# Clase 3

# Webservices con express

# Temario

1. Qué es Express
2. Instalación
3. Creación de servidores
4. Middlewares
5. **Enrutamiento**
6. Knex/Sequelize
7. Sesiones, Cookies y JWT

# Enrutamiento o Routing

El **Routing** se refiere a cómo una aplicación responde a la solicitud de un cliente a una ruta en particular (combinación de URI o vía de acceso y método de solicitud HTTP o verbo HTTP)

En esta sección aprenderemos sobre:

- API REST
  - Verbos HTTP
- Definición de rutas

# **API REST:** Definición

## ¿Qué es un API?

Una interfaz de programa de aplicación (API) define las reglas que se deben seguir para comunicarse con otros sistemas de software.

Se puede pensar en una API web como una puerta de enlace entre los clientes y los recursos de la Web.

- **Clientes**: usuarios que desean acceder a la información, pueden ser personas o sistemas de software
- **Recursos**: información que diferentes aplicaciones proporcionan a los clientes (imágenes, vídeos, texto, números o cualquier tipo de datos). La máquina encargada de entregar el recurso al cliente recibe el nombre de **servidor.**

## ¿Qué es REST?

La **transferencia de estado representacional** (REST) es una arquitectura de software que impone condiciones sobre cómo debe funcionar una API. **REST** se creó como una guía para administrar la comunicación en una red compleja como Internet. Una arquitectura basada en REST admite comunicaciones confiables y de alto rendimiento a escala. Se puede implementar y modificar fácilmente, lo que brinda visibilidad y portabilidad entre plataformas a cualquier sistema de API.

# **API REST:** Principios

- **Interfaz uniforme:** El servidor transfiere información en un formato estándar. Impone cuatro limitaciones de arquitectura.
  - Las solicitudes deben identificar los recursos
  - Los clientes tienen información suficiente del recurso para modificarlo o eliminarlo, el servidor cumple esta condición por medio del envío de metadatos
  - Los clientes reciben información sobre cómo seguir procesando los recursos. El servidor envía mensajes autodescriptivos
  - Los clientes reciben información sobre todos los demás recursos relacionados que necesitan para completar una tarea. El servidor logra esto enviando hipervínculos del recurso para que los clientes puedan descubrir dinámicamente más recursos.

- **Tecnología sin estado:** El servidor completa todas las solicitudes del cliente independientemente de todas las solicitudes anteriores. Los clientes pueden solicitar recursos en cualquier orden, y todas las solicitudes son sin estado o están aisladas del resto.

- **Sistema por capas:** el cliente puede conectarse con otros intermediarios autorizados entre el cliente y el servidor y todavía recibirá respuestas del servidor. Los servidores también pueden pasar las solicitudes a otros servidores. Es posible diseñar el servicio web RESTful para que se ejecute en varios servidores con múltiples capas, como la seguridad, la aplicación y la lógica empresarial. Las capas se mantienen invisibles para el cliente.

# **API REST:** Principios

- **Almacenamiento en caché:** Los servicios API REST admiten el almacenamiento en caché, que es el proceso de almacenar algunas respuestas en la memoria caché del cliente o de un intermediario para mejorar el tiempo de respuesta del servidor.

- **Código bajo demanda:** En el estilo de arquitectura de REST, los servidores pueden extender o personalizar temporalmente la funcionalidad del cliente transfiriendo a este el código de programación del software. Por ejemplo para realizar validaciones de formularios.

# **API REST:** Beneficios

## Escalabilidad

Los sistemas que implementan API REST pueden escalar de forma eficiente porque REST optimiza las interacciones entre el cliente y el servidor. La tecnología sin estado elimina la carga del servidor porque este no debe retener la información de solicitudes pasadas del cliente. El almacenamiento en caché bien administrado elimina de forma parcial o total algunas interacciones entre el cliente y el servidor. Todas estas características admiten la escalabilidad, sin provocar cuellos de botella en la comunicación que reduzcan el rendimiento.

## Flexibilidad

Los servicios web RESTful admiten una separación total entre el cliente y el servidor. Simplifican y desacoplan varios componentes del servidor, de manera que cada parte pueda evolucionar de manera independiente. Los cambios de la plataforma o la tecnología en la aplicación del servidor no afectan la aplicación del cliente. La capacidad de ordenar en capas las funciones de la aplicación aumenta la flexibilidad aún más. Por ejemplo, los desarrolladores pueden efectuar cambios en la capa de la base de datos sin tener que volver a escribir la lógica de la aplicación.

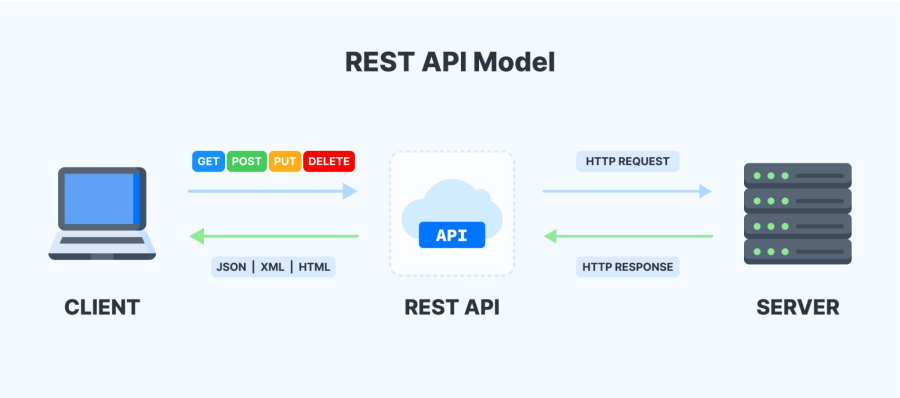
## Independencia

Las API REST son independientes de la tecnología que se utiliza. Puede escribir aplicaciones del lado del cliente y del servidor en diversos lenguajes de programación, sin afectar el diseño de la API. También puede cambiar la tecnología subyacente en cualquiera de los lados sin que se vea afectada la comunicación.

# **API REST:** Funcionamiento

La función básica de una API RESTful es la misma que navegar por Internet. Cuando requiere un recurso, el cliente se pone en contacto con el servidor mediante la API. Los desarrolladores de API explican cómo el cliente debe utilizar la API REST en la documentación de la API de la aplicación del servidor. A continuación, se indican los pasos generales para cualquier llamada a la API REST:

1. **El cliente envía una solicitud al servidor.** El cliente sigue la documentación de la API para dar formato a la solicitud de una manera que el servidor comprenda.
2. El servidor autentica al cliente y confirma que este tiene el derecho de hacer dicha solicitud.
3. El servidor recibe la solicitud y la procesa internamente.
4. Luego, devuelve una respuesta al cliente. Esta respuesta contiene información que dice al cliente si la solicitud **se procesó de manera correcta**. La respuesta también incluye **cualquier información que el cliente haya solicitado.**



# **API REST:** Solicitud o request

Las API REST requieren que las solicitudes contengan los siguientes componentes:

- **Identificador único de recursos:** En los servicios REST, el servidor por lo general identifica los recursos mediante el uso de un localizador uniforme de recursos (URL). La URL especifica la ruta hacia el recurso.

- **Método o verbo HTTP:** También llamados métodos HTTP. Estos verbos, en una petición HTTP, indican la acción que queremos realizar sobre un recurso. Los verbos más utilizados son:
  - **GET**: Obtiene el recurso indicado. Es el método que se utiliza cuando se pide el contenido de una página web, por ejemplo.
  - **HEAD**: Similar a GET, pero no se obtiene el cuerpo de respuesta, únicamente los metadatos de la cabecera.
  - **POST**: Añade datos al servidor. Siempre es un método de creación.
  - **PUT**: Actualiza datos en el servidor. Actualiza todo un recurso.
  - **DELETE**: Elimina el recurso indicado.

- **Encabezados de HTTP:** metadatos que se intercambian entre el cliente y el servidor, por ejemplo el formato de la solicitud y la respuesta, etc.

  - **Datos**: información para que los métodos HTTP funcionen correctamente.
  - **Parámetros**: que brindan más detalles sobre lo que se debe hacer (de ruta, de consulta, de cookie)

# **API REST:** Respuesta o response

Los principios de REST requieren que la respuesta del servidor contenga los siguientes componentes principales:

- **Estado:** Código de estado de tres dígitos que comunica si la solicitud se procesó de manera correcta o dio error. Por ejemplo, los códigos 2XX indican el procesamiento correcto, pero los códigos 4XX y 5XX indican errores. Los códigos 3XX indican la redirección de URL. Los más comunes son:

    - **200**: respuesta genérica de procesamiento correcto
    - **201**: respuesta de procesamiento correcto del método POST
    - **400**: respuesta incorrecta que el servidor no puede procesar
    - **404**: recurso no encontrado
    - **500**: respuesta incorrecta por un fallo en el servidor

    Consulta todos los códigos de estado aquí.

- **Cuerpo del mensaje:** Contiene la representación del recurso. El servidor selecciona un formato de representación adecuado en función de lo que contienen los encabezados de la solicitud. Los clientes pueden solicitar información en los formatos XML o JSON, lo que define cómo se escriben los datos en texto sin formato.

- **Encabezados:** La respuesta también contiene encabezados o metadatos acerca de la respuesta. Estos brindan más contexto sobre la respuesta e incluyen información como el servidor, la codificación, la fecha y el tipo de contenido.

# **Definición de rutas:** Métodos

Cada ruta puede tener una o varias funciones manejador. La definición de ruta tiene la siguiente estructura:

app.METHOD(PATH, HANDLER)

Dónde:

- **app**: es una instancia de Express o de express.Router
- **METHOD**: es un verbo HTTP en minúscula.
- **PATH**: URI o vía de acceso a la ruta.
- **HANDLER**: o callback, función que se ejecuta al recibir la petición a la ruta.

```
app.get("/book", (req, res) => {
 res.send("Solicitud con método GET" );
});
app.post("/book", (req, res) => {
 res.send("Solicitud con método POST" );
});
app.put("/book/:id", (req, res) => {
 res.send("Solicitud con método PUT" );
});
app.delete("/book/:id", (req, res) => {
 res.send("Solicitud con método DELETE" );
});
```

# **Definición de rutas:** Paths

Los **paths** pueden ser textos, patrones de textos o expresiones regulares.

```
// Path de tipo texto
app.get("/book", (req, res) => {
 res.send("book");
});
// Path de patrones de texto
app.get("/ab?cd", (req, res) => {
 res.send("Acepta las rutas /abcd y /acd" );
});
app.get("/ab+cd", (req, res) => {
 res.send("Acepta las rutas /abcd /abbcd /abbbcd, etc." );
});
app.get("/ef*gd", (req, res) => {
 res.send("Acepta las rutas / efgd /efcgd /efRABDOMgd
/ef123gd, etc.");
});
```

```
// Path con expresiones regulares
app.get(/a/, (req, res) => {
 res.send("Acepta las rutas que contengan una a" );
});
app.get(/.*fly$/, (req, res) => {
 res.send("/butterfly y /dragonfly, pero no con
/butterflyman, /dragonflyman, etc." );
});
```

# **Definición de rutas:** Handlers

Se pueden proporcionar varios **handlers** a una misma ruta, funcionan como middleware para manejar la solicitud.

Las rutas pueden tener varios Handlers, en ese caso, es importante usar el argumento **next** para pasar el control al siguiente Handler.

Existe un método especial **app.all()** para cargar middlewares en todos los verbos HTTP.

```
app.all("/book", (req, res, next) => {
 console.log("Accediendo a la sección book..." );
 next(); // pasar el control al siguiente callback
});
```

Los manejadores de rutas pueden tener la forma de una función, una matriz de funciones o combinaciones de ambas.

```
const cb0 = (req, res, next) => {
 console.log("CB0");
 next();
};

const cb1 = (req, res, next) => {
 console.log("CB1");
 next();
};

app.get("/example/handlers",
 [cb0, cb1],
 (req, res, next) => {
   console.log("La respuesta será enviada por la
siguiente función..." );
   next();
 },
 (req, res) => {
   res.send("Hola desde example/handlers!" );
 }
);
```

# **Definición de rutas:** Handlers

Se pueden crear **handlers** encadenados usando **app.route()**

Útil para la reducción de redundancia y errores tipográficos ya que se define una única ruta.

```
app.route("/example")
  .get((req, res) => {
    res.send("Get a example");
  })
  .post((req, res) => {
    res.send("Post a example");
  })
  .put((req, res) => {
    res.send("Put a example");
  });
```

# **Definición de rutas:** Response methods

Los métodos en el objeto de respuesta (res) de la tabla siguiente pueden enviar una respuesta al cliente y terminar el ciclo de solicitud/respuestas.

Si ninguno de estos métodos se invoca desde un manejador de rutas, la solicitud de cliente se dejará colgada.

Los métodos son:

- res.download(): Solicita un archivo para descargarlo.
- res.end(): Finaliza el proceso de respuesta.
- res.json(): Envía una respuesta JSON.
- res.jsonp(): Envía una respuesta JSON con soporte JSONP.
- res.redirect(): Redirecciona una solicitud.
- res.render(): Renderiza una plantilla o html.
- res.send(): Envía una respuesta de varios tipos.
- res.sendFile(): Envía un archivo como una secuencia de octetos.
- res.sendStatus(): Establece el código de estado de la respuesta y envía su representación de serie como el cuerpo de respuesta.

# Ejercicios

1. Revisar la documentación de los métodos del objeto response, en concreto de: download, json, redirect, render y sendStatus. ¿Qué devuelve cada uno?

   https://expressjs.com/en/4x/api.html#res

2. Revisar las propiedades del objeto request
   https://expressjs.com/en/4x/api.html#req

   ¿Qué devuelven las propiedades hostname, ip, params y route en nuestro ejemplo de /**book/:id**?

# ALBAÑILESDIGITALES

Developing your new career

servicio navarro de empleo

das·nano

VeriDas