



Universidad  
Nacional  
de Quilmes

# RESTful APIs

*Construcción de Interfaces de Usuario*

2do Cuatrimestre 2019



# API

## *Application Programming Interface*

- Interfaz de comunicación cliente/servidor
- Actúa como protocolo
- Simplifica la construcción de apps cliente/servidor
- Funciona como “contrato” del server con el cliente
  - ▷ Si el pedido llega como se “espera”
  - ▷ Se garantiza “determinada” respuesta



# REST

## *REpresentational State Transfer*

- Estilo de arquitectura de software cliente/servidor
- Pensado para aplicaciones web
- Genera RESTful Web services
  - ▷ Transfieren el estado de la aplicación
  - ▷ Mediante operaciones RESTful



# RESTful

- Aquellos Web services que implementan la arquitectura REST
  - ▷ No se mantiene estado (*stateless*)
  - ▷ Funciona sobre protocolo HTTP
  - ▷ Define un conjunto operaciones
  - ▷ Requiere un formato de representación



# Stateless / Stateful

- Tipos de arquitecturas de aplicaciones
- Se diferencian en cómo manejan el estado de la aplicación



# Stateful

- El server mantiene el estado en memoria
- El cliente accede al estado en memoria
  - El acceso al estado es inmediato
  - A más clientes simultáneos, más memoria consumida
  - Limitaciones de hardware
  - Problemas de concurrencia
- Típicamente en arquitecturas Desktop



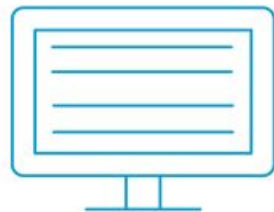
# Stateless

- Arquitectura cliente/servidor “a distancia”
- El server no necesita mantener el estado en memoria
  - Sí persistido (puede mantener “algo” memoria (performance))
- El cliente recibe una **representación del estado**
  - Acceso más lento, pero menor uso de recursos
  - Simplifica la concurrencia
- Típicamente en arquitecturas Web sobre HTTP

## YOUR SYSTEMS



## API PORTAL



**Your API Storefront**



## Developer Community



## Apps



Get your APIs to market on a portal





# HTTP

## *HyperText Transfer Protocol*

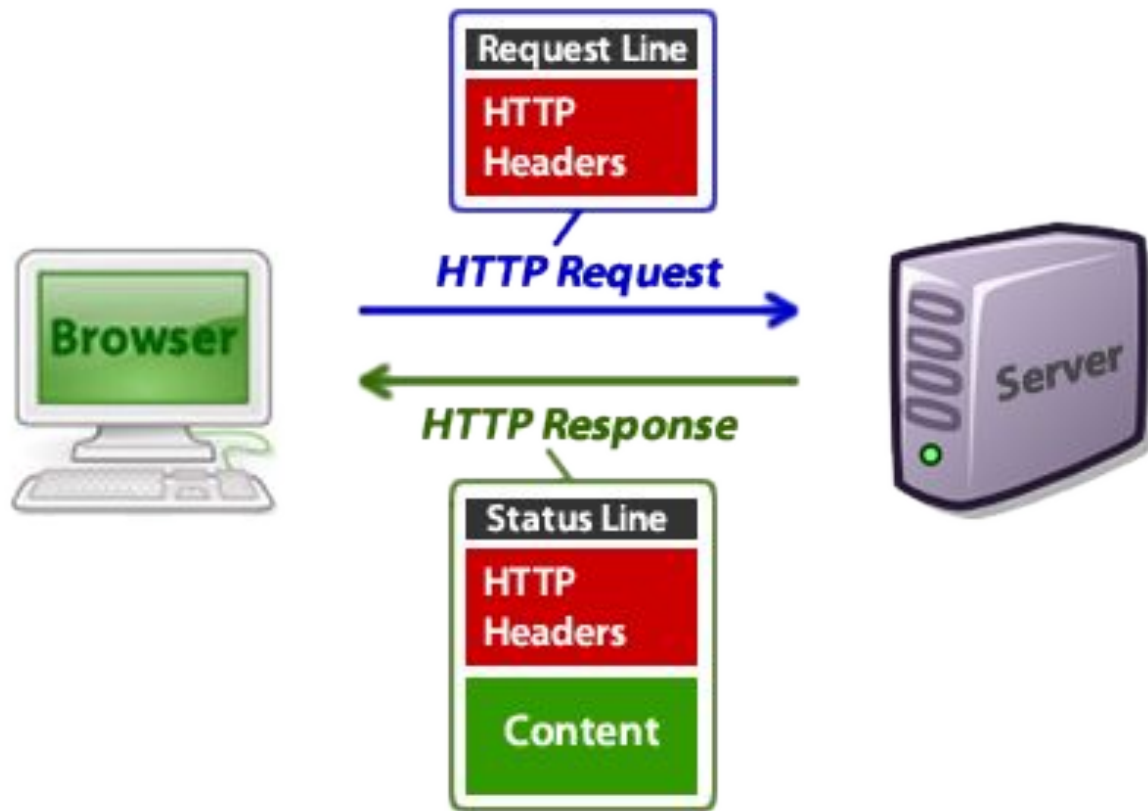
- *Hypertexto*: Texto con enlace a otros textos (links)
- Protocolo de Nivel 7 (en el Modelo OSI)
- Creado para la transferir información en la web
- Orientado a request/response en un esquema cliente/servidor
- *Stateless*: No mantiene estado, cada *response* retorna información según el *request*
- No mantiene información de requests/responses previos

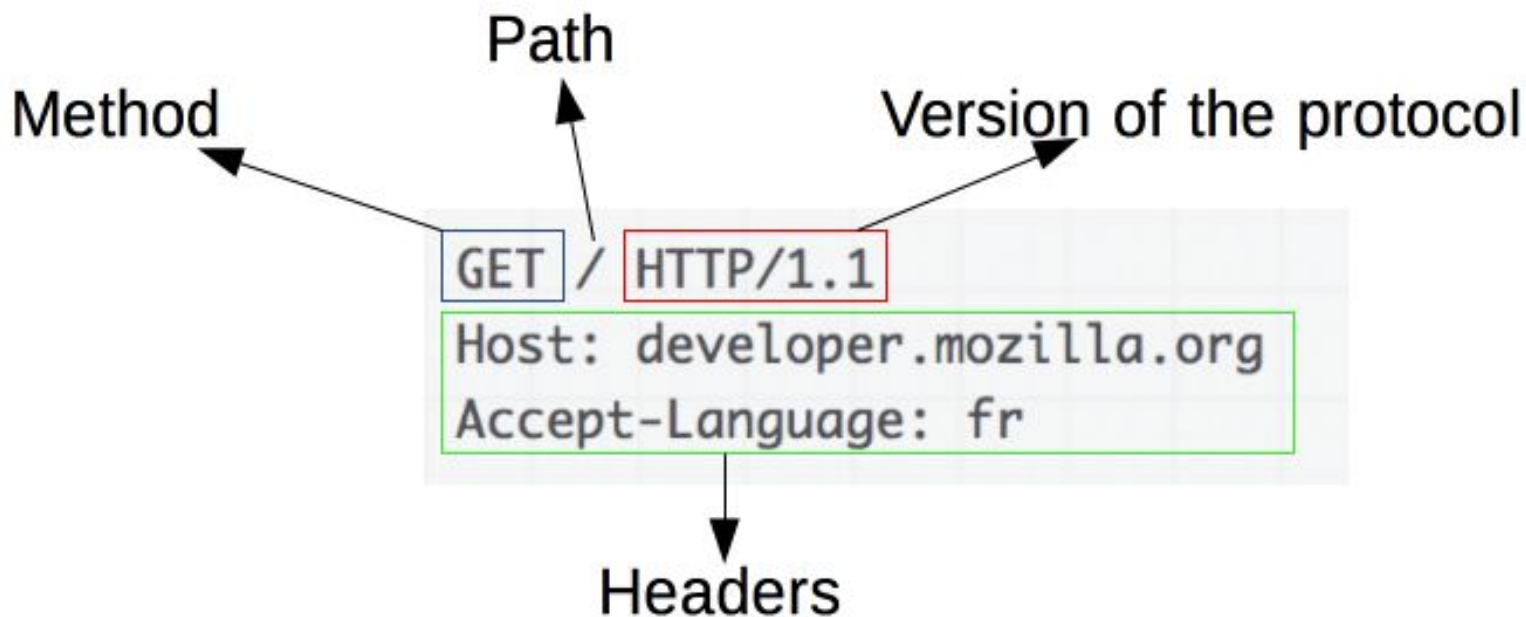


# Ciclo Request/Response

La comunicación entre cliente/servidor se da mediante requests/responses:

1. El cliente (browser) **envía** un *request* HTTP
2. El servidor web **recibe** el *request*
3. El server **procesa** el *request* ejecutando funciones en la aplicación
4. El servidor **retorna** un *response* HTTP al browser
5. El cliente (browser) **recibe** el *response* y ejecuta las funciones necesarias para mostrarle los resultados al usuario





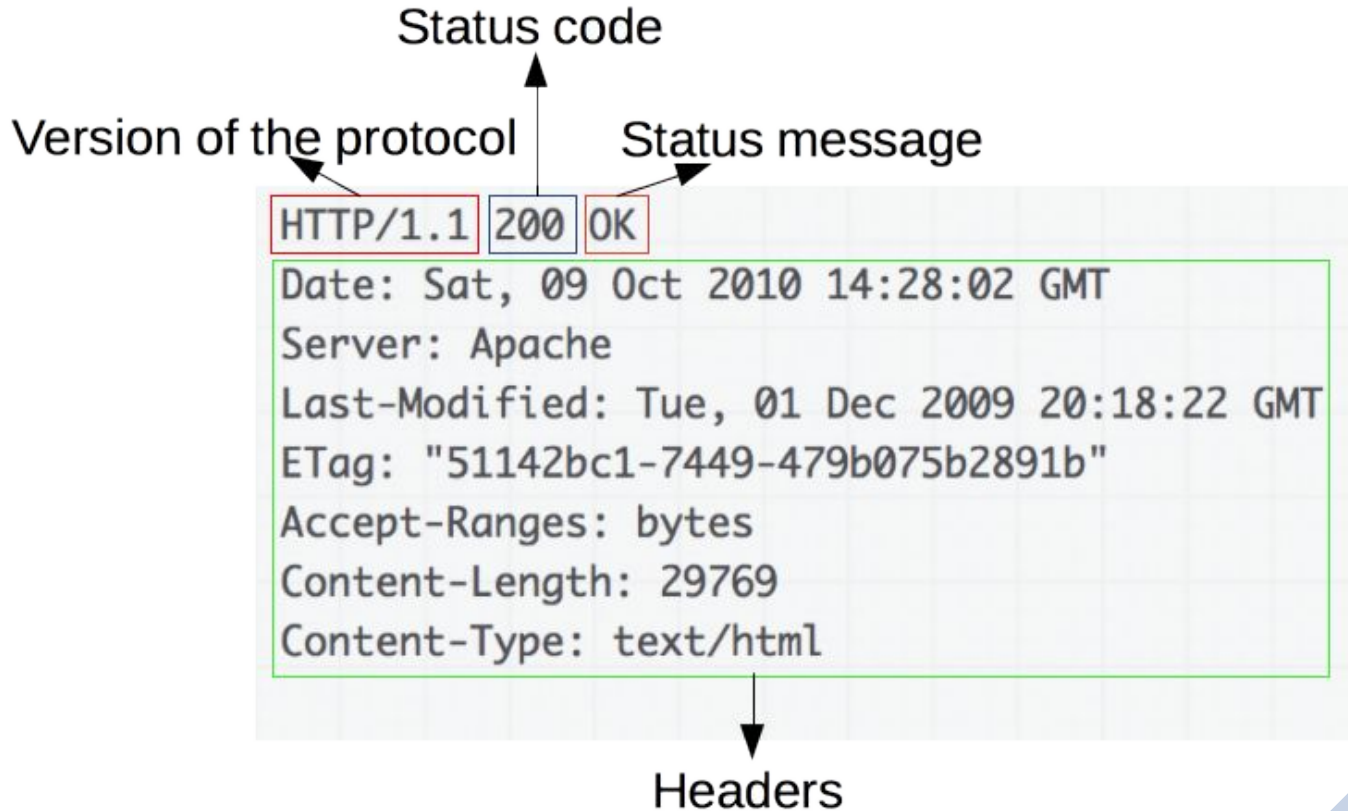


# HTTP Requests

HTTP define un conjunto de métodos (request) para indicar la acción que se desea que ejecute determinado recurso:

- **GET** solicita una representación de un determinado recurso
- **HEAD** Ídem **GET** pero no necesita el *body* en el response
- **POST** Crea un recurso específico. *Cambia el estado*
- **PUT** Reemplazar info de un recurso. *Cambia el estado*
- **DELETE** Borra un recurso específico. *Cambia el estado*

Hay algunos más, pero estos son los más frecuentes





# HTTP Response Codes

Además de responder con información del recurso, el servidor retorna un código para indicar qué tipo de respuesta se está retornando.

- **1xx** Códigos de Informational (poco usados)
- **2xx** Respuesta exitosa (salió todo bien)
- **3xx** Códigos para indicar redirección del recurso
- **4xx** Hubo errores a nivel de cliente
- **5xx** Hubo errores a nivel de servidor



# Responses Comunes (I)

- **200 OK** El request se procesó exitosamente
- **201 Created** Request exitoso y recurso creado (POST requests)
- **204 No Content** Request exitoso, no es necesario un body (DELETE)
- **301 Moved Permanently** El recurso se movió a una nueva URL
- **304 No Modified** El recurso no cambió desde el anterior request





## Responses Comunes (II)

- **400 Bad Request** El server no entiende el request (sintaxis inválida)
- **401 Unauthorized** Cliente no autenticado
- **403 Forbidden** Cliente autenticado pero sin permisos al recurso
- **404 Not Found** El server no encuentra el recurso solicitado
- **405 Method Not Allowed** Ruta válida, método no (POST, DELETE)
- **500 Internal Server Error** El server no responde



# URL

## *Uniform Resource Locator*

- Es lo que llamamos coloquialmente web address
- Referencia un recurso específico en un computadora en la web
- Tiene una estructura puntual para identificar sus partes

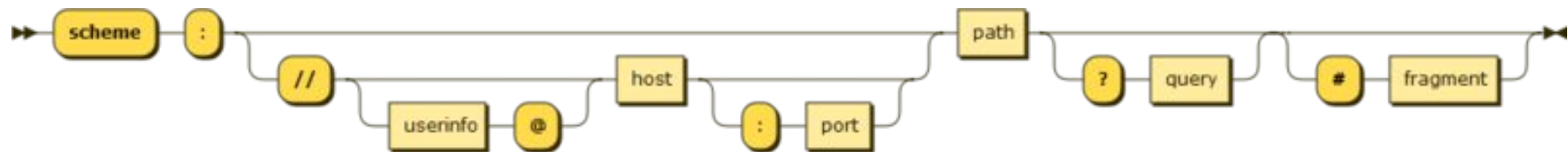
`http://localhost:4567/path/to/resource?att1=value1&att2=value2`

↓ ↓ ↓ ↓ ↓

**Esquema**      **Host**      **Puerto**      **Path**      **Fragmento**      **Query String**

↑ ↑ ↑ ↑

`https://blog.makeitreal.camp/post1#titutlo2`





# Repasando

Hasta acá vimos conceptos generales:

- Definición de API
- Arquitecturas RESTful
- Aplicaciones Stateful/Stateless
- Protocolo HTTP
  - ▷ Requests/Responses HTTP
  - ▷ URL

Vamos a meternos puntualmente como RESTful APIs



# APIs Web

- Implementan servicios
- Exponen recursos
- Permite interactuar con multiplicidad de tecnologías
- Establecen un protocolo de comunicación
- Agregan una capa de seguridad



# RESTful APIs

Servicios Web que son compatibles con los principios REST:

- Stateless
- Arquitectura Cliente-Servidor
- Uso de Caché
- Interfaz Uniforme



# API REST

## *Interface Uniforme*

- Identificación del Recurso
  - ▷ `usuarios/`, `restaurants/`, `pedidos/`, etc...
- Operaciones bien definidas
  - ▷ `GET`, `POST`, `PUT`, `DELETE`, etc...
- Sintaxis Universal
  - ▷ `GET usuarios/`, `DELETE usuarios/`, etc...
- Hypermedia
  - ▷ `application/json`, `text/html`, etc...



# API REST

## Formato de Intercambio

Se necesita un formato bien definido para intercambiar información. Los dos formatos más extendidos actualmente son:

### JSON

*JavaScript Object Notation*

```
{
  "credentials": {
    "username": "hodor",
    "password": "hodor"
  }
}
```

### XML

*Extensible Markup Language*

```
<credentials>
  <username>hodor</username>
  <password>hodor</password>
</credentials>
```





# JSON

## *Reglas de Sintaxis*

La sintaxis de JSON deriva de la sintaxis de notación de objetos de JavaScript:

- Información como par **“key”:****“value”**
- Datos separados por coma (,)
- Las llaves ({} ) contienen objetos
- Los corchetes ([]) contienen listas



# JSON

## *Tipos de datos*

Las **keys** son “*strings*”. Los **value** pueden ser:

- String      `"algún texto"`
- Number     `1 | -1 | 1.2 | -1.0`
- Object      `{"key": "value"}`
- List        `[1, "dos", {"val": 3.0}]`
- Boolean    `true | false`
- Null        `null`

```
{  
  "lugar": "Universidad Nacional de Quilmes",  
  "coordenadas": {  
    "latitud": -34.706294,  
    "longitud": -58.278522  
  },  
  "distancias": [  
    {  
      "lugar": "Obelisco",  
      "kms": 14.81,  
    }, {  
      "lugar": "Mendoza",  
      "kms": 996.52  
    },  
  ]  
}
```



# API REST

## REQUESTs

Para cada **REQUEST**, en una API REST se define la estructura a la cual el cliente se debe ajustar para recuperar o modificar un recurso. En general consiste de:

- **Verbo HTTP:** operación realizar (GET, POST, PUT, ...)
- **Protocolo aceptado:** HTTP 1.1, HTTP 1.0
- **Media Data** de intercambio aceptada: html, json, xml, ...
- **Encabezado:** (opcional) permite pasar información extra
- **Ruta** al recurso
- **Cuerpo de mensaje** (opcional) que contiene datos



# API REST

*REQUESTs >> Ejemplos*

**GET /users/23 HTTP/1.1**

**Accept:** text/html, application/json

**POST /users HTTP/1.1**

**Accept:** application/json

**Body:** {"user": {  
    "name": "Arya Stark"  
    "email": "nobody@braavos.org"  
}}



# API REST

## RESPONSEs

Por cada **REQUEST** que se recibe se debe retornar un **RESPONSE** con la información necesaria para describir lo que ocurrió:

- **HTTP Code** acorde a lo sucedido con la ejecución
- **Protocolo** de respuesta
- **Media-data** de la respuesta (idealmente según lo pedido en el request)
- **Cuerpo de mensaje** (si corresponde) con la información requerida



# API REST

## RESPONSEs » Ejemplo (I)

**GET /users/hodor** HTTP/1.1

**Accept:** text/html, application/json

HTTP/1.1 **200** (OK)

**Content-Type:** application/json

**Body:** {"user": {  
 "Username": "hodor",  
 "name": "Hodor",  
 "email": "[hodor@winterfell.com](mailto:hodor@winterfell.com)" }}



# API REST

## *RESPONSEs » Ejemplo (II)*

**POST /users HTTP/1.1**

```
Body: {"user": {  
  "username": "nymeria"  
  "name": "Arya Stark",  
  "email": "nobody@braavos.org" }}
```

**HTTP/1.1 201 (CREATED)**

**Content-type:** application/json





# CRUDs

- Generalmente nuestro modelo de negocio necesita poder
  - ▷ **crear, leer, actualizar y eliminar** recursos
  - ▷ (**C**reate, **R**ead, **U**ppdate, **D**elelete).
- A esto se le llama **CRUD**.
- Es la funcionalidad mínima que se espera de un modelo.
- El paradigma CRUD es muy común en la construcción de aplicaciones web
- Proporciona un modelo mental sobre los recursos de manera que sean completos y utilizables.



# CRUD » Estándares

Los *CRUD* se suelen armar respetando el siguiente estándar:

- Crear **POST** **/users**
- Leer (todos) **GET** **/users**
- Leer (uno) **GET** **/users/:id**
- Actualizar **PUT** **/users/:id**
- Eliminar **DELETE** **/users/:id**



# CRUD » Respuestas

## ■ **POST /users**

- 201 (Created)
- {"user": Nuevo Usuario}

## ■ **GET /users**

- 200 (OK)
- {"users": [Listado]}

## ■ **GET /users/:id**

- 200 (OK)
- {"user": Usuario Pedido}

## ■ **PUT /users/:id**

- 200 (OK)
- {"user": Usuario Actualiz.}

## ■ **DELETE /users/:id**

- 204 (No Content)
- Body: Vacío



# CRUD » Errores

## ■ **POST /users**

- ▷ 404 (Not Found)
- ▷ 409 (Conflict)

## ■ **GET /users/:id**

- ▷ 404 (Not Found)

## ■ **PUT /users/:id**

- ▷ 404 (Not Found)
- ▷ 409 (Conflict)

## ■ **DELETE /users/:id**

- ▷ 404 (Not Found)
- ▷ 405 (Method Not Allowed)



# Errores Genéricos

- 401 (Unauthorized)
- 403 (Forbidden)
- 405 (Method Not Allowed)
- 500 (Internal Server Error)



# Query Parameters

- Muchas veces es necesario agregar información a la solicitud.
- Ya sea para filtrar una búsqueda
- o bien para que la respuesta incluya más o menos información.
- Para estos casos se suelen utilizar parámetros de consulta (*query parameters*).
- Se escriben como un par **clave=valor** separados por **&**.



# Query » Ejemplos

- GET `/users?mail=gmail&born_in=1990`
  - GET `/users/123?include=orders`
  - GET `/users?page=3&per_page=25`
- 
- No es buena práctica incluir parámetros en otros métodos que no sean GET (de consulta)
  - Para enviar información (POST, PUT) se usa el *body*



# A simple web framework for Java and Kotlin

Java

Kotlin

Copy

```
import io.javalin.Javalin

fun main(args: Array<String>) {
    val app = Javalin.create().start(7000)
    app.get("/") { ctx -> ctx.result("Hello World") }
}

// You can wrap the main function
// in a Kotlin object
```





# LINKS ÚTILES

- <https://developer.mozilla.org/en-US/docs/Web/HTTP>
- <https://json.org/json-es.html>
- <https://www.restapitutorial.com/>
- <https://jsonapi.org/>
- <https://www.codecademy.com/articles/what-is-rest>
- <https://www.codecademy.com/articles/what-is-crud>
- <https://javalin.io/>
- <https://github.com/toddmotto/public-apis>



¿Preguntas?