Guía Del Desarrollador Para El Sistema Forecast

Este proyecto tiene como objetivo automatizar la predicción de consumo de productos y generar alertas para la reposición de inventario. La solución se compone de tres módulos principales:

- **AI Model**: Responsable del procesamiento de datos históricos y generación de proyecciones utilizando el modelo Prophet.
- **Backend**: Gestiona la lógica de negocio y expone una API para consumir y actualizar datos.
- **Frontend**: Proporciona una interfaz gráfica interactiva para cargar archivos, visualizar predicciones y administrar productos.

Lenguajes y herramientas utilizados:

- **Python** para el procesamiento de datos y entrenamiento del modelo.
- **Prophet** de Facebook para las predicciones de series temporales.
- **Pandas y NumPy** para el análisis y manejo de datos.
- **React + Next.js** en el frontend.
- **Node.js y Express** en el backend.

1. AI Model

1.1 Estructura de Carpetas

```
ai_model/
├── data/              # Contiene archivos de entrada como Excel o CSV
├── models/            # Modelos entrenados y comprimidos (Prophet)
├── src/
│   ├── predict.py     # Script principal de predicción
│   ├── train.py       # Entrenamiento del modelo Prophet
│   ├── requirements.txt# Dependencias del entorno
│   └── prediction_log.txt # Log de ejecuciones
```

1.2 Descripción de Archivos

- **predict.py**:
  - **Parámetros**:
    - excel: ruta del archivo Excel a procesar.
    - model: ruta del modelo Prophet entrenado.
    - transito: unidades en tránsito (si aplica).
  - **Carga de Excel**:
    - Lee el archivo Excel omitiendo las dos primeras filas (skiprows=2).

        - Elimina columnas vacías o innecesarias.
        - Valida que estén presentes columnas clave como "CODIGO", "DESCRIPCION", "STOCK TOTAL" y los consumos mensuales de 2024 y 2025.
    - **Carga del modelo Prophet**:
        - Intenta cargar el archivo .pkl.gz.
        - Si falla, registra el error pero continúa con método estadístico.
    - **Prepara los datos para Prophet**:
        - Para cada producto, arma una serie temporal (ds, y) con los consumos históricos mensuales.
        - Se guarda en un diccionario para predecir individualmente.
    - **Predicción con Prophet**:
        - Para cada producto, predice los siguientes 6 meses.
        - Calcula el MAPE comparando predicciones con los últimos dos meses de datos reales.
        - Si el error supera el 5%, se lanza una advertencia.
    - **Cálculos manuales si no hay Prophet**:
        - Si no hay modelo Prophet o para complementar, calcula:
        - Promedio histórico de consumo (PROM CONSU).
        - Consumo proyectado (PROM CONSU + Proyec).
        - Consumo diario (dividiendo entre 22 días laborales).
        - Stock de seguridad (SS = DIARIO * 19).
        - Punto de reorden (DIARIO * 44 días).
        - Cálculo de déficit, unidades a pedir, fecha de reposición, días de cobertura.
    - **Generación de JSON final**:
        - Cada producto tiene su proyección para los siguientes 6 meses.
        - Incluye configuraciones del modelo, valores históricos y alertas (por ejemplo, si el stock proyectado cubre menos de 22 días).

- **train.py**:
    - **Carga de datos** (consumo.csv):

```
df = pd.read_csv(data_path, parse_dates=['Fecha'], usecols=['Fecha', 'Consumo'])
```

        - Solo se usan las columnas Fecha y Consumo.
        - Se aseguran que las fechas estén en formato de serie temporal (ds) y los valores en y.
    - **Limpieza de datos**:
        - Elimina duplicados y verifica que no haya valores nulos.
    - **Entrenamiento del modelo**:

```
model = Prophet(...)
model.fit(df)
```

        - Se configura un modelo Prophet con poca sensibilidad (changepoint_prior_scale) y sin muestreo de incertidumbre (ligero).
        - No usa estacionalidad diaria ni semanal, solo anual.

- **Generación de predicción de prueba (opcional)**:
  - Se predicen 180 días para validar si funciona correctamente.
- **Guardado del modelo**:

```
with gzip.open(model_path, 'wb') as f:
    pickle.dump(model, f)
```

  - Se guarda comprimido en models/prophet_model.pkl.gz.

- **requirements.txt**:
  - Define las dependencias del entorno:
    - numpy
    - pandas
    - matplotlib
    - scikit-learn
    - tensorflow
    - prophet
- **prediction_log.txt**:
  - Archivo de log que registra la ejecución del sistema, incluyendo errores comunes como archivos faltantes o problemas con el modelo Prophet.
- **predicciones_completas.json / .min.json**:
  - Archivos generados por predict.py con la salida completa y minificada del sistema: proyecciones, alertas, configuración y métricas por SKU.

2. Backend

2.1 Estructura de Carpetas

```
backend/
├── config/             # Configuraciones generales (BD, constantes)
├── controllers/        # Lógica principal de cada endpoint
├── logs/               # Archivos de log del servidor
├── middlewares/        # Funciones intermedias (validaciones, autenticación)
├── models/             # Modelos de datos (reportes, usuarios)
├── routes/             # Definición de endpoints y sus rutas
├── services/           # Servicios que conectan lógica con otras capas
├── temp/, public/, utils/  # Datos temporales, archivos públicos y utilidades
├── app.js              # Archivo principal del servidor
├── .env                # Variables de entorno
├── package.json        # Dependencias del backend
```

2.2 Descripción de Archivos

- **app.js — Servidor principal**
  - Inicia un servidor Express.
  - Carga variables de entorno desde .env.
  - Configura middlewares como cors, express.json(), logs y manejo de archivos.
  - Registra las rutas desde routes/ para:

        - /api/predictions
        - /api/reports
        - /api/auth
        - entre otras.
- **config/db.js — Conexión a la base de datos**
    - Este archivo configura la conexión con la base de datos (SQLite o PostgreSQL). Utiliza una librería como sequelize o knex para hacer ORM, facilitando la interacción con modelos como report.model.js.
- **config/constants.js**

```
module.exports = {
  PREDICTION_PATH: path.join(__dirname, '../data/predicciones_completas.json'),
  // otros paths o constantes del sistema
};
```

    - Esto permite centralizar rutas a archivos generados por el AI Model como los JSON de predicción.
- **routes/predictions.routes.js**
    - Define las rutas relacionadas con predicciones:
    - GET /api/predictions: devuelve la lista general de productos.
    - GET /api/predictions/:codigo: devuelve detalles de un producto específico.
    - POST /api/predictions/refresh: recibe un archivo Excel y llama a python.service.js para ejecutar el script de predicción y actualizar los archivos JSON.
- **controllers/predictions.controller.js**
    - Controla toda la lógica de predicción:
    - Llama a los scripts de Python (train.py, predict.py) desde python.service.js.
    - Lee los archivos JSON generados y filtra la data por producto o lista general.
    - Maneja errores si el archivo no existe o el código no se encuentra.
- **services/python.service.js**
    - Encargado de ejecutar comandos del sistema como:

```
spawn('python', ['ai_model/src/predict.py'])
```

    - Este script se usa cuando se sube un archivo Excel nuevo. Genera los JSON de predicción actualizados y los deja listos para ser consumidos por la API.
- **models/report.model.js / user.model.js**
    - Define la estructura de las tablas reportes y usuarios respectivamente. Usan esquemas típicos:
    - report.model.js: filename, URL, userId, fecha.
    - user.model.js: email, password (hash), roles.
- **middlewares**
    - auth.middleware.js: verifica que el usuario esté autenticado usando JWT.
    - upload.middleware.js: valida archivos subidos, especialmente Excel.
- **utils/**
    - logger.js: sistema de logging personalizado.
    - generateToken.js: genera JWT para usuarios.

- **package.json (backend)**
  - Incluye dependencias clave como:

```
"dependencies": {
  "express": "^4.x",
  "cors": "^2.x",
  "jsonwebtoken": "^9.x",
  "morgan": "^1.x",
  "multer": "^1.x",
  "python-shell": "^3.x"
}
```

3. Backend

3.1 Estructura de Carpetas

```
frontend/
├── app/                    # Arquitectura App Router (Next.js 13+)
│   ├── components/         # Componentes reutilizables (formularios, paneles, etc.)
│   ├── styles/             # Estilos globales y layouts
│   ├── reportes/           # Vistas específicas para reportes
│   └── page.tsx            # Página principal (dashboard)
├── public/                 # Recursos estáticos (logos, íconos)
├── services/               # Conexiones con la API
├── styles/                 # Estilos generales
├── tailwind.config.js      # Configuración de Tailwind
├── next.config.ts          # Configuración de Next.js
├── package.json            # Dependencias del frontend
```

3.2 Descripción de Archivos

- **LoginForm.tsx**
  - Este componente permite a los usuarios iniciar sesión. Está construido con React y TailwindCSS, e incluye:
  - Inputs controlados para email y password.
  - Un botón de envío con validación básica.
  - Manejo de errores si las credenciales son incorrectas.
  - Se espera que llame a un endpoint de autenticación en el backend (como /api/auth/login).
  - Internamente tiene algo como esto:

```
const handleLogin = async () => {
  const res = await fetch("/api/auth/login", { ... });
  if (!res.ok) { setError("Credenciales inválidas"); }
}
```

- **reportes.ts (dentro de /services/)**
    - Este archivo define funciones para conectarse con el backend, específicamente con la API de reportes.

```
export async function saveReport(data: { filename: string; url: string }) {
  return await fetch("/api/reports", {
    method: "POST",
    body: JSON.stringify(data),
    headers: { "Content-Type": "application/json", ... },
  });
}
```

    - También incluye funciones como:
        - Obtener todos los reportes.
        - Eliminar uno.
        - Descargar archivos por su URL.
- **page.tsx (en /app/)**
    - Es la **página principal del frontend**. Generalmente incluye:
        - Un layout general (<Layout> o similar).
        - Llama a componentes clave como Dashboard, AlertsPanel, LoginForm, etc.
        - Puede tener lógica condicional para mostrar distintos elementos si el usuario está autenticado o no.
        - Renderiza gráficas o tablas con información cargada desde el backend.
    - Ejemplo simplificado:

```
return (
  <main className="container mx-auto p-4">
    <Dashboard />
    <AlertsPanel />
  </main>
);
```

- **Dashboard.tsx y AlertsPanel.tsx**
    - **Dashboard.tsx**
        - Actúa como **panel principal**.
        - Muestra resumen de productos, predicciones, consumo promedio, etc.
        - Puede mostrar tarjetas tipo dashboard con métricas clave.
    - **AlertsPanel.tsx**
        - Muestra alertas basadas en predicciones: productos con déficit, bajo stock, o próximos a punto de reorden.
        - Se conecta al backend o consume los JSON generados por predict.py.
    - Ejemplo de alerta:

```
{
  alert.stock < alert.minStock &&
  <Alert type="danger" message={`Alerta de stock para ${alert.product}`} />
}
```

- **tailwind.config.js**
    - Archivo de configuración de Tailwind CSS. Incluye:
        - Activación de modo JIT y darkMode.
        - Personalización de temas:
            - Colores personalizados (azules, grises, verdes)
            - Tipografía (fuentes modernas)
            - Extend para clases como boxShadow, borderRadius, etc.
    - Ejemplo:

```
theme: {
  extend: {
    colors: {
      primary: '#0074CF',
      secondary: '#001A30'
    }
  }
}
```