CODER HOUSE

Clase 7. Programación Backend

Express Avanzado

OBJETIVOS DE LA CLASE

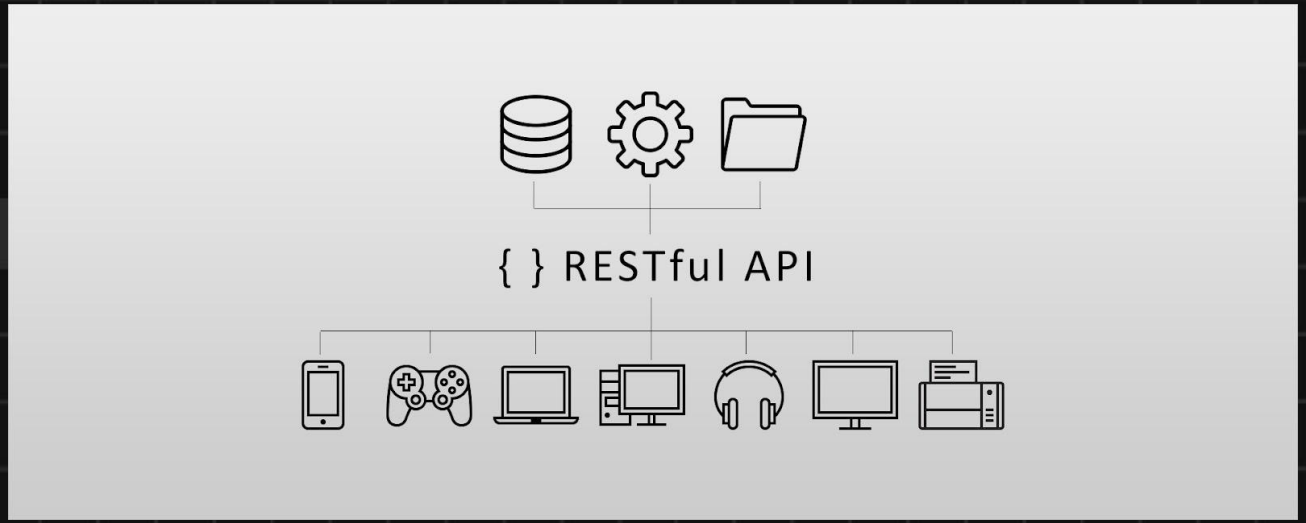
- Repasar el funcionamiento del protocolo HTTP y su uso en una aplicación RESTful.
- Comprender el concepto de API REST
- Implementar los verbos get, post, put y delete en el servidor basado en Express.
- Usar Postman para generar request.

CRONOGRAMA DEL CURSO

Clase 6	Clase 7	Clase 8
Servidores Web	**Express Avanzado**	**Router & Multer**

CODER HOUSE

Aplicaciones RESTful



CODER HOUSE

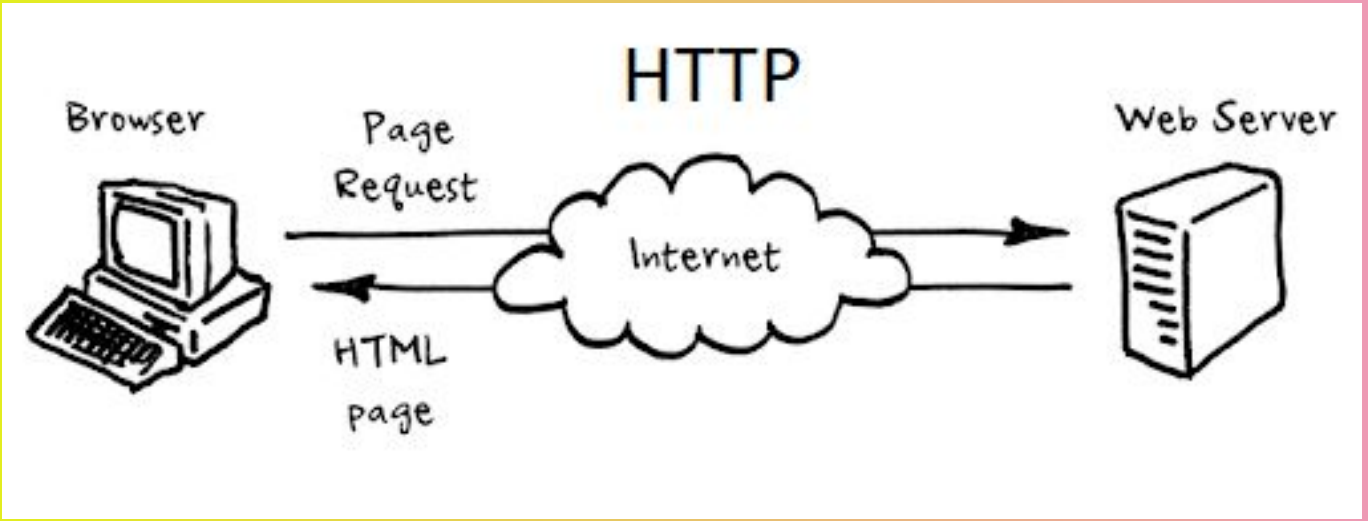
Introducción

RESTless		RESTful	
VERB	HREF	VERB	URI
POST	/users/create	POST	/users
GET	/users/1	GET	/users/1
POST	/users/1/update	PUT	/users/1
????	/users/1/delete	DELETE	/users/1

Existen diversos tipos de aplicaciones: uno de los tipos más nombrados en la actualidad son las **Aplicaciones RESTFul**.

Cuando hablamos de aplicaciones RESTful, nos referimos a aplicaciones que operan en forma de servicios web, respondiendo consultas a otros sistemas a través de internet. Dichas aplicaciones lo hacen respetando algunas reglas y convenciones que detallaremos a lo largo de esta clase

Protocolo HTTP

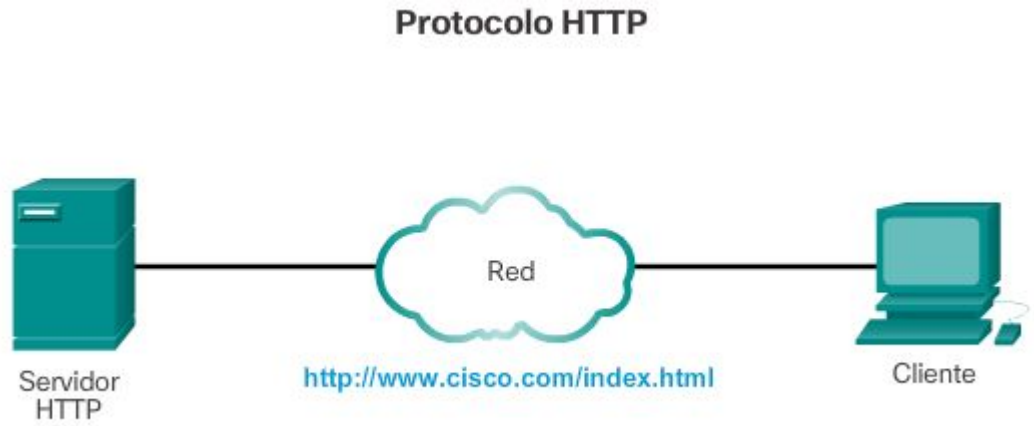


CODER HOUSE

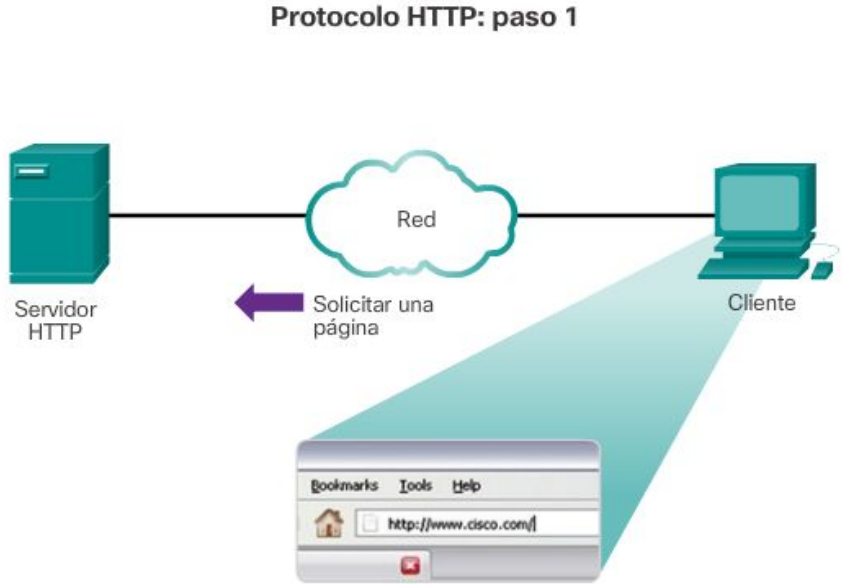
Repasemos el protocolo HTTP

- **HTTP** *(Hypertext Transfer Protocol o Protocolo de Transferencia de HiperTexto)* es, como su nombre lo dice, un protocolo (conjunto de reglas y especificaciones) que se utiliza a la hora de **intercambiar datos a través de internet**.
- El protocolo se basa en un **esquema** de **petición-respuesta**.
- Existen **clientes** que realizan solicitudes de transmisión de datos, y un **servidor** que atiende la peticiones.
- HTTP establece varios tipos de peticiones, siendo las principales: *POST, GET, PUT, y DELETE*.

Etapas de comunicación HTTP



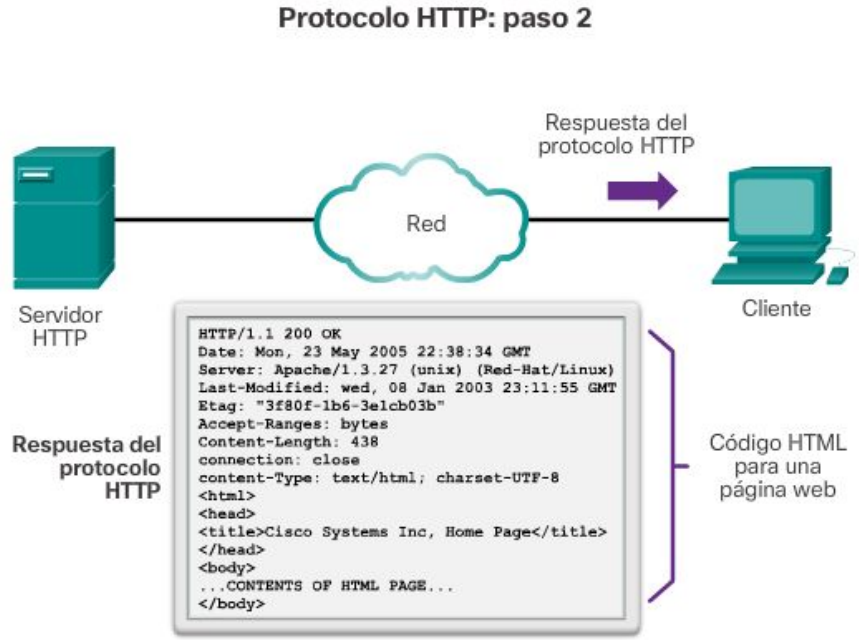
Etapas de comunicación HTTP



El cliente inicia la solicitud de protocolo HTTP a un servidor.

CODER HOUSE

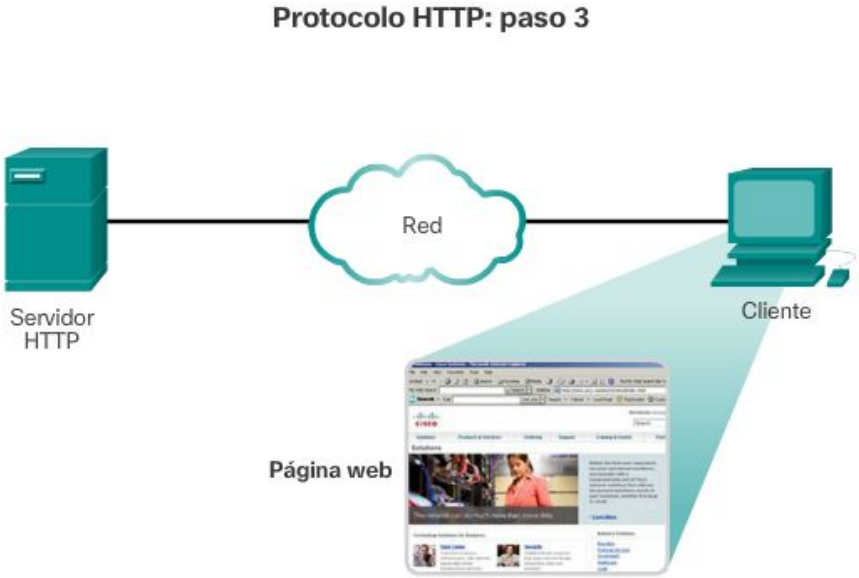
Etapas de comunicación HTTP



En respuesta a la solicitud, el servidor HTTP envía el código para una página web.

CODER HOUSE

Etapas de comunicación HTTP



El navegador interpreta el código HTML y muestra una página web.

HTTP: Códigos de estado

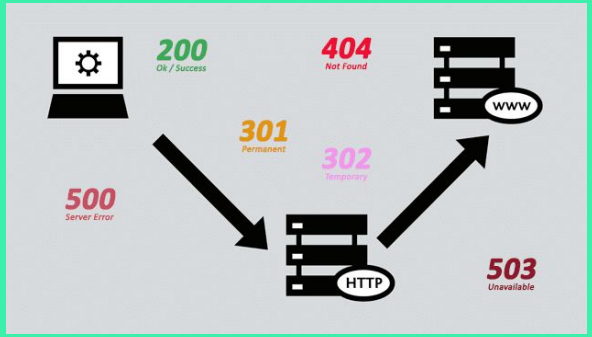
Cada mensaje de respuesta de HTTP tiene un **código** de estado numérico de tres cifras que indica el **resultado** de la petición

- ***1xx*** *(Informativo):* La petición fue recibida, y continúa su procesamiento.
- ***2xx*** *(Éxito):* La petición fue recibida con éxito, comprendida y procesada.
- ***3xx*** *(Redirección):* Más acciones son requeridas para completar la petición.
- ***4xx*** *(Error del cliente):* La petición tiene algún error, y no puede ser procesada.
- **5xx** (Error del servidor): El servidor falló al intentar procesar una petición aparentemente válida.

Códigos de Estado más comunes

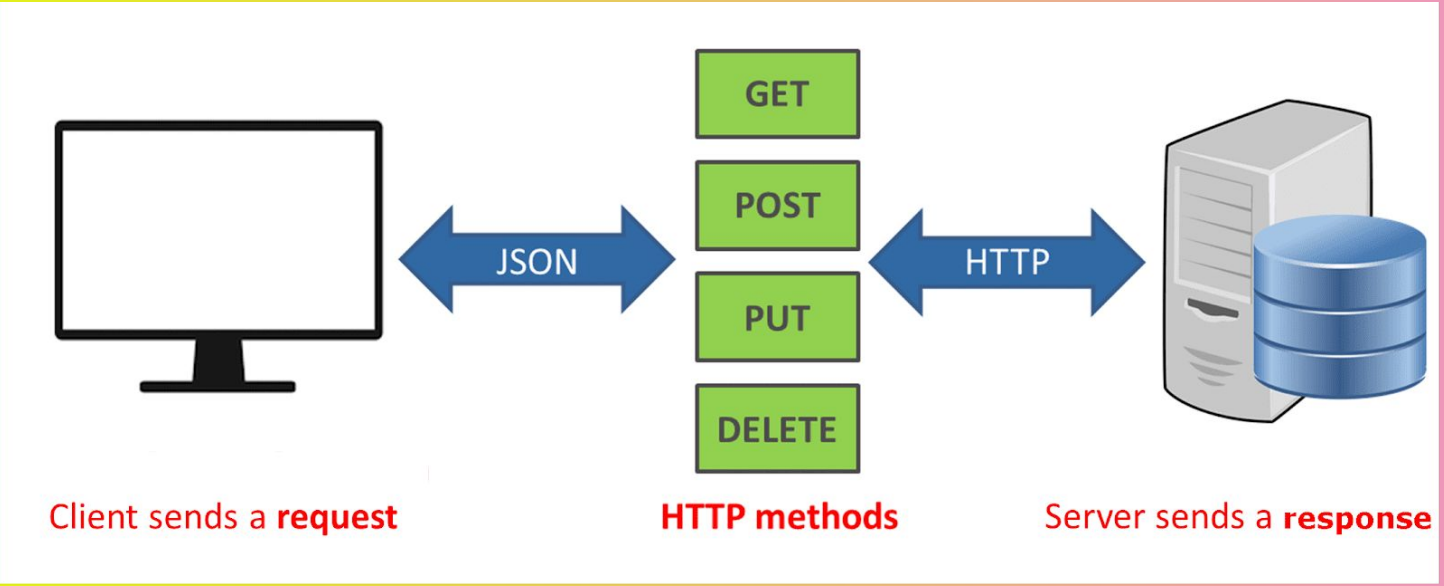
200	OK	Todo salió como lo esperado
400	Bad Request	La petición no cumple con lo esperado
404	Not Found	El recurso buscado no existe (URI inválido)
500	Internal Server Error	Error genérico del servidor al procesar una petición válida

- **1xx**: Mensaje informativo.
- **2xx:** Exito
  - 200 OK
  - 201 Created
  - 202 Accepted
  - 204 No Content
- **3xx:** Redirección
  - 300 Multiple Choice
  - 301 Moved Permanently
  - 302 Found
  - 304 Not Modified
- **4xx:** Error del cliente
  - 400 Bad Request
  - 401 Unauthorized
  - 403 Forbidden
  - 404 Not Found
- **5xx:** Error del servidor
  - 500 Internal Server Error
  - 501 Not Implemented
  - 502 Bad Gateway
  - 503 Service Unavailable

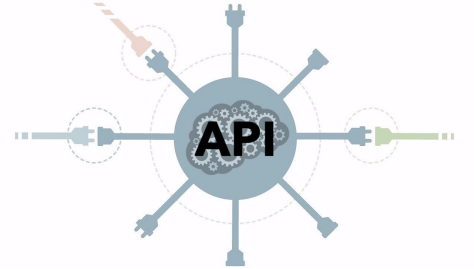


CODER HOUSE

Conceptos de API, REST y API REST



CODER HOUSE



¿Qué es una API?

- Una **API** es un conjunto de reglas y especificaciones que describen la manera en que un sistema puede comunicarse con otros.
- Definir una API en forma clara y explícita habilita y facilita el intercambio de mensajes entre sistemas.
- Permite la colaboración e interoperabilidad entre los sistemas desarrollados en distintas plataformas e incluso en distintos lenguajes.
- La **API** puede tener interfaz gráfica o ser de uso interno.
- La **API** tiene que estar acompañada con la documentación detallada que describa su operación y el formato de interacción con la misma.

{ REST } ¿Qué es REST?

- **REST** viene del inglés “REpresentational State Transfer” (o en español: Transferencia de Estado Representacional).
- *Por Representación* nos referimos a un modelo o estructura con la que representamos algo.
- *Por Estado* de una representación, hablamos de los datos que contiene ese modelo estructura.
- Transferir un Estado de Representación implica el envío de datos (con una determinada estructura) entre dos partes.
- Los dos formatos más utilizados para este tipo de transferencias de datos son **XML y JSON.**

Formatos XML y JSON

- XML

```xml
<factura>
  <cliente>Gomez</cliente>
  <emisor>Perez S.A.</emisor>
  <tipo>A</tipo>
  <items>
    <item>Producto 1</item>
    <item>Producto 2</item>
    <item>Producto 3</item>
  </items>
</factura>
```

- JSON

```json
{
  "cliente":"Gomez",
  "emisor":"Perez S.A.",
  "tipo":"A",
  "items": [
    "Producto 1",
    "Producto 2",
    "Producto 3"
  ]
}
```

¿Qué es API REST?

- Es un tipo de API que **no dispone de interfaz gráfica**.
- Se utiliza exclusivamente para **comunicación entre sistemas**, mediante el **protocolo HTTP**.
- Para que una API se considere REST, debe cumplir con las siguientes características:
  - *Arquitectura Cliente-Servidor sin estado*
  - *Cacheable*
  - *Operaciones comunes*
  - *Interfaz uniforme*
  - *Utilización de hipermedios*

Características API REST

CODER HOUSE

Arquitectura Cliente-Servidor sin estado

- Cada mensaje HTTP contiene toda la información necesaria para comprender la petición.
- Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes.
- Esta restricción mantiene al **cliente** y al **servidor débilmente acoplados**: el cliente no necesita conocer los detalles de implementación del servidor y el servidor se “despreocupa” de cómo son usados los datos que envía al cliente.

Cacheable

- Debe admitir un **sistema de almacenamiento en caché**.
- La infraestructura de red debe soportar una caché **de varios niveles**.
- Este almacenamiento **evita repetir** varias **conexiones** entre el servidor y el cliente, en casos en que peticiones idénticas fueran a generar la misma respuesta.

Operaciones comunes

- Todos los recursos detrás de nuestra API deben poder ser **consumidos** mediante **peticiones HTTP**, preferentemente sus principales (POST, GET, PUT y DELETE).
- Con frecuencia estas operaciones se equiparan a las operaciones CRUD en bases de datos (en inglés: Create, Read, Update, Delete, en español: Alta, Lectura, Modificación, y Baja).
- Al tratarse de peticiones HTTP, éstas deberán **devolver** con sus respuestas los correspondientes **códigos de estado**, informando el resultado de las mismas.

Interfaz uniforme

- En un sistema REST, cada acción (más correctamente, cada recurso) debe contar con una URI (Uniform Resource Identifier), un identificador único.
- Ésta nos facilita el acceso a la información, tanto para consultarla, como para modificarla o eliminarla, pero también para compartir su ubicación exacta a terceros.

Utilización de hipermedios

- Cada vez que se hace una petición al servidor y este devuelve una respuesta, parte de la información devuelta pueden ser también hipervínculos de navegación asociada a otros recursos del cliente.
- Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.


TU Y YO PROGRAMANDO UN REST API
NO SÉ... PIÉNSALO

BREAK

¡5/10 MINUTOS Y VOLVEMOS!

Aplicación RESTful

CODER HOUSE

Principios

- **Una aplicación RESTful** requiere un enfoque de diseño distinto a la forma típica de pensar en un sistema: lo **contrario a RPC**
- **RPC** (*Remote Procedure Calls*, llamadas a procedimientos remotos) basa su funcionamiento en las operaciones que puede realizar el sistema (acciones, usualmente verbos). *Ej: getUsuario()*
- En **REST**, por el contrario, el **énfasis** se pone **en los recursos** (usualmente sustantivos), especialmente en los nombres que se le asigna a cada tipo de recurso. *Ej. Usuarios.*
- Cada funcionalidad relacionada con este recurso tendría sus propios identificadores y peticiones en HTTP.

Por ejemplo...

- **Listar usuarios:** Petición HTTP de tipo GET a la URL: http://servicio/api/usuarios

- **Agregar usuario:** Petición HTTP de tipo POST a la URL: http://servicio/api/usuarios (Agregando a la petición el registro correspondiente con los datos del nuevo usuario)

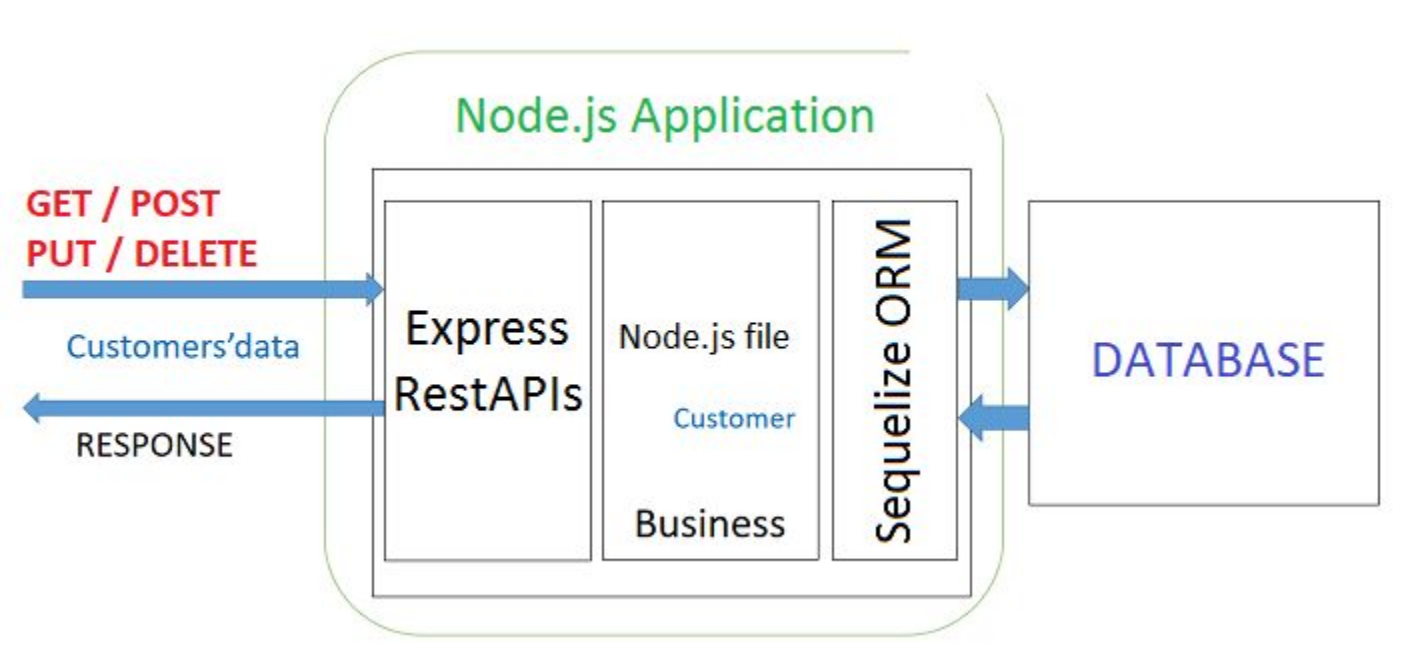
- **Obtener al usuario 1:** En caso de querer acceder a un elemento en particular dentro de un recurso, se lo puede hacer fácilmente si se conoce su identificador (URI): Petición HTTP de tipo GET a la URL: http://servicio/api/usuarios/1

Por ejemplo...

- **Modificar al usuario 1:** Para actualizar un dato del usuario, un cliente REST podría primero descargar el registro anterior usando GET. El cliente después modificaría el objeto para ese dato, y lo enviaría al servidor utilizando una petición HTTP de tipo PUT a la URL: http://servicio/api/usuarios/1.

- **Obtener usuarios con domicilio en CABA:** Si en cambio es necesario realizar una búsqueda por algún criterio, se pueden enviar parámetros en una petición HTTP. Éstos se pueden añadir al final de la misma con la siguiente sintaxis:
  Petición HTTP de tipo GET a la URL:
  http://servicio/api/usuarios?domicilio=CABA

Manejo de peticiones HTTP con Express



Express: atención de peticiones

- Para definir cómo se debe manejar cada tipo de petición usaremos los métodos nombrados de acuerdo al tipo de petición que manejan: ***get(), post(), delete(), y put()***.
- Todos reciben como primer argumento la ruta que van a estar escuchando, y *solo manejarán* peticiones que coincidan en ruta y en tipo. Luego, el segundo argumento será el callback con que se manejará la petición.
- Está tendrá dos parámetros: el primero con la petición (request) en sí y el segundo con la respuesta (response) que espera devolver.

Ejemplo de petición GET (Pedir)

Cada tipo de petición puede tener diferentes características. Por ejemplo, algunas peticiones **no requieren el envío de ningún dato extra** en particular para obtener el recurso buscado. Este es el caso de la petición GET. Como respuesta a la petición, **devolverá** el **resultado** deseado en **forma de objeto**.

```
app.get('/api/mensajes', (req, res) => {
    console.log('request recibido')

    // acá debería obtener todos los recursos de tipo ‘mensaje’

    res.json({ msg: 'Hola mundo!'})
})
```

Ejemplo de petición GET con parámetros de búsqueda

Las **peticiones** pueden **incorporar detalles** sobre la búsqueda que se quiere realizar.

- Estos parámetros se agregan al final de la URL, mediante un signo de interrogación ‘?’ y enumerando pares ‘clave=valor’ separados por un ampersand ‘&’ si hay más de uno.
- Al recibirlos, los mismos se encontrarán en el objeto ‘query’ dentro del objeto petición (req).

```
app.get('/api/mensajes', (req, res) => {
    console.log('GET request recibido')

    if (Object.entries(req.query).length > 0) {
        res.json({
            result: 'get with query params: ok',
            query: req.query
        })
    } else {
        res.json({
            result: 'get all: ok'
        })
    }
})
```

Ejemplo de petición GET con identificador

En caso de que se quiera acceder a un recurso en particular ya conocido, es necesario **enviar un identificador unívoco** en la URL.

- Para enviar este tipo de parámetros, el mismo se escribirá luego del nombre del recurso (en la URL), separado por una barra.
  Por ejemplo: ***http://miservidor.com/api/mensajes/1***

  (En este ejemplo estamos queriendo acceder al mensaje nro 1 de nuestros recursos.)

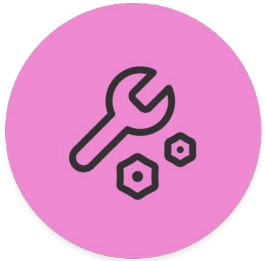
Ejemplo de petición GET con identificador

- Para acceder al campo identificador desde el lado del servidor, Express utiliza una sintaxis que permite indicar anteponiendo **‘dos puntos’** antes del nombre del campo identificador, al especificar la ruta escuchada. Luego, para acceder al valor del mismo, se hará a través del **campo ‘params’** del objeto petición (req) recibido en el callback.

```
app.get('/api/mensajes/:id', (req, res) => {
    console.log('GET request recibido')

    // acá debería hallar y devolver el recurso con id == req.params.id

    res.json(elRecursoBuscado)
})
```



Get endpoints

Vamos a practicar lo aprendido hasta ahora

Tiempo: 10 minutos

CODER HOUSE

Desafío
generico

Dada la siguiente constante: `const frase = 'Hola mundo cómo están'`

Realizar un servidor con API Rest usando node.js y express que contenga los siguientes endpoints get:

1) '/api/frase' -> devuelve la frase en forma completa en un campo ‘frase’.

2) '/api/letras/:num -> devuelve por número de orden la letra dentro de esa frase (num 1 refiere a la primera letra), en un campo ‘letra’.

3) '/api/palabras/:num -> devuelve por número de orden la palabra dentro de esa frase (num 1 refiere a la primera palabra), en un campo ‘palabra’.

Aclaraciones:

- En el caso de las consignas 2) y 3), si se ingresa un parámetro no numérico o que esté fuera del rango de la cantidad total de letras o palabras (según el caso), el servidor debe devolver un objeto con la descripción de dicho error. Por ejemplo:

➢ { error: "El parámetro no es un número" } cuando el parámetro no es numérico

➢ { error: "El parámetro está fuera de rango" } cuando no está entre 1 y el total de letras/palabras

- El servidor escuchará peticiones en el puerto 8080 y mostrará en la consola un mensaje de conexión que muestre dicho puerto, junto a los mensajes de error si ocurriesen.

Otras operaciones

Ejemplo de petición POST (Enviar)

Algunas peticiones requieren el **envío** de algún **dato** desde el **cliente hacia** el **servidor**. Por ejemplo, al crear un nuevo registro. Este es el caso de la petición **POST**. Para acceder al cuerpo del mensaje, incluído en la petición, lo haremos a través del campo 'body' del objeto petición recibido en el callback. En este caso, estamos devolviendo como respuesta el mismo registro que se envió en la petición.

```
app.post('/api/mensajes', (req, res) => {
    console.log('POST request recibido')

    // acá debería crear y guardar un nuevo recurso
    // const mensaje = req.body
```

Ejemplo de petición PUT (Actualizar)

También es posible mezclar varios mecanismos de pasaje de datos/parámetros, como es el caso de las peticiones de tipo PUT, en las que se desea actualizar un registro con uno nuevo.

- Se debe proveer el identificador del registro a reemplazar y el dato con el que se lo quiere sobreescribir.

```
app.put('/api/mensajes-json/:id', (req, res) => {
    console.log('PUT request recibido')

    // acá debo hallar al recurso con id == req.params.id
    // y luego reemplazarlo con el registro recibido en req.body

    res.json({
        result: 'ok',
        id: req.params.id,
        nuevo: req.body
    })
})
```

Ejemplo de petición DELETE (Borrar)

Si quisiéramos **eliminar** un recurso, debemos **identificar unívocamente** sobre cuál de todos los disponibles se desea realizar la operación.

```
app.delete('/api/mensajes/:id', (req, res) => {
    console.log('DELETE request recibido')

    // acá debería eliminar el recurso con id == req.params.id

    res.json({
        result: 'ok',
        id: req.params.id
    })
})
```

¡Importante! Configuración extra

Para que nuestro servidor express pueda interpretar en forma automática mensajes de tipo **JSON** en formato **urlencoded** al recibirlos, debemos indicarlo en forma explícita, agregando las siguiente líneas luego de crearlo.

```
app.use(express.json())
app.use(express.urlencoded({ extended: true }))
```

Aclaración: *{extended:true} precisa que el objeto req.body contendrá valores de cualquier tipo en lugar de solo cadenas.*

¡Sin esta línea, el servidor no sabrá cómo interpretar los objetos recibidos!

POSTMAN

API Testing and Automation

Postman

PUBLISH

Onboard developers to your API faster with Postman collections and documentation

DESIGN & MOCK

Design in Postman & use Postman's mock service

MONITOR

Create automated tests to monitor APIs for uptime, responsiveness, and correctness

DEBUG

Test APIs, examine responses, add tests and scripts

DOCUMENT

Create beautiful web-viewable documentation

AUTOMATED TESTING

Run automated tests using the Postman collection runner

CODER HOUSE

¿Qué es Postman?

Postman nace como una herramienta que principalmente nos permite crear peticiones sobre APIs de una forma muy sencilla y de esta manera, probar las APIs.

- El usuario de Postman puede ser un desarrollador que esté comprobando el funcionamiento de una API para desarrollar sobre ella o un operador que esté realizando tareas de monitorización **sobre una API.**

- **Instalación:** https://www.postman.com/downloads/

Download Postman | Try Postma

postman.com/downloads/

POSTMAN Product Use Cases Pricing Enterprise Explore Learning Center Sign In Sign Up

Download Postman

Download the app to quickly get started using the Postman API Platform. Or, if you prefer a browser experience, you can try the new web version of Postman.

The Postman app

The ever-improving Postman app (a new release every two weeks) gives you a full-featured Postman experience.

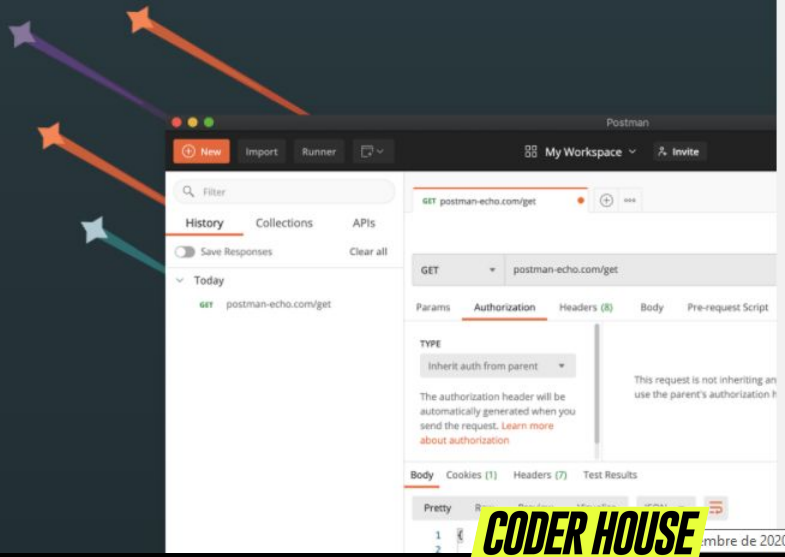
Download the App

By downloading and using Postman, I agree to the Privacy Policy and Terms.

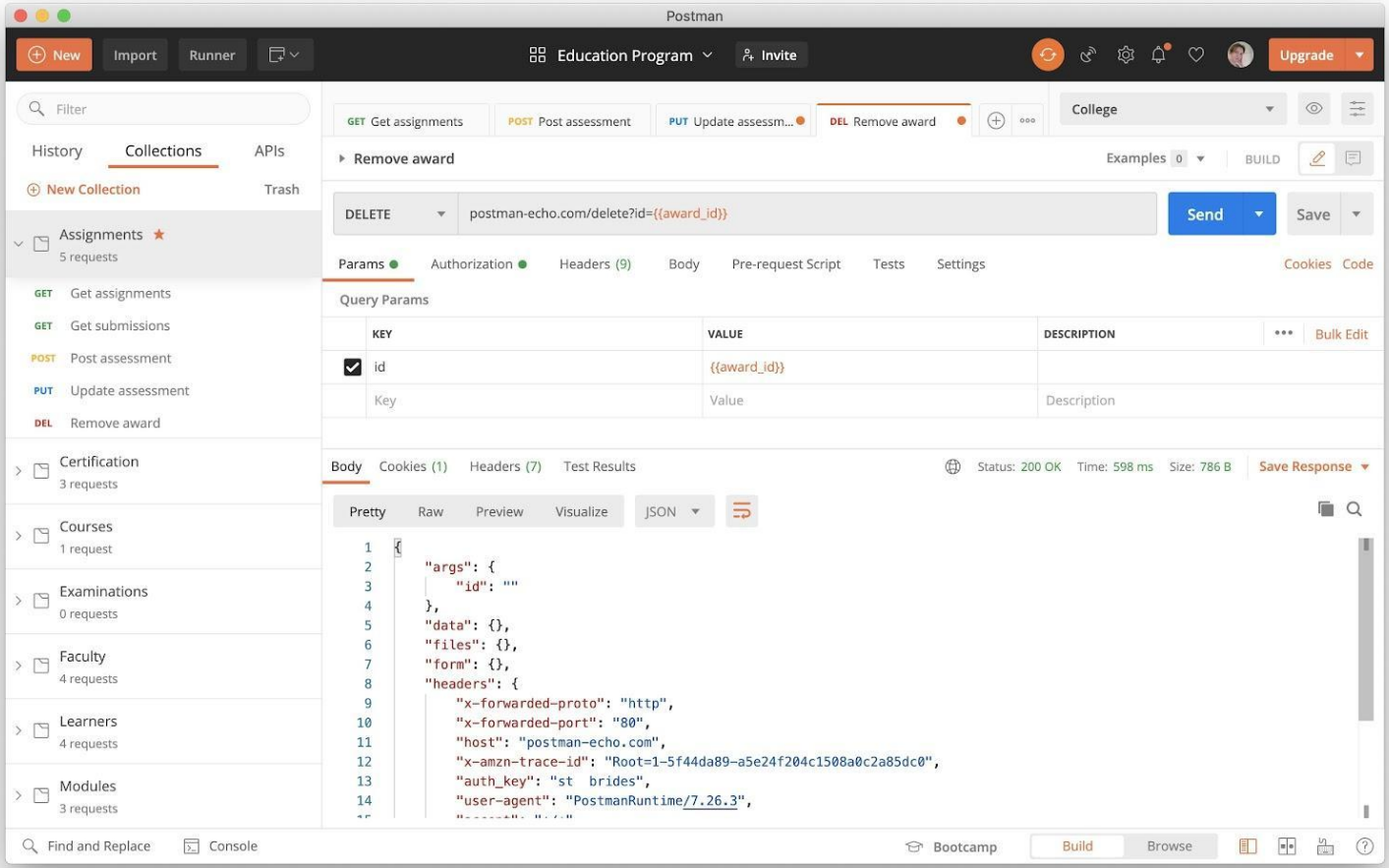
Version 7.36.0 | Release Notes | Product Roadmap

Not your OS? *Download for Mac (macOS) or Linux (x64)*

Postman on the web



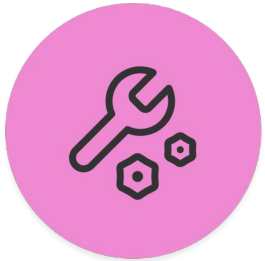
CODER HOUSE


Postman
New
Import
Runner
Education Program
Invite
Upgrade
Filter
History
Collections
APIs
New Collection
Trash
Assignments
5 requests
GET Get assignments
GET Get submissions
POST Post assessment
PUT Update assessment
DEL Remove award
Certification
3 requests
Courses
1 request
Examinations
0 requests
Faculty
4 requests
Learners
4 requests
Modules
3 requests
GET Get assignments
POST Post assessment
PUT Update assessm...
DEL Remove award
College
Remove award
Examples 0
BUILD
DELETE
postman-echo.com/delete?id={{award_id}}
Send
Save
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
Cookies
Code
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
id
{{award_id}}
Key
Value
Description
Body
Cookies (1)
Headers (7)
Test Results
Status: 200 OK
Time: 598 ms
Size: 786 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"args": {
"id": ""
},
"data": {},
"files": {},
"form": {},
"headers": {
"x-forwarded-proto": "http",
"x-forwarded-port": "80",
"host": "postman-echo.com",
"x-amzn-trace-id": "Root=1-5f44da89-a5e24f204c1508a0c2a85dc0",
"auth_key": "st brides",
"user-agent": "PostmanRuntime/7.26.3",
Find and Replace
Console
Bootcamp
Build
Browse

CODER HOUSE

Alternativas

Existen varias alternativas a postman, incluso algunas incluyen extensiones para el VSCode, como es el caso de: Thunder Client (https://www.thunderclient.io/), el cual pueden descargar desde el VSCode mismo, y utilizar de manera muy similar a Postman. Sus funcionalidades son algo más reducidas, pero para operaciones sencillas es más que suficiente (y probablemente lo sea para todo lo que haremos en la clase de hoy y para la mayoría de lo que haremos en futuras ocasiones).



Operaciones con el servidor

Tiempo: 5 minutos

CODER HOUSE

1) Desarrollar un servidor que permita realizar la suma entre dos números utilizando tres rutas en estos formatos (Ejemplo con números 5 y 6)

   a) Ruta get '/api/sumar/5/6

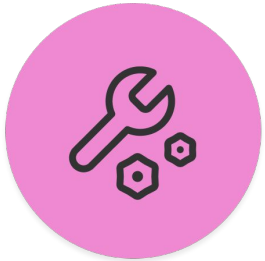
   b) Ruta get '/api/sumar?num1=5&num2=62)

   c) Ruta get '/api/operacion/5+6

   No hace falta validar los datos a sumar, asumimos que los ingresamos correctamente.

2) Implementar las rutas post, put y delete en la dirección '/api' respondiendo 'ok' + (post/put/delete) según corresponda. Probar estas rutas con Postman, verificando que el servidor responda con el mensaje correcto.

El servidor escuchará en el puerto 8080 y mostrará todos los mensajes de conexión/error que correspondan.



Servidor con get, post, put y delete

Tiempo: 10/15 minutos

CODER HOUSE

Considere la siguiente frase: ‘Frase inicial’

Realizar una aplicación de servidor node.js con express que incorpore las siguientes rutas:

1) GET '/api/frase': devuelve un objeto que como campo ‘frase’ contenga la frase completa
2) GET '/api/palabras/:pos': devuelve un objeto que como campo ‘buscada’ contenga la palabra hallada en la frase en la posición dada (considerar que la primera palabra es la #1.
3) POST '/api/palabras': recibe un objeto con una palabra bajo el campo ‘palabra’ y la agrega al final de la frase. Devuelve un objeto que como campo ‘agregada’ contenga la palabra agregada, y en el campo ‘pos’ la posición en que se agregó dicha palabra.
4) PUT '/api/palabras/:pos': recibe un objeto con una palabra bajo el campo ‘palabra’ y reemplaza en la frase aquella hallada en la posición dada. Devuelve un objeto que como campo ‘actualizada’ contenga la nueva palabra, y en el campo ‘anterior’ la anterior.

5) DELETE '/api/palabras/:pos': elimina una palabra en la frase, según la posición dada (considerar que la primera palabra tiene posición #1).

Aclaraciones:

- Utilizar Postman para probar la funcionalidad.
- El servidor escuchará peticiones en el puerto 8080 y mostrará en la consola un mensaje de conexión que muestre dicho puerto, junto a los mensajes de error si ocurriesen.

¿PREGUNTAS? 

¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

-Aplicaciones Restful

-Manejo de peticiones HTTP con Express

-Postman

OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDOLAEDUCACIÓN

CODER HOUSE