Tecnologie e applicazioni web

REST API

Filippo Bergamasco (filippo.bergamasco@unive.it)
http://www.dais.unive.it/~bergamasco/
DAIS - Università Ca'Foscari di Venezia
Anno accademico: 2017/2018

Interoperabilità

Fin dall'avvento delle prime reti di calcolatori, è emersa la necessità di creare sistemi e protocolli per permettere l'interoperabilità di vari sistemi

Un sistema software può fornire una serie di funzionalità attraverso delle interfacce standard definite a priori (**API**)

API

Un Application Programming Interface (API) permette di definire i **metodi di comunicazione** tra componenti software

Un API non è ristretta necessariamente a librerie e framework software. Quando le funzionalità offerte sono rese disponibili attraverso il web sono dette **Web APIs**

Web API e il web moderno

Lo sviluppo di Web API ha assunto una rilevanza sempre più grande per lo sviluppo del web come lo conosciamo oggi.

Trend: SPA in cui l'interazione con il server è necessaria soltanto per la fruizione di quei servizi relativi alla gestione dei dati (non alla loro visualizzazione ed eventuale business logic)

Esempio

Supponiamo che si voglia realizzare un'applicazione per la gestione della biblioteca universitaria

Funzionalità chiave fornite dal sistema:

- Ricerca di libri, autori, copie disponibili
- Inserimento nuovi dati
- Rimozione autori/libri
- Gestione utenti
- … operazioni CRUD

Esempio

Le funzionalità sono fornite da un componente software attraverso un interfaccia (API) da definire

Problematiche:

- Rappresentazione dei dati
- Come invocare ciascuna funzionalità?
- Come rappresentare le interfacce in modo sufficientemente generico da essere indipendente dalle singole piattaforme?

Un po' di storia: CORBA

I primi sistemi per facilitare l'interoperabilità tra sistemi eterogenei sono stati CORBA e Microsoft COM.

- Complessi
- Utilizzano linguaggi dedicati per descrivere le interfacce e i dati
- Non compatibili fra loro

Message-oriented middleware

Infrastruttura software o hardware che gestisce l'interoperabilità attraverso lo scambio di messaggi.

Permettono un accoppiamento più lasco tra i componenti perchè l'invio dei messaggi è **asincrono** e non implementano sistemi per la gestione di uno stato condiviso

Message-oriented middleware

La gestione dei messaggi asincroni è implementata per mezzo di code di invio e ricezione gestite dal middleware.

Il middleware può trasformare dinamicamente i messaggi instradati verso una destinazione per soddisfare i requisiti del mittente e destinatario

Messaging Models

Point-to-Point: I messaggi prodotti da un entità vengono instradati verso uno specifico destinatario, solitamente in una coda FIFO

Publish/Subscribe (Pub/Sub): Meccanismo di distribuzione molti a molti in cui messaggi sono instradati in un certo canale in cui n altri client possono restare in ascolto

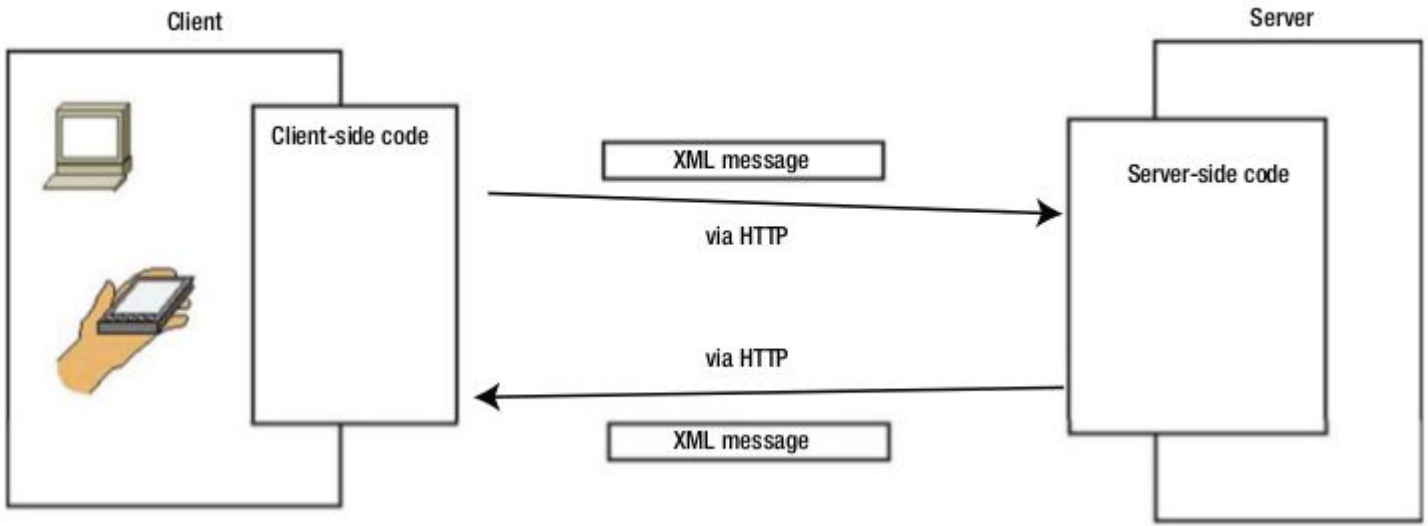
Un po' di storia: RMI

RMI (Remote Method Invocation) è stato molto utilizzato per permettere di invocare metodi di classi "remote".

- Ristretto all'ambiente Java.

XML-RPC è stato il primo sistema ad utilizzare XML e HTTP come protocollo di trasporto per invocare procedure remote

Un po' di storia: XML-RPC



Un po' di storia: XML-RPC

```
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
        <value><i4>40</i4></value>
    </param>
  </params>
</methodCall>
```

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

Un po' di storia: SOAP

XML-RPC si è evoluto nel tempo in SOAP (Simple Object Access Protocol). A dispetto del nome, rende molto più ricco (ma complesso) la definizione delle interfacce offerte

SOAP è orientato ai servizi web. Comprende un linguaggio (WSDL) per descrivere le funzionalità utilizzabili e come codificare i dati tra due sistemi

REST: Ritorno alla semplicità

Servizi web moderni tendono oggi a utilizzare codifiche dei dati più snelle (ex. JSON) e fornire le proprie API seguendo specifici **stili architetturali** (ex. REST) anziché veri e propri **protocolli e standard** (ex. SOAP)

REpresentational **S**tate **T**ransfer è uno stile architetturale definito per aiutare la creazione e la definizione di sistemi distribuiti

Utilizza HTTP come fondamento per l'architettura

REST

Essendo uno stile, e non uno standard, non ci sono delle regole formali da seguire per realizzare un sistema con un'architettura di tipo RESTful

Esistono però delle linee guida e alcuni **vincoli** importanti che aiutano la definizione di API con questo tipo di architettura

Client-server

Un'architettura di tipo REST segue il modello client-server.

- Il server gestisce una serie di servizi e attende delle richieste relative a quei servizi
- Le richieste sono effettuate da dei client attraverso un canale di comunicazione

Obiettivo: *separation of concerns* tra servizi e front-end

Stateless

Un'architettura di tipo REST non prevede uno stato condiviso tra client e server.

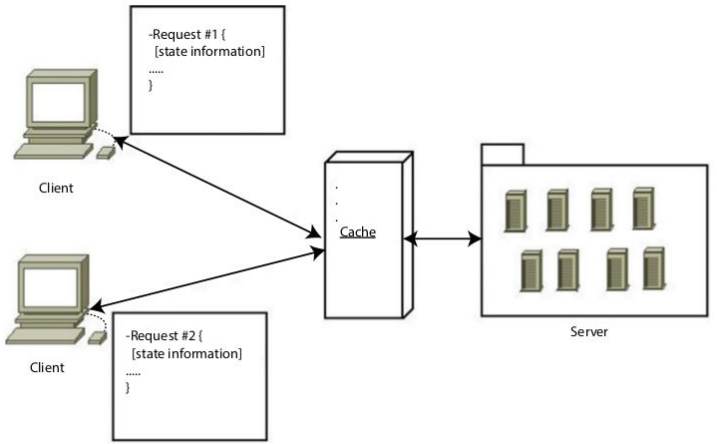
Ciascuna richiesta effettuata dal client deve contenere tutte le informazioni necessarie affinché sia soddisfatta indipendentemente dalle richieste precedenti

Stateless: vantaggi

1. **Scalabilità**: Il server non deve tenere in memoria alcuna informazione riguardo i client esistenti (e i servizi possono essere replicati su più server)
2. **Reliability**: In caso di disastro non c'è necessità di recuperare lo stato condiviso ma soltanto l'applicazione stessa
3. Più facile implementazione
4. **Visibilità**: ciascuna richiesta è atomica e può essere monitorata facilmente

Cacheable

Ciascuna richiesta può essere implicitamente o esplicitamente resa cacheable. In questo caso, richieste successive possono trovare risposta senza che il server debba necessariamente effettuare una determinata operazione



Uniform interface

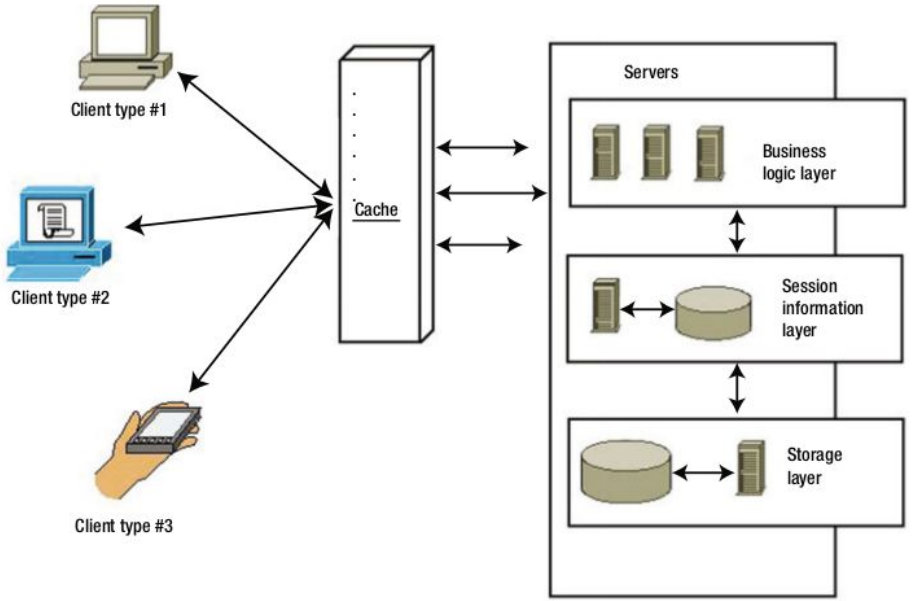
Un architettura REST deve fornire un interfaccia **unica** (uniform) verso tutti i tipi di client possibili

- Implementazione dei client **indipendente** da quella del server
- Semplifica l'implementazione dei client (non hanno "opzioni" su come usare le interfacce)
- Svantaggio: Potrebbe diminuire le performance nel caso di funzionalità in cui altre forme di comunicazione risultassero più efficienti

Layers

Un'architettura REST è pensata per la gestione di grandi quantità di traffico tipico del web.

Pertanto, è concepita come composizione di layers per gestire la complessità del sistema



Risorse

I mattoni che costituiscono la base su cui è costruita un'architettura REST sono le **risorse**.

- Definiscono l'oggetto dei servizi
- Sono le informazioni che vengono trasferite a seguito dei servizi
- Sono tutto ciò che può essere concettualizzato e definito attraverso un nome univoco

Risorse

Una risorsa possiede:

1. Un **identificativo** (URL) che la identifica univocamente in un qualsiasi istante temporale
2. Una o più **rappresentazioni** che definiscono la struttura delle sue informazioni.
3. Dei **metadati** che definiscono informazioni aggiuntive (formato, data di ultima modifica, etc)
4. Dati di **controllo**, ad esempio informazioni riguardanti il caching

Risorse: Identificativo

L'identificativo della risorsa fornisce un modo univoco per identificarla in un qualsiasi momento temporale.

L'identificativo è definito con la sintassi standard degli URL e specifica quindi il path completo di una risorsa all'interno di un determinato sistema.

Esempio:

```
/api/j-k-rowling/books/harry-potter-and-the-half-blood-prince
```

Risorse: Identificativo

Un concetto chiave è che la risorsa deve essere sempre identificabile **in un qualsiasi momento temporale**.

Esempio di cattiva risorsa:

`/api/j-k-rowling/books/last`

… come risolvere il problema?

Risorse: Identificativo

Le funzionalità fornite dal sistema rispetto ad una risorsa possono essere più complesse di una semplice “lettura”

REST prevede il concetto di **azioni** che un client può effettuare rispetto ad una certa risorsa. Possono essere mappate in metodi HTTP il cui funzionamento può essere reso più specifico **attraverso il campo query** dell'URL

Risorse: Identificativo

Esempi di azioni:

GET /api/j-k-rowling/books?filter=last

GET /api/books?q=[search term]

PUT
`/api/j-k-rowling/books/harry-potter-and-the-half-blood-prince?action=like`

Esempio di identificativo errato:

GET`/api/j-k-rowling/books/harry-potter-and-the-half-blood-prince/like`

Risorse: rappresentazione

Una singola risorsa può avere più rappresentazioni possibili (ad esempio un'immagine può essere in formato JPEG o PNG)

Solitamente un'architettura REST può gestire (contemporaneamente) diverse rappresentazioni di una stessa risorsa.

E' comune per un client poter richiedere una specifica rappresentazione

Risorse: rappresentazione

La richiesta di una specifica rappresentazione può avvenire in due modi:

1. **Content negotiation:**

   Si utilizzano gli header del protocollo HTTP per stabilire quali rappresentazioni sono disponibili e quali il client accetta:

```
Accept: text/html; q=1.0, text/*; q=0.8, image/gif; q=0.6,
image/jpeg; q=0.6, image/*;
q=0.5, */*; q=0.1
```

Risorse: rappresentazione

La richiesta di una specifica rappresentazione può avvenire in due modi:

2. **Utilizzando nomi usati per definire l'estensione dei files**

   Tecnica meno sofisticata ma più semplice per i casi più comuni

   ```
   GET /api/v1/books.json
   GET /api/v1/books.xml
   ```

Risorse: metadati

Comune per una risorsa è il concetto di possedere dei metadati che ne definiscono la struttura ed altre caratteristiche.

Un principio comune nelle architetture in stile REST è quello di **inserire dei collegamenti** ad altre risorse nei metadati di una risorsa.

Risorse: metadati

L'endpoint principale (la root che contiene tutte le risorse) può ad esempio ritornare metadati relativi a tutte le risorse principali fornite dall'interfaccia

```
GET /api/v1/
{"metadata": { "links": [
"books": {
"uri": "/books",
"content-type": "application/json"},
"authors": {
"uri": "/authors",
"content-type": "application/json"}]}
```

Definire le API

Da dove si inizia? Quali sono le linee guida per definire un servizio web con API ben congegnate?

1. I primi step sono simili a quelli comunemente effettuati per modellare i dati presenti in un determinato contesto (diagrammi er, object-oriented, etc)

   > Occorre cioè definire le risorse che saranno gestite dal sistema

Definire le API

2. Si stabiliscono poi quali sono le operazioni (azioni) che è possibile effettuare sulle risorse
- Operazioni CRUD
- Quali sono le azioni permesse su determinate risorse a seconda dei client? (autorizzazioni)
- Quali opzioni (o attributi) sono permesse su ciascuna azione? Ad esempio la lettura di una risorsa può assumere dei filtri o dei limiti alla sua dimensione

Definire le API

3. Si stabiliscono quali sono gli **endpoint**, cioè a quali URL sono disponibili le risorse, attraverso quali metodi HTTP e quali sono (se richiesto) gli status code possibili per ciascuna azione

4. Si definiscono i possibili metadati associati a ciascuna risorsa e l'eventuale inserimento di collegamenti all'interno dei metadati

Cosa rende buona un'API?

Developer friendly

Per definizione, un'API è un'interfaccia di *programmazione*. Pertanto, è orientata agli sviluppatori e non agli utenti finali.

Deve quindi il più possibile utilizzare **convenzioni** e una **nomenclatura** che renda facile comprenderne l'utilizzo anche senza leggere decine di pagine di manuale.

Cosa rende buona un'API?

Developer friendly

Il protocollo di comunicazione deve essere esplicito e godere di ampio supporto su molteplici piattaforme. Solitamente HTTP è la scelta più usata

Gli endpoint di un API devono utilizzare nomi facili da ricordare e che hanno senso rispetto alle risorse a cui si riferiscono

Cosa rende buona un'API?

Developer friendly

Come detto in precedenza, gli endpoints dovrebbero solo riferirsi a risorse e non alle azioni possibili su di essi.

Esempio di cattivi endpoint:

`/api/getAllBooks`

`/api/submitNewBook`

`/api/getNumberOfBooksInStock`

Cosa rende buona un'API?

```
/api/getAllBooks
/api/submitNewBook
/api/getNumberOfBooksInStock
```

- Esplosione di endpoint, ciascuno per ogni azione
- Quando si implementa una nuova azione non è ovvio il nome che avrà il nuovo endpoint
- Si basa su convenzioni che devono essere conosciute a priori (ex camel-case)

Cosa rende buona un'API?

Cattivo design:	REST style:
/getAllBooks	GET /books
/submitNewBook	POST /books
/updateAuthor	PUT /authors/:id
/getBooksAuthors	GET /books/:id/authors
/getNumberOfBooksOnStock	GET /books (This number can easily be returned as part of this endpoint.)
/addNewImageToBook	PUT /books/:id
/getBooksImages	GET /books/:id/images
/addCoverImage	POST /books/:id/cover_image
/listBooksCovers	GET /books (This information can be returned in this endpoint using subresources.)

Meno nomi da ricordare e semantica resa esplicita dai metodi HTTP

Cosa rende buona un'API?

Developer friendly

E' bene utilizzare un linguaggio di codifica dei dati che sia standard e la cui popolarità permetta di semplificare il lavoro al maggior numero possibile di sviluppatori

Ex. JSON, XML, etc.

Cosa rende buona un'API?

Estensibilità

Una buona API non è mai considerata completamente "finita". Questo avviene per molti motivi:

- Il business model di chi sviluppa l'API cambia nel tempo
- Nuove features sono aggiunte o rimosse
- Nuove interfacce vengono implementate per seguire la popolarità di alcune tecnologie emergenti

Cosa rende buona un'API?

Estensibilità

E' bene prevedere meccanismi per rendere esplicita la versione delle API

- Release successive delle API possono rendere non più compatibili client sviluppati per versioni precedenti
- Le funzionalità e le interfacce disponibili possono variare in funzione della versione

Cosa rende buona un'API?

Estensibilità

Un approccio comune è noto come Semantic Versioning (SemVer)

https://semver.org/

Seguendo questo principio, è buona norma inserire la versione delle API nell'URL degli endpoint

`/api/v1/j-k-rowling/books?filter=last`

Cosa rende buona un'API?

Documentazione aggiornata

Indipendentemente da quanto mnemonici siano i nomi degli endpoint, è sempre necessario produrre una buona documentazione che descrive le API

https://developers.facebook.com/docs/graph-api/using-graph-api/v2.1

Esempio di cattiva documentazione:

https://github.com/4chan/4chan-API

Cosa rende buona un'API?

Gestione degli errori

E' comune, specialmente durante lo sviluppo di API nuove, che gli utilizzatori effettuino errori nelle richieste:

- Nomi degli endpoint sbagliati
- Parametri sbagliati o mancanti

Il sistema dovrebbe prevedere di restituire messaggi di errori che permettano di capire il tipo di problema e come evitarlo

Cosa rende buona un'API?

Sicurezza

Nella progettazione di API occorre tener conto del fattore sicurezza:

Autenticazione: quali sono i soggetti che avranno accesso alle API? In che modo?

Autorizzazione: Quali sono le funzionalità a cui potranno accedere una volta che è stata stabilita la loro identità?

Cosa rende buona un'API?

Scalabilità

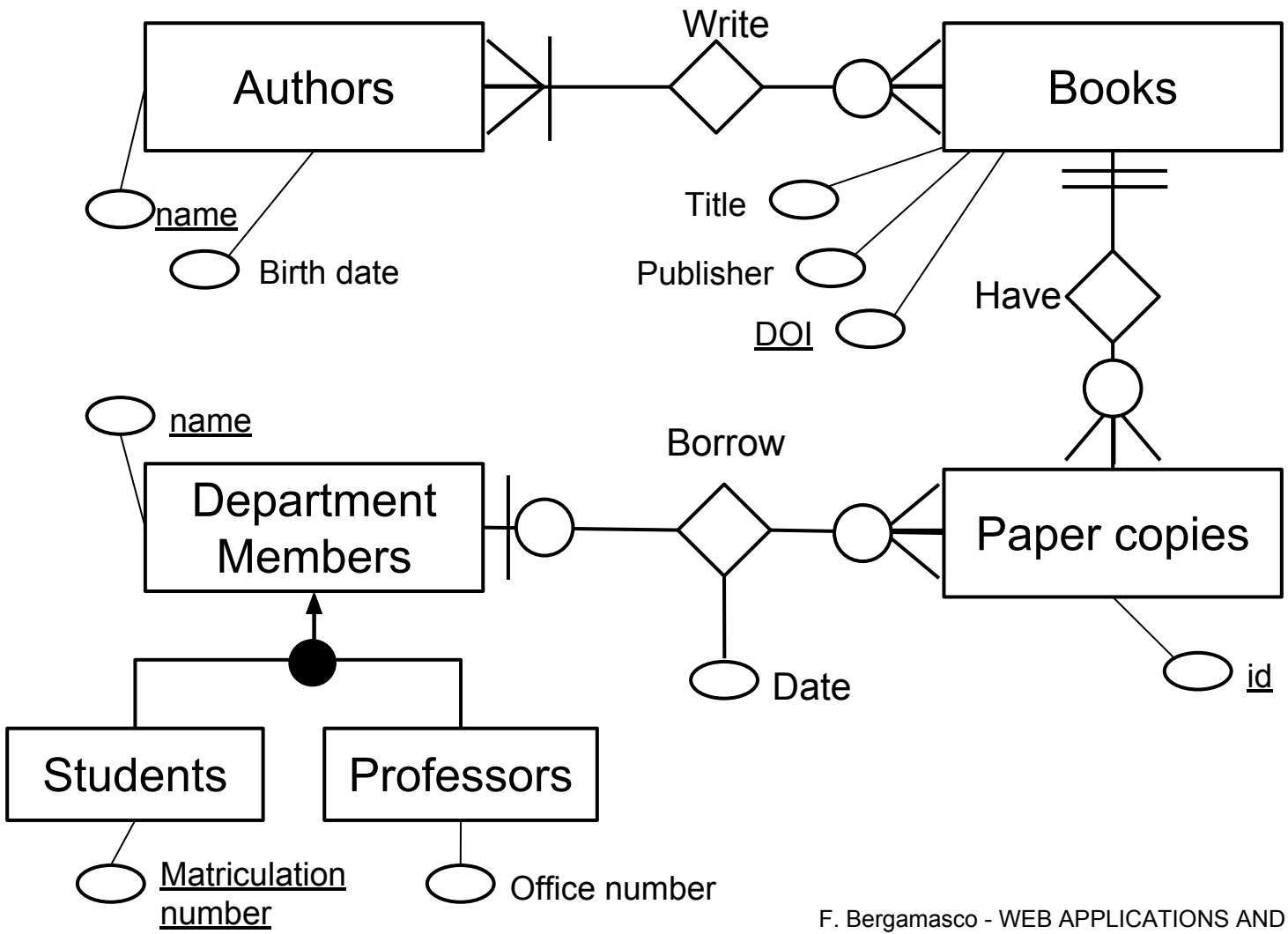
Delle buone API dovrebbero essere in grado di:

- Gestire quantità di traffico sempre più elevato senza sacrificare troppo le performance
- Occupare poche risorse se queste non sono in quel momento utilizzate

Linee guida: Mantenere l'architettura stateless e dividere le funzionalità in moduli (componenti) distinti e il più possibile indipendenti

Caso di studio

Vogliamo realizzare un web service con API in stile REST per organizzare la biblioteca del nostro dipartimento. La biblioteca gestisce diversi libri, ciascun libro è caratterizzato da una lista di autori e informazioni bibliografiche. I membri del dipartimento (studenti o professori) possono chiedere in prestito delle copie dei libri disponibili in biblioteca. Ciascun libro può essere chiesto in prestito per un periodo di tempo limitato


Authors
Write
Books
name
Birth date
Title
Publisher
DOI
Have
name
Department Members
Borrow
Paper copies
Date
id
Students
Professors
Matriculation number
Office number

Caso di studio

Risorsa	Proprietà	Descrizione
Books	Title, Authors, Publisher, DOI	La risorsa utilizzata per descrivere ciascun libro presente in biblioteca
Authors	Name, birth date, website, avatar, Books	La risorsa descrive un autore
PaperCopies	Number, Book, OwnersList, status	La risorsa descrive una copia cartacea di un determinato libro e ne registra la storia dei prestiti

Caso di studio

Risorsa	Proprietà	Descrizione
Students	Name, matriculation_number, etc.	La risorsa utilizzata per descrivere uno studente del dipartimento
Professors	Name, office_number, courses, etc.	La risorsa utilizzata per descrivere un professore del dipartimento

Caso di studio

Endpoint	Attributi	Metodo	Descrizione
/books	title: filtro di ricerca per termini del titolo topic: filtro di ricerca per argomento	GET	Ritorna la lista di tutti i libri gestiti. Gli attributi permettono di effettuare ricerche su titolo o topic
/books		POST	Crea un nuovo libro e lo salva nel database
/books/:id		GET	Ritorna tutti i dati associati ad un determinato libro
/books/:id		PUT	Modifica i dati relativi ad un determinato libro

Caso di studio

Endpoint	Attributi	Metodo	Descrizione
/books/:id/authors		GET	Ritorna la lista di tutti gli autori di un determinato libro
/authors	q: filtro di ricerca libero per tutti i dati relativi ad un autore	GET	Ritorna la lista di tutti gli autori presenti nel sistema
/authors/:id		GET	Ritorna tutte le informazioni associate ad un determinato autore
/authors/:id		PUT	Modifica i dati relativi ad un determinato autore

Caso di studio

Endpoint	Attributi	Metodo	Descrizione
/authors/:id/books		GET	Ritorna la lista di tutti i libri scritti da un determinato autore
/books/:id/copies	available: (true/false) restituisce soltanto le copie disponibili	GET	Ritorna la lista di tutte le copie cartacee di un determinato libro
/books/:id/copies		POST	Inserisce una nuova copia cartacea nel sistema
/books/:id/copies/:idc	lend: (idp) assegna la copia cartacea ad un determinato proprietario	PUT	Modifica il proprietario di una copia cartacea, aggiornandone al tempo stesso la storia

Caso di studio

Endpoint	Attributi	Metodo	Descrizione
/members		GET	Ritorna la lista di tutti i membri del dipartimento con accesso alla biblioteca
/members/students		GET	Ritorna la lista di tutti gli studenti
/members/students		POST	Inserisce un nuovo studente
/members/professors		GET	Ritorna la lista di tutti i professori
/members/professors		POST	Inserisce un nuovo professore

Caso di studio

Endpoint	Attributi	Metodo	Descrizione
/members/students/:id/copies		GET	Ritorna la lista di tutte le copie cartacee attualmente in prestito ad un dato studente
/members/professors/:id/copies		GET	Ritorna la lista di tutte le copie cartacee attualmente in prestito ad un professore

Caso di studio

Anche se non specificato in modo esplicito, ciascuna risorsa resa disponibile attraverso GET supporta i seguenti attributi:

`page=<n>`

`perpage=<m>`

Che specificano rispettivamente la pagina corrente e il numero di elementi per pagina