



Escuela de Ingeniería Informática

Escuela de Ingeniería Informática  
School of Computer Science Engineering

Universidad de Oviedo  
*Universidá d'Uviéu*  
*University of Oviedo*

# Sistemas Distribuidos e Internet

## Tema 9 Servicios Web SOAP y REST

**SOAP**  
Contract First Web Services

RESTful API  
GET PUT POST DELETE

**Dr. Edward Rolando Núñez Valdez**

nunezedward@uniovi.es

# Índice

- Introducción a servicios web
- Buenas prácticas en el diseño e implementación de una API
- Herramientas de desarrollo, prueba, validación, versionado y documentación de APIs REST
- Control de acceso mediante Token
- Intercambio de Recursos de Origen Cruzado (CORS)

# Interoperabilidad

- Desarrollo de aplicaciones centradas en la interacción persona–maquina.
- Necesidad de una comunicación máquina-máquina a través de una conexión a Internet.
- Los EJB remotos y los componentes remotos de Microsoft permiten computación distribuida...
  - Pero, no interoperabilidad
  - Todos los sistemas deben ser JAVA y usar el protocolo RMI
  - Todos los sistemas deben ser VB y usar el protocolo DCOM
- Despliegue válido en entornos corporativos
  - Pero no escalable a Internet
  - Comercio electrónico, empresa extendida, etc.
- Idea principal es que *distintos sistemas sean capaces de comunicarse, transmitir e intercambiar información entre ellos*, independientemente de la *plataforma o lenguaje de programación* en la que estén desarrollado.

# Servicios Web

## ■ Definiciones

- *"**Sistema** de software diseñado para permitir la **interacción interoperable** entre máquinas en una red", W3C*
- *"Los servicios web son **aplicaciones modulares autocontenidas** que puede describir, publicar, localizar e invocar a través de una red", IBM*
- *"Es una tecnología que utiliza un **conjunto de protocolos y estándares** que sirven para **intercambiar datos** entre aplicaciones", Wikipedia*

# Factores que influyeron en la aparición de los SW

- Computación distribuida
  - RPC, CORBA, RMI, DCOM
  - Sistemas fuertemente acoplados
- Integración de aplicaciones empresariales
  - Reacción frente a sistemas ERP monolíticos
- Aparición de XML
  - Adopción por las principales industrias
- Necesidad de intercambios B2B
- Comercio electrónico y burbuja de Internet
- Microsoft vs. Java: Compatibilidad

# Servicios web como componentes remotos

- *Mismo paradigma* que otras soluciones anteriores
  - CORBA: Common Object Request Broker Architecture
  - RMI: Remote Method Invocation
  - RPC: Remote Procedure Call
  - DCOM: Distributed Component Object Model
- **Diferencia técnica:** Se emplean tecnologías estándares abiertas Internet
  - Protocolo de transporte: TCP
  - Protocolo de Aplicación: HTTP
  - Protocolo de servicio web: REST y SOAP
  - Formato de datos: XML y JSON

# Objetivos de los servicios web

- Independencia del lenguaje y de la plataforma
  - Separación de especificación de la implementación
- Interoperabilidad
  - Utilización de estándares: XML, SOAP, WSDL, UDDI, JSON...
- Acoplamiento débil
  - Interacciones síncronas y asíncronas
- A través de Internet
  - Sin control centralizado
  - Utilización de Protocolos establecidos
  - Consideraciones de seguridad
- Modularidad y Reusabilidad de servicios
- Escalabilidad

# ¿Porqué servicios web?

## ■ Ventajas

- Facilitan tareas de los desarrolladores y usuarios.
- Interoperabilidad entre aplicaciones que se pueden ejecutar sobre distintas plataformas.
- Ligadas al mundo web (Internet).
- Al usar HTTP, pueden atravesar firewalls sin necesidad de cambiar las reglas de filtrado.
- Independencia entre el servicio web y la aplicación que lo consume.
- Fomentan el uso de estándares abiertos.
- Históricamente SOAP ha tenido el monopolio.
- REST como alternativa a SOAP.

## ■ Desventajas

- Bajo rendimiento comparado con otros modelos de computación distribuida: RMI, CORBA o DCOM.
- Pueden esquivar medidas seguridad basadas en firewalls.
- Desarrollo e interpretación puede ser compleja sino no se cuenta con las herramientas adecuadas.



# Características de los servicios web

- Un servicio web debe ser *accesible a través de Internet* usando protocolos estándares.
- Un servicio web debe ser *independientes de la plataforma y lenguaje de programación*.
- Un servicio web debe ser *autodescriptivo*.
- Un servicio web debe ser *localizable*.
- Un servicio web debe ser *modular*.

# Aplicaciones y ejemplos de servicios web

- Informes de pronóstico del tiempo.
- Reservas de viajes de líneas aéreas.
- Control de inventario.
- Tasas de tipo de cambios.
- Horario de vuelos.
- Acceso a recursos de redes sociales
  - API de Google
  - API de Twitter
  - API de Facebook
  - API PayPal
  - Etc.

 Meta for Developers

<https://developers.facebook.com/>

**PayPal** Developer

<https://developer.paypal.com/dashboard/>



**Developer Platform**

<https://developer.paypal.com/dashboard/>

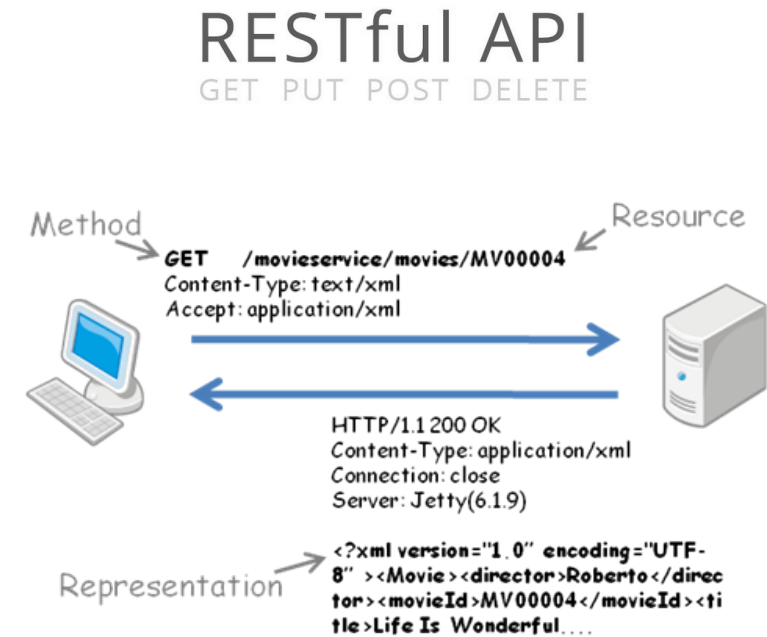
Google APIs Explorer

<https://developer.twitter.com/>

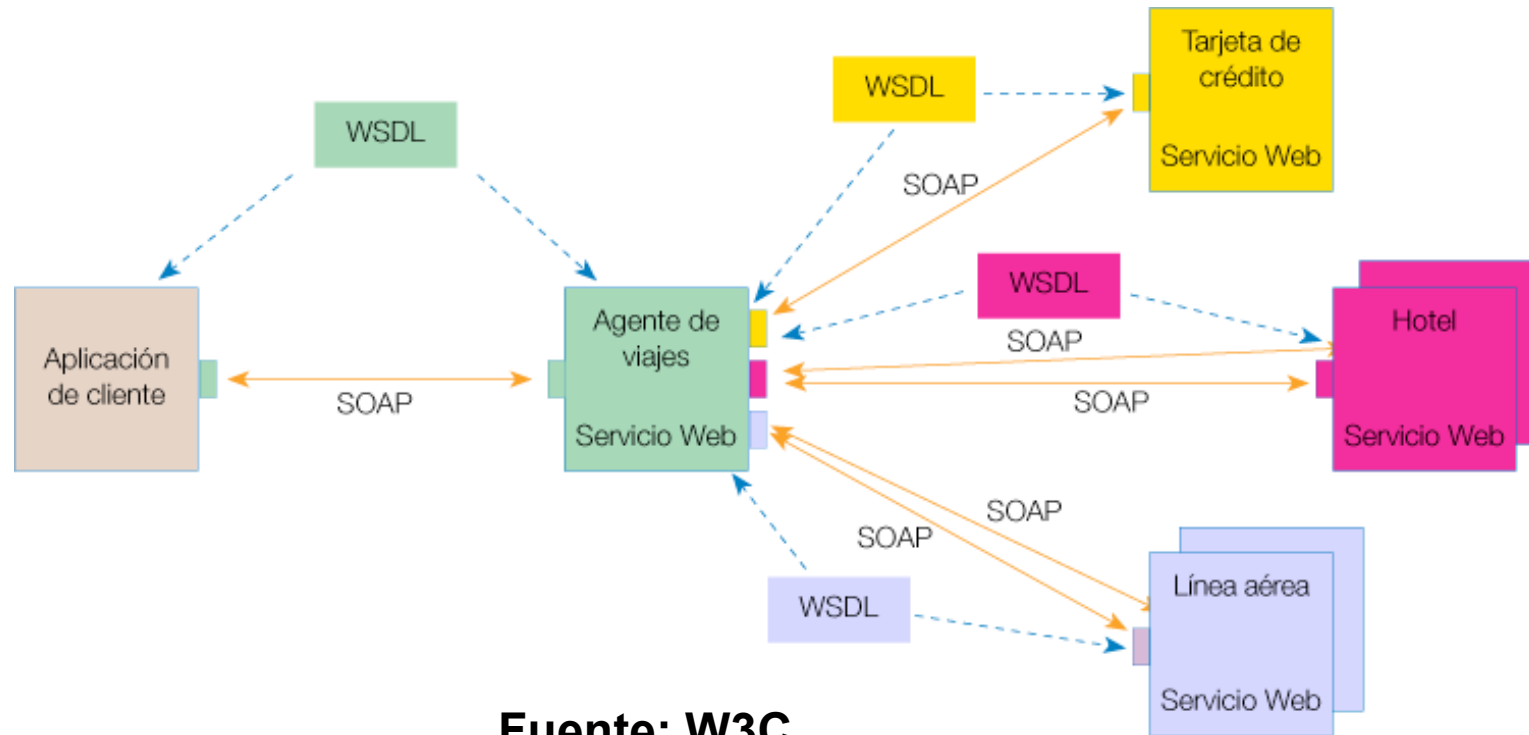
**exchangerate.host**

<https://exchangerate.host/#/docs>

# Tipos de servicios web

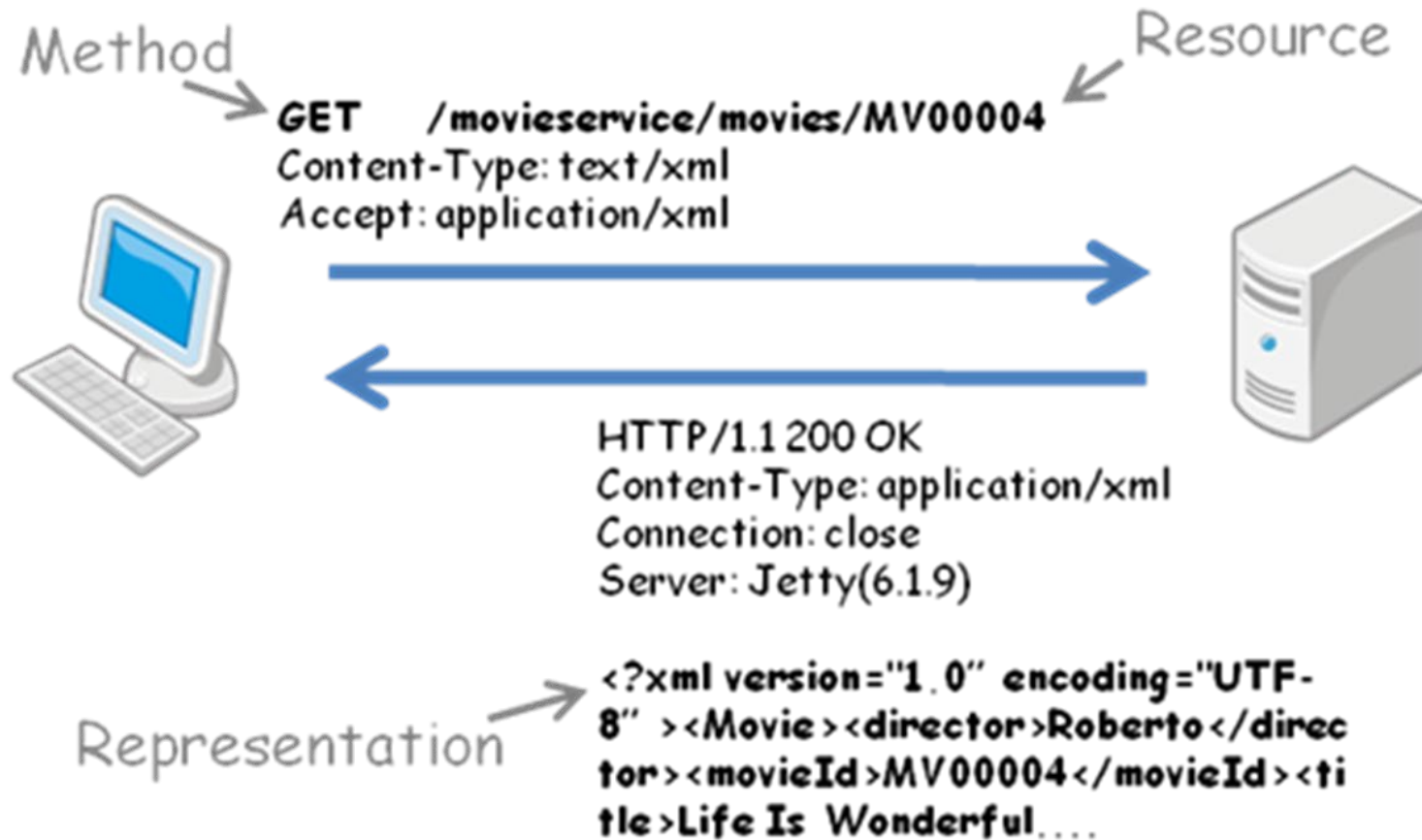


# Interoperabilidad entre múltiples servicios web



Fuente: W3C

# Arquitectura REST



RESTful API  
GET PUT POST DELETE

# Arquitectura REST

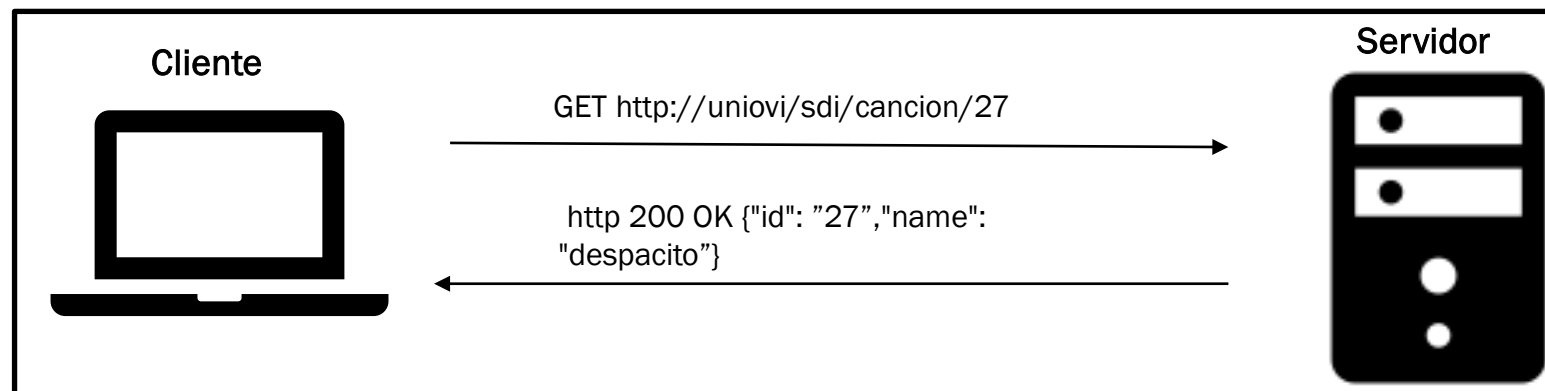
- *REST* acrónimo de *Re*presentational *State* *T*ransfer (Transferencia de estado Representacional)
- Es un *estilo de arquitectura de software* que define un *conjunto de recomendaciones* para diseñar aplicaciones débilmente acopladas que utilizan el protocolo HTTP para la transmisión de datos.
- Definido por Roy Thomas Fielding (co-autor de HTTP).
- Basado en una *filosofía orientada a recursos*, no a llamada a procedimientos/funciones remotas.
- En REST, los *datos y la funcionalidad se consideran recursos* y se accede a ellos utilizando
  - Identificadores de recursos uniformes (URI).
- No es un estándar, pero se basa en estándares
  - HTTP, URI, XML, etc.
  - Los datos que se transmiten *no tienen un formato definido*, puede ser texto simple, JSON, XML.
- Los servicios web creados siguiendo el estilo arquitectónico REST se denominan *servicios web RESTful*.

# Principios de diseño REST

- Desacoplamiento del cliente-servidor.
- Protocolo cliente-servidor sin estado.
- Sistema de capas.
- Infraestructura de almacenamiento en caché.
- Interfaz uniforme.
- Código bajo demanda (opcional).

# Principios de diseño REST

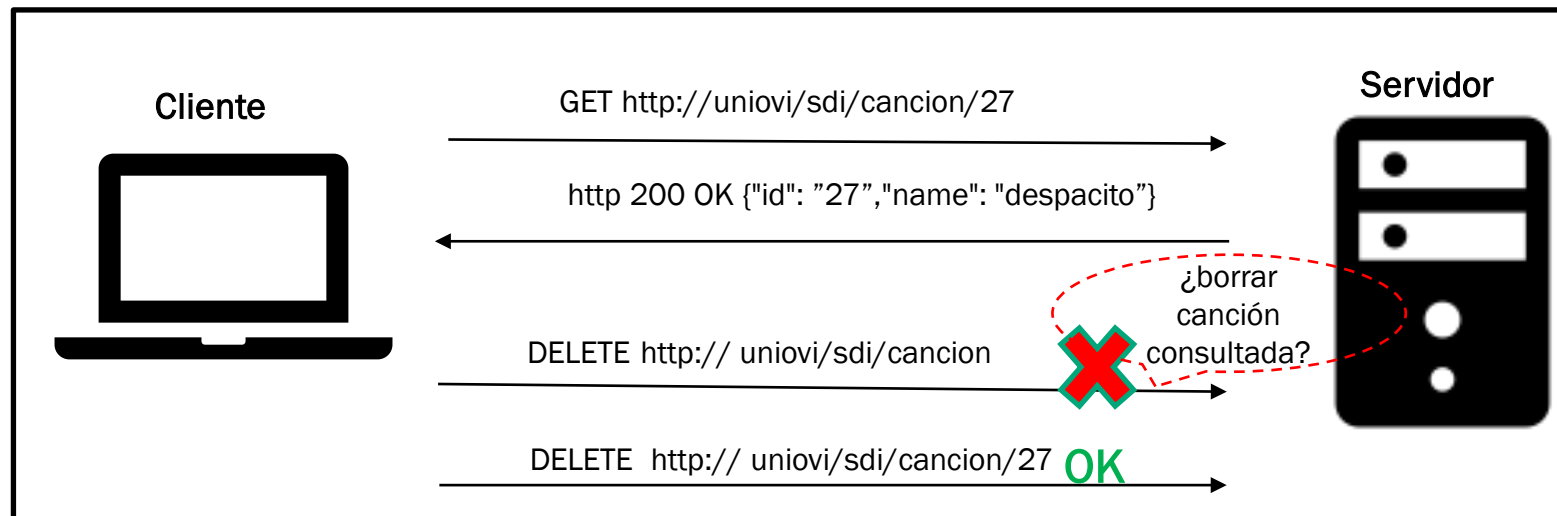
- Desacoplamiento del cliente-servidor
  - El cliente y servidor deben ser *completamente independientes*.
  - Deben interactuar entre sí *solo a través de solicitudes y respuestas*, mediante la URI que identifica al recurso solicitado.
  - *Aumentar la escalabilidad* mediante la optimización de los servicios o recursos *ofrecidos por el servidor*.





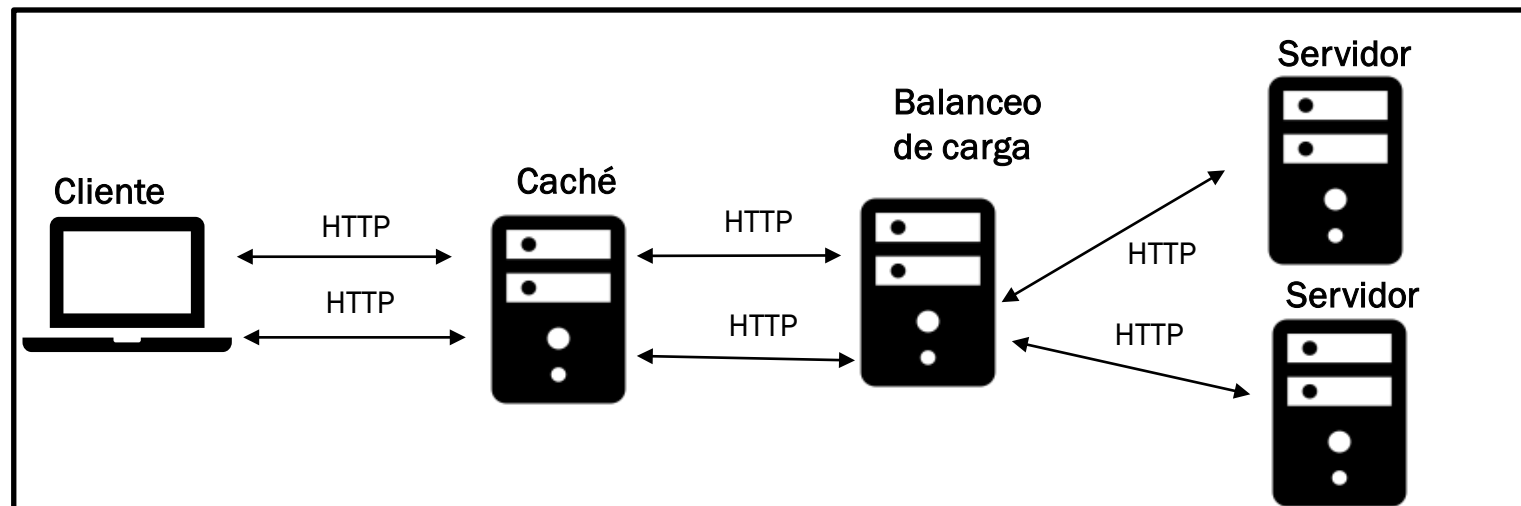
# Principios de diseño REST

- *Protocolo* cliente-servidor *sin estado*
  - Las API REST son API sin estado.
  - El servidor no debería tener que recordar nada de peticiones anteriores.
  - Cada solicitud debe incluir toda la información necesaria para que el servidor pueda procesarla.
  - REST no requieren ninguna sesión del lado del servidor.



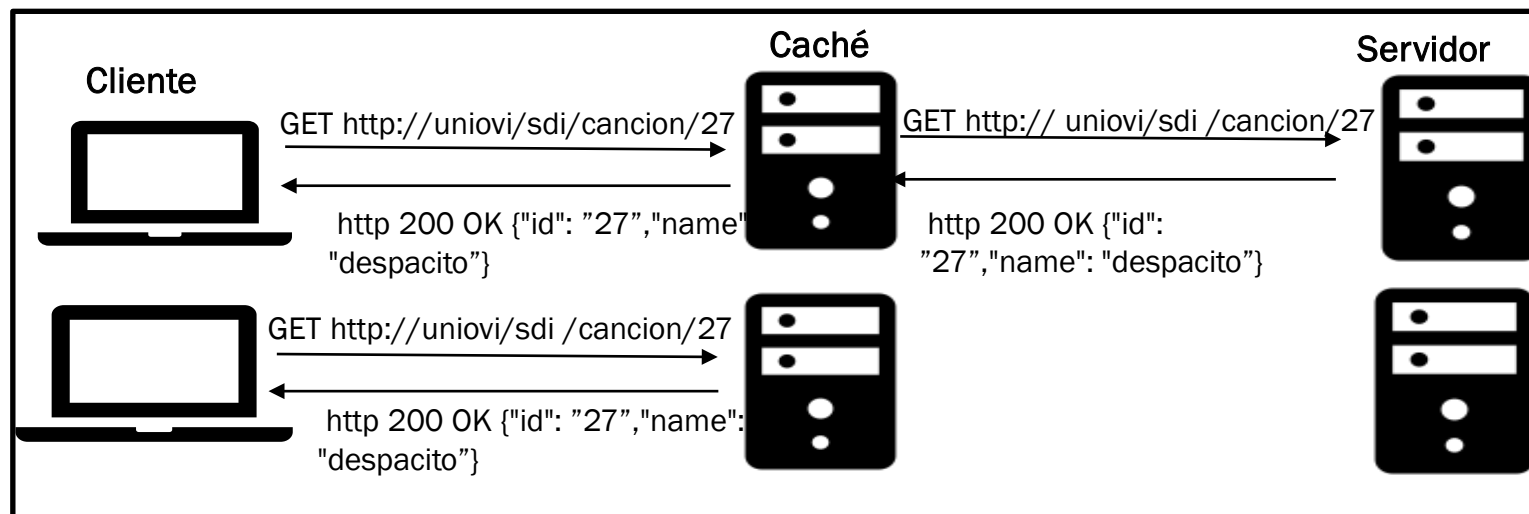
# Principios de diseño REST

- Sistema de capas
  - En las API REST, las peticiones y respuestas pasan por diferentes capas.
  - Cada una de estas capas lleva a cabo una funcionalidad dentro del sistema REST.
  - Las API REST deben diseñarse para que ni el cliente ni el servidor puedan reconocer si se comunican con la aplicación final o con un intermediario.
  - El objetivo es *mejorar la escalabilidad y la seguridad*.



# Principios de diseño REST

- Infraestructura de almacenamiento en caché
  - Los recursos de una API REST deben *poder almacenarse en caché* (en el cliente o en el servidor).
  - Los recursos deben ser declarados como cacheables o no cacheables.
  - El objetivo es *mejorar el rendimiento* en el lado del cliente y *aumentar la escalabilidad* en el lado del servidor.

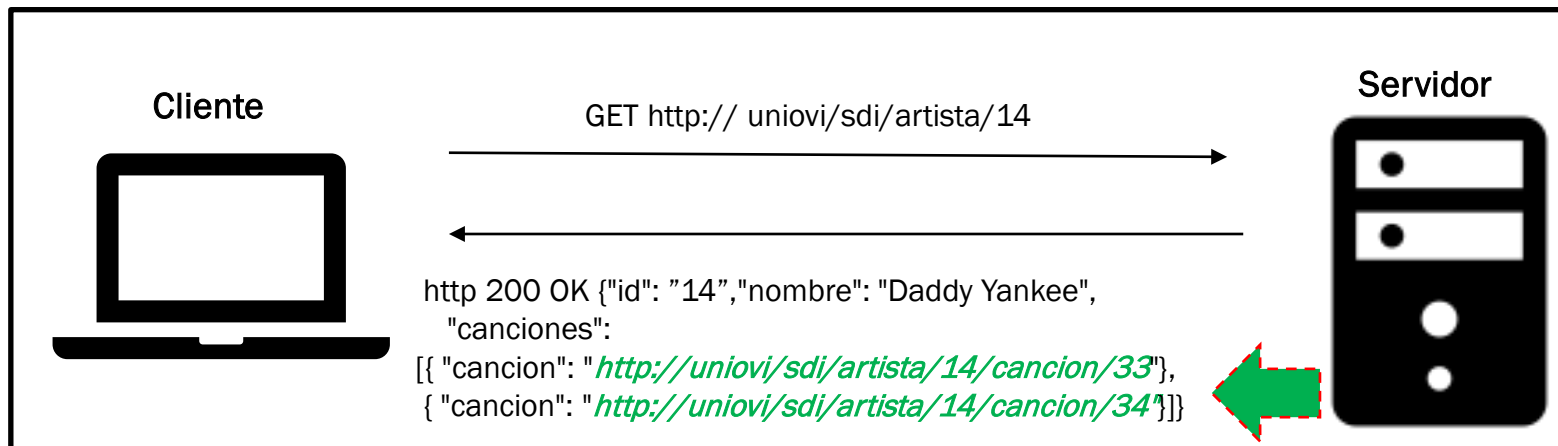


# Principios de diseño REST

- Interfaz uniforme
  - Una sintaxis universal para *identificar inequívocamente todos los recursos* (URI).
  - Los *mensajes* intercambiados son *auto-descriptivos*
    - Toda la información necesaria para entender y procesar el mensaje se encuentra en el propio mensaje.
  - La *gestión del recurso se realiza a través de su representación* (por ejemplo, HTML, JSON, XML, etc.)
    - Cuando un cliente posee la representación de un recurso, tiene suficiente información para modificar o eliminar el recurso original.
  - Una API REST tiene que cumplir con el principio *HATEOAS (Hypermedia As The Engine Of Application State) -> Hipermedia como motor del estado de la aplicación*
- Este principio de diseño *agiliza toda la arquitectura del sistema y mejora la visibilidad* de las comunicaciones.

# Principios de diseño REST

- En Principio *HATEOAS (Hypermedia As The Engine Of Application State)*
  - Un recurso REST puede contener *hipervínculos* de navegación asociada a otros recursos del cliente.
  - Todas las acciones futuras que el cliente pueda realizar se descubren dentro de las representaciones de recursos devueltas por el servidor.



# Principios de diseño REST

- Código bajo demanda (opcional)
  - En una API REST, normalmente el servidor devuelve la representación de recursos estáticos en formatos XML o JSON.
  - Los servidores pueden extender o personalizar la funcionalidad de un cliente, mediante la entrega de código ejecutable al cliente.
  - Ejemplos de esto pueden incluir componentes compilados como applets de Java y scripts del lado del cliente como JavaScript.
  - En estos casos, el código solo debe ejecutarse bajo demanda.
  - El objetivo de este principio es *permitir ampliar la funcionalidad* del sistema en el lado cliente.

# Métodos HTTP para servicios web RESTful

- Gran parte de las API REST son interfaces para gestionar datos *CRUD (Create Read Update Delete)*.
- Los métodos HTTP comprenden una parte importante de las restricciones de *"interfaz uniforme"* y proporcionan la acción correspondiente a las operaciones CRUD.

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

Fuente: <https://restapitutorial.com/lessons/httpmethods.html>

# Ventajas de REST

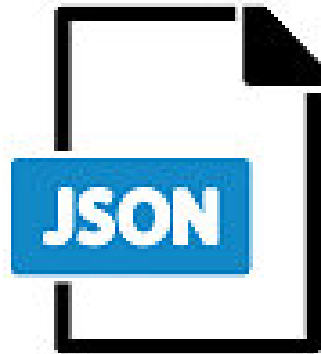
- Permite la interoperabilidad
  - Es independiente del lenguaje de programación
- Es versátil
  - Utiliza HTTP como protocolo de transporte estándar
  - Permite el intercambio de mensaje en múltiples formatos: XML o JSON
  - Es decir, utiliza el idioma de la web
- Es escalable
  - Como utiliza HTTP resulta escalable y sencillo de utilizar en sistemas que contienen Firewalls o Proxies.
- REST es bastante ligero, así que son ideales para los contextos más nuevos, como IoT, el desarrollo de aplicaciones móviles, etc.
- Es una alternativa en auge a otros protocolos estándares de intercambio de datos como SOAP



# Desventajas de REST

- Limitación en la estandarización en tipos de datos
  - Las solicitudes y respuestas no definen tipos de datos, como en SOAP
- La visibilidad a la que están expuestas las URIs públicas puede suponer un problema de *seguridad*
- Limitaciones técnicas en algunas peticiones web, debido a la longitud máxima de los parámetros soportado por algunos métodos HTTP

# Formato para intercambio de datos de REST



# ¿Porqué JSON?

- JSON: **J**ava**S**cript **O**bject **N**otation
- Es un formato ligero de almacenamiento e Intercambio de datos
- Es texto, escrito usando la **sintaxis Javascript**
- Independiente del lenguaje
- Puede ser usado con cualquier lenguaje de programación actual
- Herramientas de ayuda en casi todos los Frameworks / Lenguajes
- En general más compacto y eficiente que XML
- Integración directa con MongoDB

# **BUENAS PRÁCTICAS EN EL DISEÑO E IMPLEMENTACIÓN DE UNA API**

# Buenas prácticas en el diseño e implementación de una API

- Se debe cuidar mucho la *jerarquía entre URIs* para que el uso sea intuitivo
  - Ejemplo 3 entidades
  - <http://www.example.com/librería/libro/pagina/>
- Elegir correctamente el identificador único de cada recurso
  - <http://pais/es>
  - <http://pais/034>
- Incluir la versión de la API dentro de la URL
  - <http://software/version/ultima>
  - <http://software/version/3.2>
  - <http://software/v2/recurso>
  - Es válido (y en casos recomendable) que dos URL apunten al mismo recurso
- No usar nunca el método GET para cambios de estado de un recurso (crear, actualizar o borrar).
- Proporcionar capacidades de filtrado, ordenado y acceso a información de cada campo
  - Mejorar el rendimiento

# Buenas prácticas en el diseño e implementación de una API

- Usar singular o plural pero solo uno de ellos.
- Formato de datos
  - Retornar la respuesta en un formato que pueda ser fácilmente procesado por una aplicación, por ejemplo: *JSON o XML*
  - Usar cabeceras http para especificar el formato.
  - Permitir que el usuario elija el formato de salida.
    - GET <http://www.example.com/api/v2/recurso?salida=XML>
- Permitir operaciones de prueba
  - DELETE <http://www.example.com/api/v2/recurso?test=true>
- Retornar un código de respuesta HTTP adecuado
  - Ejemplo: 200 OK, 201 Created, etc
- Seguir las recomendaciones que estandarizan la forma de diseñar, crear, documentar y consumir API REST.
  - Iniciativa OpenAPI (OAI) -> <https://www.openapis.org/about>
  - Especificación OpenAPI o especificación Swagger -> <https://swagger.io/docs/specification/about/>

# Ejemplo de una API REST






- Ejemplo de una API web que utiliza correctamente los métodos HTTP
  - GET /v2/images => obtener imágenes
  - GET /v2/images/{image\_id} => obtener una imagen concreta
  - PUT /v2/images/{image\_id} => actualizar una imagen concreta
  - POST /v2/images/ => crear una nueva imagen
  - DELETE /v2/images/{image\_id} => eliminar una imagen concreta
- Mas información: <http://developer.openstack.org/api-ref-image-v2.html>

# **HERRAMIENTAS DE DESARROLLO, PRUEBA, VALIDACIÓN, VERSIONADO Y DOCUMENTACIÓN DE APIS REST**



# Herramientas de prueba de API REST

## ■ Top 5 de herramientas de prueba API 2023

Product	 Katalon	 REST-assured	 apigee	 APACHE JMeter	 POSTMAN
Application Under Test	API, Web, Mobile, and Desktop apps	REST API	API	Web application and API	API
Supported platform	Windows Linux MacOS	Windows Linux MacOS	Windows Linux MacOS	Windows Linux MacOS	Windows Linux MacOS
Ease of installation and usability	Easy to set up and use	<ul style="list-style-type: none"><li>• Easy to install and use</li><li>• Require basic knowledge of Java</li></ul>	Require knowledge of endpoint management	<ul style="list-style-type: none"><li>• Easy to install and use</li><li>• Require basic knowledge of Java</li></ul>	Easy to set up and use
Rating (Gartner Peer Insights)	★★★★★ 740 reviews	N/A	★★★★★ 139 reviews	N/A	★★★★★ 475 reviews

Fuente: <https://katalon.com/resources-center/blog/top-5-free-api-testing-tools>

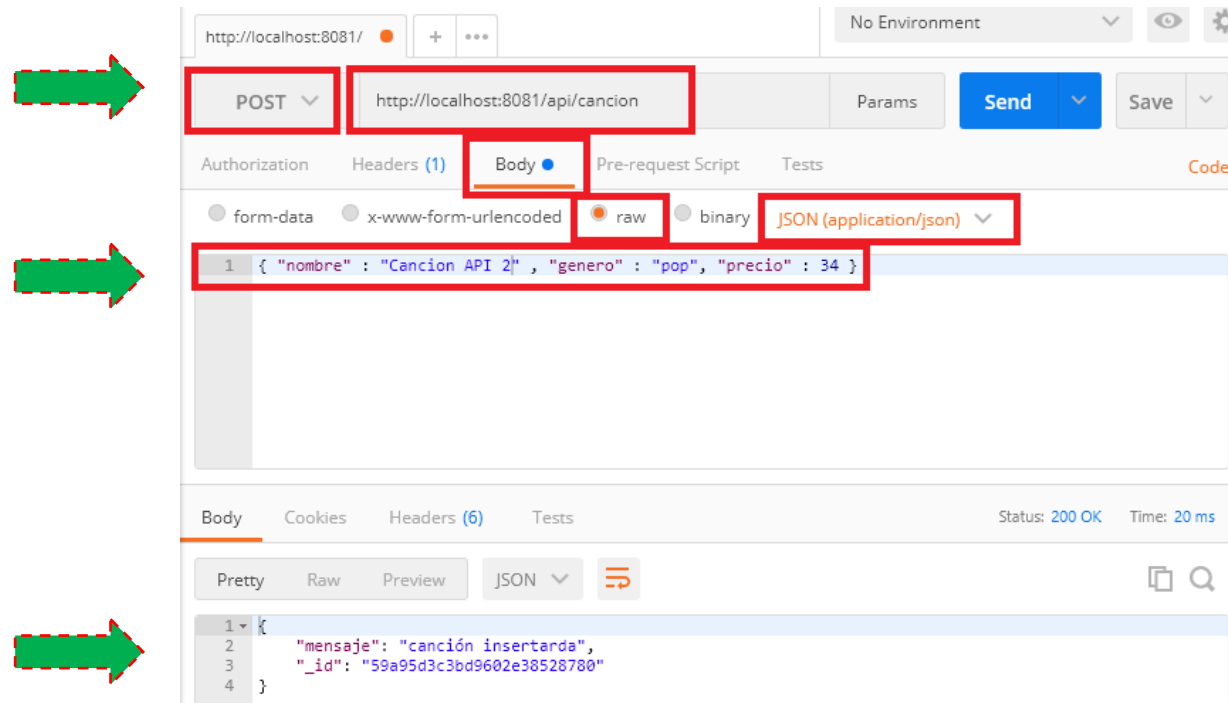
# POSTMAN

- Herramienta ideal para probar APIs REST de forma rápida y sencilla.
- Permite realizar peticiones HTTP (GET, POST, DELETE, PUT...).
- Permite modificar todos los parámetros de la petición.
- Disponible extensión para Chrome y también como aplicación standalone.



# POSTMAN

- Ejemplo de petición POST



# **CONTROL DE ACCESO E IDENTIFICACIÓN DEL CLIENTE CON TOKEN**

# Identificación del cliente con Token

- Los servicios web *REST son stateless* por lo que no suelen a hacer uso del objeto sesión
- La identificación de los usuarios se lleva a cabo comúnmente utilizando un *token de seguridad*
- Este token acompaña a todas las peticiones y suele enviarse:
  - Como *parámetro en las peticiones* GET, POST, etc.
  - o en las *HEADERS* (siento este último el lugar más común).
- El token es lo que permite a la aplicación *controlar si un usuario tiene permisos o no* para ejecutar los servicios.

# Identificación del cliente con Token

- Mecanismos de identificación basados en tokens

- **Token único**

- Cada cuenta de usuario es provista de un token único que le identifica
    - Este token se asocia a todas las peticiones realizadas a los servicios

- **Token por login**

- Cuando el cliente envía sus credenciales a un servicio específico recibe un token
    - Este token debe ser almacenado y enviado en todas las peticiones que el cliente realiza.
    - Puede caducar:
      - Después de un tiempo sin recibir peticiones (algo similar a la sesión)
      - En cada petición y reenviando uno nuevo

# Control de acceso mediante Token por login

## ■ Paso 1: Crear un servicio de autenticación

```
app.post("/api/autenticar/", function(req, res) {  
  var seguro = app.get("crypto").createHmac('sha256',  
  cifrar → app.get('clave'))  
    .update(req.body.password).digest('hex');  
  var criterio = {email : req.body.email, password : seguro}  
  Buscar → gestorBD.obtenerUsuarios(criterio, function(usuarios) {  
    if (usuarios == null || usuarios.length == 0) {  
      res.status(401); // Unauthorized  
      Responder → res.json({ autenticado : false})  
    } else {  
      res.status(200);  
      res.json({autenticado : true})  
    }  
  });  
});
```

# Control de acceso mediante Token por login

- Probar servicio de autenticación





# Control de acceso mediante Token por login

- Paso 2. Modificar el sistema para que devuelva un **token de seguridad**
  - Opción 1: Crear un *token totalmente aleatorio* y almacenarlo junto a la información del usuario en la base de datos
  - Opción 2: Crear un **token con el ID del usuario** encriptado + otro dato.
    - Ejemplo: *email del usuario + milisegundo actual* -> Timestamp  
Date.now()

# Control de acceso mediante Token por login



- Para crear los tokens de acceso en node.js se puede usar el módulo *jsonwebtoken*
  - <https://www.npmjs.com/package/jsonwebtoken>
- Hay que instalar el módulo en nuestro proyecto
- Incluimos el módulo del fichero principal de la aplicación app.js

*npm install jsonwebtoken*

```
var express = require('express');  
var app = express();  
  
var jwt = require('jsonwebtoken');  
app.set('jwt', jwt);
```

# Control de acceso mediante Token por login

- Paso 3: Modificamos la respuesta en el servicio de autenticación implementado en el paso 1.

```
app.post("/api/autenticar/", function(req, res) {  
  ...  
  gestorBD.obtenerUsuarios(criterio, function(usuarios) {  
    if (usuarios == null || usuarios.length == 0) {  
      res.status(401);  
      res.json({autenticado : false})  
    } else {  
       Generar Token  
      var token = app.get('jwt').sign(  
        {usuario: criterio.email , tiempo: Date.now()/1000},  
        "secreto");  
      res.status(200);  
       Devolver Token  
      res.json({autenticado: true, token : token  
    }); }); });  
  }  
});
```

# Control de acceso mediante Token por login

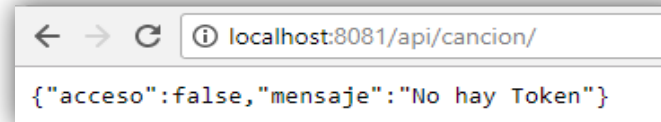
- Paso 4: Requerir el uso del TOKEN para cualquier servicio de la API REST.
  - Implementar un nuevo Router *routerUsuarioToken* en el fichero principal app.js, que permita:
    - Obtener el parámetro token (GET, POST o HEADER)
    - Verificar el token (des-encryptándolo)
    - Si no hay token devolver un mensaje de error "No hay token".

# Control de acceso mediante Token por login

```
...
var routerUsuarioToken = express.Router();
routerUsuarioToken.use(function(req, res, next) {
  Obtener T → var token = req.body.token || req.query.token || req.headers['token'];
  Verificar T → if (token != null) {
    jwt.verify(token, 'secreto', function(err, infoToken) {
      Token inválido → if (err || (Date.now()/1000 - infoToken.tiempo) > 240 ){
        res.status(403); // Forbidden
        res.json({ acceso : false, error: 'Token inválido o caducado' });
        return;
      } else {
        Token válido → res.usuario = infoToken.usuario;
        next();
      }
    });
  } else {
    No hay Token → res.status(403); // Forbidden
    res.json({ acceso : false, mensaje: 'No hay Token' });
  }
});
// Aplicar routerUsuarioToken
app.use('/api/cancion', routerUsuarioToken);
});
```

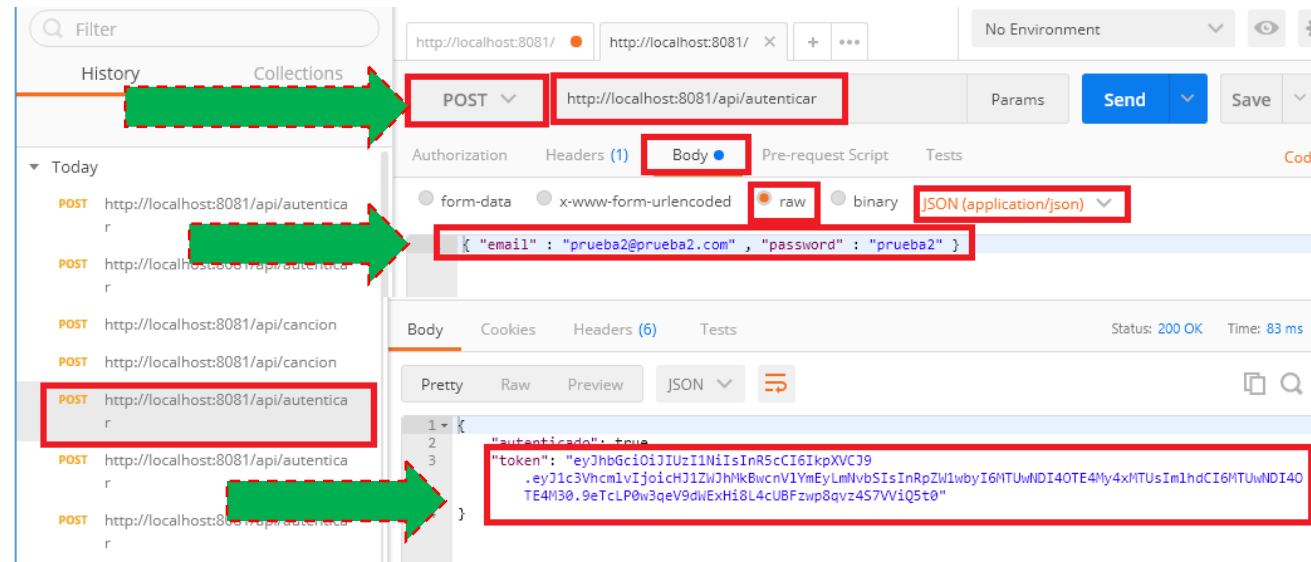
# Control de acceso mediante Token por login

- Paso 5: Probar sin TOKEN
  - Probamos a acceder al servicio <http://localhost:8081/api/cancion> desde el navegador.



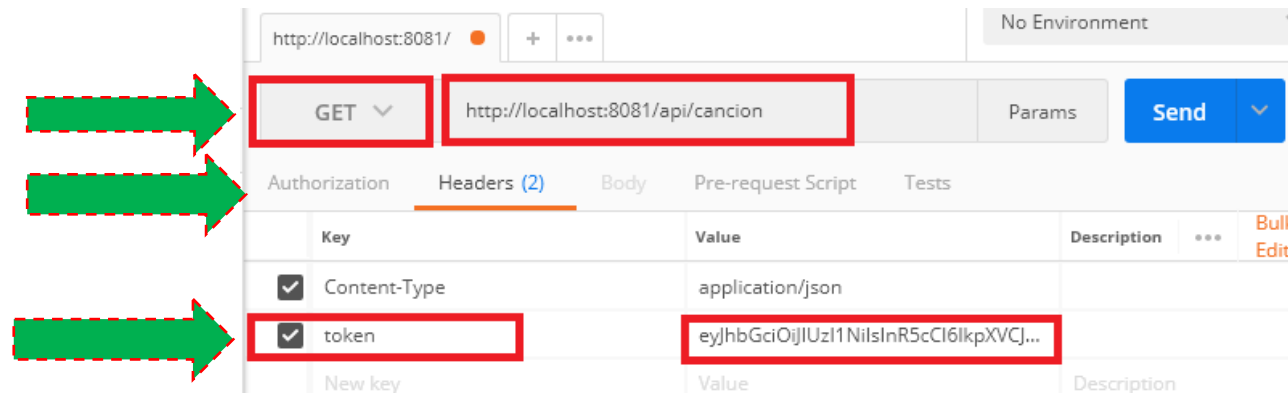
# Control de acceso mediante Token por login

- Paso 6: Probar con TOKEN válido
  - Paso 6.1: Obtener un token válido



# Control de acceso mediante Token por login

- Paso 6: Probar con TOKEN válido
  - Paso 6.2: copiamos el token y creamos una nueva petición dirigida a obtener la lista de canciones mediante GET





# **INTERCAMBIO DE RECURSOS DE ORIGEN CRUZADO (CORS)**

# Implementación cliente REST

- Un *cliente REST* es cualquier aplicación software que consuma un servicio web REST
- Esta aplicación podría ejecutarse en
  - La misma máquina
  - Otra máquina diferente
- Los clientes pueden estar implementados en cualquier lenguaje de programación
  - No tienen porque estar en el mismo que el Servicio Web
  - Los principales lenguajes de programación soportan el uso de SW REST
- Para utilizar el servicio web REST debemos conocer los detalles
  - URLs
  - Parámetros que reciben
  - Parámetros que retornan
  - Etc.

# Intercambio de Recursos de Origen Cruzado (CORS)

- Control de acceso HTTP
  - Por seguridad, los navegadores y servidores Web restringen las solicitudes HTTP de origen distintos al dominio al que pertenecen.
  - El Intercambio de Recursos de Origen Cruzado (CORS)
    - Es un mecanismo que utiliza para *añadir cabeceras HTTP* adicionales para *permitir que un programa* obtenga *permiso para acceder* a recursos seleccionados desde un servidor de *origen distinto (dominio)* al que pertenece.
  - CORS permite transferencia segura de datos en dominios cruzados entre navegadores y servidores Web.

# Intercambio de Recursos de Origen Cruzado (CORS)

- Control de acceso HTTP
  - Ejemplo cabeceras

```
Access-Control-Allow-Origin: *  
Access-Control-Allow-Credentials: true  
Access-Control-Allow-Methods: <Method>, <Method> ..  
Access-Control-Allow-Headers: <header-name>, <header-name>..
```

# Intercambio de Recursos de Origen Cruzado (CORS)

- Access-Control-Allow-Origin
  - Indica si la respuesta puede ser compartida con recursos del **ORIGIN** dado.
  - EL ORIGIN (origen) del contenido web se define por el esquema (**protocolo**), el host (**dominio**) y *el puerto* de la URL utilizada para acceder a él.
  - Dos objetos tienen el mismo origen solo cuando el esquema, el host y el puerto coinciden.

# Intercambio de Recursos de Origen Cruzado (CORS)

## ■ Access-Control-Allow-Origin

### ■ Sintaxis

```
Access-Control-Allow-Origin: *  
Access-Control-Allow-Origin: <ORIGIN>
```

### ■ Directiva

- \* : Permite que cualquier origen URL pueda acceder al recurso
- <ORIGIN>: Especifica que URI puede acceder al recurso.
  - Access-Control-Allow-Origin: http://example.com

# Intercambio de Recursos de Origen Cruzado (CORS)

- Access-Control-Allow-Credentials
  - Cuando este encabezado es verdadero significa que el servidor permite que las credenciales del usuario se incluyan en solicitudes de origen cruzado
  - Las credenciales son cookies, encabezados de autorización o certificados de clientes TLS
- Access-Control-Allow-Methods: <Method>, <Method>
  - Este encabezado especifica el método o los métodos permitidos al acceder al recurso en respuesta a una solicitud.
  - GET, POST, OPTIONS, PUT, etc
- Access-Control-Allow-Headers: <header-name>, <header-name>..
  - Este encabezado se usa para indicar qué encabezados HTTP se pueden usar durante la solicitud actual.
  - Origin, Content-Type, token, etc.

# Intercambio de Recursos de Origen Cruzado (CORS)

- Habilitar CORS en una API REST desarrollada en Node.js

```
var express = require('express');
var app = express();
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Credentials", "true");
  res.header("Access-Control-Allow-Methods", "POST, GET, DELETE, UPDATE, PUT");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept, token");
  next();
});
```



# **EJEMPLO CLIENTE JQUERY-AJAX**

# Cliente jQuery-Ajax

- Ejemplo de un cliente que accede a los servicios web de una tienda música.
- La aplicación va a ser de una única página (SPA)
  - El propósito de estas aplicaciones es dar una experiencia de usuario más fluida, *haciendo aparecer y desaparecer componentes dinámicamente*.
- Utilizaremos jQuery y AJAX.

# Cliente jQuery-Ajax

- Esta nueva aplicación podría ser desplegada dentro de una aplicación express o cualquier otro servidor web
- En este caso para simplificar el desarrollo vamos a hacer que esta aplicación *sea parte de la aplicación de tienda de música desarrollada en prácticas.*
  - No es lo ideal

# Cliente jQuery-Ajax

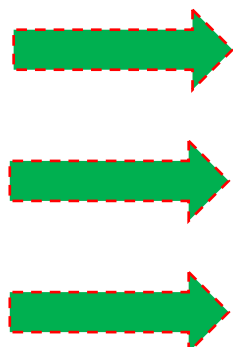
- Paso1: Creamos un fichero **cliente.html** en el directorio /public/ del proyecto tienda de música desarrollada en práctica.
  - Incluir los scripts de jquery y bootstrap
  - Sobre este HTML cargaremos dinámicamente diferentes componentes/widgets

```
<title>jQuery uoMusic </title>
<meta charset="utf-8"/>
<meta name="viewport" content="width=device-width, initial-scale=1"/>
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"/>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
</head>
```

# Cliente jQuery-Ajax

## ■ Paso2: Modificar cliente.HTML

- Añadir el DIV donde se cargarán dinámicamente los componentes
- Añadir un Script para incluir las variables generales usadas para hacer peticiones a la API REST. (**token** y **URLbase** )
- El script para cargar dinámicamente el widget del login.



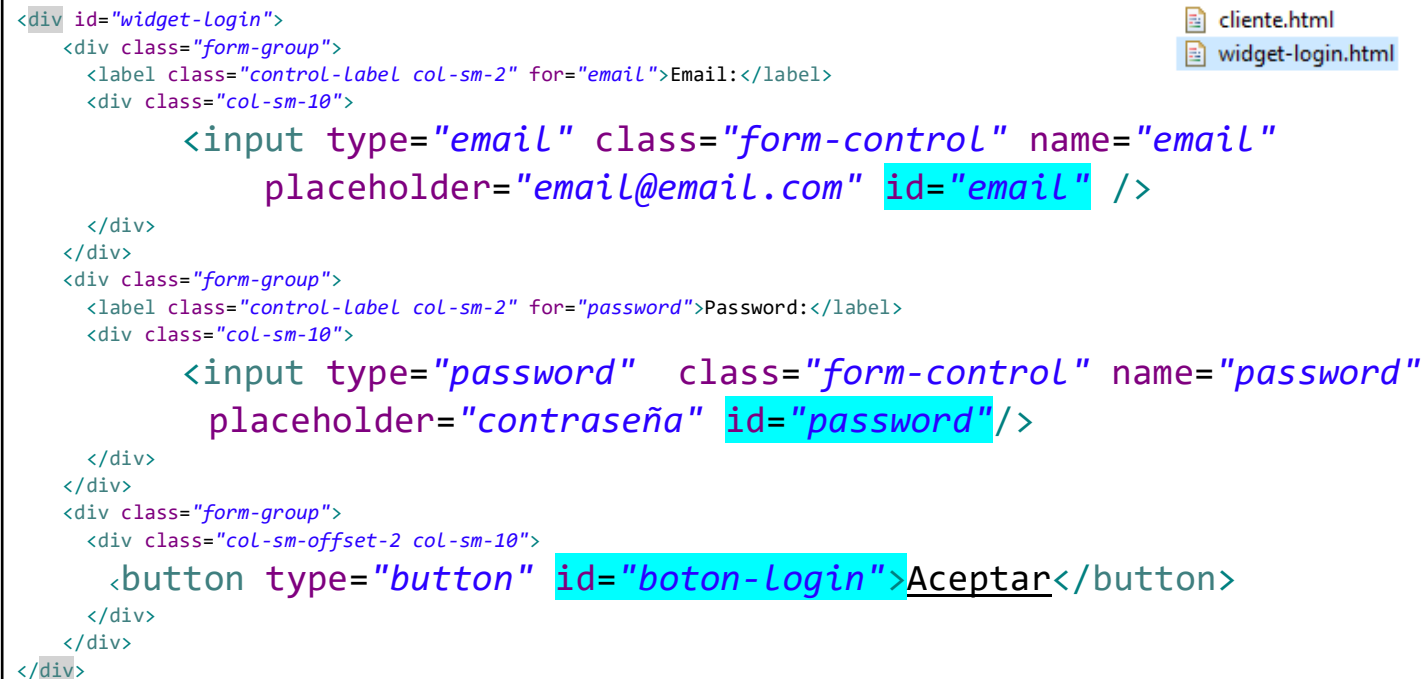
```
<!-- Contenido -->
<div class="container" id="contenedor-principal"></div>
<script>
    var token;
    var URLbase = "http://localhost:8081/api";

    $( "#contenedor-principal" ).load( "widget-login.html"
</script>
</body>
```

# Cliente jQuery-Ajax

- Paso 3: Creamos un nuevo fichero **widget-login.html** en la carpeta **/public/**.
  - *No va a tener la etiqueta `<form>`, no se va a enviar por el método tradicional sino usando jQuery.*

```
<div id="widget-login">
  <div class="form-group">
    <label class="control-label col-sm-2" for="email">Email:</label>
    <div class="col-sm-10">
      <input type="email" class="form-control" name="email"
        placeholder="email@email.com" id="email" />
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-2" for="password">Password:</label>
    <div class="col-sm-10">
      <input type="password" class="form-control" name="password"
        placeholder="contraseña" id="password"/>
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <button type="button" id="boton-login">Aceptar</button>
    </div>
  </div>
</div>
```



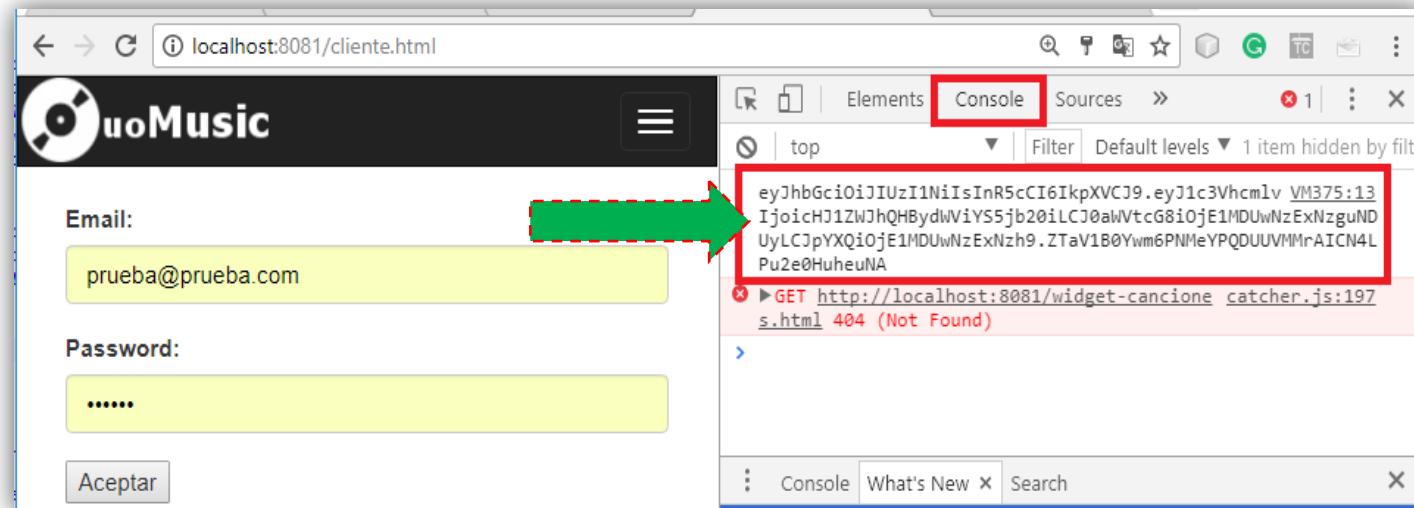
# Cliente jQuery-Ajax

- Paso 4: A continuación del código HTML implementaremos el script.

```
<script>
$("#boton-login").click(function() {
    $.ajax({url: URLbase + "/autenticar", type: "POST",
        data: {email : $("#email").val(), password : $("#password").val()},
        dataType: 'json',
        success: function(respuesta) {
            console.log(respuesta.token); // <- Prueba
            token = respuesta.token;
            $("#contenedor-principal" ).load( "widget-canciones.html");
        }, error : function (error){
            $("#widget-login" ).prepend("<div class='alert alert-
danger'>Usuario no encontrado</div>");
        }});});
</script>
```

# Cliente jQuery-Ajax

- Paso 5: Desde `http://localhost:8081/cliente.html` comprobamos que la API devuelve el token





# Cliente jQuery-Ajax

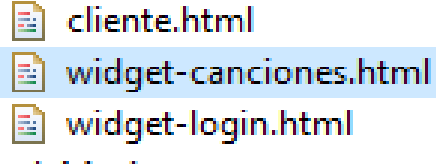
- Paso 6: Crear un widget para listar canciones
  - Disponiendo del *token de seguridad* ya podemos hacer peticiones a API REST
  - Obtener lista de canciones
  - Insertar, borrar, modificar un registro de canciones

# Cliente jQuery-Ajax

## ■ Paso 6.1: Crear fichero HTML **widget-canciones.html**

```
<div id="widget-canciones" >
  <button class="btn" onclick="cargarCanciones()" >Actualizar</button>
  <table class="table table-hover">
    <thead>
      <tr>
        <th>Nombre</th>
        <th>Genero</th>
        <th>Precio</th>
        <th class="col-md-1"></th>
      </tr>
    </thead>
    <tbody id="tablaCuerpo">

    </tbody>
  </table>
</div>
```



A file explorer icon showing a list of files: cliente.html, widget-canciones.html (highlighted), and widget-login.html.

# Cliente jQuery-Ajax

- Paso 6.2: A continuación de la vista incluimos el script con la función **cargarCanciones()**

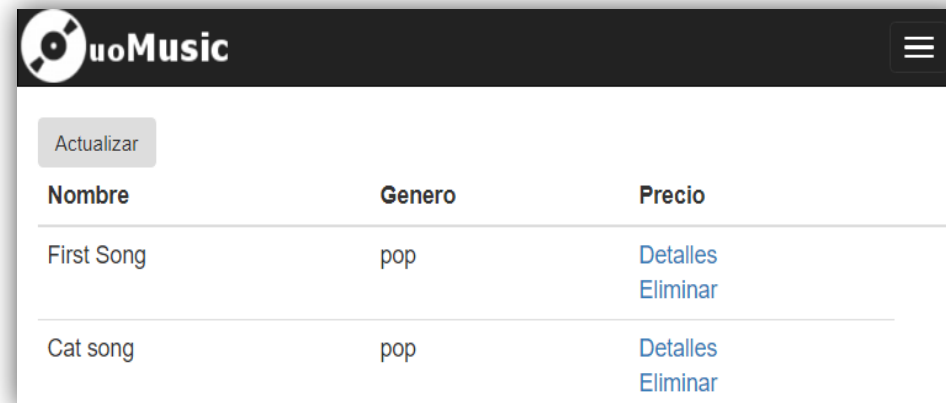
```
<script>
var canciones;
function cargarCanciones(){
    $.ajax({
        url: URLbase + "/cancion", type: "GET", data: {},
        dataType: 'json', headers: { "token": token },
        success: function(respuesta) {
            canciones = respuesta;
            actualizarTabla(canciones);
        }, error : function (error){
            $( "#contenedor-principal" ).load("widget-login.html");
        }});
    ...
}
```

# Cliente jQuery-Ajax

```
...  
function actualizarTabla(cancionesMostrar){  
    $( "#tablaCuerpo" ).empty(); // Vaciar la tabla  
    for (i = 0; i < cancionesMostrar.length; i++) {  
        $( "#tablaCuerpo" ).append(  
            "<tr id="+cancionesMostrar[i]._id+">" +  
            "<td>"+cancionesMostrar[i].nombre+"</td>" +  
            "<td>"+cancionesMostrar[i].genero+"</td>" +  
            "<td>"+cancionesMostrar[i].precio+"</td>" +  
            "<td>"+  
            "<a onclick=detalles('"+cancionesMostrar[i]._id+"')>Detalles</a><br>" +  
            "<a onclick=eliminar('"+cancionesMostrar[i]._id+"')>Eliminar</a>" +  
            "</td>"+ "</tr>" );  
    }  
}  
cargarCanciones(); // ejecutar al cargar la página  
</script>
```

# Cliente jQuery-Ajax

- Paso 7: En **cliente.html** implementar la función **widgetCanciones()**
  - Se ejecuta al pulsar la opción del menú "Canciones".



The screenshot shows the uoMusic application interface. At the top is a dark header with the uoMusic logo and a hamburger menu icon. Below the header is a light gray button labeled 'Actualizar'. The main content area contains a table with three columns: 'Nombre', 'Genero', and 'Precio'. The table lists two songs: 'First Song' and 'Cat song', both in the 'pop' genre. For each song, there are two links: 'Detalles' and 'Eliminar'.

Nombre	Genero	Precio
First Song	pop	<a href="#">Detalles</a> <a href="#">Eliminar</a>
Cat song	pop	<a href="#">Detalles</a> <a href="#">Eliminar</a>

# Cliente jQuery-Ajax

- Para crear las demás funcionalidades de nuestra aplicación
  - Eliminar, insertar, editar, etc.
- Una vez obtenido el token de seguridad, debemos:
  - Crear la vista HTML correspondiente (widget)
  - Definir el Script que hará la petición a la API REST y recargará la información correspondiente.
  - Crear las opciones de menú correspondientes (si fuera necesario)
- Podemos incluir otras funcionalidades como *filtrado*, *ordenación*, *almacenar el token* en una Cookie para que la información no se pierda al refrescar el cliente, Etc.

# **CLIENTE REST EN NODE.JS**

# Cliente REST en Node.js

- Consumir un servicio web REST desde una app Node.js.
- Por ejemplo, el servicio web *REST comercial* <http://fixer.io/> que permite hacer transformaciones de divisas.
- Este servicio retorna un objeto JSON.
- En node.js *módulo request* permite hacer peticiones GET a servicios web externos
- *Vamos a incluir la posibilidad de poder los precios de una canción en otra divisa, ejemplo en dólares*



# Cliente REST en NODE.JS

- Paso 1: Install el módulo *request* de Node.js.

```
npm install request --save
```

- Paso 2: Incluimos el módulo en el fichero principal de **app.js**



```
var express = require('express');  
var app = express();  
var rest = require('request');  
app.set('rest', rest);
```

# Cliente REST en NODE.JS

- Paso 3: Por ejemplo, se llama la API REST desde, el controlador que implementa la función que responde a *GET/cancion/:id*
- Debemos declarar un objeto configuración especificando la información de la petición a realizar, **url, method, headers**
- El módulo necesita dos parámetros para realizar la petición:
  - El objeto de configuración
  - Función de callback con parámetros (error, response, body).

# Cliente REST en NODE.JS

- El servicio (<http://Fixer.io>) nos retorna un objeto, con las variables **base**, **date**, **rates** (objeto)

```
{"base":"EUR","date":"2017-10-12","rates":{"USD":1.1856}}
```

- Para obtener el precio en dólares hay que multiplicar el precio actual por el valor de la variable USD.
- Hay que agregar una nueva variable **usd** a la **canción** que se envía a la vista.

# Cliente REST en NODE.JS

```
app.get('/cancion/:id', function (req, res) {  
  var criterio = { "_id" : gestorBD.mongo.ObjectID(req.params.id)};  
  gestorBD.obtenerCanciones(criterio, function(canciones){  
    if ( canciones == null ){  
      res.send(respuesta);  
    } else {  
      var configuracion = {url: "http://api.fixer.io/latest?symbols=USD",  
                           method: "get",  
                           headers: {"token": "ejemplo",}}  
      var rest = app.get("rest");  
      rest(configuracion, function (error, response, body) {  
        console.log("cod: "+response.statusCode+" Cuerpo :"+body);  
        var objetoRespuesta = JSON.parse(body);  
        var cambioUSD = objetoRespuesta.rates.USD;  
        canciones[0].usd = cambioUSD * canciones[0].precio;  
        var respuesta = swig.renderFile('views/bcancion.html',  
                                         {cancion : canciones[0]});  
        res.send(respuesta);  
      })  
    }  
  });  
});
```

Configurar servicio

Llamar servicio

Parsear respuesta

Enviar respuesta

# Cliente REST en NODE.JS

- Paso 4: Finalmente mostramos el nuevo campo *cancion.usd* en la vista html.

```
<a href="/cancion/comprar/{{cancion._id.toString()}}">  
{{cancion.precio }} € - {{ cancion.usd }}$ </a>
```

4 € - 4.724 \$



Escuela de Ingeniería Informática

Escuela de Ingeniería Informática  
School of Computer Science Engineering

Universidad de Oviedo  
*Universidá d'Uviéu*  
*University of Oviedo*

# Sistemas Distribuidos e Internet

## Tema 9 Servicios Web SOAP y REST

**SOAP**  
Contract First Web Services

RESTful API  
GET PUT POST DELETE

**Dr. Edward Rolando Núñez Valdez**

nunezedward@uniovi.es