INTRODUCCIÓN AL DESARROLLO BACKEND

CON NODEJS

2023

ANDÉN Centro de Innovación y Emprendimientos Tecnológicos

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA UTN - FRC SEU

UTN Facultad Regional Córdoba

Agencia CÓRDOBA JOVEN

CÓRDOBA entre todos

Desarrollando APIs con Express.js

- Introducción a Express.js
- API Restful con Express.js
- Middlewares
- Enrutamiento
- Manejo de errores
- Validación de datos en las solicitudes

Introducción a Express.js

- Express.js es un **framework** de aplicaciones web para Node.js.
- Es ampliamente utilizado en el desarrollo de APIs y aplicaciones web.
- Ofrece una estructura y un conjunto de herramientas para facilitar la creación de servidores web en Node.js.

Ventajas de utilizar Express.js

- **Ligero y minimalista:** Express.js es conocido por su enfoque minimalista y su bajo nivel de abstracción, lo que lo hace fácil de aprender y utilizar.
- **Flexibilidad:** Express.js brinda una gran flexibilidad para configurar y personalizar aplicaciones web según las necesidades del proyecto.
- **Manejo de rutas:** Express.js facilita la definición de rutas y el manejo de solicitudes HTTP.
- Middleware: Express.js cuenta con una amplia gama de middleware que permite agregar funcionalidades adicionales a las aplicaciones de manera sencilla.
- **Gran comunidad y documentación:** Express.js tiene una comunidad activa de desarrolladores y una abundante documentación que facilita el aprendizaje y la resolución de problemas.

Express JS

Configuración inicial de un proyecto de Express.js

- Para comenzar a usar Express.js, es necesario tener Node.js y npm instalados en el sistema.
- Se puede inicializar un proyecto de Express.js utilizando el comando "npm init" en la línea de comandos.
- Una vez creado el proyecto, se instala Express.js mediante el comando "npm install express".

Creación de un servidor básico con Express.js

- Para crear un servidor básico con Express.js, se requiere el siguiente código.
- En este código, se importa el módulo de Express.js, se crea una instancia de la aplicación y se inicia el servidor en el puerto 3000.

```
const express = require('express');
const app = express();

const port = 3000;

app.listen(port, () => {
 console.log(`Servidor Express.js en funcionamiento en el puerto ${port}`);
});
```

Enrutamiento en Express.js

- Express.js utiliza enrutamiento para **definir las acciones que se deben ejecutar según la ruta (URL)** y **el método HTTP de una solicitud**.
- Se define una ruta utilizando el método correspondiente de Express.js (como "get", "post", etc.) y se especifica el controlador que se debe ejecutar para esa ruta.
- Por ejemplo:

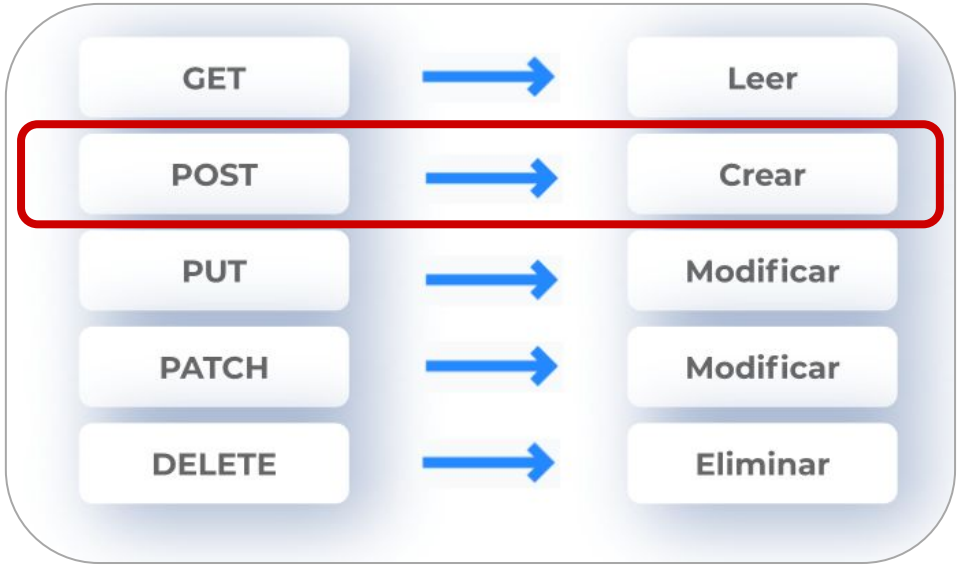
```
app.get('/', (req, res) => {
  res.send('¡Hola, mundo!');
});
```

API Restful: Verbos HTTP en Express.js

- Los verbos HTTP, como GET, POST, PUT, DELETE, etc., se utilizan para indicar la acción que se debe realizar en una solicitud.
- Express.js proporciona métodos correspondientes para manejar los diferentes verbos HTTP.
- Por ejemplo, para manejar una solicitud POST:

```
app.post('/productos', (req, res) => {
  // Lógica para crear un nuevo usuario
  res.send('Producto creado correctamente');
});
```

En este código, se define una ruta para crear un nuevo producto utilizando el verbo POST.



API RestFul: Parámetros Express.js

Los parámetros se utilizan para enviar datos adicionales junto con las solicitudes HTTP. Estos parámetros pueden clasificarse en diferentes tipos, dependiendo de cómo se envíen y se utilicen en las solicitudes:

- **Parámetros de consulta (Query Parameters):**
  - Se especifican en la URL después del signo de interrogación (?).
  - Se utilizan para filtrar, ordenar o paginar los resultados de una solicitud.
  - Ejemplo: https://api.ejemplo.com**/productos**?categoria=electronics&max_precio=100
- **Parámetros de ruta (Path Parameters):**
  - Se incluyen directamente en la URL y forman parte de la ruta.
  - Se utilizan para identificar un recurso específico en una solicitud.
  - Ejemplo: https://api.ejemplo.com**/productos/{id_producto}**
- **Parámetros de cuerpo (Body Parameters):**
  - Se incluyen en el cuerpo de la solicitud HTTP.
  - Se utilizan para enviar datos más complejos, como JSON o XML.
  - Comúnmente utilizados en solicitudes POST, PUT o PATCH.
  - Ejemplo (JSON): { "name": "John", "age": 25 }
- **Parámetros de encabezado (Header Parameters):**
  - Se incluyen en el encabezado de la solicitud HTTP.
  - Proporcionan información adicional sobre la solicitud o el cliente.
  - Ejemplo: Authorization: Bearer <token>

Parámetros de consulta (Query Parameters)

- En este ejemplo, se utiliza un parámetro de consulta (query param) en Express.js para realizar una búsqueda de productos en una categoría específica.
- La ruta "/productos" acepta el parámetro "categoria" en la URL y lo utiliza para filtrar los productos en esta categoría.

```
const express = require('express');
const app = express();

app.get('/productos', (req, res) => {
  const categoria = req.query.categoria;
  // Aquí puedes usar el parámetro de categoría para realizar una búsqueda de productos en esa categoría
  res.send(`Realizar búsqueda de productos en la categoría "${categoria}"`);
});

...
```

Parámetros de ruta (Path Parameters)

- Express.js permite capturar valores dinámicos en las rutas utilizando parámetros.
- Los parámetros se definen con ":" seguido del nombre del parámetro en la ruta.
- Por ejemplo:

```
const express = require('express');
const app = express();

app.get('/productos/:id', (req, res) => {
  const productoId = req.params.id;
  // Lógica para obtener información del usuario con el ID especificado
  res.send(`Información del producto con ID ${productoId}`);
});
```

En este código, se define una ruta con el parámetro ":id" para obtener información de un producto específico.

Parámetros de cuerpo (Body Parameters)

- En este ejemplo, se utiliza un parámetro de cuerpo (body param) en Express.js para guardar un nuevo producto.
- La ruta "/productos" espera una solicitud POST con los datos del producto en el cuerpo de la solicitud.
- Los datos del producto se pueden guardar en una base de datos u realizar otras operaciones relacionadas con el producto.

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/productos', (req, res) => {
  const producto = req.body;
  // Aquí puedes guardar el nuevo producto en la base de datos o realizar otras operaciones relacionadas con el producto
  res.send(`Guardar nuevo producto: ${JSON.stringify(producto)}`);
});
```

Parámetros de encabezado (Header Params)

- En este ejemplo, se utiliza el método req.header() para obtener el valor del header param "Authorization".
- Puedes realizar acciones como autenticar al usuario o aplicar validaciones adicionales basadas en este valor.
- La respuesta simplemente envía el token de autorización recibido en el header.

```
const express = require('express');
const app = express();

app.get('/productos', (req, res) => {
  const authToken = req.header('Authorization');
  // Aquí puedes usar el authToken para autenticar al usuario o realizar validaciones adicionales
  res.send(`Token de autorización: ${authToken}`);
});
```

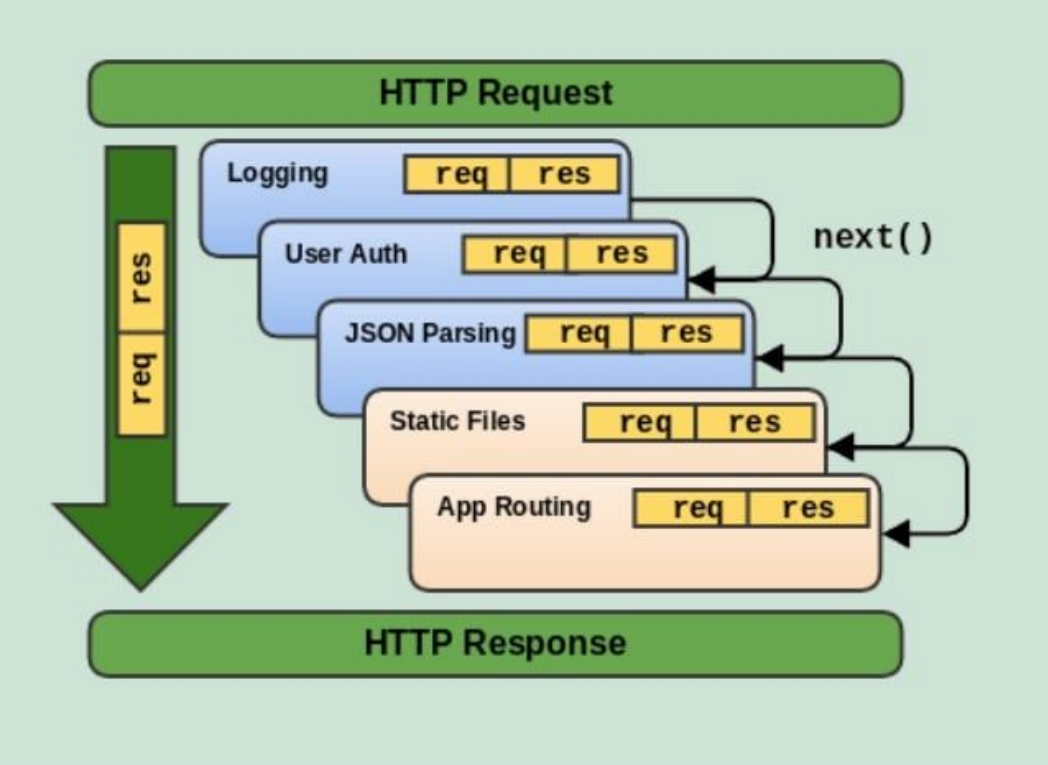
Uso de middleware en Express.js

- Express.js utiliza middleware para procesar solicitudes antes de que lleguen a las rutas definidas.
- El middleware puede realizar tareas como la validación de datos, el registro de solicitudes, la autenticación, etc.
- Se puede utilizar middleware incorporado o crear middleware personalizado.
- Por ejemplo:

```
const express = require('express');
const logger = require('logger');
// Middleware para analizar el cuerpo de las solicitudes en formato
JSON
app.use(express.json());
// Middleware personalizado para registrar las solicitudes
app.use(logger);
```

En este código, se utilizan middleware incorporado y personalizado.

Uso de middleware en Express.js




```
app.get('/productos/:id', (req, res, next) => {
  const productoId = req.params.id;

  Producto.findById(productoId, (err, producto) => {
    if (err) {
      // Error al consultar la base de datos
      return next(err);
    }

    if (!producto) {
      // Producto no encontrado
      return next({ status: 404, message: 'Producto no encontrado'});
    }

    // Producto encontrado, devolver respuesta exitosa
    res.json(producto);
  });
});
```

Organizando el Enrutamiento

- Express.js proporciona un **enrutador incorporado** para manejar el enrutamiento de manera modular y organizada.
- **El enrutador se puede utilizar para agrupar rutas relacionadas y organizar la lógica de manejo de solicitudes.**
- Ejemplo de código:

```
const express = require('express');
const routerProductos = express.Router();

routerProductos.post('/', (req, res) => {
  res.send('Crear Producto');
});

routerProductos.get('/', (req, res) => {
  res.send('Consultar productos');
});

app.use('/productos', routerProductos);
```

- En este código, se crea un enrutador utilizando express.Router() y se definen rutas dentro de él.
- El enrutador se monta en la aplicación principal utilizando app.use() .

API RestFul: Respuestas con JSON

- Express.js facilita el envío de respuestas en formato JSON a través del método res.json().
- Este método convierte automáticamente un objeto JavaScript en formato JSON y lo envía como respuesta al cliente.
- Ejemplo de código:

```
const express = require('express');
const app = express();

app.get('/productos', (req, res) => {
  const productos = [
    { id: 1, nombre: 'Tablet'},
    { id: 2, nombre: 'PC'}
  ];
  res.json(productos);
});
```

- En este código, se define una ruta "/productos" que devuelve un arreglo de objetos de productos en formato JSON.
- Al llamar a res.json(usuarios), Express.js automáticamente establece los encabezados adecuados y convierte el objeto productos en formato JSON antes de enviarlo al cliente.

API RestFul: Manejo de Errores

Descripción de los posibles errores que pueden ocurrir en una API Restful:

- Errores de sintaxis y validación de datos (400 Bad Request, 404 Not Found).
- Errores de autenticación y autorización (401 Unauthorized, 403 Forbidden).
- Errores de consulta a la base de datos (503 Service Unavailable).
- Errores internos del servidor (500 Internal Server Error).


```
{
    "error":{
        "mensaje":"Error General",
        "tipo":"general",
        "codigo":"1234",

    }
}
```

Más info sobre Status Codes: https://www.restapitutorial.com/httpstatuscodes.html

Middleware Manejo de Errores

- En el archivo app.js agregamos un Middleware para el manejo de errores con el siguiente código:

```
app.use((err, req, res, next) => {
  // Manejo de errores
  res.status(err.status || 500);
  res.json({
    error: err.message
  });
});
```

- El middleware de manejo de errores se define con una función que toma cuatro parámetros: err, req, res y next.
- **err** es el objeto que contiene la información del error.
- **req** es el objeto de solicitud.
- **res** es el objeto de respuesta.
- **next** es una función que se utiliza para pasar al siguiente middleware.

Middleware Manejo de Errores

```
app.get('/productos/:id', (req, res, next) => {
  const productoId = req.params.id;

  Producto.findById(productoId, (err, producto) => {
    if (err) {
      // Error al consultar la base de datos
      return next(err);
    }

    if (!producto) {
      // Producto no encontrado
      return next({ status: 404, message: 'Producto no encontrado'});
    }

    // Producto encontrado, devolver respuesta exitosa
    res.json(producto);
  });
});
```

Validación de datos

- Problemas que pueden surgir al no validar los datos en las solicitudes:
  - Crear objetos con datos incorrectos o no válidos en la base de datos.
  - Ejecución de operaciones no deseadas debido a parámetros incorrectos.
  - Vulnerabilidades de seguridad, como inyección de código malicioso.

Validación **manual** de datos

```
app.post('/productos, (req, res) => {
  const { nombre, precio, descripcion } = req.body;

  if (!nombre || !precio || !descripcion) {
    return res.status(400).json({ error: 'Faltan campos requeridos' });
  }

  if (precio <= 0) {
    return res.status(400).json({ error: 'El precio no es válido' });
  }

  // Resto de la lógica de producto
});
```

Validación de datos con Joi

```
const Joi = require('joi');
// Definir un esquema de validación con Joi
const schema = Joi.object({
  nombre: Joi.string().required(),
  edad: Joi.number().min(18).max(99).required(),
  email: Joi.string().email().required(),
  suscrito: Joi.boolean().default(false)
});

// Validar los datos
const datos = {
  nombre: 'Juan',
  edad: 25,
  email: 'juan@example.com'
};

const { error, value } = schema.validate(datos);

if (error) {
  console.log('Error de validación:', error.details[0].message);
} else {
  console.log('Datos validados:', value);
}
```

Antes de usarlo debemos instalarlo con npm con el siguiente comando:
> npm install joi

Joi en una API

```
const Joi = require('joi');

// Definir esquema de validación con Joi
const productoSchema = Joi.object({
  nombre: Joi.string().required(),
  precio: Joi.number().min(0).required(),
  descripcion: Joi.string().optional()
});

// Ruta para crear un nuevo producto
app.post('/productos', (req, res) => {
  const { error, value } = productoSchema.validate(req.body);

  if (error) {
    // Si la validación falla, se envía una respuesta de error con los detalles
    return res.status(400).json({ error: error.details[0].message });
  }

  // Si la validación es exitosa, se puede utilizar el objeto "value" que contiene los datos validados
  // Aquí puedes guardar los datos en la base de datos, realizar alguna operación, etc.
  res.status(201).json({ message: 'Producto creado exitosamente' });
});
```

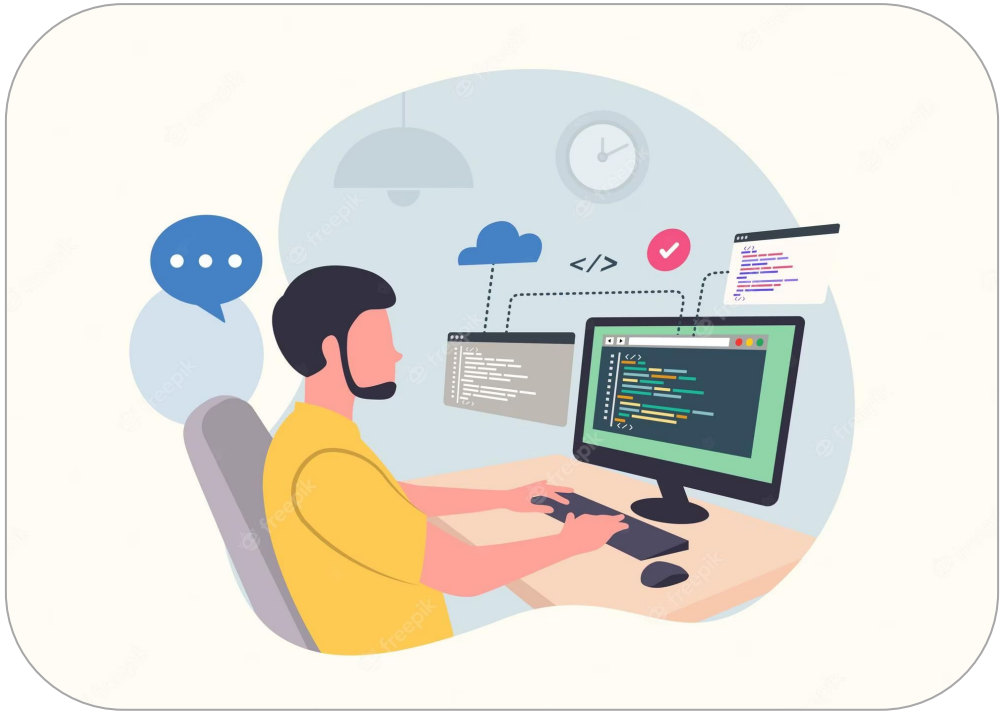
Bueno, Vamo a Codea!!!

1. Ejemplo con Express.js
2. Ejemplo con Router.
3. Middlewares

Express JS

Actividad 4: Paso a Paso API con Express.js

- Seguir las instrucciones de la actividad publicada en la UVE.



MUCHAS GRACIAS

ANDÉN Centro de Innovación y Emprendimientos Tecnológicos

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA UTN - FRC SEU

UTN Facultad Regional Córdoba

Agencia CÓRDOBA JOVEN

CÓRDOBA entre todos