Implementación del Patrón Creacional Factory Method en Proyecto Node.js

1. Objetivo El objetivo de esta implementación es utilizar el patrón creacional **Factory Method** para generar instancias de objetos encargados de obtener información de diferentes APIs de criptomonedas. Este patrón permite que el sistema sea **extensible**, ya que se pueden añadir nuevas fuentes de datos o monedas sin necesidad de modificar el código existente en los endpoints, manteniendo la arquitectura limpia y fácil de mantener.

---

1. Estructura de Archivos Se organizó el proyecto de la siguiente manera para mantener claridad y separación de responsabilidades:

```
backend/
│
├─ index.js                  # Punto de entrada del servidor Express
├─ routes/
│  └─ cryptoRoutes.js        # Contiene la definición de los endpoints
relacionados con criptomonedas
├─ fetchers/
│  ├─ CryptoFetcherFactory.js # Fábrica que decide qué fetcher crear según la
fuente de datos
│  └─ CoinMarketCapFetcher.js # Implementación específica para obtener datos
desde CoinMarketCap
├─ package.json              # Registro de dependencias y scripts del
proyecto
└─ .env                      # Variables de entorno para almacenar claves de
API de forma segura
```

---

1. Implementación Paso a Paso

3.1. CoinMarketCapFetcher.js

Este archivo contiene la clase que se encarga de conectarse a la API de CoinMarketCap y obtener los datos de criptomonedas. Se utiliza `node-fetch` para realizar las solicitudes HTTP y `dotenv` para manejar la API key de forma segura.

```
// CoinMarketCapFetcher.js
const fetch = require("node-fetch"); // Versión 2 compatible con require()
require('dotenv').config(); // Carga variables de entorno

class CoinMarketCapFetcher {
    async getCryptoData(symbol) {
        // Se construye la URL dinámica según el símbolo de la criptomoneda
        const response = await fetch(
            `https://pro-api.coinmarketcap.com/v1/cryptocurrency/quotes/
latest?symbol=${symbol}`,
            {
```

```
                headers: {
                    "X-CMC_PRO_API_KEY": process.env.CMC_API_KEY
                }
            }
        );
        const data = await response.json(); // Convierte la respuesta a JSON
        return data; // Retorna la información obtenida
    }
}

module.exports = CoinMarketCapFetcher; // Exporta la clase para que la
fábrica pueda usarla
```

Explicación: Cada vez que se necesite obtener información de CoinMarketCap, se crea una instancia de esta clase, lo que permite que la lógica de obtención de datos esté encapsulada y separada de la lógica del servidor.

3.2. CryptoFetcherFactory.js

La fábrica es responsable de crear la instancia correcta de un fetcher según la fuente de datos que se especifique. Esto permite que, si en el futuro se agregan más fuentes de datos, no se tenga que cambiar la lógica de los endpoints.

```
// CryptoFetcherFactory.js
const CoinMarketCapFetcher = require("./CoinMarketCapFetcher");

class CryptoFetcherFactory {
    static getFetcher(source) {
        switch (source) {
            case "CMC":
                return new CoinMarketCapFetcher();
            // Se pueden agregar más fetchers en el futuro
            default:
                throw new Error("Fuente no soportada");
        }
    }
}

module.exports = CryptoFetcherFactory;
```

Explicación: La fábrica actúa como un punto único de creación de objetos. Esto es útil porque los endpoints no necesitan conocer la implementación interna de cada fetcher; solo solicitan uno por su nombre.

3.3. cryptoRoutes.js

Definición del endpoint que utiliza la fábrica para obtener datos de criptomonedas.

```
// cryptoRoutes.js
const express = require("express");
const router = express.Router();
const CryptoFetcherFactory = require("../fetchers/CryptoFetcherFactory");

router.get("/crypto/:symbol", async (req, res) => {
    try {
        // La fábrica devuelve la instancia del fetcher correspondiente
        const fetcher = CryptoFetcherFactory.getFetcher("CMC");
        const data = await fetcher.getCryptoData(req.params.symbol); //
Obtiene los datos de la API
        res.json(data); // Envía la respuesta al cliente
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

module.exports = router;
```

Explicación: El endpoint es independiente de la implementación de la API. Solo solicita al fetcher los datos y los devuelve al cliente, logrando desacoplar la lógica de negocio de la infraestructura del servidor.

3.4. index.js

Archivo principal que inicializa el servidor Express y registra los endpoints.

```
// index.js
const express = require("express");
const app = express();
const cryptoRoutes = require("./routes/cryptoRoutes");

app.use("/api", cryptoRoutes); // Todos los endpoints de criptomonedas
estarán bajo /api

app.listen(3000, () => {
    console.log("Servidor corriendo en http://localhost:3000");
});
```

Explicación: Aquí se configura Express, se registran las rutas y se levanta el servidor en el puerto 3000.

---

1. Notas Importantes

2. **Dependencias necesarias:**

```
npm install express node-fetch@2 dotenv
```

3. **Archivo .env:**

```
CMC_API_KEY=TU_API_KEY_DE_COINMARKETCAP
```

   Esto permite mantener la API key segura y no exponerla en el código.

4. **Node-fetch:** Se utilizó la versión 2 para compatibilidad con `require()` en Node.js 20, evitando errores relacionados con módulos ES.

5. **Extensibilidad:** Se pueden agregar más fetchers para otras APIs simplemente creando nuevas clases en la carpeta `fetchers` y agregándolas en la fábrica.

6. **Ventajas del Factory Method:**

7. Desacopla la creación de objetos de su uso.
8. Facilita la extensión sin modificar código existente.
9. Mejora el mantenimiento y la organización del proyecto.
10. Permite centralizar la lógica de creación y configuración de fetchers.