
        'iss': 'api.example.com',
        'aud': 'example.com'
    }

    # Secret key (dovrebbe essere in variabile ambiente)
    secret = 'your-256-bit-secret'

    # Genera token
    token = jwt.encode(payload, secret, algorithm='HS256')

    return token

def verify_jwt(token):
    """Verifica e decodifica un JWT"""

    secret = 'your-256-bit-secret'

    try:
        # Decodifica e verifica
        payload = jwt.decode(
            token,
            secret,
            algorithms=['HS256'],
            audience='example.com',
            issuer='api.example.com'
        )
        return payload

    except jwt.ExpiredSignatureError:
        raise Exception('Token scaduto')
    except jwt.InvalidAudienceError:
        raise Exception('Audience non valida')
    except jwt.InvalidIssuerError:
        raise Exception('Issuer non valido')
    except jwt.InvalidSignatureError:
        raise Exception('Firma non valida')

# Utilizzo
token = generate_jwt(12345, 'user@example.com', 'admin')
print(f"Token: {token}")

payload = verify_jwt(token)
print(f"User ID: {payload['sub']}")
print(f"Role: {payload['role']}")
```

7.4.6 JWT vs Session-based Auth

Aspetto	JWT	Session
Stato	Stateless	Stateful
Storage	Client-side	Server-side
Scalabilità	Eccellente	Richiede session store condiviso
Revoca	Difficile	Facile
Dimensione	Maggiore	Minore (solo ID)
Sicurezza	Token può essere rubato	Session ID può essere rubato

Tabella 7.2: Confronto JWT vs Session

7.5 OAuth 2.0

7.5.1 Introduzione

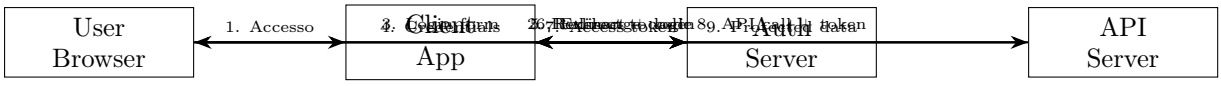
OAuth 2.0 è un framework di autorizzazione che permette ad applicazioni di terze parti di ottenere accesso limitato a servizi HTTP. È lo standard de facto per autorizzazione API.

7.5.2 Ruoli OAuth 2.0

- **Resource Owner**: L'utente che possiede i dati
- **Client**: L'applicazione che vuole accedere ai dati
- **Authorization Server**: Server che autentica e autorizza
- **Resource Server**: Server che ospita le risorse protette (API)

7.5.3 Authorization Code Flow

Il flow più sicuro per applicazioni web con backend.



7.5.4 Esempio Authorization Code Flow

Listing 7.11: Step 1: Redirect dell'utente

```
# L'applicazione reindirizza l'utente all'authorization server
https://auth.example.com/oauth/authorize?
  response_type=code&
  client_id=YOUR_CLIENT_ID&
  redirect_uri=https://yourapp.com/callback&
  scope=read:users write:posts&
  state=random_state_string
```

Listing 7.12: Step 2: Callback con authorization code

```
# Dopo il login, l'utente viene reindirizzato a:
https://yourapp.com/callback?
  code=AUTHORIZATION_CODE&
  state=random_state_string
```

Listing 7.13: Step 3: Scambio code con access token

```
curl -X POST https://auth.example.com/oauth/token \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "grant_type=authorization_code" \
  -d "code=AUTHORIZATION_CODE" \
  -d "redirect_uri=https://yourapp.com/callback" \
  -d "client_id=YOUR_CLIENT_ID" \
  -d "client_secret=YOUR_CLIENT_SECRET"
```

Listing 7.14: Response con Access Token

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "def50200a1b2c3d4e5f6...",
  "scope": "read:users write:posts"
}
```

7.5.5 Client Credentials Flow

Per comunicazioni machine-to-machine senza utente.

Listing 7.15: Client Credentials Request

```
curl -X POST https://auth.example.com/oauth/token \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "grant_type=client_credentials" \
  -d "client_id=YOUR_CLIENT_ID" \
  -d "client_secret=YOUR_CLIENT_SECRET" \
  -d "scope=api:read api:write"
```

7.5.6 Refresh Token Flow

Listing 7.16: Refresh Access Token

```
curl -X POST https://auth.example.com/oauth/token \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "grant_type=refresh_token" \
  -d "refresh_token=def50200a1b2c3d4e5f6..." \
  -d "client_id=YOUR_CLIENT_ID" \
  -d "client_secret=YOUR_CLIENT_SECRET"
```

7.5.7 Scopes e Permissions

Gli scope definiscono le permission richieste:

Listing 7.17: Esempi di Scopes

```
# Scopes granulari
read:users          # Leggere informazioni utenti
write:users         # Modificare utenti
delete:users        # Eliminare utenti
read:posts          # Leggere posts
write:posts         # Creare/modificare posts
admin:all           # Accesso amministrativo completo

```

```
# Scopes combinati
scope=read:users read:posts write:posts
```

7.6 API Keys

7.6.1 Utilizzo

Le API Key sono semplici token statici usati principalmente per:

- Identificare il client/applicazione
- Rate limiting per applicazione
- Tracking dell'utilizzo API

Nota Importante

Le API Keys **NON** dovrebbero essere usate per autenticazione utente, ma solo per identificare l'applicazione client.

7.6.2 Posizionamento API Key

Listing 7.18: API Key in Header (RACCOMANDATO)

```
curl -X GET https://api.example.com/data \
  -H "X-API-Key: your-api-key-here"
```

Listing 7.19: API Key in Query String (sconsigliato)

```
curl -X GET "https://api.example.com/data?api_key=your-api-key-here"
```

Perché Query String è Sconsigliato

- Le API key appaiono nei log del server
- Possono essere salvate nella cronologia del browser
- Vengono inviate nel Referer header
- Sono visibili nelle URL condivise

7.7 Best Practices di Sicurezza

7.7.1 HTTPS Obbligatorio

Regola Fondamentale

Tutte le API che richiedono autenticazione DEVONO usare HTTPS. Nessuna eccezione.

7.7.2 Token Expiration

Listing 7.20: Implementazione Token Expiration

```
{
  "access_token": "short-lived-token",
  "expires_in": 900,           // 15 minuti
  "refresh_token": "long-lived-token",
  "refresh_expires_in": 2592000  // 30 giorni
}
```

7.7.3 Token Revocation

Listing 7.21: Endpoint di Revoca Token

```
curl -X POST https://auth.example.com/oauth/revoke \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "token=access_or_refresh_token" \
  -d "client_id=YOUR_CLIENT_ID" \
  -d "client_secret=YOUR_CLIENT_SECRET"
```

7.7.4 Rate Limiting per Auth Endpoints

Listing 7.22: Response con Rate Limit su Login

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/json
Retry-After: 60

{
  "error": "too_many_requests",
  "error_description": "Too many login attempts. Try again in 60 seconds
     .",
  "retry_after": 60
}
```

7.7.5 Secure Storage

Storage dei Token

- **Web**: httpOnly cookies (per refresh token) + memoria (access token)
- **Mobile**: Keychain (iOS) / Keystore (Android)
- **Desktop**: Sistema di credential storage dell'OS
- **MAI**: localStorage senza encryption, plain text files

7.8 OpenAPI Specification

7.8.1 Definizione Security Schemes

Listing 7.23: OpenAPI 3.0 - Security Schemes

```
openapi: 3.0.0
info:
  title: Example API
```

```yaml
  version: 1.0.0

components:
  securitySchemes:
    # HTTP Basic Auth
    basicAuth:
      type: http
      scheme: basic

    # Bearer Token
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT

    # API Key
    apiKey:
      type: apiKey
      in: header
      name: X-API-Key

    # OAuth 2.0
    oauth2:
      type: oauth2
      flows:
        authorizationCode:
          authorizationUrl: https://auth.example.com/oauth/authorize
          tokenUrl: https://auth.example.com/oauth/token
          scopes:
            read:users: Read user information
            write:users: Modify user information
            admin:all: Full administrative access

        clientCredentials:
          tokenUrl: https://auth.example.com/oauth/token
          scopes:
            api:read: Read API data
            api:write: Write API data

# Security globale (applicata a tutti gli endpoint)
security:
  - bearerAuth: []

paths:
  /public/status:
    get:
      summary: Public endpoint
      security: []  # Override: nessuna autenticazione richiesta
      responses:
        '200':
          description: API status

  /users:
    get:
      summary: List users
      security:
        - bearerAuth: []
        - oauth2: [read:users]
```

```
      responses:
        '200':
          description: List of users
        '401':
          description: Unauthorized

  /admin/users:
    delete:
      summary: Delete user (admin only)
      security:
        - oauth2: [admin:all]
      responses:
        '204':
          description: User deleted
        '403':
          description: Forbidden - insufficient permissions
```

7.9 Esempi Pratici Completi

7.9.1 Postman Collection

Listing 7.24: Postman Collection per Authentication

```
{
  "info": {
    "name": "API Authentication Examples",
    "schema": "https://schema.getpostman.com/json/collection/v2.1.0/"
  },
  "auth": {
    "type": "bearer",
    "bearer": [
      {
        "key": "token",
        "value": "{{access_token}}",
        "type": "string"
      }
    ]
  },
  "item": [
    {
      "name": "Auth",
      "item": [
        {
          "name": "Login",
          "event": [
            {
              "listen": "test",
              "script": {
                "exec": [
                  "const response = pm.response.json();",
                  "pm.environment.set('access_token', response.
                     access_token);",
                  "pm.environment.set('refresh_token', response.
                     refresh_token);"
                ]
              }
            }
```

```
          ],
          "request": {
            "method": "POST",
            "header": [],
            "body": {
              "mode": "raw",
              "raw": "{\n  \"email\": \"user@example.com\",\n  \"
                 password\": \"password123\"\n}"
            },
            "url": "{{base_url}}/auth/login"
          }
        },
        {
          "name": "Refresh Token",
          "request": {
            "method": "POST",
            "header": [],
            "body": {
              "mode": "raw",
              "raw": "{\n  \"refresh_token\": \"{{refresh_token}}\"\n}"
            },
            "url": "{{base_url}}/auth/refresh"
          }
        }
      ]
    },
    {
      "name": "Protected Resources",
      "item": [
        {
          "name": "Get Current User",
          "request": {
            "method": "GET",
            "header": [],
            "url": "{{base_url}}/users/me"
          }
        }
      ]
    }
  ],
  "variable": [
    {
      "key": "base_url",
      "value": "https://api.example.com"
    }
  ]
}
```

7.9.2 Script di Test Completo

Listing 7.25: Bash Script per Testing Authentication

```
#!/bin/bash

API_BASE="https://api.example.com"
EMAIL="user@example.com"
PASSWORD="password123"
```

```
6
7  echo "=== API Authentication Test ==="
8
9  # 1. Login
10 echo -e "\n1. Login..."
11 LOGIN_RESPONSE=$(curl -s -X POST "$API_BASE/auth/login" \
12   -H "Content-Type: application/json" \
13   -d "{\"email\":\"$EMAIL\",\"password\":\"$PASSWORD\"}")
14
15 echo "$LOGIN_RESPONSE" | jq .
16
17 # Estrai token
18 ACCESS_TOKEN=$(echo "$LOGIN_RESPONSE" | jq -r '.access_token')
19 REFRESH_TOKEN=$(echo "$LOGIN_RESPONSE" | jq -r '.refresh_token')
20
21 if [ "$ACCESS_TOKEN" == "null" ]; then
22   echo "Login fallito!"
23   exit 1
24 fi
25
26 echo "Access Token: $ACCESS_TOKEN"
27
28 # 2. Richiesta autenticata
29 echo -e "\n2. Get current user..."
30 curl -s -X GET "$API_BASE/users/me" \
31   -H "Authorization: Bearer $ACCESS_TOKEN" | jq .
32
33 # 3. Test endpoint non autenticato
34 echo -e "\n3. Test senza token (should fail)..."
35 curl -s -X GET "$API_BASE/users/me" | jq .
36
37 # 4. Refresh token
38 echo -e "\n4. Refresh access token..."
39 REFRESH_RESPONSE=$(curl -s -X POST "$API_BASE/auth/refresh" \
40   -H "Content-Type: application/json" \
41   -d "{\"refresh_token\":\"$REFRESH_TOKEN\"}")
42
43 NEW_ACCESS_TOKEN=$(echo "$REFRESH_RESPONSE" | jq -r '.access_token')
44 echo "New Access Token: $NEW_ACCESS_TOKEN"
45
46 # 5. Logout
47 echo -e "\n5. Logout..."
48 curl -s -X POST "$API_BASE/auth/logout" \
49   -H "Authorization: Bearer $ACCESS_TOKEN" | jq .
50
51 echo -e "\n=== Test completato ==="
```

7.10 Riepilogo

Metodo	**Caso d'Uso**	**Pro**	**Contro**
Basic Auth	Admin tools, script interni	Semplice	Credenziali in ogni request
API Key	Identificazione app, public API	Facile da implementare	Non per auth utenti
Bearer Token	App web/mobile	Flessibile, revocabile	Richiede endpoint login
JWT	Microservizi, app distribuite	Stateless, self-contained	Difficile da revocare
OAuth 2.0	Accesso terze parti	Standard, granular permissions	Complesso

Tabella 7.3: Confronto Metodi di Autenticazione

Capitolo 8

Rate Limiting

Il rate limiting è una tecnica fondamentale per proteggere le API da abusi, attacchi DoS, e garantire un uso equo delle risorse. In questo capitolo esploreremo algoritmi, implementazioni e best practices.

8.1 Introduzione

8.1.1 Cos'è il Rate Limiting

Il rate limiting limita il numero di richieste che un client può effettuare in un determinato periodo di tempo.

Obiettivi del Rate Limiting

- **Protezione da abusi**: Prevenire spam e scraping massivo
- **Disponibilità**: Garantire che l'API rimanga disponibile per tutti
- **Controllo costi**: Limitare l'uso di risorse computazionali
- **Fairness**: Distribuire equamente le risorse tra gli utenti
- **Sicurezza**: Mitigare attacchi brute-force e DoS

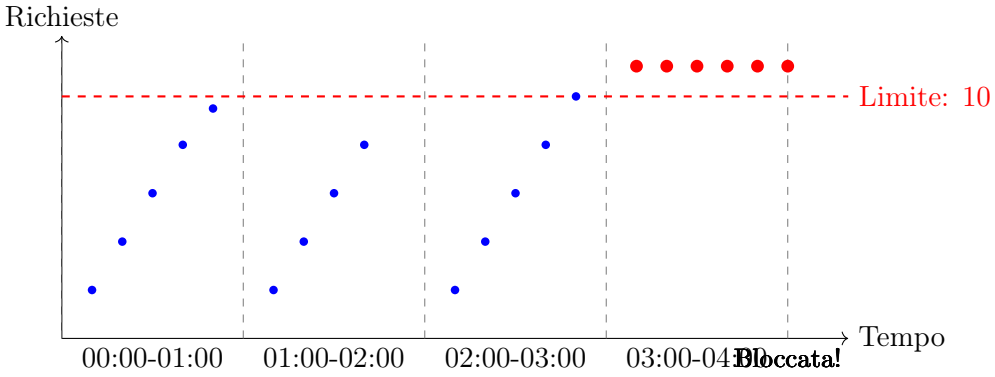
8.1.2 Livelli di Rate Limiting

1. **Per IP Address**: Limita richieste da un singolo IP
2. **Per User/API Key**: Limita per utente autenticato
3. **Per Endpoint**: Limiti diversi per endpoint diversi
4. **Globale**: Limite totale di richieste all'API
5. **Per Piano**: Limiti differenziati (free, pro, enterprise)

8.2 Algoritmi di Rate Limiting

8.2.1 Fixed Window Counter

L'algoritmo più semplice: conta le richieste in finestre temporali fisse.



Implementazione

Listing 8.1: Fixed Window in Python/Redis

```python
import time
import redis

class FixedWindowRateLimiter:
    def __init__(self, redis_client, window_seconds=60, max_requests=10):
        self.redis = redis_client
        self.window = window_seconds
        self.max_requests = max_requests

    def is_allowed(self, user_id):
        """Verifica se la richiesta e' consentita"""

        # Calcola inizio della finestra corrente
        current_window = int(time.time() / self.window)
        key = f"rate_limit:{user_id}:{current_window}"

        # Incrementa contatore
        current_count = self.redis.incr(key)

        # Imposta scadenza se e' la prima richiesta della finestra
        if current_count == 1:
            self.redis.expire(key, self.window)

        # Verifica limite
        return current_count <= self.max_requests

    def get_remaining(self, user_id):
        """Ritorna richieste rimanenti"""
        current_window = int(time.time() / self.window)
        key = f"rate_limit:{user_id}:{current_window}"
        current = int(self.redis.get(key) or 0)
        return max(0, self.max_requests - current)

# Utilizzo
redis_client = redis.Redis(host='localhost', port=6379, db=0)
limiter = FixedWindowRateLimiter(redis_client)

user_id = "user123"
if limiter.is_allowed(user_id):
    print("Richiesta consentita")
```

```
else:
    print("Rate limit superato")
```

Vantaggi e Svantaggi

Vantaggi

- Semplice da implementare
- Efficiente in termini di memoria
- Facile da capire

Svantaggi

- **Burst al confine delle finestre**: Un client può fare 2x richieste al confine
- Non uniforme: tutte le richieste all'inizio della finestra

8.2.2 Sliding Window Log

Mantiene un log delle timestamp di ogni richiesta.

Listing 8.2: Sliding Window Log con Redis

```
import time
import redis

class SlidingWindowLog:
    def __init__(self, redis_client, window_seconds=60, max_requests=10):
        self.redis = redis_client
        self.window = window_seconds
        self.max_requests = max_requests

    def is_allowed(self, user_id):
        """Verifica se la richiesta e' consentita"""
        key = f"rate_limit_log:{user_id}"
        now = time.time()
        window_start = now - self.window

        # Rimuovi richieste fuori dalla finestra
        self.redis.zremrangebyscore(key, 0, window_start)

        # Conta richieste nella finestra corrente
        current_count = self.redis.zcard(key)

        if current_count < self.max_requests:
            # Aggiungi timestamp corrente
            self.redis.zadd(key, {str(now): now})
            self.redis.expire(key, self.window)
            return True

        return False

    def get_remaining(self, user_id):
        """Ritorna richieste rimanenti"""
        key = f"rate_limit_log:{user_id}"
```

```
        now = time.time()
        window_start = now - self.window

        self.redis.zremrangebyscore(key, 0, window_start)
        current = self.redis.zcard(key)
        return max(0, self.max_requests - current)
```

Caratteristiche

- **Pro**: Precisione massima, nessun burst al confine
- **Contro**: Uso memoria elevato (memorizza ogni timestamp)

8.2.3 Sliding Window Counter

Combina Fixed Window e Sliding Window per efficienza e precisione.

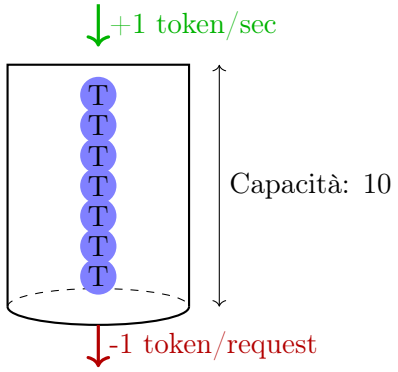
Listing 8.3: Sliding Window Counter

```
import time
import redis

class SlidingWindowCounter:
    def __init__(self, redis_client, window_seconds=60, max_requests=10):
        self.redis = redis_client
        self.window = window_seconds
        self.max_requests = max_requests

    def is_allowed(self, user_id):
        """
        Usa finestre da 1 minuto e calcola un peso
        basato sulla posizione nella finestra corrente
        """
        now = time.time()
        current_window = int(now / self.window)
        previous_window = current_window - 1

        # Chiavi per finestre
        current_key = f"rate_limit:{user_id}:{current_window}"
        previous_key = f"rate_limit:{user_id}:{previous_window}"

        # Conta nella finestra corrente
        current_count = int(self.redis.get(current_key) or 0)

        # Conta nella finestra precedente
        previous_count = int(self.redis.get(previous_key) or 0)

        # Calcola peso della finestra precedente
        elapsed_in_current = now % self.window
        weight = 1 - (elapsed_in_current / self.window)

        # Stima richieste con peso
        estimated_count = (previous_count * weight) + current_count

        if estimated_count < self.max_requests:
            # Incrementa contatore finestra corrente
            count = self.redis.incr(current_key)
```

```
            if count == 1:
                self.redis.expire(current_key, self.window * 2)
            return True

        return False
```

8.2.4 Token Bucket

Algoritmo basato su "token" che si riempiono nel tempo.



Listing 8.4: Token Bucket Implementation

```
import time
import redis

class TokenBucket:
    def __init__(self, redis_client, capacity=10, refill_rate=1):
        """
        capacity: numero massimo di token
        refill_rate: token aggiunti per secondo
        """
        self.redis = redis_client
        self.capacity = capacity
        self.refill_rate = refill_rate

    def is_allowed(self, user_id):
        """Verifica se ci sono token disponibili"""
        key = f"token_bucket:{user_id}"

        # Usa script Lua per atomicità
        lua_script = """
        local key = KEYS[1]
        local capacity = tonumber(ARGV[1])
        local refill_rate = tonumber(ARGV[2])
        local now = tonumber(ARGV[3])

        local bucket = redis.call('HMGET', key, 'tokens', 'last_refill')
        local tokens = tonumber(bucket[1])
        local last_refill = tonumber(bucket[2])

        if tokens == nil then
            tokens = capacity
            last_refill = now
        end

```

```
        -- Calcola token da aggiungere
        local elapsed = now - last_refill
        local tokens_to_add = elapsed * refill_rate
        tokens = math.min(capacity, tokens + tokens_to_add)

        -- Consuma un token se disponibile
        if tokens >= 1 then
            tokens = tokens - 1
            redis.call('HMSET', key, 'tokens', tokens, 'last_refill',
                now)
            redis.call('EXPIRE', key, 3600)
            return 1
        else
            return 0
        end
        """

        result = self.redis.eval(
            lua_script,
            1,
            key,
            self.capacity,
            self.refill_rate,
            time.time()
        )

        return result == 1

    def get_tokens(self, user_id):
        """Ritorna token attuali"""
        key = f"token_bucket:{user_id}"
        bucket = self.redis.hmget(key, 'tokens', 'last_refill')

        if bucket[0] is None:
            return self.capacity

        tokens = float(bucket[0])
        last_refill = float(bucket[1])
        elapsed = time.time() - last_refill
        tokens = min(self.capacity, tokens + (elapsed * self.refill_rate
            ))

        return int(tokens)
```

8.2.5 Leaky Bucket

Simile a Token Bucket ma con output a tasso costante.

Listing 8.5: Leaky Bucket Implementation

```
import time
import redis

class LeakyBucket:
    def __init__(self, redis_client, capacity=10, leak_rate=1):
        """
        capacity: dimensione bucket
        leak_rate: richieste processate per secondo
```

```
        """
        self.redis = redis_client
        self.capacity = capacity
        self.leak_rate = leak_rate

    def is_allowed(self, user_id):
        """Aggiungi richiesta al bucket"""
        key = f"leaky_bucket:{user_id}"

        lua_script = """
        local key = KEYS[1]
        local capacity = tonumber(ARGV[1])
        local leak_rate = tonumber(ARGV[2])
        local now = tonumber(ARGV[3])

        local bucket = redis.call('HMGET', key, 'level', 'last_leak')
        local level = tonumber(bucket[1]) or 0
        local last_leak = tonumber(bucket[2]) or now

        -- Calcola quanta acqua e' fuoriuscita
        local elapsed = now - last_leak
        local leaked = elapsed * leak_rate
        level = math.max(0, level - leaked)

        -- Prova ad aggiungere la richiesta
        if level < capacity then
            level = level + 1
            redis.call('HMSET', key, 'level', level, 'last_leak', now)
            redis.call('EXPIRE', key, 3600)
            return 1
        else
            return 0
        end
        """

        result = self.redis.eval(
            lua_script,
            1,
            key,
            self.capacity,
            self.leak_rate,
            time.time()
        )

        return result == 1
```

8.2.6 Confronto Algoritmi

8.3 HTTP Headers per Rate Limiting

8.3.1 Standard Headers

Listing 8.6: Headers di Rate Limiting (Draft IETF)

```
RateLimit-Limit: 100
RateLimit-Remaining: 85
RateLimit-Reset: 1699876543
```

Algoritmo	Pro	Contro	Caso d'Uso	Memoria
Fixed Window	Semplice, efficiente	Burst al confine	API semplici	Bassa
Sliding Log	Preciso	Alto uso memoria	API critiche	Alta
Sliding Counter	Bilanciato	Approssimazione	Uso generale	Media
Token Bucket	Permette burst	Complesso	API con spike	Media
Leaky Bucket	Output uniforme	Rigido	Rate costante	Media

Tabella 8.1: Confronto Algoritmi Rate Limiting

8.3.2 Headers Legacy (X- prefix)

Molte API usano ancora headers con prefisso `X-`:

Listing 8.7: Headers X-RateLimit-*

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 85
X-RateLimit-Reset: 1699876543
```

8.3.3 Descrizione Headers

Header	Descrizione
`RateLimit-Limit`	Numero massimo di richieste consentite nella finestra
`RateLimit-Remaining`	Numero di richieste rimanenti nella finestra corrente
`RateLimit-Reset`	Unix timestamp di quando il limite si resetterà
`Retry-After`	Secondi da attendere prima di riprovare (usato con 429)

Tabella 8.2: HTTP Headers Rate Limiting

8.3.4 Response di Successo

Listing 8.8: Response 200 con Rate Limit Headers

```
HTTP/1.1 200 OK
Content-Type: application/json
RateLimit-Limit: 100
RateLimit-Remaining: 85
RateLimit-Reset: 1699876543

{
  "data": {
    "users": [...]
  }
}
```

8.3.5 Response quando Limite Superato

Listing 8.9: Response 429 Too Many Requests

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/json
```

```
RateLimit-Limit: 100
RateLimit-Remaining: 0
RateLimit-Reset: 1699876543
Retry-After: 43

{
  "error": {
    "code": "rate_limit_exceeded",
    "message": "API rate limit exceeded",
    "limit": 100,
    "remaining": 0,
    "reset_at": "2023-11-13T12:35:43Z",
    "retry_after": 43
  }
}
```

8.4 Implementazione Middleware

8.4.1 Express.js Middleware

Listing 8.10: Rate Limiting Middleware Node.js

```
const redis = require('redis');
const client = redis.createClient();

const rateLimitMiddleware = (options = {}) => {
  const {
    windowMs = 60000,      // 1 minuto
    max = 100,             // 100 richieste
    keyGenerator = (req) => req.ip,
    handler = (req, res) => {
      res.status(429).json({
        error: 'Too many requests, please try again later.'
      });
    }
  } = options;

  return async (req, res, next) => {
    const key = `rate_limit:${keyGenerator(req)}`;
    const now = Date.now();
    const windowStart = now - windowMs;

    try {
      // Rimuovi entry vecchie
      await client.zRemRangeByScore(key, 0, windowStart);

      // Conta richieste nella finestra
      const count = await client.zCard(key);

      // Headers
      res.setHeader('RateLimit-Limit', max);
      res.setHeader('RateLimit-Remaining', Math.max(0, max - count - 1))
          ;
      res.setHeader('RateLimit-Reset',
        Math.ceil((now + windowMs) / 1000));

      if (count >= max) {
```

```
        res.setHeader('Retry-After', Math.ceil(windowMs / 1000));
        return handler(req, res);
      }

      // Aggiungi richiesta corrente
      await client.zAdd(key, { score: now, value: `${now}` });
      await client.expire(key, Math.ceil(windowMs / 1000));

      next();
    } catch (error) {
      console.error('Rate limit error:', error);
      next(); // Fail open in caso di errore
    }
  };
};

// Utilizzo
const express = require('express');
const app = express();

// Rate limit globale
app.use(rateLimitMiddleware({
  windowMs: 60000,
  max: 100
}));

// Rate limit specifico per login
app.post('/api/login',
  rateLimitMiddleware({
    windowMs: 300000,  // 5 minuti
    max: 5,            // 5 tentativi
    keyGenerator: (req) => req.body.email || req.ip
  }),
  (req, res) => {
    // Login logic
  }
);
```

8.4.2 Flask Decorator

Listing 8.11: Rate Limiting Decorator Python/Flask

```
from functools import wraps
from flask import request, jsonify
import redis
import time

redis_client = redis.Redis(host='localhost', port=6379, db=0)

def rate_limit(max_requests=100, window_seconds=60,
               key_func=lambda: request.remote_addr):
    """
    Decorator per rate limiting
    """
    def decorator(f):
        @wraps(f)
        def wrapped(*args, **kwargs):
```

```
            # Genera chiave
            key = f"rate_limit:{key_func()}"
            now = time.time()
            window_start = now - window_seconds

            # Rimuovi richieste fuori finestra
            redis_client.zremrangebyscore(key, 0, window_start)

            # Conta richieste
            current_count = redis_client.zcard(key)

            # Calcola reset time
            reset_time = int(now + window_seconds)

            # Headers
            from flask import make_response
            response = None

            if current_count >= max_requests:
                # Limite superato
                response = make_response(jsonify({
                    'error': 'rate_limit_exceeded',
                    'message': 'Too many requests',
                    'limit': max_requests,
                    'retry_after': window_seconds
                }), 429)
                response.headers['Retry-After'] = str(window_seconds)
            else:
                # Aggiungi richiesta
                redis_client.zadd(key, {str(now): now})
                redis_client.expire(key, window_seconds)

                # Esegui funzione
                response = make_response(f(*args, **kwargs))

            # Aggiungi headers rate limit
            response.headers['RateLimit-Limit'] = str(max_requests)
            response.headers['RateLimit-Remaining'] = str(
                max(0, max_requests - current_count - 1)
            )
            response.headers['RateLimit-Reset'] = str(reset_time)

            return response

        return wrapped
    return decorator

# Utilizzo
@app.route('/api/data')
@rate_limit(max_requests=100, window_seconds=60)
def get_data():
    return jsonify({'data': 'some data'})

@app.route('/api/login', methods=['POST'])
@rate_limit(
    max_requests=5,
    window_seconds=300,
    key_func=lambda: request.json.get('email', request.remote_addr)
```

```
)
def login():
    # Login logic
    return jsonify({'token': 'abc123'})
```

8.5 Rate Limiting Avanzato

8.5.1 Limiti per Piano di Servizio

Listing 8.12: Rate Limit basato su Piano Utente

```
RATE_LIMIT_TIERS = {
    'free': {
        'requests_per_hour': 100,
        'requests_per_day': 1000,
        'burst': 10
    },
    'pro': {
        'requests_per_hour': 1000,
        'requests_per_day': 20000,
        'burst': 50
    },
    'enterprise': {
        'requests_per_hour': 10000,
        'requests_per_day': 500000,
        'burst': 200
    }
}

def get_user_tier(user_id):
    """Ottieni tier dell'utente dal database"""
    # Query database
    user = db.users.find_one({'id': user_id})
    return user.get('tier', 'free')

def check_rate_limit(user_id):
    """Verifica rate limit con tier-based limits"""
    tier = get_user_tier(user_id)
    limits = RATE_LIMIT_TIERS[tier]

    # Verifica limite orario
    hourly_key = f"rate:{user_id}:hour:{int(time.time() / 3600)}"
    hourly_count = redis_client.incr(hourly_key)
    redis_client.expire(hourly_key, 3600)

    if hourly_count > limits['requests_per_hour']:
        return False, 'hourly_limit_exceeded'

    # Verifica limite giornaliero
    daily_key = f"rate:{user_id}:day:{int(time.time() / 86400)}"
    daily_count = redis_client.incr(daily_key)
    redis_client.expire(daily_key, 86400)

    if daily_count > limits['requests_per_day']:
        return False, 'daily_limit_exceeded'

    return True, None
```

8.5.2 Limiti per Endpoint

Listing 8.13: Rate Limit Differenziato per Endpoint

```
ENDPOINT_LIMITS = {
    '/api/search': {
        'max_requests': 20,
        'window_seconds': 60
    },
    '/api/upload': {
        'max_requests': 5,
        'window_seconds': 60
    },
    '/api/users': {
        'max_requests': 100,
        'window_seconds': 60
    },
    'default': {
        'max_requests': 60,
        'window_seconds': 60
    }
}

def get_endpoint_limit(endpoint):
    """Ottieni configurazione rate limit per endpoint"""
    return ENDPOINT_LIMITS.get(endpoint, ENDPOINT_LIMITS['default'])

@app.before_request
def check_endpoint_rate_limit():
    """Middleware per rate limiting per endpoint"""
    endpoint = request.endpoint
    user_id = get_current_user_id()

    config = get_endpoint_limit(endpoint)
    key = f"rate:{user_id}:{endpoint}"

    if not is_allowed(key, config['max_requests'],
                      config['window_seconds']):
        return jsonify({
            'error': 'rate_limit_exceeded',
            'endpoint': endpoint,
            'limit': config['max_requests']
        }), 429
```

8.5.3 Limiti Dinamici

Listing 8.14: Rate Limiting Dinamico Basato su Carico

```
def get_dynamic_rate_limit():
    """Calcola rate limit dinamico basato sul carico server"""

    # Ottieni metriche sistema
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent

    # Calcola limite basato su carico
    if cpu_usage > 80 or memory_usage > 80:
        # Carico alto: riduci limite
```

```
        return 50
    elif cpu_usage > 60 or memory_usage > 60:
        # Carico medio
        return 100
    else:
        # Carico basso: limite normale
        return 200

def adaptive_rate_limit(user_id):
    """Rate limiting adattivo"""
    limit = get_dynamic_rate_limit()
    key = f"rate:adaptive:{user_id}"

    return is_allowed(key, limit, window_seconds=60)
```

8.6 Best Practices

8.6.1 Comunicazione Chiara

Documentare i Limiti

- Documentare chiaramente i rate limits nella API docs
- Includere limiti per ogni piano di servizio
- Specificare come vengono contate le richieste
- Fornire esempi di headers di risposta

8.6.2 Graceful Degradation

Listing 8.15: Response con Suggerimenti

```
{
  "error": {
    "code": "rate_limit_exceeded",
    "message": "You have exceeded your rate limit",
    "current_usage": 1050,
    "limit": 1000,
    "reset_at": "2023-11-13T13:00:00Z",
    "suggestions": [
      "Wait 15 minutes before retrying",
      "Upgrade to Pro plan for higher limits",
      "Use webhooks instead of polling"
    ],
    "documentation_url": "https://docs.example.com/rate-limits"
  }
}
```

8.6.3 Monitoring

Listing 8.16: Logging e Metriche Rate Limiting

```
import logging
from prometheus_client import Counter, Histogram

```

```
# Metriche Prometheus
rate_limit_exceeded = Counter(
    'rate_limit_exceeded_total',
    'Total rate limit exceeded events',
    ['user_tier', 'endpoint']
)

rate_limit_usage = Histogram(
    'rate_limit_usage_percentage',
    'Rate limit usage percentage',
    ['user_tier']
)

def log_rate_limit_event(user_id, tier, endpoint, exceeded):
    """Log eventi rate limiting"""

    if exceeded:
        logging.warning(
            f"Rate limit exceeded - User: {user_id}, "
            f"Tier: {tier}, Endpoint: {endpoint}"
        )
        rate_limit_exceeded.labels(
            user_tier=tier,
            endpoint=endpoint
        ).inc()

    # Log usage
    usage = get_current_usage(user_id)
    limit = get_limit(tier)
    percentage = (usage / limit) * 100

    rate_limit_usage.labels(user_tier=tier).observe(percentage)
\end{lstlisting>

\section{OpenAPI Specification}

\begin{lstlisting}[caption=Documentare Rate Limiting in OpenAPI]
openapi: 3.0.0
info:
  title: Rate Limited API
  version: 1.0.0
  description: |
    ## Rate Limiting

    This API implements rate limiting to ensure fair usage.

    ### Limits by Plan

    | Plan       | Requests/Hour | Requests/Day |
    |------------|---------------|--------------|
    | Free       | 100           | 1,000        |
    | Pro        | 1,000         | 20,000       |
    | Enterprise | 10,000        | 500,000      |

    ### Headers

    All responses include rate limit headers:
    - `RateLimit-Limit`: Maximum requests allowed
```

```
    - `RateLimit-Remaining`: Requests remaining
    - `RateLimit-Reset`: Unix timestamp of reset

    ### 429 Response

    When limit is exceeded, API returns 429 status with Retry-After
       header.

paths:
  /api/search:
    get:
      summary: Search resources
      description: |
        Search endpoint with stricter rate limiting (20 req/min)
      responses:
        '200':
          description: Search results
          headers:
            RateLimit-Limit:
              schema:
                type: integer
              description: Maximum requests per window
            RateLimit-Remaining:
              schema:
                type: integer
              description: Remaining requests in window
            RateLimit-Reset:
              schema:
                type: integer
              description: Unix timestamp when limit resets
          content:
            application/json:
              schema:
                type: object

        '429':
          description: Rate limit exceeded
          headers:
            RateLimit-Limit:
              schema:
                type: integer
            RateLimit-Remaining:
              schema:
                type: integer
              example: 0
            RateLimit-Reset:
              schema:
                type: integer
            Retry-After:
              schema:
                type: integer
              description: Seconds to wait before retrying
          content:
            application/json:
              schema:
                type: object
                properties:
                  error:
```

```yaml
                    type: object
                    properties:
                      code:
                        type: string
                        example: rate_limit_exceeded
                      message:
                        type: string
                      limit:
                        type: integer
                      reset_at:
                        type: string
                        format: date-time
```

8.7 Testing

8.7.1 Script di Test

Listing 8.17: Test Rate Limiting con cURL

```bash
#!/bin/bash

API_URL="https://api.example.com/data"
TOKEN="your-api-token"

echo "=== Rate Limit Test ==="

# Invia richieste fino al limite
for i in {1..105}; do
  echo "Request #$i"

  response=$(curl -s -w "\n%{http_code}" \
    -H "Authorization: Bearer $TOKEN" \
    "$API_URL")

  status_code=$(echo "$response" | tail -n1)
  body=$(echo "$response" | head -n-1)

  # Estrai headers (richiede -i flag in curl)
  limit=$(echo "$body" | jq -r '.headers."RateLimit-Limit" // empty')
  remaining=$(echo "$body" | jq -r '.headers."RateLimit-Remaining" //
     empty')

  echo "  Status: $status_code"
  echo "  Remaining: $remaining"

  if [ "$status_code" == "429" ]; then
    echo "  RATE LIMIT HIT!"
    retry_after=$(echo "$body" | jq -r '.error.retry_after')
    echo "  Retry after: $retry_after seconds"
    break
  fi

  # Pausa breve
  sleep 0.1
done
```

8.7.2 Test Unitari

Listing 8.18: Unit Test Rate Limiter

```python
import unittest
import time
from rate_limiter import FixedWindowRateLimiter

class TestRateLimiter(unittest.TestCase):
    def setUp(self):
        self.redis = fakeredis.FakeStrictRedis()
        self.limiter = FixedWindowRateLimiter(
            self.redis,
            window_seconds=60,
            max_requests=10
        )

    def test_allows_requests_under_limit(self):
        """Verifica che richieste sotto il limite siano consentite"""
        user_id = "test_user"

        for i in range(10):
            self.assertTrue(self.limiter.is_allowed(user_id))

    def test_blocks_requests_over_limit(self):
        """Verifica che richieste oltre il limite siano bloccate"""
        user_id = "test_user"

        # Riempi il limite
        for i in range(10):
            self.limiter.is_allowed(user_id)

        # La 11esima deve essere bloccata
        self.assertFalse(self.limiter.is_allowed(user_id))

    def test_resets_after_window(self):
        """Verifica reset dopo la finestra"""
        user_id = "test_user"

        # Riempi limite
        for i in range(10):
            self.limiter.is_allowed(user_id)

        # Simula passaggio tempo
        time.sleep(61)

        # Deve permettere di nuovo
        self.assertTrue(self.limiter.is_allowed(user_id))

if __name__ == '__main__':
    unittest.main()
```

8.8 Riepilogo

Checklist Rate Limiting

- [ ] Scegli algoritmo appropriato per il tuo caso d'uso
- [ ] Implementa rate limiting a più livelli (IP, user, endpoint)
- [ ] Usa headers standard (RateLimit-*)
- [ ] Fornisci response 429 chiare con Retry-After
- [ ] Documenta limiti chiaramente
- [ ] Monitora e logga eventi di rate limiting
- [ ] Implementa limiti differenziati per piani
- [ ] Testa il comportamento sotto carico
- [ ] Considera fail-open per garantire disponibilità
- [ ] Fornisci meccanismi di upgrade/whitelist

Capitolo 9

Paginazione e Filtraggio

La paginazione e il filtraggio sono essenziali per gestire grandi quantità di dati in modo efficiente. Questo capitolo esplora le diverse strategie, best practices e implementazioni.

9.1 Introduzione

9.1.1 Perché Paginare

Problemi Senza Paginazione

- **Performance**: Query che ritornano milioni di record sono lente
- **Memoria**: Caricare tutti i dati in memoria può causare crash
- **Network**: Trasferire grandi dataset consuma banda
- **UX**: L'utente non può elaborare migliaia di risultati insieme
- **Timeout**: Richieste troppo lunghe possono andare in timeout

9.1.2 Tipi di Paginazione

1. **Offset-based**: Usa offset e limit (tradizionale)
2. **Cursor-based**: Usa puntatori opachi (efficiente per stream)
3. **Page-based**: Numerazione pagine (user-friendly)
4. **Keyset-based**: Usa chiavi ordinabili (efficiente e stabile)

9.2 Offset-based Pagination

9.2.1 Concetto

La paginazione basata su offset usa due parametri:

- `limit`: numero di elementi per pagina
- `offset`: numero di elementi da saltare

9.2.2 Esempio Base

Listing 9.1: Request con Offset e Limit

```
# Prima pagina (elementi 0-9)
curl "https://api.example.com/users?limit=10&offset=0"

# Seconda pagina (elementi 10-19)
curl "https://api.example.com/users?limit=10&offset=10"

# Terza pagina (elementi 20-29)
curl "https://api.example.com/users?limit=10&offset=20"
```

9.2.3 Struttura Response

Listing 9.2: Response Paginata Completa

```
{
  "data": [
    {
      "id": 10,
      "name": "Mario Rossi",
      "email": "mario@example.com"
    },
    {
      "id": 11,
      "name": "Laura Bianchi",
      "email": "laura@example.com"
    }
  ],
  "pagination": {
    "total": 1543,
    "count": 10,
    "per_page": 10,
    "current_page": 2,
    "total_pages": 155,
    "links": {
      "first": "https://api.example.com/users?limit=10&offset=0",
      "prev": "https://api.example.com/users?limit=10&offset=0",
      "self": "https://api.example.com/users?limit=10&offset=10",
      "next": "https://api.example.com/users?limit=10&offset=20",
      "last": "https://api.example.com/users?limit=10&offset=1540"
    }
  }
}
```

9.2.4 Implementazione SQL

Listing 9.3: Query con LIMIT e OFFSET

```
-- PostgreSQL / MySQL
SELECT *
FROM users
ORDER BY id
LIMIT 10 OFFSET 20;

-- SQL Server
```

```sql
SELECT *
FROM users
ORDER BY id
OFFSET 20 ROWS
FETCH NEXT 10 ROWS ONLY;
```

9.2.5 Implementazione Backend

Listing 9.4: Offset Pagination in Python/Flask

```python
from flask import Flask, request, jsonify
from sqlalchemy import func

app = Flask(__name__)

@app.route('/api/users')
def get_users():
    # Parametri paginazione
    page = request.args.get('page', 1, type=int)
    per_page = request.args.get('per_page', 10, type=int)

    # Validazione
    per_page = min(per_page, 100)  # Max 100 per pagina
    offset = (page - 1) * per_page

    # Query
    total = db.session.query(func.count(User.id)).scalar()
    users = User.query\
        .order_by(User.id)\
        .limit(per_page)\
        .offset(offset)\
        .all()

    # Calcola pagine totali
    total_pages = (total + per_page - 1) // per_page

    # Costruisci links
    base_url = request.base_url
    links = {
        'self': f"{base_url}?page={page}&per_page={per_page}",
        'first': f"{base_url}?page=1&per_page={per_page}",
        'last': f"{base_url}?page={total_pages}&per_page={per_page}"
    }

    if page > 1:
        links['prev'] = f"{base_url}?page={page-1}&per_page={per_page}"

    if page < total_pages:
        links['next'] = f"{base_url}?page={page+1}&per_page={per_page}"

    return jsonify({
        'data': [user.to_dict() for user in users],
        'pagination': {
            'total': total,
            'count': len(users),
            'per_page': per_page,
            'current_page': page,
```

```
            'total_pages': total_pages,
            'links': links
        }
    })
```

9.2.6 Vantaggi e Svantaggi

Vantaggi

- Facile da implementare
- Supporta salto a pagine specifiche
- Intuitivo per l'utente
- Calcolo facile delle pagine totali

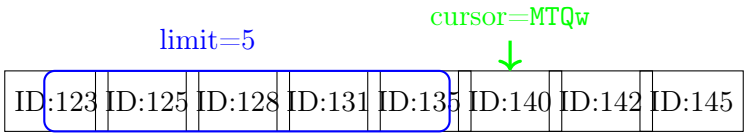
Svantaggi

- **Performance degradata**: OFFSET alto è lento (deve scansionare record precedenti)
- **Inconsistenza**: Se i dati cambiano, elementi possono essere duplicati o saltati
- **Non scalabile**: Con milioni di record diventa inutilizzabile

9.3 Cursor-based Pagination

9.3.1 Concetto

Usa un cursore (puntatore opaco) per identificare la posizione corrente. Ogni pagina include un cursore per la pagina successiva.



9.3.2 Esempio Request

Listing 9.5: Cursor-based Pagination

```
# Prima pagina
curl "https://api.example.com/users?limit=10"

# Pagina successiva (usa cursor dalla response)
curl "https://api.example.com/users?limit=10&cursor=eyJpZCI6MTB9"

# Cursore e' opaco (es. Base64 di {"id": 10})
echo "eyJpZCI6MTB9" | base64 -d
# Output: {"id":10}
```

9.3.3 Struttura Response

Listing 9.6: Response con Cursori

```
{
  "data": [
    {"id": 1, "name": "User 1"},
    {"id": 2, "name": "User 2"},
    {"id": 5, "name": "User 5"},
    {"id": 7, "name": "User 7"},
    {"id": 10, "name": "User 10"}
  ],
  "paging": {
    "cursors": {
      "before": "eyJpZCI6MX0=",
      "after": "eyJpZCI6MTB9"
    },
    "next": "https://api.example.com/users?limit=10&cursor=eyJpZCI6MTB9
       ",
    "has_next": true
  }
}
```

9.3.4 Implementazione

Listing 9.7: Cursor Pagination Implementation

```
import base64
import json
from flask import Flask, request, jsonify

def encode_cursor(data):
    """Codifica dati in cursore opaco"""
    json_str = json.dumps(data)
    return base64.b64encode(json_str.encode()).decode()

def decode_cursor(cursor):
    """Decodifica cursore"""
    try:
        json_str = base64.b64decode(cursor.encode()).decode()
        return json.loads(json_str)
    except:
        return None

@app.route('/api/users')
def get_users_cursor():
    limit = request.args.get('limit', 10, type=int)
    limit = min(limit, 100)  # Max 100

    cursor = request.args.get('cursor')

    # Costruisci query
    query = User.query.order_by(User.id)

    # Applica cursore se presente
    if cursor:
        cursor_data = decode_cursor(cursor)
        if cursor_data and 'id' in cursor_data:
            query = query.filter(User.id > cursor_data['id'])

```

```
    # Fetch limit + 1 per sapere se ci sono piu' risultati
    users = query.limit(limit + 1).all()

    # Check se ci sono piu' pagine
    has_next = len(users) > limit
    if has_next:
        users = users[:limit]

    # Costruisci response
    data = [user.to_dict() for user in users]

    response = {'data': data}

    if users:
        # Cursori
        first_cursor = encode_cursor({'id': users[0].id - 1})
        last_cursor = encode_cursor({'id': users[-1].id})

        # Next URL
        next_url = None
        if has_next:
            next_url = f"{request.base_url}?limit={limit}&cursor={
               last_cursor}"

        response['paging'] = {
            'cursors': {
                'before': first_cursor,
                'after': last_cursor
            },
            'next': next_url,
            'has_next': has_next
        }

    return jsonify(response)
```

9.3.5 Query SQL con Cursor

Listing 9.8: Cursor-based SQL Query

```
-- Assumendo cursor = {"id": 100}
SELECT *
FROM users
WHERE id > 100
ORDER BY id
LIMIT 10;

-- Per cursore con timestamp
SELECT *
FROM posts
WHERE created_at < '2023-11-13 12:00:00'
ORDER BY created_at DESC
LIMIT 20;
```

9.3.6 Vantaggi e Svantaggi

Vantaggi

- Performance costante (usa indici)
- Nessun problema con dati che cambiano
- Scalabile a grandi dataset
- Ideale per stream infiniti

Svantaggi

- Non puoi saltare a pagine arbitrarie
- Nessun conteggio totale
- Più complesso da implementare
- Richiede campo ordinabile univoco

9.4 Page-based Pagination

9.4.1 Concetto

Simile a offset ma usa numero di pagina invece di offset.

Listing 9.9: Page Number Pagination

```
# Pagina 1
curl "https://api.example.com/users?page=1&per_page=10"

# Pagina 2
curl "https://api.example.com/users?page=2&per_page=10"

# Equivale a: offset = (page - 1) * per_page
```

9.4.2 Link Headers (RFC 5988)

Listing 9.10: Paginazione con Link Headers

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://api.example.com/users?page=1&per_page=10>; rel="first",
      <https://api.example.com/users?page=3&per_page=10>; rel="prev",
      <https://api.example.com/users?page=5&per_page=10>; rel="next",
      <https://api.example.com/users?page=155&per_page=10>; rel="last"

{
  "data": [...]
}
\end{lstlisting>

\subsection{Implementazione Link Headers}

\begin{lstlisting}[language=Python, caption=Generazione Link Headers]
from flask import Response
```

```python
import json

def generate_link_header(base_url, page, per_page, total_pages):
    """Genera Link header per paginazione"""
    links = []

    # First
    links.append(f'<{base_url}?page=1&per_page={per_page}>; rel="first"')

    # Prev
    if page > 1:
        links.append(
            f'<{base_url}?page={page-1}&per_page={per_page}>; rel="prev"'
        )

    # Next
    if page < total_pages:
        links.append(
            f'<{base_url}?page={page+1}&per_page={per_page}>; rel="next"'
        )

    # Last
    links.append(
        f'<{base_url}?page={total_pages}&per_page={per_page}>; rel="last"'
    )

    return ', '.join(links)

@app.route('/api/users')
def get_users_with_link_header():
    page = request.args.get('page', 1, type=int)
    per_page = request.args.get('per_page', 10, type=int)

    # Query (vedi implementazione precedente)
    # ...

    # Crea response
    response_data = json.dumps({'data': data})
    response = Response(response_data, mimetype='application/json')

    # Aggiungi Link header
    link_header = generate_link_header(
        request.base_url, page, per_page, total_pages
    )
    response.headers['Link'] = link_header

    return response
```

9.5 Keyset Pagination

9.5.1 Concetto

Usa valori della chiave di ordinamento invece di offset.

Listing 9.11: Keyset Pagination

```
# Prima richiesta
curl "https://api.example.com/posts?limit=10&sort=created_at"

# Richieste successive usano l'ultimo valore
curl "https://api.example.com/posts?limit=10&sort=created_at&after
    =2023-11-13T12:00:00Z"
```

9.5.2 Implementazione

Listing 9.12: Keyset SQL Query

```
-- Prima pagina
SELECT id, title, created_at
FROM posts
ORDER BY created_at DESC, id DESC
LIMIT 10;

-- Pagine successive (assumendo ultimo created_at = '2023-11-13
    12:00:00', id = 1543)
SELECT id, title, created_at
FROM posts
WHERE (created_at, id) < ('2023-11-13 12:00:00', 1543)
ORDER BY created_at DESC, id DESC
LIMIT 10;
```

Listing 9.13: Keyset Pagination in Python

```
@app.route('/api/posts')
def get_posts_keyset():
    limit = request.args.get('limit', 10, type=int)
    after_time = request.args.get('after_time')
    after_id = request.args.get('after_id', type=int)

    query = Post.query.order_by(
        Post.created_at.desc(),
        Post.id.desc()
    )

    # Applica keyset
    if after_time and after_id:
        query = query.filter(
            db.or_(
                Post.created_at < after_time,
                db.and_(
                    Post.created_at == after_time,
                    Post.id < after_id
                )
            )
        )

    posts = query.limit(limit + 1).all()

    has_next = len(posts) > limit
    if has_next:
        posts = posts[:limit]

```

```python
    # Response
    data = [post.to_dict() for post in posts]

    response = {'data': data}

    if posts and has_next:
        last_post = posts[-1]
        next_url = (
            f"{request.base_url}?limit={limit}"
            f"&after_time={last_post.created_at.isoformat()}"
            f"&after_id={last_post.id}"
        )
        response['next'] = next_url

    return jsonify(response)
```

9.6 Filtraggio

9.6.1 Query Parameters per Filtri

Listing 9.14: Esempi di Filtraggio

```
# Filtro semplice
curl "https://api.example.com/users?status=active"

# Filtri multipli
curl "https://api.example.com/users?status=active&role=admin"

# Filtro con range
curl "https://api.example.com/products?price_min=10&price_max=100"

# Filtro con data
curl "https://api.example.com/posts?created_after=2023-01-01"

# Filtro con pattern matching
curl "https://api.example.com/users?email_like=@gmail.com"
```

9.6.2 Operatori di Filtro

Operatore	Query Param	Esempio
Uguale	`field=value`	`status=active`
Non uguale	`field[ne]=value`	`status[ne]=deleted`
Maggiore	`field[gt]=value`	`age[gt]=18`
Maggiore/uguale	`field[gte]=value`	`price[gte]=100`
Minore	`field[lt]=value`	`stock[lt]=10`
Minore/uguale	`field[lte]=value`	`created[lte]=2023-12-31`
In lista	`field[in]=v1,v2`	`status[in]=active,pending`
Like	`field[like]=pattern`	`name[like]=%john%`

Tabella 9.1: Operatori di Filtro Comuni

9.6.3 Implementazione Filtri Dinamici

Listing 9.15: Sistema di Filtri Flessibile

```
from sqlalchemy import and_, or_

class FilterBuilder:
    """Costruisce filtri dinamici da query parameters"""

    # Mappa operatori a funzioni SQLAlchemy
    OPERATORS = {
        'eq': lambda field, value: field == value,
        'ne': lambda field, value: field != value,
        'gt': lambda field, value: field > value,
        'gte': lambda field, value: field >= value,
        'lt': lambda field, value: field < value,
        'lte': lambda field, value: field <= value,
        'in': lambda field, value: field.in_(value.split(',')),
        'like': lambda field, value: field.like(value),
        'ilike': lambda field, value: field.ilike(value),
    }

    # Campi filtrabili (whitelist per sicurezza)
    FILTERABLE_FIELDS = {
        'User': ['status', 'role', 'email', 'created_at'],
        'Product': ['category', 'price', 'in_stock']
    }

    @classmethod
    def build_filters(cls, model, query_params):
        """Costruisce filtri da query parameters"""
        filters = []
        model_name = model.__name__

        for param, value in query_params.items():
            # Parse parametro: field[operator] o field
            if '[' in param:
                field_name, operator = param.replace(']', '').split('[')
            else:
                field_name = param
                operator = 'eq'

            # Validazione campo
            if field_name not in cls.FILTERABLE_FIELDS.get(model_name,
                []):
                continue

            # Ottieni campo del model
            field = getattr(model, field_name, None)
            if not field:
                continue

            # Applica operatore
            if operator in cls.OPERATORS:
                filter_func = cls.OPERATORS[operator]
                filters.append(filter_func(field, value))

        return filters

```

```python
@app.route('/api/users')
def get_filtered_users():
    # Costruisci filtri da query params
    filters = FilterBuilder.build_filters(User, request.args)

    # Applica filtri
    query = User.query
    if filters:
        query = query.filter(and_(*filters))

    # Paginazione
    page = request.args.get('page', 1, type=int)
    per_page = request.args.get('per_page', 10, type=int)

    # Execute
    users = query.paginate(page=page, per_page=per_page)

    return jsonify({
        'data': [user.to_dict() for user in users.items],
        'pagination': {
            'page': page,
            'per_page': per_page,
            'total': users.total,
            'pages': users.pages
        }
    })
```

9.6.4 Esempio Completo con Filtri

Listing 9.16: Query Complessa

```
curl -G "https://api.example.com/products" \
  --data-urlencode "category=electronics" \
  --data-urlencode "price[gte]=100" \
  --data-urlencode "price[lte]=500" \
  --data-urlencode "in_stock=true" \
  --data-urlencode "brand[in]=Sony,Samsung,LG" \
  --data-urlencode "name[like]=%TV%" \
  --data-urlencode "sort=-price,name" \
  --data-urlencode "page=2" \
  --data-urlencode "per_page=20"
```

9.7 Ordinamento

9.7.1 Query Parameters per Sort

Listing 9.17: Parametri di Ordinamento

```
# Ordinamento singolo (ascendente)
curl "https://api.example.com/users?sort=name"

# Ordinamento discendente (prefisso -)
curl "https://api.example.com/users?sort=-created_at"

# Ordinamento multiplo
curl "https://api.example.com/users?sort=status,-created_at,name"
```

9.7.2 Implementazione Sort

Listing 9.18: Dynamic Sorting

```
class SortBuilder:
    """Costruisce ordinamenti dinamici"""

    # Whitelist campi ordinabili
    SORTABLE_FIELDS = {
        'User': ['id', 'name', 'email', 'created_at', 'status'],
        'Product': ['id', 'name', 'price', 'created_at']
    }

    @classmethod
    def build_sort(cls, model, sort_param):
        """
        Costruisce ordinamento da parametro sort
        Formato: campo1,-campo2,campo3
        Prefisso - indica discendente
        """
        if not sort_param:
            return []

        model_name = model.__name__
        sort_fields = []

        for field_expr in sort_param.split(','):
            # Determina direzione
            if field_expr.startswith('-'):
                field_name = field_expr[1:]
                direction = 'desc'
            else:
                field_name = field_expr
                direction = 'asc'

            # Validazione
            if field_name not in cls.SORTABLE_FIELDS.get(model_name, [])
                :
                continue

            # Ottieni campo
            field = getattr(model, field_name, None)
            if not field:
                continue

            # Aggiungi ordinamento
            if direction == 'desc':
                sort_fields.append(field.desc())
            else:
                sort_fields.append(field.asc())

        return sort_fields

@app.route('/api/users')
def get_sorted_users():
    # Ordinamento
    sort_param = request.args.get('sort')
    sort_fields = SortBuilder.build_sort(User, sort_param)

```

```
    # Query
    query = User.query
    if sort_fields:
        query = query.order_by(*sort_fields)
    else:
        query = query.order_by(User.id)  # Default

    # ... paginazione ...
```

9.8 Field Selection (Sparse Fieldsets)

9.8.1 Concetto

Permette al client di specificare quali campi vuole ricevere.

Listing 9.19: Field Selection

```
# Solo alcuni campi
curl "https://api.example.com/users?fields=id,name,email"

# Campi per risorse correlate
curl "https://api.example.com/posts?fields[posts]=id,title&fields[author
    ]=name,email"
```

9.8.2 Implementazione

Listing 9.20: Dynamic Field Selection

```
def select_fields(obj, fields):
    """Seleziona solo i campi richiesti"""
    if not fields:
        return obj.to_dict()

    field_list = fields.split(',')
    result = {}

    for field in field_list:
        if hasattr(obj, field):
            value = getattr(obj, field)
            # Converti datetime, etc
            if isinstance(value, datetime):
                value = value.isoformat()
            result[field] = value

    return result

@app.route('/api/users')
def get_users_sparse():
    fields = request.args.get('fields')

    users = User.query.limit(10).all()

    data = [select_fields(user, fields) for user in users]

    return jsonify({'data': data})
```

9.9 HATEOAS e Link Navigation

9.9.1 Hypermedia Controls

Listing 9.21: Response con HATEOAS Links

```
{
  "data": [
    {
      "id": 1,
      "name": "Product 1",
      "price": 99.99,
      "_links": {
        "self": {
          "href": "https://api.example.com/products/1"
        },
        "category": {
          "href": "https://api.example.com/categories/electronics"
        },
        "reviews": {
          "href": "https://api.example.com/products/1/reviews"
        }
      }
    }
  ],
  "_links": {
    "self": {
      "href": "https://api.example.com/products?page=2"
    },
    "first": {
      "href": "https://api.example.com/products?page=1"
    },
    "prev": {
      "href": "https://api.example.com/products?page=1"
    },
    "next": {
      "href": "https://api.example.com/products?page=3"
    },
    "last": {
      "href": "https://api.example.com/products?page=50"
    }
  },
  "_meta": {
    "total_items": 500,
    "item_count": 10,
    "items_per_page": 10,
    "total_pages": 50,
    "current_page": 2
  }
}
```

9.9.2 HAL (Hypertext Application Language)

Listing 9.22: HAL Format Response

```
{
  "_links": {
    "self": {"href": "/orders/523"},
```

```
    "warehouse": {"href": "/warehouses/56"},
    "invoice": {"href": "/invoices/873"}
  },
  "currency": "USD",
  "status": "shipped",
  "total": 10.20,
  "_embedded": {
    "items": [
      {
        "_links": {"self": {"href": "/products/123"}},
        "name": "Widget",
        "price": 5.10,
        "quantity": 2
      }
    ]
  }
}
```

9.10 Best Practices

9.10.1 Limiti Ragionevoli

Listing 9.23: Validazione Parametri Paginazione

```python
def validate_pagination_params(page, per_page):
    """Valida e normalizza parametri paginazione"""

    # Limiti di sicurezza
    MAX_PER_PAGE = 100
    DEFAULT_PER_PAGE = 10

    # Normalizza
    page = max(1, page)
    per_page = min(max(1, per_page), MAX_PER_PAGE)

    # Warn se al massimo
    if per_page >= MAX_PER_PAGE:
        logger.warning(
            f"Client requested max pagination size: {per_page}"
        )

    return page, per_page
```

9.10.2 Caching

Listing 9.24: Cache Headers per Risposte Paginate

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: public, max-age=60
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
Vary: Accept, Accept-Encoding

{
  "data": [...]
}
```

9.10.3 Documentazione Parametri

Listing 9.25: OpenAPI Schema per Paginazione

```yaml
openapi: 3.0.0
paths:
  /users:
    get:
      summary: List users
      parameters:
        - name: page
          in: query
          description: Page number (1-indexed)
          schema:
            type: integer
            minimum: 1
            default: 1

        - name: per_page
          in: query
          description: Items per page
          schema:
            type: integer
            minimum: 1
            maximum: 100
            default: 10

        - name: sort
          in: query
          description: |
            Sort fields (comma-separated).
            Prefix with - for descending.
            Example: -created_at,name
          schema:
            type: string
          example: "-created_at,name"

        - name: status
          in: query
          description: Filter by status
          schema:
            type: string
            enum: [active, inactive, pending]

        - name: created_after
          in: query
          description: Filter by creation date
          schema:
            type: string
            format: date-time

      responses:
        '200':
          description: Paginated users list
          content:
            application/json:
              schema:
                type: object
                properties:
```

```
                    data:
                      type: array
                      items:
                        $ref: '#/components/schemas/User'
                    pagination:
                      type: object
                      properties:
                        total:
                          type: integer
                        page:
                          type: integer
                        per_page:
                          type: integer
                        pages:
                          type: integer
                        links:
                          type: object
                          properties:
                            first:
                              type: string
                              format: uri
                            prev:
                              type: string
                              format: uri
                            self:
                              type: string
                              format: uri
                            next:
                              type: string
                              format: uri
                            last:
                              type: string
                              format: uri
```

9.11 Esempi Completi

9.11.1 Client JavaScript

Listing 9.26: Fetch Paginato in JavaScript

```
class PaginatedAPI {
  constructor(baseUrl, token) {
    this.baseUrl = baseUrl;
    this.token = token;
  }

  async fetchPage(endpoint, params = {}) {
    const queryString = new URLSearchParams(params).toString();
    const url = `${this.baseUrl}${endpoint}?${queryString}`;

    const response = await fetch(url, {
      headers: {
        'Authorization': `Bearer ${this.token}`,
        'Accept': 'application/json'
      }
    });

```

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

    return response.json();
  }

  async *fetchAllPages(endpoint, params = {}) {
    let page = 1;
    let hasMore = true;

    while (hasMore) {
      const data = await this.fetchPage(endpoint, {
        ...params,
        page,
        per_page: 100
      });

      yield data.data;

      hasMore = data.pagination.links.next !== null;
      page++;
    }
  }

  async fetchAll(endpoint, params = {}) {
    const allData = [];

    for await (const pageData of this.fetchAllPages(endpoint, params)) {
      allData.push(...pageData);
    }

    return allData;
  }
}

// Utilizzo
const api = new PaginatedAPI('https://api.example.com', 'token123');

// Fetch singola pagina
const page1 = await api.fetchPage('/users', {page: 1, per_page: 20});

// Fetch tutte le pagine (generator)
for await (const users of api.fetchAllPages('/users', {status: 'active
    '})) {
  console.log(`Fetched ${users.length} users`);
  processUsers(users);
}

// Fetch tutti i dati
const allUsers = await api.fetchAll('/users');
```

9.11.2 Script Bash Completo

Listing 9.27: Paginazione in Bash

```
#!/bin/bash
```

```bash

API_URL="https://api.example.com"
TOKEN="your-api-token"

fetch_all_pages() {
  local endpoint=$1
  local page=1
  local total_fetched=0

  while true; do
    echo "Fetching page $page..."

    response=$(curl -s \
      -H "Authorization: Bearer $TOKEN" \
      "$API_URL$endpoint?page=$page&per_page=100")

    # Estrai dati
    data=$(echo "$response" | jq '.data')
    count=$(echo "$data" | jq 'length')

    if [ "$count" -eq 0 ]; then
      echo "No more data"
      break
    fi

    # Processa dati
    echo "$data" | jq -c '.[]' | while read item; do
      # Processa ogni item
      echo "$item"
    done

    total_fetched=$((total_fetched + count))
    echo "Total fetched: $total_fetched"

    # Check se ci sono altre pagine
    has_next=$(echo "$response" | jq -r '.pagination.links.next // empty
      ')

    if [ -z "$has_next" ]; then
      echo "Reached last page"
      break
    fi

    page=$((page + 1))
  done
}

# Utilizzo
fetch_all_pages "/users"
```

Strategia	Pro	Contro	Quando Usarla
Offset	Semplice, pagine numerate	Lento con offset alto	Dataset piccoli/medi
Cursor	Performance costante	Non salta pagine	Stream, feed, dataset grandi
Page	User-friendly	Come offset	UI con navigazione pagine
Keyset	Veloce, stabile	Complesso	Dataset grandi ordinati

Tabella 9.2: Confronto Strategie di Paginazione

9.12 Riepilogo

Checklist Paginazione/Filtraggio

- Implementa limiti massimi (max 100-1000 item per pagina)
- Fornisci link navigazionali (next, prev, first, last)
- Documenta parametri e formati chiaramente
- Valida e sanitizza tutti i parametri
- Usa indici database per campi filtrabili/ordinabili
- Implementa field selection per ridurre payload
- Considera cache per richieste comuni
- Fornisci metadati (total, pages, etc)

Capitolo 10

Gestione degli Errori

Una gestione degli errori chiara e consistente è fondamentale per l'esperienza degli sviluppatori che utilizzano la tua API. Questo capitolo esplora best practices, standard e implementazioni.

10.1 Principi Fondamentali

10.1.1 Caratteristiche di un Buon Error Response

Requisiti Error Response

- **Status HTTP corretto**: Usa il codice di stato appropriato
- **Messaggio chiaro**: Spiega cosa è andato storto
- **Codice errore**: Identificatore machine-readable
- **Dettagli**: Informazioni aggiuntive per il debugging
- **Azioni suggerite**: Come risolvere il problema
- **Consistenza**: Stessa struttura per tutti gli errori

10.1.2 Cosa NON Fare

Anti-pattern da Evitare

- Ritornare sempre 200 OK (anche per errori)
- Esporre stack trace in produzione
- Messaggi generici tipo "Error" o "Something went wrong"
- Includere informazioni sensibili (password, token)
- Cambiare struttura errori tra endpoint
- Usare HTML invece di JSON per errori

10.2 Struttura Error Response

10.2.1 Formato Base

Listing 10.1: Struttura Error Response Standard

```
{
  "error": {
    "code": "validation_error",
    "message": "Validation failed for one or more fields",
    "details": [
      {
        "field": "email",
        "message": "Invalid email format",
        "code": "invalid_format"
      },
      {
        "field": "age",
        "message": "Must be at least 18",
        "code": "min_value"
      }
    ],
    "request_id": "req_abc123xyz",
    "timestamp": "2023-11-13T12:34:56Z",
    "documentation_url": "https://docs.example.com/errors/
       validation_error"
  }
}
```

10.2.2 Campi Principali

Campo	Tipo	Descrizione
`code`	string	Codice errore machine-readable
`message`	string	Descrizione human-readable
`details`	array	Dettagli specifici (es. campi validazione)
`request_id`	string	ID per tracciamento e debugging
`timestamp`	string	Quando è avvenuto l'errore
`documentation_url`	string	Link alla documentazione

Tabella 10.1: Campi Error Response

10.3 RFC 7807 Problem Details

10.3.1 Introduzione

RFC 7807 definisce uno standard per error responses in API HTTP.

Listing 10.2: RFC 7807 Problem Details Format

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: it

{
  "type": "https://example.com/probs/insufficient-credit",
  "title": "Credito insufficiente",
  "status": 403,
  "detail": "Il tuo account corrente ha solo 30 crediti, ma questa
     operazione richiede 50 crediti.",
  "instance": "/account/12345/transactions/abc",
```

```
  "balance": 30,
  "required": 50,
  "accounts": [
    "/account/12345",
    "/account/67890"
  ]
}
```

10.3.2 Campi RFC 7807

Campo	Obbligatorio	Descrizione
`type`	No (default: about:blank)	URI che identifica il tipo di problema
`title`	No	Breve descrizione human-readable
`status`	No	HTTP status code (ridondante ma utile)
`detail`	No	Spiegazione specifica dell'occorrenza
`instance`	No	URI che identifica questa specifica occorrenza

Tabella 10.2: Campi Standard RFC 7807

10.3.3 Campi Extension

RFC 7807 permette campi personalizzati:

Listing 10.3: Problem Details con Extension Fields

```
{
  "type": "https://api.example.com/errors/rate-limit",
  "title": "Rate limit exceeded",
  "status": 429,
  "detail": "You have exceeded the rate limit of 100 requests per hour",
  "instance": "/api/users?page=5",

  "limit": 100,
  "remaining": 0,
  "reset_at": "2023-11-13T13:00:00Z",
  "retry_after": 1847
}
```

10.3.4 Implementazione RFC 7807

Listing 10.4: Problem Details Class

```
from flask import jsonify, make_response
from datetime import datetime

class ProblemDetails:
    """RFC 7807 Problem Details implementation"""

    def __init__(self, status, title=None, detail=None,
                 problem_type=None, instance=None, **extensions):
        self.status = status
        self.title = title
```

```
        self.detail = detail
        self.type = problem_type or "about:blank"
        self.instance = instance
        self.extensions = extensions

    def to_dict(self):
        """Converti in dizionario"""
        problem = {
            "type": self.type,
            "status": self.status
        }

        if self.title:
            problem["title"] = self.title
        if self.detail:
            problem["detail"] = self.detail
        if self.instance:
            problem["instance"] = self.instance

        # Aggiungi extension fields
        problem.update(self.extensions)

        return problem

    def to_response(self):
        """Crea Flask response"""
        response = make_response(
            jsonify(self.to_dict()),
            self.status
        )
        response.headers['Content-Type'] = 'application/problem+json'
        return response

# Utilizzo
@app.errorhandler(429)
def rate_limit_exceeded(error):
    problem = ProblemDetails(
        status=429,
        title="Rate limit exceeded",
        detail="You have exceeded your rate limit",
        problem_type="https://api.example.com/errors/rate-limit",
        instance=request.path,
        limit=100,
        remaining=0,
        reset_at=datetime.utcnow().isoformat()
    )
    return problem.to_response()
```

10.4 Codici di Errore

10.4.1 Struttura Codici Errore

Listing 10.5: Convenzione Codici Errore

```
# Formato: categoria_tipo
validation_error
authentication_failed
```

```
authorization_denied
resource_not_found
rate_limit_exceeded
server_error

# Con sotto-categorie
validation_email_invalid
validation_password_weak
validation_required_field
authentication_token_expired
authentication_invalid_credentials
```

10.4.2 Catalogo Errori

HTTP Status	Code	Descrizione
400	`bad_request`	Request malformata
400	`validation_error`	Errori di validazione
400	`invalid_json`	JSON non valido
401	`unauthorized`	Autenticazione mancante
401	`invalid_token`	Token non valido
401	`token_expired`	Token scaduto
403	`forbidden`	Accesso negato
403	`insufficient_permissions`	Permessi insufficienti
404	`not_found`	Risorsa non trovata
409	`conflict`	Conflitto (es. duplicato)
422	`unprocessable_entity`	Semantica non valida
429	`rate_limit_exceeded`	Troppi richieste
500	`internal_server_error`	Errore server generico
503	`service_unavailable`	Servizio non disponibile

Tabella 10.3: Codici Errore Comuni

10.5 Errori di Validazione

10.5.1 Formato Dettagliato

Listing 10.6: Validation Error Response

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": {
    "code": "validation_error",
    "message": "Request validation failed",
    "errors": [
      {
        "field": "email",
        "value": "not-an-email",
        "code": "invalid_format",
        "message": "Must be a valid email address"
      },
      {
        "field": "password",
        "code": "too_short",
```

```
        "message": "Password must be at least 8 characters",
        "constraint": {
          "min_length": 8,
          "actual_length": 5
        }
      },
      {
        "field": "age",
        "value": 15,
        "code": "out_of_range",
        "message": "Age must be between 18 and 120",
        "constraint": {
          "min": 18,
          "max": 120
        }
      },
      {
        "field": "terms_accepted",
        "code": "required",
        "message": "This field is required"
      }
    ]
  }
}
```

10.5.2 Implementazione Validation

Listing 10.7: Validation Error Handler

```python
from marshmallow import Schema, fields, validate, ValidationError

class UserSchema(Schema):
    email = fields.Email(required=True)
    password = fields.Str(
        required=True,
        validate=validate.Length(min=8, max=128)
    )
    age = fields.Int(
        validate=validate.Range(min=18, max=120)
    )
    terms_accepted = fields.Bool(required=True)

def validate_request(schema_class, data):
    """Valida dati con Marshmallow schema"""
    schema = schema_class()

    try:
        validated = schema.load(data)
        return validated, None

    except ValidationError as err:
        # Converti errori Marshmallow in formato API
        errors = []

        for field, messages in err.messages.items():
            for message in messages if isinstance(messages, list) else [
                messages]:
```

```
                error_detail = {
                    "field": field,
                    "message": message,
                    "code": determine_error_code(message)
                }

                # Aggiungi valore se presente
                if field in data:
                    error_detail["value"] = data[field]

                errors.append(error_detail)

        return None, {
            "error": {
                "code": "validation_error",
                "message": "Request validation failed",
                "errors": errors
            }
        }

@app.route('/api/users', methods=['POST'])
def create_user():
    data = request.get_json()

    # Valida
    validated_data, error = validate_request(UserSchema, data)

    if error:
        return jsonify(error), 400

    # Processa dati validati
    user = User(**validated_data)
    db.session.add(user)
    db.session.commit()

    return jsonify(user.to_dict()), 201
```

10.6 Errori di Autenticazione e Autorizzazione

10.6.1 401 Unauthorized

Listing 10.8: 401 Unauthorized Response

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example.com", error="invalid_token",
  error_description="The access token expired"
Content-Type: application/json

{
  "error": {
    "code": "token_expired",
    "message": "Your authentication token has expired",
    "expired_at": "2023-11-13T12:00:00Z",
    "actions": [
      "Refresh your token using the refresh endpoint",
      "Re-authenticate to obtain a new token"
    ],
```

```
    "links": {
      "refresh": "https://api.example.com/auth/refresh",
      "login": "https://api.example.com/auth/login"
    }
  }
}
```

10.6.2 403 Forbidden

Listing 10.9: 403 Forbidden Response

```
HTTP/1.1 403 Forbidden
Content-Type: application/json

{
  "error": {
    "code": "insufficient_permissions",
    "message": "You don't have permission to access this resource",
    "required_permissions": ["admin:write"],
    "current_permissions": ["user:read", "user:write"],
    "resource": "/api/admin/users",
    "actions": [
      "Contact your administrator to request elevated permissions",
      "Use an account with appropriate permissions"
    ]
  }
}
\end{lstlisting>

\subsection{Implementazione}

\begin{lstlisting}[language=Python, caption=Auth Error Handlers]
from functools import wraps
from flask import request, jsonify
import jwt

def require_auth(required_permissions=None):
    """Decorator per richiedere autenticazione"""
    def decorator(f):
        @wraps(f)
        def wrapped(*args, **kwargs):
            # Estrai token
            auth_header = request.headers.get('Authorization')

            if not auth_header or not auth_header.startswith('Bearer '):
                return jsonify({
                    "error": {
                        "code": "unauthorized",
                        "message": "Missing or invalid Authorization
                            header",
                        "expected_format": "Authorization: Bearer <token
                            >"
                    }
                }), 401

            token = auth_header[7:]

```

```
            # Verifica token
            try:
                payload = jwt.decode(
                    token,
                    app.config['SECRET_KEY'],
                    algorithms=['HS256']
                )
            except jwt.ExpiredSignatureError:
                return jsonify({
                    "error": {
                        "code": "token_expired",
                        "message": "Authentication token has expired",
                        "links": {
                            "refresh": "/auth/refresh",
                            "login": "/auth/login"
                        }
                    }
                }), 401
            except jwt.InvalidTokenError:
                return jsonify({
                    "error": {
                        "code": "invalid_token",
                        "message": "Authentication token is invalid"
                    }
                }), 401

            # Verifica permessi
            if required_permissions:
                user_permissions = payload.get('permissions', [])

                missing = set(required_permissions) - set(
                    user_permissions)
                if missing:
                    return jsonify({
                        "error": {
                            "code": "insufficient_permissions",
                            "message": "Insufficient permissions",
                            "required": list(required_permissions),
                            "missing": list(missing),
                            "current": user_permissions
                        }
                    }), 403

            # Aggiungi user al context
            request.current_user = payload
            return f(*args, **kwargs)

        return wrapped
    return decorator

# Utilizzo
@app.route('/api/admin/users', methods=['DELETE'])
@require_auth(required_permissions=['admin:delete'])
def delete_user():
    # Solo accessibile con permesso admin:delete
    pass
```

10.7 Errori 404 Not Found

10.7.1 Resource Not Found

Listing 10.10: 404 Not Found Response

```
HTTP/1.1 404 Not Found
Content-Type: application/json

{
  "error": {
    "code": "resource_not_found",
    "message": "User with ID 12345 not found",
    "resource_type": "User",
    "resource_id": "12345",
    "suggestions": [
      "Verify the user ID is correct",
      "The user may have been deleted",
      "Check if you have access to this user"
    ],
    "links": {
      "list_users": "https://api.example.com/users",
      "search": "https://api.example.com/users/search"
    }
  }
}
```

10.7.2 Endpoint Not Found

Listing 10.11: 404 Endpoint Not Found

```
HTTP/1.1 404 Not Found
Content-Type: application/json

{
  "error": {
    "code": "endpoint_not_found",
    "message": "The requested endpoint does not exist",
    "path": "/api/v1/userz",
    "method": "GET",
    "suggestions": [
      "Check the URL spelling",
      "Verify you are using the correct API version",
      "Consult the API documentation"
    ],
    "similar_endpoints": [
      "/api/v1/users",
      "/api/v1/user/{id}"
    ],
    "documentation_url": "https://docs.example.com/api/v1"
  }
}
```

10.8 Errori 409 Conflict

10.8.1 Duplicate Resource

Listing 10.12: 409 Conflict - Duplicato

```
HTTP/1.1 409 Conflict
Content-Type: application/json

{
  "error": {
    "code": "duplicate_resource",
    "message": "A user with this email already exists",
    "conflicting_field": "email",
    "conflicting_value": "user@example.com",
    "existing_resource": {
      "id": 789,
      "url": "https://api.example.com/users/789"
    },
    "actions": [
      "Use a different email address",
      "Update the existing user instead of creating a new one",
      "Use the login endpoint if you already have an account"
    ]
  }
}
```

10.8.2 Concurrent Modification

Listing 10.13: 409 Conflict - Versione

```
HTTP/1.1 409 Conflict
Content-Type: application/json

{
  "error": {
    "code": "concurrent_modification",
    "message": "The resource has been modified by another request",
    "resource_type": "Document",
    "resource_id": "doc_123",
    "expected_version": 5,
    "current_version": 7,
    "last_modified_by": "user@example.com",
    "last_modified_at": "2023-11-13T12:30:00Z",
    "actions": [
      "Fetch the latest version of the resource",
      "Merge your changes with the current version",
      "Use optimistic locking with the correct version"
    ],
    "links": {
      "fetch_latest": "https://api.example.com/documents/doc_123"
    }
  }
}
```

10.9 Errori 429 Rate Limit

Listing 10.14: 429 Too Many Requests

```
HTTP/1.1 429 Too Many Requests
```

```
Retry-After: 3600
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1699876543
Content-Type: application/json

{
  "error": {
    "code": "rate_limit_exceeded",
    "message": "API rate limit exceeded",
    "limit": 100,
    "window": "1 hour",
    "reset_at": "2023-11-13T13:00:00Z",
    "retry_after_seconds": 3600,
    "current_usage": {
      "hourly": 100,
      "daily": 850
    },
    "limits": {
      "hourly": 100,
      "daily": 1000
    },
    "suggestions": [
      "Wait 1 hour before making more requests",
      "Implement exponential backoff in your client",
      "Consider upgrading to a higher tier plan"
    ],
    "upgrade_url": "https://example.com/pricing"
  }
}
```

10.10 Errori 500 Server

10.10.1 Internal Server Error

Listing 10.15: 500 Internal Server Error (Production)

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json

{
  "error": {
    "code": "internal_server_error",
    "message": "An unexpected error occurred",
    "request_id": "req_abc123xyz789",
    "timestamp": "2023-11-13T12:34:56.789Z",
    "actions": [
      "Try your request again",
      "If the problem persists, contact support with the request ID",
      "Check the API status page for known issues"
    ],
    "links": {
      "status": "https://status.example.com",
      "support": "https://support.example.com",
      "documentation": "https://docs.example.com"
    }
  }
```

```
}
```

Importante: Sicurezza

MAI esporre stack trace, query SQL, o dettagli interni in produzione! Logga i dettagli server-side e fornisci solo un request_id al client.

10.10.2 Error Logging

Listing 10.16: Logging Errori Server

```python
import logging
import traceback
import uuid
from datetime import datetime

logger = logging.getLogger(__name__)

@app.errorhandler(Exception)
def handle_unexpected_error(error):
    """Gestisce errori non previsti"""

    # Genera request ID univoco
    request_id = f"req_{uuid.uuid4().hex}"

    # Log completo server-side
    logger.error(
        f"Unexpected error [{request_id}]: {str(error)}",
        extra={
            'request_id': request_id,
            'path': request.path,
            'method': request.method,
            'user': getattr(request, 'current_user', None),
            'timestamp': datetime.utcnow().isoformat(),
            'stacktrace': traceback.format_exc()
        }
    )

    # Response minimale al client
    return jsonify({
        "error": {
            "code": "internal_server_error",
            "message": "An unexpected error occurred",
            "request_id": request_id,
            "timestamp": datetime.utcnow().isoformat()
        }
    }), 500
```

10.11 Errori 503 Service Unavailable

Listing 10.17: 503 Service Unavailable

```
HTTP/1.1 503 Service Unavailable
Retry-After: 300
Content-Type: application/json
```

```

{
  "error": {
    "code": "service_unavailable",
    "message": "The API is temporarily unavailable",
    "reason": "scheduled_maintenance",
    "retry_after_seconds": 300,
    "estimated_restore": "2023-11-13T14:00:00Z",
    "status_page": "https://status.example.com",
    "details": {
      "maintenance_window": {
        "start": "2023-11-13T13:00:00Z",
        "end": "2023-11-13T14:00:00Z",
        "duration_minutes": 60
      },
      "affected_services": ["API", "OAuth"]
    }
  }
}
```

10.12 Global Error Handler

10.12.1 Middleware Centralizzato

Listing 10.18: Centralized Error Handler

```python
from flask import Flask, jsonify, request
from werkzeug.exceptions import HTTPException
import logging

app = Flask(__name__)
logger = logging.getLogger(__name__)

class APIError(Exception):
    """Base class per errori API"""

    def __init__(self, message, status_code=400,
                 code=None, details=None, **kwargs):
        super().__init__(message)
        self.message = message
        self.status_code = status_code
        self.code = code or self.__class__.__name__
        self.details = details or {}
        self.extensions = kwargs

    def to_dict(self):
        error = {
            "code": self.code,
            "message": self.message
        }

        if self.details:
            error["details"] = self.details

        error.update(self.extensions)

        return {"error": error}
```

```python

# Errori specifici
class ValidationError(APIError):
    def __init__(self, message, errors, **kwargs):
        super().__init__(
            message,
            status_code=400,
            code="validation_error",
            errors=errors,
            **kwargs
        )

class NotFoundError(APIError):
    def __init__(self, resource_type, resource_id, **kwargs):
        super().__init__(
            f"{resource_type} with ID {resource_id} not found",
            status_code=404,
            code="resource_not_found",
            resource_type=resource_type,
            resource_id=resource_id,
            **kwargs
        )

class UnauthorizedError(APIError):
    def __init__(self, message="Unauthorized", **kwargs):
        super().__init__(
            message,
            status_code=401,
            code="unauthorized",
            **kwargs
        )

# Global error handlers
@app.errorhandler(APIError)
def handle_api_error(error):
    """Gestisce APIError custom"""
    response = jsonify(error.to_dict())
    response.status_code = error.status_code
    return response

@app.errorhandler(HTTPException)
def handle_http_exception(error):
    """Gestisce eccezioni HTTP Werkzeug"""
    return jsonify({
        "error": {
            "code": error.name.lower().replace(" ", "_"),
            "message": error.description,
            "status": error.code
        }
    }), error.code

@app.errorhandler(404)
def handle_not_found(error):
    """404 personalizzato"""
    return jsonify({
        "error": {
            "code": "endpoint_not_found",
            "message": "The requested endpoint does not exist",
```

```
            "path": request.path,
            "method": request.method
        }
    }), 404

@app.errorhandler(Exception)
def handle_unexpected_error(error):
    """Catch-all per errori non previsti"""
    logger.exception("Unexpected error")

    return jsonify({
        "error": {
            "code": "internal_server_error",
            "message": "An unexpected error occurred"
        }
    }), 500

# Utilizzo negli endpoint
@app.route('/api/users/<int:user_id>')
def get_user(user_id):
    user = User.query.get(user_id)

    if not user:
        raise NotFoundError("User", user_id)

    return jsonify(user.to_dict())

@app.route('/api/users', methods=['POST'])
def create_user():
    data = request.get_json()

    # Validazione
    errors = validate_user_data(data)
    if errors:
        raise ValidationError(
            "Validation failed",
            errors=errors
        )

    # ...
```

10.13 Client Error Handling

10.13.1 JavaScript Example

Listing 10.19: Client Error Handling

```
class APIClient {
  async request(endpoint, options = {}) {
    try {
      const response = await fetch(
        `${this.baseUrl}${endpoint}`,
        {
          ...options,
          headers: {
            'Authorization': `Bearer ${this.token}`,
            'Content-Type': 'application/json',
```

```
            ...options.headers
          }
        }
      );

      // Parse JSON (anche per errori)
      const data = await response.json();

      if (!response.ok) {
        throw new APIError(data.error, response.status);
      }

      return data;

    } catch (error) {
      if (error instanceof APIError) {
        throw error;
      }

      // Network error o altro
      throw new APIError({
        code: 'network_error',
        message: 'Failed to connect to API'
      }, 0);
    }
  }

  async retryWithBackoff(fn, maxRetries = 3) {
    for (let attempt = 0; attempt <= maxRetries; attempt++) {
      try {
        return await fn();
      } catch (error) {
        if (error instanceof APIError) {
          // Retry su errori temporanei
          if ([429, 503].includes(error.status)) {
            if (attempt < maxRetries) {
              const delay = error.retryAfter ||
                Math.min(1000 * Math.pow(2, attempt), 10000);

              console.log(`Retrying in ${delay}ms...`);
              await new Promise(resolve => setTimeout(resolve, delay));
              continue;
            }
          }

          // Non fare retry su errori client
          if (error.status >= 400 && error.status < 500) {
            throw error;
          }
        }

        // Ultimo tentativo
        if (attempt === maxRetries) {
          throw error;
        }

        // Exponential backoff
        const delay = Math.min(1000 * Math.pow(2, attempt), 10000);
```

```
        await new Promise(resolve => setTimeout(resolve, delay));
      }
    }
  }
}

class APIError extends Error {
  constructor(errorData, status) {
    super(errorData.message);
    this.name = 'APIError';
    this.code = errorData.code;
    this.status = status;
    this.details = errorData.details;
    this.retryAfter = errorData.retry_after_seconds;
  }

  isValidationError() {
    return this.code === 'validation_error';
  }

  isAuthError() {
    return [401, 403].includes(this.status);
  }

  isRetryable() {
    return [429, 503].includes(this.status) || this.status >= 500;
  }
}

// Utilizzo
const api = new APIClient('https://api.example.com', token);

try {
  const user = await api.retryWithBackoff(
    () => api.request('/users/123')
  );

  console.log('User:', user);

} catch (error) {
  if (error instanceof APIError) {
    if (error.isValidationError()) {
      // Mostra errori validazione
      error.details.errors.forEach(err => {
        console.error(`${err.field}: ${err.message}`);
      });
    } else if (error.isAuthError()) {
      // Redirect a login
      window.location.href = '/login';
    } else {
      // Errore generico
      console.error(`API Error: ${error.message}`);
    }
  }
}
```

10.14 Testing Errors

10.14.1 Unit Tests

Listing 10.20: Testing Error Responses

```python
import unittest
from app import app, db
from app.models import User

class TestErrorHandling(unittest.TestCase):
    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True

    def test_404_not_found(self):
        """Test 404 per risorsa inesistente"""
        response = self.app.get('/api/users/99999')

        self.assertEqual(response.status_code, 404)

        data = response.get_json()
        self.assertIn('error', data)
        self.assertEqual(data['error']['code'], 'resource_not_found')

    def test_validation_error(self):
        """Test errore validazione"""
        response = self.app.post(
            '/api/users',
            json={'email': 'invalid-email'}
        )

        self.assertEqual(response.status_code, 400)

        data = response.get_json()
        self.assertEqual(data['error']['code'], 'validation_error')
        self.assertIn('errors', data['error'])

    def test_unauthorized(self):
        """Test accesso senza token"""
        response = self.app.get('/api/protected')

        self.assertEqual(response.status_code, 401)

        data = response.get_json()
        self.assertEqual(data['error']['code'], 'unauthorized')

    def test_rate_limit(self):
        """Test rate limiting"""
        # Supera il limite
        for i in range(105):
            response = self.app.get('/api/data')

        self.assertEqual(response.status_code, 429)

        data = response.get_json()
        self.assertEqual(data['error']['code'], 'rate_limit_exceeded')
        self.assertIn('Retry-After', response.headers)

```

```
if __name__ == '__main__':
    unittest.main()
```

10.15 OpenAPI Specification

Listing 10.21: Error Schemas in OpenAPI

```
openapi: 3.0.0
components:
  schemas:
    Error:
      type: object
      required:
        - error
      properties:
        error:
          type: object
          required:
            - code
            - message
          properties:
            code:
              type: string
              example: validation_error
            message:
              type: string
              example: Validation failed
            details:
              type: object
            request_id:
              type: string
              example: req_abc123
            timestamp:
              type: string
              format: date-time

    ValidationError:
      allOf:
        - $ref: '#/components/schemas/Error'
        - type: object
          properties:
            error:
              type: object
              properties:
                errors:
                  type: array
                  items:
                    type: object
                    properties:
                      field:
                        type: string
                      code:
                        type: string
                      message:
                        type: string

paths:
```

```
  /users/{id}:
    get:
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: User found
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'

        '401':
          description: Unauthorized
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              example:
                error:
                  code: unauthorized
                  message: Missing or invalid authentication

        '403':
          description: Forbidden
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'

        '404':
          description: User not found
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              example:
                error:
                  code: resource_not_found
                  message: User with ID 123 not found
                  resource_type: User
                  resource_id: "123"

        '500':
          description: Internal server error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
```

10.16 Best Practices

Checklist Error Handling

- Usa HTTP status code appropriati
- Fornisci messaggi chiari e actionable
- Includi codici errore machine-readable
- Mantieni struttura consistente
- Aggiungi request_id per debugging
- Non esporre dettagli interni in production
- Logga errori server-side con context
- Documenta tutti gli errori possibili
- Testa scenari di errore
- Fornisci suggerimenti per risolvere
- Considera localizzazione messaggi
- Implementa retry logic lato client

10.17 Riepilogo

Status	Quando Usare	Esempio
400	Errore client generico	Request malformata, JSON invalido
401	Autenticazione fallita	Token mancante/scaduto
403	Autorizzazione negata	Permessi insufficienti
404	Risorsa non trovata	User ID inesistente
409	Conflitto	Email già esistente, versione obsoleta
422	Validazione semantica	Dati sintatticamente validi ma semanticamente no
429	Rate limit	Troppe richieste
500	Errore server	Eccezione non gestita
503	Servizio non disponibile	Manutenzione, sovraccarico

Tabella 10.4: Guida Status Codes per Errori

Capitolo 11

Documentazione e OpenAPI

La documentazione è fondamentale per l'adozione e l'utilizzo corretto di un'API. OpenAPI Specification (precedentemente Swagger) è lo standard de facto per documentare REST API.

11.1 Introduzione a OpenAPI

11.1.1 Cos'è OpenAPI

OpenAPI Specification (OAS) è uno standard per descrivere REST API in formato machine-readable (YAML o JSON).

Benefici di OpenAPI

- **Documentazione interattiva**: Genera Swagger UI automaticamente
- **Code generation**: Genera client/server in vari linguaggi
- **Validazione**: Valida request/response automaticamente
- **Testing**: Genera test cases automaticamente
- **Mocking**: Crea mock server per sviluppo
- **Contract-first**: Progetta API prima di implementare

11.1.2 Versioni OpenAPI

- **OpenAPI 2.0** (Swagger 2.0): Versione legacy
- **OpenAPI 3.0**: Versione attuale (focus di questo capitolo)
- **OpenAPI 3.1**: Versione più recente, compatibile con JSON Schema

11.2 Struttura Base OpenAPI 3.0

11.2.1 Documento Minimale

Listing 11.1: OpenAPI 3.0 - Struttura Minimale

```
openapi: 3.0.3
info:
  title: My API
  version: 1.0.0
```

```
  description: API description

servers:
  - url: https://api.example.com/v1
    description: Production server

paths:
  /users:
    get:
      summary: List users
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
```

11.2.2 Sezioni Principali

1. **openapi**: Versione della spec
2. **info**: Metadati dell'API
3. **servers**: URL dei server
4. **paths**: Endpoint e operazioni
5. **components**: Componenti riutilizzabili (schemas, responses, etc)
6. **security**: Schemi di sicurezza
7. **tags**: Raggruppamento endpoint
8. **externalDocs**: Documentazione esterna

11.3 Info Object

Listing 11.2: Info Object Completo

```
info:
  title: E-commerce API
  version: 2.1.0
  description: |
    # E-commerce REST API

    API completa per gestione e-commerce con:
    - Gestione prodotti e categorie
    - Carrello e ordini
    - Autenticazione utenti
    - Pagamenti

    ## Autenticazione

    Usa OAuth 2.0 per autenticazione.
```

```
    Vedi la sezione Security per dettagli.

    ## Rate Limiting

    - Free tier: 100 req/hour
    - Pro tier: 1000 req/hour

  termsOfService: https://example.com/terms
  contact:
    name: API Support
    email: api@example.com
    url: https://example.com/support
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
```

11.4 Servers

Listing 11.3: Configurazione Server

```
servers:
  - url: https://api.example.com/v1
    description: Production server

  - url: https://staging-api.example.com/v1
    description: Staging server

  - url: http://localhost:3000/v1
    description: Local development server

  # Server con variabili
  - url: https://{environment}.example.com/v{version}
    description: Configurable server
    variables:
      environment:
        default: api
        enum:
          - api
          - staging
          - sandbox
        description: Server environment
      version:
        default: '1'
        enum:
          - '1'
          - '2'
        description: API version
```

11.5 Paths e Operations

11.5.1 Definizione Path

Listing 11.4: Path con Operazioni Complete

```
paths:
```

```yaml
  /users:
    get:
      summary: List all users
      description: |
        Returns a paginated list of users.
        Supports filtering, sorting, and field selection.
      operationId: listUsers
      tags:
        - Users
      parameters:
        - $ref: '#/components/parameters/PageParam'
        - $ref: '#/components/parameters/PerPageParam'
        - name: status
          in: query
          description: Filter by user status
          required: false
          schema:
            type: string
            enum: [active, inactive, pending]
        - name: sort
          in: query
          description: Sort fields (comma-separated, prefix with - for
             desc)
          schema:
            type: string
            example: "-created_at,name"
      responses:
        '200':
          description: List of users
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/UserList'
        '400':
          $ref: '#/components/responses/BadRequest'
        '401':
          $ref: '#/components/responses/Unauthorized'
      security:
        - bearerAuth: []

    post:
      summary: Create a new user
      description: Creates a new user with the provided data
      operationId: createUser
      tags:
        - Users
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/UserCreate'
            examples:
              basic:
                summary: Basic user creation
                value:
                  email: user@example.com
                  name: Mario Rossi
```

```yaml
                password: SecurePass123!
            admin:
              summary: Admin user creation
              value:
                email: admin@example.com
                name: Admin User
                password: SecurePass123!
                role: admin
      responses:
        '201':
          description: User created successfully
          headers:
            Location:
              schema:
                type: string
                format: uri
              description: URL of the created user
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
        '400':
          $ref: '#/components/responses/ValidationError'
        '409':
          description: User already exists
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              example:
                error:
                  code: duplicate_resource
                  message: A user with this email already exists

  /users/{userId}:
    parameters:
      - name: userId
        in: path
        required: true
        description: Unique user identifier
        schema:
          type: integer
          format: int64
          minimum: 1
        example: 12345

    get:
      summary: Get user by ID
      description: Returns detailed information about a specific user
      operationId: getUser
      tags:
        - Users
      responses:
        '200':
          description: User details
          content:
            application/json:
              schema:
```

```
117                 $ref: '#/components/schemas/User'
118         '404':
119           $ref: '#/components/responses/NotFound'
120       security:
121         - bearerAuth: []
122
123     put:
124       summary: Update user
125       description: Updates an existing user (full replacement)
126       operationId: updateUser
127       tags:
128         - Users
129       requestBody:
130         required: true
131         content:
132           application/json:
133             schema:
134               $ref: '#/components/schemas/UserUpdate'
135       responses:
136         '200':
137           description: User updated successfully
138           content:
139             application/json:
140               schema:
141                 $ref: '#/components/schemas/User'
142         '404':
143           $ref: '#/components/responses/NotFound'
144       security:
145         - bearerAuth: []
146
147     patch:
148       summary: Partially update user
149       description: Updates specific fields of a user
150       operationId: patchUser
151       tags:
152         - Users
153       requestBody:
154         required: true
155         content:
156           application/json:
157             schema:
158               $ref: '#/components/schemas/UserPatch'
159       responses:
160         '200':
161           description: User updated
162           content:
163             application/json:
164               schema:
165                 $ref: '#/components/schemas/User'
166         '404':
167           $ref: '#/components/responses/NotFound'
168       security:
169         - bearerAuth: []
170
171     delete:
172       summary: Delete user
173       description: Permanently deletes a user
174       operationId: deleteUser
```

```
      tags:
        - Users
      responses:
        '204':
          description: User deleted successfully
        '404':
          $ref: '#/components/responses/NotFound'
      security:
        - bearerAuth: [admin:delete]
```

11.6 Components - Schemas

11.6.1 Schema Object

Listing 11.5: Definizione Schemas Riutilizzabili

```
components:
  schemas:
    User:
      type: object
      required:
        - id
        - email
        - name
      properties:
        id:
          type: integer
          format: int64
          readOnly: true
          description: Unique user identifier
          example: 12345
        email:
          type: string
          format: email
          description: User email address
          example: user@example.com
        name:
          type: string
          minLength: 1
          maxLength: 100
          description: Full name of the user
          example: Mario Rossi
        avatar_url:
          type: string
          format: uri
          nullable: true
          description: URL to user avatar image
          example: https://cdn.example.com/avatars/12345.jpg
        role:
          type: string
          enum: [user, admin, moderator]
          default: user
          description: User role
        status:
          type: string
          enum: [active, inactive, pending, suspended]
          default: pending
```

```yaml
        created_at:
          type: string
          format: date-time
          readOnly: true
          description: Timestamp of user creation
          example: "2023-11-13T12:34:56Z"
        updated_at:
          type: string
          format: date-time
          readOnly: true
          description: Timestamp of last update
        metadata:
          type: object
          additionalProperties: true
          description: Custom metadata
          example:
            source: web_signup
            referral: campaign_123

    UserCreate:
      type: object
      required:
        - email
        - name
        - password
      properties:
        email:
          type: string
          format: email
        name:
          type: string
          minLength: 1
          maxLength: 100
        password:
          type: string
          format: password
          minLength: 8
          maxLength: 128
          writeOnly: true
          description: User password (min 8 characters)
        role:
          type: string
          enum: [user, admin]
          default: user

    UserUpdate:
      type: object
      required:
        - email
        - name
      properties:
        email:
          type: string
          format: email
        name:
          type: string
          minLength: 1
          maxLength: 100
```

```yaml
        avatar_url:
          type: string
          format: uri
          nullable: true

    UserPatch:
      type: object
      minProperties: 1
      properties:
        name:
          type: string
          minLength: 1
          maxLength: 100
        avatar_url:
          type: string
          format: uri
          nullable: true
        status:
          type: string
          enum: [active, inactive, suspended]

    UserList:
      type: object
      required:
        - data
        - pagination
      properties:
        data:
          type: array
          items:
            $ref: '#/components/schemas/User'
        pagination:
          $ref: '#/components/schemas/Pagination'

    Pagination:
      type: object
      properties:
        total:
          type: integer
          description: Total number of items
          example: 1543
        count:
          type: integer
          description: Number of items in current page
          example: 10
        per_page:
          type: integer
          description: Items per page
          example: 10
        current_page:
          type: integer
          description: Current page number
          example: 2
        total_pages:
          type: integer
          description: Total number of pages
          example: 155
        links:
```

```yaml
        type: object
        properties:
          first:
            type: string
            format: uri
          prev:
            type: string
            format: uri
            nullable: true
          self:
            type: string
            format: uri
          next:
            type: string
            format: uri
            nullable: true
          last:
            type: string
            format: uri

    Error:
      type: object
      required:
        - error
      properties:
        error:
          type: object
          required:
            - code
            - message
          properties:
            code:
              type: string
              description: Machine-readable error code
              example: validation_error
            message:
              type: string
              description: Human-readable error message
              example: Validation failed
            details:
              type: object
              description: Additional error details
            request_id:
              type: string
              description: Request ID for debugging
              example: req_abc123xyz
            timestamp:
              type: string
              format: date-time

    ValidationError:
      allOf:
        - $ref: '#/components/schemas/Error'
        - type: object
          properties:
            error:
              type: object
              properties:
```

```
                errors:
                  type: array
                  items:
                    type: object
                    properties:
                      field:
                        type: string
                      code:
                        type: string
                      message:
                        type: string
```

11.6.2 Composizione con allOf, oneOf, anyOf

Listing 11.6: Schema Composition

```
components:
  schemas:
    # Eredità con allOf
    AdminUser:
      allOf:
        - $ref: '#/components/schemas/User'
        - type: object
          properties:
            admin_level:
              type: integer
              minimum: 1
              maximum: 10
            permissions:
              type: array
              items:
                type: string

    # oneOf - esattamente uno
    Pet:
      oneOf:
        - $ref: '#/components/schemas/Cat'
        - $ref: '#/components/schemas/Dog'
      discriminator:
        propertyName: pet_type
        mapping:
          cat: '#/components/schemas/Cat'
          dog: '#/components/schemas/Dog'

    Cat:
      type: object
      required: [pet_type, meow_volume]
      properties:
        pet_type:
          type: string
          enum: [cat]
        meow_volume:
          type: integer

    Dog:
      type: object
      required: [pet_type, bark_volume]
```

```
      properties:
        pet_type:
          type: string
          enum: [dog]
        bark_volume:
          type: integer

    # anyOf - uno o più
    SearchResult:
      anyOf:
        - $ref: '#/components/schemas/User'
        - $ref: '#/components/schemas/Product'
        - $ref: '#/components/schemas/Order'
```

11.7 Components - Parameters

Listing 11.7: Parametri Riutilizzabili

```
components:
  parameters:
    PageParam:
      name: page
      in: query
      description: Page number (1-indexed)
      required: false
      schema:
        type: integer
        minimum: 1
        default: 1

    PerPageParam:
      name: per_page
      in: query
      description: Items per page
      required: false
      schema:
        type: integer
        minimum: 1
        maximum: 100
        default: 10

    SortParam:
      name: sort
      in: query
      description: |
        Sort fields (comma-separated).
        Prefix with - for descending order.
      schema:
        type: string
      example: "-created_at,name"

    FieldsParam:
      name: fields
      in: query
      description: Comma-separated list of fields to include
      schema:
        type: string
```

```
      example: "id,name,email"

    IfMatchHeader:
      name: If-Match
      in: header
      description: ETag for optimistic locking
      required: false
      schema:
        type: string
      example: '"33a64df551425fcc55e4d42a148795d9f25f89d4"'

    AcceptLanguage:
      name: Accept-Language
      in: header
      description: Preferred language for response
      schema:
        type: string
        default: en
      example: "it-IT,it;q=0.9,en;q=0.8"
```

11.8 Components - Responses

Listing 11.8: Response Riutilizzabili

```
components:
  responses:
    BadRequest:
      description: Bad request
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          example:
            error:
              code: bad_request
              message: The request could not be understood

    Unauthorized:
      description: Unauthorized - authentication required
      headers:
        WWW-Authenticate:
          schema:
            type: string
          example: Bearer realm="example.com"
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          example:
            error:
              code: unauthorized
              message: Authentication required

    Forbidden:
      description: Forbidden - insufficient permissions
      content:
        application/json:
```

```
          schema:
            $ref: '#/components/schemas/Error'
          example:
            error:
              code: forbidden
              message: You don't have permission to access this resource

    NotFound:
      description: Resource not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          example:
            error:
              code: resource_not_found
              message: The requested resource was not found

    ValidationError:
      description: Validation error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ValidationError'

    RateLimitExceeded:
      description: Rate limit exceeded
      headers:
        X-RateLimit-Limit:
          schema:
            type: integer
        X-RateLimit-Remaining:
          schema:
            type: integer
        X-RateLimit-Reset:
          schema:
            type: integer
        Retry-After:
          schema:
            type: integer
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
```

11.9 Security Schemes

Listing 11.9: Security Schemes Completi

```
components:
  securitySchemes:
    # HTTP Basic Auth
    basicAuth:
      type: http
      scheme: basic
      description: HTTP Basic Authentication

```

```yaml
    # Bearer Token
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
      description: JWT Bearer token authentication

    # API Key
    apiKey:
      type: apiKey
      in: header
      name: X-API-Key
      description: API key for application identification

    # OAuth 2.0
    oauth2:
      type: oauth2
      description: OAuth 2.0 authentication
      flows:
        authorizationCode:
          authorizationUrl: https://auth.example.com/oauth/authorize
          tokenUrl: https://auth.example.com/oauth/token
          refreshUrl: https://auth.example.com/oauth/refresh
          scopes:
            read:users: Read user information
            write:users: Create and modify users
            delete:users: Delete users
            read:orders: Read orders
            write:orders: Create and modify orders
            admin:all: Full administrative access

        clientCredentials:
          tokenUrl: https://auth.example.com/oauth/token
          scopes:
            api:read: Read API data
            api:write: Write API data

        password:
          tokenUrl: https://auth.example.com/oauth/token
          refreshUrl: https://auth.example.com/oauth/refresh
          scopes:
            read:users: Read user information
            write:users: Modify user information

# Security globale (applicata a tutti gli endpoint)
security:
  - bearerAuth: []

# Override per endpoint specifici
paths:
  /public/status:
    get:
      summary: Public API status
      security: []  # Nessuna autenticazione richiesta

  /admin/users:
    delete:
      summary: Delete user (admin only)
```

```
      security:
        - oauth2: [admin:all]
```

11.10 Request Body

Listing 11.10: Request Body con Content-Type Multipli

```
paths:
  /users/{userId}/avatar:
    post:
      summary: Upload user avatar
      operationId: uploadAvatar
      parameters:
        - name: userId
          in: path
          required: true
          schema:
            type: integer
      requestBody:
        required: true
        content:
          # Immagine
          image/jpeg:
            schema:
              type: string
              format: binary
          image/png:
            schema:
              type: string
              format: binary

          # Multipart form
          multipart/form-data:
            schema:
              type: object
              required:
                - file
              properties:
                file:
                  type: string
                  format: binary
                  description: Avatar image file
                description:
                  type: string
                  description: Optional image description
            encoding:
              file:
                contentType: image/png, image/jpeg
      responses:
        '200':
          description: Avatar uploaded
          content:
            application/json:
              schema:
                type: object
                properties:
                  avatar_url:
```

```
                    type: string
                    format: uri

  /data/import:
    post:
      summary: Import data from file
      requestBody:
        required: true
        content:
          # JSON
          application/json:
            schema:
              $ref: '#/components/schemas/ImportData'

          # CSV
          text/csv:
            schema:
              type: string
            example: |
              name,email,age
              Mario Rossi,mario@example.com,30
              Laura Bianchi,laura@example.com,25

          # XML
          application/xml:
            schema:
              type: object
```

11.11 Callbacks e Webhooks

Listing 11.11: Definizione Callbacks

```
paths:
  /webhooks/subscribe:
    post:
      summary: Subscribe to webhooks
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required:
                - url
                - events
              properties:
                url:
                  type: string
                  format: uri
                  description: Webhook callback URL
                events:
                  type: array
                  items:
                    type: string
                    enum: [user.created, user.updated, order.placed]
      responses:
        '201':
```

```
          description: Webhook subscription created

      # Callback definition
      callbacks:
        userEvent:
          '{$request.body#/url}':
            post:
              summary: User event notification
              requestBody:
                required: true
                content:
                  application/json:
                    schema:
                      type: object
                      properties:
                        event:
                          type: string
                          enum: [user.created, user.updated]
                        timestamp:
                          type: string
                          format: date-time
                        data:
                          $ref: '#/components/schemas/User'
              responses:
                '200':
                  description: Webhook received successfully
```

11.12 Spec Completa E-commerce API

Listing 11.12: OpenAPI 3.0 Completa - E-commerce

```
openapi: 3.0.3
info:
  title: E-commerce API
  version: 2.0.0
  description: |
    # E-commerce REST API

    API completa per piattaforma e-commerce.

    ## Funzionalità

    - **Prodotti**: Gestione catalogo prodotti e categorie
    - **Carrello**: Carrello acquisti multi-item
    - **Ordini**: Gestione ordini e pagamenti
    - **Utenti**: Autenticazione e profili utente

    ## Autenticazione

    L'API usa OAuth 2.0. Ottieni un access token tramite:

    ```bash
    curl -X POST https://api.example.com/oauth/token \
      -d grant_type=client_credentials \
      -d client_id=YOUR_CLIENT_ID \
      -d client_secret=YOUR_CLIENT_SECRET
    ```
```

```
27
28      Includi il token nelle richieste:
29
30      ```
31      Authorization: Bearer YOUR_ACCESS_TOKEN
32      ```
33
34    contact:
35      name: API Support
36      email: api-support@example.com
37      url: https://example.com/support
38    license:
39      name: MIT
40      url: https://opensource.org/licenses/MIT
41
42  servers:
43    - url: https://api.example.com/v2
44      description: Production
45    - url: https://sandbox-api.example.com/v2
46      description: Sandbox
47
48  tags:
49    - name: Products
50      description: Product catalog management
51    - name: Categories
52      description: Product categories
53    - name: Cart
54      description: Shopping cart operations
55    - name: Orders
56      description: Order management
57    - name: Users
58      description: User management
59    - name: Auth
60      description: Authentication
61
62  paths:
63    # Products
64    /products:
65      get:
66        summary: List products
67        tags: [Products]
68        parameters:
69          - $ref: '#/components/parameters/PageParam'
70          - $ref: '#/components/parameters/PerPageParam'
71          - name: category
72            in: query
73            schema:
74              type: string
75          - name: price_min
76            in: query
77            schema:
78              type: number
79          - name: price_max
80            in: query
81            schema:
82              type: number
83          - name: in_stock
84            in: query
```

```yaml
          schema:
            type: boolean
      responses:
        '200':
          description: Products list
          content:
            application/json:
              schema:
                type: object
                properties:
                  data:
                    type: array
                    items:
                      $ref: '#/components/schemas/Product'
                  pagination:
                    $ref: '#/components/schemas/Pagination'

    post:
      summary: Create product
      tags: [Products]
      security:
        - oauth2: [write:products]
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ProductCreate'
      responses:
        '201':
          description: Product created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Product'

  /products/{productId}:
    parameters:
      - name: productId
        in: path
        required: true
        schema:
          type: string

    get:
      summary: Get product details
      tags: [Products]
      responses:
        '200':
          description: Product details
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Product'
        '404':
          $ref: '#/components/responses/NotFound'

  # Cart
```

```yaml
  /cart:
    get:
      summary: Get current cart
      tags: [Cart]
      security:
        - bearerAuth: []
      responses:
        '200':
          description: Cart contents
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Cart'

  /cart/items:
    post:
      summary: Add item to cart
      tags: [Cart]
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [product_id, quantity]
              properties:
                product_id:
                  type: string
                quantity:
                  type: integer
                  minimum: 1
      responses:
        '200':
          description: Item added
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Cart'

  # Orders
  /orders:
    get:
      summary: List orders
      tags: [Orders]
      security:
        - bearerAuth: []
      parameters:
        - $ref: '#/components/parameters/PageParam'
        - name: status
          in: query
          schema:
            type: string
            enum: [pending, processing, shipped, delivered, cancelled]
      responses:
        '200':
          description: Orders list
```

```
          content:
            application/json:
              schema:
                type: object
                properties:
                  data:
                    type: array
                    items:
                      $ref: '#/components/schemas/Order'

    post:
      summary: Create order from cart
      tags: [Orders]
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/OrderCreate'
      responses:
        '201':
          description: Order created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Order'

components:
  schemas:
    Product:
      type: object
      properties:
        id:
          type: string
        name:
          type: string
        description:
          type: string
        price:
          type: number
          format: float
        currency:
          type: string
          example: EUR
        category:
          type: string
        images:
          type: array
          items:
            type: string
            format: uri
        in_stock:
          type: boolean
        stock_quantity:
          type: integer

```

```
    ProductCreate:
      type: object
      required: [name, price, category]
      properties:
        name:
          type: string
        description:
          type: string
        price:
          type: number
          minimum: 0
        category:
          type: string

    Cart:
      type: object
      properties:
        id:
          type: string
        user_id:
          type: string
        items:
          type: array
          items:
            $ref: '#/components/schemas/CartItem'
        subtotal:
          type: number
        total:
          type: number

    CartItem:
      type: object
      properties:
        product:
          $ref: '#/components/schemas/Product'
        quantity:
          type: integer
        price:
          type: number

    Order:
      type: object
      properties:
        id:
          type: string
        order_number:
          type: string
        status:
          type: string
          enum: [pending, processing, shipped, delivered, cancelled]
        items:
          type: array
          items:
            $ref: '#/components/schemas/OrderItem'
        total:
          type: number
        created_at:
          type: string
```

```
        format: date-time

    OrderItem:
      type: object
      properties:
        product_id:
          type: string
        product_name:
          type: string
        quantity:
          type: integer
        price:
          type: number

    OrderCreate:
      type: object
      required: [shipping_address, payment_method]
      properties:
        shipping_address:
          $ref: '#/components/schemas/Address'
        payment_method:
          type: string
          enum: [credit_card, paypal, bank_transfer]

    Address:
      type: object
      required: [street, city, postal_code, country]
      properties:
        street:
          type: string
        city:
          type: string
        postal_code:
          type: string
        country:
          type: string
          pattern: '^[A-Z]{2}$'

    Pagination:
      type: object
      properties:
        total:
          type: integer
        page:
          type: integer
        per_page:
          type: integer
        pages:
          type: integer

    Error:
      type: object
      properties:
        error:
          type: object
          properties:
            code:
              type: string
```

```
            message:
              type: string

  parameters:
    PageParam:
      name: page
      in: query
      schema:
        type: integer
        minimum: 1
        default: 1

    PerPageParam:
      name: per_page
      in: query
      schema:
        type: integer
        minimum: 1
        maximum: 100
        default: 10

  responses:
    NotFound:
      description: Resource not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'

  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT

    oauth2:
      type: oauth2
      flows:
        clientCredentials:
          tokenUrl: https://api.example.com/oauth/token
          scopes:
            read:products: Read products
            write:products: Create/update products
            read:orders: Read orders
            write:orders: Create orders

security:
  - bearerAuth: []
```

11.13 Swagger UI

11.13.1 Hosting Swagger UI

Listing 11.13: Swagger UI HTML

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>API Documentation</title>
  <link rel="stylesheet" href="https://unpkg.com/swagger-ui-dist@5/
     swagger-ui.css" />
</head>
<body>
  <div id="swagger-ui"></div>

  <script src="https://unpkg.com/swagger-ui-dist@5/swagger-ui-bundle.js"
     ></script>
  <script src="https://unpkg.com/swagger-ui-dist@5/swagger-ui-standalone
     -preset.js"></script>
  <script>
    window.onload = () => {
      window.ui = SwaggerUIBundle({
        url: '/api/openapi.yaml',
        dom_id: '#swagger-ui',
        deepLinking: true,
        presets: [
          SwaggerUIBundle.presets.apis,
          SwaggerUIStandalonePreset
        ],
        plugins: [
          SwaggerUIBundle.plugins.DownloadUrl
        ],
        layout: "StandaloneLayout"
      });
    };
  </script>
</body>
</html>
\end{lstlisting>

\subsection{Servire OpenAPI con Flask}

\begin{lstlisting}[language=Python, caption=Flask OpenAPI Endpoint]
from flask import Flask, send_file
import yaml

app = Flask(__name__)

@app.route('/api/openapi.yaml')
def openapi_spec():
    """Serve OpenAPI specification"""
    return send_file('openapi.yaml', mimetype='application/x-yaml')

@app.route('/api/openapi.json')
def openapi_spec_json():
    """Serve OpenAPI as JSON"""
    with open('openapi.yaml') as f:
        spec = yaml.safe_load(f)
    return jsonify(spec)

@app.route('/docs')
def api_docs():
    """Serve Swagger UI"""
    return send_file('swagger-ui.html')
```

11.14 Code Generation

11.14.1 Generare Client

Listing 11.14: OpenAPI Generator - Client

```
# Installa OpenAPI Generator
npm install -g @openapitools/openapi-generator-cli

# Genera client Python
openapi-generator-cli generate \
  -i openapi.yaml \
  -g python \
  -o ./client/python \
  --additional-properties=packageName=ecommerce_client

# Genera client JavaScript/TypeScript
openapi-generator-cli generate \
  -i openapi.yaml \
  -g typescript-axios \
  -o ./client/typescript

# Genera client Java
openapi-generator-cli generate \
  -i openapi.yaml \
  -g java \
  -o ./client/java \
  --additional-properties=library=okhttp-gson
```

11.14.2 Utilizzo Client Generato

Listing 11.15: Uso Client Python Generato

```
from ecommerce_client import ApiClient, Configuration
from ecommerce_client.api import ProductsApi, OrdersApi
from ecommerce_client.models import ProductCreate

# Configurazione
config = Configuration(
    host="https://api.example.com/v2",
    access_token="YOUR_ACCESS_TOKEN"
)

# Client
with ApiClient(config) as api_client:
    # Products API
    products_api = ProductsApi(api_client)

    # List products
    products = products_api.list_products(
        page=1,
        per_page=10,
        category="electronics"
    )

    print(f"Found {len(products.data)} products")

    # Create product
```

```
    new_product = ProductCreate(
        name="Laptop",
        price=999.99,
        category="electronics"
    )

    created = products_api.create_product(new_product)
    print(f"Created product ID: {created.id}")
```

11.15 Validazione

11.15.1 Validare Spec OpenAPI

Listing 11.16: Validazione OpenAPI

```
# Con openapi-generator
openapi-generator-cli validate -i openapi.yaml

# Con speccy
npm install -g speccy
speccy lint openapi.yaml

# Con swagger-cli
npm install -g @apidevtools/swagger-cli
swagger-cli validate openapi.yaml
\end{lstlisting>

\subsection{Validare Request/Response}

\begin{lstlisting}[language=Python, caption=Validazione Runtime con
    OpenAPI]
from flask import Flask, request, jsonify
from openapi_core import create_spec
from openapi_core.validation.request.validators import RequestValidator
from openapi_core.contrib.flask import FlaskOpenAPIRequest

app = Flask(__name__)

# Carica spec
with open('openapi.yaml') as f:
    spec_dict = yaml.safe_load(f)
spec = create_spec(spec_dict)

# Middleware validazione
@app.before_request
def validate_request():
    """Valida request contro OpenAPI spec"""
    if request.method in ['POST', 'PUT', 'PATCH']:
        openapi_request = FlaskOpenAPIRequest(request)
        validator = RequestValidator(spec)
        result = validator.validate(openapi_request)

        if result.errors:
            errors = [
                {
                    'path': '.'.join(str(p) for p in err.schema_path),
                    'message': err.message
```

```
42                  }
43                  for err in result.errors
44              ]
45
46              return jsonify({
47                  'error': {
48                      'code': 'schema_validation_error',
49                      'message': 'Request does not match OpenAPI schema',
50                      'errors': errors
51                  }
52              }), 400
```

11.16 Best Practices

Checklist OpenAPI

- Usa versione OpenAPI 3.0 o superiore
- Includi esempi per tutti gli schema
- Documenta tutti i parametri e proprietà
- Definisci tutti i possibili status code
- Usa $ref per riutilizzare componenti
- Testa la spec con validator
- Mantieni spec sincronizzata con codice
- Versionna la spec insieme al codice
- Genera e testa i client
- Usa descrizioni Markdown per formattazione

11.17 Riepilogo

OpenAPI 3.0 è lo standard per documentare REST API. Fornisce:

- Documentazione interattiva con Swagger UI
- Validazione automatica request/response
- Code generation per client e server
- Contract-first development
- Testing automatizzato

Investire tempo in una spec OpenAPI completa ripaga enormemente in termini di developer experience e manutenibilità.

Capitolo 12

Best Practices

Questo capitolo raccoglie le best practices per progettare, implementare e mantenere REST API di qualità production-ready.

12.1 Design API

12.1.1 Naming Conventions

Regole per URL

- Usa **nomi al plurale** per collezioni: `/users`, `/products`
- Usa **kebab-case** per URL multi-parola: `/order-items`
- **Evita verbi** negli URL (usa HTTP methods)
- Usa **sostantivi** per identificare risorse
- Mantieni gerarchia **massimo 3 livelli**

Listing 12.1: Esempi Naming - Good vs Bad

```
# GOOD
GET    /users
GET    /users/123
POST   /users
GET    /users/123/orders
GET    /products?category=electronics
DELETE /users/123/sessions

# BAD
GET    /getUsers
GET    /user/123           # singolare
POST   /createUser         # verbo
GET    /user_orders/123    # snake_case
GET    /users/123/orders/456/items/789/details  # troppo profondo
```

12.1.2 Resource Hierarchies

Listing 12.2: Gerarchia Risorse

```
# Risorsa principale
GET /users/123
```

```
3
4  # Sotto-risorsa
5  GET /users/123/orders
6
7  # Risorsa correlata specifica
8  GET /users/123/orders/456
9
10 # EVITARE gerarchie troppo profonde
11 # BAD: /users/123/orders/456/items/789/details
12 # BETTER: /order-items/789 o /users/123/order-items/789
```

12.1.3 Versionamento API

Listing 12.3: Strategie di Versionamento

```
1  # URL versioning (RACCOMANDATO)
2  GET https://api.example.com/v1/users
3  GET https://api.example.com/v2/users
4
5  # Header versioning
6  GET https://api.example.com/users
7  Accept: application/vnd.example.v2+json
8
9  # Query parameter (SCONSIGLIATO)
10 GET https://api.example.com/users?version=2
11
12 # Custom header
13 GET https://api.example.com/users
14 API-Version: 2
```

Quando Incrementare la Versione

Incrementa la versione major quando fai **breaking changes**:

- Rimuovi endpoint o campi
- Cambi tipo di dato di un campo
- Modifichi comportamento esistente in modo incompatibile
- Cambi struttura errori

Non serve nuova versione per:

- Aggiungi nuovi endpoint
- Aggiungi nuovi campi opzionali
- Fix di bug che non cambiano contratto

12.2 HATEOAS (Hypermedia as the Engine of Application State)

12.2.1 Principio

HATEOAS è un constraint REST che richiede che le risposte includano link per navigare l'API.

Listing 12.4: Response con HATEOAS

```
{
  "id": 123,
  "name": "Mario Rossi",
  "email": "mario@example.com",
  "status": "active",

  "_links": {
    "self": {
      "href": "https://api.example.com/users/123"
    },
    "orders": {
      "href": "https://api.example.com/users/123/orders"
    },
    "avatar": {
      "href": "https://api.example.com/users/123/avatar"
    },
    "deactivate": {
      "href": "https://api.example.com/users/123/deactivate",
      "method": "POST"
    }
  },

  "_actions": {
    "update": {
      "href": "https://api.example.com/users/123",
      "method": "PUT",
      "fields": [
        {"name": "name", "type": "string", "required": true},
        {"name": "email", "type": "email", "required": true}
      ]
    }
  }
}
```

12.2.2 Benefici HATEOAS

- **Discoverability**: Client può esplorare API seguendo link
- **Decoupling**: Client meno dipendente da URL hardcoded
- **Evolvibilità**: Server può cambiare URL senza rompere client
- **Self-documenting**: API più facile da capire

12.2.3 Implementazione

Listing 12.5: Generazione Link HATEOAS

```
from flask import url_for, request

def add_links(resource, resource_type, resource_id):
    """Aggiungi link HATEOAS a risorsa"""

    base_url = request.url_root.rstrip('/')

    resource['_links'] = {
```

```
        'self': {
            'href': f"{base_url}/api/{resource_type}/{resource_id}"
        }
    }

    # Link specifici per tipo risorsa
    if resource_type == 'users':
        resource['_links']['orders'] = {
            'href': f"{base_url}/api/users/{resource_id}/orders"
        }

        if resource.get('status') == 'active':
            resource['_links']['deactivate'] = {
                'href': f"{base_url}/api/users/{resource_id}/deactivate"
                    ,
                'method': 'POST'
            }

    return resource

# Uso
user = get_user(123)
user_with_links = add_links(user, 'users', 123)
```

12.3 Caching

12.3.1 Cache Headers

Listing 12.6: HTTP Cache Headers

```
# Cache pubblica (CDN, proxy, browser)
HTTP/1.1 200 OK
Cache-Control: public, max-age=3600
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
Last-Modified: Mon, 13 Nov 2023 12:00:00 GMT
Expires: Mon, 13 Nov 2023 13:00:00 GMT

# Cache privata (solo browser)
Cache-Control: private, max-age=300

# No cache (rivalidazione sempre richiesta)
Cache-Control: no-cache

# No store (non cachare mai)
Cache-Control: no-store, no-cache, must-revalidate

# Cache con validazione
Cache-Control: public, max-age=3600, must-revalidate
```

12.3.2 ETag per Validazione

Listing 12.7: Conditional Requests con ETag

```
# Prima richiesta
GET /api/users/123
Accept: application/json
```

```
4
5  # Response con ETag
6  HTTP/1.1 200 OK
7  ETag: "abc123"
8  Cache-Control: private, max-age=60
9  Content-Type: application/json
10
11 {"id": 123, "name": "Mario"}
12
13 # Richiesta successiva con validazione
14 GET /api/users/123
15 If-None-Match: "abc123"
16
17 # Se non modificato
18 HTTP/1.1 304 Not Modified
19 ETag: "abc123"
20
21 # Se modificato
22 HTTP/1.1 200 OK
23 ETag: "xyz789"
24 Content-Type: application/json
25
26 {"id": 123, "name": "Mario Rossi"}
```

12.3.3 Implementazione ETag

Listing 12.8: ETag Generation e Validation

```
1  import hashlib
2  from flask import request, jsonify, make_response
3
4  def generate_etag(data):
5      """Genera ETag da dati"""
6      content = json.dumps(data, sort_keys=True)
7      return hashlib.md5(content.encode()).hexdigest()
8
9  @app.route('/api/users/<int:user_id>')
10 def get_user(user_id):
11     user = User.query.get_or_404(user_id)
12     user_dict = user.to_dict()
13
14     # Genera ETag
15     etag = generate_etag(user_dict)
16
17     # Verifica If-None-Match
18     if_none_match = request.headers.get('If-None-Match')
19     if if_none_match == etag:
20         # Risorsa non modificata
21         response = make_response('', 304)
22         response.headers['ETag'] = etag
23         return response
24
25     # Risorsa modificata o prima richiesta
26     response = make_response(jsonify(user_dict))
27     response.headers['ETag'] = etag
28     response.headers['Cache-Control'] = 'private, max-age=60'
29
```

```
30      return response
```

12.3.4 Vary Header

Listing 12.9: Vary Header per Cache Varianti

```
1  # Cache dipende da Accept header
2  HTTP/1.1 200 OK
3  Content-Type: application/json
4  Vary: Accept
5  Cache-Control: public, max-age=3600
6
7  # Cache dipende da lingua e encoding
8  HTTP/1.1 200 OK
9  Vary: Accept-Language, Accept-Encoding
10 Cache-Control: public, max-age=3600
```

12.4 Compressione

12.4.1 Content Encoding

Listing 12.10: Compressione Response

```
1  # Request con Accept-Encoding
2  GET /api/users
3  Accept-Encoding: gzip, deflate, br
4
5  # Response compressa
6  HTTP/1.1 200 OK
7  Content-Type: application/json
8  Content-Encoding: gzip
9  Content-Length: 1234
10 Vary: Accept-Encoding
11
12 [binary compressed data]
```

12.4.2 Implementazione con Flask

Listing 12.11: Compressione Flask

```
1  from flask import Flask
2  from flask_compress import Compress
3
4  app = Flask(__name__)
5
6  # Abilita compressione automatica
7  Compress(app)
8
9  # Configurazione
10 app.config['COMPRESS_MIMETYPES'] = [
11     'application/json',
12     'text/html',
13     'text/css',
14     'text/xml',
15     'application/javascript',
```

```
]
app.config['COMPRESS_LEVEL'] = 6  # 1-9
app.config['COMPRESS_MIN_SIZE'] = 500  # bytes

@app.route('/api/users')
def get_users():
    # Response automaticamente compressa se > 500 bytes
    return jsonify(users)
```

Quando Comprimere

- **SI**: Response > 500 bytes, JSON, XML, testo
- **NO**: Immagini (già compresse), response piccole (overhead)
- **Algoritmi**: gzip (standard), br (brotli, migliore ratio)

12.5 CORS (Cross-Origin Resource Sharing)

12.5.1 CORS Headers

Listing 12.12: CORS Response Headers

```
# Preflight request (OPTIONS)
OPTIONS /api/users
Origin: https://example.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Content-Type, Authorization

# Preflight response
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://example.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 86400
Access-Control-Allow-Credentials: true

# Actual request
POST /api/users
Origin: https://example.com
Content-Type: application/json
Authorization: Bearer token123

# Response
HTTP/1.1 201 Created
Access-Control-Allow-Origin: https://example.com
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: Location, X-Request-ID
```

12.5.2 Configurazione CORS

Listing 12.13: CORS con Flask-CORS

```
from flask import Flask
from flask_cors import CORS

```

```python
app = Flask(__name__)

# CORS per tutti gli endpoint
CORS(app)

# CORS con configurazione
CORS(app, resources={
    r"/api/*": {
        "origins": [
            "https://example.com",
            "https://app.example.com"
        ],
        "methods": ["GET", "POST", "PUT", "DELETE"],
        "allow_headers": ["Content-Type", "Authorization"],
        "expose_headers": ["X-Total-Count", "Link"],
        "supports_credentials": True,
        "max_age": 3600
    }
})

# CORS per endpoint specifici
@app.route('/api/public/data')
@cross_origin(origins="*")  # Pubblico
def public_data():
    return jsonify(data)

@app.route('/api/private/data')
@cross_origin(
    origins=["https://example.com"],
    supports_credentials=True
)
def private_data():
    return jsonify(data)
```

CORS Security

- **MAI** usare `Access-Control-Allow-Origin: *` con credenziali
- Specifica esattamente gli origin permessi
- Limita methods e headers a quelli necessari
- Valida sempre l'origin server-side

12.6 Security Headers

12.6.1 Headers di Sicurezza Essenziali

Listing 12.14: Security Headers

```
# Strict Transport Security (HTTPS only)
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload

# Previeni clickjacking
X-Frame-Options: DENY

# Previeni MIME sniffing
```

```
X-Content-Type-Options: nosniff

# XSS Protection (legacy, usa CSP)
X-XSS-Protection: 1; mode=block

# Content Security Policy
Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'

# Referrer Policy
Referrer-Policy: strict-origin-when-cross-origin

# Permissions Policy
Permissions-Policy: geolocation=(), camera=(), microphone=()
```

12.6.2 Implementazione Security Headers

Listing 12.15: Security Headers Middleware

```python
from flask import Flask

app = Flask(__name__)

@app.after_request
def add_security_headers(response):
    """Aggiungi security headers a tutte le response"""

    # HSTS
    response.headers['Strict-Transport-Security'] = \
        'max-age=31536000; includeSubDomains; preload'

    # Prevent clickjacking
    response.headers['X-Frame-Options'] = 'DENY'

    # Prevent MIME sniffing
    response.headers['X-Content-Type-Options'] = 'nosniff'

    # XSS Protection
    response.headers['X-XSS-Protection'] = '1; mode=block'

    # Content Security Policy
    response.headers['Content-Security-Policy'] = \
        "default-src 'none'; script-src 'self'; connect-src 'self'"

    # Referrer Policy
    response.headers['Referrer-Policy'] = 'strict-origin-when-cross-origin'

    # Permissions Policy
    response.headers['Permissions-Policy'] = \
        'geolocation=(), camera=(), microphone=()'

    # Remove server header
    response.headers.pop('Server', None)

    return response
```

12.7 Input Validation

12.7.1 Validazione Completa

Listing 12.16: Input Validation Best Practices

```python
from marshmallow import Schema, fields, validate, ValidationError
from flask import request, jsonify

class UserCreateSchema(Schema):
    """Schema validazione creazione utente"""

    email = fields.Email(
        required=True,
        error_messages={
            'required': 'Email is required',
            'invalid': 'Invalid email format'
        }
    )

    name = fields.Str(
        required=True,
        validate=[
            validate.Length(min=1, max=100),
            validate.Regexp(
                r'^[a-zA-Z\s]+$',
                error='Name can only contain letters and spaces'
            )
        ]
    )

    age = fields.Int(
        required=False,
        validate=validate.Range(min=18, max=120)
    )

    password = fields.Str(
        required=True,
        load_only=True,  # Non includere in dump
        validate=[
            validate.Length(min=8, max=128),
            validate.Regexp(
                r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)',
                error='Password must contain uppercase, lowercase, and
                    number'
            )
        ]
    )

    role = fields.Str(
        validate=validate.OneOf(['user', 'admin', 'moderator']),
        missing='user'
    )

@app.route('/api/users', methods=['POST'])
def create_user():
    # Valida JSON
    if not request.is_json:
        return jsonify({
```

```
            'error': 'Content-Type must be application/json'
        }), 400

    data = request.get_json()

    # Valida schema
    schema = UserCreateSchema()
    try:
        validated_data = schema.load(data)
    except ValidationError as err:
        return jsonify({
            'error': {
                'code': 'validation_error',
                'message': 'Validation failed',
                'errors': err.messages
            }
        }), 400

    # Validazione business logic
    if User.query.filter_by(email=validated_data['email']).first():
        return jsonify({
            'error': {
                'code': 'duplicate_email',
                'message': 'Email already exists'
            }
        }), 409

    # Crea utente
    user = User(**validated_data)
    db.session.add(user)
    db.session.commit()

    return jsonify(user.to_dict()), 201
```

12.7.2 Sanitizzazione Input

Listing 12.17: Input Sanitization

```
import bleach
from html import escape

def sanitize_string(value, allow_html=False):
    """Sanitizza stringa"""

    if not isinstance(value, str):
        return value

    # Trim whitespace
    value = value.strip()

    if allow_html:
        # Permetti solo tag sicuri
        allowed_tags = ['b', 'i', 'u', 'em', 'strong', 'a', 'p', 'br']
        allowed_attrs = {'a': ['href', 'title']}
        value = bleach.clean(
            value,
            tags=allowed_tags,
```

```
            attributes=allowed_attrs,
            strip=True
        )
    else:
        # Escape HTML
        value = escape(value)

    return value

def sanitize_dict(data, allow_html_fields=None):
    """Sanitizza tutti i campi di un dict"""
    allow_html_fields = allow_html_fields or []

    sanitized = {}
    for key, value in data.items():
        if isinstance(value, str):
            sanitized[key] = sanitize_string(
                value,
                allow_html=key in allow_html_fields
            )
        elif isinstance(value, dict):
            sanitized[key] = sanitize_dict(value, allow_html_fields)
        elif isinstance(value, list):
            sanitized[key] = [
                sanitize_string(v) if isinstance(v, str) else v
                for v in value
            ]
        else:
            sanitized[key] = value

    return sanitized
```

12.8 Idempotenza

12.8.1 Idempotency Keys

Listing 12.18: Uso Idempotency-Key

```
# Request con idempotency key
POST /api/payments
Idempotency-Key: a7b3c9d2-1234-5678-9abc-def012345678
Content-Type: application/json

{
  "amount": 100.00,
  "currency": "EUR",
  "source": "card_123"
}

# Prima chiamata: esegue operazione
HTTP/1.1 201 Created

# Chiamate successive con stesso key: ritorna stesso risultato
HTTP/1.1 200 OK
Idempotent-Replayed: true
```

12.8.2 Implementazione Idempotenza

Listing 12.19: Idempotency Middleware

```python
import uuid
from functools import wraps
from flask import request, jsonify
import redis

redis_client = redis.Redis()

def idempotent(expire_seconds=86400):
    """Decorator per endpoint idempotenti"""

    def decorator(f):
        @wraps(f)
        def wrapped(*args, **kwargs):
            # Estrai idempotency key
            key = request.headers.get('Idempotency-Key')

            if not key:
                return jsonify({
                    'error': 'Idempotency-Key header required'
                }), 400

            # Valida formato UUID
            try:
                uuid.UUID(key)
            except ValueError:
                return jsonify({
                    'error': 'Idempotency-Key must be valid UUID'
                }), 400

            # Chiave cache
            cache_key = f"idempotency:{key}"

            # Verifica se già eseguito
            cached = redis_client.get(cache_key)

            if cached:
                # Ritorna risposta cached
                import json
                response_data = json.loads(cached)
                response = make_response(
                    jsonify(response_data['body']),
                    response_data['status']
                )
                response.headers['Idempotent-Replayed'] = 'true'
                return response

            # Esegui operazione
            response = f(*args, **kwargs)

            # Cache response
            if 200 <= response.status_code < 300:
                response_data = {
                    'status': response.status_code,
                    'body': response.get_json()
                }
```

```
                redis_client.setex(
                    cache_key,
                    expire_seconds,
                    json.dumps(response_data)
                )

            return response

        return wrapped
    return decorator

# Uso
@app.route('/api/payments', methods=['POST'])
@idempotent(expire_seconds=86400)  # 24 ore
def create_payment():
    # Logica creazione pagamento
    payment = create_payment_logic()
    return jsonify(payment), 201
```

12.9 Logging e Monitoring

12.9.1 Structured Logging

Listing 12.20: Structured Logging

```
import logging
import json
import uuid
from datetime import datetime
from flask import request, g

class JSONFormatter(logging.Formatter):
    """Formatta log in JSON"""

    def format(self, record):
        log_data = {
            'timestamp': datetime.utcnow().isoformat(),
            'level': record.levelname,
            'message': record.getMessage(),
            'logger': record.name,
            'request_id': getattr(g, 'request_id', None),
            'user_id': getattr(g, 'user_id', None),
            'path': getattr(g, 'path', None),
            'method': getattr(g, 'method', None),
        }

        # Aggiungi exception se presente
        if record.exc_info:
            log_data['exception'] = self.formatException(record.exc_info
                )

        # Aggiungi extra fields
        if hasattr(record, 'extra'):
            log_data.update(record.extra)

        return json.dumps(log_data)

```

```
# Configura logging
handler = logging.StreamHandler()
handler.setFormatter(JSONFormatter())
logger = logging.getLogger('api')
logger.addHandler(handler)
logger.setLevel(logging.INFO)

@app.before_request
def before_request():
    """Setup request context"""
    g.request_id = str(uuid.uuid4())
    g.path = request.path
    g.method = request.method

    # Log request
    logger.info('Request started', extra={
        'query_params': dict(request.args),
        'headers': dict(request.headers)
    })

@app.after_request
def after_request(response):
    """Log response"""
    logger.info('Request completed', extra={
        'status_code': response.status_code,
        'response_size': response.content_length
    })

    # Aggiungi request ID a response
    response.headers['X-Request-ID'] = g.request_id

    return response
```

12.9.2 Metriche con Prometheus

Listing 12.21: Prometheus Metrics

```
from prometheus_client import Counter, Histogram, Gauge
from prometheus_client import make_wsgi_app
from werkzeug.middleware.dispatcher import DispatcherMiddleware

# Definisci metriche
http_requests_total = Counter(
    'http_requests_total',
    'Total HTTP requests',
    ['method', 'endpoint', 'status']
)

http_request_duration_seconds = Histogram(
    'http_request_duration_seconds',
    'HTTP request duration',
    ['method', 'endpoint']
)

active_requests = Gauge(
    'active_requests',
    'Number of active requests'
```

```
)

# Middleware metriche
@app.before_request
def before_request_metrics():
    active_requests.inc()
    g.start_time = time.time()

@app.after_request
def after_request_metrics(response):
    active_requests.dec()

    # Incrementa counter
    http_requests_total.labels(
        method=request.method,
        endpoint=request.endpoint or 'unknown',
        status=response.status_code
    ).inc()

    # Registra durata
    duration = time.time() - g.start_time
    http_request_duration_seconds.labels(
        method=request.method,
        endpoint=request.endpoint or 'unknown'
    ).observe(duration)

    return response

# Esponi metriche su /metrics
app.wsgi_app = DispatcherMiddleware(
    app.wsgi_app,
    {'/metrics': make_wsgi_app()}
)
```

12.10 Health Checks

12.10.1 Endpoint Health

Listing 12.22: Health Check Endpoint

```
from flask import jsonify
import psutil

@app.route('/health')
def health_check():
    """Health check semplice"""
    return jsonify({
        'status': 'healthy',
        'timestamp': datetime.utcnow().isoformat()
    })

@app.route('/health/detailed')
def detailed_health():
    """Health check dettagliato"""

    health_data = {
        'status': 'healthy',
```

```
        'timestamp': datetime.utcnow().isoformat(),
        'checks': {}
    }

    # Database check
    try:
        db.session.execute('SELECT 1')
        health_data['checks']['database'] = {
            'status': 'healthy',
            'response_time_ms': 5
        }
    except Exception as e:
        health_data['status'] = 'unhealthy'
        health_data['checks']['database'] = {
            'status': 'unhealthy',
            'error': str(e)
        }

    # Redis check
    try:
        redis_client.ping()
        health_data['checks']['redis'] = {
            'status': 'healthy'
        }
    except Exception as e:
        health_data['status'] = 'degraded'
        health_data['checks']['redis'] = {
            'status': 'unhealthy',
            'error': str(e)
        }

    # System resources
    health_data['system'] = {
        'cpu_percent': psutil.cpu_percent(),
        'memory_percent': psutil.virtual_memory().percent,
        'disk_percent': psutil.disk_usage('/').percent
    }

    # Status code
    status_code = 200 if health_data['status'] == 'healthy' else 503

    return jsonify(health_data), status_code
```

12.11 Deprecation

12.11.1 Comunicare Deprecazione

Listing 12.23: Deprecation Headers

```
# Endpoint deprecato
GET /api/v1/users

HTTP/1.1 200 OK
Deprecation: true
Sunset: Sat, 31 Dec 2024 23:59:59 GMT
Link: <https://api.example.com/v2/users>; rel="alternate"
```

```
Warning: 299 - "This endpoint is deprecated and will be removed on
    2024-12-31"

{
  "data": [...],
  "deprecation_notice": {
    "deprecated": true,
    "sunset_date": "2024-12-31",
    "alternative": "https://api.example.com/v2/users",
    "migration_guide": "https://docs.example.com/migration/v1-to-v2"
  }
}
```

12.11.2 Implementazione Deprecation

Listing 12.24: Deprecation Decorator

```python
from functools import wraps
from datetime import datetime

def deprecated(sunset_date, alternative=None, message=None):
    """Marca endpoint come deprecato"""

    def decorator(f):
        @wraps(f)
        def wrapped(*args, **kwargs):
            response = make_response(f(*args, **kwargs))

            # Headers
            response.headers['Deprecation'] = 'true'
            response.headers['Sunset'] = sunset_date

            if alternative:
                response.headers['Link'] = f'<{alternative}>; rel="
                    alternate"'

            warning = message or f"This endpoint is deprecated and will
                be removed on {sunset_date}"
            response.headers['Warning'] = f'299 - "{warning}"'

            # Log deprecation usage
            logger.warning(
                'Deprecated endpoint accessed',
                extra={
                    'endpoint': request.endpoint,
                    'sunset_date': sunset_date,
                    'user_agent': request.user_agent.string
                }
            )

            return response

        return wrapped
    return decorator

# Uso
@app.route('/api/v1/users')
```

```python
@deprecated(
    sunset_date='Sat, 31 Dec 2024 23:59:59 GMT',
    alternative='https://api.example.com/v2/users',
    message='Migra a v2 per nuove funzionalità'
)
def get_users_v1():
    return jsonify(users)
```

12.12 Checklist API Production-Ready

Checklist Completa

Design

- URL RESTful con nomi plurali
- Versionamento implementato
- HATEOAS per navigazione
- Documentazione OpenAPI completa

Sicurezza

- HTTPS obbligatorio
- Autenticazione robusta (OAuth 2.0/JWT)
- Autorizzazione granulare
- Input validation e sanitization
- Rate limiting implementato
- Security headers configurati
- CORS correttamente configurato

Performance

- Caching implementato (ETag, Cache-Control)
- Compressione abilitata
- Paginazione per liste
- Database indexes su campi filtrabili
- Query optimization

Affidabilità

- Error handling consistente
- Idempotenza per operazioni critiche
- Health check endpoint
- Graceful degradation

- Timeout configurati

Observability

- Structured logging
- Metriche (Prometheus/StatsD)
- Request ID tracking
- Error tracking (Sentry)
- Performance monitoring (APM)

Developer Experience

- Documentazione chiara e aggiornata
- Esempi pratici (cURL, code samples)
- SDK/client libraries
- Sandbox environment
- Changelog e migration guides

12.13 Riepilogo

Le best practices per REST API includono:

1. **Design**: URL RESTful, versionamento, HATEOAS
2. **Performance**: Caching, compressione, paginazione
3. **Sicurezza**: HTTPS, autenticazione, validazione, security headers
4. **Affidabilità**: Error handling, idempotenza, health checks
5. **Osservabilità**: Logging, metriche, monitoring
6. **Developer Experience**: Documentazione, esempi, SDK

Seguire queste pratiche garantisce API robuste, sicure, performanti e facili da usare.

Appendice A

Appendice: HTTP Headers Reference

Questa appendice fornisce una reference completa degli HTTP headers più comunemente usati nelle REST API.

A.1 Request Headers

A.1.1 General Headers

Header	Descrizione ed Esempio
`Host`	Specifica l'host e porta del server. `Host: api.example.com`
`User-Agent`	Identifica il client che effettua la richiesta. `User-Agent: Mozilla/5.0 (Windows NT 10.0)`
`Referer`	URL della pagina da cui proviene la richiesta. `Referer: https://example.com/page`

Tabella A.1: General Request Headers

A.1.2 Content Negotiation

Header	Descrizione ed Esempio

`Accept`	Media types accettati dal client.
	`Accept: application/json`
	`Accept: application/json, text/html; q=0.9`

`Accept-Language`	Lingue preferite per la risposta. `Accept-Language: it-IT, it; q=0.9, en; q=0.8`
`Accept-Encoding`	Encoding di compressione accettati. `Accept-Encoding: gzip, deflate, br`
`Accept-Charset`	Character set accettati (deprecato, usa UTF-8). `Accept-Charset: utf-8`

Tabella A.2: Content Negotiation Headers

Header	Descrizione ed Esempio

`Authorization`	Credenziali di autenticazione.
	`Authorization: Basic dXNlcjpwYXNz`
	`Authorization: Bearer eyJhbGciOiJIUzI1NiIs...`

`Proxy-Authorization`	Credenziali per autenticazione proxy. `Proxy-Authorization: Basic cHJveHk6cGFzcw==`
`API-Key`	API key custom (non standard). `X-API-Key: your-api-key-here`

Tabella A.3: Authentication Headers

Header	Descrizione ed Esempio
`If-Modified-Since`	Richiede risorsa solo se modificata dopo data. `If-Modified-Since: Wed, 21 Oct 2023 07:28:00 GMT`
`If-None-Match`	Richiede risorsa solo se ETag è cambiato. `If-None-Match: "abc123"`
`If-Match`	Esegue operazione solo se ETag corrisponde (optimistic locking). `If-Match: "abc123"`
`If-Unmodified-Since`	Esegue operazione solo se non modificata dopo data. `If-Unmodified-Since: Wed, 21 Oct 2023 07:28:00 GMT`
`If-Range`	Richiede range solo se risorsa non modificata. `If-Range: "abc123"`

`Cache-Control`	Direttive di caching per la richiesta.
	`Cache-Control: no-cache`
	`Cache-Control: max-age=0`

Tabella A.4: Cache Control Request Headers

Header	Descrizione ed Esempio

`Content-Type`	Media type del body della richiesta.
	`Content-Type: application/json`
	`Content-Type: application/x-www-form-urlencoded`
	`Content-Type: multipart/form-data; boundary=--WebKitFormBoundary`

`Content-Length`	Dimensione del body in bytes. `Content-Length: 348`
`Content-Encoding`	Encoding applicato al body. `Content-Encoding: gzip`

Tabella A.5: Content Request Headers

Header	Descrizione ed Esempio

`Range`	Richiede un range specifico di byte.
	`Range: bytes=0-1023`
	`Range: bytes=1024-`
	`Range: bytes=-2048` (ultimi 2048 bytes)

Tabella A.6: Range Request Headers

A.1.3 Authentication

A.1.4 Cache Control

A.1.5 Content Headers

A.1.6 Range Requests

A.1.7 Custom Headers

A.2 Response Headers

A.2.1 General Response Headers

A.2.2 Content Response Headers

A.2.3 Cache Control

A.2.4 Redirection

A.2.5 Authentication

A.2.6 Rate Limiting

A.2.7 Security Headers

A.2.8 CORS

A.2.9 Pagination

A.2.10 Custom API Headers

A.2.11 Range Responses

A.3 Direttive Cache-Control

A.3.1 Request Directives

A.3.2 Response Directives

A.3.3 Esempi Comuni

Header	Descrizione ed Esempio
`X-Request-ID`	ID univoco per tracciare la richiesta. `X-Request-ID: a7b3c9d2-1234-5678-9abc-def012345678`
`X-Correlation-ID`	ID per correlazione tra microservizi. `X-Correlation-ID: trace-abc-123`
`Idempotency-Key`	Chiave per garantire idempotenza. `Idempotency-Key: a7b3c9d2-1234-5678-9abc-def012345678`
`X-Forwarded-For`	IP originale del client (tramite proxy). `X-Forwarded-For: 203.0.113.195, 70.41.3.18`
`X-Forwarded-Proto`	Protocollo originale (http/https). `X-Forwarded-Proto: https`
`X-Forwarded-Host`	Host originale richiesto. `X-Forwarded-Host: api.example.com`

Tabella A.7: Custom Request Headers

Header	Descrizione ed Esempio
`Date`	Data e ora di generazione della risposta. `Date: Wed, 13 Nov 2023 12:00:00 GMT`
`Server`	Informazioni sul server (spesso rimosso per sicurezza). `Server: nginx/1.18.0`

`Connection`	Opzioni di connessione.
	`Connection: keep-alive`
	`Connection: close`

Tabella A.8: General Response Headers

Listing A.1: Cache-Control Examples

```
# No caching (dati sensibili)
Cache-Control: no-store, no-cache, must-revalidate

# Cache privata 5 minuti
Cache-Control: private, max-age=300

# Cache pubblica 1 ora
Cache-Control: public, max-age=3600

# Cache pubblica 1 anno (asset statici immutabili)
Cache-Control: public, max-age=31536000, immutable

# Cache CDN 1 ora, browser 5 minuti
Cache-Control: public, max-age=300, s-maxage=3600

# Usa cache stale mentre rivalidazione
Cache-Control: public, max-age=600, stale-while-revalidate=300

# Usa cache stale se server down
Cache-Control: public, max-age=3600, stale-if-error=86400
```

Header	Descrizione ed Esempio
`Content-Type`	Media type della risposta. `Content-Type: application/json; charset=utf-8`
`Content-Length`	Dimensione del body in bytes. `Content-Length: 1234`
`Content-Encoding`	Encoding di compressione applicato. `Content-Encoding: gzip`
`Content-Language`	Lingua del contenuto. `Content-Language: it-IT`

`Content-Disposition`	Come il contenuto dovrebbe essere mostrato.
	`Content-Disposition: inline`
	`Content-Disposition: attachment; filename="data.json"`

Tabella A.9: Content Response Headers

Header	Descrizione ed Esempio

`Cache-Control`	Direttive di caching.
	`Cache-Control: public, max-age=3600`
	`Cache-Control: private, no-cache`
	`Cache-Control: no-store, no-cache, must-revalidate`

`ETag`	Identificatore univoco della versione della risorsa. `ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"` `ETag: W/"abc123"` (weak ETag)
`Expires`	Data di scadenza della cache (legacy, usa Cache-Control). `Expires: Wed, 13 Nov 2023 13:00:00 GMT`
`Last-Modified`	Data ultima modifica della risorsa. `Last-Modified: Wed, 13 Nov 2023 12:00:00 GMT`
`Age`	Tempo in secondi che l'oggetto è stato in cache. `Age: 3600`

`Vary`	Headers che influenzano la cache.
	`Vary: Accept-Encoding`
	`Vary: Accept, Accept-Language, Accept-Encoding`

Tabella A.10: Cache Control Response Headers

A.4 Media Types Comuni

A.5 Status Code Reference

A.5.1 1xx Informational

A.5.2 2xx Success

A.5.3 3xx Redirection

A.5.4 4xx Client Errors

A.5.5 5xx Server Errors

A.6 Quick Reference

Listing A.2: Esempio Request Completo

```
POST /api/users HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Type: application/json
Accept: application/json
```

Header	Descrizione ed Esempio
`Location`	URL a cui il client dovrebbe essere reindirizzato. Usato con 201 Created, 3xx redirects. `Location: https://api.example.com/users/123`

Tabella A.11: Redirection Headers

Header	Descrizione ed Esempio

`WWW-Authenticate`	Schema di autenticazione richiesto (con 401).
	`WWW-Authenticate: Basic realm="API"`
	`WWW-Authenticate: Bearer realm="api.example.com"`
	`WWW-Authenticate: Bearer error="invalid_token"`

`Proxy-Authenticate`	Schema di autenticazione proxy richiesto (con 407). `Proxy-Authenticate: Basic realm="Proxy"`

Tabella A.12: Authentication Response Headers

```
Accept-Language: it-IT, it; q=0.9
Accept-Encoding: gzip, deflate, br
User-Agent: MyApp/1.0
X-Request-ID: a7b3c9d2-1234-5678-9abc-def012345678
Idempotency-Key: b8c4d0e3-2345-6789-0bcd-ef1234567890
Content-Length: 123

{
  "name": "Mario Rossi",
  "email": "mario@example.com",
  "password": "SecurePass123!"
}
```

Listing A.3: Esempio Response Completo

```
HTTP/1.1 201 Created
Date: Wed, 13 Nov 2023 12:00:00 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 456
Location: https://api.example.com/users/12345
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
Cache-Control: private, no-cache
X-Request-ID: a7b3c9d2-1234-5678-9abc-def012345678
X-Response-Time: 234
RateLimit-Limit: 100
RateLimit-Remaining: 95
RateLimit-Reset: 1699876543
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
X-Frame-Options: DENY

{
  "id": 12345,
  "name": "Mario Rossi",
  "email": "mario@example.com",
  "created_at": "2023-11-13T12:00:00Z",
  "_links": {
    "self": {
      "href": "https://api.example.com/users/12345"
    }
  }
```

Header	Descrizione ed Esempio
`RateLimit-Limit`	Numero massimo di richieste consentite. `RateLimit-Limit: 100`
`RateLimit-Remaining`	Numero di richieste rimanenti. `RateLimit-Remaining: 85`
`RateLimit-Reset`	Unix timestamp di reset del limite. `RateLimit-Reset: 1699876543`

`X-RateLimit-*`	Varianti legacy con prefisso X-.
	`X-RateLimit-Limit: 100`
	`X-RateLimit-Remaining: 85`
	`X-RateLimit-Reset: 1699876543`

`Retry-After`	Secondi da attendere prima di riprovare (con 429, 503).
	`Retry-After: 3600`
	`Retry-After: Wed, 13 Nov 2023 13:00:00 GMT`

Tabella A.13: Rate Limiting Headers

Header	Descrizione ed Esempio
`Strict-Transport-Security`	Forza HTTPS (HSTS). `Strict-Transport-Security: max-age=31536000; includeSubDomains; preload`
`Content-Security-Policy`	Politica di sicurezza dei contenuti. `Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline'`
`X-Content-Type-Options`	Previene MIME sniffing. `X-Content-Type-Options: nosniff`

`X-Frame-Options`	Controlla se la pagina può essere in frame (previene clickjacking).
	`X-Frame-Options: DENY`
	`X-Frame-Options: SAMEORIGIN`

`X-XSS-Protection`	Abilita filtro XSS browser (legacy, usa CSP). `X-XSS-Protection: 1; mode=block`

`Referrer-Policy`	Controlla informazioni inviate nel Referer header.
	`Referrer-Policy: strict-origin-when-cross-origin`
	`Referrer-Policy: no-referrer`

`Permissions-Policy`	Controlla feature browser disponibili. `Permissions-Policy: geolocation=(), camera=(), microphone=()`

Tabella A.14: Security Headers

```
}
```

A.7 Riepilogo

Questa appendice ha fornito una reference completa degli HTTP headers più comuni. Punti chiave:

- **Request headers**: Content-Type, Accept, Authorization, cache headers
- **Response headers**: Content-Type, Cache-Control, ETag, security headers
- **Custom headers**: Rate limiting, pagination, API-specific
- **CORS**: Access-Control-* headers per cross-origin requests
- **Security**: HSTS, CSP, X-Frame-Options, etc

Usa questa appendice come quick reference durante lo sviluppo API.

Header	Descrizione ed Esempio

`Access-Control-Allow-Origin`	Origin permessi per CORS.
	`Access-Control-Allow-Origin: https://example.com`
	`Access-Control-Allow-Origin: *`

`Access-Control-Allow-Methods`	HTTP methods permessi. `Access-Control-Allow-Methods: GET, POST, PUT, DELETE`
`Access-Control-Allow-Headers`	Headers permessi nelle richieste. `Access-Control-Allow-Headers: Content-Type, Authorization`
`Access-Control-Allow-Credentials`	Se le credenziali sono permesse. `Access-Control-Allow-Credentials: true`
`Access-Control-Expose-Headers`	Headers esposti al JavaScript client. `Access-Control-Expose-Headers: X-Total-Count, Link`
`Access-Control-Max-Age`	Tempo cache preflight in secondi. `Access-Control-Max-Age: 86400`

Tabella A.15: CORS Headers

Header	Descrizione ed Esempio
`Link`	Link per navigazione paginazione (RFC 5988). `Link: <https://api.example.com/users?page=2>; rel="next", <https://api.example.com/users?page=1>; rel="prev"`
`X-Total-Count`	Numero totale di item (custom). `X-Total-Count: 1543`
`X-Page`	Numero pagina corrente (custom). `X-Page: 2`
`X-Per-Page`	Item per pagina (custom). `X-Per-Page: 10`

Tabella A.16: Pagination Headers

Header	Descrizione ed Esempio
`X-Request-ID`	ID della richiesta per debugging. `X-Request-ID: a7b3c9d2-1234-5678-9abc-def012345678`
`X-Response-Time`	Tempo di elaborazione della richiesta in ms. `X-Response-Time: 234`
`X-API-Version`	Versione API (alternativa a URL versioning). `X-API-Version: 2.0`
`Deprecation`	Indica endpoint deprecato. `Deprecation: true`
`Sunset`	Data di rimozione endpoint. `Sunset: Sat, 31 Dec 2024 23:59:59 GMT`
`Warning`	Avvisi (deprecation, etc). `Warning: 299 - "This endpoint is deprecated"`
`Idempotent-Replayed`	Indica che la risposta è stata ripetuta da cache idempotency. `Idempotent-Replayed: true`

Tabella A.17: Custom API Response Headers

Header	Descrizione ed Esempio

`Accept-Ranges`	Indica se il server supporta range requests.
	`Accept-Ranges: bytes`
	`Accept-Ranges: none`

`Content-Range`	Range di byte restituito (con 206 Partial Content). `Content-Range: bytes 0-1023/4096` `Content-Range: bytes */4096` (dimensione sconosciuta)

Tabella A.18: Range Response Headers

Direttiva	Descrizione
`no-cache`	Richiede rivalidazione con server prima di usare cache
`no-store`	Non cachare mai questa richiesta/risposta
`max-age=<seconds>`	Età massima accettabile della cache
`max-stale[=<seconds>]`	Accetta cache stale fino a N secondi
`min-fresh=<seconds>`	Cache deve essere fresca per almeno N secondi
`no-transform`	Non trasformare il contenuto
`only-if-cached`	Usa solo cache, non contattare server

Tabella A.19: Cache-Control Request Directives

Direttiva	Descrizione
`public`	Può essere cachato da chiunque (CDN, proxy, browser)
`private`	Può essere cachato solo dal browser (non da proxy)
`no-cache`	Può cachare ma deve rivalidare prima di usare
`no-store`	Non cachare mai
`max-age=<seconds>`	Tempo massimo di validità in secondi
`s-maxage=<seconds>`	Come max-age ma solo per cache condivise (CDN, proxy)
`must-revalidate`	Cache stale DEVE essere rivalidata
`proxy-revalidate`	Come must-revalidate ma solo per cache condivise
`no-transform`	Cache non deve trasformare il contenuto
`immutable`	Contenuto non cambierà mai (performance optimization)
`stale-while-revalidate=<seconds>`	Permette cache stale mentre rivalidazione in background
`stale-if-error=<seconds>`	Permette cache stale se server in errore

Tabella A.20: Cache-Control Response Directives

Media Type	Utilizzo
`application/json`	JSON (default per REST API)
`application/xml`	XML
`application/x-www-form-urlencoded`	Form data (POST form)
`multipart/form-data`	Form con file upload
`text/plain`	Testo semplice
`text/html`	HTML
`text/csv`	CSV
`application/pdf`	PDF
`image/jpeg`	JPEG image
`image/png`	PNG image
`image/gif`	GIF image
`image/svg+xml`	SVG image
`application/octet-stream`	Binary data generico
`application/problem+json`	RFC 7807 Problem Details
`application/hal+json`	HAL (Hypertext Application Language)
`application/vnd.api+json`	JSON:API

Tabella A.21: Media Types Comuni

Code	Nome	Quando Usare
100	Continue	Upload grandi file (dopo request headers)
101	Switching Protocols	Upgrade a WebSocket

Tabella A.22: 1xx Status Codes

Code	Nome	Quando Usare
200	OK	GET successo, PUT/PATCH ritorna risorsa
201	Created	POST crea risorsa (includi Location header)
202	Accepted	Richiesta accettata ma processing asincrono
204	No Content	DELETE successo, PUT senza body response
206	Partial Content	Range request successo

Tabella A.23: 2xx Status Codes

Code	Nome	Quando Usare
301	Moved Permanently	Risorsa spostata permanentemente
302	Found	Redirect temporaneo (legacy, usa 307)
303	See Other	POST redirect a GET
304	Not Modified	Cache validation (con ETag/If-Modified-Since)
307	Temporary Redirect	Redirect temporaneo (mantiene method)
308	Permanent Redirect	Redirect permanente (mantiene method)

Tabella A.24: 3xx Status Codes

Code	Nome	Quando Usare
400	Bad Request	Request malformata, JSON invalido
401	Unauthorized	Autenticazione richiesta/fallita
403	Forbidden	Autenticato ma senza permessi
404	Not Found	Risorsa o endpoint non trovato
405	Method Not Allowed	HTTP method non supportato
406	Not Acceptable	Accept header non soddisfatto
409	Conflict	Conflitto (duplicato, versione)
410	Gone	Risorsa eliminata permanentemente
415	Unsupported Media Type	Content-Type non supportato
422	Unprocessable Entity	Errore validazione semantica
429	Too Many Requests	Rate limit superato

Tabella A.25: 4xx Status Codes

Code	Nome	Quando Usare
500	Internal Server Error	Errore server generico
501	Not Implemented	Funzionalità non implementata
502	Bad Gateway	Errore da upstream server
503	Service Unavailable	Server temporaneamente non disponibile
504	Gateway Timeout	Timeout da upstream server

Tabella A.26: 5xx Status Codes

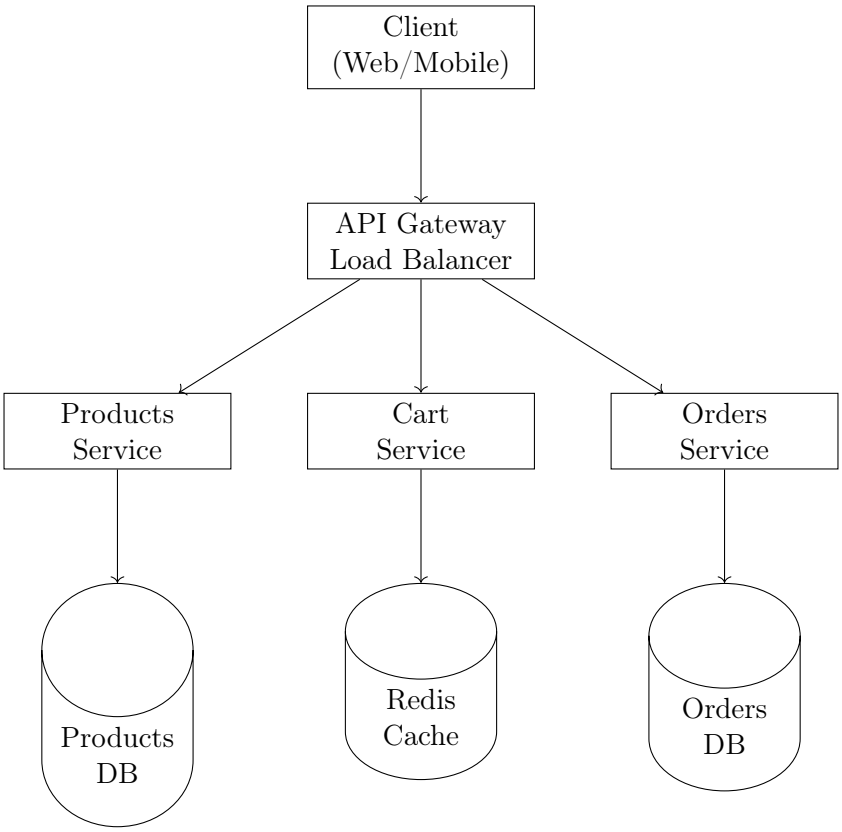
Appendice B

Appendice: Esempi di API Complete

Questa appendice presenta tre esempi completi di REST API: E-commerce, Social Media e Payments. Ogni esempio include architettura, endpoint, esempi curl e spec OpenAPI.

B.1 E-commerce API

B.1.1 Architettura



B.1.2 Endpoint Principali

B.1.3 Esempi cURL

Listing B.1: E-commerce API - Esempi cURL

```
#!/bin/bash

API_URL="https://api.ecommerce.example.com/v1"
```

Method	Endpoint	Descrizione
GET	/products	Lista prodotti con filtri
GET	/products/{id}	Dettagli prodotto
GET	/categories	Lista categorie
POST	/cart/items	Aggiungi item al carrello
GET	/cart	Visualizza carrello
DELETE	/cart/items/{id}	Rimuovi item da carrello
POST	/orders	Crea ordine da carrello
GET	/orders	Lista ordini utente
GET	/orders/{id}	Dettagli ordine
POST	/orders/{id}/cancel	Annulla ordine

Tabella B.1: E-commerce API Endpoints

```
TOKEN="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

echo "=== E-commerce API Examples ==="

# 1. Lista prodotti con filtri
echo -e "\n1. Lista prodotti categoria elettronica"
curl -s "$API_URL/products" \
  -H "Authorization: Bearer $TOKEN" \
  -G \
  --data-urlencode "category=electronics" \
  --data-urlencode "price_min=100" \
  --data-urlencode "price_max=1000" \
  --data-urlencode "in_stock=true" \
  --data-urlencode "sort=-price" \
  --data-urlencode "page=1" \
  --data-urlencode "per_page=10" \
  | jq .

# 2. Dettagli prodotto
echo -e "\n2. Dettagli prodotto specifico"
curl -s "$API_URL/products/prod_123" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 3. Aggiungi al carrello
echo -e "\n3. Aggiungi prodotto al carrello"
curl -s -X POST "$API_URL/cart/items" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "product_id": "prod_123",
    "quantity": 2,
    "variant_id": "var_456"
  }' \
  | jq .

# 4. Visualizza carrello
echo -e "\n4. Visualizza carrello corrente"
curl -s "$API_URL/cart" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 5. Aggiorna quantita item
```

```
echo -e "\n5. Aggiorna quantita item carrello"
curl -s -X PATCH "$API_URL/cart/items/item_789" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "quantity": 3
  }' \
  | jq .

# 6. Applica coupon
echo -e "\n6. Applica codice sconto"
curl -s -X POST "$API_URL/cart/coupons" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "code": "SAVE20"
  }' \
  | jq .

# 7. Crea ordine
echo -e "\n7. Crea ordine da carrello"
ORDER_RESPONSE=$(curl -s -X POST "$API_URL/orders" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: $(uuidgen)" \
  -d '{
    "shipping_address": {
      "name": "Mario Rossi",
      "street": "Via Roma 123",
      "city": "Milano",
      "postal_code": "20100",
      "country": "IT",
      "phone": "+39 333 1234567"
    },
    "billing_address": {
      "name": "Mario Rossi",
      "street": "Via Roma 123",
      "city": "Milano",
      "postal_code": "20100",
      "country": "IT"
    },
    "payment_method": "credit_card",
    "payment_details": {
      "card_token": "tok_visa_1234"
    }
  }')

echo "$ORDER_RESPONSE" | jq .
ORDER_ID=$(echo "$ORDER_RESPONSE" | jq -r '.id')

# 8. Traccia ordine
echo -e "\n8. Traccia ordine"
curl -s "$API_URL/orders/$ORDER_ID/tracking" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 9. Storia ordini
echo -e "\n9. Storia ordini utente"
```

```
curl -s "$API_URL/orders" \
  -H "Authorization: Bearer $TOKEN" \
  -G \
  --data-urlencode "status=delivered" \
  --data-urlencode "date_from=2023-01-01" \
  --data-urlencode "sort=-created_at" \
  | jq .

# 10. Richiedi reso
echo -e "\n10. Richiedi reso prodotto"
curl -s -X POST "$API_URL/orders/$ORDER_ID/returns" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "items": [
      {
        "order_item_id": "item_123",
        "quantity": 1,
        "reason": "defective",
        "description": "Schermo difettoso"
      }
    ]
  }' \
  | jq .
```

B.1.4 Response Esempi

Listing B.2: E-commerce - Product Response

```
{
  "id": "prod_123",
  "sku": "LAPTOP-DELL-XPS13",
  "name": "Dell XPS 13 Laptop",
  "description": "Laptop ultraleggero con display InfinityEdge",
  "category": {
    "id": "cat_electronics",
    "name": "Electronics",
    "slug": "electronics"
  },
  "brand": "Dell",
  "price": {
    "amount": 1299.99,
    "currency": "EUR",
    "formatted": "  1  .299,99"
  },
  "compare_at_price": {
    "amount": 1499.99,
    "currency": "EUR"
  },
  "discount_percentage": 13,
  "images": [
    {
      "url": "https://cdn.example.com/products/prod_123/img1.jpg",
      "alt": "Dell XPS 13 - Vista frontale",
      "position": 1
    }
  ],
```

```
  "variants": [
    {
      "id": "var_456",
      "name": "16GB RAM / 512GB SSD",
      "sku": "LAPTOP-DELL-XPS13-16-512",
      "price": 1299.99,
      "in_stock": true,
      "stock_quantity": 15
    }
  ],
  "attributes": {
    "color": "Silver",
    "weight": "1.2kg",
    "dimensions": "30.2 x 19.9 x 1.5 cm"
  },
  "rating": {
    "average": 4.7,
    "count": 342
  },
  "in_stock": true,
  "stock_quantity": 15,
  "shipping": {
    "free_shipping": true,
    "estimated_days": 2
  },
  "created_at": "2023-10-01T10:00:00Z",
  "updated_at": "2023-11-13T12:00:00Z",
  "_links": {
    "self": {
      "href": "https://api.example.com/products/prod_123"
    },
    "category": {
      "href": "https://api.example.com/categories/cat_electronics"
    },
    "reviews": {
      "href": "https://api.example.com/products/prod_123/reviews"
    }
  }
}
```

Listing B.3: E-commerce - Cart Response

```
{
  "id": "cart_789",
  "user_id": "user_456",
  "items": [
    {
      "id": "item_001",
      "product": {
        "id": "prod_123",
        "name": "Dell XPS 13 Laptop",
        "image": "https://cdn.example.com/products/prod_123/thumb.jpg"
      },
      "variant": {
        "id": "var_456",
        "name": "16GB RAM / 512GB SSD"
      },
      "quantity": 2,
      "unit_price": 1299.99,
```

```
      "total_price": 2599.98
    }
  ],
  "coupon": {
    "code": "SAVE20",
    "discount_amount": 520.00,
    "discount_type": "percentage",
    "discount_value": 20
  },
  "subtotal": 2599.98,
  "discount": 520.00,
  "shipping": 0.00,
  "tax": 415.99,
  "total": 2495.97,
  "currency": "EUR",
  "item_count": 2,
  "expires_at": "2023-11-14T12:00:00Z",
  "_links": {
    "checkout": {
      "href": "https://api.example.com/checkout"
    }
  }
}
```

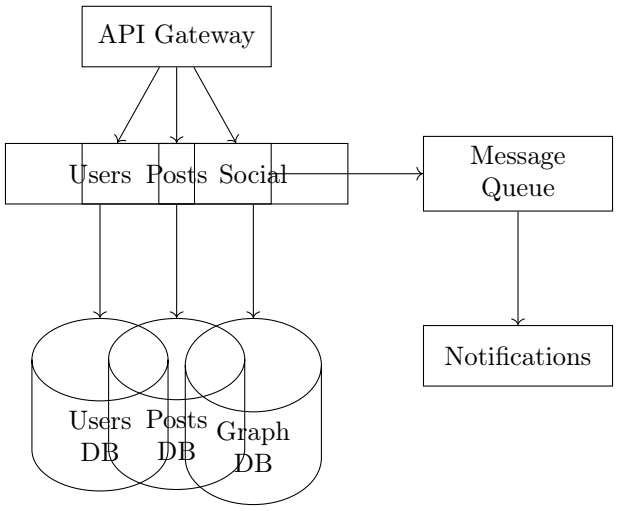
Listing B.4: E-commerce - Order Response

```
{
  "id": "ord_abc123",
  "order_number": "ORD-2023-001234",
  "status": "processing",
  "payment_status": "paid",
  "fulfillment_status": "unfulfilled",
  "customer": {
    "id": "user_456",
    "email": "mario@example.com",
    "name": "Mario Rossi"
  },
  "items": [
    {
      "id": "item_001",
      "product_id": "prod_123",
      "variant_id": "var_456",
      "name": "Dell XPS 13 Laptop - 16GB/512GB",
      "quantity": 2,
      "unit_price": 1299.99,
      "total": 2599.98
    }
  ],
  "shipping_address": {
    "name": "Mario Rossi",
    "street": "Via Roma 123",
    "city": "Milano",
    "postal_code": "20100",
    "country": "IT",
    "phone": "+39 333 1234567"
  },
  "billing_address": {
    "name": "Mario Rossi",
    "street": "Via Roma 123",
```

```
    "city": "Milano",
    "postal_code": "20100",
    "country": "IT"
  },
  "subtotal": 2599.98,
  "discount": 520.00,
  "shipping": 0.00,
  "tax": 415.99,
  "total": 2495.97,
  "currency": "EUR",
  "payment_method": "credit_card",
  "payment_details": {
    "last4": "1234",
    "brand": "Visa"
  },
  "tracking": {
    "carrier": "DHL",
    "tracking_number": "DHL123456789",
    "tracking_url": "https://dhl.com/track/DHL123456789"
  },
  "created_at": "2023-11-13T12:00:00Z",
  "updated_at": "2023-11-13T12:00:00Z",
  "_links": {
    "self": {
      "href": "https://api.example.com/orders/ord_abc123"
    },
    "tracking": {
      "href": "https://api.example.com/orders/ord_abc123/tracking"
    },
    "invoice": {
      "href": "https://api.example.com/orders/ord_abc123/invoice"
    },
    "cancel": {
      "href": "https://api.example.com/orders/ord_abc123/cancel",
      "method": "POST"
    }
  }
}
```

B.2 Social Media API

B.2.1 Architettura



B.2.2 Endpoint Principali

Method	Endpoint	Descrizione
GET	/users/{id}	Profilo utente
GET	/users/{id}/posts	Post dell'utente
POST	/users/{id}/follow	Segui utente
DELETE	/users/{id}/follow	Smetti di seguire
POST	/posts	Crea post
GET	/feed	Feed personalizzato
POST	/posts/{id}/like	Like a post
POST	/posts/{id}/comments	Commenta post
GET	/notifications	Notifiche utente
POST	/messages	Invia messaggio

Tabella B.2: Social Media API Endpoints

B.2.3 Esempi cURL

Listing B.5: Social Media API - Esempi cURL

```
#!/bin/bash

API_URL="https://api.social.example.com/v1"
TOKEN="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

echo "=== Social Media API Examples ==="

# 1. Ottieni profilo utente
echo -e "\n1. Profilo utente"
curl -s "$API_URL/users/me" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 2. Aggiorna profilo
echo -e "\n2. Aggiorna bio profilo"
curl -s -X PATCH "$API_URL/users/me" \
```

```
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "bio": "Software developer, coffee enthusiast",
    "website": "https://example.com"
  }' \
  | jq .

# 3. Upload foto profilo
echo -e "\n3. Upload foto profilo"
curl -s -X POST "$API_URL/users/me/avatar" \
  -H "Authorization: Bearer $TOKEN" \
  -F "file=@avatar.jpg" \
  | jq .

# 4. Crea post
echo -e "\n4. Crea nuovo post"
POST_RESPONSE=$(curl -s -X POST "$API_URL/posts" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "content": "Just deployed my new REST API!     ",
    "visibility": "public",
    "media": [
      {
        "type": "image",
        "url": "https://cdn.example.com/images/screenshot.jpg"
      }
    ],
    "tags": ["#api", "#development"]
  }')

echo "$POST_RESPONSE" | jq .
POST_ID=$(echo "$POST_RESPONSE" | jq -r '.id')

# 5. Feed personalizzato
echo -e "\n5. Feed personalizzato"
curl -s "$API_URL/feed" \
  -H "Authorization: Bearer $TOKEN" \
  -G \
  --data-urlencode "page=1" \
  --data-urlencode "per_page=20" \
  --data-urlencode "filter=following" \
  | jq .

# 6. Like a post
echo -e "\n6. Like post"
curl -s -X POST "$API_URL/posts/$POST_ID/likes" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 7. Commenta post
echo -e "\n7. Commenta post"
curl -s -X POST "$API_URL/posts/$POST_ID/comments" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "content": "Great work!     ",
```

```
    "mentions": ["@user123"]
  }' \
  | jq .

# 8. Segui utente
echo -e "\n8. Segui utente"
curl -s -X POST "$API_URL/users/user_789/follow" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 9. Lista follower
echo -e "\n9. Lista follower"
curl -s "$API_URL/users/me/followers" \
  -H "Authorization: Bearer $TOKEN" \
  -G \
  --data-urlencode "page=1" \
  --data-urlencode "per_page=50" \
  | jq .

# 10. Cerca utenti
echo -e "\n10. Cerca utenti"
curl -s "$API_URL/search/users" \
  -H "Authorization: Bearer $TOKEN" \
  -G \
  --data-urlencode "q=mario" \
  --data-urlencode "limit=10" \
  | jq .

# 11. Notifiche
echo -e "\n11. Notifiche non lette"
curl -s "$API_URL/notifications" \
  -H "Authorization: Bearer $TOKEN" \
  -G \
  --data-urlencode "unread=true" \
  | jq .

# 12. Marca notifiche come lette
echo -e "\n12. Marca tutte notifiche lette"
curl -s -X POST "$API_URL/notifications/mark-read" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 13. Invia messaggio diretto
echo -e "\n13. Invia messaggio diretto"
curl -s -X POST "$API_URL/messages" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "recipient_id": "user_789",
    "content": "Ciao! Come va?",
    "attachments": []
  }' \
  | jq .

# 14. Conversazioni
echo -e "\n14. Lista conversazioni"
curl -s "$API_URL/conversations" \
  -H "Authorization: Bearer $TOKEN" \
```

```
  | jq .
```

B.2.4 Response Esempi

Listing B.6: Social Media - User Profile

```
{
  "id": "user_456",
  "username": "mario_rossi",
  "display_name": "Mario Rossi",
  "bio": "Software developer, coffee enthusiast    ",
  "avatar_url": "https://cdn.example.com/avatars/user_456.jpg",
  "cover_url": "https://cdn.example.com/covers/user_456.jpg",
  "website": "https://mariorossi.com",
  "location": "Milano, Italy",
  "verified": false,
  "stats": {
    "followers": 1234,
    "following": 567,
    "posts": 89
  },
  "joined_at": "2022-01-15T10:00:00Z",
  "is_following": false,
  "is_followed_by": false,
  "_links": {
    "self": {
      "href": "https://api.example.com/users/user_456"
    },
    "posts": {
      "href": "https://api.example.com/users/user_456/posts"
    },
    "followers": {
      "href": "https://api.example.com/users/user_456/followers"
    },
    "following": {
      "href": "https://api.example.com/users/user_456/following"
    }
  }
}
```

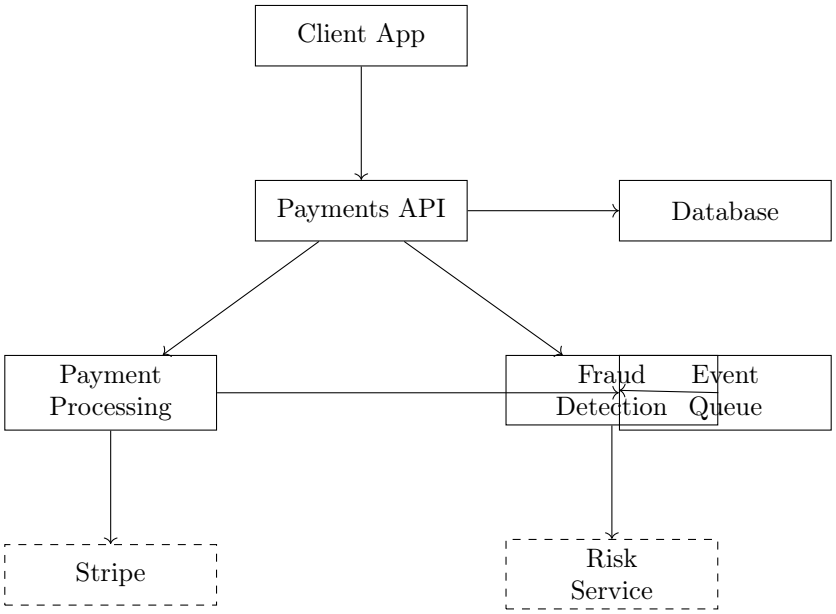
Listing B.7: Social Media - Post

```
{
  "id": "post_123",
  "author": {
    "id": "user_456",
    "username": "mario_rossi",
    "display_name": "Mario Rossi",
    "avatar_url": "https://cdn.example.com/avatars/user_456.jpg",
    "verified": false
  },
  "content": "Just deployed my new REST API!     ",
  "media": [
    {
      "type": "image",
      "url": "https://cdn.example.com/posts/post_123/img1.jpg",
      "width": 1200,
      "height": 800
```

```
    }
  ],
  "tags": ["#api", "#development"],
  "mentions": [],
  "visibility": "public",
  "stats": {
    "likes": 42,
    "comments": 7,
    "shares": 3,
    "views": 234
  },
  "is_liked": false,
  "is_bookmarked": false,
  "created_at": "2023-11-13T12:00:00Z",
  "edited_at": null,
  "_links": {
    "self": {
      "href": "https://api.example.com/posts/post_123"
    },
    "comments": {
      "href": "https://api.example.com/posts/post_123/comments"
    },
    "likes": {
      "href": "https://api.example.com/posts/post_123/likes"
    }
  }
}
```

B.3 Payments API

B.3.1 Architettura



B.3.2 Endpoint Principali

B.3.3 Esempi cURL

Method	Endpoint	Descrizione
POST	/payment-methods	Aggiungi metodo pagamento
GET	/payment-methods	Lista metodi pagamento
DELETE	/payment-methods/{id}	Elimina metodo
POST	/payments	Crea pagamento
GET	/payments/{id}	Dettagli pagamento
POST	/payments/{id}/refund	Rimborso
POST	/payouts	Crea payout
GET	/balance	Saldo account
GET	/transactions	Storia transazioni

Tabella B.3: Payments API Endpoints

Listing B.8: Payments API - Esempi cURL

```bash
#!/bin/bash

API_URL="https://api.payments.example.com/v1"
TOKEN="sk_live_abc123..."

echo "=== Payments API Examples ==="

# 1. Aggiungi carta di credito
echo -e "\n1. Aggiungi metodo di pagamento"
CARD_RESPONSE=$(curl -s -X POST "$API_URL/payment-methods" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "type": "card",
    "card": {
      "number": "4242424242424242",
      "exp_month": 12,
      "exp_year": 2025,
      "cvc": "123"
    },
    "billing_details": {
      "name": "Mario Rossi",
      "email": "mario@example.com",
      "address": {
        "line1": "Via Roma 123",
        "city": "Milano",
        "postal_code": "20100",
        "country": "IT"
      }
    }
  }')

echo "$CARD_RESPONSE" | jq .
PAYMENT_METHOD_ID=$(echo "$CARD_RESPONSE" | jq -r '.id')

# 2. Lista metodi di pagamento
echo -e "\n2. Lista metodi di pagamento"
curl -s "$API_URL/payment-methods" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 3. Crea pagamento
echo -e "\n3. Crea pagamento"
```

```
PAYMENT_RESPONSE=$(curl -s -X POST "$API_URL/payments" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: $(uuidgen)" \
  -d "{
    \"amount\": 10000,
    \"currency\": \"eur\",
    \"payment_method\": \"$PAYMENT_METHOD_ID\",
    \"description\": \"Ordine #ORD-2023-001234\",
    \"metadata\": {
      \"order_id\": \"ord_abc123\",
      \"customer_id\": \"cus_456\"
    },
    \"receipt_email\": \"mario@example.com\"
  }")

echo "$PAYMENT_RESPONSE" | jq .
PAYMENT_ID=$(echo "$PAYMENT_RESPONSE" | jq -r '.id')

# 4. Conferma pagamento (per 3D Secure)
echo -e "\n4. Conferma pagamento"
curl -s -X POST "$API_URL/payments/$PAYMENT_ID/confirm" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "return_url": "https://example.com/payment/complete"
  }' \
  | jq .

# 5. Dettagli pagamento
echo -e "\n5. Dettagli pagamento"
curl -s "$API_URL/payments/$PAYMENT_ID" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 6. Lista pagamenti
echo -e "\n6. Lista pagamenti"
curl -s "$API_URL/payments" \
  -H "Authorization: Bearer $TOKEN" \
  -G \
  --data-urlencode "limit=10" \
  --data-urlencode "status=succeeded" \
  --data-urlencode "created[gte]=2023-11-01" \
  | jq .

# 7. Crea rimborso
echo -e "\n7. Crea rimborso parziale"
curl -s -X POST "$API_URL/payments/$PAYMENT_ID/refunds" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: $(uuidgen)" \
  -d '{
    "amount": 5000,
    "reason": "requested_by_customer",
    "metadata": {
      "reason_details": "Prodotto danneggiato"
    }
  }' \
```

```
  | jq .

# 8. Saldo account
echo -e "\n8. Saldo account"
curl -s "$API_URL/balance" \
  -H "Authorization: Bearer $TOKEN" \
  | jq .

# 9. Crea payout
echo -e "\n9. Crea payout"
curl -s -X POST "$API_URL/payouts" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: $(uuidgen)" \
  -d '{
    "amount": 50000,
    "currency": "eur",
    "method": "bank_account",
    "description": "Payout mensile Novembre 2023"
  }' \
  | jq .

# 10. Transazioni
echo -e "\n10. Storia transazioni"
curl -s "$API_URL/transactions" \
  -H "Authorization: Bearer $TOKEN" \
  -G \
  --data-urlencode "limit=20" \
  --data-urlencode "type=payment" \
  | jq .

# 11. Crea subscription
echo -e "\n11. Crea abbonamento"
curl -s -X POST "$API_URL/subscriptions" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d "{
    \"customer\": \"cus_456\",
    \"items\": [
      {
        \"price\": \"price_pro_monthly\",
        \"quantity\": 1
      }
    ],
    \"payment_method\": \"$PAYMENT_METHOD_ID\",
    \"trial_period_days\": 14
  }" \
  | jq .

# 12. Webhook endpoint test
echo -e "\n12. Test webhook signature"
curl -s -X POST "$API_URL/webhooks/test" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "webhook_id": "wh_abc123",
    "event_type": "payment.succeeded"
  }' \
```

```
  | jq .
```

B.3.4 Response Esempi

Listing B.9: Payments - Payment Response

```
{
  "id": "pay_abc123",
  "object": "payment",
  "amount": 10000,
  "amount_received": 10000,
  "currency": "eur",
  "status": "succeeded",
  "payment_method": {
    "id": "pm_xyz789",
    "type": "card",
    "card": {
      "brand": "visa",
      "last4": "4242",
      "exp_month": 12,
      "exp_year": 2025,
      "fingerprint": "abc123"
    },
    "billing_details": {
      "name": "Mario Rossi",
      "email": "mario@example.com",
      "address": {
        "line1": "Via Roma 123",
        "city": "Milano",
        "postal_code": "20100",
        "country": "IT"
      }
    }
  },
  "description": "Ordine #ORD-2023-001234",
  "receipt_email": "mario@example.com",
  "receipt_url": "https://pay.example.com/receipts/pay_abc123",
  "metadata": {
    "order_id": "ord_abc123",
    "customer_id": "cus_456"
  },
  "fees": [
    {
      "type": "stripe_fee",
      "amount": 320,
      "currency": "eur",
      "description": "Stripe processing fee"
    }
  ],
  "net": 9680,
  "refunded": false,
  "refunds": {
    "total_count": 0,
    "total_amount": 0,
    "data": []
  },
  "risk_score": 15,
```

```
  "risk_level": "normal",
  "created_at": "2023-11-13T12:00:00Z",
  "updated_at": "2023-11-13T12:00:05Z",
  "_links": {
    "self": {
      "href": "https://api.example.com/payments/pay_abc123"
    },
    "refund": {
      "href": "https://api.example.com/payments/pay_abc123/refunds",
      "method": "POST"
    },
    "receipt": {
      "href": "https://pay.example.com/receipts/pay_abc123"
    }
  }
}
```

Listing B.10: Payments - Webhook Event

```
{
  "id": "evt_abc123",
  "type": "payment.succeeded",
  "created_at": "2023-11-13T12:00:05Z",
  "data": {
    "object": {
      "id": "pay_abc123",
      "amount": 10000,
      "currency": "eur",
      "status": "succeeded",
      "metadata": {
        "order_id": "ord_abc123"
      }
    },
    "previous_attributes": {
      "status": "processing"
    }
  },
  "livemode": true,
  "pending_webhooks": 1,
  "request": {
    "id": "req_xyz789",
    "idempotency_key": "a7b3c9d2-1234-5678-9abc-def012345678"
  }
}
```

B.4 Postman Collections

B.4.1 E-commerce Collection

Listing B.11: Postman Collection - E-commerce

```
{
  "info": {
    "name": "E-commerce API",
    "schema": "https://schema.getpostman.com/json/collection/v2.1.0/"
  },
  "auth": {
```

```
    "type": "bearer",
    "bearer": [
      {"key": "token", "value": "{{access_token}}"}
    ]
  },
  "variable": [
    {"key": "base_url", "value": "https://api.ecommerce.example.com/v1
       "},
    {"key": "access_token", "value": ""},
    {"key": "product_id", "value": ""},
    {"key": "order_id", "value": ""}
  ],
  "item": [
    {
      "name": "Products",
      "item": [
        {
          "name": "List Products",
          "request": {
            "method": "GET",
            "url": {
              "raw": "{{base_url}}/products?category=electronics&page
                 =1",
              "query": [
                {"key": "category", "value": "electronics"},
                {"key": "page", "value": "1"}
              ]
            }
          }
        },
        {
          "name": "Get Product",
          "request": {
            "method": "GET",
            "url": "{{base_url}}/products/{{product_id}}"
          }
        }
      ]
    },
    {
      "name": "Cart",
      "item": [
        {
          "name": "Add to Cart",
          "event": [
            {
              "listen": "test",
              "script": {
                "exec": [
                  "pm.test(\"Status is 200\", () => {",
                  "  pm.response.to.have.status(200);",
                  "});"
                ]
              }
            }
          ],
          "request": {
            "method": "POST",
```

```
            "url": "{{base_url}}/cart/items",
            "body": {
              "mode": "raw",
              "raw": "{\n  \"product_id\": \"{{product_id}}\",\n  \"
                 quantity\": 2\n}"
            }
          }
        }
      ]
    }
  ]
}
```

B.5 Riepilogo

Questa appendice ha fornito tre esempi completi di REST API:

1. **E-commerce API**: Gestione prodotti, carrello, ordini
2. **Social Media API**: Profili, post, feed, notifiche
3. **Payments API**: Pagamenti, rimborsi, abbonamenti

Ogni esempio include:

- Architettura sistema
- Endpoint completi
- Esempi cURL pratici
- Response dettagliate
- Postman collections

Usa questi esempi come riferimento per progettare le tue REST API.