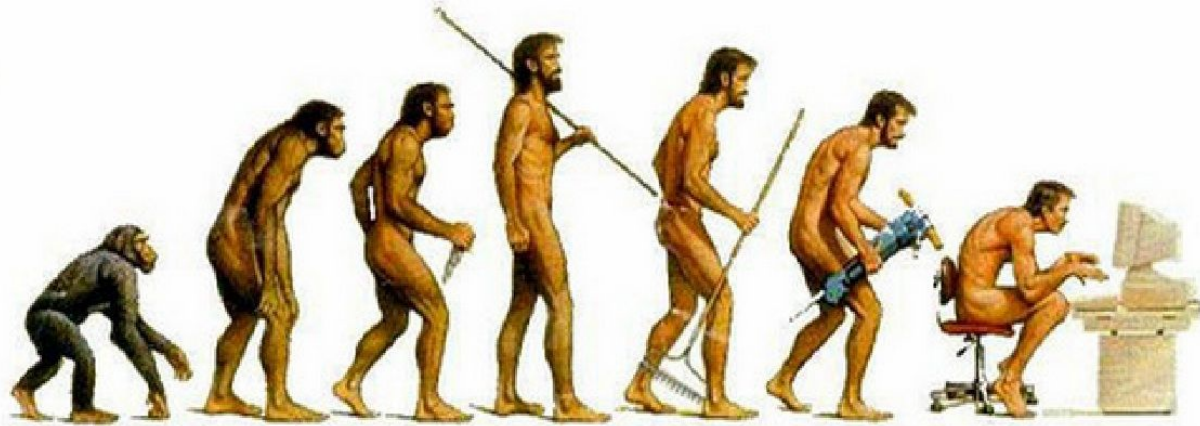


Web Services

SSR vs CSR - API REST

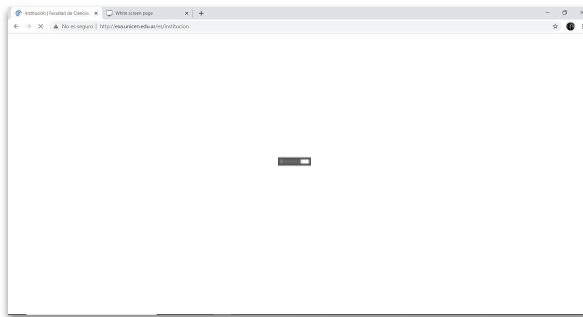
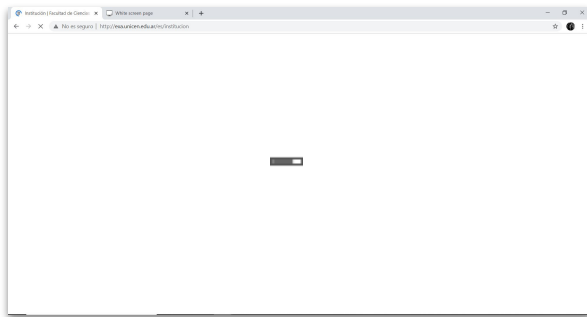


EVOLUCIÓN DE LA WEB

En los últimos 20 años, la web ha evolucionado drásticamente, pasando de documentos estáticos con unos cuantos estilos e imágenes, a complejas y dinámicas aplicaciones web.

Server Side Rendering (SSR)

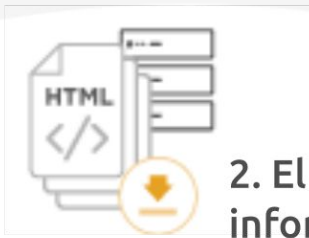
Todos los recursos del sitio o app están alojados o son creados **dentro del servidor**. Cuando un usuario realiza una solicitud, el servidor envía el documento HTML final completo para que el browser lo muestre.



SSR



1. Sitio requerido por el usuario



2. El servidor busca toda la información que necesita y genera el HTML completo

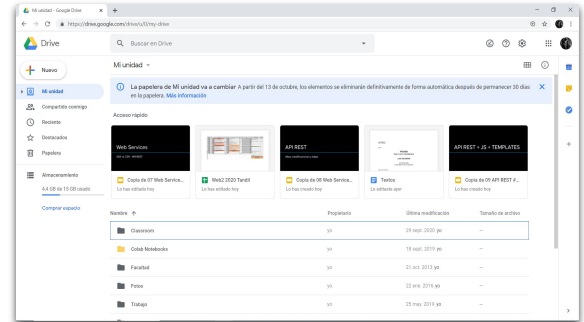
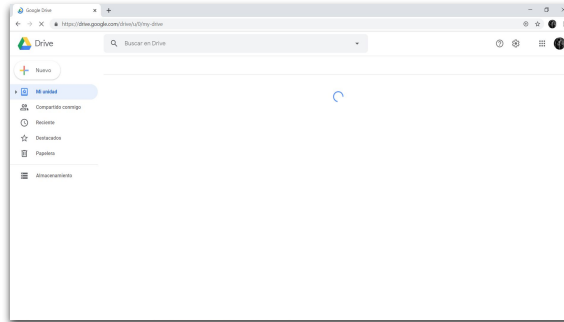
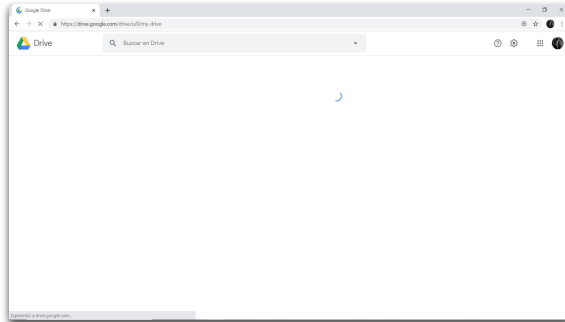


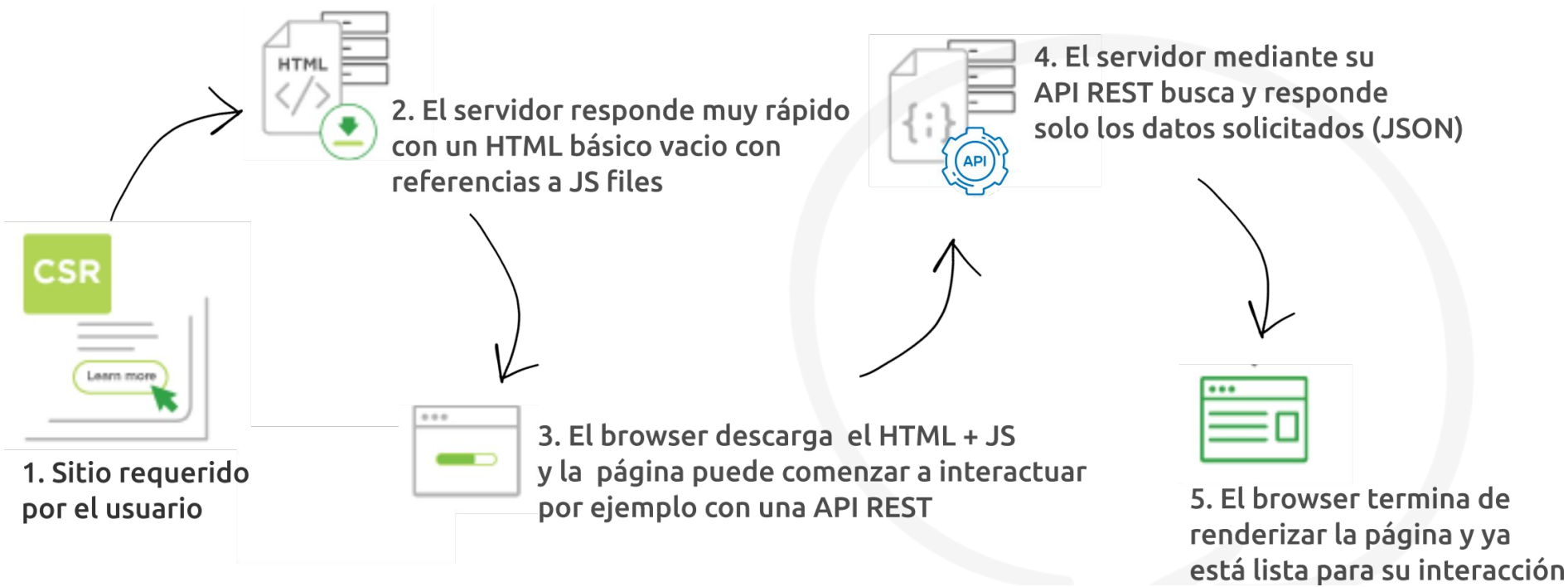
3. El browser descarga la página y ya está lista para su interacción

SERVER SIDE RENDERING (ENFOQUE TRADICIONAL)

Client Side Rendering (CSR)

En lugar de obtener todo el contenido del documento HTML desde el servidor, se recibe un documento HTML básico vacío y el **render se completa** del lado del cliente usando JavaScript.



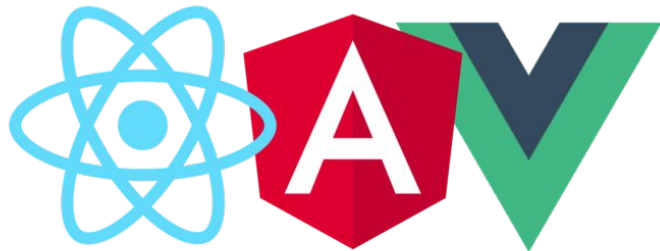


CLIENT SIDE RENDERING

Client Side Rendering

Client-side-rendering (CSR) implica renderizar las páginas directamente en el browser usando javascript. Toda la lógica, templates, ruteo y obtención de información es manejada en el lado del cliente.

- *Usado para hacer SPA*
- *Muchos frameworks disponibles*



SSR VS CSR



SSR

- Optimizado para SEO
- Página inicial carga más rápido
- Buena para sitios semi-estáticos con baja interacción

VENTAJAS

- La página completa se recarga
- Mayor transferencia por solicitud
- Pobre UX

DESVENTAJAS

CSR

- Buena UX
- Renderización rápida después de la carga inicial
- Excelente para aplicaciones web

VENTAJAS

- Dificulta SEO si no se implementa correctamente
- Página inicial carga más lento
- Requiere librería externa para utilizarlo fácil

DESVENTAJAS

Enfoque híbrido:

Combina las dos técnicas para lograr una mejor usabilidad desde el punto de vista de la performance y la respuesta al usuario.

- Es lo que vamos a hacer nosotros en el TPE.
- Por ej, Facebook utiliza este enfoque.

**Las SPA tiene que obtener
los datos desde algún lugar.
¿Desde donde lo hacen?**

¿Qué son los web services?

Son **componentes** de una aplicación específicos para **intercambiar información** entre aplicaciones.

Permiten la **interoperabilidad** entre distintas plataformas y sistemas por medio de protocolos estándar y abiertos.



<https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice>

Web Services (WS)

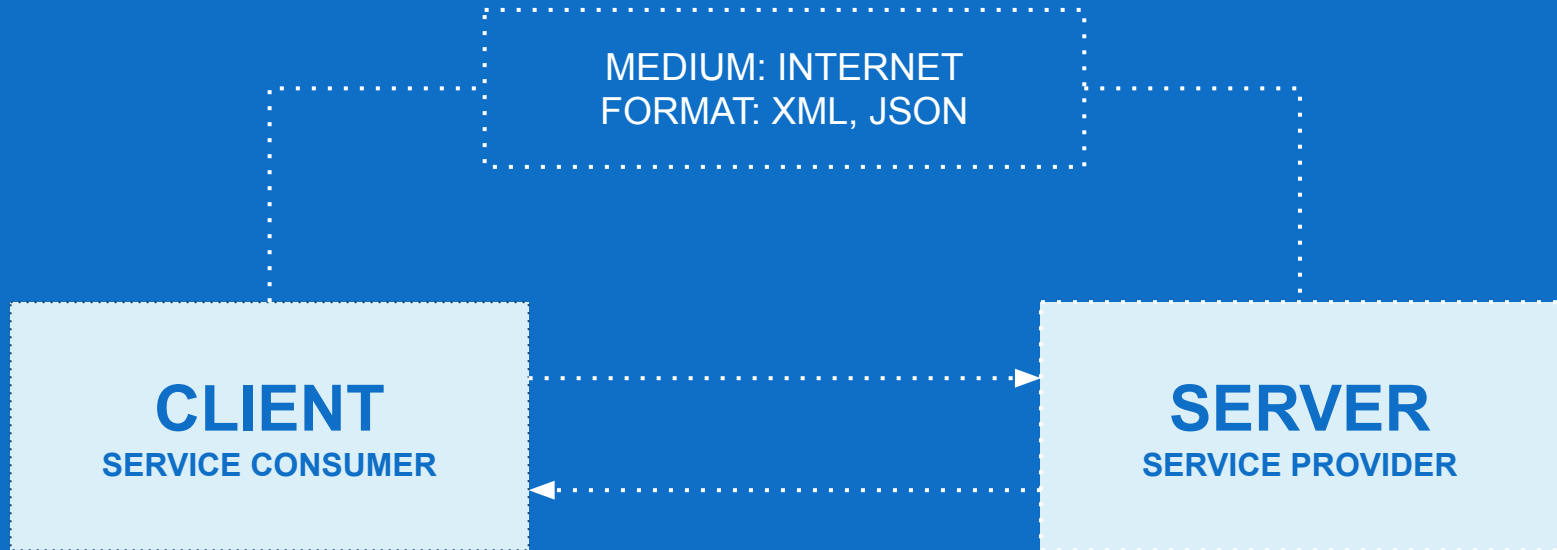


Actualmente la mayoría de los sistemas utiliza servicios web.

- Los sistemas se empiezan a comunicar entre ellos.
- Comparten información.

WEB SERVICES

DISPONIBLES A TRAVÉS DE INTERNET



Servicios web

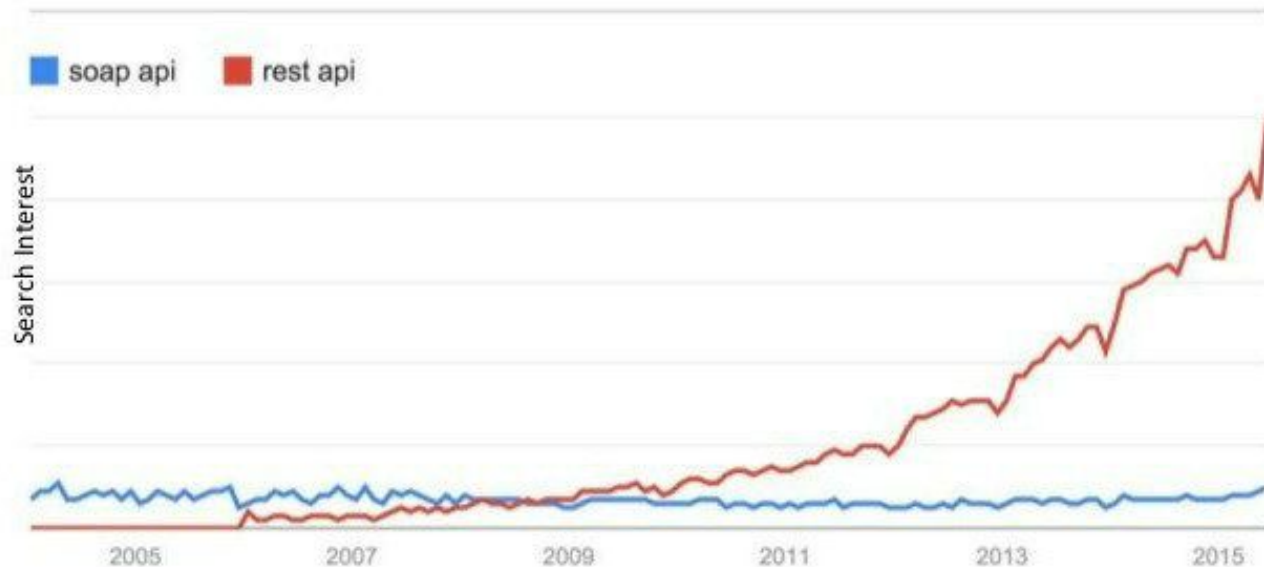
Existen diferentes **protocolos** y **arquitecturas** de servicios.

Los principales son:

- **SOAP:** Simple Object Access Protocol
muy usado en sistemas corporativos
- **REST:** Representational State Transfer
muy usado en la internet
- **GraphQL:** Query language for APIs
arquitectura alternativa a REST

REST vs SOAP

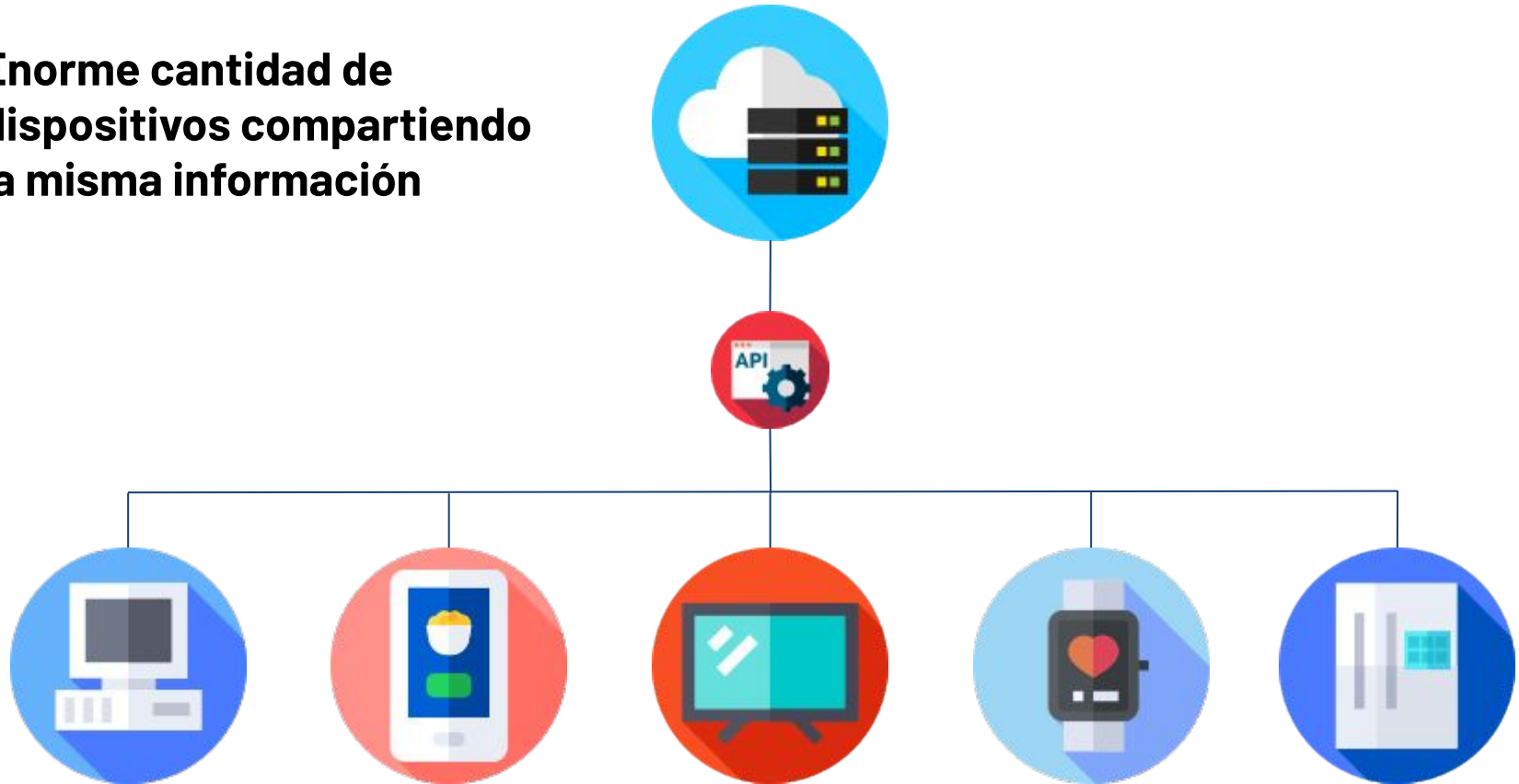
REST WS Popularity



Source: Google Trends, June 10, 2015

Servicios WEB - API Rest

Enorme cantidad de dispositivos compartiendo la misma información



Ejemplos de WS

Catálogo de API's públicas del Estado Nacional

<https://www.argentina.gob.ar/onti/software-publico/catalogo/apis>

Open Weather API

<https://openweathermap.org/api>

Toonify API

<https://deepai.org/machine-learning-model/toonify>

API REST

API REST

REPASO

REST (REpresentational State Transfer)

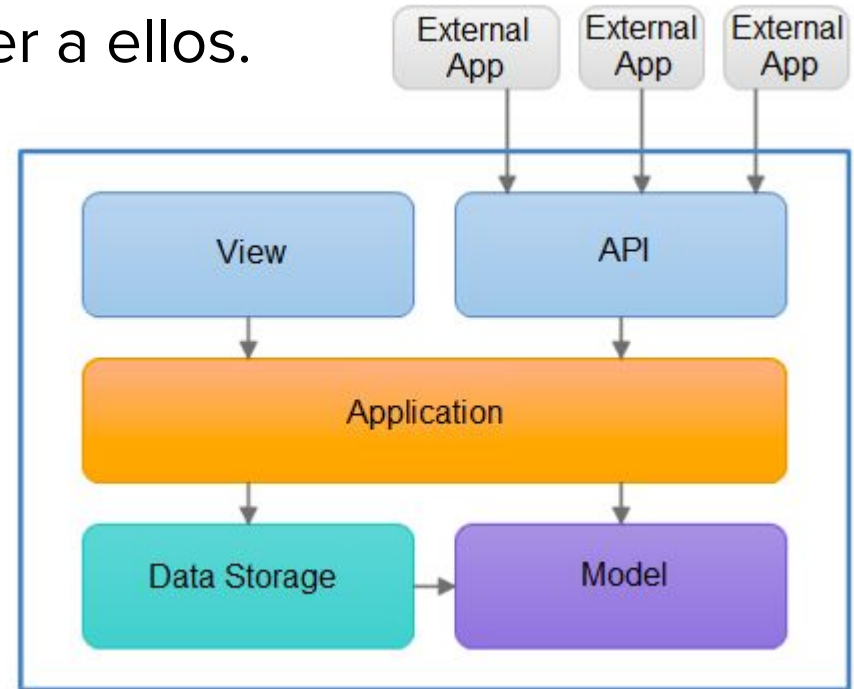
Arquitectura de desarrollo web que **se apoya totalmente en el protocolo HTTP**. Proporciona una **API** que utiliza cada uno de los métodos del protocolo.

- Es el tipo de arquitectura más natural y estándar para crear APIs para servicios orientados a Internet.
- Una URI representa un **recurso** al que se puede acceder o modificar mediante los métodos del protocolo HTTP (POST, GET, PUT, DELETE).

API: Application Programming Interface

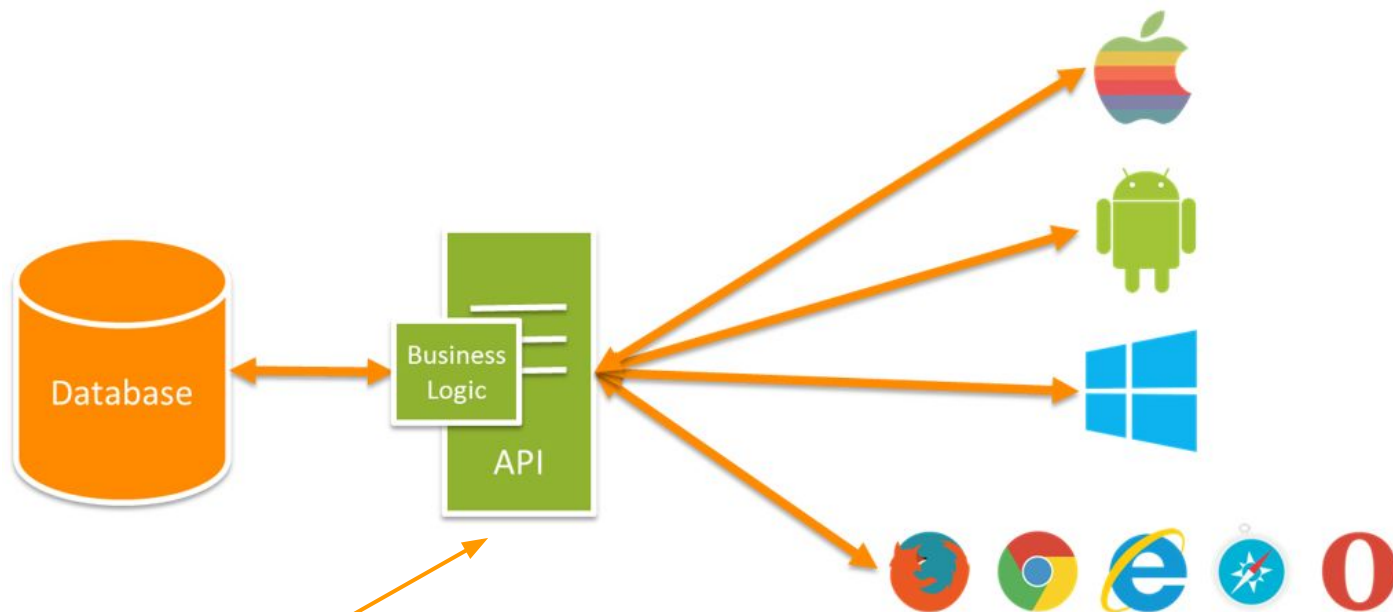
REPASO

- Es una *interfaz*.
- Permiten la utilización desde el exterior del sistema.
- Definen datos y cómo acceder a ellos.
- Implementada como:
 - Procedimientos
 - Funciones
 - Objetos y Métodos
 - Servicios web



Un acercamiento a la arquitectura

REPASO



Application Programming Interface

API REST

REST define un conjunto de principios arquitectónicos por los que se pueden diseñar **Servicios Web** que se centran en los **recursos de un sistema**.

Si la implementación concreta sigue al menos estos principios básicos de diseño, se lo conoce como **RESTfull Web Services**:

- **Interfaz unificada (Recurso + URI + Verbo HTTP)**
- **Cliente-Servidor**
- **Stateless**

API REST- 1er Principio (Interfaz Unificada)

RECURSO

Cualquier información que se pueda nombrar puede ser un **recurso**. (abstracción es la clave principal)

USUARIO

FACTURA

CLIENTE

URI (ENDPOINT)

Son los puntos de entrada con las que el cliente accede a un recurso.

<http://www.example.com/api/usuario>

<http://www.example.com/api/usuario/5>

MÉTODO DEL RECURSO

Se utilizan métodos HTTP de manera explícita junto a cada recurso para realizar la transacción deseada.

GET

POST

PUT

DELETE

USUARIOS

ID

DNI

NAME

FACTURAS

CLIENTES

*“Un **usuario** puede ver la lista de **facturas** de cada **cliente**”*

API REST - EJ ENDPOINT'S

REPASO

- **GET** /factura
Accede al listado de facturas
- **POST** /factura
Crea una factura nueva
- **GET** /factura/:id_fact (ej /factura/123)
Accede al detalle de una factura
- **PUT** /factura/:id_fact (ej /factura/123)
Edita la factura, sustituyendo la información enviada
- **DELETE** /factura/:id_fact (ej /factura/123)
Elimina la factura

<http://www.example.com/api/getFactura/123>



<http://www.example.com/api/eliminarFactura/123>



API REST - Respuestas

REPASO

Cada solicitud a una API REST debe responderse con un **código de respuesta**.

Indican el resultado de una solicitud

- 1xx: Informational
- 2xx: Success
- 3xx: Redirection
- 4xx: Client Error
- 5xx: Server Error

A dark blue rectangular graphic with the text "everything is going to be" in a white script font at the top. Below it is a green circle followed by the text "200 OK" in a large, bold, white sans-serif font.

everything is going to be
● 200 OK

Características SW

Ventajas:

- + Cada servicio puede actualizarse independientemente mientras respete su interfaz
- + Pueden programarse en diferentes lenguajes
- + Facilita la reutilización
- + Desacopla partes no relacionadas del sistema
- + Se pueden usar servicios existentes

Desventajas:

- Costo de cambiar la especificación de servicios
- Complejidad de cambiar las API y de planificar cambios a futuro

Implementando una API REST

Armando una API REST

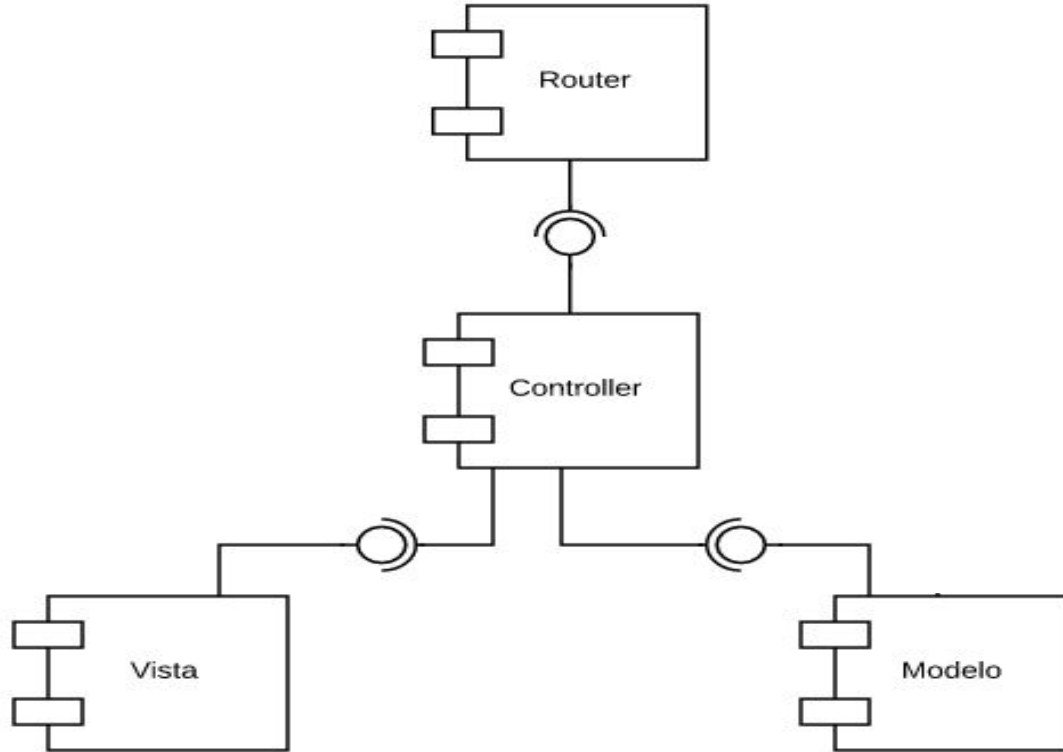
¿Cuál es la diferencia de las API REST frente a renderizar HTML como veníamos haciendo?

1. Las urls no son tan libres: **recurso + verbo**
2. Responden JSON y no HTML
3. Tengo que manejar el código de respuesta

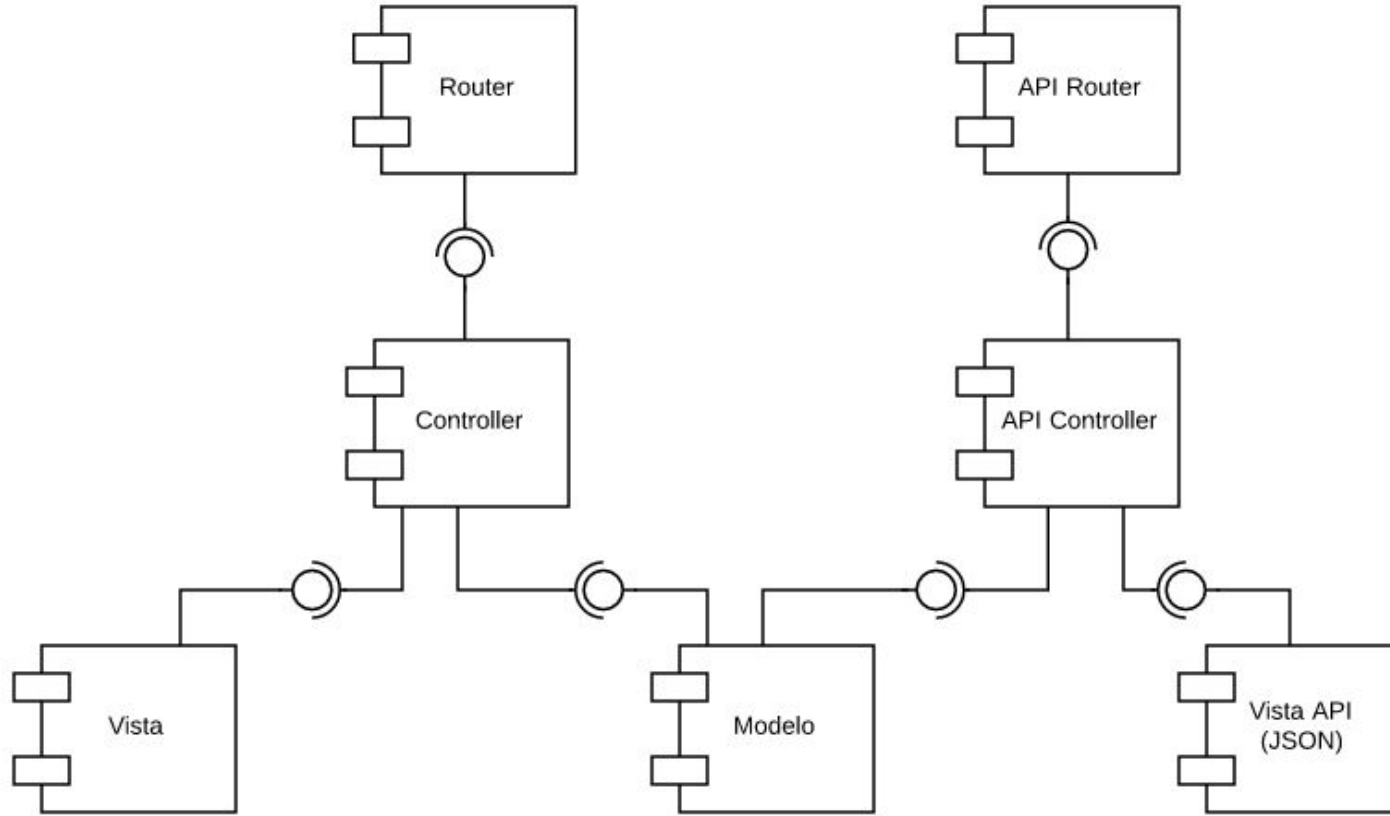
Las API REST siguen exactamente los mismos principios que la renderización del lado del servidor.

¿Como programarían una API Rest?

Hasta ahora teníamos:



Arquitectura MVC y MVC para REST



Esto es (parte de) lo que tienen que hacer en el TPE Parte 2

Reutilización y unión de componentes descripción gráfica



Let's Work

Manos a la obra!

Vamos a programar nuestra propia **API Rest** para nuestra App de Tareas:

¿Qué servicios vamos a tener?

- **Mostrar** tareas
- **Mostrar** tarea
- **Agregar** Tarea
- **Eliminar** tarea
- **Completar** Tarea (EDITAR)



Diseñemos los endpoints

Recurso principal: **TAREAS**

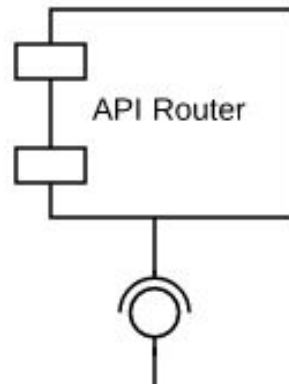
- **GET** api/tareas
Devuelve todas las tareas
- **POST** api/tareas
Crea una tarea nueva
- **GET** api/tarea/:ID (ej api/tarea/123)
Devuelve una tarea específica
- **PUT** api/tarea/:ID (ej api/tarea/123)
Edita la tarea, sustituyendo la información enviada
- **DELETE** api/tarea/:ID (ej api/tarea/123)
Elimina una tarea específica

¿Qué hay que hacer?

1. ROUTER API

Diseñar un nuevo router (API ROUTER) para que procese los llamados a los nuevos servicios.

1. Procesa llamados del tipo
“api/recurso/:params”
2. Rutea según recurso y verbo



.htaccess

1. Redirigimos la solicitud a un router-api.php

```
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^api/(.*)$ route-api.php?resource=$1 [QSA,L,END]
RewriteRule ^(.*)$ route.php?action=$1 [QSA,L]
</IfModule>
```

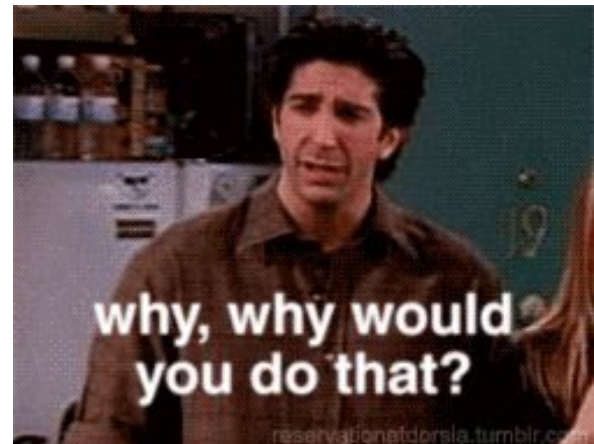
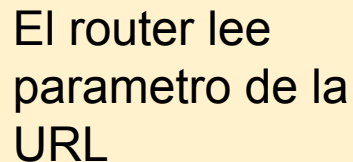


Tabla de ruteo

2. Ruteamos en base al recurso y verbo.

URL	Verbo	Controller	Metodo
tareas	GET	ApiTaskController	obtenerTareas
tareas	POST	ApiTaskController	crearTarea
tareas/:ID	GET	ApiTaskController	obtenerTarea
...			



El router lee
parametro de la
URL

El router tiene en
cuenta el verbo
HTTP

Router 2.0

Router Avanzado

Para implementar la tabla de ruteo, podemos utilizar un Router previamente creado.

Lo descargamos, y lo ponemos en la carpeta `/libs`.

<https://gitlab.com/unicen/Web2/livecoding2019/bolivar/todo-list/blob/master/Router.php>

```
<?php
require_once 'libs/Router.php';

// crea el router
$router = new Router();

// define la tabla de ruteo
$router->addRoute('tareas', 'GET', 'ApiTaskController', 'obtenerTareas');
$router->addRoute('tareas', 'POST', 'ApiTaskController', 'crearTarea');
$router->addRoute('tareas/:ID', 'GET', 'ApiTaskController', 'obtenerTarea');

// rutea
$router->route($_GET['resource'], $_SERVER['REQUEST_METHOD']);
```

Router Avanzado

Las rutas se definen usando el método:

```
$router->addRoute($resource: string, $httpMethod: string,  
    $controller: string, $methodController: string);
```

Los **controllers** obtienen los parámetros del recurso a través de un **arreglo asociativo** que es enviado a los métodos con el nombre *\$params*.

```
public function getTarea($params = null) {  
    $id = $params[':ID'];  
    ...  
}
```

API - Obtener tareas

(GET) api/tareas

TareasApiController

TareasApiController

Clase para manejar el recurso **tareas**.

```
class TareasApiController {  
    private $model;  
    private $view;  
    public function __construct() {  
        $this->model = new TareasModel();  
        $this->view = new APIView();  
    }  
    ...  
}
```

APIView

APIView

Es una vista **común** para todos los servicios.

- Maneja el **código de respuesta**.
- Devuelve la información en **formato JSON**.

Resultado esperado

Response Content Type

application/json

```
[
  {
    "id_tarea": "integer",
    "tarea": "string",
    "realizada": "boolean"
  }
]
```

API View - Implementación

- **Método response**

- Responsabilidades
 - Setear header con resultado de la operación
 - Encode data en formato JSON

- **Método requestStatus**

- Responsabilidad: dar un mensaje asociado a un código de respuesta

API View - Implementación

```
class APIView() {  
  
    public function response($data, $status) {  
        ...  
    }  
  
    private function _requestStatus($code){  
        ...  
    }  
}
```

API Class - Response

```
public function response($data, $status) {  
    header("Content-Type: application/json");  
    header("HTTP/1.1 " . $status . " " . $this->_requestStatus($status));  
    echo json_encode($data);  
}
```



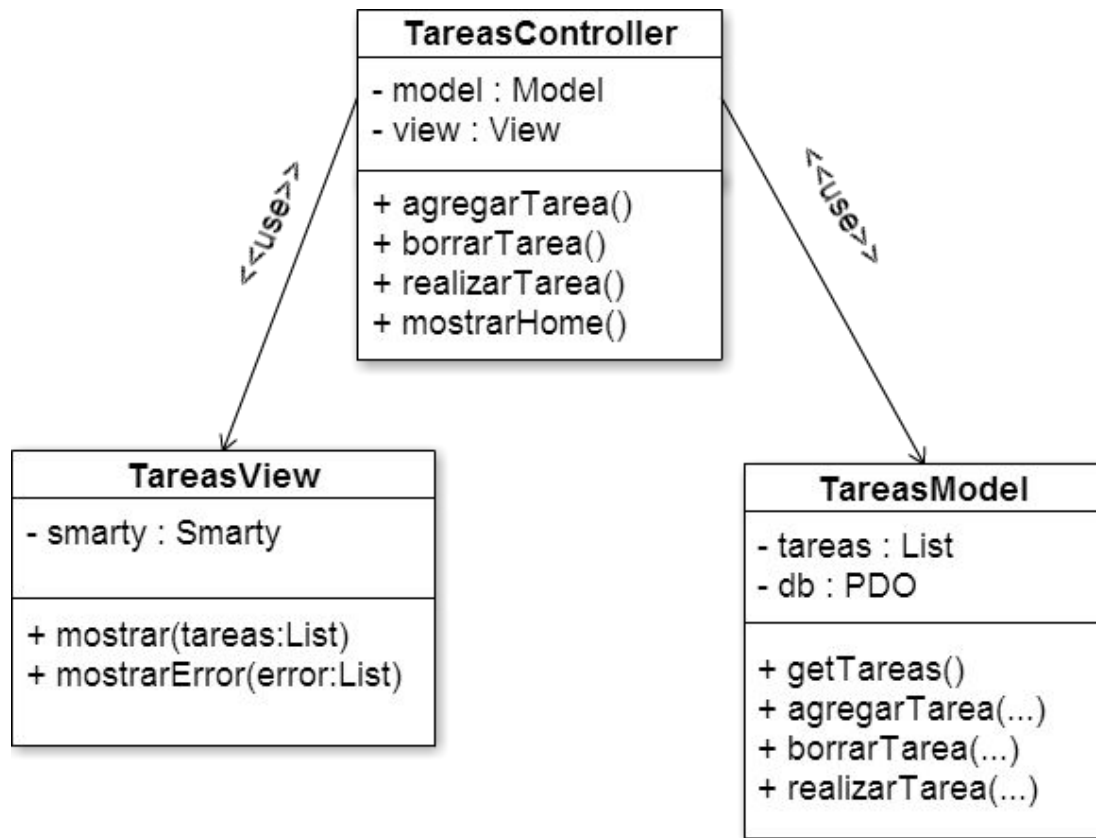
API Class - Request Status

Devuelve el **status** de respuesta según el código solicitado.

```
private function _requestStatus($code){  
    $status = array(  
        200 => "OK",  
        404 => "Not found",  
        500 => "Internal Server Error"  
    );  
    return (isset($status[$code]))? $status[$code] : $status[500];  
}
```

Ejemplo API Tarea

ENDPOINT api/tarea



AJUSTAR A NUEVA ARQUIT

Tarea GET

GET Tarea

La API tiene que permitir traer todas las tareas:

/api/tareas

La API tiene que permitir traer una tarea:

/api/tareas/:ID

TareasAPIController

```
function get($params = []) {  
    $tareas = $this->model->getTareas();  
    return $this->view->response($tareas, 200);  
}
```

Cómo probamos nuestra API REST?

Necesitamos algo para consumirla (llamarla/invocarla), sin necesidad de hacer un cliente en JS para ir probando:

Vamos a usar una herramienta para testear API's



POSTMAN

<https://www.postman.com/downloads/>



Advanced REST client



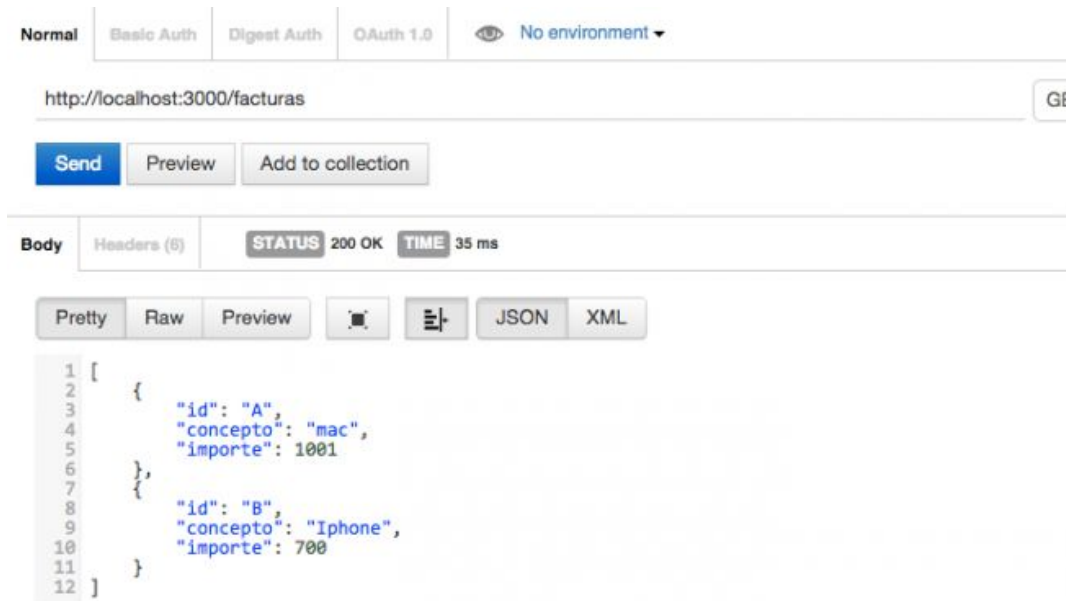
- Permite construir y gestionar peticiones a servicios REST (POST, GET, Etc.).
- Definir la petición que a realizar.



- Le damos enviar y la petición será lanzada contra nuestro servidor. Espera la respuesta (XML/JSON/Texto).



- Captura las respuestas y muestra el resultado de una forma clara y ordenada.



Resultado GET api/tarea

```
[ {  
  "id": "58",  
  "tarea": "Primer tarea",  
  "realizada": "0"  
},  
 {  
  "id": "59",  
  "tarea": "Otra tarea",  
  "realizada": "0"  
}  
]
```



GET Tarea

La API tiene que permitir traer todas las tareas:

/api/tareas

La API tiene que permitir traer una tarea:

/api/tareas/:ID

TareasAPIController

```
function get($params = []) {  
    if(empty($params)){  
        $tareas = $this->model->getTareas();  
        return $this->view->response($tareas,200);  
    }  
    else {  
        $tarea = $this->model->getTarea($params[":ID"]);  
        if(!empty($tarea)) {  
            return $this->view->response($tarea,200);  
        }.... ELSE?  
    }  
}
```

Próxima Clase!

Vamos a completar los servicios que faltan

- Borrado de tarea
- Alta de tarea
- Actualización



Resultado



TANDIL: <https://gitlab.com/unicen/Web2/livecoding2019/tandil/todo-list/commit/0c512e91a7b82869de5114cda6c0904662f0ee2f>

BOLIVAR: <https://gitlab.com/unicen/Web2/livecoding2019/bolivar/todo-list/commit/ac5fafa0dab20f7f0222cc6111a24a9348855932>

RAUCH: <https://gitlab.com/unicen/Web2/livecoding2020/rauch/todolist/-/commit/19477c1590ce40522872d03c21dcd15fe815b495>

Refs:

- <https://restfulapi.net/>
- <https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/>
- <https://blog.philippbauer.de/restful-api-design-best-practices/>
- <https://www.moesif.com/blog/api-guide/api-design-guidelines/>

Referencias

<http://coreymaynard.com/blog/creating-a-restful-api-with-php/>