**guiaProyecto  
  
🧭 Guía paso a paso extendida para construir caminando-online**

1. **📌 Definición del propósito y alcance**
   1. Establecer qué problema resuelve el proyecto
   2. Identificar el público objetivo
   3. Definir funcionalidades principales (comparación de precios, pagos, etc.)
2. **🧱 Diseño de la arquitectura del sistema**
   1. Elegir tecnologías (Node.js, Express, HTML/CSS/JS, etc.)
   2. Definir estructura de carpetas (ver regla estructuraCarpetas)
   3. Planificar separación entre frontend, backend, scraping y pagos
3. **📁 Inicialización del entorno de desarrollo**
   1. Crear carpeta raíz del proyecto
   2. Inicializar repositorio Git
   3. Configurar .gitignore
   4. Crear estructura de carpetas base
   5. Crear archivo README.md en carpeta docs/
   6. Crear archivo settings.js en carpeta config/
4. **⚙️ Configuración del entorno técnico**
   1. Instalar dependencias (Node.js, Express, etc.)
   2. Configurar entorno local (puertos, variables de entorno)
   3. Crear archivo settings.js con parámetros del sistema
5. **🌐 Desarrollo del frontend básico**
   1. Crear index.html con estructura inicial
   2. Crear styles.css para estilos generales
   3. Crear main.js para lógica principal del sitio
6. **🧩 Implementación de funcionalidades del frontend**
   1. Menú de productos (productoMenu.js)
   2. Agregar productos (agregarProducto.js)
   3. Comparar precios (comparar.js)
   4. Helpers reutilizables (helpers.js)
7. **🕷️ Desarrollo de scripts de scraping**
   1. Crear scripts para cada supermercado
   2. Obtener nombre, precio y unidad de productos
   3. Guardar resultados en archivos temporales
8. **🔙 Desarrollo del backend**
   1. Crear servidor con Express (server.js)
   2. Configurar rutas (routes/comparar.js)
   3. Conectar frontend con backend vía API
9. **🗃️ Manejo de datos**
   1. Crear productos.json con productos simulados
   2. Crear resultados.json para almacenar resultados
   3. Validar y actualizar datos desde el backend
10. **💳 Integración de pagos**
    1. Configurar APIs de MercadoPago, MODO, Belo
    2. Crear scripts para generar links de pago
    3. Conectar pagos con lógica de ahorro y resultados
11. **🧪 Pruebas del sistema**
    1. Probar scraping con productos reales
    2. Validar comparación de precios
    3. Simular pagos y verificar flujo completo
12. **🧼 Optimización y refactorización**
    1. Mejorar rendimiento de scraping
    2. Modularizar código del frontend y backend
    3. Eliminar redundancias y aplicar buenas prácticas
13. **🛡️ Seguridad y validaciones**
    1. Validar entradas del usuario
    2. Proteger rutas del backend
    3. Configurar CORS y sanitización de datos
14. **📄 Documentación del proyecto**
    1. Crear README.md con guía de uso
    2. Documentar funciones clave y estructura
    3. Incluir ejemplos de instalación y ejecución
15. **🚀 Despliegue del sistema**
    1. Elegir plataforma de hosting (Render, Vercel, etc.)
    2. Configurar entorno de producción
    3. Subir proyecto y probar en vivo
16. **📈 Monitoreo y mantenimiento**
    1. Configurar logs y alertas
    2. Actualizar productos y precios periódicamente

**Mejorar funcionalidades según feedbackShortcuts que pedi que guarde a Copilot:**

* **“Objetivo y Desarrollo Inicial del proyecto”: objetivoInicialProyecto**

"Estoy creando un sitio web que se llama "Caminando.Online" para que el usuario pueda elegir sus productos de la lista de las 5 cadenas de supermercados mas importantes de Argentina: Carrefour, Coto, Jumbo, Dia y Vea. El sistema a traves de scraping de los productos le compara en que cadena de supermercados puede obtener el mejor precio por cada producto elegido y lo presenta en forma de tabla. Ademas, la plataforma quiero que cuente con una opcion en forma de columna, a la derecha, destacada, con el nombre "Caminando Online" en la que junte la mejor opcion de cada producto. Al final de la tabla tiene que estar sumado el total del costo de todos los productos seleccionados por cada supermercado y en el de "Caminando Online" debe mostrarse cuanto se ahorra con respecto al promedio de los otros supermercados. Debajo de eso, tiene que haber un boton para cada supermercado que diga "comprar" y tambien uno debajo de la columna de "Caminando Online". Si el usuario hace click en el "comprar" de algun supermercado, la plataforma tiene que disparar la orden de compra al sitio web de ese supermercado especifico, en cambio, si el usuario hace click en el boton de "Comprar" de la columna de "Caminando Online" la plataforma debe disparar la compra a cada supermercado cuyo producto haya sido incluido en la lista por tener el mejor precio. Si el usuario utiliza esta Ademas, quisiera que se consideren todas las opciones de promociones y descuentos que ofrecen cada una de las cadenas de supermercados para seleccionar el precio mas barato. Si el usuario elige esta opcion y el descuento con el promedio de supermercados es mayor a $10.000 (diez mil pesos argentinos) entonces la plataforma debe realizar un cargo a su tarjeta como pago por el servicio de $2.500 (dos mil quinientos pesos argentinos) No estoy seguro de si la mejor opcion es que el usuario se cree una cuenta y cargue los datos de su cuenta de supermercado, o si es posible conectarse directamente con cada web de cada supermercado. (Necesito entender mejor como desarrollar esta parte y cual seria el flow mas simplificado de cara al usuario) Ademas, se tiene que integrar un procesador de pago para poder realizar las compras con tarjeta de credito, por ejemplo: MODO, MercadoPago o Belo. La plataforma no debe almacenar datos de tarjetas de credito."

* **guiaProyecto**: Hace referencia a la estructura de la hoja 1
* **paginasComparar**

no, antes de eso quiero que recuerdes que "las UNICAS PAGINAS" con las que vamos a trabajar "para comparar" son: Carrefour: https://www.carrefour.com.ar/ Disco: https://www.disco.com.ar/ Vea: https://www.vea.com.ar/ Jumbo: https://www.jumbo.com.ar/ Dia: https://diaonline.supermercadosdia.com.ar/ Recuerda: No quiero me sugieras, ni menciones, ni analices ni tomes como referencia Airbnb, ni booking, ni zonaprop, ni nada de otras paginas que no sean las que te mencione. Recuerda este mensaje bajo el nombre "Paginas a comparar" para que cuando yo te diga que recuerdes cuales son las Paginas a Comparar vuelvas a leer, analizar y entender este mensaje.

**ContextoJuan:**Ahora quiero que recuerdes el contexto de quien soy yo: Me llamo Juan. Yo no se programar, tengo un conocimiento extremadamente limitado de programacion, puedo resolver algunas cosas basicas de github y sublime text pero no mucho mas. Tampoco se como se estructura desde el lado de la programacion el proceso para encarar este proyecto, o sea, si se empieza por el index, si se empieza por el scraping, por que seccion, eso no lo se. Todo esto lo voy a estar armando con vos y por lo tanto necesito que actues como el mejor programador senior, me enseñes y me ayudes a entender como hacer el paso a paso de este proyecto desde el minuto 0.

**estructuraCarpetas:**

caminando-online/

├── assets/ # (opcional) si se usan en backend o scraping

│ ├── fonts/

│ └── img/

├── backend/ # Lógica del servidor y API

│ ├── server.js

│ ├── config/

│ │ ├── db.js

│ │ └── settings.js

│ ├── controllers/

│ │ ├── productController.js

│ │ └── userController.js

│ ├── middlewares/

│ │ └── authMiddleware.js

│ ├── models/

│ │ ├── Product.js

│ │ └── user.js

│ ├── routes/

│ │ ├── comparar.js

│ │ ├── products.js

│ │ └── users.js

│ └── validators/

│ └── productValidator.js

├── config/ # Configuraciones generales del proyecto (pendiente de uso)

├── data/ # Datos simulados o temporales

│ ├── db/

│ ├── productos.json

│ └── resultados.json

├── docs/ # Documentación del proyecto

├── env/ # Archivos relacionados al entorno

├── frontend/ # Todo lo relacionado al cliente

│ ├── index.html # Página principal del sitio

│ ├── css/

│ │ └── styles.css # Hoja principal de estilos

│ ├── js/

│ │ ├── agregarProducto.js

│ │ ├── comparar.js

│ │ ├── helpers.js

│ │ ├── main.js

│ │ └── productoMenu.js

│ └── assets/ # Logos, íconos, ilustraciones, fuentes

│ ├── fonts/

│ └── img/

├── node\_modules/

├── pagos/

│ ├── belo.js

│ ├── mercadoPago.js

│ └── modo.js

├── scraping/

│ ├── carrefour.js

│ ├── dia.js

│ ├── disco.js

│ ├── jumbo.js

│ └── vea.js

├── .env

├── .gitignore

├── package.json

├── package-lock.json

└── requirements.txt

**📌 Paso 1A – ¿Qué problema resuelve caminando-online?**

**Problema identificado:** En Argentina, los precios de productos básicos varían significativamente entre supermercados, y los consumidores no tienen una forma sencilla y rápida de comparar precios antes de hacer sus compras. Esta falta de transparencia genera:

* 🛒 Gasto innecesario por desconocimiento de mejores precios
* ⏱️ Pérdida de tiempo al visitar múltiples sitios o locales
* 📉 Dificultad para planificar compras eficientes y ahorrar

**Solución propuesta por caminando-online:** Crear una plataforma web que permita a los usuarios:

* Comparar precios de productos entre distintos supermercados en tiempo real
* Visualizar cuánto pueden ahorrar eligiendo el supermercado más conveniente
* Acceder a links de pago directo para comprar desde la plataforma
* Recibir recomendaciones personalizadas según sus hábitos de compra

**👥 Paso 1B – Público objetivo de caminando-online**

**Segmentos principales:**

1. 🧍‍♂️ **Consumidores urbanos con hábitos digitales**
   * Personas que ya compran online o comparan precios por su cuenta
   * Valoran la eficiencia, el ahorro y la comodidad
   * Suelen usar MercadoPago, MODO o apps bancarias
2. 👨‍👩‍👧‍👦 **Familias que hacen compras mensuales o semanales**
   * Buscan optimizar su presupuesto
   * Están dispuestas a cambiar de supermercado si el ahorro lo justifica
   * Pueden beneficiarse de recomendaciones personalizadas
3. 🎓 **Estudiantes y jóvenes con presupuesto limitado**
   * Compran productos básicos y buscan precios bajos
   * Son nativos digitales, usan apps y redes sociales
   * Pueden viralizar la herramienta si les resulta útil
4. 🧓 **Adultos mayores con asistencia digital**
   * Reciben ayuda de familiares para hacer compras online
   * Pueden beneficiarse si la interfaz es clara y accesible
   * Representan un segmento creciente en el comercio electrónico

**Características comunes:**

* Viven en zonas urbanas o semiurbanas con acceso a supermercados grandes
* Tienen acceso a internet y dispositivos móviles
* Están motivados por el ahorro y la simplicidad

**Área geográfica de enfoque inicial:**

* 🏙️ **Capital Federal (CABA)**
* 🌳 **Zona Norte del Gran Buenos Aires**
  + Incluye partidos como Vicente López, San Isidro, Tigre, San Fernando, y Pilar

**🧩 Paso 1C – Funcionalidades principales de caminando-online**

**1. 🛒 Comparación de precios entre supermercados**

* El usuario selecciona productos desde un menú o buscador
* El sistema muestra precios actualizados por supermercado
* Se calcula el ahorro potencial al elegir el supermercado más conveniente

**2. 💳 Generación de links de pago**

* Una vez elegidos los productos, el sistema genera links de pago por supermercado
* Integración con APIs de MercadoPago, MODO y Belo
* El usuario puede pagar directamente desde la plataforma

**3. 📦 Agregado y edición de productos**

* El usuario puede agregar productos personalizados al carrito
* Posibilidad de editar cantidades, unidades y preferencias
* Se guarda el historial de productos frecuentes

**4. 📊 Visualización de ahorro**

* Se muestra cuánto se ahorra en total y por producto
* Comparación gráfica entre supermercados
* Recomendaciones automáticas para maximizar el ahorro

**5. 👤 Creación de cuenta y conexión con cuentas de supermercado**

* El usuario puede registrarse y crear un perfil personal
* Se conecta con sus cuentas de supermercados (Carrefour, Coto, etc.) mediante login o token
* El sistema detecta descuentos, promociones o beneficios exclusivos que le corresponden
* Se ajustan los precios mostrados según sus beneficios personales

**6. 🔍 Búsqueda inteligente de productos**

* Buscador con autocompletado y sugerencias
* Filtros por categoría, marca, unidad, precio mínimo/máximo

**7. 🔄 Actualización periódica de precios**

* Scraping automatizado de supermercados seleccionados
* Actualización diaria o semanal de precios
* Validación de datos para evitar errores o inconsistencias

**8. 🧠 Recomendaciones personalizadas (futuro)**

* Algoritmo que aprende del historial de compras
* Sugiere productos y supermercados según hábitos del usuario

**🧱 Paso 2A – Elección de tecnologías**

**Frontend:**

* HTML5, CSS3, JavaScript (vanilla para empezar)
* Posible uso de librerías como Axios para llamadas a la API
* Framework futuro: React (si el proyecto escala)

**Backend:**

* Node.js con Express.js como framework principal
* API REST para comunicación con el frontend
* Middleware para validaciones, autenticación y manejo de errores

**Scraping:**

* Node.js con librerías como Puppeteer o Cheerio
* Scripts separados por supermercado
* Ejecución programada con cron jobs o servicios externos

**Base de datos (futuro):**

* Inicialmente archivos JSON para simular datos
* Escalable a MongoDB o PostgreSQL según necesidad

**Pagos:**

* Integración con APIs de MercadoPago, MODO y Belo
* Generación de links de pago desde el backend

**🗂️ Paso 2B – Estructura de carpetas (ver regla estructuraCarpetas)**

Vamos a definir esta estructura en el próximo paso, pero ya la mencionamos como parte del diseño. Tendrá separación clara entre:

* /frontend → HTML, CSS, JS del cliente
* /backend → servidor Express, rutas, lógica de negocio
* /scraping → scripts para extraer precios
* /data → archivos JSON simulados
* /payments → lógica de integración con APIs de pago

**🔄 Paso 2C – Separación de responsabilidades**

**Modularización clara:**

|  |  |
| --- | --- |
| **Módulo** | **Responsabilidad principal** |
| Frontend | Interfaz de usuario, interacción y visualización |
| Backend | API, lógica de negocio, validaciones |
| Scraping | Extracción de precios desde sitios externos |
| Pagos | Generación y gestión de links de pago |
| Datos | Almacenamiento temporal y simulación de productos |

**🧩 Punto 2C: Separación de responsabilidades**

**🎨 Frontend (index.html, css/, js/, assets/)**

* **Responsabilidad**: Interfaz de usuario y experiencia visual.
* **Tecnologías sugeridas**: HTML, CSS, JavaScript puro o con frameworks ligeros (como Alpine.js o Vue si querés escalar).
* **Funciones clave**:
  + Mostrar productos y precios
  + Interacción con el comparador
  + Menú de navegación y filtros
  + Llamadas al backend vía fetch o axios

**🧠 Backend (backend/)**

* **Responsabilidad**: Lógica del servidor, manejo de rutas, procesamiento de datos.
* **Tecnologías sugeridas**: Node.js con Express.
* **Funciones clave**:
  + Recibir solicitudes del frontend (comparar, agregar producto, etc.)
  + Coordinar scraping y devolver resultados
  + Validar datos y aplicar lógica de negocio
  + Servir archivos estáticos si no usás un servidor separado

**🕷️ Scraping (scraping/)**

* **Responsabilidad**: Obtener precios actualizados desde supermercados.
* **Tecnologías sugeridas**: Puppeteer o Cheerio con Node.js..
* **Funciones clave**:
  + Ejecutar scripts por supermercado
  + Extraer nombre, precio, unidad, y disponibilidad
  + Guardar resultados en data/resultados.json
  + Ser invocados por el backend o por cron jobs

**💳 Pagos (pagos/)**

* **Responsabilidad**: Integración con plataformas de pago.
* **Tecnologías sugeridas**: SDKs oficiales de Mercado Pago, MODO, Belo.
* **Funciones clave**:
  + Generar links de pago o QR
  + Validar transacciones
  + Mostrar resumen de compra y ahorro
  + Enviar confirmación (opcional)

**🔗 Comunicación entre módulos**

|  |  |  |
| --- | --- | --- |
| **Módulo origen** | **Módulo destino** | **Tipo de comunicación** |
| Frontend | Backend | HTTP (fetch/axios) |
| Backend | Scraping | Función interna o cron |
| Backend | Pagos | SDK/API externa |
| Scraping | Data | Escritura de JSON |
| Backend | Frontend | JSON (respuesta HTTP) |

**conexionesBackend  
🔌 Conexión entre Frontend y Backend**

**📍 Endpoints del Backend**

plaintext

GET /api/productos → Obtener lista de productos

POST /api/productos → Agregar nuevo producto

GET /api/comparar?nombre=... → Comparar precios de un producto

GET /api/supermercados → Listar supermercados disponibles

POST /api/pago → Iniciar proceso de pago

GET /api/configuracion → Obtener parámetros del sistema

**🧭 Uso desde el Frontend**

* main.js o comparar.js hace fetch('/api/comparar?nombre=yerba') para mostrar precios.
* agregarProducto.js envía POST con datos del nuevo producto.
* productoMenu.js carga la lista de productos con GET /api/productos.
* helpers.js puede centralizar llamadas a la API para reutilización.

**🔄 Flujo de Datos entre Módulos**

plaintext

[Usuario en Frontend]

↓

Busca "yerba" → Frontend hace GET /api/comparar?nombre=yerba

↓

[Backend recibe solicitud]

↓

Verifica si hay datos actualizados en data/resultados.json

↓

Si no hay → Invoca módulo scraping/yerba.js

↓

Scraping obtiene precios → Guarda en data/resultados.json

↓

Backend procesa y responde al Frontend con JSON

↓

Frontend muestra precios y ahorro potencial

↓

Usuario elige opción → Frontend hace POST /api/pago

↓

Backend usa pagos/mercadoPago.js → Genera link o QR

↓

Frontend muestra resumen y link de pago

**🧠 Bonus: Flujo para agregar productos**

plaintext

Usuario carga producto → Frontend hace POST /api/productos

↓

Backend guarda en productos.json

↓

Frontend actualiza menú de productos

Checkpoint script Carrefour:  
  
**from selenium import webdriver**

**from selenium.webdriver.firefox.options import Options**

**from selenium.webdriver.common.by import By**

**from selenium.webdriver.support.ui import WebDriverWait**

**from selenium.webdriver.support import expected\_conditions as EC**

**import time**

**IGNORAR = {"ofertas", "destacados"}**

**IGNORAR\_GRUPOS = {"gama de precios"}**

**def setup\_driver():**

**opts = Options()**

**return webdriver.Firefox(options=opts)**

**def main():**

**print("🚀 Iniciando script Carrefour")**

**driver = setup\_driver()**

**wait = WebDriverWait(driver, 30)**

**driver.get("https://www.carrefour.com.ar/")**

**print("🌐 Página cargada")**

**time.sleep(3)**

**try:**

**categorias\_btn = wait.until(EC.element\_to\_be\_clickable((By.XPATH, "//span[text()='Categorías']")))**

**driver.execute\_script("arguments[0].click();", categorias\_btn)**

**print("✅ Click en 'Categorías'")**

**except:**

**print("❌ No se encontró el botón 'Categorías'")**

**driver.quit()**

**return**

**try:**

**wait.until(EC.presence\_of\_element\_located((By.CSS\_SELECTOR, "ul.carrefourar-mega-menu-0-x-menuContainer")))**

**time.sleep(2)**

**items = driver.find\_elements(By.CSS\_SELECTOR, "ul.carrefourar-mega-menu-0-x-menuContainer > li")**

**print("📋 Categorías detectadas:")**

**for item in items:**

**try:**

**link = item.find\_element(By.TAG\_NAME, "a")**

**text = link.text.strip().lower()**

**href = link.get\_attribute("href")**

**print(f"— {text} → {href}")**

**if text in IGNORAR or not text or not href:**

**continue**

**print(f"✅ Primera categoría válida: {text}")**

**driver.execute\_script("arguments[0].click();", link)**

**break**

**except:**

**continue**

**else:**

**print("❌ No se encontró ninguna categoría válida")**

**driver.quit()**

**return**

**time.sleep(3)**

**try:**

**privacidad\_btn = driver.find\_element(By.CSS\_SELECTOR, "button#onetrust-accept-btn-handler")**

**driver.execute\_script("arguments[0].click();", privacidad\_btn)**

**print("✅ Aviso de privacidad cerrado")**

**time.sleep(1)**

**except:**

**pass**

**wait.until(lambda d: d.execute\_script("return document.readyState") == "complete")**

**print("⏳ Página completamente cargada")**

**filtro = wait.until(EC.presence\_of\_element\_located((By.CSS\_SELECTOR, ".vtex-flex-layout-0-x-flexCol--filterCol")))**

**print("📌 Contenedor de filtros detectado")**

**driver.execute\_script("""**

**const filtro = arguments[0];**

**const rect = filtro.getBoundingClientRect();**

**window.scrollBy({ top: rect.top - 100, behavior: 'instant' });**

**""", filtro)**

**time.sleep(1)**

**titulos\_raw = filtro.find\_elements(By.CSS\_SELECTOR, "div[class\*='filterTitle']")**

**print(f"🔍 {len(titulos\_raw)} grupos de filtros detectados")**

**grupo\_valido = 0**

**for i, titulo in enumerate(titulos\_raw):**

**nombre = titulo.text.strip()**

**if not nombre or nombre.lower() in IGNORAR\_GRUPOS:**

**continue**

**grupo\_valido += 1**

**print(f"— Grupo {grupo\_valido}: '{nombre}'")**

**try:**

**driver.execute\_script("arguments[0].scrollIntoView({behavior: 'smooth', block: 'center'});", titulo)**

**driver.execute\_script("arguments[0].click();", titulo)**

**print(f"✅ Grupo '{nombre}' expandido")**

**try:**

**ver\_mas = titulo.find\_element(By.XPATH, "./following::button[contains(@class,'seeMoreButton')]")**

**driver.execute\_script("arguments[0].click();", ver\_mas)**

**print(f"✅ Click en botón 'ver más' del grupo '{nombre}'")**

**except:**

**print(f"⚠️ No se encontró botón 'ver más' en grupo '{nombre}'")**

**except:**

**print(f"❌ No se pudo expandir grupo '{nombre}'")**

**except Exception as e:**

**print(f"❌ Error en ejecución: {e}")**

**with open("diagnostico.html", "w", encoding="utf-8") as f:**

**f.write(driver.page\_source)**

**driver.save\_screenshot("diagnostico.png")**

**input("Presiona Enter para cerrar...")**

**driver.quit()**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Jerarquia Base de datos**

**🛒 supermercados/**

**├── raw\_carrefour**

**│ └── filtros\_raw**

**├── raw\_jumbo**

**│ └── filtros\_raw**

**├── raw\_coto**

**│ └── filtros\_raw**

**├── raw\_vea**

**│ └── filtros\_raw**

**├── raw\_dia**

**│ └── filtros\_raw**

**├── normalized\_carrefour**

**│ └── filtros**

**├── normalized\_jumbo**

**│ └── filtros**

**├── normalized\_coto**

**│ └── filtros**

**├── normalized\_vea**

**│ └── filtros**

**├── normalized\_dia**

**│ └── filtros**

**├── metadata\_carrefour**

**│ └── menu\_categories**

**├── metadata\_jumbo**

**│ └── menu\_categories**

**├── metadata\_coto**

**│ └── menu\_categories**

**├── metadata\_vea**

**│ └── menu\_categories**

**├── metadata\_dia**

**│ └── menu\_categories**

**├── offers\_carrefour**

**│ └── ofertas**

**├── offers\_jumbo**

**│ └── ofertas**

**├── offers\_coto**

**│ └── ofertas**

**├── offers\_vea**

**│ └── ofertas**

**└── offers\_dia**

**└── ofertas**

**👤 administracion/**

**└── admin**

**├── usuarios**

**├── roles**

**└── logs**

**📊 auditoria\_analytics/**

**├── analytics**

**│ └── metrics**

**├── audit**

**│ └── historial**

**├── scheduler**

**│ └── tareas**

**└── logs**

**└── scraper\_logs**

**💰 precios\_productos/**

**├── pricing\_history**

**│ └── precios**

**└── products**

**└── catalogo**

**🔧 sistema\_configuracion/**

**├── alerts**

**│ └── notificaciones**

**├── sessions**

**│ └── tokens**

**└── config\_global**

**└── parametros**

**def store\_filters\_por\_grupo(data: dict, category\_name: str, collections):**

**normalized\_name = normalize\_name(category\_name)**

**timestamp = datetime.now(timezone.utc)**

**for group\_name, opciones in data.items():**

**doc = {**

**"category": category\_name,**

**"normalized": normalized\_name,**

**"group": group\_name,**

**"options": opciones,**

**"timestamp": timestamp**

**}**

**collections["raw"].replace\_one(**

**{"normalized": normalized\_name, "group": group\_name},**

**doc,**

**upsert=True**

**)**

**print(f"📂 filtros\_raw actualizado por grupo para categoría: {category\_name}")**

**Armemos en forma de bullet points el paso a paso del scraper:**

**- Entrar a la web home**

**- Abrir el menu de categorias**

**- Seleccionar la primer categoria**

**- Esperar a que cargue la pagina**

**- Identificar el menu de la izquierda**

**- Darle ver mas a todas las**

| **Campo** | **Descripción breve** | **Ejemplo (Coca-Cola 2.25 L)** | **Fuente HTML / JSON** |
| --- | --- | --- | --- |
| **SKU ID** | **Identificador único del SKU** | **14837** | **<div id="product-id-14837"> / window.\_\_STATE\_\_** |
| **Product ID** | **ID del producto padre** | **2000001** | **window.\_\_STATE\_\_** |
| **EAN** | **Código de barras** | **7790895000012** | **Especificaciones técnicas** |
| **referenceId** | **Referencia interna** | **COCA225LT** | **JSON embebido** |
| **Nombre del producto** | **Título oficial** | **Coca-Cola 2.25 L** | **<h1> / JSON** |
| **Descripción** | **Texto descriptivo corto** | **Gaseosa sabor original en botella PET** | **<div class="product-description">** |
| **Precio final** | **Precio con descuento aplicado** | **$4.400** | **<span class="sellingPriceValue">** |
| **Precio original** | **Precio antes de descuento** | **$5.000** | **<span class="listPriceValue">** |
| **Descuento** | **Porcentaje de ahorro** | **12 %** | **<span class="savingsPercentage">** |
| **Precio por unidad** | **Precio calculado por unidad de medida** | **$1.955,56 / L** | **<div class="custom\_specification\_wrapper">** |
| **Disponibilidad** | **Estado de stock** | **En stock** | **<span class="availability"> / JSON** |
| **Cantidad mínima** | **Mínimo pedible** | **1** | **<input min="1">** |
| **Imágenes** | **URLs de imagen principal y galería** | **https://…/coca-cola-225.jpg** | **<img src="…"> / JSON** |
| **Vídeos** | **URLs de videos promocionales** | **https://…/promo.mp4** | **<video> / JSON** |
| **Categoría** | **Ruta de categoría / breadcrumbs** | **Bebidas > Gaseosas** | **Breadcrumbs / JSON** |
| **Marca** | **Marca del producto** | **Coca-Cola** | **Tabla specs / JSON** |
| **Specs (varias)** | **Contenido neto, envase, sabor, etc.** | **Contenido: 2.25 L; Envase: PET; Sabor: Original** | **<td>…</td> / JSON** |
| **SEO / Metadatos** | **title, description, canonical, Open Graph** | **<title>, og:title, og:description** | **<head>** |
| **JSON-LD** | **Estructura Product, Offer, AggregateRating** | **rating 4.5, reviews 27** | **<script type="application/ld+json">** |
| **Reseñas** | **Rating promedio y reseñas individuales** | **4.5 / 27** | **JSON-LD / HTML** |
| **Promociones** | **Highlights, campañas y etiquetas activas** | **2do al 50 %, Ahorrames** | **<span data-highlight-name>** |
| **API Price / Stock** | **Endpoints internos de precios y stock por SKU** | **/api/pricing/14837, /api/stock-keeping** | **Llamadas fetch** |
| **Seller Info** | **Información de vendedor en marketplace** | **sellerId 123; Tienda oficial** | **JSON embebido** |
| **Tracking Analytics** | **Eventos dataLayer, UTM y seguimiento de impresiones** | **dataLayer.push({event:'productView'})** | **JS inline** |

**1. Inicialización y preparativos**

1. **Configurar entorno**
   * **Definir variables de entorno:**
     + **BASE\_URL (e.g. https://www.tienda.com)**
     + **API\_PRODUCTS\_SEARCH (e.g. /api/catalog\_system/pub/products/search)**
     + **API\_SKU\_DETAILS (e.g. /api/catalog\_system/pub/products/sku)**
   * **Cargar dependencias: HTTP client (Requests, Axios), concurrencia (ThreadPool, asyncio), logger.**
2. **Pool de proxies y rate-limit**
   * **Leer lista de proxies si aplica.**
   * **Definir máximos de solicitudes por segundo (e.g. 5 rps).**
   * **Implementar backoff exponencial en caso de 429/5xx.**
3. **Logger y trazabilidad**
   * **Iniciar logger con formatos:**
   * **[timestamp] [LEVEL] [categoría:page:batch:sku] mensaje**
   * **Crear IDs de ejecución (runId) para agrupar métricas.**
4. **Conexión a MongoDB**
   * **Pool de conexiones único.**
   * **Definir colección products con índices: skuId, categoryPath, runId.**

**2. Extracción de categorías**

1. **GET a home o endpoint de categorías**
   * **Preferir API interna si existe (p.e. /api/catalog\_system/pub/category/tree).**
   * **Fallback: parsear HTML de menú principal.**
2. **Mapear cada categoría**
   * **Extraer { name, slug, path: "Bebidas/Gaseosas", categoryId }.**
   * **Guardar lista en memoria: categories = [...].**
3. **Validación**
   * **Cada categoría debe tener path y categoryId.**
   * **Log de conteo: Total categorías: N.**

**3. Recorrido de listados por categoría**

**Para cada category en categories:**

1. **Construir query base**
2. **GET {BASE\_URL}{API\_PRODUCTS\_SEARCH}?fq=categoryPath:{category.path}&page=0&pageSize={pageSize}**
3. **Petición inicial**
   * **Esperar respuesta JSON.**
   * **Extraer totalRecords, pageSize.**
   * **Calcular totalPages = ceil(totalRecords/pageSize).**
4. **Iterar páginas (i = 0 … totalPages – 1)**
   * **Para cada i:**
   * **GET {BASE\_URL}{API\_PRODUCTS\_SEARCH}?fq=categoryPath:{path}&page={i}&pageSize={pageSize}**
   * **Validar status == 200. Reintentar hasta 3 veces con backoff.**
   * **Extraer array products: cada item con { productId, link, items: [{itemId: skuId, name}] }.**
   * **Agregar link y skuId a urlQueue.**
   * **Log:**
   * **[INFO] categ=Bebidas/Gaseosas page=3 fetchedRecords=50 totalPages=20**
5. **Finalizar listado**
   * **Verificar urlQueue.length == totalRecords.**
   * **Si hay discrepancia > 5%, marcar categoría para revisión.**

**4. Procesamiento concurrente de productos**

1. **Definir batch y concurrencia**
   * **Tamaño de batch: 20 URLs.**
   * **Concurrencia: 5–10 hilos o tasks asíncronas.**
2. **Worker por URL de producto Para cada skuUrl en urlQueue:**
   1. **GET directo a endpoint de SKU (recomendado)**
   2. **GET {BASE\_URL}{API\_SKU\_DETAILS}/{skuId}**
      * **Si no existe, caer al GET de página HTML y parsear JSON embebido.**
   3. **Validar respuesta**
      * **status == 200. Reintentos hasta 3 veces.**
      * **Parsear JSON en productData.**
   4. **Parseo de campos clave**
      * **Identificadores:**
        + **skuId, productId, referenceId, array de ean**
      * **Precios y promociones:**
        + **sellingPrice, listPrice, savings, priceTags, pricePerUnit**
      * **Imágenes / media:**
        + **imageUrls[], videos[]**
      * **Especificaciones técnicas (specificationGroups):**
        + **Cada { name: marca/envase/sabor/tipo… , values: […] }**
      * **Categorías y clusters (productClusterProperties):**
        + **{ name: "category", values: ["Gaseosas"] }**
      * **Promociones activas (clusterHighlights):**
        + **{ name: "Ahorrames", type: "collection" }, { name: "2do al 50%", type: "teaser" }**
      * **Stock y disponibilidad:**
        + **availabilityLevel, quantityBySeller[]**
      * **Rating y reseñas (jsonLD o productData.review):**
        + **AggregateRating.ratingValue, reviewCount, reseñas individuales si existen**
      * **SEO / metadata:**
        + **extraer metaTitle, metaDescription, canonicalUrl de productData.seo o JSON-LD.**
   5. **Validaciones y fallbacks**
      * **Si algún campo es null o vacío:**
        + **Registrar en errors array con { field, skuId, reason }.**
        + **Dependiendo de criticidad, marcar registro “incompleto” o “fail”.**
   6. **Estructura de documento a MongoDB**

**json**

**{**

**runId,**

**categoryPath,**

**skuId,**

**productId,**

**referenceId,**

**eans: [...],**

**name,**

**seo: { title, description, canonical },**

**prices: { sellingPrice, listPrice, savings, perUnit },**

**images: [...],**

**videos: [...],**

**specGroups: [...],**

**clusterProps: [...],**

**highlights: [...],**

**availability: { level, quantityBySeller },**

**rating: { value, count },**

**filtersImplicit: [ /\* specs + clusters + highlights \*/ ],**

**timestamp: ISODate()**

**}**

* 1. **Insertar/actualizar en MongoDB**
     + **Upsert {"skuId": skuId, "runId": runId}.**
     + **Registrar metadatos de inserción y versión de scraper.**
  2. **Logging final de URL procesada**
  3. **[INFO] sku=14837 status=OK fields=42 errors=0**

**5. Post-procesamiento y métricas**

1. **Validar integridad**
   * **Consultar conteo de registros vs totalRecords por categoría.**
   * **Detectar missingSkuIds y poner en retryQueue.**
2. **Reporte de errores**
   * **Generar CSV/JSON con lista de skuId y fields faltantes.**
   * **Alarmas si > 5% de registros incompletos.**
3. **Limpieza y cierre**
   * **Vaciar urlQueue, retryQueue.**
   * **Cerrar conexiones HTTP y MongoDB.**
   * **Guardar logs agregados al sistema de monitoreo.**
4. **Programación de re-scraping**
   * **Agendar retryQueue para próxima ejecución o alertar manualmente.**

**Notas finales**

* **No dependes de la navegación visual ni de iterar por cada filtro.**
* **Aprovechas toda la metadata embebida en JSON de VTEX, garantizando consistencia.**
* **Control absoluto de errores, logging y trazabilidad por runId, categoryPath, skuId.**
* **El pipeline escala horizontalmente: múltiples instancias pueden compartir urlQueue y retryQueue via Redis o similar.**

**Con este enfoque, minimizas tiempo de scraping, aprovechas APIs oficiales y mantienes un control riguroso en cada etapa.**

**1. Carga de configuración**

**1.1. Leer variables de entorno (.env) 1.2. Definir supermercado (disco, jumbo, etc.) 1.3. Generar runId único para esta ejecución 1.4. Iniciar logs con [INFO] inicio de ejecución**

**2. Conexión a MongoDB**

**2.1. Conectarse a la base precios\_productos 2.2. Seleccionar colección products.catalogo 2.3. Validar conexión activa 2.4. Loguear [INFO] conexión a MongoDB establecida**

**3. Lectura de categorías**

**3.1. Leer desde supermercados\_metadata\_<supermercado>.categorias\_metadata 3.2. Validar que cada categoría tenga URL y nombre 3.3. Loguear [INFO] categorías detectadas: N 3.4. Si no hay categorías válidas, abortar**

**4. Inicialización de Selenium**

**4.1. Iniciar webdriver.Chrome() con opciones headless 4.2. Validar que el driver responde 4.3. Loguear [INFO] Selenium iniciado**

**5. Iteración por categoría**

**Para cada categoría:**

**5.1. Cargar URL**

**5.1.1. driver.get(URL\_categoria) 5.1.2. Esperar carga con WebDriverWait 5.1.3. Loguear [INFO] categoría cargada: {nombre}**

**5.2. Scroll infinito**

**5.2.1. Ejecutar scroll hasta que no cambie el scrollHeight 5.2.2. Esperar entre scrolls 5.2.3. Loguear cada scroll ejecutado**

**5.3. Activación de filtros (si aplica)**

**5.3.1. Detectar si hay filtros visibles 5.3.2. Hacer click en cada filtro relevante 5.3.3. Esperar recarga del DOM 5.3.4. Loguear [INFO] filtro aplicado: {nombre}**

**5.4. Extracción de productos**

**5.4.1. Obtener driver.page\_source 5.4.2. Parsear con BeautifulSoup 5.4.3. Extraer cada producto con sus campos:**

* **skuId, productId, nombre, precio, imagen, link, promociones, specs, disponibilidad 5.4.4. Validar que cada producto tenga skuId y nombre 5.4.5. Loguear [INFO] productos encontrados: N**

**5.5. Persistencia en MongoDB**

**5.5.1. Construir documento con:**

**json**

**{**

**runId,**

**supermercado,**

**categoria,**

**skuId,**

**productId,**

**nombre,**

**precio,**

**specs,**

**imagenes,**

**promociones,**

**disponibilidad,**

**filtros\_aplicados,**

**timestamp**

**}**

**5.5.2. Guardar con updateOne({ skuId, runId }, { $set: doc }, { upsert: true }) 5.5.3. Loguear [INFO] producto guardado: {skuId}**

**6. Cierre por categoría**

**6.1. Vaciar variables temporales 6.2. Loguear [INFO] categoría finalizada: {nombre}**

**7. Finalización global**

**7.1. Cerrar driver.quit() 7.2. Cerrar conexión a MongoDB 7.3. Loguear [INFO] ejecución finalizada: {runId} 7.4. Salir con process.exit(0)**