Unidad 1: Principios Básicos de JavaScript y Backend

Introducción a la Programación Backend y JavaScript

Introducción a la Programación de Backend

La programación de backend se refiere a todo lo que ocurre "detrás de escena" en una aplicación web. Mientras que el frontend se ocupa de las interacciones de los usuarios en sus pantallas, el backend es responsable de la lógica del servidor, la gestión de bases de datos, la autenticación de usuarios y la implementación de reglas de negocio. El backend es, esencialmente, el cerebro de una aplicación web, procesando las solicitudes de los usuarios, manteniendo la seguridad y la coherencia de los datos, y devolviendo las respuestas adecuadas al frontend.

Diferencia entre Frontend y Backend

Es importante entender la diferencia entre frontend y backend para apreciar cómo trabajan juntos en una aplicación web completa.

Frontend: Es la parte con la que los usuarios interactúan directamente. Incluye el diseño, la estructura, los botones, menús, imágenes y todo lo visible en pantalla. Se desarrolla con lenguajes como HTML, CSS y JavaScript, y se apoya en frameworks como React, Angular y Vue.js para crear interfaces dinámicas.

Backend: Es la parte que maneja la lógica de negocio, la base de datos, la autenticación y el almacenamiento de datos. Recibe las solicitudes del frontend, las procesa y devuelve las respuestas. Utiliza tecnologías como Node.js, Python, Java o PHP, junto con bases de datos como MySQL, PostgreSQL o MongoDB, y servidores como Apache o Nginx.

En resumen, el frontend se encarga de la lógica de presentación; el backend, de la lógica funcional.

Stack MERN

Stack de tecnologías

Un "stack" de tecnologías es el conjunto de herramientas, lenguajes y frameworks que se utilizan en conjunto para crear una aplicación. Puede incluir herramientas para el frontend, backend y bases de datos. Contar con un stack definido ayuda a asegurar que las partes de una aplicación trabajen bien juntas.

¿Qué es MERN?

MERN es un stack que permite desarrollar aplicaciones completas usando JavaScript de extremo a extremo. El acrónimo representa:

MongoDB: Base de datos NoSQL que almacena datos en documentos similares a objetos de JavaScript.

ExpressJS: Framework para Node.js que simplifica la creación de aplicaciones web y APIs.

ReactJS: Biblioteca de JavaScript para construir interfaces de usuario dinámicas y reutilizables.

NodeJS: Entorno de ejecución de JavaScript en el servidor, que maneja solicitudes, conexiones y lógica del backend.

¿Cómo trabajan juntas las tecnologías del stack MERN?

Frontend con ReactJS: Se encarga de mostrar la interfaz. Los usuarios interactúan con botones, formularios, menús, etc.

Servidor con ExpressJS y NodeJS: Cuando el usuario realiza una acción, como enviar un formulario, ReactJS envía una solicitud al servidor. Express y NodeJS la procesan.

Base de datos con MongoDB: Si la acción necesita consultar o guardar información, el servidor se comunica con MongoDB.

Respuesta al frontend: Una vez que se obtiene o guarda la información, el servidor responde a ReactJS, que actualiza la pantalla del usuario.

Beneficios del Stack MERN

- Todo el stack utiliza JavaScript, lo que facilita trabajar en frontend y backend con el mismo lenguaje.
- React permite construir componentes reutilizables que agilizan el desarrollo.
- MongoDB permite manejar grandes volúmenes de datos con flexibilidad.
- Permite desarrollar aplicaciones full-stack modernas con herramientas integradas entre sí.

Tipos de Datos en JavaScript

Tipos primitivos

String: Texto entre comillas. Ejemplo: "Hola", 'Juan'

Number: Números enteros o decimales. Ejemplo: 25, 3.14

Boolean: Solo puede ser true o false. Se usa en condiciones.

Null: Representa ausencia intencional de valor. Ejemplo: null

Undefined: Variable declarada pero sin valor asignado. Ejemplo: undefined

Tipos objeto

Object: Colección de pares clave-valor. Ejemplo: {nombre: "Ana", edad: 30}

Array: Lista ordenada de valores. Ejemplo: [1, 2, 3] o ["uno", true, 5]

Estos tipos de datos son esenciales para organizar y manipular información en JavaScript.

Paso a paso en Node

Paso 1: Abrir el archivo

Utiliza un editor de texto como Visual Studio Code, Sublime Text o Atom para abrir el archivo JavaScript que quieres ejecutar.

Paso 2: Ejecutar el archivo con Node.js

Abre la terminal de tu sistema y navega a la carpeta del archivo con el comando cd. Luego ejecuta el archivo con:

node nombreDelArchivo.js
Por ejemplo: node app.js

Esto iniciará la ejecución del archivo usando Node.js.

Nota importante

Asegúrate de tener Node.js instalado en tu computadora. Si no lo tienes, puedes descargarlo desde su sitio web oficial y seguir las instrucciones de instalación según tu sistema operativo. Una vez instalado, podrás ejecutar archivos JavaScript desde la terminal sin problemas.

Variables y Estructuras en JavaScript

Declaración y Uso de Variables en JavaScript

En JavaScript, las variables son fundamentales para almacenar y manipular datos en el código. Para declarar variables, existen tres palabras clave principales:

`let`, `const`y `var`. Cada una de ellas tiene características y usos específicos que es importante entender. Para comprenderlas mejor, es importante enteder el concepto de “alcance”.

Alcance (Scope)

El alcance de una variable determina dónde puedo “llamar y usar” a la variable. En JavaScript, el alcance se divide principalmente en:

- Alcance global: Variables accesibles desde cualquier parte del código.
- Alcance de función: Variables accesibles solo dentro de la función donde fueron declaradas.
- Alcance de bloque: Variables accesibles solo dentro del bloque de código (delimitado por `{}`) donde fueron declaradas.

Usar

`let` y `const` en lugar de `var` es generalmente recomendado para evitar problemas de alcance y mejorar la legibilidad y mantenibilidad del código.

`let`

Es la forma moderna de declarar variables y tienen alcance de bloque. Esto significa que las variables declaradas con

`let` solo son accesibles dentro del bloque de código en el que fueron definidas (como en funciones, bloques `if`, bucles `for`, etc.). Una de las principales ventajas de `let` es que permite modificar el valor de la variable después de su declaración.

`const`

Se utiliza para declarar constantes, es decir, variables cuyo valor no debe cambiar una vez asignado. Al igual que

`let`, `const` tiene alcance de bloque. Sin embargo, a diferencia de `let`, una variable declarada con `const` debe ser inicializada en el momento de su declaración y no puede ser reasignada posteriormente. Es importante destacar que si la constante es un objeto o un array, sus propiedades o elementos sí pueden modificarse, pero no se puede reasignar la variable a un nuevo objeto o array.

`var`

Es la forma más antigua de declarar variables en JavaScript. Una variable declarada con

`var` tiene alcance global. Esto significa que puede ser accedida desde cualquier parte del código dentro de su función contenedora o en todo el script si está fuera de una función. Sin embargo, `var` tiene un comportamiento peculiar conocido como hoisting, donde las declaraciones de variables se elevan al inicio de su contexto (función o global) durante la ejecución, lo que puede llevar a errores sutiles si no se entiende bien. Su uso está limitado y no se recomienda su implementación en aplicaciones modernas.

Introducción a ECMAScript y sus versiones

ECMAScript es el estándar que define cómo funciona JavaScript. A lo largo de los años, han surgido varias versiones de ECMAScript, cada una introduciendo nuevas características que mejoran la sintaxis y la funcionalidad del lenguaje. Una de las versiones más importantes es ES6, también conocida como ECMAScript 2015.

Novedades de ES6

ES6 trajo una serie de mejoras significativas a JavaScript. A continuación, destacamos algunas de las más importantes:

1. `let` y `const` Antes de ES6, las variables en JavaScript se declaraban principalmente usando `var`, lo que podía llevar a comportamientos inesperados debido a su alcance de función (function scope). ES6 introdujo `let` y `const`, que permiten un control más preciso del alcance de las variables.

- `let`: Declara variables con un alcance de bloque (block scope), lo que significa que solo están disponibles dentro del bloque donde fueron declaradas.
- `const`: Similar a `let`, pero se usa para declarar constantes, es decir, variables cuyo valor no puede ser reasignado después de su inicialización.

2.
3. Funciones Flecha (`arrow functions`) ES6 introdujo una nueva forma de escribir funciones usando la sintaxis de "funciones flecha". Estas funciones son más concisas y no vinculan su propio valor de `this`, lo que resulta útil en ciertas situaciones, especialmente cuando se trabaja con métodos de objetos o callbacks. Ejemplo de función flecha: const sumar = (a, b) => a + b;

Mejora en la Sintaxis y Funcionalidad

Las características introducidas en ES6 no solo hacen que el código sea más legible y mantenible, sino que también evitan errores comunes.

`let` y `const` ayudan a gestionar mejor el alcance de las variables, reduciendo la posibilidad de errores debido a variables redefinidas o accesibles fuera de su contexto. Las funciones flecha simplifican la escritura de funciones, especialmente en código donde las funciones anónimas se usan con frecuencia.

Funciones y Scopes en JavaScript

Funciones y Scopes en JavaScript

Funciones en JavaScript

En JavaScript, las funciones son bloques de código reutilizables que permiten ejecutar tareas específicas dentro de un programa. Son fundamentales para el desarrollo de aplicaciones, ya que permiten organizar el código de manera modular y eficiente. Una función puede aceptar entradas (conocidas como parámetros) y devolver un resultado.

Funciones Tradicionales

La sintaxis para declarar una función tradicional en JavaScript es mediante la palabra clave

`function`, seguida por el nombre de la función, los parámetros entre paréntesis `()`, y el cuerpo de la función con las instrucciones a ejecutar entre llaves `{}`. Las funciones tradicionales pueden ser declaradas en cualquier parte del código y son "hoisted", lo que significa que pueden ser invocadas antes de su definición en el código.

Ejemplo de una Función Tradicional

Una función tradicional podría verse así:

`function sumar(a, b) { return a + b; }`. En este caso, `sumar` es el nombre de la función, `a` y `b` son los parámetros, y la función devuelve la suma de `a` y `b`.

Funciones Flecha (Arrow Functions)

Introducidas en ECMAScript 6 (ES6), las funciones flecha proporcionan una forma más concisa de escribir funciones. Se declaran usando la sintaxis

`(parámetros) => { cuerpo de la función }`. Las funciones flecha no tienen su propio contexto `this`; en cambio, heredan el `this` del contexto en el que fueron definidas. Esto las hace especialmente útiles en funciones anónimas y callbacks, donde el manejo del contexto puede ser complicado.

Diferencias Clave entre Funciones Tradicionales y Flecha

1. Sintaxis: Las funciones flecha eliminan la necesidad de la palabra clave `function` y, si el cuerpo de la función es una sola expresión, también permiten omitir las llaves `{}` y el `return`.
2. Contexto de `this`: En funciones tradicionales, `this` hace referencia al objeto que invoca la función. En funciones flecha, `this` se refiere al contexto léxico (el entorno donde la función fue definida).

Ejemplo de una Función Flecha

Una función flecha equivalente a la función tradicional anterior sería:

`const sumar = (a, b) => a + b;`. Aquí, la función `sumar` es una expresión de función almacenada en una constante.

Uso en Desarrollo de Aplicaciones

Las funciones son esenciales en JavaScript para estructurar el código de manera clara y eficiente. Las funciones tradicionales son útiles cuando se requiere flexibilidad en la gestión de

`this`, mientras que las funciones flecha son preferibles para callbacks y funciones que no necesitan su propio contexto `this`. El uso adecuado de ambas formas de funciones contribuye a un código más limpio, modular y mantenible.

Scopes en JavaScript

El "scope" o alcance en JavaScript se refiere al contexto en el cual las variables y funciones pueden ser accedidas o referenciadas. En términos simples, el scope determina la visibilidad de una variable dentro de diferentes partes del código.

Scope Global y Scope Local

Scope Global

Cuando una variable se declara fuera de cualquier función o bloque, se dice que tiene un scope global. Esto significa que la variable es accesible desde cualquier parte del código, independientemente de dónde se encuentre. Las variables globales permanecen en la memoria durante toda la ejecución del script, lo que puede afectar el rendimiento y causar conflictos si se utilizan nombres de variables similares en diferentes partes del programa.

Scope Local

Una variable tiene un scope local cuando es declarada dentro de una función o bloque. Las variables locales solo son accesibles dentro de la función o bloque en el que fueron declaradas, y no pueden ser referenciadas fuera de ese contexto. Esto es útil para evitar conflictos entre variables que podrían tener el mismo nombre pero deben ser utilizadas en diferentes partes del código sin interferir entre sí.

Dentro del scope local, hay dos tipos importantes de alcance:

- Alcance de función: Una variable declarada dentro de una función solo puede ser utilizada dentro de esa función.
- Alcance de bloque: Con la introducción de `let` y `const` en ES6, las variables pueden ser limitadas al bloque de código (por ejemplo, dentro de un `if`, `for`, o `while`) en el que fueron declaradas, no siendo accesibles fuera de ese bloque.

Cómo el Scope Afecta la Accesibilidad de Variables

El scope influye directamente en la accesibilidad de las variables. Las variables globales pueden ser accedidas desde cualquier lugar del código, mientras que las variables locales solo pueden ser accedidas dentro del bloque o función donde fueron declaradas. Entender y manejar adecuadamente el scope es crucial para evitar errores comunes, como la sobrescritura accidental de variables globales o el intento de acceder a variables locales desde fuera de su alcance.

Un buen manejo del scope permite a los desarrolladores escribir código más seguro, modular y eficiente, reduciendo la posibilidad de errores y mejorando la mantenibilidad del software.

Template Strings y Clases en JavaScript

Explicación sobre las Template Strings en JavaScript

Las template strings o cadenas de plantilla son una característica introducida en ES6 que permite crear strings de manera más sencilla y flexible en JavaScript. A diferencia de las cadenas de texto tradicionales, que se definen con comillas simples o dobles, las template strings se definen usando acentos graves (```).

Uso de Template Strings

Una de las principales ventajas de las template strings es la capacidad de insertar variables y expresiones directamente dentro de la cadena sin necesidad de concatenar manualmente. Para insertar una variable o expresión dentro de una template string, se utiliza la sintaxis

`${expresión}`.
Ventajas sobre la concatenación tradicional:

1. Legibilidad: Con las template strings, el código es más legible y fácil de mantener, especialmente cuando se trabaja con múltiples variables o expresiones dentro de una cadena.
2. Inserción directa de variables: En lugar de concatenar strings con el operador `+`, las template strings permiten insertar variables directamente en la cadena, evitando errores comunes y haciendo el código más limpio.
3. Expresiones complejas: Además de variables simples, se pueden incluir expresiones más complejas dentro de las template strings, como operaciones matemáticas o llamadas a funciones, lo que añade más flexibilidad al código.

En resumen, las template strings simplifican y mejoran la forma en que se manejan las cadenas en JavaScript, ofreciendo una alternativa más poderosa y versátil a la concatenación tradicional.

Usando Template Strings

Clases y Objetos en JavaScript

En JavaScript, una clase es un molde para crear objetos que comparten propiedades y métodos comunes. Las clases son una parte fundamental de la programación orientada a objetos (OOP) y permiten estructurar el código de manera más organizada y modular.

Definición de una Clase

Para crear una clase en JavaScript, se utiliza la palabra clave

`class`, seguida del nombre de la clase. Dentro de la clase, se define un método especial llamado constructor, que sirve para configurar las propiedades del objeto al instanciar la clase.

Propiedades

Las propiedades son características del objeto que se almacenan como variables dentro de la clase. Estas propiedades se definen dentro del constructor de la clase y se inicializan con valores que se pasan como argumentos al crear una instancia del objeto.

Por ejemplo, si se define una clase

`Coche`, las propiedades podrían incluir `marca`, `modelo` y `año`. Estas propiedades representan las características que cada instancia del objeto `Coche` tendrá.

Métodos

Los métodos son funciones definidas dentro de una clase que describen los comportamientos o acciones que un objeto puede realizar. Estos métodos pueden manipular las propiedades del objeto y realizar operaciones basadas en ellas.

Por ejemplo, en la clase

`Coche`, un método podría ser `arrancar`, que simula el encendido del coche, o `acelerar`, que aumenta la velocidad del coche.

Creación de Instancias

Una instancia es un objeto creado a partir de una clase. Cada instancia tiene sus propias copias de las propiedades definidas en la clase, pero comparte los métodos con otras instancias. Al crear una instancia, se utiliza la palabra clave

`new`seguida del nombre de la clase y los valores iniciales de las propiedades.
Por ejemplo, al crear una instancia de la clase:

`Coche`, podríamos especificar que se trata de un coche de la marca Toyota, modelo Corolla, del año 2020. Esta instancia tendrá sus propias propiedades `marca`, `modelo` y `año`, y podrá utilizar los métodos definidos en la clase `Coche`.

Ejemplo Conceptual

Imagina que defines una clase :

`Coche` con las propiedades `marca`, `modelo`, y `año`, y un método `arrancar`. Luego, creas dos instancias: un Toyota Corolla 2020 y un Ford Mustang 2021. Aunque ambos coches comparten la misma estructura (definida por la clase `Coche`), cada uno tiene sus propios valores específicos para `marca`, `modelo`, y `año`. Ambos pueden utilizar el método `arrancar`, pero ese método actuará sobre las propiedades específicas de cada coche.

Conclusión

Las clases en JavaScript son herramientas poderosas para estructurar y organizar el código de manera eficiente, permitiendo la creación de objetos que comparten una estructura común pero que pueden tener comportamientos y datos únicos. Al entender cómo definir clases, crear métodos y propiedades, y generar instancias, los desarrolladores pueden construir aplicaciones más modulares, reutilizables y mantenibles.

Resumen y Conclusión

Repaso y Conclusión

En esta unidad, hemos explorado los principios fundamentales de JavaScript, un lenguaje de programación clave en el desarrollo web. Comenzamos con una comprensión básica de cómo JavaScript permite manipular elementos en el navegador y construir aplicaciones dinámicas. Aprendimos sobre la declaración y uso de variables, comprendiendo las diferencias entre

`var`, `let` y `const`, y cómo el alcance (scope) de estas variables afecta su accesibilidad y comportamiento en el código.
También profundizamos en el concepto de funciones, comprendiendo su estructura, cómo se definen y utilizan para modularizar el código, y cómo las funciones flecha simplifican la escritura y el manejo de funciones anónimas. Luego, avanzamos en el estudio de las funciones flecha, que ofrecen una sintaxis más concisa en comparación con las funciones tradicionales.

Finalmente, abordamos los fundamentos de la programación orientada a objetos (OOP) en JavaScript. Aprendimos a definir clases, a establecer propiedades y métodos dentro de ellas, y a crear instancias de objetos. Este enfoque orientado a objetos facilita la creación de estructuras de código más organizadas, reutilizables y fáciles de mantener.

En resumen, esta unidad ha proporcionado una base sólida en los aspectos clave de JavaScript, preparando a los estudiantes para abordar proyectos de desarrollo con una comprensión clara de las herramientas y técnicas modernas del lenguaje.
