REST

REST, acronimo di Representational State Transfer, è uno stile architetturale per la progettazione di servizi web che utilizzano il protocollo HTTP per trasmettere dati. È stato introdotto per la prima volta da Roy Fielding nella sua tesi di dottorato nel 2000 e da allora è diventato uno dei principali approcci per la creazione di API web scalabili, interoperabili e basate su risorse.

Ecco alcune caratteristiche chiave di REST:

1. **Architettura basata su risorse**:
   In REST, i dati sono organizzati come risorse, che possono essere qualsiasi cosa, da un singolo oggetto a un insieme di dati correlati. Ogni risorsa è identificata da un URI (Uniform Resource Identifier).
2. **Operazioni CRUD**:
   Le operazioni CRUD (Create, Read, Update, Delete) sono eseguite sulle risorse utilizzando i metodi HTTP corrispondenti: POST, GET, PUT/PATCH, DELETE rispettivamente. Ad esempio, per recuperare una risorsa, si utilizza una richiesta HTTP GET al relativo URI.
3. **Stateless**:
   REST è un'architettura stateless, il che significa che ogni richiesta HTTP contiene tutte le informazioni necessarie per elaborare la richiesta. Non viene mantenuto alcuno stato di sessione sul server.
4. **Interfaccia uniforme**:
   REST utilizza un'interfaccia uniforme per le interazioni tra client e server, che include l'uso di URI per identificare le risorse, i metodi HTTP per specificare l'azione desiderata, i formati di rappresentazione dei dati (come JSON o XML) per trasferire informazioni e i codici di stato HTTP per indicare l'esito dell'operazione.
5. **Scalabilità e modularità**:
   REST promuove la scalabilità e la modularità dei servizi web, consentendo ai client di accedere e manipolare risorse tramite interfacce standardizzate.

6. **Cacheability**:
   Le risposte REST possono essere cacheate per migliorare le prestazioni e ridurre il carico sui server.
7. **Layered system**:
   Le architetture REST sono organizzate in strati, consentendo l'aggiunta di nuovi servizi o modifiche ai servizi esistenti senza influenzare il funzionamento degli altri strati.

RESTful API:

Una RESTful API è un'interfaccia di programmazione delle applicazioni (API) che segue i principi di REST (Representational State Transfer). Questo significa che l'API rispetta gli standard e le convenzioni di REST per la progettazione e l'implementazione delle sue risorse e delle relative interazioni.

URI: Uniform Resource Identifier:

Un URI è una stringa di caratteri che identifica univocamente una risorsa su internet. Nella progettazione di una RESTful API, ogni risorsa è identificata da un URI, che può includere il nome del dominio, il percorso della risorsa e, eventualmente, parametri di query.

Cosa vuol dire basato su risorse?

Essere "basati su risorse" significa che i servizi web seguono il concetto di risorse come l'unità principale di interazione. Le risorse possono rappresentare qualsiasi cosa, da oggetti del mondo reale a concetti astratti, e sono identificate da URI univoci. Le operazioni CRUD (Create, Read, Update, Delete) vengono eseguite sulle risorse utilizzando i metodi HTTP appropriati.

Rappresentazioni:

Le risorse in una RESTful API possono avere diverse rappresentazioni, che possono essere trasferite tra client e server. Ad esempio, una risorsa può essere rappresentata come JSON, XML, HTML, testo semplice, ecc. I client possono negoziare il tipo di rappresentazione preferito utilizzando gli header HTTP Accept.

Vincoli REST:

I vincoli REST sono un insieme di regole e principi che devono essere seguiti per progettare un'API RESTful. Alcuni dei principali vincoli includono:

- Client-server separation
- Stateless
- Cacheability
- Layered system
- Uniform interface
- Code on demand (opzionale)

Interfaccia uniforme:

L'interfaccia uniforme è uno dei vincoli chiave di REST. Significa che le interfacce per interagire con le risorse sono standardizzate e prevedibili. Ciò include l'uso di URI per identificare le risorse, i metodi HTTP per specificare l'azione desiderata (GET, POST, PUT, DELETE), i formati di rappresentazione dei dati (JSON, XML) e i codici di stato HTTP per indicare l'esito dell'operazione.

HATEOAS (Hypertext As The Engine Of Application State):

HATEOAS è un principio che suggerisce di includere collegamenti ipertestuali (link) nelle rappresentazioni delle risorse, in modo che i client possano scoprire e navigare le varie operazioni disponibili. In pratica, questo significa che i client possono esplorare dinamicamente l'API seguendo i link, senza dover dipendere da conoscenze pregresse sulle operazioni possibili.

Implementazioni di HATEOAS:

L'implementazione di HATEOAS in una RESTful API implica l'inclusione di collegamenti ipertestuali (link) nelle rappresentazioni delle risorse. Questi link forniscono ai client informazioni su quali altre risorse possono essere esplorate o manipolate e quali azioni possono essere eseguite. Ad esempio, un link potrebbe indicare al client di seguire un percorso specifico per ottenere risorse correlate o eseguire determinate operazioni su una risorsa.

Stateless:

Lo stato è mantenuto interamente sul client nella comunicazione RESTful. Questo significa che ogni richiesta HTTP contiene tutte le informazioni necessarie per elaborare la richiesta, e non viene mantenuto alcuno stato di sessione sul server. Questo favorisce la scalabilità orizzontale e semplifica il bilanciamento del carico del server.

Cacheable:

Le risposte fornite da una RESTful API possono essere cacheate per migliorare le prestazioni e ridurre il carico sui server. Le risorse che cambiano raramente o che richiedono risorse significative per essere generate possono essere contrassegnate come cacheable, consentendo ai client di memorizzare localmente le risposte e riutilizzarle per richieste successive.

Client-server:

Il principio della separazione tra client e server è fondamentale in REST. Il client e il server sono due entità separate e indipendenti che comunicano tra loro attraverso l'uso di richieste HTTP e risposte. Questa separazione consente una maggiore scalabilità e una maggiore modularità del sistema.

Sistema stratificato:

REST favorisce l'architettura stratificata, in cui i componenti del sistema sono organizzati in strati. Ogni strato ha una funzione specifica e comunica solo con gli strati adiacenti. Questo consente una maggiore flessibilità e la possibilità di aggiungere, rimuovere o modificare i componenti del sistema senza influenzare gli altri strati.

Code on demand:

Il vincolo "Code on demand" è un vincolo opzionale di REST che consente al server di trasferire codice eseguibile al client. Ad esempio, il server potrebbe restituire codice JavaScript che il client può eseguire per personalizzare o estendere il comportamento dell'applicazione. Tuttavia, questo vincolo è raramente utilizzato in pratica e non è essenziale per rispettare i principi REST.

Usare la semantica dei verbi HTTP:

Nel design di un'API REST, è importante utilizzare i verbi HTTP in modo semantico per definire le azioni che si desidera eseguire sulle risorse.

- **GET**: Utilizzato per recuperare informazioni esistenti o risorse.
- **POST**: Utilizzato per creare nuove risorse.
- **PUT**: Utilizzato per aggiornare completamente una risorsa esistente.
- **PATCH**: Utilizzato per aggiornare parzialmente una risorsa esistente.
- **DELETE**: Utilizzato per rimuovere una risorsa esistente.

Status Code più comuni:

Alcuni dei codici di stato HTTP più comuni utilizzati nelle API REST includono:

- **200 OK**: Indica che la richiesta è stata elaborata con successo.
- **201 Created**: Indica che la risorsa è stata creata con successo.
- **400 Bad Request**: Indica un errore nella richiesta del client.
- **404 Not Found**: Indica che la risorsa richiesta non è stata trovata.
- **500 Internal Server Error**: Indica un errore interno del server.

Design delle risorse:

Le risorse dovrebbero essere progettate per essere rappresentative dei concetti del dominio dell'applicazione. Ogni risorsa dovrebbe avere un URI univoco e rappresentare una singola entità o un insieme di dati correlati.

Path alle risorse:

I percorsi delle risorse dovrebbero essere intuitivi e riflettere la struttura gerarchica delle risorse nell'applicazione. Ad esempio:

- `/users`: Risorsa che rappresenta gli utenti.
- `/users/{id}`: Risorsa che rappresenta un utente specifico.

Range, ricerche, filtri, sort e paginazione:

Le API REST possono supportare operazioni di range, ricerche, filtri, ordinamento e paginazione per gestire grandi volumi di dati. Questo può essere realizzato

utilizzando parametri di query nell'URI delle richieste HTTP.

CORS (Cross-Origin Resource Sharing):

CORS è un meccanismo che consente a una pagina web di rendere richieste a un'altra risorsa appartenente a un dominio diverso. Le API REST dovrebbero supportare CORS per consentire l'accesso da parte di applicazioni client esterne.

OpenAPI/Swagger:

OpenAPI (noto anche come Swagger) è uno standard per la descrizione delle API RESTful utilizzato per definire la struttura, la documentazione e il comportamento delle API. Un documento OpenAPI definisce i percorsi, i parametri, i metodi e altri dettagli delle API REST.

Sviluppare una rest API

1. Inizializzazione del progetto:

- Assicurati di avere Node.js installato sul tuo sistema.
- Crea una nuova directory per il progetto e inizializza un nuovo progetto Node.js eseguendo `npm init` e seguendo le istruzioni per creare il file `package.json`.
- Installa Express eseguendo `npm install express`.

2. Creazione del server Express:

- Crea un file `server.js` e inizia a definire il server Express:

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

// Definizione degli endpoint
```

```
app.get('/api/posts', (req, res) => {
  res.json({ message: 'GET request to /api/posts' });
});

// Avvio del server
app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

3. Definizione degli endpoint:

- Aggiungi altri endpoint per gestire le varie operazioni CRUD sulle risorse. Ad esempio:

```
app.post('/api/posts', (req, res) => {
  // Logica per creare un nuovo post
  res.json({ message: 'POST request to /api/posts' });
});

app.put('/api/posts/:id', (req, res) => {
  // Logica per aggiornare un post esistente
  res.json({ message: `PUT request to /api/posts/${req.param
s.id}` });
});

app.delete('/api/posts/:id', (req, res) => {
  // Logica per eliminare un post esistente
  res.json({ message: `DELETE request to /api/posts/${req.par
ams.id}` });
});
```

4. Test dell'API utilizzando Fetch API:

- Crea un file HTML (`index.html`) e utilizza Fetch API per effettuare richieste all'API:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>REST API Test</title>
</head>
<body>
  <button onclick="getPosts()">Get Posts</button>
  <script>
    async function getPosts() {
      const response = await fetch('/api/posts');
      const data = await response.json();
      console.log(data);
    }
  </script>
</body>
</html>
```

5. Avvio del server:

- Avvia il server eseguendo `node server.js` dalla tua console.
- Apri `index.html` nel tuo browser e prova a fare clic sul pulsante "Get Posts" per testare la richiesta GET all'API.

6. Esplora altre operazioni CRUD:

- Modifica `index.html` per testare le richieste POST, PUT e DELETE all'API.
- Aggiungi logica nel server.js per gestire queste richieste e rispondere di conseguenza.