

Documentación de Node.js: Introducción a Express (con ES Modules)

1) ¿Qué es Express? REPASO

Express.js es un framework minimalista y flexible para la creación de aplicaciones web y APIs en **Node.js**. Express es popular en el desarrollo web debido a su simplicidad y a la flexibilidad que ofrece, permitiendo que los desarrolladores integren solo los módulos necesarios para cada aplicación. Esto se diferencia de otros frameworks más grandes y complejos que pueden hacer que la aplicación sea menos flexible y difícil de mantener.

Express permite crear aplicaciones de una sola página, aplicaciones web multipágina y APIs completas de forma rápida y organizada. Su naturaleza minimalista permite un control detallado sobre los componentes y la posibilidad de personalizar la aplicación mediante middleware y módulos adicionales.

2) Características Avanzadas de Express REPASO

Aquí cubrimos dos características avanzadas de Express: el uso de **rutas anidadas y parámetros** y la capacidad de **modularizar el enrutamiento** con `Express.Router`.

1. Soporte para Rutas Anidadas y Parámetros

Express permite definir rutas dinámicas con **parámetros de ruta** que pueden capturar partes específicas de la URL. Esto es útil para crear rutas anidadas o complejas que permiten acceder a recursos específicos (por ejemplo, usuarios y sus posts, o categorías y productos en una tienda en línea).

Ejemplo de Rutas Anidadas y Parámetros

En el siguiente ejemplo, usamos una ruta con parámetros para gestionar posts específicos de un usuario:

```
// Definición de una ruta con parámetros para usuarios y sus posts
app.get('/usuarios/:userId/posts/:postId', (req, res) => {
  const { userId, postId } = req.params;
  res.send(`Mostrando el post ${postId} del usuario ${userId}`);
});
```

Aquí:

- `:userId` y `:postId` son **parámetros de ruta** que capturan valores específicos de la URL.
- `req.params` contiene los valores extraídos de la URL. Por ejemplo, si la URL solicitada es `/usuarios/123/posts/456`, entonces `req.params.userId` sería `"123"` y `req.params.postId` sería `"456"`.

Este enfoque es especialmente útil para manejar estructuras de URLs más complejas, como /categorias/:categoryId/productos/:productId.

2. Enrutamiento Avanzado con Express.Router

Para aplicaciones grandes, el código de enrutamiento puede volverse complejo si todas las rutas están en un solo archivo. Express ofrece **Express.Router**, una herramienta que permite crear “mini aplicaciones” de rutas que se pueden importar y usar en la aplicación principal. Esto ayuda a mantener el código modular, organizado y fácil de mantener.

Express.Router permite:

- Dividir las rutas en archivos separados, organizando el código en módulos.
- Asignar grupos de rutas a una URL base específica en el servidor principal.

Ejemplo de Enrutamiento Modular con Express.Router

1. Definir las Rutas en un Módulo

Creamos un archivo `blog.mjs` en la carpeta `routes` que contiene las rutas para los blogs:

```
// routes/blog.mjs
import express from 'express';

const router = express.Router();

// Definición de las rutas del blog
router.get('/', (req, res) => res.send('Lista de blogs'));
router.get('/:id', (req, res) => res.send(`Blog con ID: ${req.params.id}`));
router.post('/', (req, res) => res.send('Nuevo blog creado'));

export default router;
```

2. Importar el Módulo de Rutas en la Aplicación Principal

En el archivo principal `server.mjs`, importamos el módulo de rutas y lo usamos en nuestra aplicación Express con una URL base `/blogs`:

```
// server.mjs
import express from 'express';
import blogRouter from './routes/blog.js'; // Importamos el módulo de rutas

const app = express();

app.use('/blogs', blogRouter); // Asignamos las rutas al prefijo '/blogs'

const PORT = 3000;
app.listen(PORT, () => console.log(`Servidor corriendo en el puerto ${PORT}`));
```

En este caso:

- La URL base `/blogs` hace que todas las rutas definidas en `blog.js` respondan con ese prefijo.
- `GET /blogs` responderá con "Lista de blogs".
- `GET /blogs/:id` responderá con "Blog con ID: <id>" .
- `POST /blogs` permitirá crear un nuevo blog.

Este enfoque modular permite agrupar rutas relacionadas en archivos separados, manteniendo el código claro y organizado.

3) Tipos de Peticiones HTTP en Express REPASO

En esta sección, aprenderemos a configurar cada tipo de petición HTTP (`GET`, `POST`, `PUT`, y `DELETE`) utilizando **enrutado avanzado** con `Express.Router`. Este enfoque modular facilita la organización del código, especialmente en aplicaciones grandes.

3.1 Configuración del Módulo de Rutas

Para estructurar las rutas en módulos, crearemos un archivo de rutas `usuarios.js` en una carpeta `routes`, donde manejaremos las diferentes peticiones para la ruta `/usuarios`.

1. Crear el Archivo de Rutas `usuarios.js`:

```
// routes/usuarios.js
import express from 'express';
const router = express.Router();

// Obtener todos los usuarios
router.get('/', (req, res) => {
    res.send('Aquí está la lista de todos los usuarios');
});

// Obtener un usuario específico por ID
router.get('/:id', (req, res) => {
    const { id } = req.params;
    res.send(`Aquí está la información del usuario con ID: ${id}`);
});

// Crear un nuevo usuario
router.post('/', (req, res) => {
    const nuevoUsuario = req.body;
    res.send(
        `Nuevo usuario creado con los datos: ${JSON.stringify(nuevoUsuario)}`
    );
});

// Actualizar un usuario por ID
router.put('/:id', (req, res) => {
```

```

const { id } = req.params;
const datosActualizados = req.body;
res.send(
  `Usuario con ID: ${id} ha sido actualizado con los datos:
  ${JSON.stringify(datosActualizados)}`
);
});

// Eliminar un usuario por ID
router.delete('/:id', (req, res) => {
  const { id } = req.params;
  res.send(`Usuario con ID: ${id} ha sido eliminado`);
});

export default router;

```

2. Integrar el Módulo de Rutas en el Servidor Principal:

En el archivo principal `server.js`, importamos y utilizamos el módulo de rutas para asignarle el prefijo `/usuarios`.

```

// server.js
import express from 'express';
import usuariosRouter from './routes/usuarios.js';

const app = express();

// Middleware para manejar JSON en el cuerpo de las solicitudes
app.use(express.json());

// Asignamos el módulo de rutas al prefijo '/usuarios'
app.use('/usuarios', usuariosRouter);

const PORT = 3000;
app.listen(PORT, () =>
  console.log(`Servidor corriendo en el puerto ${PORT}`)
);

```

3.2 Tipos de Peticiones HTTP en el Módulo `usuarios.js`

A continuación, explicamos cada tipo de petición HTTP implementado en el módulo de rutas `usuarios.js`:

1. Peticiones GET

- **Obtener todos los usuarios:** Devuelve una lista de todos los usuarios.
- **Obtener un usuario específico:** Utiliza un parámetro `id` para recuperar información de un usuario específico.

```
// Obtener todos los usuarios
router.get('/', (req, res) => {
  res.send('Aquí está la lista de todos los usuarios');
});

// Obtener un usuario específico por ID
router.get('/:id', (req, res) => {
  const { id } = req.params;
  res.send(`Aquí está la información del usuario con ID: ${id}`);
});
```

2. Peticiones POST

- **Crear un nuevo usuario:** Recibe los datos del nuevo usuario en el cuerpo de la solicitud y simula la creación de un nuevo usuario.

```
// Crear un nuevo usuario
router.post('/', (req, res) => {
  const nuevoUsuario = req.body;
  res.send(
    `Nuevo usuario creado con los datos: ${JSON.stringify(nuevoUsuario)}`
  );
});
```

3. Peticiones PUT

- **Actualizar un usuario existente:** Utiliza el parámetro `id` para identificar al usuario y los datos en el cuerpo de la solicitud para actualizar su información.

```
// Actualizar un usuario por ID
router.put('/:id', (req, res) => {
  const { id } = req.params;
  const datosActualizados = req.body;
  res.send(
    `Usuario con ID: ${id} ha sido actualizado con los datos:
      ${JSON.stringify(datosActualizados)}`
  );
});
```

4. Peticiones DELETE

- **Eliminar un usuario:** Utiliza el parámetro `id` para identificar al usuario a eliminar y simula su eliminación.

```
// Eliminar un usuario por ID
router.delete('/:id', (req, res) => {
  const { id } = req.params;
  res.send(`Usuario con ID: ${id} ha sido eliminado`);
});
```

Resumen de los Métodos HTTP en Express

Método	Acción principal	Ejemplo en una tienda
POST	Crear un nuevo recurso	Registrar un nuevo usuario
DELETE	Eliminar un recurso existente	Borrar un producto
PUT	Actualizar (o crear) un recurso	Cambiar el precio de un producto
GET	Solicitar/ver información del servidor	Ver lista de productos

Este resumen presenta los métodos HTTP principales utilizados en Express, con una acción típica asociada y ejemplos en el contexto de una tienda.

Módulo 4: Introducción Teórica a Middleware en Express para Node.js

En el desarrollo de aplicaciones con Express, el middleware es fundamental para controlar el flujo de solicitudes y respuestas. Los middleware son funciones que operan entre la solicitud del cliente y la respuesta del servidor, permitiendo realizar operaciones como validación de datos, autenticación, registro de solicitudes y manipulación de respuestas. En este módulo, veremos cómo configurar middleware personalizado, aplicar middleware a rutas específicas con `Router`, y usaremos el motor de plantillas EJS para renderizar vistas.

¿Qué es un Middleware?

En Express, un middleware es una función que se ejecuta entre la solicitud y la respuesta, permitiendo realizar operaciones intermedias antes de finalizar el proceso. Un middleware recibe tres parámetros: `req` (la solicitud), `res` (la respuesta) y `next` (la función para pasar el control al siguiente middleware). Al invocar `next()`, la solicitud pasa al siguiente middleware en la pila, hasta llegar al manejador de ruta final.

Ejemplo Básico de Middleware

Comencemos con un middleware simple que registra en la consola cada solicitud recibida. Este middleware puede aplicarse globalmente para monitorear y depurar la actividad de la aplicación.

```
import express from 'express';

const app = express();

// Middleware global para registrar solicitudes
const loggerMiddleware = (req, res, next) => {
  console.log(`Request received: ${req.method} ${req.url}`);
  next(); // pasa el control al siguiente middleware o ruta
};
```

```

app.use(loggerMiddleware);

app.get('/', (req, res) => {
  res.send('Hello World');
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});

```

Explicación línea a línea:

1. import express from 'express'; — Importamos Express usando `import`.
2. const app = express(); — Creamos una instancia de la aplicación Express.
3. const loggerMiddleware = (req, res, next) => {...}; — Definimos el middleware `loggerMiddleware`.
4. console.log(...) — Imprime en la consola el método y la URL de cada solicitud recibida.
5. next(); — Permite que la solicitud continúe hacia el siguiente middleware o ruta.
6. app.use(loggerMiddleware); — Registra el middleware globalmente en la aplicación.
7. app.get('/', (req, res) => {...}); — Define una ruta simple que responde con "Hello World".
8. app.listen(3000, () => {...}); — Inicia el servidor en el puerto 3000.

Middleware Específico para Rutas con Router

Para aplicaciones de gran escala, Express permite organizar rutas y middleware en módulos utilizando `Router`. Esto facilita la organización del código y la aplicación de middleware específico a conjuntos de rutas.

Justificación: La estructura modular ayuda a organizar mejor las rutas y a aplicar middleware únicamente donde es necesario, mejorando la mantenibilidad.

Archivo `userRoutes.js`:

```

import { Router } from 'express';

const router = Router();

// Middleware de autenticación para usuarios
const userAuthMiddleware = (req, res, next) => {
  console.log('User Authentication Middleware');
  if (!req.headers.authorization) {
    return res.status(401).send('Unauthorized');
  }
  next();
};

// Aplicamos el middleware de autenticación a todas las rutas de usuario
router.use(userAuthMiddleware);

// Rutas de usuario

```

```

router.get('/', (req, res) => {
  res.send('User List');
});

router.get('/:id', (req, res) => {
  res.send(`User Profile: ${req.params.id}`);
});

export default router;

```

Archivo app.js:

```

import express from 'express';
import userRoutes from './userRoutes.js';

const app = express();

// Usamos el router para las rutas de usuario
app.use('/users', userRoutes);

app.listen(3000, () => {
  console.log('Server running on port 3000');
});

```

Explicación línea a línea de userRoutes.js:

1. import { Router } from 'express'; – Importamos Router desde Express.
2. const router = Router(); – Creamos una instancia de Router.
3. const userAuthMiddleware = (req, res, next) => {...}; – Definimos un middleware de autenticación de usuario.
4. if (!req.headers.authorization) {...} – Verifica el encabezado de autorización.
5. next(); – Si la autenticación es correcta, permite que la solicitud continúe.
6. router.use(userAuthMiddleware); – Aplica el middleware de autenticación a todas las rutas del Router.
7. router.get('/', (req, res) => {...}); – Define la ruta principal de usuarios.
8. export default router; – Exporta el Router para usarlo en app.js.

Middleware Comunes en Express

Express se beneficia de middleware populares que manejan funcionalidades comunes como la seguridad, el registro de solicitudes y el análisis de datos JSON. Aquí algunos ejemplos:

Official Express Middleware Modules

Middleware Module	Description	Replaces Built-in Function (Express 3)
-------------------	-------------	--

Middleware Module	Description	Replaces Built-in Function (Express 3)
body-parser	Parse HTTP request body. See also: <code>body</code> , <code>co-body</code> , <code>raw-body</code> .	<code>express.bodyParser</code>
compression	Compress HTTP responses.	<code>express.compress</code>
connect-rid	Generate unique request ID.	NA
cookie-parser	Parse cookie header and populate <code>req.cookies</code> . See also <code>cookies</code> and <code>keygrip</code> .	<code>express.cookieParser</code>
cookie-session	Establish cookie-based sessions.	<code>express.cookieSession</code>
cors	Enable cross-origin resource sharing (CORS) with various options.	NA
errorhandler	Development error-handling/debugging.	<code>express.errorHandler</code>
method-override	Override HTTP methods using header.	<code>express.methodOverride</code>
morgan	HTTP request logger.	<code>express.logger</code>
multer	Handle multi-part form data.	<code>express.bodyParser</code>
response-time	Record HTTP response time.	<code>express.responseTime</code>
serve-favicon	Serve a favicon.	<code>express.favicon</code>
serve-index	Serve directory listing for a given path.	<code>express.directory</code>
serve-static	Serve static files.	<code>express.static</code>
session	Establish server-based sessions (development only).	<code>express.session</code>
timeout	Set a timeout period for HTTP request processing.	<code>express.timeout</code>
vhost	Create virtual domains.	<code>express.vhost</code>

Additional Popular Middleware Modules

Estos modulos son tambien muy populares aunque no son mantenidos por Express

Middleware Module	Description
-------------------	-------------

Middleware Module	Description
cls-rtracer	Middleware for CLS-based request id generation. Adds request ids into your logs.
connect-image-optimus	Optimize image serving by switching images to <code>.webp</code> or <code>.jxr</code> if possible.
error-handler-json	An error handler for JSON APIs (fork of <code>api-error-handler</code>).
express-debug	Development tool that adds info about template variables (<code>locals</code>), session, etc.
express-partial-response	Filters JSON responses based on the <code>fields</code> query-string (similar to Google API's Partial Response).
express-simple-cdn	Use a CDN for static assets, with support for multiple hosts.
express-slash	Handles routes with and without trailing slashes.
express-uncapitalize	Redirects HTTP requests containing uppercase to a canonical lowercase form.
helmet	Helps secure your apps by setting various HTTP headers.
join-io	Joins files on the fly to reduce the number of requests.
passport	Authentication using “strategies” such as OAuth, OpenID, and others. More info .
static-expiry	Adds fingerprint URLs or caching headers for static assets.
view-helpers	Provides common helper methods for views.
sriracha-admin	Dynamically generate an admin site for Mongoose.

Módulo 5: Introducción Teórica a Express Validator Middleware en Node.js

`express-validator` es una biblioteca de middleware para Express que facilita la validación y la sanitización de datos en aplicaciones Node.js. Permite validar la entrada de datos de manera rápida y estructurada, garantizando que solo información correcta y en el formato deseado fluya hacia la lógica de negocio. Este módulo proporcionará una base teórica de `express-validator`, una guía detallada sobre su configuración, y ejemplos paso a paso que muestran su uso práctico en combinación con `Router`.

¿Por Qué Usar `express-validator`?

La validación de datos es crucial en aplicaciones web. Permite evitar errores y problemas de seguridad al filtrar datos incorrectos o maliciosos, protegiendo tanto la aplicación como sus usuarios. `express-validator` se integra perfectamente en el flujo de middleware de Express, permitiendo definir reglas de

validación en cada ruta y, si es necesario, sanitizar los datos para adaptarlos a un formato seguro y esperado.

Configuración de `express-validator`

Para comenzar a usar `express-validator`, primero debemos instalar la biblioteca y luego configurarla en el proyecto. Usaremos `Router` para organizar nuestras rutas y middleware.

1. **Instalar `express-validator`**: Primero, ejecuta el siguiente comando en la terminal:

```
npm install express-validator
```

2. **Configuración en el Proyecto**: Importamos `express-validator` y usamos sus funciones para definir validaciones específicas en cada ruta.

Ejemplo de Validación de Datos de Usuario

En este ejemplo, vamos a validar los datos de un formulario de registro de usuario. Usaremos `Router` para organizar las rutas y aplicaremos varias validaciones a campos comunes: `email`, `password`, y `username`.

Justificación Teórica: Al validar los datos en el lado del servidor, podemos asegurarnos de que solo la información correcta llegue a nuestra base de datos. Las validaciones de `express-validator` nos permiten verificar que los datos tengan el formato adecuado y cumplen con nuestras reglas, como una longitud mínima para la contraseña o el formato de correo electrónico.

Archivo `userRoutes.js`:

```
import { Router } from 'express';
import { body, validationResult } from 'express-validator';

const router = Router();

// Ruta de registro de usuario con validaciones
router.post(
  '/register',
  [
    body('email')
      .isEmail()
      .withMessage('Please enter a valid email address'),
    body('password')
      .isLength({ min: 6 })
      .withMessage('Password must be at least 6 characters long'),
    body('username')
      .notEmpty()
      .withMessage('Username is required')
      .isAlphanumeric()
      .withMessage('Username must be alphanumeric')
  ],
  (req, res) => {
```

```

        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array() });
        }
        res.send('User registered successfully');
    }
);

export default router;

```

Explicación Paso a Paso:

1. import { body, validationResult } from 'express-validator'; — Importamos las funciones `body` y `validationResult` de `express-validator` para validar y verificar los resultados.
2. router.post('/register', [...], (req, res) => {...}); — Definimos una ruta `POST` en `/register` para el registro de usuarios.
3. body('email').isEmail()... — Valida que el campo `email` tenga formato de correo electrónico.
4. .withMessage('...') — Agrega un mensaje de error si la validación falla.
5. body('password').isLength({ min: 6 })... — Verifica que la contraseña tenga al menos 6 caracteres.
6. body('username').notEmpty().isAlphanumeric() — Verifica que el `username` no esté vacío y sea alfanumérico.
7. const errors = validationResult(req); — Extrae los errores de validación de la solicitud actual.
8. if (!errors.isEmpty()) {...} — Si hay errores, responde con un código de estado `400` y muestra los errores.
9. res.send('User registered successfully'); — Si no hay errores, responde con un mensaje de éxito.

Este middleware permite validar múltiples campos de manera simple y estructurada, garantizando que los datos cumplan con los requisitos definidos.

Ejemplo Avanzado: Sanitización y Validación Compleja

`express-validator` no solo permite validar datos, sino también sanitizarlos para eliminar caracteres innecesarios o potencialmente dañinos. En este ejemplo, agregaremos una validación y sanitización más compleja para los campos de entrada.

Ejemplo Completo con Validación y Sanitización

Archivo `userRoutes.js`:

```

import { Router } from 'express';
import { body, validationResult } from 'express-validator';

const router = Router();

// Ruta de registro de usuario con validaciones y sanitización
router.post(

```

```

' /register',
[
  body('email')
    .isEmail()
    .withMessage('Please enter a valid email address')
    .normalizeEmail(), // Sanitiza el correo electrónico
  body('password')
    .isLength({ min: 6 })
    .withMessage('Password must be at least 6 characters long')
    .trim()
    .escape(), // Elimina espacios al principio y final y escapa caracteres HTML
  body('username')
    .notEmpty()
    .withMessage('Username is required')
    .isAlphanumeric()
    .withMessage('Username must be alphanumeric')
    .trim()
    .escape() // Sanitiza el nombre de usuario
],
(req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.send('User registered successfully');
}
);

export default router;

```

Explicación Paso a Paso:

1. `body('email').normalizeEmail();` — Aplica sanitización al correo electrónico para que tenga un formato estándar (ej. elimina espacios).
2. `body('password').trim().escape();` — Aplica sanitización a la contraseña, eliminando espacios en blanco y caracteres especiales.
3. `body('username').trim().escape();` — Sanitiza el `username` eliminando espacios y escapando caracteres HTML.

La sanitización añade una capa adicional de seguridad, asegurando que los datos lleguen al servidor en el formato correcto y evitando potenciales inyecciones de código.

Gestión de Errores y Respuesta

En una aplicación, es importante gestionar de manera clara y estructurada los errores de validación para facilitar la corrección de los datos por parte de los usuarios. `express-validator` facilita la extracción y el envío de los errores en formato JSON.

Justificación Teórica: Proporcionar mensajes de error detallados ayuda a los usuarios a entender por qué sus datos no cumplen los requisitos y a corregirlos fácilmente.

Ejemplo de Respuesta de Errores con `express-validator`:

```
import { Router } from 'express';
import { body, validationResult } from 'express-validator';

const router = Router();

router.post(
  '/register',
  [
    body('email').isEmail().withMessage('Invalid email format').normalizeEmail(),
    body('password').isLength({ min: 6 })
      .withMessage('Password must be at least 6 characters').trim().escape(),
    body('username').notEmpty().withMessage('Username is required')
      .isAlphanumeric().withMessage('Username must be alphanumeric').trim().escape()
  ],
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({
        message: 'Validation failed',
        errors: errors.array().map(error => ({
          field: error.param,
          message: error.msg,
        }))
      });
    }
    res.send('User registered successfully');
  }
);

export default router;
```

Explicación:

1. `errors.array().map(error => {...})` — Transformamos la lista de errores para devolver un formato más legible.
2. `res.status(400).json({...})` — Enviamos una respuesta estructurada en JSON que contiene el campo y el mensaje de error específico.

Comparación con otras Herramientas: `express-validator vs Joi`

`express-validator` es ideal para proyectos en Express porque permite definir validaciones como middleware directamente en las rutas, integrándose en el flujo de trabajo del servidor. Sin embargo, existen otras bibliotecas como `Joi`, que es una herramienta más flexible y extensible, aunque no se integra directamente como middleware en Express.

Comparación:

- **Integración:** `express-validator` se adapta fácilmente al flujo de Express, mientras que `Joi`

requiere configuraciones adicionales.

- **Flexibilidad:** Joi ofrece una sintaxis más detallada y una mayor cantidad de opciones de validación.
- **Ecosistema:** express-validator se basa en la biblioteca validator.js, centrada en validación y sanitización de datos web.

Ejemplo Completo de Uso de express-validator en Conjunto con Router y express.json

Archivo app.js:

```
import express from 'express';
import userRoutes from './userRoutes.js';

const app = express();

app.use(express.json()); // Middleware para analizar JSON
app.use('/users', userRoutes);

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Archivo userRoutes.js:

```
import { Router } from 'express';
import { body, validationResult } from 'express-validator';

const router = Router();

// Validación y sanitización en la ruta de registro de usuario
router.post(
  '/register',
  [
    body('email').isEmail().withMessage('Please enter a valid email').normalizeEmail(),
    body('password').isLength({ min: 6 })
      .withMessage('Password must be at least 6 characters').trim().escape(),
    body('username').notEmpty().withMessage('Username is required')
      .isAlphanumeric().withMessage('Username must be alphanumeric').trim().escape()
  ],
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    res.send('User registered successfully');
  }
);

export default router;
```

Este ejemplo completo usa `Router`, define validaciones y sanitizaciones para datos de usuario, y gestiona errores en un formato claro. La combinación de `express-validator` y `Router` en Express es una solución robusta y práctica para validar datos en aplicaciones web.

Módulo 6: Buenas Prácticas de `express-validator` en Node.js

Al usar `express-validator` en aplicaciones Node.js, es importante seguir buenas prácticas para mejorar la organización, seguridad y eficiencia del código. Estas prácticas permiten estructurar las validaciones de datos de manera clara y confiable, optimizando el flujo de validación y facilitando el mantenimiento.

1. Modularizar las Validaciones

Es recomendable separar las validaciones de las rutas principales, especialmente en aplicaciones grandes. Crear funciones específicas para cada conjunto de validaciones ayuda a mantener el código ordenado y reutilizable.

Ejemplo:

```
// validationRules.js
import { body } from 'express-validator';

export const registerValidationRules = () => [
    body('email').isEmail().withMessage('Please enter a valid email address'),
    body('password').isLength({ min: 6 })
        .withMessage('Password must be at least 6 characters long'),
    body('username').notEmpty().withMessage('Username is required')
        .isAlphanumeric().withMessage('Username must be alphanumeric')
];
```

Uso en las Rutas:

```
import { Router } from 'express';
import { registerValidationRules } from './validationRules.js';
import { validationResult } from 'express-validator';

const router = Router();

router.post('/register', registerValidationRules(), (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    res.send('User registered successfully');
});

export default router;
```

2. Centralizar la Gestión de Errores

Manejar los errores de validación en un middleware separado ayuda a mantener las rutas limpias y a unificar el manejo de errores.

```
// errorMiddleware.js
import { validationResult } from 'express-validator';

export const handleValidationErrors = (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    next();
};
```

Uso en las Rutas:

```
router.post('/register', registerValidationRules(), handleValidationErrors, (req, res) => {
    res.send('User registered successfully');
});
```

3. Usar Sanitización en Datos de Entrada

La sanitización elimina caracteres innecesarios o potencialmente peligrosos en los datos de entrada, mejorando la seguridad.

```
body('email').isEmail().normalizeEmail(),
body('password').trim().escape(),
body('username').trim().escape()
```

4. Evitar Validaciones Redundantes

Evita validar datos innecesariamente, especialmente si ya han sido validados en el frontend o en pasos anteriores del flujo de datos.

5. Documentar las Validaciones

Proporciona documentación clara sobre los requisitos y restricciones de cada campo, lo cual facilita la integración de otros desarrolladores y la comprensión de la API.

6. Usar Mensajes de Error Claros

Los mensajes de error detallados ayudan a los usuarios a entender y corregir sus entradas.

7. Validar en el Cliente También

Implementa validaciones del lado del cliente para mejorar la experiencia del usuario, sin depender exclusivamente de ellas.

8. No Depender Exclusivamente de Validaciones del Lado del Cliente

Dado que los datos pueden ser manipulados antes de llegar al servidor, siempre valida en el servidor para garantizar la seguridad.

9. Formato Consistente de Respuesta de Errores

Devuelve errores en un formato JSON consistente, lo cual facilita el manejo de errores en el frontend.

```
return res.status(400).json({
  status: 'error',
  message: 'Validation failed',
  errors: errors.array().map(error => ({
    field: error.param,
    message: error.msg,
  })),
});
```

10. Validar en el Momento Adecuado

Aplica las validaciones cuando los datos son recibidos por primera vez en el servidor para evitar errores posteriores.

Ejemplo Completo Aplicando Buenas Prácticas

Archivo validationRules.js:

```
import { body } from 'express-validator';

export const registerValidationRules = () => [
  body('email').isEmail().withMessage('Please enter a valid email').normalizeEmail(),
  body('password').isLength({ min: 6 })
    .withMessage('Password must be at least 6 characters long').trim().escape(),
  body('username').notEmpty().withMessage('Username is required')
    .isAlphanumeric().withMessage('Username must be alphanumeric').trim().escape()
];
```

Archivo `errorMiddleware.js`:

```
import { validationResult } from 'express-validator';

export const handleValidationErrors = (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({
            status: 'error',
            message: 'Validation failed',
            errors: errors.array().map(error => ({
                field: error.param,
                message: error.msg,
            }))
        });
    }
    next();
};
```

Archivo `userRoutes.js`:

```
import { Router } from 'express';
import { registerValidationRules } from './validationRules.js';
import { handleValidationErrors } from './errorMiddleware.js';

const router = Router();

router.post('/register', registerValidationRules(), handleValidationErrors, (req, res) => {
    res.send('User registered successfully');
});

export default router;
```

Archivo `app.js`:

```
import express from 'express';
import userRoutes from './userRoutes.js';

const app = express();

app.use(express.json());
app.use('/users', userRoutes);

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Estas buenas prácticas aseguran que `express-validator` se utilice de manera eficiente, limpia y segura,

optimizando la validación de datos en una aplicación Node.js y mejorando la experiencia del desarrollador y del usuario.

Módulo 7: Introducción a EJS para Node.js

EJS (Embedded JavaScript) es un motor de plantillas que permite integrar JavaScript en archivos HTML, facilitando la generación de contenido dinámico en aplicaciones web. Al usar EJS con Express en Node.js, puedes crear vistas dinámicas que presentan datos de forma interactiva y permiten una estructura modular y mantenible para tus páginas web.

¿Qué es EJS?

EJS es un motor de plantillas que permite la incrustación de JavaScript en HTML. A diferencia de otros motores como Pug o Handlebars, EJS utiliza una sintaxis de JavaScript tradicional y sencilla que facilita el aprendizaje y la integración en aplicaciones Node.js. Con EJS, es posible pasar datos dinámicos a la vista y renderizar HTML con lógica condicional, bucles, y otros elementos propios de JavaScript, mejorando la interactividad y personalización de las aplicaciones web.

Instalación de EJS

Para empezar a utilizar EJS, primero debes instalar el paquete en tu proyecto:

```
npm install ejs
```

Luego, configura EJS como el motor de vistas en Express:

```
import express from 'express';
const app = express();

app.set('view engine', 'ejs');

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

¿Por Qué Usar EJS?

EJS permite construir vistas dinámicas y modulares, lo cual es fundamental en aplicaciones web modernas. Al usar EJS, puedes separar la lógica de presentación (frontend) de la lógica de negocio (backend), facilitando el mantenimiento y el desarrollo de nuevas funcionalidades.

Ejemplo Básico: Renderización de una Vista con Datos Dinámicos

A continuación, vamos a crear una vista que muestra un saludo personalizado basado en datos dinámicos proporcionados por el servidor.

Paso 1: Crear la Ruta

Define una ruta en Express que renderice una vista EJS y envíe datos dinámicos a la misma:

Archivo `app.js`:

```
import express from 'express';
const app = express();

app.set('view engine', 'ejs');

app.get('/greeting', (req, res) => {
  const name = "Carlos";
  res.render('greeting', { name });
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Explicación Paso a Paso:

1. `app.set('view engine', 'ejs');` — Configura EJS como el motor de vistas de la aplicación.
2. `app.get('/greeting', (req, res) => {...});` — Define una ruta GET en `/greeting` que renderizará la vista.
3. `res.render('greeting', { name });` — Renderiza la vista `greeting.ejs` y envía un objeto con el dato `name` a la vista.

Paso 2: Crear la Vista en EJS

Crea un archivo llamado `greeting.ejs` en una carpeta llamada `views`:

Archivo `views/greeting.ejs`:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Greeting</title>
</head>
<body>
<h1>Hello, <%= name %>!</h1>
</body>
</html>
```

Explicación de la Vista:

1. `<%= name %>` — Incrusta el valor de `name` en el HTML. EJS usa `<%= ... %>` para mostrar variables,

permitiendo que los datos dinámicos se integren en el HTML.

Al acceder a `http://localhost:3000/greeting`, el servidor renderizará la vista `greeting.ejs`, y mostrará `Hello, Carlos!`.

Ejemplo Intermedio: Estructuras de Control en EJS

EJS permite usar estructuras de control, como condicionales y bucles, para manejar datos de forma dinámica. Esto es útil para crear listas, mostrar mensajes condicionales y estructurar el HTML.

Ejemplo: Lista de Productos

Paso 1: Crear la Ruta

Define una nueva ruta que envíe un arreglo de productos a la vista:

Archivo `app.js`:

```
app.get('/products', (req, res) => {
  const products = [
    { name: 'Laptop', price: 1500 },
    { name: 'Smartphone', price: 700 },
    { name: 'Tablet', price: 300 },
  ];
  res.render('products', { products });
});
```

Paso 2: Crear la Vista con un Bucle

Archivo `views/products.ejs`:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Product List</title>
</head>
<body>
<h1>Product List</h1>
<ul>
  <% products.forEach(product => { %>
    <li><%= product.name %> - $<%= product.price %></li>
  <% }); %>
</ul>
</body>
</html>
```

Explicación de la Vista:

1. `<% products.forEach(product => { %> ... <% }) ; %>` — Usa un bucle `forEach` para iterar sobre el arreglo de productos y mostrar cada uno en una lista.
2. `<%= product.name %>` y `<%= product.price %>` — Muestra el nombre y precio de cada producto.

Este código generará una lista de productos con nombre y precio.

Buenas Prácticas para Usar EJS

1. **Organizar las Vistas en Módulos:** Al estructurar la aplicación, organiza las vistas en carpetas según su funcionalidad. Por ejemplo, coloca las vistas de usuario en `views/users` y las vistas de productos en `views/products`.
2. **Usar `partials` para Reutilizar Código:** EJS permite incluir `partials`, que son fragmentos de código reutilizables, como el encabezado, el pie de página y la barra de navegación.

Ejemplo de `partial` para Encabezado:

Archivo `views/partials/header.ejs`:

```
<header>
  <h1>My Website</h1>
  <nav>
    <a href="/">Home</a>
    <a href="/products">Products</a>
    <a href="/contact">Contact</a>
  </nav>
</header>
```

Uso del `partial` en la Vista Principal:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>My Website</title>
</head>
<body>
  <%- include('partials/header') %> <!-- Incluye el encabezado -->
  <main>
    <h2>Welcome to my website</h2>
    <p>Here you'll find our products and services.</p>
  </main>
</body>
</html>
```

3. **Escapar Datos para Evitar Inyecciones:** Usa `<%= ... %>` para mostrar datos de usuario y evitar ataques de inyección. EJS escapa automáticamente el contenido de `<%= ... %>` para prevenir riesgos de seguridad.

4. **Evitar Lógica Compleja en Vistas:** EJS permite usar JavaScript en las vistas, pero es recomendable mantener la lógica compleja en el backend para evitar sobrecargar las vistas y facilitar el mantenimiento.
-

Ejemplo Completo: Aplicación con EJS y Router en Express

Para demostrar el uso de EJS en conjunto con `Router`, vamos a crear una pequeña aplicación que muestra una lista de usuarios y el perfil de cada uno.

Paso 1: Configurar el `Router` y las Rutas

Archivo `app.js`:

```
import express from 'express';
import userRoutes from './routes/userRoutes.js';

const app = express();

app.set('view engine', 'ejs');
app.use('/users', userRoutes);

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Paso 2: Crear las Rutas de Usuario

Archivo `routes/userRoutes.js`:

```
import { Router } from 'express';

const router = Router();

// Lista de usuarios simulada
const users = [
  { id: 1, name: 'Alice', age: 30 },
  { id: 2, name: 'Bob', age: 25 },
  { id: 3, name: 'Charlie', age: 35 },
];

// Ruta para listar usuarios
router.get('/', (req, res) => {
  res.render('users', { users });
});

// Ruta para mostrar el perfil de un usuario específico
router.get('/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (user) {
```

```

        res.render('userProfile', { user });
    } else {
        res.status(404).send('User not found');
    }
});

export default router;

```

Paso 3: Crear las Vistas users.ejs y userProfile.ejs

Archivo views/users.ejs:

```

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>User List</title>
</head>
<body>
<h1>User List</h1>
<ul>
<%
users.forEach(user => { %>
    <li>
        <a href="/users/<%= user.id %>"><%= user.name %> (Age: <%= user.age %></a>
    </li>
<% }); %>
</ul>
</body>
</html>

```

Archivo views/userProfile.ejs:

```

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>User Profile</title>
</head>
<body>
<h1>User Profile</h1>
<p>Name: <%= user.name %></p>
<p>Age: <%= user.age %></p>
</body>
</html>

```

Explicación:

1. `router.get('/')` – Define la ruta principal para listar usuarios.
2. `router.get('/:id')` – Define la ruta de perfil, encontrando el usuario por ID y renderizando

`userProfile.ejs`.

3. En `users.ejs`, se genera una lista de usuarios con enlaces a sus perfiles individuales.

4. En `userProfile.ejs`, se muestra la información detallada del usuario.

Resumen

EJS es un motor de plantillas sencillo y poderoso que facilita la creación de contenido dinámico en aplicaciones Node.js. Al seguir una estructura modular con `Router` y buenas prácticas como el uso de `partials` y la sanitización de datos, puedes construir aplicaciones web seguras, eficientes y mantenibles.