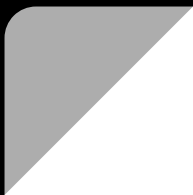


# Arquitecturas Web y REST APIs

Diseño de Aplicaciones II - 2021

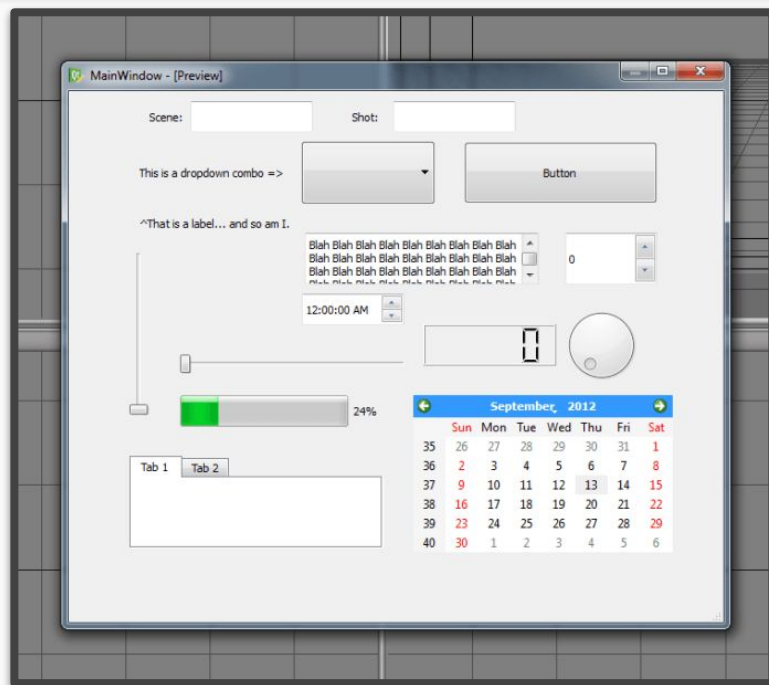


# Agenda:

- 1) Introducción. ¿Qué es una aplicación web? ¿Cómo funciona?
- 2) Arquitecturas Web y HTTP como protocolo
- 3) REST APIs
- 4) Buenas prácticas a la hora de diseñar una API Rest

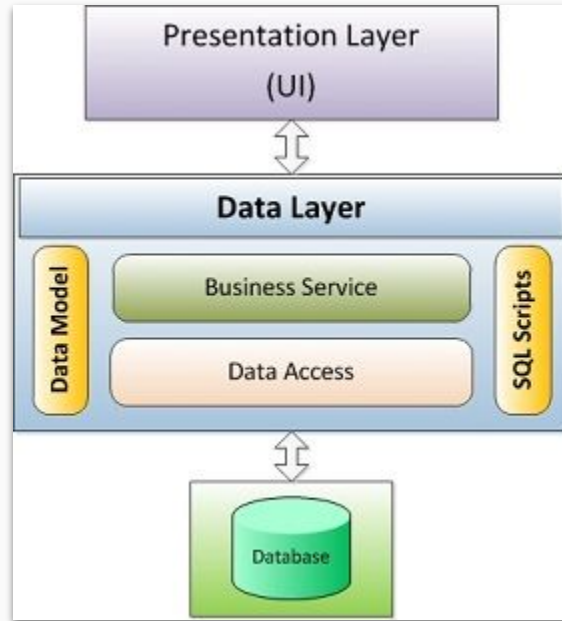
# Introducción a aplicaciones Web

# ¿Cómo describimos la aplicación construida en DA1?



- Escritorio
- Desktop
- Centralizada
- Windows Forms (.NET Framework)
- Single-user
- SQL Server
- Misma máquina

# ¿Cómo describimos la aplicación construida en DA1?

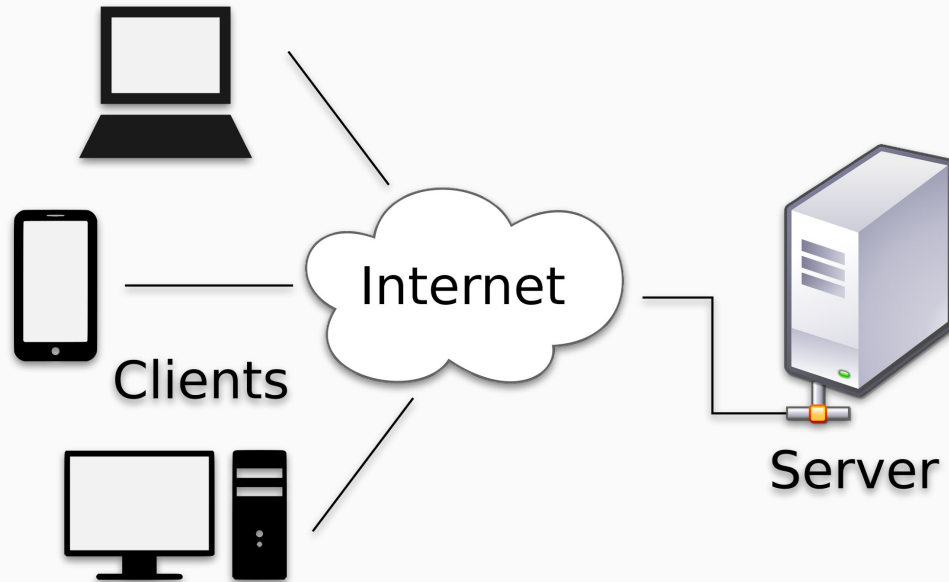


# ¿Qué es una aplicación web?

Una aplicación web es una aplicación informática **distribuida** cuya interfaz de usuario es accesible desde un cliente web, normalmente un navegador web (Browser/Navegador).

Estas aplicaciones son distribuidas gracias a que siguen una arquitectura **cliente-servidor**.

# Cliente-Servidor

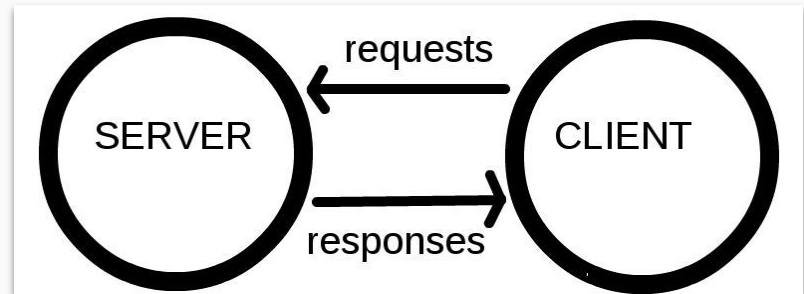


# Cliente-Servidor

En una aplicación que sigue la arquitectura **cliente-servidor**, se tienen dos grandes roles:

**Cliente:** quien quiere acceder a cierta información o realizar una operación en particular (realiza **requests/solicitudes**)

**Servidor:** quien tiene la capacidad de cómputo, procesamiento y acceso a la información (datos) para responder a la solicitud enviada por el cliente





# Completando la definición: el cliente

- Una aplicación web consiste de conjunto de páginas **HTML** junto a otros componentes JS, CSS y otros *assets* (imágenes, video, música, etc) que se transmiten por medio del protocolo **HTTP** del servidor al cliente y brindando distintas funcionalidades a un usuario final.
- Nuestro cliente (el **browser**) es quien se encarga de “entender” o renderizar el HTML y mostrarselo a los usuarios.

# Completando la definición: el servidor



Un servidor web se puede entender tanto desde software o hardware:

- desde **hardware**: es una computadora que almacena el software de un web server y los componentes (páginas HTML, CSS, JS, imágenes, etc) de una o varias páginas web.
- desde **software**: es una aplicación que controla la forma en la que se accede a ciertos recursos hospedados en una cierta máquina, y provee como mínimo, la posibilidad de interactuar mediante HTTP (aunque puede hablar otros protocolos).

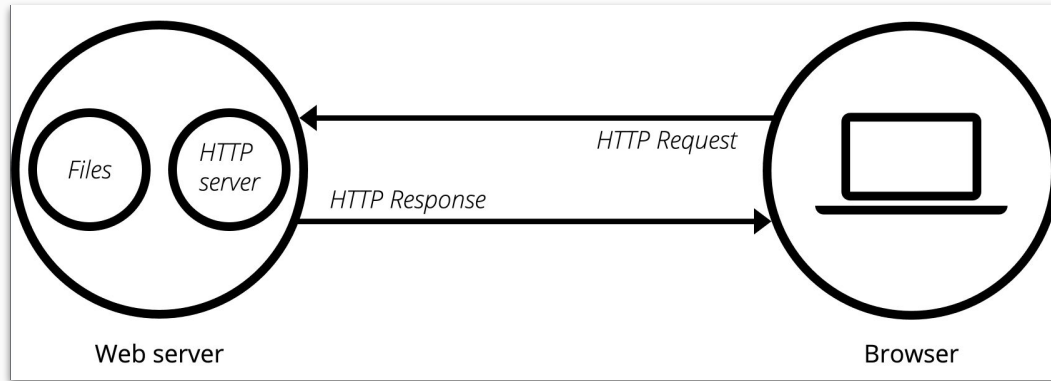
# Completando la definición: el servidor

Por ejemplo: Internet Information Server (IIS), es el servidor Web de Microsoft que corre sobre plataformas Windows. Los servicios que ofrece son: FTP, SMTP, NNTP y **HTTP/HTTPS**.

Otros: Apache HTTP Server, Apache TomCat, nginx, Glassfish, etc



# Completando la definición: el servidor



# Contenido estático vs dinámico

Un web server puede ser estático (solo sirve contenido estático, sin modificarlo) o **dinámico**.

El servidor que construiremos en este curso será un servidor de tipo **dinámico**.

Esto significa que consta de lo mismo que un servidor estático + software adicional, lo que por lo general llamamos “**application server**”, más alguna base de datos.

Les decimos dinámicos porque el “app server” actualiza los datos almacenados antes de enviárselos al cliente para que los renderice en el browser.

# DEMO: YouTube

*Entrar a YouTube y examinar qué pasa cuando cargamos una página (abrir inspector de código y ver requests en la pestaña Network)*

# AJAX

Asynchronous JavaScript And XML (AJAX).

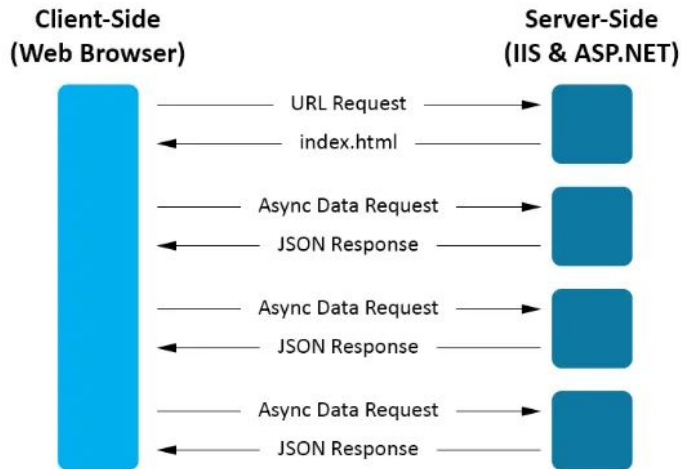
Basado en:

- HTML, CSS
- Browser document object model (DOM), JavaScript
- XML, JavaScript Object Notation (JSON)

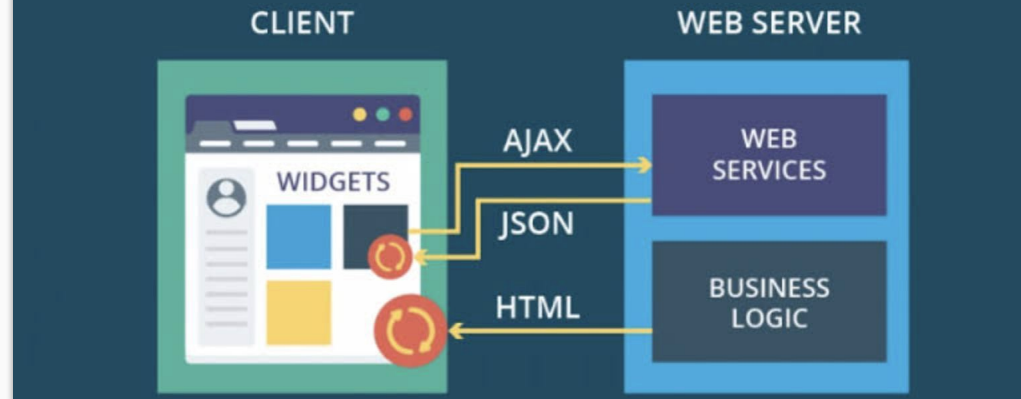
Permite ejecutar requests en forma asincrónica, obteniendo una respuesta sin tener que hacer un refresh total de toda la página; se envía al servidor solo lo necesario y se obtiene de este solo lo necesario

# Flujo de una aplicación web

## Native Web Application Flow



## Web Application Architecture





# Aplicaciones Web: características

Un servidor web se puede entender tanto desde software o hardware:

- Comunicación mediante **HTTP** sobre TCP/IP
- Paradas sobre el “lenguaje” **HTML**, además de CSS y JSS
- Arquitectura en **capas**
- Procesamiento en servidor
- Acceso a bases de datos
- Distintos tipos de clientes

HTTP

# HTTP: HyperText Transfer Protocol

- Uno de los protocolos más importantes de Internet
- HTTP define cómo los navegadores y los servidores Web interactúan entre sí
- Está basado en texto y es transmitido sobre conexiones TCP.
- Es un protocolo cliente-servidor.
- Se usa para ver memes y para tratar pacientes con cáncer

# HTTP: características

- 1) **Basado en texto:** no está encriptado - intercambia texto que podemos leer.
- 2) **Orientado a la conexión:**
  - a) Funciona sobre protocolo **TCP** (Transport Control Protocol) de modo conectado
  - b) Se establece una conexión punto a punto entre el cliente y el servidor
- 3) **Stateless (sin estado):** cada request/response entre el cliente y el servidor es independiente al anterior. No hay forma de mantener “un canal” de comunicación “abierto” permanentemente.

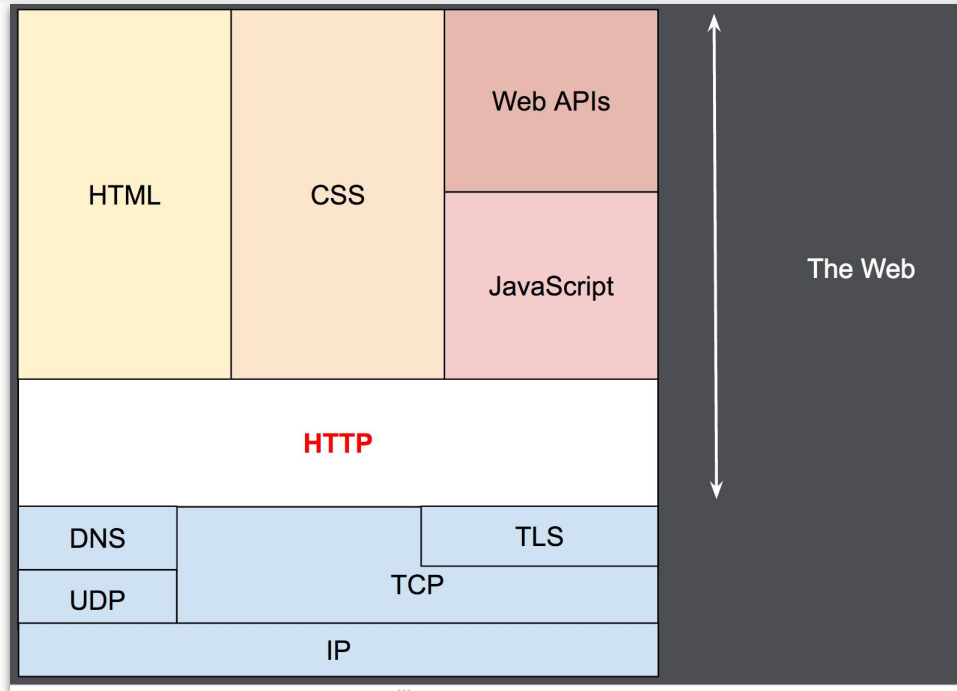
# HTTP: características

Sin embargo, podemos hacerlo “stateful”, gracias a las cookies. Nos permiten manejar la sesiones stateful (con estado).

4) Es extensible: gracias a los headers

5) Existe la variante HTTPS: (S = “Secure”) Utiliza en protocolo de seguridad SSL (Secure Socket Layer) para la autenticación y encriptación de la información intercambiada entre el cliente y el servidor.

# HTTP: características



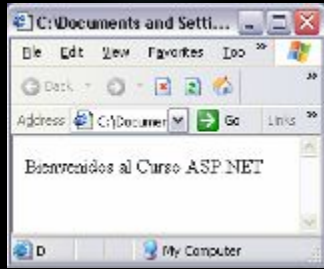
# Cliente

http://www.curso.com/inicio.html

Internet DNS



IP=66.45.26.25 Puerto: 80



**HTTP Request**

# Servidor



*Se abre conexión TCP*

www.curso.com

66.45.26.25:80

**HTTP Response**

*Index.html*

```
<html>
<body>
Bienvenidos al
Curso
</body>
</html>
```

# HTTP: estructura de una request HTTP

Una request HTTP consta de las siguientes partes:

- 1) **Método/Verbo**: el cual indica la acción en cuestión a desarrollar en la request. Estos pueden ser: **GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS**, etc.
- 2) **URL (Uniform Resource Locator)**: la ruta del recurso a obtener
  - a) **Path**: es lo que está después del primer “/” en la URL, sin considerar aspectos como el protocolo y el puerto
  - b) **Query**: son aquellos parámetros que se adjuntan después del path comenzando con un “?” y separados por “&”



# HTTP: estructura de una request HTTP

Un mensaje HTTP consta de las siguientes partes:

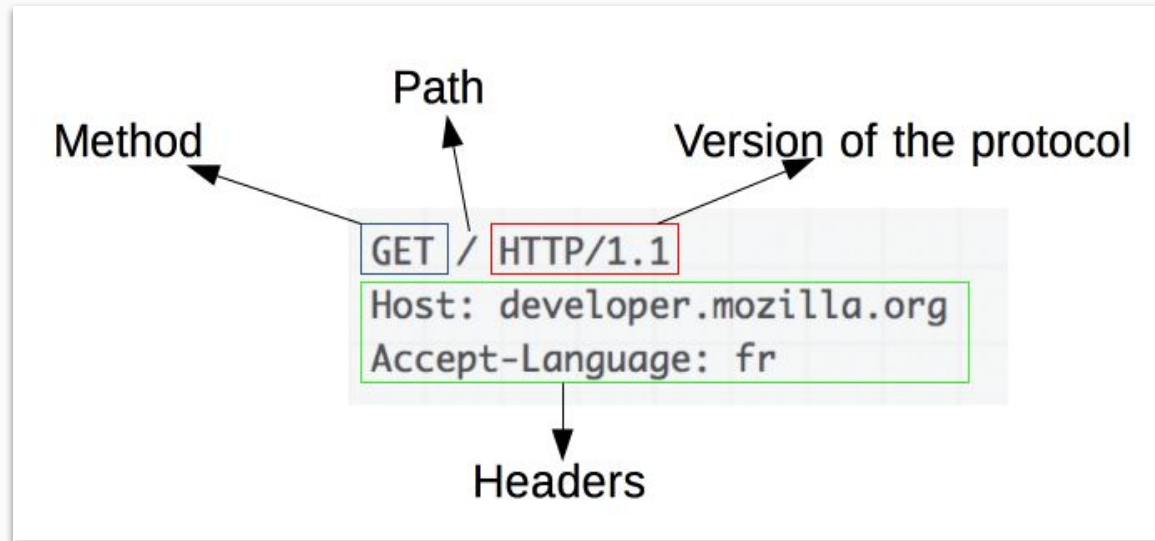
**3) Versión del protocolo:** HTTP/1.1, HTTP/2.0

**4) Body** (opcional): cuerpo del mensaje, es información que el cliente envía en siguiendo un cierto tipo de contenido (ej: JSON, XML, Text, etc).

**5) Headers** (opcionales): metadata del mensaje enviada en forma de “cabecera” que brinda información adicional al servidor. Sigue el formato **Key:Value**

# Ejemplo de una request HTTP

Request:



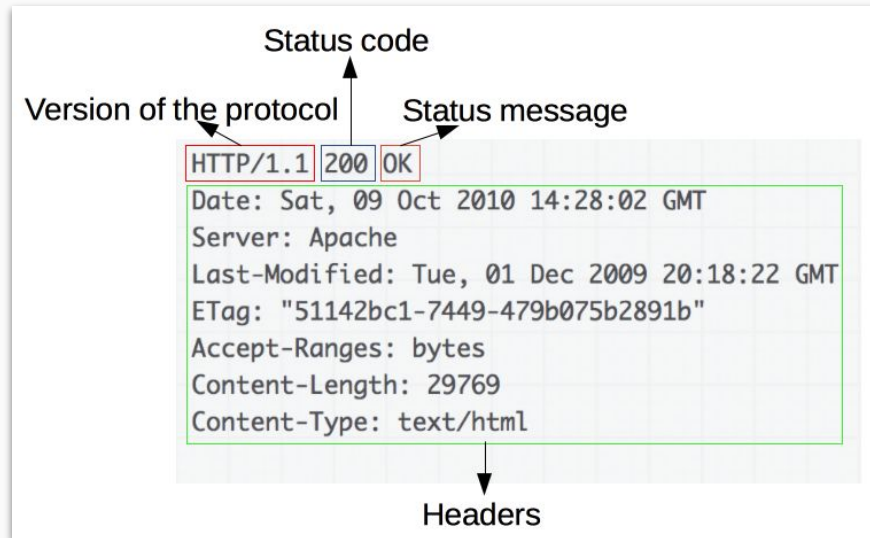
# HTTP: estructura de una response HTTP

Una response HTTP consta de:

- 1) **Código de estado (*status code*)**: indica el resultado de la operación en el servidor, mostrando si fue exitosa y porqué. Algunos de los más comunes: *200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error, etc.*
- 2) **Versión del protocolo HTTP**
- 3) **Opcionalmente**: body y headers (siguen mismos lineamientos que request)

# Ejemplo de una response HTTP

Response:



# Ejemplo de varias response HTTP

Status	Method	File	Domain	Type
200	GET	/	www.practicalecommerce.com	html
200	GET	colabs-css.css	www.practicalecommerce.com	css
200	GET	style.css?v=20170316-1212	www.practicalecommerce.com	css
200	GET	css?family=Fjalla+One Open+Sans+Condensed:300,700 Ope...	fonts.googleapis.com	css
200	GET	font-awesome.min.css	www.practicalecommerce.com	css
200	GET	ggicptch.css?ver=1.27	www.practicalecommerce.com	css
200	GET	highlight-default.css?ver=1.0	www.practicalecommerce.com	css
200	GET	wpp.css?ver=3.3.4	www.practicalecommerce.com	css
200	GET	jquery.js?ver=1.12.4	www.practicalecommerce.com	js
200	GET	jquery-migrate.min.js?ver=1.4.1	www.practicalecommerce.com	js
200	GET	jquery.cookie.js?ver=4.7.2	www.practicalecommerce.com	js
200	GET	jquery.sooperfish.js?ver=4.7.2	www.practicalecommerce.com	js
200	GET	jquery.flexslider-min.js?ver=4.7.2	www.practicalecommerce.com	js
200	GET	zero.js?ver=4.7.2	www.practicalecommerce.com	js
200	GET	default.css	www.practicalecommerce.com	css
200	GET	shortcodes.css	www.practicalecommerce.com	css
200	GET	custom.css	www.practicalecommerce.com	css
200	GET	social-icon.png	www.practicalecommerce.com	png
200	GET	search-icon.png	www.practicalecommerce.com	png
200	GET	PEC_hires_no_tag_trans.png	www.practicalecommerce.com	png
200	GET	3-Ways-to-Use-the-AdWords-Keyword-Planner-288x196.jpg	www.practicalecommerce.com	jpeg
200	GET	Boost-Site-Search-Results-with-Unique-Trendy-Keywords-...	www.practicalecommerce.com	jpeg
200	GET	4-Areas-for-Email-Marketing-Growth-288x196.jpg	www.practicalecommerce.com	jpeg
200	GET	5-Shipping-Tips-for-Ecommerce-Beginners-288x201.jpg	www.practicalecommerce.com	jpeg
200	GET	SEO-How-to-Part-8-Architecture-and-Internal-Linking-288x...	www.practicalecommerce.com	jpeg
200	GET	thumbnail-4-288x200.jpg	www.practicalecommerce.com	jpeg
200	GET	Legal-Issues-for-Membership-based-Ecommerce-Sites-288...	www.practicalecommerce.com	jpeg
200	GET	For-Content-Marketing-Ideas-Try-BuzzSumo-288x200.png	www.practicalecommerce.com	png

# URL en un mensaje HTTP

Se usan para referenciar a un recurso que se desea acceder. Ejemplo:

<https://www.youtube.com/watch?v=unJDABuKWzA>

Protocolo://      dominio      / path      ?      query

Los diferentes parámetros se separan por “&”, los espacios en blanco se sustituyen por “+”, y los caracteres especiales se representan con “%xx” donde “xx” es el código ASCII en hexadecimal del carácter.

# URL en un mensaje HTTP

PROTOCOL

PORT

QUERY

<http://www.prolificidea.com:8080/some/path?a=x&b=y>

DOMAIN

RESOURCE

# Códigos de estado HTTP

Los agrupamos en 5 categorías, siendo esta definida por el primer dígito del código. El estándar RFC de HTTP define:

- **1xx informational response** – se recibió la request y se proseguirá a procesar
- **2xx successful** – la request se recibió, entendió, procesó y aceptó correctamente
- **3xx redirection** – indica que se deben realizar acciones adicionales para completar la request.
- **4xx client error** – la request contiene errores de sintaxis del cliente o no puede ser completada por motivos ajenos al servidor
- **5xx server error** – el servidor falló al completar la request



# Códigos de estado HTTP: más ejemplos

HTTP Status Codes		
<b>Level 200 (Success)</b> 200 : OK 201 : Created 203 : Non-Authoritative Information 204 : No Content	<b>Level 400</b> 400 : Bad Request 401 : Unauthorized 403 : Forbidden 404 : Not Found 409 : Conflict	<b>Level 500</b> 500 : Internal Server Error 503 : Service Unavailable 501 : Not Implemented 504 : Gateway Timeout 599 : Network timeout 502 : Bad Gateway

APIs

# La Web y las URIs

artistas/The Beatles/Abbey Road



artistas/{artista}/{album}

artistas/The beatles?album=Abbey Road



artistas/{artista}?album={album}

# Formato de datos

Encabezados HTTP indican:

- Formatos de datos aceptados (Request)
- Formato de los datos en la respuesta (Response)

Encabezados comunes

- Accept (Request), Content-Type (Response)

Algunos tipos de encabezados:

- text/html, text/css,
- image/gif, image/jpeg,
- application/atom+xml, application/json,
- video/mp4

# Operaciones de una API HTTP

Todas las operaciones de una API se hacen mediante los verbos HTTP:

- **GET** = “dame cierta información” (Retrieve)
- **POST** = “acá va información nueva” (Create)
- **PUT** = “acá va una actualización” (Update)
- **DELETE** = “borrá esta información” (Delete)


Nuestra API será CRUD (Create, Retrieve, Update, and Delete)




# Web Service: ¿qué es?

El término tiene distintos significados dependiendo del contexto

Para nosotros será una forma de **exponer un API de manera independiente** de la **tecnología** en un nodo de red.



Esto significará que un cliente podrá ejecutar un método o función **remota** y obtener un resultado, sin estar atado a una plataforma o vendedor específico.



# Web Service: mecanismos

Los tres mecanismos más populares para implementar servicios web son:

- **POX** (plain old XML)
- **SOAP** (Simple Object Access Protocol)
- **REST (Representational State Transfer)**

Los tres proveen mecanismos para acceder a información o invocar operaciones con distintos grados de rigurosidad.



# Método GET

Se recomienda que se utilice solo para la solicitud de datos (obtención de recursos)

Es una operación **idempotente** (podemos consultar muchas veces y obtener el mismo resultado).

Un HTTP GET nunca debería ser utilizado para modificar datos (es “seguro”)

Ej: **GET /index.html HTTP/1.1**

**GET /api/users/12345**



# Método POST

Se recomienda que se utilice para la creación de recursos

No es una operación **idempotente** (dos llamadas consecutivas dan resultados diferentes).

Se recomienda utilizar para toda aquella operación cuyo resultado no sea idempotente.

Ej: **POST /api/users** (llamadas repetidas a /users siempre van a estar creando un usuario diferente, lo que da un resultado diferente)

# Método PUT

Se utiliza para actualizar un recurso

Es una operación **idempotente**

Originalmente se usaba para crear recursos, a veces se sigue usando como POST

Ej: **PUT /api/users/12345**

# Método DELETE

Se utiliza para eliminar un recurso

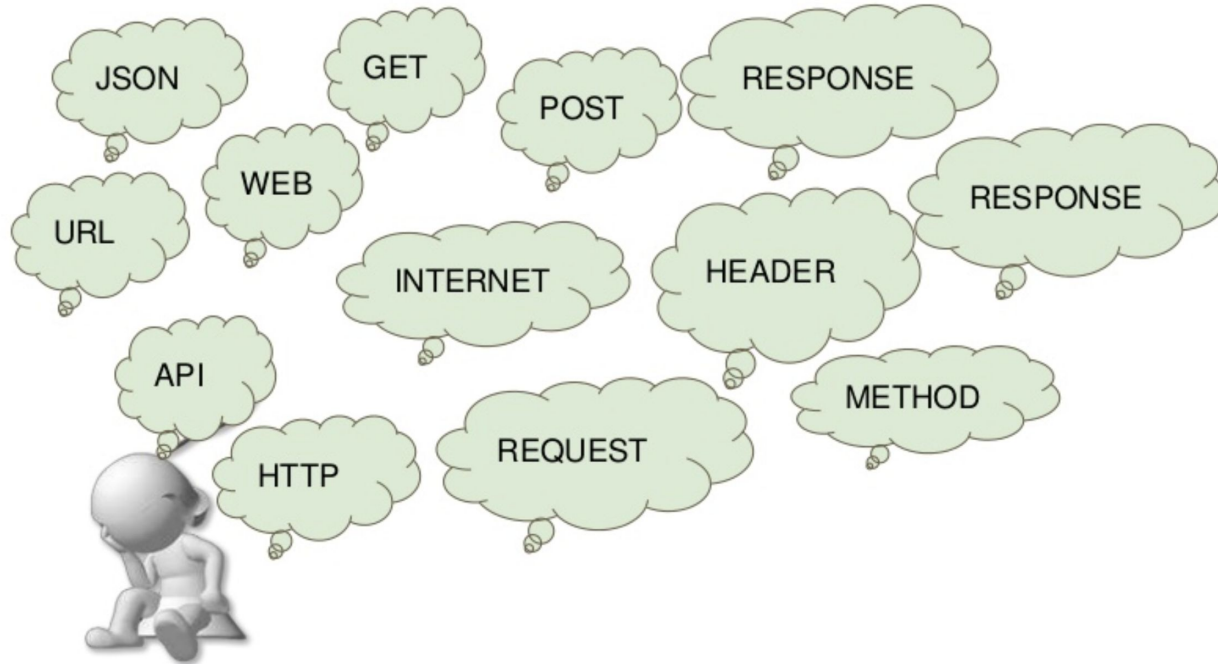
Es una operación **idempotente según** el RFC de HTTP

Sin embargo, hay cierta ambigüedad. ¿Qué pasa si hacemos un delete de un recurso y luego hacemos otro consecutivamente? La única forma es “marcar a borrar”.

Ej: **DELETE /api/users/12345**

REST

# REST



# REST: introducción

**RE**presentational **State** Transfer

Es un estilo arquitectónico que define un conjunto de restricciones y características para la construcción de APIs. **No es un estándar.**

Le pone orden (arquitectura) a todos los conceptos relacionados al mundo web y de las aplicaciones y servicios web.

Fue presentado en 2008 en [una tesis doctoral de Roy Fielding.](#)

# REST: conceptos básicos

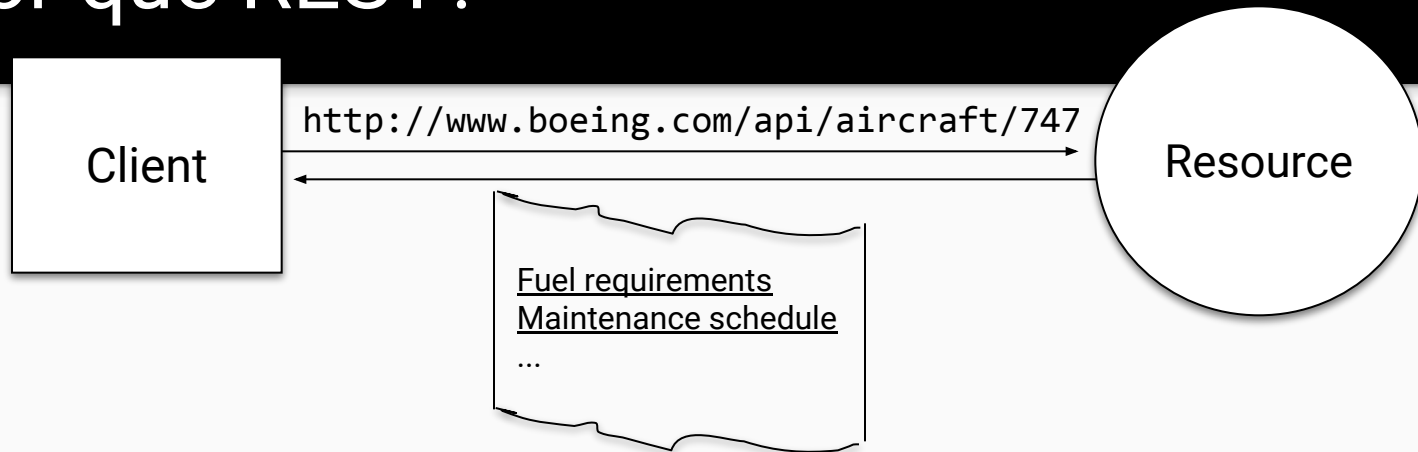
REST debe estar basada sobre una forma (protocolo) de comunicación.  
Ejemplo: HTTP.

Los recursos se acceden mediante una interfaz común basada sobre un conjunto de métodos. Por ejemplo: los métodos HTTP.

En REST el servidor expone la manipulación de un conjunto de recursos mientras que los clientes se comunican con él para poder usarlos.

¿Cómo identificamos los recursos? **URIs**.

# ¿Por qué REST?



El cliente referencia un recurso web utilizando una URL, al tiempo que se devuelve una **representación** del mismo (por ejemplo como html, json, xml, etc).

La representación deja al cliente en un nuevo **estado**.

Cuando el cliente selecciona un hiperlink en Boeing747.html accede a un nuevo recurso, el que lo deja en un nuevo estado, por lo que se dice que cada recurso **transfiere** al cliente de un estado al otro.



# REST: restricciones

Como ya dijimos, define **seis** restricciones:

- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on Demand

# REST: Uniform Interface

Define una interfaz para la interacción cliente-servidor. Permite simplificar y desacoplar la arquitectura, lo cual permite que cada parte evolucione independientemente.

# REST: Uniform Interface

## Principios de una interfaz uniforme:

- **Basarse en recursos**
- **Manipular recursos** a través de sus representaciones: cuando un cliente maneja una representación de un recurso, contiene la metadata suficiente para modificar o modificar un recurso en el servidor.
- **Mensajes auto-descriptivos:** cada mensaje contiene la información suficiente para indicar como procesar el mensaje. Por ejemplo, cada mensaje debe especificar cómo se parsea (ej: el Mime Type). También pueden explicar si se pueden cachear.

Todo esto es permitido por HTTP.

REST es orientado a recursos.

---

# REST: URIs

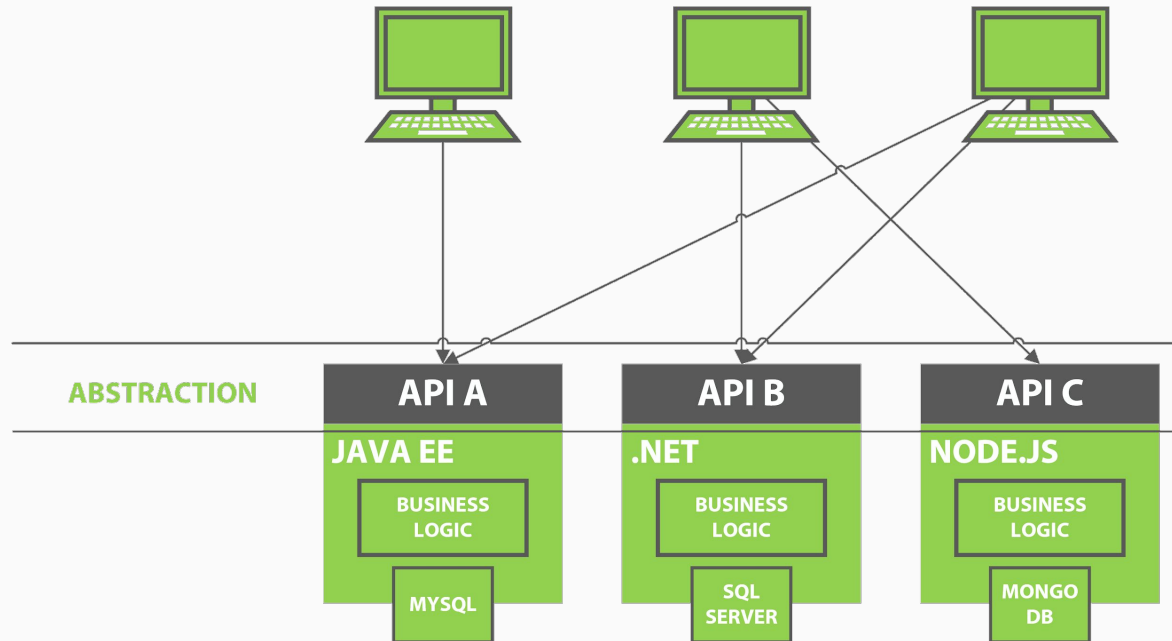
**Basados en la URI:** Uniform Resource Identifier

Cada parte de una URI se mapea de forma semántica al dominio, compuesto de recursos. La idea es enviar una representación de esos recursos (que vienen por ejemplo de una base de datos),

Al usar una representación común (ej: JSON), en REST no importa cómo están almacenados los recursos ni la tecnología con la que esté construida la API.

**REST es orientado a los recursos**, las URIs bien definidas serán cruciales para que quienes consuman la API entiendan como manipular dichos recursos.

# REST: URIs

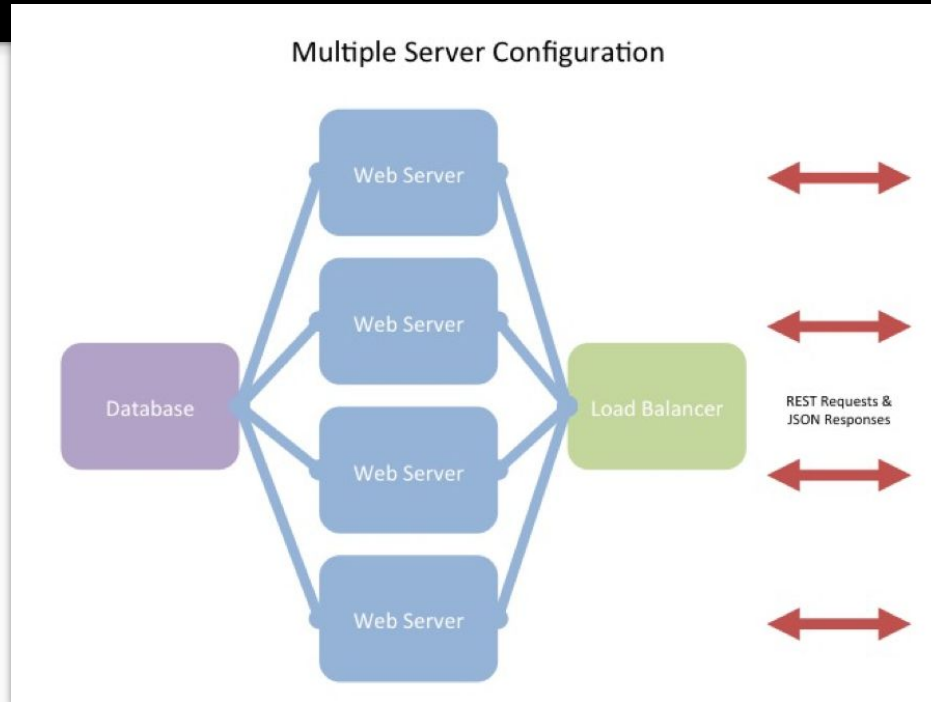


# REST: Stateless

## REST no maneja estado (no se maneja estado a nivel del servidor)

Significa que toda la información (el estado) para interpretar una solicitud, está contenida en la request en sí misma (en su URI, en el body, en headers, en query string, etc).

# REST: Stateless





REST es Stateless (a nivel del cliente)

---

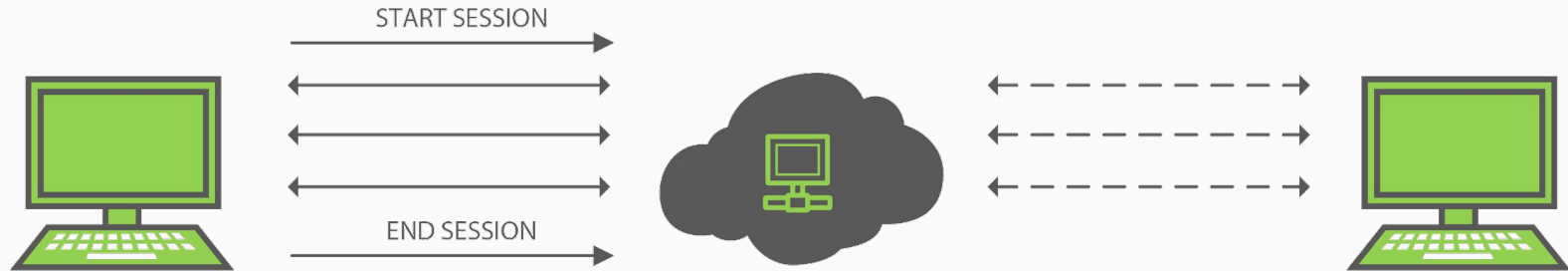
# REST: Stateless -> + escalabilidad

Esto permite hacer **sistemas escalables** y distribuidos más fácil. Los componentes stateless pueden ser fácilmente redesplegados si algo falla, y se pueden escalar individualmente si algo falla.

Esto es gracias a qué cualquier request puede ser dirigida a cualquier instancia/nodo/componente

Statelessness permite entonces que los servidores no tengan que mantener, actualizar o comunicar el estado de una sesión. Adicionalmente, los balanceadores de carga no se tienen que asegurar la “afinidad de una sesión” en sistemas stateless.

# REST: Stateless -> + escalabilidad



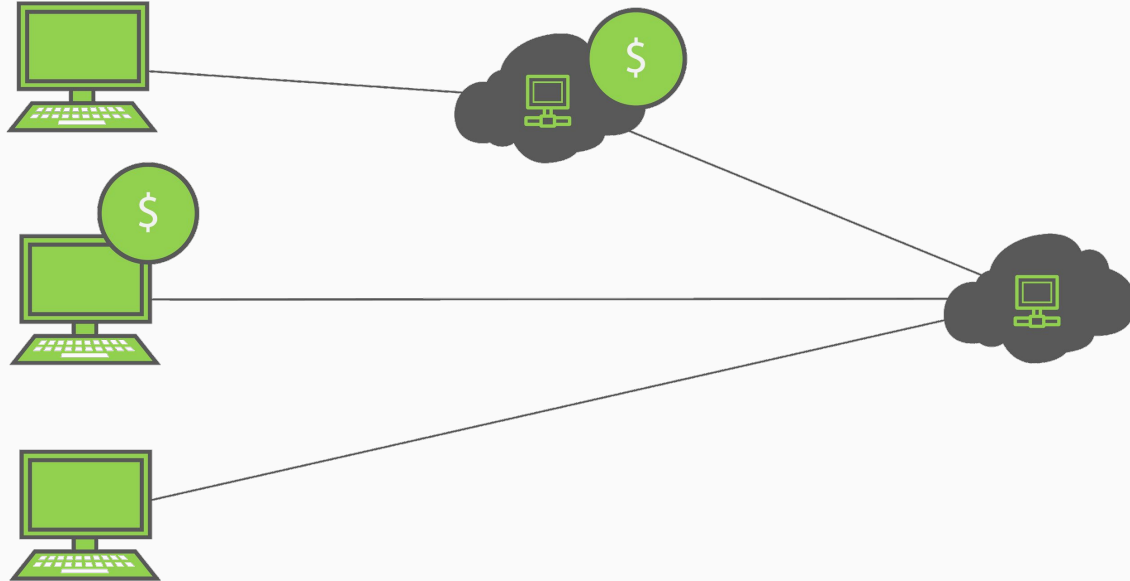
# REST: Cacheable

Los clientes pueden cachear las respuestas.

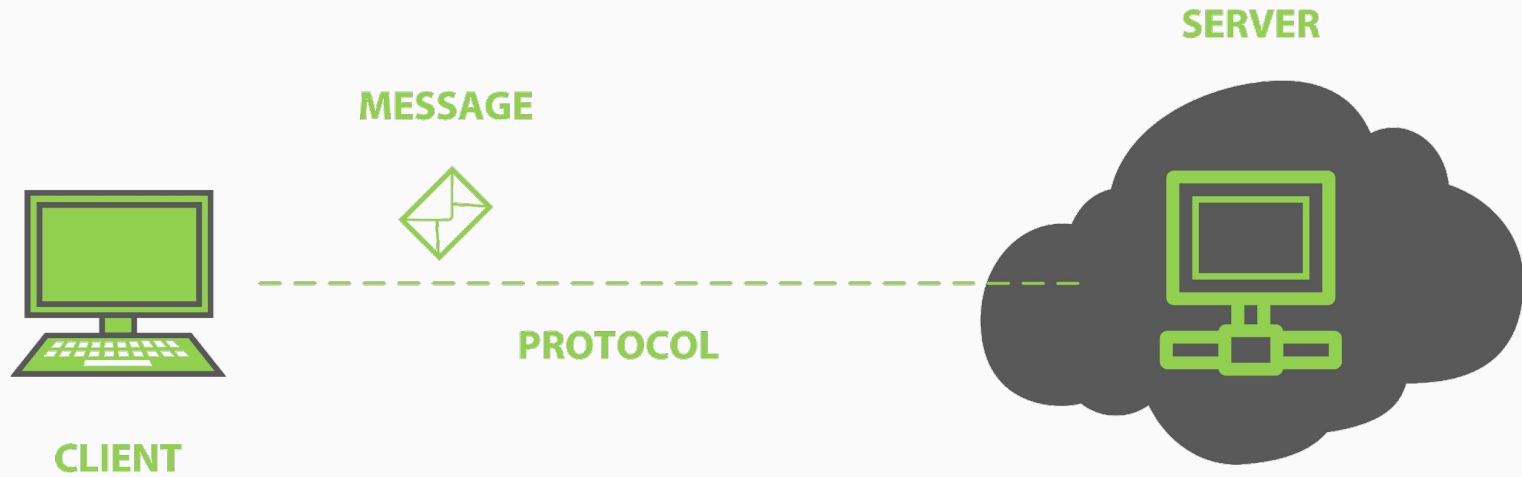
Por ello, las respuestas deben definirse de alguna forma como “cacheables” o no, para evitar que los clientes reusen estado o datos en requests subsiguientes cuando no deberían hacerlo.

Un caching bien manejado, permite eliminar parcialmente o totalmente algunas interacciones cliente servidor, lo que mejora la escalabilidad y la performance.

# REST: Cacheable



# REST: Cliente-Servidor



# REST: Cliente-Servidor

Una API REST debe ser cliente-servidor. La interfaz uniforme permite separar clientes de servidores.

Esto, como ya dijimos, permite que los clientes no se preocupen de cómo los datos son almacenados, lo cual es interno al servidor. Por ende, el código del cliente es más portable.

Del mismo modo, al servidor no le preocupa manejar la interfaz de usuario o el estado del usuario, de manera que los servidores son más escalables.

Clientes y servidores pueden ser  
reemplazados y desarrollados  
independientemente, siempre y cuando  
la interfaz no se altere





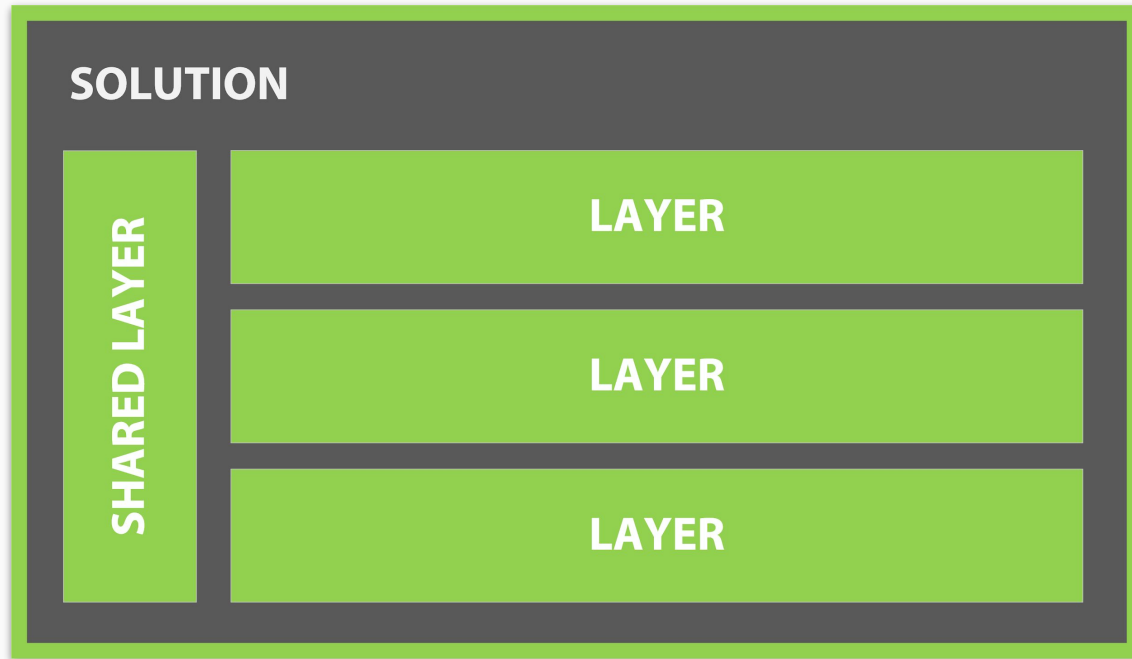
# REST: Tiered y Layered system

Un cliente no puede de forma sencilla decir si está conectado al servidor final o a un intermediario (tier)

Los servidores intermediarios permiten mejorar la escalabilidad a través de habilitar el balanceo de carga y cachés compartidos. Cada capa también puede exigir políticas de seguridad.

A su vez, se basa en una layered architecture (arquitectura en capas):

# REST: Layered-System



# REST: restricciones - ¿dudas?

Como ya dijimos, define **seis** restricciones:

- ~~*Uniform Interface*~~
- ~~*Stateless*~~
- ~~*Cacheable*~~
- ~~*Client Server*~~
- ~~*Layered System*~~
- ~~*Code on Demand*~~

# REST: aclaraciones

- REST no es un estándar.
  - No aparece en las especificaciones de W3C.
- REST es simplemente una arquitectura o estilo arquitectónico.
- REST promueve el uso de otros estándares:
  - HTTP
  - URL
  - JSON/XML/HTML/GIF/JPEG/etc. (Representación de recursos)
  - text/xml, text/html, image/gif, image/jpeg, etc. (Resource Types, MIME Types)

# REST: aclaraciones

Todas las interacciones entre el cliente y el servicio se realizan mediante operaciones HTTP. Se aprovecha la interface CRUD.

- Obtener información (HTTP GET)
- Crear Información (HTTP POST)
- Actualizar Información (HTTP PUT)
- Eliminar Información (HTTP DELETE)

# REST: uso de verbo HTTP

Método HTTP	Operación CRUD	Descripción
POST	INSERT	Agrega un recurso nuevo
PUT	UPDATE	Actualiza un recurso existente
GET	READ	Obtiene un recurso. El recurso no debería cambiar (idempotente)
DELETE	DELETE	Borra un recurso

# Diseño de APIs REST: buenas prácticas

# REST: URLs

- Las urls son el principal elemento para definir la “usabilidad” de la API.
- Una regla sugerida: solo 2 urls (endpoints) por recurso
- **Sustantivos SI, verbos NO**
  - HTTP para operar sobre las colecciones



# REST: sustantivos

Usar sustantivos en plural y no verbos en una URI. No mezclar singular y plural.

Objetivo	Método HTTP	Incorrecto ❌	Correcto ✅
Crear un nuevo usuario	POST	/insertUser	/users
Obtiene lista de usuarios	GET	/getAllUsers	/users
Obtiene el balance de un usuario	GET	/getUserBalance	/users/10/balance
Borrar un usuario	DELETE	/deleteUser o /user/10/delete	/users/10

# REST: más ejemplos

Resource	POST create	GET read	PUT update	DELETE delete
<b>/dogs</b>	Create a new dog	List dogs	Bulk update dogs	Delete all dogs
<b>/dogs/1234</b>	Error	Show Bo	If exists update Bo  If not error	Delete Bo

# REST: asociaciones

Podemos usar subrecursos para denotar asociaciones entre recursos. Ej:

- *GET /car/99/drivers* (obtengo todos los conductores asociados al auto 99)
- *GET /car/99/drivers/4* (obtengo el conductor 4 del auto 99)

Nunca extender las URIs **más de tres niveles**: **recurso/id/subrecurso**

```
GET /owners/5678/dogs
```

```
POST /owners/5678/dogs
```

# REST: query strings

Los podemos usar para llevar la complejidad a la derecha del “?”

**Muy útiles en búsquedas que usan filtros.**

```
GET /dogs?color=red&state=running&location=park
```

# REST: manejo de errores

## Facebook

HTTP Status Code: 200

```
{"type" : "OAuthException", "message": "(#803) Some of the aliases you requested do not exist: foo.bar"}
```

## Twilio

HTTP Status Code: 401

```
{"status" : "401", "message": "Authenticate", "code": 20003, "more info": "http://www.twilio.com/docs/errors/20003"}
```

## SimpleGeo

HTTP Status Code: 401

```
{"code" : 401, "message": "Authentication Required"}
```

# REST: códigos de estado

## Google GData

200 201 304 400 401 403 404 409 410 500

## Netflix

200 201 304 400 401 403 404 412 500

## Digg

200 400 401 403 404 410 500 503

# REST: ¿cuáles códigos usar?

- 200 - OK
- 400 - Bad Request
- 500 - Internal Server Error

- 201 - Created
- 304 - Not Modified
- 404 - Not Found
- 401 - Unauthorized
- 403 - Forbidden

```
{"developerMessage" : "Verbose, plain language description of  
the problem for the app developer with hints about how to fix  
it.", "userMessage": "Pass this message on to the app user if  
needed.", "errorCode" : 12345, "more info":  
"http://dev.teachdogrest.com/errors/12345"}
```

# REST: Notación de atributos

## **Twitter**

"created\_at": "Thu Nov 03 05:19:38 +0000 2011"

## **Bing**

"DateTime": "2011-10-29T09:35:00Z"

## **Foursquare**

"createdAt": 1320296464



# REST: ¿respuestas sin recursos?

```
/convert?from=EUR&to=CNY&amount=100
```

Se deben documentar como excepciones a la regla

# REST: no usar GET para alterar estado

El GET debe ser idempotente.

Debemos usar POST, PUT y DELETE para alterar estado.

No usemos requests GET con query parameters para alterar estado:

- *GET /users/711?activate o;*
- *GET /users/711/activate*

# REST: no usar GET para alterar estado

El GET debe ser idempotente.

Debemos usar POST, PUT y DELETE para alterar estado.

No usemos requests GET con query parameters para alterar estado:

- *GET /users/711?activate o;*
- *GET /users/711/activate*

# REST: usar headers HTTP para indicar formato

Tanto el cliente como el servidor tienen que indicar el formato se usa para la comunicación.

Este formato lo especificamos a través de headers HTTP:

- **Content-Type:** define el formato del mensaje
- **Accept:** define la lista de formatos de respuesta que se consideran aceptables por el cliente

# REST: el versionado es importante

Siempre se estila desarrollar APIs que estén versionadas.

Por lo general se suele usar un simple número ordinal (ej: v1, v2, etc).

Se usa la URL de la API comenzando con la letra “v”:

***/api/v1/users***

¿Dudas?

# Intro

What's this presentation about? Use this slide to introduce yourself and give a high level overview of the topic you're about to explain.

# First point

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua

Incididunt ut labore et dolore

Consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua



## Second point

Lorem ipsum dolor sit amet,  
consectetur adipiscing elit, sed do  
eiusmod tempor incididunt ut labore et  
dolore magna aliqua

Incididunt ut labore et dolore

Consectetur adipiscing elit, sed do  
eiusmod tempor incididunt ut labore et  
dolore magna aliqua

XX%

Use this slide to show a major stat. It can help enforce the presentation's main message or argument.

# Final point

A one-line description of it



“This is a super-important quote”



- From an expert

This is the most  
important takeaway  
that everyone has to  
remember.

# Thanks!

Contact us:

Your Company  
123 Your Street  
Your City, ST 12345

[no\\_reply@example.com](mailto:no_reply@example.com)

[www.example.com](http://www.example.com)

