

# Web App Price Scraper

I want you to act as a software developer. I will provide some specific information about a web app requirements, and it will be your job to come up with an architecture and code for developing secure app with a JS framework. My first request is "I want an app to run on a server and periodically scrape websites for price and stock and warn for reductions".

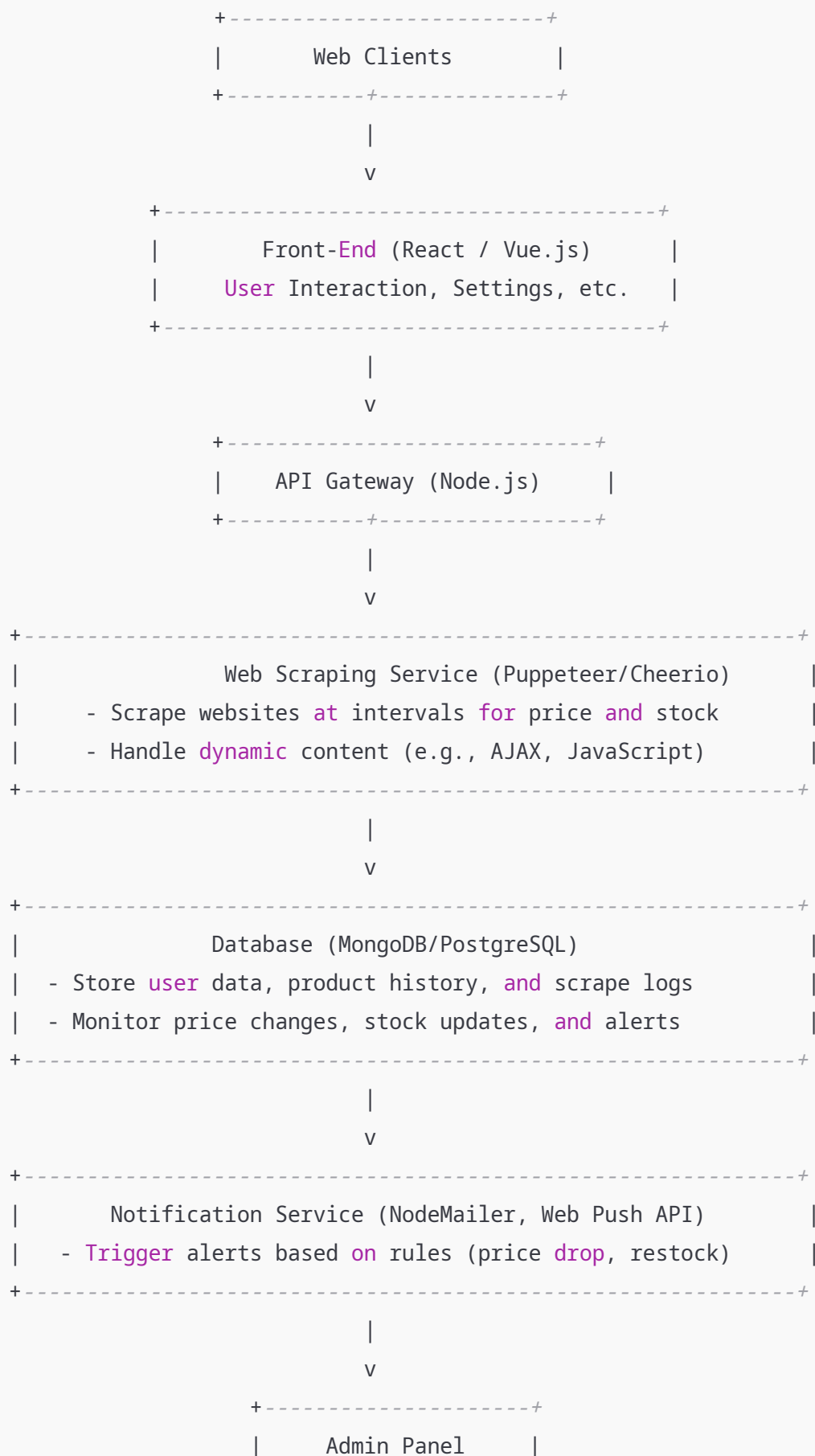
To design a web app that periodically scrapes websites for price and stock data, warns about reductions, and is secure, let's break down the key aspects:

## 1. App Requirements

- **Core Features:**
  - Periodically scrape websites for product prices and stock availability.
  - Monitor specific products across multiple websites.
  - Track historical prices and identify reductions.
  - Alert users when a price drops or stock becomes available.
- **Technology Stack:**
  - **Front-End:** JavaScript Framework (React, Vue.js, etc.)
  - **Back-End:** Node.js with Express.js or Next.js for server-side functionality.
  - **Database:** MongoDB (NoSQL) or PostgreSQL (relational) to store product data, price history, user preferences, etc.
  - **Scraping Tool:** Puppeteer or Cheerio.js for web scraping.
  - **Scheduler:** Node-cron or a similar library for periodic scraping.
  - **Authentication:** JSON Web Tokens (JWT) for securing user logins.
  - **Notifications:** Email alerts (NodeMailer) or web push notifications.

## 2. High-Level Architecture

```
sql
```



```
| (Admin Product List)|
+-----+
```

### 3. Main Components

#### 1. Web Scraper:

- Use **Puppeteer** for headless browser-based scraping, especially if sites rely on JavaScript.
- For static websites, **Cheerio.js** (lightweight, fast) can extract HTML content.
- **Node-cron** will schedule scraping tasks at defined intervals (e.g., hourly or daily).

#### 2. Backend (API):

- **Express.js/Next.js** to create RESTful APIs.
- Implement endpoints for managing:
  - User profiles (sign up, login, preferences)
  - Product tracking (add/remove products, set alert thresholds)
  - Notifications (fetching alerts, viewing alert history)
- Secure APIs using JWT for authentication.

#### 3. Database:

- Use **MongoDB** or **PostgreSQL** to store user preferences, product history, and tracked data.
- Store scrape timestamps and historical prices for comparisons.
- Use indices on product URLs and timestamps to optimize querying performance.

#### 4. Alerts/Notifications:

- Send notifications via email or push notifications when price reductions or stock changes are detected.
- Set user-defined alert thresholds (e.g., notify only if a price drops by 10% or more).

#### 5. Front-End:

- **React.js** or **Vue.js** for managing the user interface, product lists, and alert settings.

- Create forms for user authentication, product management, and notification preferences.
- Show price histories for tracked products and allow users to configure scraping intervals.

## 6. Security:

- Use **JWT** for secure authentication.
- Ensure input validation and sanitation to avoid injection attacks.
- Implement rate limiting and anti-bot measures to prevent scraping abuse.
- HTTPS for secure data transfer.

---

## 4. Sample Code Structure

### 1. Backend (Express.js):

- Basic API endpoint to start scraping:

javascript

```
// server.js
const express = require('express');
const cron = require('node-cron');
const puppeteer = require('puppeteer');
const { checkForPriceDrops } = require('./scrapeService');

const app = express();

app.use(express.json());

app.post('/track-product', (req, res) => {
  // Save product details to DB
  // Example: req.body = { url, targetPrice, userId }
  res.send({ message: "Product added for tracking!" });
});
```

```

app.get('/alerts', (req, res) => {
  // Fetch price reduction alerts from DB for the user
  res.send({ alerts: [] });
});

// Set up scheduled scraping
cron.schedule('0 * * * *', async () => {
  await checkForPriceDrops();
  console.log('Scraping done!');
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));

// scrapeService.js
const puppeteer = require('puppeteer');

async function checkForPriceDrops() {
  // Get products from the database to track
  const products = await getTrackedProducts();

  const browser = await puppeteer.launch();
  for (let product of products) {
    const page = await browser.newPage();
    await page.goto(product.url);
    const price = await page.evaluate(() => {
      // Replace with correct selector
      return document.querySelector('.price').innerText;
    });

    // Compare and update the database with the latest price
    if (parseFloat(price) < product.targetPrice) {
      // Trigger notification alert for the user
      sendAlert(product.userId, product.url, price);
    }
  }
  await browser.close();
}

module.exports = { checkForPriceDrops };

```

## 2. Frontend (React.js):

- Simple product tracking form:

javascript

```
import React, { useState } from 'react';

function TrackProduct() {
  const [url, setUrl] = useState('');
  const [targetPrice, setTargetPrice] = useState('');

  const handleSubmit = async (e) => {
    e.preventDefault();
    const res = await fetch('/track-product', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ url, targetPrice })
    });
    const data = await res.json();
    alert(data.message);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>Product URL:</label>
      <input type="text" value={url} onChange={(e) => setUrl(e.target.value)} />
      <label>Target Price:</label>
      <input type="number" value={targetPrice} onChange={(e) =>
setTargetPrice(e.target.value)} />
      <button type="submit">Track Product</button>
    </form>
  );
}

export default TrackProduct;
```

## 5. Next Steps

- **Improve the scraping service:** Handle website-specific dynamic content (AJAX, delayed loads).
- **Integrate notification system:** Set up email notifications with NodeMailer.
- **Optimize for performance:** Use caching and reduce the load on servers with proper scheduling.
- **Enhance security:** Use proper input validation and deploy on a secure cloud server (with HTTPS).

Would you like to go deeper into any of these aspects?