Tecnologie e applicazioni web

REST API

Filippo Bergamasco (filippo.bergamasco@unive.it)
http://www.dais.unive.it/~bergamasco/
DAIS - Università Ca'Foscari di Venezia
Academic year: 2021/2022

Interoperability

The need to create techniques and protocols to allow the interoperability of various systems emerged since the advent of first computer networks.

Basic principle: a software system can provide a set of functionalities through standard interfaces defined in advance.

API

An Application Programming Interface (API) defines communication methods and protocols between different software components.

An API is not limited to frameworks and libraries. When functions are deployed throughout the web they are called **Web APIs**.

Web APIs in the modern web

Web APIs have been significant for the development of the Web as we know it today.

Trend: SPA in which server interaction is limited to services related to data management only (data persistence, consistency, etc.)

Example

Suppose that we want to create an application to manage our university library

Features to implement:

- Search for books, authors, available copies
- Insertion of new books/authors/etc.
- Deletion of books/authors/etc.
- User management
- … CRUD operations

Example

Such features will be available through a software component exposing some APIs to be defined

Problems:

- How to encode the data?
- How to invoke each function?
- How to give a high-level representation of interfaces? (ie. independently to a specific software platform)

A bit of history: CORBA

The first systems designed to ease the interoperability between heterogeneous systems were CORBA and Microsoft COM

Problems:

- They used custom languages to describe interfaces and data
- Not compatible with each other
- Way too complex!

A bit of history: RMI & XML-RPC

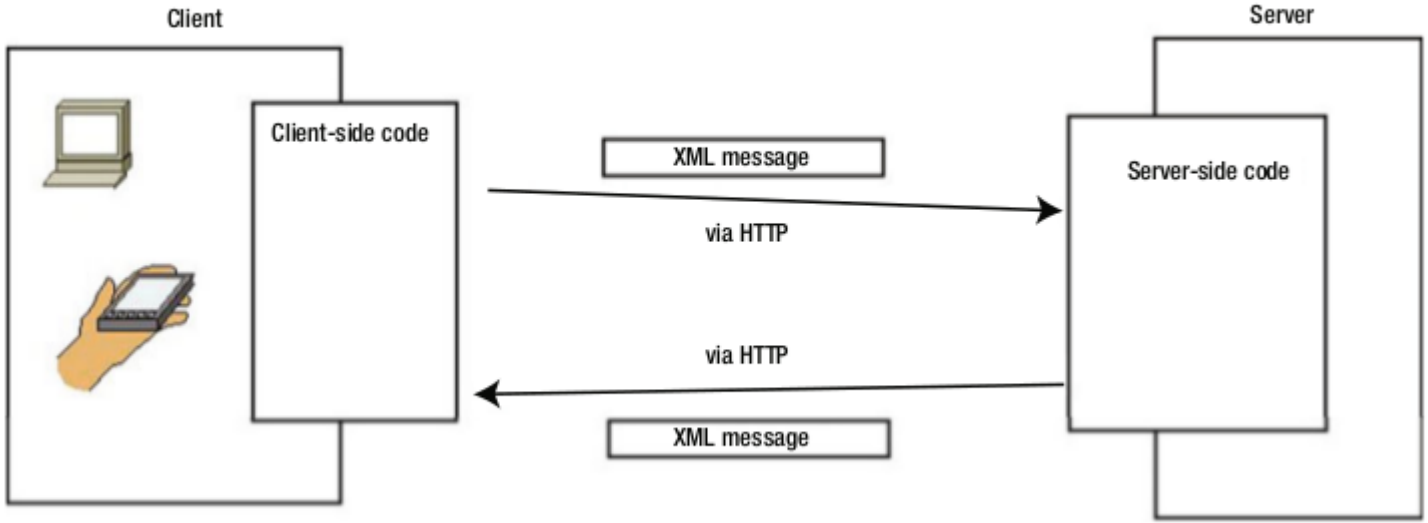
RMI (Remote Method Invocation) have been very popular to invoke methods of remote classes

- Restricted to Java ecosystem.

XML-RPC was the first standard designed to use XML and HTTP to invoke remote procedures.

- Simple and effective in practice!

XML-RPC


Client
Client-side code
XML message
via HTTP
via HTTP
XML message
Server
Server-side code

A bit of history: 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>
```

A bit of history: SOAP

XML-RPC evolved in SOAP (Simple Object Access Protocol). Despite its name, interface definition is a lot richer but also significantly more complex.

SOAP is designed to develop web services. Comprise a language (WSDL) to describe functions (input-output parameters) and data encoding

REST: simplicity is the key

Today's trend is to use simpler encodings (ex. JSON) and APIs following specific **architectural styles** (like REST) instead of formal **protocols and standard** (like SOAP)

REpresentational **S**tate **T**ransfer is an architectural style defined to help create and organize distributes systems.

Based on HTTP

REST

Being a style, and not a standard, means that there are no formal rules to follow to create a RESTful architecture

We have instead some **guidelines** and some **constraints** to help defining APIs with this architectural style

Client-server

A REST architecture follows the client-server model.

- Server exposes a set of services and waits for requests related to such services
- Requests are performed (by the clients) through a certain communication channel

Goal: *separation of concerns* between services and front-end

Stateless

A REST architecture should avoid any persistent state between client and server.

Each request done from the client must have all the information required for the server to understand it, without taking advantage of any stored data.

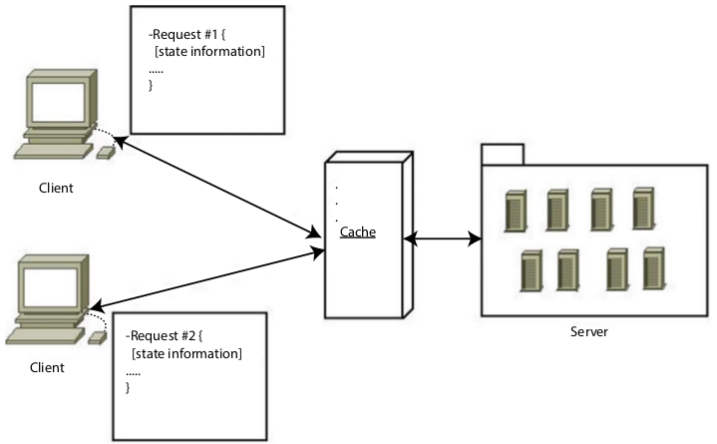
Each request is independent to the others.

Advantages of being stateless

1. **Scalability**: By not having to store data between requests, the server can free resources faster
2. **Reliability**: A stateless system can recover from a failure much easier than one that isn't, since the only thing to recover is the application itself.
3. **Easier implementation**
4. **Visibility**: Monitoring (logging) the system becomes easy when all the information required is inside the request

Cacheable

Every response to a request must be explicitly or implicitly set as cacheable. By caching the responses we can bypass some interactions (ex. to the database) to improve the performance



Uniform interface

A REST architecture should provide a uniform interface to all the possible clients

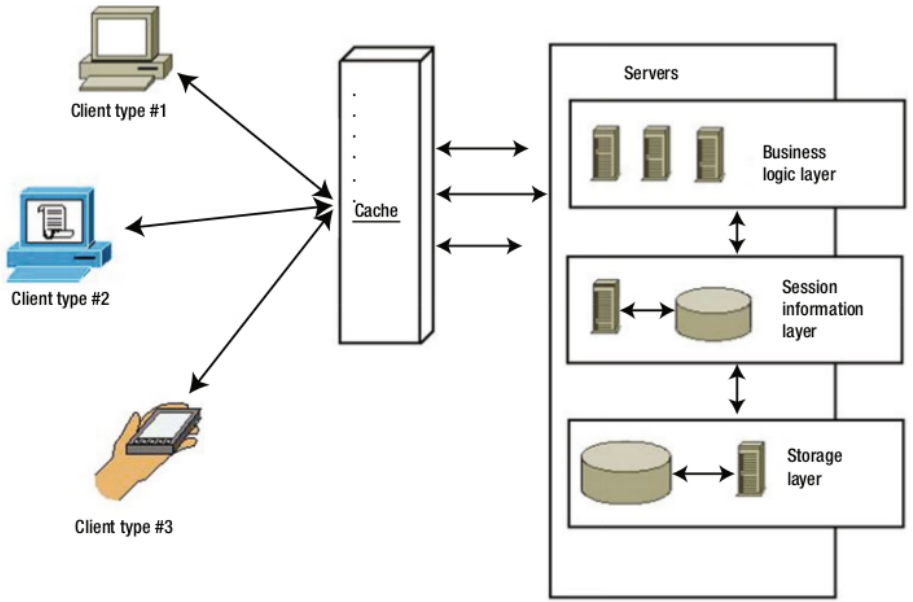
- Clients can be implemented independently from the server
- Client implementation is easier because they have less “options” on how to use the interfaces

Drawback: Having a standardized interface for all interactions might decrease the performance when a more optimized form of communication exists

Layered system

A REST architecture is designed to work properly with the massive amount of traffic that exists in the web.

Components are separated into layers to simplify the complexity and keep component coupling in check



Resources

The main building blocks of a REST architecture are the resources. A resource represent:

- What the services are going to be about
- The type of information that is going to be transferred and their related actions
- **An abstraction of anything that can be conceptualized in our application**

Resources

A resource has:

1. An **identifier** (URL) to uniquely identify a concept at any given time
2. One or more **representations** describing the structure of the information.
3. Some **metadata** to specify additional useful information (format, last modification date, etc)
4. **control data,** to specify cache-directives, additional constraints, etc.

Resources: Identifier

The resource identifier should provide:

- A unique way of identification
- The full path to the resource

It is defined using the standard URL syntax specifying the full resource path inside the server system

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

Resources: Identifier

The identifier of each resource must be able to reference it unequivocally **at any given point in time**.

Example of a bad resource identifier:

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

… how to solve the problem?

Resources: Identifier

REST-style APIs support a whole spectrum of operations that can be performed on a certain resource.

REST provides the concept of **actions** that a client can perform against a certain resource. Actions can be **mapped to HTTP methods** whose semantic can be made more specific using the URL's **query** section.

Resources: identifier

Good examples:

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`

Wrong example:

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

Resources: representation

A resource can have multiple representations (ex. an image can be in JPG and PNG format)

A REST architecture can manage (at the same time) different representations of a certain resource.

A client can request a specific representation among the different alternatives

Resources: representation

A client can ask for a certain representation in two ways:

1. **Content negotiation:**

   HTTP protocol headers are used to communicate what are the available representations and what representations are accepted by the client

   ```
   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
   ```

Resources: representation

A client can ask for a certain representation in two ways:

2. **Using resource name extension**

   Less sophisticated, but easier to implement for common cases (note: not based on mime types)

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

Resources: metadata

A resource can include some metadata to define its structure and other characteristics.

A common principle for REST architectures is to **insert links to other resources into resource metadata.**

Resources: metadata

The main endpoint (ie. root containing all the resources) usually provides metadata describing all the resources managed by the interface.

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

Defining a good API

How to start? What are the guidelines to define a web service with well-defined APIs?

1. The first step is to model entities (and data) belonging to a certain context (ER diagram, object-oriented models, etc.)
    - This allow us to **define the resources** managed by our system

Defining a good API

2. We decide what operations (actions) can be performed on each resource
   - CRUD operations
   - Authorizations (what actions are allowed to a certain client?)
   - What options or attributes are required on each action? For example one can retrieve (get) a resource using some filters or limiting its dimension

Defining a good API

3. We decide what are the **endpoints**, ie. what is the URL of each resource, what methods are used and what are the possible status codes for each action

4. We define the metadata associated to each resource

What makes a good API?

Developer friendly

By definition, an API is a programming interface. Therefore it must be oriented to other developers and not to the end users.

A good API should use **naming conventions** that can be easily understood even without documentation.

What makes a good API?

Developer friendly

Communication protocol must be explicit and supported by multiple platforms. HTTP is the most popular choice

Endpoints should use simple and meaningful names with respect to the corresponding resources

What makes a good API?

Developer friendly

Endpoints should refer to resources and not to actions (verbs) that we can perform on them.

Example of badly designed endpoints:

`/api/getAllBooks`

`/api/submitNewBook`

`/api/getNumberOfBooksInStock` … why?

What makes a good API?

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

- Endpoint “explosion”! We need to create an endpoint for each pair (resource,action)
- When we implement a new action it is not clear how to name the new endpoint
- Must use conventions known a-priori (ex camel-case)

What makes a good API?

Bad 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.)

REST style requires less endpoints to remember and provide an explicit semantics thanks to HTTP methods

What makes a good API?

Developer friendly

We should use standard languages to encode the data. A good language should be popular enough to simplify its implementation by using publicly available libraries and frameworks

Ex. JSON, XML, etc.

What makes a good API?

Should be extensible

A good API is never fully “finished”, for different reasons:

- The business model can change over time
- New features are added and the old ones are removed
- New interfaces might be implemented to follow new emerging technologies (ex. from XML to JSON)

What makes a good API?

Should be extensible

We should use mechanism to explicitly mark the API version

- Subsequent releases might broke the backward compatibility with older clients
- Features and interfaces may vary with different API versions

What makes a good API?

Should be extensible

A common approach is the so called Semantic Versioning (SemVer):

https://semver.org/

Following this principle, is a good idea to insert the API version in the endpoint URL itself:

`/api/v1/j-k-rowling/books`

What makes a good API?

Documentation

It is a good practice to produce a concise documentation to describe API endpoints

Example of good documentation:

https://developers.facebook.com/docs/graph-api/overview

What makes a good API?

Handling of exceptions/errors

It is common, in particular when developing new APIs, that wrong API calls are performed by the clients:

- Wrong endpoint names
- Wrong or missing parameters

A good REST-style architecture should return meaningful error messages, explaining what's the problem and how to avoid it

What makes a good API?

Security

When designing a new API, we should always take security into account:

Authentication: what clients are allowed to use certain APIs? And how?

Authorizations: What features a client might use once its identity is verified?

What makes a good API?

Scalability

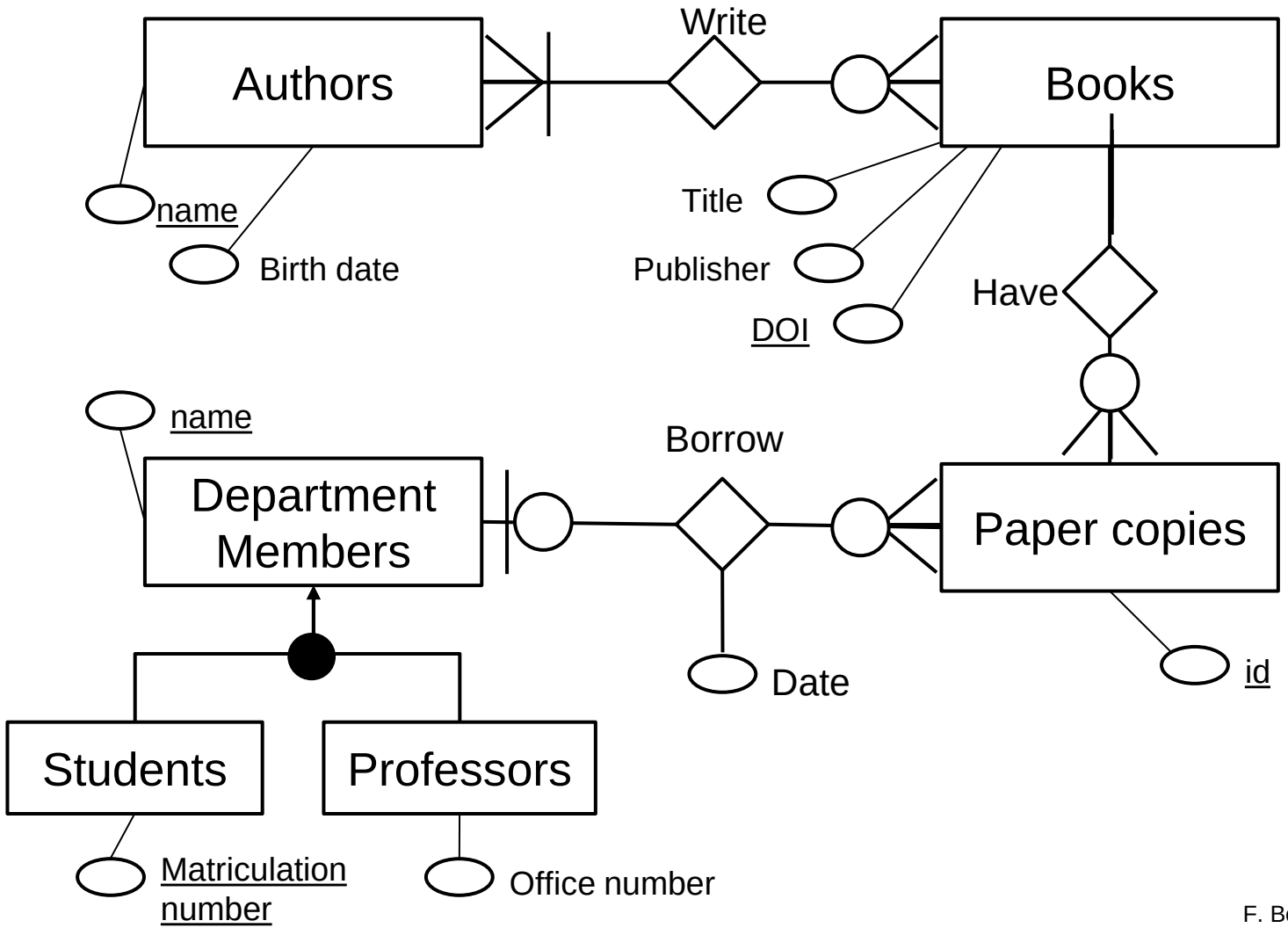
Good APIs should:

- Manage heavy traffic loads without sacrificing the performance
- Occupy the minimum amount of resources needed

Suggestion: keep the architecture stateless and divide the functions into separate independent components.

Case study

Let’s create a web service with REST-style APIs to manage our departmental library. The library should manage multiple books, characterized by different authors and relevant bibliographic info. Department members (students or professors) can borrow paper copies if available. Each book can be lend for a limited period of time


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

Case study

Resource	Property	Description
Books	Title, Authors, Publisher, DOI	Resource used to describe each book in our library
Authors	Name, birth date, website, avatar, Books	Resource used to describe an author
PaperCopies	Number, Book, OwnersList, status	Resource used to describe a paper copy of a certain book, storing the list of previous owners

Case study

Resource	Property	Description
Students	Name, matriculation_number, etc.	Resource used to describe a student
Professors	Name, office_number, courses, etc.	Resource used to describe a professor

Case study

Endpoint	Attributes	Method	Description
/books	title: search filter for terms in the book title topic: filter for topics	GET	Returns the list of all the books. Attributes allows to restrict the list to a certain title or topics
/books		POST	Inserts a new book in the system
/books/:id		GET	Returns data associated to a certain book
/books/:id		PUT	Modifies data associated to a certain book

Case study

Endpoint	Attributes	Method	Description
/books/:id/authors		GET	Returns the book authors
/authors	q: free search filter	GET	Returns a list of all the authors managed by the system
/authors/:id		GET	Returns all the data associated to a certain author
/authors/:id		PUT	Modifies the data associated to a certain author

Case study

Endpoint	Attributes	Method	Description
/authors/:id/books		GET	Returns a list of all the books by a certain author
/books/:id/copies	available: (true/false) limit to available paper copies	GET	Returns a list of all the paper copies of a book
/books/:id/copies		POST	Creates a new paper copy
/books/:id/copies/:idc	lend: (idp) assings a paper copy to a new owner	PUT	Modifies the current owner of a paper copy, updating the owners history

Case study

Endpoint	Attributes	Method	Description
/members		GET	Returns a list of all department members
/members/students		GET	Returns a list of managed students
/members/students		POST	Creates a new student
/members/professors		GET	Returns a list of all professors
/members/professors		POST	Creates a new professor

Case study

Endpoint	Attributes	Method	Description
/members/students/:id/copies		GET	Returns a list of all paper copies currently lend to a certain student
/members/professors/:id/copies		GET	Returns a list of all paper copies currently lend to a certain professor

Case study

Each resource accessible via GET also supports the following attributes:

`page=<n>`

`perpage=<m>`

Specifying the current page and the number of elements per page.