

API



F.A.Q

What is
an API?

La Web Programable

- La **Web** es una **fuentes** inagotable de **datos**
- **Aplicaciones** más interesantes **extraen datos** de la **Web**
- ... o **actualizan datos** en otras aplicaciones
- **Servidores** entregan **XML** o **JSON** en lugar de HTML + CSS
- **Cliente** debe parsear e **interpretar** lo recibido
- Al igual que la solicitud manual se usa **HTTP** para acceder a los datos remotos



Apparently our open API is giving our customers
unprecedented control over their own lives and
allowing them to seize control of their destinies.
So please shut it down.

**I AM AN
API FANBOY.**

YellowAPI.com®



Generic API request

Se identifican 3 elementos

Scope

Acción/Operación

Datos

Criterios de Clasificación

- En general todos comparten una cosa: **usan HTTP**
 - un “**envelope**” en el body del request
 - el **contenido** mismo en el body
- Difieren en cuanto a
 - dónde va la información del **método** (operación)
 - dónde va la información del **scope y datos**

Especies

- **operación**
 - en el método del protocolo HTTP (set limitado)
 - en el URI
 - Ej. Flickr: `http://www.flickr.com/services/rest?method=flickr.photos.search`
 - en el sobre
- **scope**
 - en el URI
 - Ej. Flickr: `http://www.flickr.com/services/restmethod=flickr.photos.search&tags=penguin`
 - en el sobre

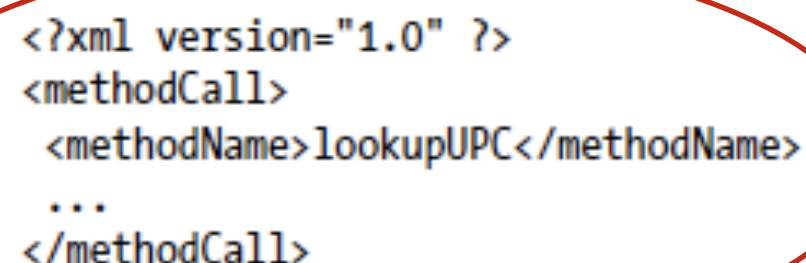
RESTful WS, Resource Oriented WS

- **operación** en el **método** (RESTful)
- **scope** en el **path** (resource oriented arch)
- Ejemplos
 - servicios que exponen protocolo ATOM (RSS)
 - amazon S3
 - la mayor parte de los servicios Yahoo
 - Twitter
 - sitios Web estáticos!
 - muchas aplicaciones Web

RPC-Style Architecture

- muy importante en software empresarial (SOA)
- servicio **toma sobres** cargados y **devuelve sobres**
- **operación** y **scope** al interior del **sobre**
- varios tipos de sobre, el más popular **SOAP** pero puede usarse XML-RPC u otro
- Modelo de cómputo de procedure call

```
POST /rpc HTTP/1.1
Host: www.upcdatabase.com
User-Agent: XMLRPC::Client (Ruby 1.8.4)
Content-Type: text/xml; charset=utf-8
Content-Length: 158
Connection: keep-alive
```



```
<?xml version="1.0" ?>
<methodCall>
  <methodName>lookupUPC</methodName>
  ...
</methodCall>
```

REST-RPC Híbridos

- Modelo similar a procedure call pero método no va en el sobre sino en el URL Path
- HTTP es en realidad usado como un sobre
- Usar un GET para modificar datos rompe la arquitectura de la Web
- A veces se les llama HTTP-POX (HTTP + plain old xml)

WSDL y WADL

- **WSDL describe** todos los detalles de un **servicio SOAP** (métodos, argumentos, tipo de datos, etc)
 - pieza fundamental, imprescindible
- **WADL** (Web App Description Language) permite describir un Web Service RESTful
 - mucho menos necesaria
 - pocos servicios proveen este archivo

Pero si con SOAP es muy simple ...

- SOAP
- WSDL
- (UDDI)

is it really ?

- WS-*: web services technologies
 - WS-Transfer
 - WS-Enumeration
 - WS-Choreography
 - WS-Security
 - WS-Addressing
 - WS-Trust
 - WS-Transactions
 - WS-Federation
 - WS-SecureConversation
 - WS-Reliability
 - WS-Eventing ...

WS-I Basic Profile

- Web Services Interoperability Organization
- **The WS-I Basic Profile**, a specification from the Web Services Interoperability industry consortium (WS-I), provides interoperability guidance for core Web Services specifications such as SOAP, WSDL, and UDDI

WS-I Basic Profile 1.1

- 1. Introduction
 - 1.1. Relationships to Other Profiles
 - 1.2. Changes from Basic Profile Version 1.0
 - 1.3. Guiding Principles
 - 1.4. Notational Conventions
 - 1.5. Profile Identification and Versioning
- 2. Profile Conformance
 - 2.1. Conformance Requirements
 - 2.2. Conformance Targets
 - 2.3. Conformance Scope
 - 2.4. Claiming Conformance
- 3. Messaging
 - 3.1. SOAP Envelopes
 - 3.1.1. SOAP Envelope Structure
 - 3.1.2. SOAP Envelope Namespace
 - 3.1.3. SOAP Body Namespace Qualification
 - 3.1.4. Disallowed Constructs
 - 3.1.5. SOAP Trailers
 - 3.1.6. SOAP encodingStyle Attribute
 - 3.1.7. SOAP mustUnderstand Attribute
 - 3.1.8. xsi:type Attributes
 - 3.1.9. SOAP1.1 attributes on SOAP1.1 elements
 - 3.2. SOAP Processing Model
 - 3.2.1. Mandatory Headers
 - 3.2.2. Generating mustUnderstand Faults
 - 3.2.3. SOAP Fault Processing
 - 3.3. SOAP Faults
 - 3.3.1. Identifying SOAP Faults
 - 3.3.2. SOAP Fault Structure
 - 3.3.3. SOAP Fault Namespace Qualification
 - 3.3.4. SOAP Fault Extensibility
 - 3.3.5. SOAP Fault Language
 - 3.3.6. SOAP Custom Fault Codes
 - 3.4. Use of SOAP in HTTP
 - 3.4.1. HTTP Protocol Binding
 - 3.4.2. HTTP Methods and Extensions
 - 3.4.3. SOAPAction HTTP Header
 - 3.4.4. HTTP Success Status Codes
 - 3.4.5. HTTP Redirect Status Codes
 - 3.4.6. HTTP Client Error Status Codes
 - 3.4.7. HTTP Server Error Status Codes
 - 3.4.8. HTTP Cookies
- 4. Service Description
 - 4.1. Required Description
 - 4.2. Document Structure
 - 4.2.1. WSDL Schema Definitions
 - 4.2.2. WSDL and Schema Import
 - 4.2.3. WSDL Import location Attribute Structure
 - 4.2.4. WSDL Import location Attribute Semantics
 - 4.2.5. Placement of WSDL import Elements
 - 4.2.6. XML Version Requirements
 - 4.2.7. XML Namespace declarations
 - 4.2.8. WSDL and the Unicode BOM
 - 4.2.9. Acceptable WSDL Character Encodings
 - 4.2.10. Namespace Coercion
 - 4.2.11. WSDL documentation Element
 - 4.2.12. WSDL Extensions

¿¿Basic??

- 4.3. Types
 - 4.3.1. QName References
 - 4.3.2. Schema targetNamespace Structure
 - 4.3.3. soapenc:Array
 - 4.3.4. WSDL and Schema Definition Target Namespaces
- 4.4. Messages
 - 4.4.1. Bindings and Parts
 - 4.4.2. Bindings and Faults
 - 4.4.3. Declaration of part Elements
- 4.5. Port Types
 - 4.5.1. Ordering of part Elements
 - 4.5.2. Allowed Operations
 - 4.5.3. Distinctive Operations
 - 4.5.4. parameterOrder Attribute Construction
 - 4.5.5. Exclusivity of type and element Attributes
- 4.6. Bindings
 - 4.6.1. Use of SOAP Binding
- 4.7. SOAP Binding
 - 4.7.1. Specifying the transport Attribute
 - 4.7.2. HTTP Transport
 - 4.7.3. Consistency of style Attribute
 - 4.7.4. Encodings and the use Attribute
 - 4.7.5. Multiple Bindings for portType Elements
 - 4.7.6. Operation Signatures
 - 4.7.7. Multiple Ports on an Endpoint
 - 4.7.8. Child Element for Document-Literal Bindings
 - 4.7.9. One-Way Operations
 - 4.7.10. Namespaces for soapbind Elements
 - 4.7.11. Consistency of portType and binding Elements
 - 4.7.12. Describing headerfault Elements
 - 4.7.13. Enumeration of Faults
 - 4.7.14. Type and Name of SOAP Binding Elements
 - 4.7.15. name Attribute on Faults
 - 4.7.16. Omission of the use Attribute
 - 4.7.17. Default for use Attribute
 - 4.7.18. Consistency of Envelopes with Descriptions
 - 4.7.19. Response Wrappers
 - 4.7.20. Part Accessors
 - 4.7.21. Namespaces for Children of Part Accessors
 - 4.7.22. Required Headers
 - 4.7.23. Allowing Undescribed Headers
 - 4.7.24. Ordering Headers
 - 4.7.25. Describing SOAPAction
 - 4.7.26. SOAP Binding Extensions
- 4.8. Use of XML Schema
- 5. Service Publication and Discovery
 - 5.1. bindingTemplates
 - 5.2. tModels
- 6. Security
 - 6.1. Use of HTTPS
- Appendix A: Referenced Specifications
- Appendix B: Extensibility Points
- Appendix C: Defined Terms
- Appendix D: Acknowledgements

Ejemplo RESTful Service

API de Github

<https://developer.github.com/v3/>

Ejemplo API Github

List commits on a repository

```
GET /repos/:owner/:repo/commits
```

operación
en método

Parameters

scope en path

Name	Type	Description
sha	string	SHA or branch to start listing commits from. Default: the repository's default branch (usually <code>master</code>).
path	string	Only commits containing this file path will be returned.
author	string	GitHub login or email address by which to filter by commit author.
since	string	Only commits after this date will be returned. This is a timestamp in ISO 8601 format: <code>YYYY-MM-DDTHH:MM:SSZ</code> .
until	string	Only commits before this date will be returned. This is a timestamp in ISO 8601 format: <code>YYYY-MM-DDTHH:MM:SSZ</code> .

Response

```
Status: 200 OK
Link: <https://api.github.com/resource?page=2>; rel="next"
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4999

[
  {
    "url": "https://api.github.com/repos/octocat/Hello-World/commits/6dcb09b5b57875f334f61aebed695e2e4193db5e",
    "sha": "6dcb09b5b57875f334f61aebed695e2e4193db5e",
    "html_url": "https://github.com/octocat/Hello-World/commit/6dcb09b5b57875f334f61aebed695e2e4193db5e"
```

Ejemplo de Uso

Recorrer los commits del syllabus del curso

- **GET /**: se obtiene **repository_url** para construir URL del repositorio
- **GET /repos/IIC2513-2015-2/syllabus**: se obtiene el recurso que representa el repositorio y se obtiene **commits_url**
- **GET /repos/IIC2513-2015-2/syllabus/commits**: obtenemos un JSON Array de commits que podemos recorrer

Observaciones

- Programa puede simplificarse mucho si sabemos por ejemplo que el URL que necesitamos es [/repos/IIC2513-2015-2/syllabus/commits](#)
- Restful API permite que un programa use la Web en una forma similar a como lo haría una persona
- no se requiere de lenguajes de descripción (autodescriptivo)
- muchas de las APIs ofrecidas como RESTful

Código que accede a Servicio RESTful

- la mayoría de los lenguajes modernos tiene buen soporte para esto (librerías o built-in)
- Ruby viene con dos librerías para clientes HTTP: `open-uri` y `Net::HTTP`
- Existen otras base como `HTTPClient` y `Excon`
- Y existen wrappers que facilitan la interacción y que se basan según disponibilidad en alguna de las librerías base: `Faraday`, `Rest-Client`, `HTTPParty`

Best Practices

- Resource Oriented
 - cada cosa interesante de la aplicación debe ser expuesta como recurso (en una URI)
 - todos los accesos a través de interfaz http uniforme (GET, POST, etc)
 - La mayoría de los RESTful WS exponen un número muy grande de URIs (en comparación con RPC-Style)

- Statelessness
 - en el sentido de application state - info sobre el camino que el cliente ha tomado en la aplicación
 - application state se mantiene en el cliente
 - se envía lo que se necesite del estado en cada request
- Conectedness
 - links permiten pasar al cliente de un estado de aplicación a otro
 - la web humana es un muy buen ejemplo

REST en Rails

- routing - cada recurso debe exponer su URI
- **resources** permite automatizar mucho

Prefix	Verb	URI Pattern	Controller#Action
movies	GET	/movies{.:format}	movies#index
	POST	/movies{.:format}	movies#create
new_movie	GET	/movies/new{.:format}	movies#new
edit_movie	GET	/movies/:id/edit{.:format}	movies#edit
movie	GET	/movies/:id{.:format}	movies#show
	PATCH	/movies/:id{.:format}	movies#update
	PUT	/movies/:id{.:format}	movies#update
	DELETE	/movies/:id{.:format}	movies#destroy

Formats

- Las rutas creadas como recurso aceptan opcionalmente un formato
- `/movies(.:format)`
- El formato se puede usar para responder de manera diferente dependiendo de lo solicitado por el cliente

```
class MoviesController < ApplicationController
  def index
    respond_to do |format|
      format.html { @posts = Post.find(:all, limit: 5) }
      format.json { render json: Post.find(:all) }
    end
  end
end
```

Autenticación

- A menos que la API se quiera exponer sólo para requests Ajax (especialmente útil para Single Page Applications), la autenticación no debería ser por cookies
- Se genera un API access token que identifica de manera única a cada usuario, y éste se entrega en cada request
- La obtención del access token puede ser manual (usuario lo obtiene desde la web) o mediante protocolos como OAuth

Cómo construir la API en una aplicación Rails

- Si bien uno puede tener acciones que respondan tanto HTML y JSON dependiendo del formato, muchas veces será conveniente tener un set separado de controllers (¡o incluso una aplicación diferente!) que provean la API
- No siempre las acciones provistas en la API coinciden con la UI expuesta a usuarios
- Mezclar ambos formatos de respuesta en la misma acción puede resultar en acciones más complejas de lo necesario
- Rails incluso cuenta con una versión simplificada exclusivamente para construir APIs: `rails new --api`