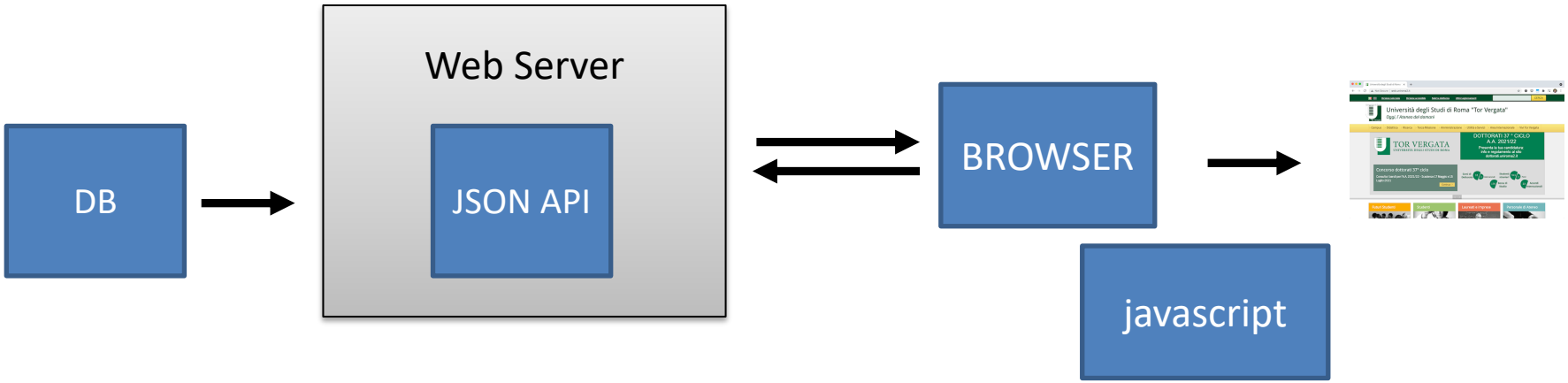
Api JSON

Architettura



REST

Representation State Transfer (REST)

- **“Uno stile architetturale ancora più vincolato** pensato per applicazioni Web **affidabili**.

- Il Web REST è un **sottoinsieme del WWW basato su HTTP**, in cui gli agenti forniscono una **semantica di interfaccia uniforme** — in sostanza: **creazione, lettura, aggiornamento ed eliminazione** delle risorse (CRUD) — invece di interfacce arbitrarie o specifiche per ogni applicazione.

- Le risorse vengono **manipolate solo tramite lo scambio di rappresentazioni**.

- Inoltre, le interazioni REST sono **stateless (senza stato)**, nel senso che **il significato di un messaggio non dipende dallo stato della conversazione**.”

- Proposto da Roy Fielding nella tesi di dottorato [2]

[1] https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest
[2] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Che cos'è REST?

- Architettura per applicazioni web affidabili
- Basato su principi del Web: HTTP, URI, rappresentazioni
- Obiettivo: semplificare l'interazione client-server con **interfacce uniformi**

📌 *REST è una guida architetturale, non uno standard.*

Principi Fondamentali di REST

- **Identificazione delle risorse** (mediante URI)
- **Interfaccia uniforme** (CRUD con metodi HTTP)
- **Manipolazione tramite rappresentazioni**
- **Comunicazione senza stato (Stateless)**
- **Risorse autodescrittive**
- **Navigabilità tramite link (HATEOAS)**

Identificazione delle risorse

- **Definizione:** Una **risorsa** è qualsiasi entità significativa per il sistema.

- **Esempi di URI:**
  - /clients/1234
  - /orders/2019/98765
  - /products?color=red

 *Le URI devono essere leggibili e semantiche.*

Operazioni CRUD

- Acronimo per:
  create, read (aka retrieve), update, and delete

- Operazioni di base che posso fare su una risorsa
  - **C**reate (creare una risorsa)
  - **R**ead o **R**etrieve (leggere una risorsa)
  - **U**pdate (aggiornare una risorsa)
  - **D**elete (eliminare una risorsa)

REST e CRUD

Metodo HTTP	Operazione CRUD	Descrizione
POST	Create	Crea una nuova risorsa
GET	Read	Ottiene una risorsa esistente
PUT	Update	Aggiorna una risorsa o ne modifica lo stato
DELETE	Delete	Elimina una risorsa

Esempio

Risorsa	**GET** read	**POST** create	**PUT** update	**DELETE**
/books	Ritorna una lista di libri	Crea un nuovo libro	Aggiorna i dati di tutti i libri	Elimina tutti i libri
/books/145	Ritorna uno specifico libro	metodo non consentito (405)	Aggiorna uno specifico libro	Elimina uno specifico libro

Rappresentazione di Risorse

- Le risorse sono codificate ed inviate al client
  - al suo interno il server le memorizza come vuole

- Caratteristiche della rappresentazione:
  - Understandability
  - Completeness
  - Linkablity

- Formati tipici: JSON e XML

Specificare la rappresentazione

- Client e server possono specificare il formato per la risorsa
  - dicono il MIME Type

- Client:
  - **Accept**

```
GET /clienti/1234
HTTP/1.1
Host: www.myapp.com
Accept: application/vnd.myapp.cliente+xml
```

- Server:
  - **Content-Type**

Entry Point

- Unico punto di ingresso API (es. /api/v1/)

- Fornisce:
  - Info versioni
  - Liste di collezioni
  - Risorse principali

Evitare strutture URL ambigue o opache

Struttura delle URL

URL	Description
/api	The API entry point
/api/:coll	A top-level collection named “coll”
/api/:coll/:id	The resource “id” inside collection “coll”
/api/:coll/:id/:subcoll	Sub-collection “subcoll” under resource “id”
/api/:coll/:id/:subcoll/:subid	The resource “subid” inside “subcoll”

```
/endpoint
   /collection1
      /resource1
      /resource2
      /resource3
   /collection2
      /resource1
      /resource2
…
```

Comunicazione Stateless

- **comunicazione stateless:** ciascuna richiesta non ha alcuna relazione con le richieste precedenti e successive
  - La responsabilità della gestione dello stato dell'applicazione non deve essere conferita al server, ma rientra nei compiti del client.
  - La principale ragione di questa scelta è la scalabilità: mantenere lo stato di una sessione ha un costo in termini di risorse sul server e all'aumentare del numero di client tale costo può diventare insostenibile.
  - Inoltre, con una comunicazione senza stato è possibile creare cluster di server che possono rispondere ai client senza vincoli sulla sessione corrente, ottimizzando le prestazioni globali dell'applicazione.

Stateless!!!

- Lo stato va manteniuto nel client
  - Il server per rispondere non deve ricordare una richiesta precedente

- Esempio
  - paging:
    - https://reqres.in/api/users?page=1
    - https://reqres.in/api/users?page=nextpage
  - login
    - ogni richiesta è autenticata singolarmente

REST e Status Code

- 200 – OK – Tutto bene
- 201 – OK – E' stata creata una nuova risorsa
- 204 – OK – La risorsa è stata cancellata con successo
- 304 – Not modified – I dati non sono cambiati. Il cliente può utilizzare i dati nella cache
- 400 – Bad Request – Richiesta non valida. L'errore esatto dovrebbe essere spiegato nel payload dell' errore (di cui ne parleremo a breve). Per esempio. "Il JSON non è valido"
- 401 – Unauthorized – La richiesta richiede una autenticazione dell'utente
- 403 – Forbidden – Il server ha capito la richiesta, ma in base ai diritti del richiedente l'accesso non è consentito.
- 404 – Not Found – Non vi è alcuna risorsa dietro l'URI richiesto.
- 422 – Unprocessable Entity – deve essere usato se il server non può elaborare il enitity, ad esempio se un'immagine non può essere formattata o campi obbligatori sono mancanti nel payload.
- 500 – Internal Server Error – gli sviluppatori di API dovrebbero evitare questo errore. Se si verifica un errore globale dell'applicazione, lo stacktrace deve esere loggato e non inviato nella risposta all'utente.

REST vs Web Classico

- Nel mondo WEB viene utilizzato il metodo GET per eseguire qualsiasi tipo di interazione con il server.

```
GET /addCustomer?name=Rossi
```

GET deve solo **recuperare** dati, non modificarli!!

- Per creare una risorsa in REST uso la POST

```
POST /customers
Body: { "name": "Rossi" }
```

Web Operations

`products` `users` `orders`

`http://my-url/addNewProduct`

`/getProduct`

`/updateProduct`

`/deleteProduct`

`/getProductbyOrder`

`/getOrderbyUser`

REST Operations

`/addNewProduct`	`POST /products`
`/getProduct`	`GET /products/3`
`/updateProduct`	`PUT /products/3`
	`PATCH /products/3`
`/deleteProduct`	`DELETE /products/3`
`/getProductbyOrder`	`GET /orders/4/products`
`/getOrderbyUser`	`GET /users/9/orders`

JSON formatting


```
{
  "id": 1,
  "name": "cerulean",
  "year": 2000,
  "color": "#98B2D1",
  "pantone_value": "15-4020"
}
```

JSON formatting

JSEND
https://github.com/omniti-labs/jsend

```
{
  "id": 1,
  "name": "cerulean",
  "year": 2000,
  "color": "#98B2D1",
  "pantone_value": "15-4020"
}
```

```
{
  "status": "success",
  "data": {
    "id": 1,
    "name": "cerulean",
    "year": 2000,
    "color": "#98B2D1",
    "pantone_value": "15-4020"
  }
}
```

JSON formatting

JSEND
https://github.com/omniti-labs/jsend

```
{
  "id": 1,
  "name": "cerulean",
  "year": 2000,
  "color": "#98B2D1",
  "pantone_value": "15-4020"
}
```

```
{
  "status": "success",
  "data": {
    "id": 1,
    "name": "cerulean",
    "year": 2000,
    "color": "#98B2D1",
    "pantone_value": "15-4020"
  }
}
```

1. JSON API - JSON API covers creating and updating resources as well, not just responses.
2. JSend - Simple and probably what you are already doing.
3. OData JSON Protocol - Very complicated.

EXPRESS REST API

GET

```
app.get('/api/v1/products', (req, res) => {
  res.status(200).json({
    status: 'success',
    data: {
      products: products,
    },
  });
});
```

GET

```
app.get('/api/v1/products/:id', (req, res) => {
  console.log(req.params);

  const prod = products.find((el) => el.id == req.params.id);
  console.log(prod);
  if (prod == undefined) {
    res.status(404).json({
      status: 'fail',
      message: 'ID non trovato',
    });
  } else {
    res.status(200).json({
      status: 'success',
      data: {
        product: prod,
      },
    });
  }
});
```

POST

```
app.post('/api/v1/products', (req, res) => {
  const newId = products[products.length - 1].id + 1;
  const newProd = Object.assign({ id: newId }, req.body);

  products.push(newProd);
  res.status(201).json({
    status: 'success',
    data: { product: newProd },
  });
});
```

PUT/PATCH

```
app.patch('/api/v1/products/:id', (req, res) => {
  const prod = products.find((el) => el.id == req.params.id);
  if (prod == undefined) {
    res.status(404).json({
      status: 'fail',
      message: 'ID non trovato',
    });
  } else {
    // Update ....

    res.status(200).json({
      status: 'success',
      data: {
        product: prod,
      },
    });
  }
});
```

DELETE

```
app.delete('/api/v1/products/:id', (req, res) => {
  const prod = products.find((el) => el.id == req.params.id);
  if (prod == undefined) {
    res.status(404).json({
      status: 'fail',
      message: 'ID non trovato',
    });
  } else {
    // Delete ....
    res.status(204).json({
      status: 'success',
      data: null,
    });
  }
});
```