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

Repasando...

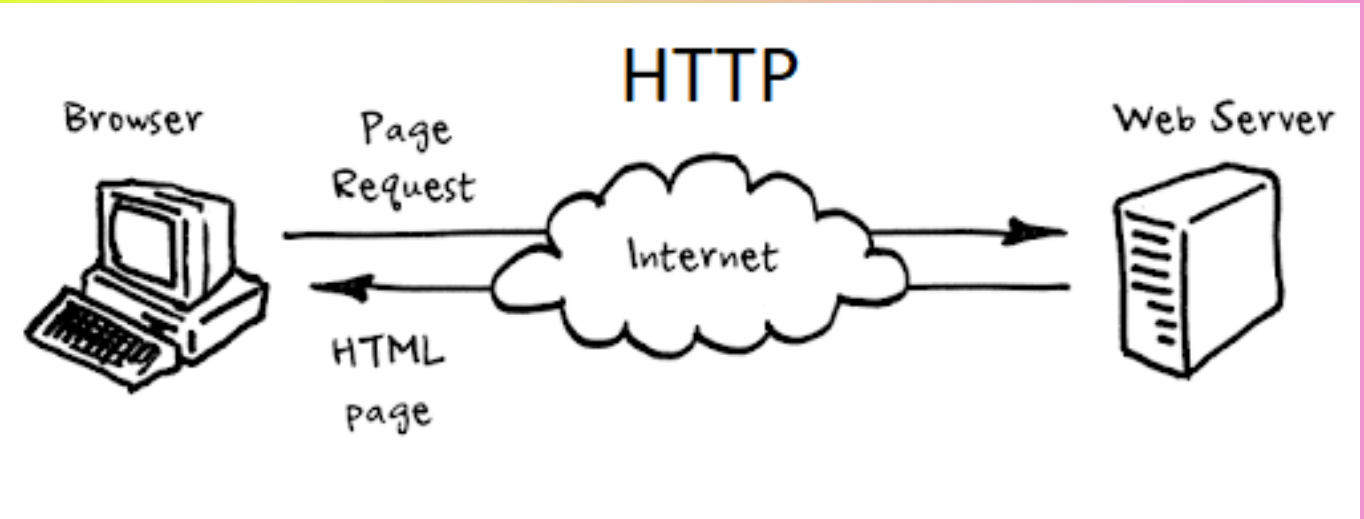
CODER HOUSE

- **Recordemos** que en el **desarrollo web** existen dos participantes básicos, el **cliente** y el **servidor**.
- El **cliente**, en su definición más básica es la **aplicación que recibe las órdenes del usuario**. Normalmente está cargado de algoritmos para la **presentación de datos** y **validación de entradas**.
- El **servidor**, también en su definición más básica es una **aplicación que procesa datos de entrada y regresa una salida**. Esta interacción la conocemos como esquema **solicitud** y **respuesta**.

Servidores Web

¿Alguna pregunta hasta ahora?

Protocolo HTTP

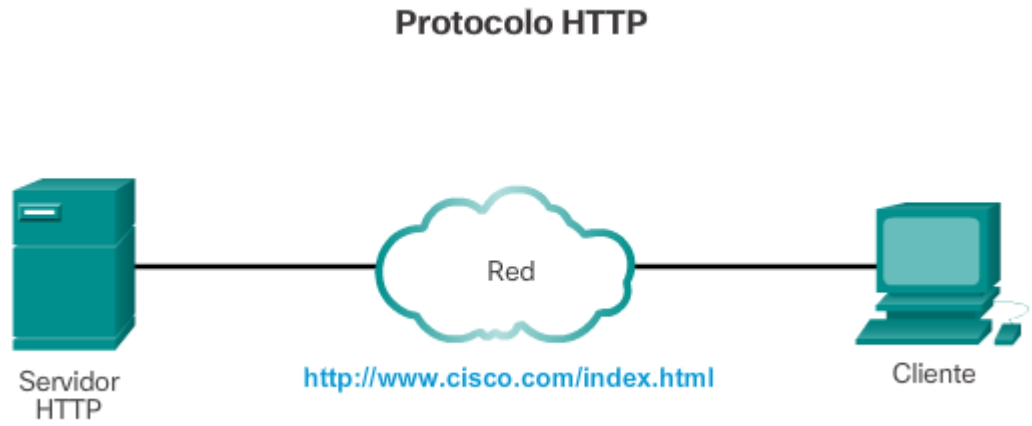


CODER HOUSE

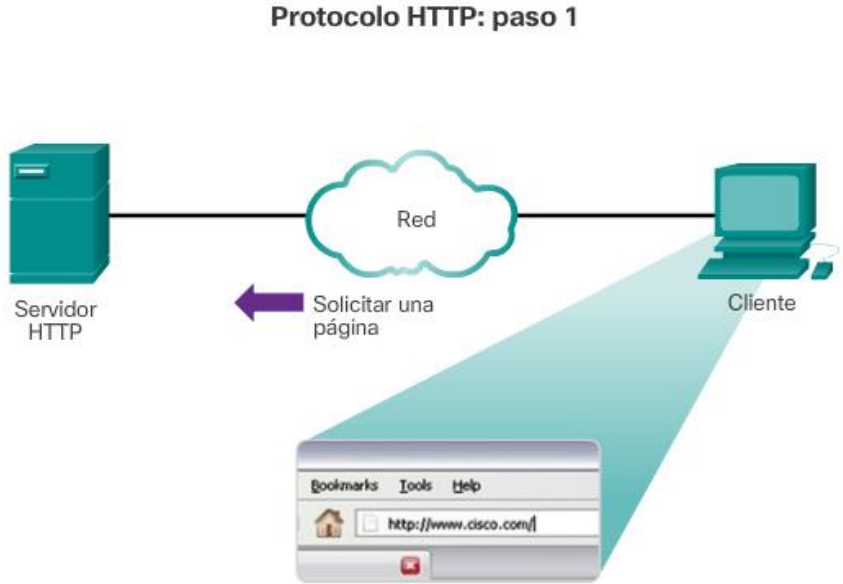
El protocolo HTTP

- **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 las peticiones.
- HTTP establece varios tipos de peticiones, siendo las principales: *POST, GET, PUT, y DELETE*.

Etapas de comunicación HTTP



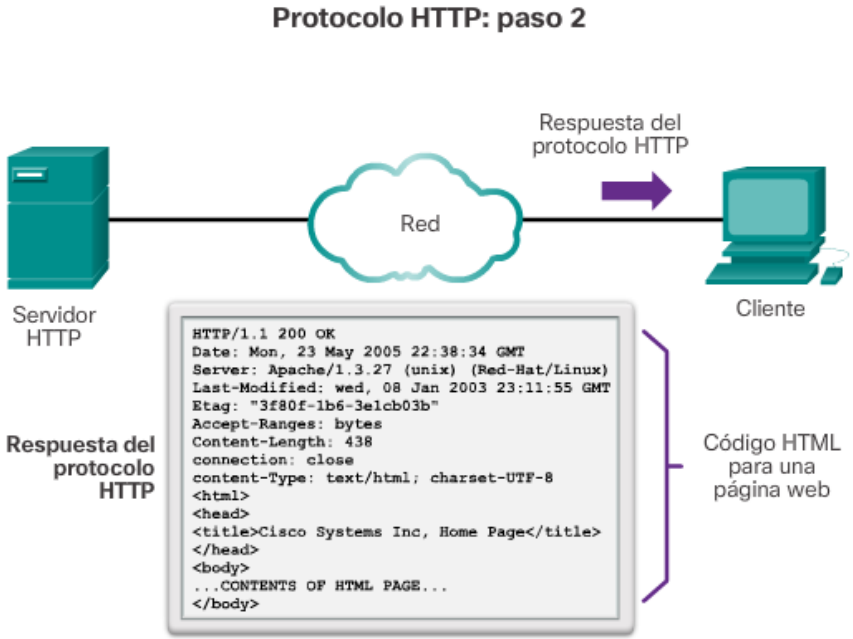
Etapas de comunicación HTTP



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

CODER HOUSE

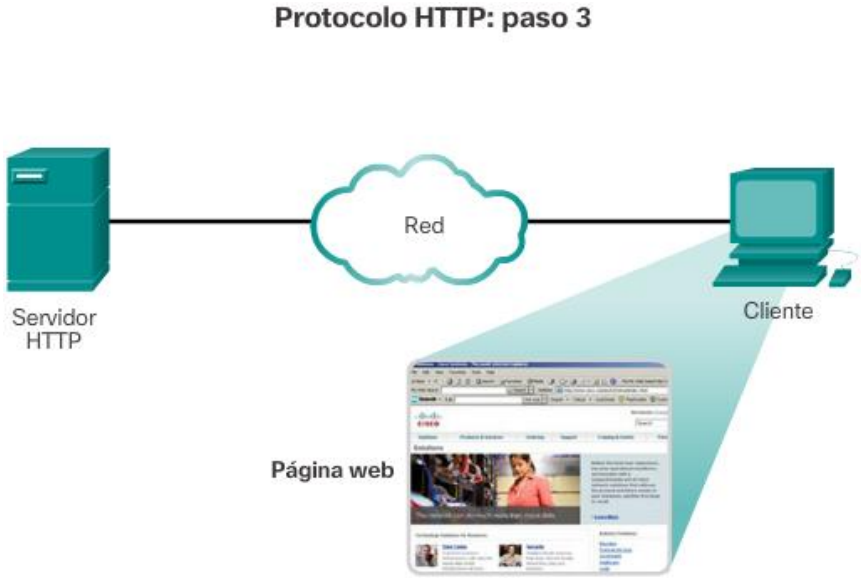
Etapas de comunicación HTTP



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

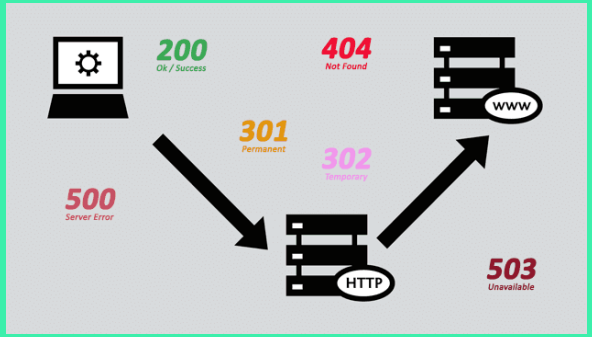
Cada mensaje de respuesta de HTTP tiene un **código** de estado numérico de tres cifras que indica el **resultado** de la petición

- ***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

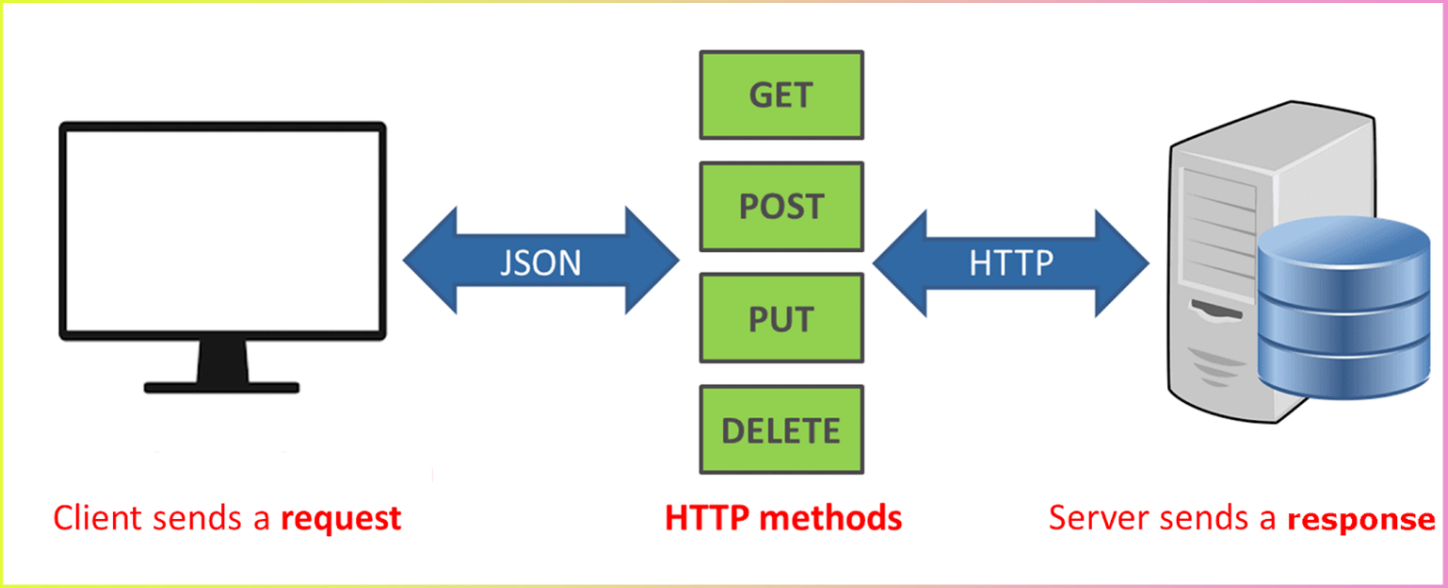
- **1xx**: Mensaje informativo.
- **2xx:** Exito
  - 200 OK
  - 201 Created
  - 202 Accepted
  - 204 No Content
- **3xx:** Redirección
  - 300 Multiple Choice
  - 301 Moved Permanently
  - 302 Found
  - 304 Not Modified
- **4xx:** Error del cliente
  - 400 Bad Request
  - 401 Unauthorized
  - 403 Forbidden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



CODER HOUSE

¿Alguna pregunta hasta ahora?

API & REST



CODER HOUSE

- **Interfaz de programación de aplicaciones**: es un **programa** que encapsula la complejidad de un sistema y sus recursos en una **interfaz simple** para el uso de otros sistemas.
- Una API define las **reglas** que se deben seguir para **comunicarse** con otros sistemas de software.
- Las API **permiten** que sus productos y servicios se comuniquen con otros, **sin necesidad de saber** cómo están **implementados**.

API

Ejemplos de API

CODER HOUSE

- **Transferencia de Estado Representacional**: Es una **arquitectura** de software que **impone condiciones** y recomendaciones sobre cómo debe funcionar una **API**.
- Estas condiciones van desde los **datos enviados en la solicitud**, cómo los **datos** que se entregarán en la **respuesta**.
- Las API que siguen el **estilo arquitectónico** de REST se llaman **API REST**.
- Los servicios web que implementan una arquitectura de REST son llamados **servicios web RESTful**.

REST

- **Interfaz uniforme**: Transferencia de información en un formato estándar. Sin embargo, esta información puede ser en un formato diferente en la representación interna.
- **Tecnología sin estado**: Debe establecer un método de comunicación que complete las solicitudes del cliente independientemente de las solicitudes anteriores.
- **Sistema por capas**: El procesamiento de una solicitud puede llevarse a cabo en diferentes servidores con múltiples capas. Sin embargo, esto debe ser invisible al cliente.
- **Almacenamiento en caché**: Soportar el almacenamiento de respuestas en memoria caché para mejorar el tiempo de respuesta.
- **Código bajo demanda**: Los servidores pueden extender o personalizar temporalmente la funcionalidad del cliente transfiriendo a este el código de programación del software. Este principio fue mejorado con el concepto de hipermedios.

Principios REST

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"
  ]
}
```

¿Alguna pregunta hasta ahora?

BREAK

¡5/10 MINUTOS Y VOLVEMOS!

API RESTful

CODER HOUSE

API RESTful = API que implementa la arquitectura REST 😎

- La **API RESTful** es una **interfaz** que dos sistemas de computación utilizan para intercambiar información de manera segura a través de Internet.
- La mayoría de las aplicaciones para empresas deben comunicarse con otras aplicaciones internas o de terceros para llevar a cabo varias tareas.
- **Por ejemplo**, para generar nóminas mensuales, su sistema interno de cuentas debe compartir datos con el sistema bancario de su cliente para automatizar la facturación y comunicarse con una aplicación interna de planillas de horarios.
- Las API RESTful admiten este **intercambio de información** porque siguen **estándares de comunicación** de software seguros, confiables y eficientes.

API RESTful

{RESTful API}

- En **REST** 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**.
- Estas funcionalidades o acciones se **soportan** sobre los **verbos HTTP**: **GET, POST, PUT, DELETE**.
- También las respuestas se **consolidan** tomando en cuenta los **códigos de estatus HTTP** y su relación con la respuesta: **200, 201, 400, 404, 422, 500, 508, entre otros**.

Ejemplo

Recurso	POST	GET	PUT	DELETE
/usuario	Crea un nuevo usuario *(status code 201)*	Obtener todos los usuarios *(status code 200)*	Actualización de todos los usuarios (No recomendado ⃠) *(status code 204)*	Borrar todos los usuarios (No recomendado ⃠) *(status code 204)*
/usuario/001	No permitido ⃠	Obtener al usuario con id 001 *(status code 200)*	Actualizar al usuario con id 001 *(status code 204)*	Borrar al usuario con id 001 *(status code 204)*

Create Read Update Delete

CODER HOUSE

Los recursos se nombran como Sustantivos 😎

/usuarios  , /productos  ,
/proveedores  , entre otros

Verbos no por favor 🙏

/crear_usuario ⃠, /actualizar_producto ⃠, /obtener_proveedores ⃠, entre otros

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

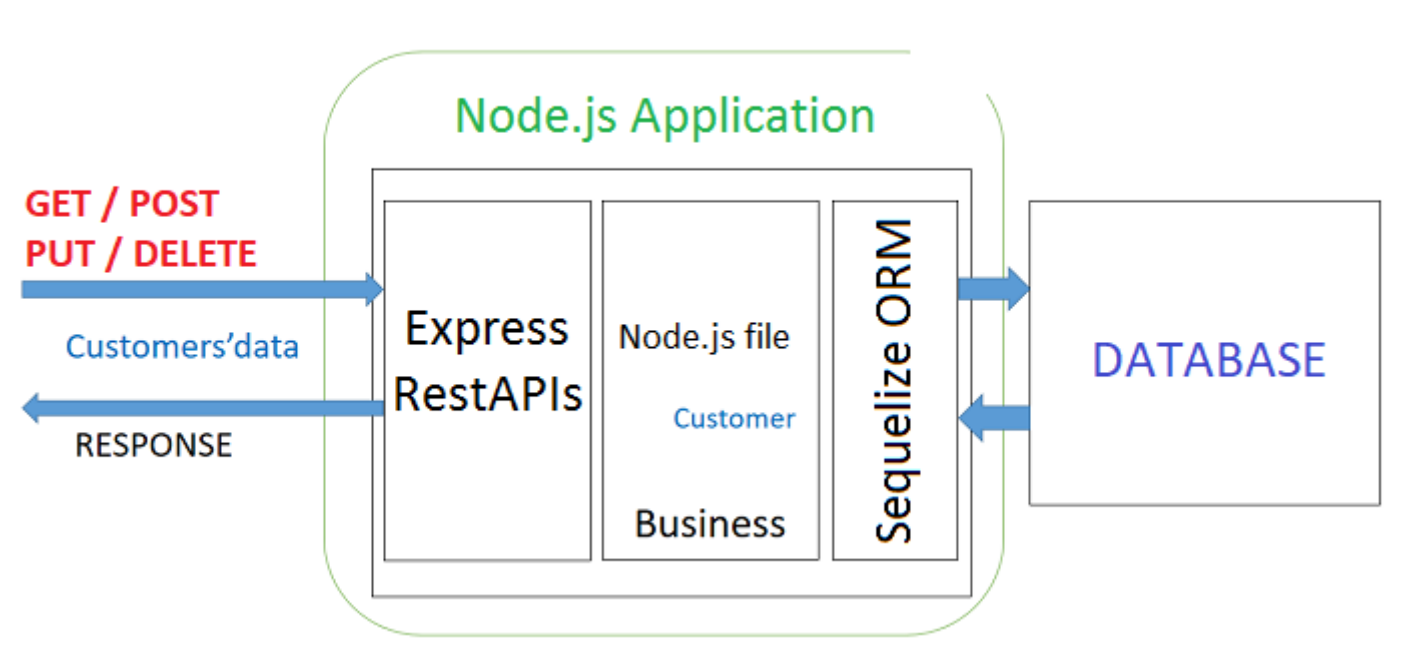
¿Qué es Postman?

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/

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!

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.

1) 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