Unidad 8 - Framework

Módulos en JavaScript

Inicialmente, JavaScript se usaba para pequeñas tareas de scripting en páginas web. Hoy en día, es la base de aplicaciones completas, tanto en navegadores como en entornos como Node.js.

Para manejar la complejidad de los proyectos, JavaScript permite dividir el código en módulos reutilizables. Node.js incorporó esta capacidad hace tiempo, y ahora todos los navegadores modernos soportan módulos de forma nativa.

Exportar características de un módulo

Para que un módulo comparta funcionalidades, se deben **exportar** elementos con la palabra clave `export`.

La forma más sencilla de utilizar esta sentencia es escribiéndola antes de lo que se quiere exportar.

```
// exportación directa
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);
}
```

También se pueden exportar múltiples elementos al final del archivo:

```
export { name, draw, reportArea, reportPerimeter };
```

Se pueden exportar funciones, variables y clases, pero solo desde el nivel más externo (no dentro de funciones).

Importar características en un script

Para importar un módulo en otro archivo, usamos `import`:

```
import { name, draw, reportArea, reportPerimeter } from "./modules/square.js";
```

- **Entre llaves** `{}` → Importamos elementos específicos.
- **Desde** `from "..."` → Indicamos la ruta del módulo.

```
const myCanvas = create("myCanvas", document.body, 480, 320);
const reportList = createReportList(myCanvas.id);

const square1 = draw(myCanvas.ctx, 50, 50, 100, "blue");
```

```
reportArea(square1.length, reportList);
reportPerimeter(square1.length, reportList);
```

Nota: Los valores importados son de solo lectura, como una constante (`const`).

Importar recursos no JavaScript

Además de código, se pueden importar archivos JSON, CSS y otros formatos:

```
import colors from "./colors.json" with { type: "json" };
import styles from "./styles.css" with { type: "css" };
```

Uso de módulos en HTML

Para usar módulos en un HTML, se debe declarar `type="module"` en el script:

```
<script type="module" src="main.js"></script>
```

También se puede escribir código directamente en el HTML:

```
<script type="module">
  // Código del módulo
</script>
```

`import` y `export` solo funcionan en scripts con `type="module"` (sin esta declaración, se generará un error de sintaxis).

Diferencias entre módulos y scripts clásicos

- Se ejecutan solo en servidores web (no en `file://`).
- Siempre funcionan en modo estricto (`"use strict"`).
- Se ejecutan automáticamente después de que el documento está listo, como si tuvieran el atributo `defer`.
- Se ejecutan una sola vez, incluso si son llamados múltiples veces.
- No afectan el **ámbito global**: las variables importadas solo existen dentro del módulo que las recibe.

Exportación por defecto (`default export`)

Además de la exportación nombrada, existe la **exportación por defecto**, que permite exportar un solo valor sin necesidad de llaves `{}`.

```
export default function randomSquare() {
  // Crea un cuadrado aleatorio
}
```

```
import randomSquare from "./modules/square.js";

// Equivalente a:
// import { default as randomSquare } from "./modules/square.js";
```

La exportación por defecto facilita la compatibilidad con otros sistemas de módulos.

Node.js

Node.js es un entorno de ejecución de JavaScript de código abierto, multiplataforma y gratuito que permite ejecutar código JavaScript fuera del navegador. Utiliza el motor V8 de Google Chrome y está diseñado para manejar operaciones de entrada/salida de manera asincrónica y sin bloqueo.

A diferencia de otros entornos, Node.js opera en un solo hilo, gestionando múltiples solicitudes simultáneamente sin necesidad de crear nuevos subprocesos. En lugar de bloquear la ejecución mientras espera respuestas de operaciones de E/S (como lectura de archivos o acceso a bases de datos), Node.js continúa ejecutando otras tareas y reanuda la operación cuando recibe la respuesta.

Esto permite manejar miles de conexiones concurrentes de manera eficiente, reduciendo la sobrecarga de administración de subprocesos y mejorando el rendimiento en aplicaciones de alto tráfico.

Instalación de Node.js

Para instalar Node.js, visita su sitio web oficial: https://nodejs.org/es. Se recomienda descargar la versión **LTS (Long-Term Support)** para mayor estabilidad.

Se puede instalar mediante el instalador oficial o a través de un gestor de paquetes.

Para verificar la instalación, ejecuta en la terminal:

```
node --version
```

Si el comando se ejecuta correctamente, Node.js está instalado y su versión se mostrará en pantalla.

Ejecutar Node.js

Ejecutar un archivo JavaScript

```
node script.js
```

Ejecutar código en el entorno interactivo (REPL)

Node.js proporciona un entorno interactivo llamado **REPL (Read-Eval-Print Loop)**, donde se pueden ejecutar comandos de JavaScript en tiempo real.

Para iniciarlo, simplemente escribe: `node`

Aparecerá el prompt de Node.js (`>`) donde puedes ingresar comandos, por ejemplo:

```
> 3 + 4
7
```

Para salir del REPL, usa `Ctrl + C` dos veces o escribe `.exit`.

Diferencias entre JavaScript en el navegador y en Node.js

- `console.log()` funciona en ambos entornos, aunque en los navegadores los mensajes pueden aparecer con colores diferentes en la consola.
- El objeto `window` no existe en Node.js, ya que es exclusivo del navegador.
- El objeto `globalThis` es el equivalente de `window` en Node.js, pero apunta a `global` en lugar de `window`.

Módulos en Node.js

El patrón de módulos permite organizar el código en componentes reutilizables e independientes. Cada módulo tiene su propia interfaz pública y solo expone las funciones y variables necesarias para interactuar con otros módulos.

Sistemas de módulos en Node.js

Cuando surgió Node.js en 2009, JavaScript no tenía un sistema de módulos nativo, por lo que Node implementó su propio sistema: **CommonJS**.

Más tarde, con la introducción de **ECMAScript Modules (ESM)** en 2015, JavaScript adoptó un estándar oficial. Node.js es compatible con ESM desde la versión 12 (2019).

Extensiones de archivos en Node.js:

- `.cjs` → Para módulos CommonJS
- `.mjs` → Para módulos ECMAScript
- `.js` → Puede ser CommonJS o ESM, dependiendo del archivo `package.json`

CommonJS (Sistema heredado)

CommonJS usa `require` para importar y `module.exports` para exportar.

```
// Importar un módulo
const otroModulo = require('./otroModulo');

// Exportar un módulo
module.exports = {
  foo: 'bar'
};
```

ECMAScript Modules (ESM) - Estándar actual

ESM usa `import` y `export` para gestionar módulos.

```
// Importar un módulo
import otroModulo from './otroModulo';

// Exportar un módulo
export const foo = 'bar';
```

ESM es más moderno, portátil y compatible con navegadores y servidores. Además, se cargan de forma asíncrona, lo que mejora la eficiencia.

Módulos nativos de Node.js

Node.js incluye varios módulos nativos que permiten interactuar con el sistema operativo, el sistema de archivos y más.

Módulo "os" (Información del sistema operativo)

```
import os from "node:os";

console.log("Sistema operativo:", os.platform());
console.log("Versión:", os.release());
console.log("Arquitectura:", os.arch());
console.log("CPUs:", os.cpus());
console.log("Memoria libre (bytes):", os.freemem());
console.log("Memoria total (bytes):", os.totalmem());
```

Módulo "fs" (Sistema de archivos)

```
import fs from "node:fs";

const stats = fs.statSync('./archivo.txt');

console.log(
   stats.isFile(), // ¿Es un archivo?
   stats.isDirectory(), // ¿Es un directorio?
   stats.isSymbolicLink(), // ¿Es un enlace simbólico?
   stats.size // Tamaño en bytes
);
```

Gestión de paquetes en Node.js

¿Qué es un paquete en Node.js?

Un **paquete** es una carpeta con un archivo `package.json` que describe su contenido y dependencias.

Campos clave en `package.json`

```
{
  "type": "module",
  "main": "./index.js"
}
```

- `type` → Define si el proyecto usa CommonJS (`commonjs`) o ESM (`module`).
- `main` → Especifica el archivo principal del paquete.
- `packageManager` → Recomienda el gestor de paquetes preferido.
- `exports` e `imports` → Definen qué archivos pueden importarse desde fuera del paquete.

Gestores de paquetes en Node.js

Existen varios gestores de paquetes para administrar dependencias en Node.js:

Gestor	Descripción
npm (Node Package Manager)	Viene preinstalado con Node.js y usa `package-lock.json` para manejar versiones exactas de dependencias.
Yarn	Alternativa a npm, optimizada para rendimiento y eficiencia. Usa `yarn.lock` para garantizar versiones coherentes.
pnpm (*Performant npm*)	Reduce el espacio en disco almacenando módulos en un único lugar y evitando duplicados.

npm (Node Package Manager)

npm es tanto un **repositorio de paquetes** como una **herramienta de gestión de dependencias** para proyectos en Node.js. Se instala automáticamente junto con Node.js y permite instalar, actualizar y gestionar paquetes de manera eficiente.

Para verificar qué versión de npm tienes instalada, usa:

```
npm --version
```

Inicializar un proyecto

Para iniciar un proyecto con npm y generar el archivo `package.json`, usa:

```
npm init
```

Este comando te pedirá información sobre el proyecto. Si quieres aceptar los valores por defecto, usa:

```
npm init --yes
```

Ejemplo de un `package.json` generado:

```
{
  "name": "mi-proyecto",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
     "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Campos clave en `package.json`

- `name` **y** `version` → Identifican el proyecto.

  ```
  {
     "name": "mi-app",
     "version": "1.0.0"
  }
  ```

  La versión sigue el formato SemVer (MAJOR.MINOR.PATCH).

- `scripts` → Permiten definir comandos personalizados.

  ```
  {
     "scripts": {
        "dev": "astro dev",
        "start": "astro dev",
        "build": "astro build",
     }
  }
  ```

  Para ejecutar un script:

  ```
  npm run dev
  ```

- `dependencies` **y** `devDependencies` → Almacenan las librerías utilizadas en el proyecto.

  ```
  {
     "dependencies": {
        "express": "^5.0.0",
        "react": "^18.3.1"
     },
  ```

```
  "devDependencies": {
    "vitest": "^2.1.1",
  }
}
```

`dependencies` → *Para producción.*

`devDependencies` → *Solo necesarias en desarrollo.*

- `keywords` → Etiquetas descriptivas.

```
{
  "keywords": ["nodejs", "backend", "API"]
}
```

- `license` → Define la licencia del proyecto.

```
{
  "license": "MIT"
}
```

- `repository` → Especifica la URL del repositorio del proyecto.

```
{
  "repository": {
    "type": "git",
    "url": "https://github.com/usuario/mi-proyecto"
  }
}
```

Instalar paquetes

Instalar todas las dependencias

Si el proyecto ya tiene un `package.json`, usa:

```
npm install
```

Esto instalará todas las dependencias en la carpeta `node_modules`.

Instalar un paquete específico

```
npm install <nombre-paquete>
```

Alias: `npm add`, `npm i`, `npm in`.

Por defecto, se añade a `dependencies`. Para instalar en otra sección:

- `-D` o `--save-dev` → Guarda en `devDependencies`.
- `-O` o `--save-optional` → Guarda en `optionalDependencies`.
- `--no-save` → No lo guarda en `package.json`.

```
npm install picocolors
```

Esto agregará en `package.json`:

```
"dependencies": {
   "picocolors": "^1.1.1"
}
```

Actualizar paquetes

Para actualizar todas las dependencias según lo definido en `package.json`:

```
npm update
```

Para actualizar solo un paquete específico:

```
npm update <nombre-paquete>
```

Desinstalar paquetes

Para eliminar un paquete del proyecto:

```
npm uninstall <nombre-paquete>
```

Alias: `npm remove`, `npm rm`, `npm r`, `npm un`, `npm unlink`.

React

React es una **biblioteca de código abierto** diseñada para la construcción de **interfaces de usuario** interactivas y dinámicas, ideal para desarrollar **aplicaciones de una sola página (SPA)**. Es mantenida por **Meta (Facebook)** y la comunidad de código abierto.

Su enfoque se basa en la creación de **componentes reutilizables**, que son piezas individuales de una interfaz de usuario.

Incorporar React en una página web

React se puede utilizar directamente en una página web importando su biblioteca junto con ReactDOM:

```
<body>
  <div id="app"></div>
```

```
  <script type="module">
    import React from "https://esm.sh/react@18.2.0";
    import ReactDOM from "https://esm.sh/react-dom@18.2.0/client";

    // Crear elementos de React
    const button1 = React.createElement("button", { "data-id": "123" }, "Me gusta");
    const button2 = React.createElement("button", { "data-id": "123" }, "Me gusta");
    const button3 = React.createElement("button", { "data-id": "123" }, "Me gusta");
    const div = React.createElement(React.Fragment, null, [button1, button2, button3]);

    // Renderizar en el root
    const appDomElement = document.getElementById("app");
    const root = ReactDOM.createRoot(appDomElement);
    root.render(div);
  </script>
</body>
```

En este ejemplo se usan dos bibliotecas clave:

- **React**: React permite **crear y gestionar componentes de la interfaz**. Su función principal en este caso es `createElement`, que permite construir elementos de React.

  ```
  import React from "https://esm.sh/react@18.2.0";

  // Crear elementos
  const button1 = React.createElement("button", { "data-id": "123" }, "Me gusta");
  const button2 = React.createElement("button", { "data-id": "123" }, "Me gusta");
  const button3 = React.createElement("button", { "data-id": "123" }, "Me gusta");

  const div = React.createElement(React.Fragment, null, [button1, button2, button3]);
  ```

  `React.createElement(type, props, ...children)` → Permite crear un **elemento React** con una estructura específica.

  - `type` → El tipo de elemento (por ejemplo, `"div"`, `"button"`, un componente, `React.Fragment`, etc.).
  - `props` → Un objeto con propiedades (o `null` si no tiene).
  - `children` → Contenido del componente (puede ser texto, otros elementos o un array de elementos).
  - **Devuelve** un objeto React Element que luego se renderiza en el DOM.

- **ReactDOM**: `react-dom` es el paquete que conecta React con el DOM del navegador. Proporciona métodos para renderizar y administrar los componentes dentro de un nodo HTML.

  ```
  import ReactDOM from "https://esm.sh/react-dom@18.2.0/client";
  ```

```
const appDomElement = document.getElementById("app");
const root = ReactDOM.createRoot(appDomElement);
root.render(div);
```

`ReactDOM.createRoot(domNode, options?)` → Crea un contenedor raíz donde se renderizarán los componentes de React.

- `domNode` → Nodo del DOM donde se montará React.
- `options?` → Parámetros opcionales (poco usados).
- **Devuelve** un objeto con métodos como `render()` y `unmount()`.

`root.render(reactNode)` → Renderiza un elemento React dentro de la raíz de React en el DOM.