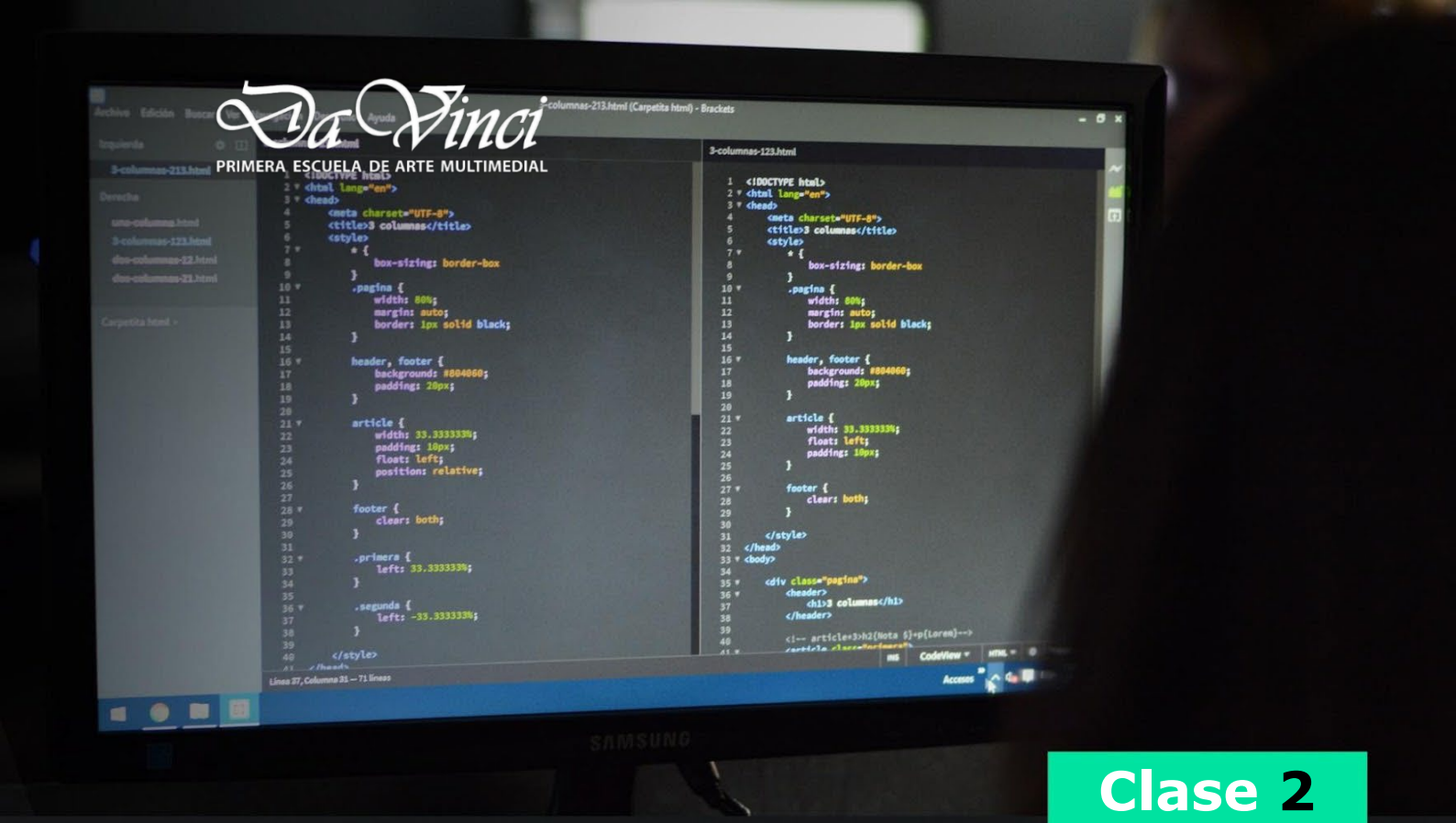

Clase 2

Diseño y Programación Web

Materia:
Aplicaciones Híbridas

Docente contenidista: MARCOS GALBÁN, Camila Belén

Herramientas y Gestión de Proyectos

¿Qué es Git?

Git es un sistema de control de versiones distribuido que permite a los desarrolladores rastrear cambios en el código fuente durante el desarrollo de software. Fue creado por Linus Torvalds en 2005 para el desarrollo del núcleo de Linux. Git facilita la colaboración entre programadores y ayuda a gestionar el historial de versiones de un proyecto.

Características de Git

- **Distribuido:** Cada desarrollador tiene una copia completa del historial del proyecto.
- **Eficiente:** Git está diseñado para manejar proyectos grandes con velocidad y eficiencia.
- **Rastreo de cambios:** Permite rastrear los cambios realizados en los archivos y revertir a versiones anteriores.

¿Qué es GitHub?

GitHub es una plataforma basada en la web que utiliza Git para el control de versiones y permite a los desarrolladores almacenar, gestionar y colaborar en proyectos de software. Además, GitHub ofrece herramientas adicionales como la integración continua, la gestión de proyectos, y la documentación.

Funcionalidades de GitHub:

- **Repositorios:** Lugar donde se almacena el código del proyecto.
- **Colaboración:** Facilita el trabajo en equipo mediante pull requests y revisiones de código.
- **Gestión de proyectos:** Herramientas para organizar y planificar el trabajo.
- **GitHub Actions:** Automatización de flujos de trabajo y CI/CD.

Comandos principales de Git

Configuración inicial

```
git config --global user.name "Tu Nombre"
git config --global user.email "tuemail@ejemplo.com"
```

Crear un nuevo repositorio

```
git init
```

Clonar un repositorio existente

```
git clone https://github.com/usuario/repositorio.git
```

Verificar el estado del repositorio

```
git status
```

Añadir archivos al área de preparación (staging)

```
git add archivo.txt
# O para añadir todos los archivos modificados
git add .
```

Confirmar los cambios en el repositorio local

```
git commit -m "Mensaje del commit"
```

Ver el historial de commits

```
git log
```

Enviar cambios al repositorio remoto

```
git push origin main
```

Obtener los cambios del repositorio remoto

```
git pull origin main
```

Crear y cambiar a una nueva rama

```
git checkout -b nueva-rama
```

Cambiar a una rama existente

```
git checkout nombre-rama
```

Fusionar una rama con otra

```
git merge nombre-rama
```

Resolver conflictos de fusión

```
# Editar los archivos con conflictos y luego
git add archivo-resuelto.txt
git commit -m "Resolver conflictos de fusión"
```

Eliminar una rama

```
git branch -d nombre-rama
```

¿Qué es NPM?

NPM (Node Package Manager) es el gestor de paquetes por defecto para Node.js, una plataforma de JavaScript utilizada principalmente para construir aplicaciones de servidor y herramientas de desarrollo. NPM facilita la gestión de dependencias y permite a los desarrolladores compartir y reutilizar código de manera eficiente.

Funcionalidades de NPM

Gestión de Paquetes:

- Permite instalar, actualizar y desinstalar paquetes (bibliotecas y herramientas) de manera sencilla.
- Administra las versiones de los paquetes para asegurar la compatibilidad del proyecto.

Registro de Paquetes:

- NPM cuenta con un registro (registro de NPM) donde los desarrolladores pueden publicar sus paquetes y herramientas para que otros los utilicen.
- El registro de NPM alberga miles de paquetes de código abierto disponibles para la comunidad.

Scripts de NPM

- Permite definir y ejecutar scripts personalizados para automatizar tareas comunes del desarrollo, como pruebas, compilaciones y despliegues.

Comandos principales de NPM

Inicialización de un proyecto

```
npm init
```

Este comando inicializa un nuevo proyecto Node.js creando un archivo package.json, donde se almacenan las dependencias del proyecto y otra información relevante.

Instalación de paquetes

```
npm install nombre-paquete
```

Instala el paquete especificado y lo añade a las dependencias del proyecto en package.json.

```
npm install
```

Instala todas las dependencias listadas en package.json.

Instalación de paquetes de manera global

```
npm install -g nombre-paquete
```

Instala el paquete especificado de manera global en el sistema, haciéndolo accesible desde cualquier lugar en la línea de comandos.

Desinstalación de paquetes

```
npm uninstall nombre-paquete
```

Desinstala el paquete especificado y lo elimina de las dependencias en package.json.

Actualización de paquetes

```
npm update nombre-paquete
```

Actualiza el paquete especificado a la última versión compatible.

Listado de paquetes instalados

```
npm list
```

Muestra una lista de todos los paquetes instalados en el proyecto y sus versiones.

Ejecución de scripts definidos en package.json

```
NPM run nombre-script
```

Ejecuta un script definido en la sección scripts del archivo package.json. Por ejemplo, si package.json tiene un script definido como "start": "node app.js", puedes ejecutarlo con:

```
npm run start
```

¿Qué es un package.json?

El archivo package.json es un archivo de configuración clave en proyectos Node.js que contiene información esencial sobre el proyecto. Este archivo define las dependencias del proyecto, scripts de ejecución, metadatos del proyecto (como el nombre y la versión), y otra información relevante para la gestión del proyecto.

Funciones principales del package.json

Identificación del Proyecto:

Define el nombre, versión, descripción, autor y otros metadatos del proyecto.

Dependencias:

Lista las bibliotecas y herramientas que el proyecto necesita para funcionar correctamente.

Scripts:

Define comandos personalizados que se pueden ejecutar con npm run.

Configuración del Proyecto:

Especifica configuraciones adicionales, como motores de Node.js requeridos, configuración de linter, etc.

Cómo crear un package.json desde cero

Paso 1: Inicializar un proyecto con NPM

Para crear un package.json desde cero, utiliza el siguiente comando en la terminal:

```
npm init
```

Este comando te guiará a través de una serie de preguntas para configurar tu **package.json**. También puedes usar **npm init -y** para aceptar las configuraciones predeterminadas automáticamente.

Paso 2: Configuración del package.json

Durante el proceso de inicialización, se te pedirá que proporciones la siguiente información:

- Nombre del paquete: (nombre predeterminado): El nombre del proyecto.
- Versión: (1.0.0): La versión inicial del proyecto.
- Descripción: Una breve descripción del proyecto.
- Punto de entrada: El archivo principal del proyecto (index.js).
- Comando de prueba: El comando para ejecutar las pruebas del proyecto.
- Repositorio: La URL del repositorio del proyecto.
- Palabras clave: Palabras clave que describen el proyecto.
- Autor: El autor del proyecto.
- Licencia: (ISC): La licencia del proyecto.

Ejemplo de package.json

```
{
  "name": "mi-proyecto",
  "version": "1.0.0",
  "description": "Este es un ejemplo de proyecto Node.js",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no se especificaron pruebas\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/usuario/mi-proyecto.git"
  },
  "keywords": [
    "node",
    "npm",
    "ejemplo"
  ],
  "author": "Tu Nombre",
  "license": "ISC",
}
```

Crear package.json manualmente

También se puede crear el archivo package.json manualmente con el siguiente contenido básico y luego agregar configuraciones adicionales según sea necesario:

```
{
  "name": "mi-proyecto",
  "version": "1.0.0",
  "description": "Este es un ejemplo de proyecto Node.js",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no se especificaron pruebas\" && exit 1"
  },
  "author": "Tu Nombre",
  "license": "ISC"
}
```

¿Qué es un archivo .env?

Un archivo .env es un archivo de configuración que se utiliza para definir variables de entorno en proyectos de software. Este archivo es especialmente útil para almacenar configuraciones y credenciales sensibles, como claves de API, configuraciones de base de datos, y otras variables de entorno que no se deben almacenar directamente en el código fuente.

Características del archivo .env:

- Almacenamiento seguro: Permite mantener las credenciales y configuraciones fuera del código fuente, reduciendo el riesgo de exponer información sensible.
- Facilidad de configuración: Simplifica la gestión de diferentes configuraciones para distintos entornos (desarrollo, pruebas, producción).
- Compatibilidad: Es compatible con muchas herramientas y frameworks, especialmente en el ecosistema de Node.js mediante la biblioteca dotenv.

Ejemplo de archivo .env:

```
# Configuración de la base de datos
DB_HOST=localhost
DB_USER=root
DB_PASS=secret

# Configuración del servidor
PORT=3000
DEBUG=true

# Claves de API
API_KEY=your_api_key_here
```

Uso del archivo .env en Node.js con dotenv

Para utilizar un archivo .env en un proyecto Node.js, se puede usar la biblioteca dotenv. A continuación se detallan los pasos para configurar y usar dotenv.

Paso 1: Instalar dotenv

Primero, instalar la biblioteca dotenv utilizando NPM:

```
npm install dotenv
```

Paso 2: Crear un archivo .env

Crear un archivo .env en la raíz de tu proyecto y definiendo las variables de entorno.

Paso 3: Cargar las variables de entorno en la aplicación

En el archivo principal (por ejemplo, index.js), cargar las variables de entorno al inicio del archivo:

```
require('dotenv').config();

// Acceder a las variables de entorno
const port = process.env.PORT || 3000;
const dbHost = process.env.DB_HOST;
const dbUser = process.env.DB_USER;
const dbPass = process.env.DB_PASS;
const apiKey = process.env.API_KEY;
```

4: Añadir .env al archivo .gitignore

Para asegurarte de que el archivo .env no se suba al control de versiones (por ejemplo, GitHub), añade .env a tu archivo .gitignore:

```
# .gitignore
node_modules/
.env
```

¿Qué es un README?

Un archivo README es un documento que suele acompañar el código fuente de un proyecto y proporciona información esencial sobre el proyecto. Este archivo generalmente está escrito en formato Markdown (.md) y es una de las primeras cosas que los usuarios o desarrolladores verán cuando exploren tu repositorio. El archivo README.md ayuda a los usuarios a entender el propósito del proyecto, cómo configurarlo, usarlo y contribuir a él.

Contenido típico de un archivo README

Un archivo README.md bien estructurado puede contener varias secciones, dependiendo de la naturaleza y complejidad del proyecto. Algunas de las secciones más comunes que se incluyen en un archivo README.md son:

- **Título:** El nombre del proyecto.
- **Descripción:** Una breve descripción del proyecto y su propósito.
- **Tabla de Contenidos:** Opcional, pero útil para proyectos largos. Facilita la navegación.
- **Instalación:** Instrucciones sobre cómo instalar y configurar el proyecto.
- **Uso:** Ejemplos y explicaciones sobre cómo utilizar el proyecto.
- **Contribución:** Guía sobre cómo los otros desarrolladores pueden contribuir al proyecto.
- **Licencia:** Información sobre la licencia bajo la cual se distribuye el proyecto.
- **Autores y Reconocimientos:** Información sobre los autores del proyecto y cualquier agradecimiento a colaboradores o recursos utilizados.

Ejemplo de un archivo README.md

```
# Mi Proyecto

Una breve descripción de lo que hace mi proyecto.

## Tabla de Contenidos

- [Instalación](#instalación)
- [Uso](#uso)
- [Contribución](#contribución)
- [Licencia](#licencia)
- [Autores](#autores)

## Instalación

Estas son las instrucciones para instalar y configurar mi proyecto.

git clone https://github.com/usuario/mi-proyecto.git
cd mi-proyecto
npm install
```

Consejos para un buen README

- Claridad: Asegurarse de que las instrucciones sean claras y fáciles de seguir.
- Ejemplos: Proporcionar ejemplos prácticos para mostrar cómo usar el proyecto.
- Mantenimiento: Mantener el README.md actualizado con cualquier cambio importante en el proyecto.
- Formato: Usar encabezados, listas, enlaces y código formateado para mejorar la legibilidad.

Importancia de un buen README

- Primera impresión: Es la primera impresión que los usuarios y desarrolladores tienen de tu proyecto.
- Facilita la adopción: Ayuda a los nuevos usuarios a entender rápidamente cómo utilizar y contribuir al proyecto.
- Documentación: Sirve como documentación principal del proyecto, explicando su propósito, uso y cómo configurarlo.

Crear un README.md con Markdown

Markdown es un lenguaje de marcado ligero que se utiliza para formatear texto. Es muy popular en la creación de archivos README.md para proyectos. Algunas de las sintaxis más comunes en Markdown:

Títulos

Se utiliza el símbolo # para crear títulos. El número de # determina el nivel del título.

```
# Título de Nivel 1
## Título de Nivel 2
### Título de Nivel 3
#### Título de Nivel 4
##### Título de Nivel 5
###### Título de Nivel 6
```

Texto en Negrita y Cursiva

Negrita: Se rodea el texto con dos asteriscos ** o dos guiones bajos __.

Cursiva: Se rodea el texto con un asterisco * o un guion bajo _.

Negrita y Cursiva: Se usa tres asteriscos *** o tres guiones bajos ___.

```
**Texto en negrita**
__Texto en negrita__

*Texto en cursiva*
_Texto en cursiva_

***Texto en negrita y cursiva***
___Texto en negrita y cursiva___
```

Listas

Listas desordenadas: Se utiliza -, * o +.

Listas ordenadas: Se utiliza números seguidos de un punto 1..

```
- Elemento de lista desordenada
* Elemento de lista desordenada
+ Elemento de lista desordenada

1. Elemento de lista ordenada
2. Otro elemento de lista ordenada
3. Y otro más
```

Links

Link en línea: [texto del enlace](URL)

Link a un encabezado en el mismo documento: [texto del enlace](#encabezado)

```
[Google](https://www.google.com)
[Ir a la sección de uso](#uso)
```

Imágenes

```
![Texto alternativo](URL-de-la-imagen)
```

Citas

Se utiliza el símbolo > para crear citas.

```
> Esta es una cita en Markdown.
```

Código

Código en línea: Se rodea el texto con una comilla simple invertida ` .

Bloques de código: Se rodea el texto con tres comillas simples invertidas ``` .

```
markdown
`Código en línea`

\```
Bloque de código
Con múltiples líneas
```\
```

### Tablas

Se utiliza | para separar las columnas y - para separar el encabezado de la tabla del cuerpo.

```
Encabezado 1	Encabezado 2
Fila 1, Col 1	Fila 1, Col 2
Fila 2, Col 1	Fila 2, Col 2
```
```

Línea Horizontal

Se utiliza tres o más guiones, asteriscos o guiones bajos.

```
---

***

___
```

Módulos en Node.js: CommonJS vs ES Modules (ESM)

En Node.js, los módulos son piezas de código que se pueden importar y exportar entre archivos para organizar y reutilizar el código de manera eficiente. Existen dos sistemas principales de módulos en Node.js: CommonJS y ES Modules (ESM).

CommonJS

CommonJS es el sistema de módulos original en Node.js. Los módulos CommonJS utilizan la función require para importar dependencias y module.exports o exports para exportar módulos.

Exportaciones en CommonJS

Exportar un solo valor o función

```
function add(a, b) {
     return a + b;
   }

   module.exports = add;
```

Exportar un objeto

```
function add(a, b) {
     return a + b;
   }
```

```
function subtract(a, b) {
     return a - b;
   }

   module.exports = { add, subtract };
```

Exportar múltiples valores o funciones

```
function add(a, b) {
     return a + b;
   }

   function subtract(a, b) {
     return a - b;
   }

   exports.add = add;
   exports.subtract = subtract;
```

Importaciones en CommonJS

Importar un solo valor o función

```
const add = require('./math');

   console.log(add(2, 3)); // 5
```

Importar un objeto

```
const math = require('./math');

   console.log(math.add(2, 3)); // 5
   console.log(math.subtract(5, 2)); // 3
```

Desestructuración de importaciones

```
const { add, subtract } = require('./math');

   console.log(add(2, 3)); // 5
   console.log(subtract(5, 2)); // 3
```

ES Modules (ESM)

ES Modules es el sistema de módulos estandarizado en ECMAScript (ES6 y posteriores). Utiliza la sintaxis import y export para manejar módulos. Node.js ha soportado ESM de manera completa desde la versión 12.

Configuración de ES Modules en Node.js

Para usar ES Modules en Node.js, debe asegurarse de que Node.js sepa que se está usando módulos ESM.

- **Usar la extensión .mjs:** Los archivos con extensión .mjs se tratan automáticamente como ES Modules.
- **Configurar package.json:** Se puede especificar el tipo de módulo en el archivo package.json agregando "type": "module".

```
{
    "name": "mi-proyecto",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "type": "module",
    "scripts": {
      "start": "node app.mjs"
    },
    "author": "",
    "license": "ISC"
  }
```

Exportaciones en ES Modules

Exportar un solo valor o función por defecto

```
export default function add(a, b) {
     return a + b;
   }
```

Exportar múltiples valores o funciones

```
export function add(a, b) {
     return a + b;
   }

export function subtract(a, b) {
     return a - b;
   }
```

Exportar un objeto

```
const math = {
     add(a, b) {
       return a + b;
     },
     subtract(a, b) {
       return a - b;
     }
   };

   export default math;
```

Importaciones en ES Modules

Importar un valor o función por defecto

```
import add from './math.mjs';

   console.log(add(2, 3)); // 5
```

Importar múltiples valores o funciones

```
import { add, subtract } from './math.mjs';

   console.log(add(2, 3)); // 5
   console.log(subtract(5, 2)); // 3
```

Importar todo el módulo como un objeto

```
import * as math from './math.mjs';

   console.log(math.add(2, 3)); // 5
   console.log(math.subtract(5, 2)); // 3
```

Diferencias clave entre CommonJS y ESM

Sintaxis:

- **CommonJS:** Utiliza require para importar y module.exports o exports para exportar.
- **ESM:** Utiliza import y export.

Ejecución:

- **CommonJS:** Los módulos son evaluados de manera síncrona.

- **ESM:** Los módulos pueden ser evaluados de manera asíncrona, lo que permite importaciones dinámicas.

Cache:

- Ambos sistemas almacenan en caché los módulos una vez que se importan, pero ESM tiene un comportamiento más estrictamente definido en cuanto a la resolución y la importación de módulos.

Compatibilidad:

- **CommonJS:** Es el sistema de módulos original de Node.js y está ampliamente soportado.
- **ESM:** Es el estándar moderno y tiene soporte completo en Node.js desde la versión 12, con mejor interoperabilidad con el ecosistema JavaScript en general.

Consejos para elegir entre CommonJS y ESM

- **Proyectos nuevos:** Si se está comenzando un nuevo proyecto, es recomendable usar ESM por ser el estándar moderno y tener mejor compatibilidad con el ecosistema JavaScript.
- **Proyectos existentes:** Si se está trabajando en un proyecto que ya usa CommonJS, puede ser más sencillo continuar usando CommonJS para evitar incompatibilidades y refactorizaciones innecesarias.
- **Interoperabilidad:** Si se necesita interoperabilidad con librerías que usan uno u otro sistema de módulos, se puede usar herramientas como Babel para transpilar el código y asegurar compatibilidad.

Cómo Combinar CommonJS y ES Modules en un Proyecto Node.js

En algunos casos, puede ser necesario combinar módulos CommonJS y ES Modules (ESM) en el mismo proyecto. A continuación se muestra cómo se puede hacerlo de manera efectiva utilizando herramientas y configuraciones adecuadas.

Importar Módulos CommonJS en Módulos ES Modules

Para importar módulos CommonJS en un archivo ES Module, se utiliza la función import con la sintaxis adecuada. Node.js permite esta interoperabilidad de manera nativa.

Archivo funciones.cjs (CommonJS)

```
function add(a, b) {
    return a + b;
  }

  function subtract(a, b) {
    return a - b;
  }

  module.exports = { add, subtract };
```

Archivo app.mjs (ES Modules)

```
import math from './math.cjs';

console.log(math.add(2, 3)); // 5
console.log(math.subtract(5, 2)); // 3
```

Importar Módulos ES Modules en Módulos CommonJS

Para importar módulos ES Modules en un archivo CommonJS, se utiliza import con la sintaxis dinámica import(). Esto devuelve una promesa que resuelve el módulo importado.

Archivo funciones.mjs (ES Modules)

```
export function add(a, b) {
    return a + b;
  }

export function subtract(a, b) {
    return a - b;
  }
```

Archivo app.cjs (CommonJS)

```
(async () => {
    const math = await import('./math.mjs');

    console.log(math.add(2, 3)); // 5
    console.log(math.subtract(5, 2)); // 3
  })();
```