API REST

Guida allo sviluppo di API conformi alla specifica REST

Developer Student Clubs

Sapienza University of Rome

WHO AM I

Emanuele Giona

GitHub: /emanuelegiona

Twitter: @emanuele_giona

LinkedIn: /emanuelegiona

Studente magistrale di Computer Science presso Sapienza Università di Roma

Google DSC Sapienza Core Team

Appassionato di sviluppo back end, AI & machine learning, Internet of Things, open source

Download slides: dsc-sapienza/REST-talk1

Perché usare REST

Applicazioni di rete

Le applicazioni distribuite appaiono come un'unica applicazione centralizzata, quando in realtà sono eseguite (in tutto od in parte) presso macchine separate.

Un'applicazione di rete è un tipo di applicazione distribuita che fa uso di connettività di rete, non necessariamente in modo trasparente.

L'implementazione di un'applicazione di rete deve quindi seguire alcune linee guida e paradigmi affinché possa supportare queste architetture.

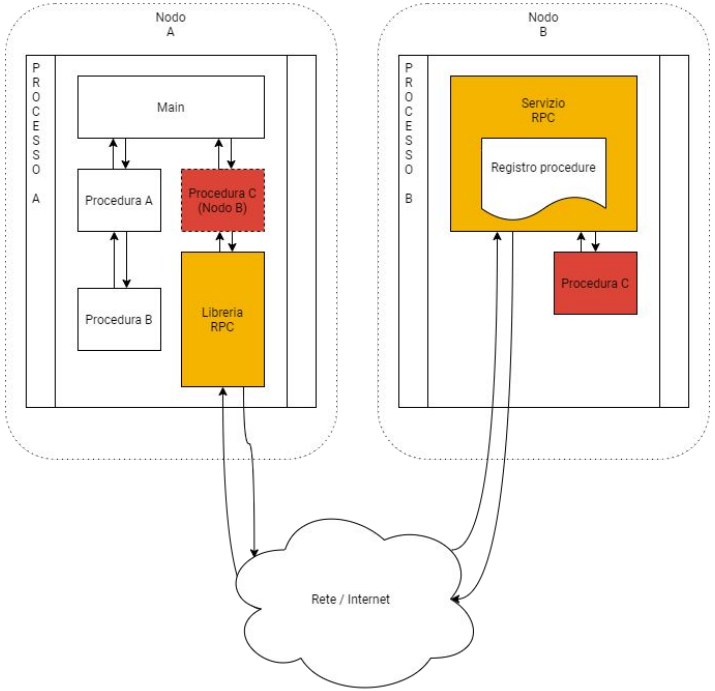
Cenni storici: Remote Procedure Call

Il modello RPC è basato su chiamate di procedure remote, ovvero implementate presso processi esterni al chiamante.

I parametri ed il valore di ritorno di una procedura remota sono inviati/ricevuti automaticamente per via della libreria RPC.

Pros: efficiente utilizzo di rete, conciso

Cons: altamente specializzato, *procedurale*



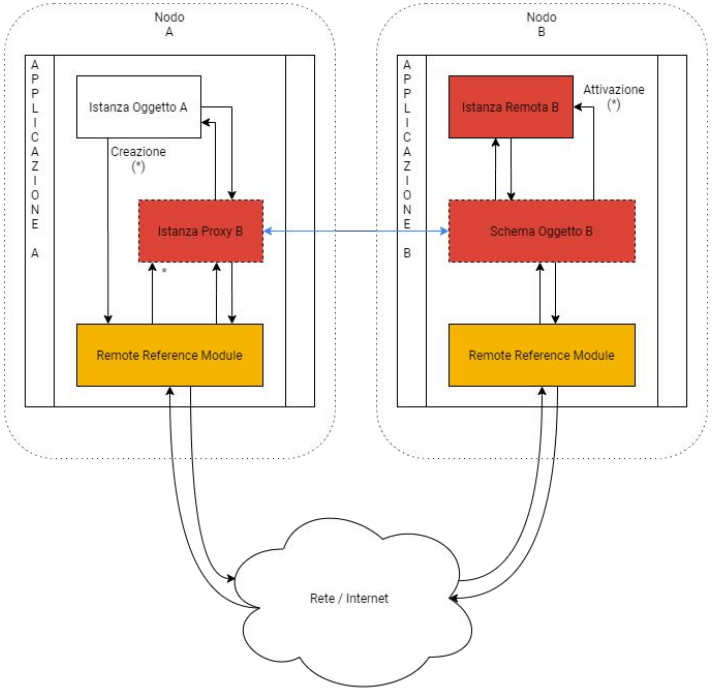
Cenni storici: Simple Object Access Protocol

Il modello SOAP è un'implementazione del modello RPC sfruttando il paradigma della programmazione ad oggetti.

Sfrutta il linguaggio WSDL per la generazione automatica di interfacce e classi.

Pros: *robusto*, basato su oggetti

Cons: *non indipendente*, poco efficiente



WSDL: Web Service Description Language (basato su XML)

Intuizioni

Perché utilizzare meccanismi complessi, antiquati, o poco efficienti?

KISS: Keep It Simple, Stupid

Per l'implementazione possiamo sfruttare protocolli più immediati, ben affermati ed efficienti, mentre per l'architettura si può prendere spunto dalle lezioni imparate con RPC e SOAP.

Perché non usare HTTP come protocollo di comunicazione, rimuovere l'esigenza di un oggetto distribuito, permettere un'interazione dinamica nel tempo, ed utilizzare un tipo di dati facilmente interpretabile...

Princìpi REST

REST: REpresentational State Transfer

Stile architetturale per sistemi distribuiti presentato nel 2000.

Princìpi:

1. Client-server
2. Stateless
3. Cacheable
4. Interfaccia uniforme
5. Stratificato
6. Codice on-demand (opzionale)

1: Client-server

Separazione delle responsabilità:

- Client
  interfaccia utente
- Server
  logica di applicazione, risorse (archiviazione dati, …)

Pros: scalabilità, semplificazione dei singoli componenti, evoluzione indipendente dei componenti.

1: Client-server, cos’altro?

Non tutte le applicazioni di rete devono seguire il modello client-server, imponendo una chiara distinzione di responsabilità tra i componenti.

Nelle applicazioni peer-to-peer (P2P) infatti ogni componente ha le stesse responsabilità degli altri, implementando localmente tutti gli strati concettuali (interfaccia utente, logica di applicazione, risorse condivise).

2: Stateless

Ogni richiesta del client contiene tutte le informazioni necessarie per essere processata, senza sfruttare dati memorizzati sul server riguardo le precedenti azioni dell'utente. Il client gestisce interamente lo stato dell'applicazione.

Pros: trasparenza, affidabilità, scalabilità.

Cons: dati ripetuti in una serie di richieste, dipendente dalla corretta implementazione dei vari client.

Ad esempio: visitare la prossima pagina dei risultati di ricerca.

2: Stateful vs stateless

La pagina correntemente servita all'utente è salvata nella sessione tra client e server.

Il link alla prossima pagina è espresso così:

/exams/next-page

Il server mantiene quindi lo stato dell'interazione con il client: stateful = non REST!

Cosa succederebbe se la richiesta venisse ripetuta per via di qualche errore di esecuzione lato client? Problema di consistenza ed idempotenza!

La pagina correntemente servita al client è direttamente codificata nell'URL in questo modo:

/exams?page=2

Il link alla prossima pagina può essere facilmente immaginato.

In questo caso, il server non deve salvare quale pagina è correntemente servita in quanto il client la richiede esplicitamente: stateless!

3: Cacheable

I dati di una risposta possono essere esplicitamente od implicitamente etichettati come cacheable o non-cacheable: una risposta cacheable permette il proprio riutilizzo per future richieste equivalenti.

Pros: completa *eliminazione* di alcune richieste.

Cons: consistenza di cache da verificare.

4: Interfaccia uniforme

Tutti i componenti sono gestiti tramite un'interfaccia comune, non specifica alle necessità dell'applicazione. Quattro vincoli sono imposti:

1. Identificazione delle risorse
2. Manipolazione delle risorse tramite rappresentazioni
3. Messaggi autodescrittivi
4. Hypermedia as the engine of the application state (HATEOAS)

Pros: implementazione e manutenzione semplificata.

Cons: efficienza ridotta.

5: Stratificato

Organizzazione gerarchica delle funzionalità dell'applicazione, in base al livello di astrazione. Ogni strato interagisce unicamente con lo strato immediatamente superiore od inferiore.

Pros: indipendenza implementativa, supporto sistemi legacy, semplificazione del sistema.

Cons: overhead nell'esecuzione delle richieste.

6: Codice on-demand (opzionale)

La specifica REST permette il download di funzionalità nella forma di applet o script eseguibili da parte del client. È un requisito opzionale in quanto riduce la trasparenza di sistema.

Pros: estensibilità.

Cons: ridotta trasparenza.

Elementi di una API REST

Elementi fondamentali

Il concetto fondamentale di una API REST è la risorsa.

Qualsiasi informazione a cui possa essere dato un nome è una risorsa. Una specifica risorsa ha un identificativo univoco associato che viene usato in tutte le operazioni che le riguardano.

Lo stato di una risorsa in un qualsiasi momento è la rappresentazione di quella risorsa: consiste di dati, metadati e link che possono servire al client per evolvere nello stato successivo. Il formato dati di una rappresentazione è detto media type: viene usato per instruire il client su come processare la rappresentazione.

Attenzione: una risorsa rappresenta il concetto di entità, non un'entità in particolare.

Concetto di entità vs istanza di entità

1. documento
2. esame
3. libro

Perfette risorse REST.

1. patente AB123456C
2. progettazione di algoritmi
3. “Deep Learning”, Goodfellow, Bengio, Courville

Non sono risorse REST, piuttosto dati e/o metadati di una particolare entità.

REST ed HTTP

Una vera API REST sfrutta a pieno la specifica HTTP:

- Utilizzo dei metodi HTTP appropriati per le operazioni
  - **GET** per la lettura di dati da un'istanza
  - **POST** per la creazione di una nuova istanza (non va specificato l'identificativo nella richiesta)
  - **PUT** per la modifica di un'istanza
  - **DELETE** per la cancellazione di un'istanza
- Parametri query string per filtrare, abilitare la paginazione, ordinare
  - Una sola URL con diverse opzioni tramite query string è più flessibile di molteplici URL specifiche

Identificazione delle risorse

Per ottenere i dati di una patente con numero AB123456C:

1. GET /documents/get-driving-license/AB123456C
2. POST /documents/driving-license/AB123456C
3. GET /documents/driving-license/AB123456C

quale usereste?

Identificazione delle risorse

Per ottenere i dati di una patente con numero AB123456C:

1. GET /documents/get-driving-license/AB123456C
   No! Il nome della risorsa non segue i princìpi REST
2. POST /documents/driving-license/AB123456C
   No! Il metodo POST non è appropriato per leggere dati
3. GET /documents/driving-license/AB123456C
   Sì! L'identificativo è passato correttamente e GET è il metodo HTTP appropriato

Rappresentazione di risorse (HATEOAS)

Hypermedia as the engine of the application state (HATEOAS) è un concetto fondamentale nell'architettura REST: ogni risorsa deve non solo essere rappresentata in base all'attuale stato del sistema, ma anche contenere al suo interno i link per passare in un nuovo stato.

Le risorse REST possono essere rappresentate facilmente tramite JSON: oltre a contenere valori per ogni campo in base all'istanza selezionata, possono essere inseriti anche i link per le operazioni che possono essere svolte in base allo stato attuale del sistema.

Esempio di risposta alla richiesta (prossima pagina):

GET /exam/algorithms-design

Rappresentazione di risorse (HATEOAS)

```
{
  “id”: “algorithms-design”,
  “name”: “Progettazione di algoritmi”,
  “prof”: 10,
  “course”: 58,
}
```

```
{
  “id”: “algorithms-design”,
  “name”: “Progettazione di algoritmi”,
  “prof”: 10,
  “course”: 58,
  “links”: [
    {
      “href”: “algorithms-design/take-part”
      “rel”: “prenota”,
      “type”: “POST”
    }
  ]
}
```

Quale delle due è una risposta REST?

Rappresentazione di risorse (HATEOAS)

```
{
  “id”: “algorithms-design”,
  “name”: “Progettazione di algoritmi”,
  “prof”: 10,
  “course”: 58,
}
```

No! Il client deve conoscere a priori la relazione tra un esame e l’azione della prenotazione, ed anche il link per realizzare la funzionalità.
Questo non segue il principio di interfaccia uniforme, in quanto vìola il requisito HATEOAS.

```
{
  “id”: “algorithms-design”,
  “name”: “Progettazione di algoritmi”,
  “prof”: 10,
  “course”: 58,
  “links”: [
    {
      “href”: “algorithms-design/take-part”
      “rel”: “prenota”,
      “type”: “POST”
    }
  ]
}
```

Sì! La risposta contiene il link per effettuare la prenotazione, ed il metodo POST è giusto.

Rappresentazione di risorse (HATEOAS)

Affinché sia rispettato pienamente il requisito HATEOAS, sia i dati contenuti nella risposta che i link possono variare a seconda dello stato del sistema: in questo modo sarà proprio l'hypermedia stesso a rappresentarlo.

Ad esempio: il link "prenota" sarà disponibile finché l'esame non sarà prenotato; da quel momento in poi, potrà essere sostituito con i link "cancella prenotazione" e "visualizza note".

REST in pratica

Implementazione di una API REST

1. Invia/ricevi file JSON
   Standard di fatto per la trasmissione dei dati via web; sempre meglio specificare il Content-Type pari ad **application/json** per migliorare la compatibilità
2. Identificativi risorse sotto forma di nomi singolari (plurali solo se si tratta di lista e collezioni)
   Esempi: GET /exams - GET /exam/algorithms-design
3. URL innestate per rendere esplicita la gerarchia tra le risorse
   Esempio: POST /exam/algorithms-design/take-part
4. Gestione degli errori tramite i codici standard HTTP (200, 404, ...)
   Gli errori standard permettono ai client di gestire in modo uniforme sia richieste ad API esterne che interne
5. La sicurezza non è un optional per le API REST!
   Let's Encrypt è un progetto supportato dalla Linux Foundation per ottenere certificati SSL/TSL gratis: https://letsencrypt.org/
6. Implementare un sistema di versioning
   Esempio: GET v1/exam/algorithms-design

Le vostre API sono REST?

Riferimenti

Roy T. Fielding, “REST Architectural Style”, PhD dissertation Chapter 5 (2000).
URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Roy T. Fielding, “REST APIs must be hypertext-driven”, Untangled (2008).
URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

RESTful API tutorial.
URL: https://restfulapi.net/

John Au-Yeung, “Best practices for REST API design”, StackOverflow Blog (2020).
URL: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design

Grazie per l’attenzione