Fundamentos y gestión de módulos

Desarrollo de Sistemas Web - Back End

Versión 1.0 F. Act 05/09/2024

Fundamentos

- Node.js es una plataforma de JavaScript del lado del servidor que utiliza un modelo de ejecución **single-threaded** para manejar operaciones de entrada/salida de manera eficiente. Su diseño **no bloqueante, concurrente y asincrónico** permite que el hilo único maneje múltiples operaciones simultáneamente sin detenerse en espera de resultados, gracias a un loop de eventos que gestiona las tareas y los eventos asincrónicos de manera fluida. Utiliza el motor **V8** para ejecutar JavaScript y convertirlo en código máquina nativo, mejorando el rendimiento.

- Node.js ofrece un sistema modular con **módulos integrados** como http para crear servidores web y fs para interactuar con el sistema de archivos, así como la capacidad de definir y usar **módulos propios**. El gestor de paquetes **NPM** facilita la instalación y gestión de dependencias mediante el archivo package.json, que también maneja las versiones y las dependencias de desarrollo.

- En cuanto a la configuración y ejecución, Node.js permite la gestión de **variables de entorno** para diferentes configuraciones de aplicación a través de archivos JSON o módulos específicos. El modelo de eventos de Node.js facilita la creación de aplicaciones asincrónicas mediante el uso de **APIs asincrónicas** y eventos personalizados. La arquitectura **MVC** se utiliza para estructurar aplicaciones de manera modular, separando las responsabilidades en Modelos, Vistas y Controladores para una mejor organización y mantenimiento del código.

Módulos

Un módulo en Node.js es un archivo de código JavaScript que proporciona un conjunto de funcionalidades y puede ser reutilizado en diferentes partes de una aplicación. Los módulos son una forma de dividir y organizar el código de tu aplicación de manera más eficiente y mantenible.

Hay dos tipos de módulos en Node.js:

- Los módulos nativos son módulos que forman parte de la propia instalación de Node.js y proporcionan funcionalidades esenciales para el funcionamiento del sistema.

- Los módulos de terceros son módulos que se han creado por terceros y se pueden instalar a través de NPM (Node Package Manager) como veremos más adelante.

module.exports en Node.js

En Node.js, cada archivo se considera un módulo independiente. La manera en que estos módulos exportan funcionalidades para ser utilizados en otros archivos es a través del objeto module.exports.

Qué es module.exports:

- module.exports es un objeto especial en cada módulo que se utiliza para definir qué se exportará desde el módulo para que otros módulos puedan usarlo.
- Es una propiedad del objeto module que representa el módulo actual.

Uso en el Código:

- **Definición del Módulo**: El código que definis dentro del archivo es encapsulado en un módulo. Cuando quieres que ciertas partes de ese código sean accesibles desde otros archivos, necesitas exportarlas.
- **Exportar la Clase**: Otros archivos puedan importarla y utilizarla.

Organización de archivos y carpetas

Organizar los archivos en una aplicación Node.js de manera efectiva es crucial para mantener el proyecto manejable y escalable.

En este documento te doy una estructura de carpetas y nombres de archivos que puedes utilizar para una aplicación Node.js, junto con descripciones para cada uno.

https://docs.google.com/document/d/1-MdCUFPyzNsIgwSAHKW3Cf5z7zNFtduwVoJwVo046J8/edit?usp=sharing

Módulos nativos más comunes

- **http: Proporciona funcionalidades para crear servidores y clientes HTTP.**
- **fs: Proporciona funcionalidades para leer y escribir archivos en el sistema de archivos del servidor.**
- **path: Proporciona funcionalidades para trabajar con rutas de archivos y directorios.**
- **util: Proporciona funcionalidades de utilidad diversa, como heredar de prototipos y ejecutar funciones de manera asíncrona.**
- **assert: Proporciona funcionalidades para realizar pruebas de afirmaciones en el código.**
- **net: Permite trabajar con la red utilizando diferentes protocolos.**
- **events: Módulo para implementar emisores de eventos (el patrón observador de node).**
- **os: Permite acceder a información al nivel del sistema operativo.**

El módulo del sistema de archivos (fs)

Uno de los módulos incorporados más utilizados en Node.js es el módulo `fs`, que se refiere al sistema de archivos. Este módulo proporciona una variedad de funciones para trabajar con archivos y directorios en el sistema de archivos.

Para leer un archivo de texto almacenado localmente, simplemente usamos el módulo `fs` para acceder y extraer el contenido del archivo, especificando su ruta y otros parámetros necesarios. El módulo `fs` ofrece métodos tanto síncronos como asíncronos para cubrir diferentes necesidades y situaciones, permitiendo manejar operaciones de archivo de manera eficiente.

Al igual que con otros módulos en Node.js, utilizamos la función `require` para importar las funciones del módulo `fs` y así tener acceso a las operaciones del sistema de archivos. La importación del módulo se realiza mediante el siguiente código:

```
const fs = require('fs');
```

Módulo HTTP


Protocolo HTTP
Para comunicarnos utilizamos una cadena de texto (URL) como identificador del host y del recurso
http ://user:pass @ desarrolloutil.com : 80 /account/data ? key=value& #hash
Protocolo
Autenticación
Hostname
Puerto
Ruta
Query/Search
Hash

Módulos propios

Además de utilizar módulos de Node, también podemos crear nuestros propios módulos para organizar mejor nuestro proyecto. Si no lo hacemos, tendremos todo el código en un único archivo y eso suele generar mucho desorden, sobre todo cuando los proyectos y el código crecen.

Para crear un propio módulo, debemos crear un archivo separado, por ejemplo, uno que se llame operaciones_basicas.js y agregar las siguientes líneas:

```
exports.producto = function (a, b) {
    return a * b;
};
exports.suma = function (a, b) {
    return a + b;
};
```

Exports es el objeto que se va a exportar y, por lo tanto, que se asignará al momento de hacer un require('./operaciones_basicas')

```
const operaciones_basicas = require('./operaciones_basicas');
console.log('El producto de 4 x 10 es ' + operaciones_basicas.producto(4, 10))
```

Gestor de paquetes: NPM (Node Package Manager)

Cuando instalamos Node.js, también obtuvimos una herramienta poderosa: **npm** (Node Package Manager).

npm es el gestor de paquetes para JavaScript. Nos permite:

- **Instalar y gestionar paquetes**: Añade módulos reutilizables a nuestros proyectos.
- **Publicar nuestros propios módulos**: Comparte tu código con otros desarrolladores.

npm se instala automáticamente con Node.js y es esencial para el desarrollo en JavaScript.

Para más información, manuales y documentación, visita su sitio web: npmjs.com.

Comparación entre CommonJS y ES6

CommonJS utiliza require para importar y module.exports para exportar.

ES6 utiliza import para importar y export para exportar.

Compatibilidad:

CommonJS es el sistema de módulos utilizado por defecto en Node.js.

ES6 es más moderno y está diseñado para el navegador, pero Node.js lo soporta si se configura adecuadamente ("type": "module" en package.json).

Carga de Módulos:

CommonJS carga módulos de manera sincrónica, lo que es adecuado para el entorno de Node.js.

ES6 permite la carga de módulos de manera asíncrona, lo que es útil en aplicaciones web.

El comando npm

El comando npm se usa en la línea de comandos de Node.js. Se invoca con npm seguido de la operación deseada. Por ejemplo:

npm install async

Este comando instala el paquete async en tu proyecto, guardándolo en la carpeta node_modules. Luego, puedes incluirlo en tu código con require:

require('async');

No necesitas especificar la ruta al paquete porque npm lo instala en node_modules, donde require lo encuentra automáticamente.

Subcomandos comunes de npm

help: Muestra ayuda sobre npm y sus subcomandos. Usa npm subcomando -h para obtener información sobre un subcomando específico, como npm install -h.

info: Proporciona información sobre un paquete específico.

init: Crea un archivo package.json para gestionar las dependencias de tu proyecto.

install: Instala paquetes, ya sea globalmente o localmente en tu proyecto.

¿Qué es package.json?

`package.json` es un archivo que se encuentra en la raíz de los proyectos Node.js. Este archivo gestiona toda la configuración y dependencias del proyecto. Es esencial para cualquier proyecto de Node.js porque:

1. **Define el proyecto**: Contiene metadatos como el nombre, la versión y la descripción del proyecto.
2. **Gestiona dependencias**: Especifica qué librerías y versiones se necesitan para que el proyecto funcione.
3. **Facilita scripts**: Permite definir comandos personalizados que se pueden ejecutar con `npm run`.
4. **Configura el entorno**: Define la versión de Node.js requerida y otras configuraciones.

El archivo package.json

Es esencial en proyectos de Node.js y JavaScript. Este archivo JSON contiene información clave sobre el proyecto y sus dependencias. A continuación, se destacan algunos de sus componentes principales:

- **name**: Nombre del proyecto.
- **version**: Versión actual del proyecto.
- **description**: Breve descripción del proyecto.
- **main**: Archivo principal del proyecto (por defecto suele ser index.js).
- **scripts**: Define comandos personalizados que se pueden ejecutar con npm run. Ejemplo: test, start, build.
- **dependencies**: Lista de paquetes que el proyecto necesita para funcionar. Se instalan con npm install.
- **devDependencies**: Paquetes necesarios solo durante el desarrollo, como herramientas de testing y construcción.
- **engines**: Específica versiones de Node.js o npm requeridas para el proyecto.

No es estrictamente necesario tener un archivo package.json en un proyecto de Node.js, pero es altamente recomendable por varias razones. El archivo package.json es una parte integral del ecosistema Node.js y proporciona varias funcionalidades importantes que facilitan el desarrollo y la gestión del proyecto

Razones para Incluir package.json

1. **Gestión de Dependencias**:
   - **Instalación de Paquetes**: package.json lista todas las dependencias de tu proyecto (librerías y módulos externos que tu aplicación necesita). Esto permite que cualquier desarrollador que trabaje en el proyecto pueda instalar todas las dependencias necesarias simplemente ejecutando npm install.
   - **Versiones Específicas**: Define versiones específicas de los paquetes para asegurar que todos los entornos (desarrollo, pruebas, producción) usen las mismas versiones de las dependencias, evitando problemas de compatibilidad.
2. **Configuración del Proyecto**:
   - **Scripts de Comando**: Puedes definir scripts personalizados en package.json para tareas comunes como iniciar el servidor, correr pruebas, o compilar código. Por ejemplo, npm start para iniciar la aplicación, npm test para ejecutar las pruebas, etc.
   - **Configuraciones de Entorno**: Define configuraciones del proyecto como el nombre, versión, descripción, autor, y licencias.

Razones para Incluir package.json

1. **Gestión de Versiones**:
   - **Versionado Semántico**: Puedes definir la versión de tu aplicación, lo que es útil para el control de versiones y la gestión de lanzamientos.
2. **Dependencias de Desarrollo**:
   - **Herramientas de Desarrollo**: Incluye dependencias específicas para el desarrollo y pruebas, como linters, herramientas de construcción, y frameworks de pruebas.
3. **Documentación**:
   - **Información del Proyecto**: Proporciona una vista rápida de las dependencias y scripts disponibles, así como información básica sobre el proyecto. Esto es útil para nuevos desarrolladores que se unen al proyecto.

Control de versiones

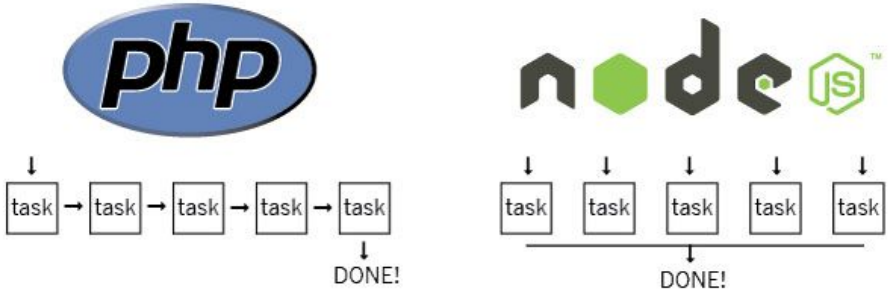
El modelo de control de versiones utilizado en npm se denomina control de versiones semántico. El control de versiones semántico es una convención formal para determinar el número de versión de las nuevas versiones de software. El estándar ayuda a los usuarios de software a comprender la gravedad de los cambios en cada nueva distribución.

Modelo de eventos

El flujo de ejecución de JavaScript en el navegador, así como en Node.js, está basado en un event loop (loop de eventos). Entender como este loop de eventos funciona es importante para optimizaciones y en algunos casos para utilizar la arquitectura correcta.

Una de las características más potentes de node.js es el trabajo con eventos. El trabajo con eventos es la asincronía por naturaleza, ya que nunca se sabe cuándo ocurrirá un evento. Normalmente en nuestras aplicaciones, comenzaremos la ejecución de servidores y los servicios que éstos lancen, y el sistema se quedará esperando que ocurra algún evento.

APIs Asíncronas



Usar APIs asíncronas sin bloques es aún más importante en Node que en el navegador, porque Node es un entorno de ejecución controlado por eventos de un solo hilo. "Un solo hilo" quiere decir que todas las peticiones al servidor son ejecutadas en el mismo hilo (en vez de dividirse en procesos separados). Este modelo es extremadamente eficiente en términos de velocidad y recursos del servidor, pero eso significa que si alguna de sus funciones llama a métodos sincrónicos que tomen demasiado tiempo en completarse, bloquearán no solo la solicitud actual, sino también cualquier otra petición que esté siendo manejada por tu aplicación web.

Eventos en NodeJS

En Node.js, el módulo events proporciona una implementación del patrón de diseño "EventEmitter". Este módulo permite que los objetos emitan eventos y que otros objetos se registren para escuchar esos eventos. Es una herramienta poderosa para manejar la comunicación entre diferentes partes de una aplicación y para crear un flujo de eventos asíncrono.

Instalación

El módulo events es parte del núcleo de Node.js, por lo que no necesitas instalarlo por separado. Puedes utilizarlo directamente en tus scripts de Node.js.

Uso Básico

Aquí hay un ejemplo básico de cómo usar el módulo `events` para crear un emisor de eventos (`EventEmitter`), emitir eventos y escuchar esos eventos.

```
// Importar el módulo 'events'
const EventEmitter = require('events');

// Crear una clase que extiende EventEmitter
class MyEmitter extends EventEmitter {}

// Crear una instancia del emisor de eventos
const myEmitter = new MyEmitter();

// Registrar un escuchador para el evento 'event'
myEmitter.on('event', () => {
  console.log('¡Un evento ha sido emitido!');
});

// Emitir el evento 'event'
myEmitter.emit('event');
```

El patrón de diseño CMV (Controller-Model-View) (Modelo-Vista-Controlador)

Es un enfoque similar al popular patrón MVC (Model-View-Controller) pero con algunas diferencias sutiles en el enfoque y la terminología. El patrón CMV se utiliza para estructurar aplicaciones web y mejorar la separación de responsabilidades.

1. Definición de Componentes

- **Model**: Maneja los datos de la aplicación.
- **View**: Se encarga de la presentación y la interfaz de usuario.
- **Controller**: Gestiona la interacción entre el modelo y la vista.