**WEEK - 1**

**Step 1: Project Setup**

* Created server.js in **root of backend**.
* Installed dependencies: express, mongoose, cors, dotenv.
* Connected Express server to MongoDB.
* Configured CORS to allow frontend (http://localhost:5173).

**Flow**:

Frontend (http://localhost:5173) → Express (port 8000) → MongoDB

**Step 2: User System (with Portfolio in schema)**

* Created **User model** in models/User.js.
  + Fields: username, email, password, portfolio (array of stock symbols).
* Created **userAuth routes** in routes/userAuth.js.
  + POST /api/auth/signup → Register new user.
  + POST /api/auth/login → Authenticate existing user.
* Hooked routes into server.js under /api/auth.
* Tested with Postman → working fine.

**Flow**:

Client → POST /api/auth/signup → Save user in MongoDB

Client → POST /api/auth/login → Verify user from MongoDB

**🔗 Flow Chart Representation (Till Now)**

┌────────────────────┐

│ Frontend (React) │

│ http://localhost:5173

└─────────┬──────────┘

│

▼

┌────────────────────┐

│ Express Server │

│ (server.js, port 8000)

└──────┬─────────────┘

│

┌─────────────┴────────────────┐

│ │

▼ ▼

┌───────────────────┐ ┌─────────────────────┐

│ /api/auth/signup │ │ /api/auth/login │

│ (register user) │ │ (authenticate user)│

└───────────────────┘ └─────────────────────┘

│

▼

┌────────────────────┐

│ MongoDB (Users) │

│ { username, email, │

│ password, │

│ portfolio: [] } │

└────────────────────┘

**FRONTEND**

**🛠 Workflow (Signup.jsx)**

1. **State Management**
2. const [form, setForm] = useState({ username: "", email: "", password: "" });
3. const [loading, setLoading] = useState(false);
   * form → ek object hai jisme hum **username, email, password** store karte hain.
   * loading → ek flag jo batata hai ki API request chal rahi hai ya nahi.
4. **Navigation Setup**
5. const nav = useNavigate();
   * useNavigate hook React Router se aata hai. Isse signup ke baad user ko **Login page** pe bhej dete hain.
6. **Input Change Handler**
7. const onChange = (e) => setForm({ ...form, [e.target.name]: e.target.value });
   * Har input field ke name attribute ke base par value update hoti hai.
   * Example: agar username input change hua → form.username update hoga.
   * Yeh ek **generic handler** hai jo sabhi input fields ke liye kaam karega.
8. **Form Submission**
9. const submit = async (e) => {
10. e.preventDefault();
11. setLoading(true);
12. try {
13. const res = await API.post("/api/auth/signup", form);
14. alert(res.data.message || "User created");
15. nav("/login");
16. } catch (err) {
17. alert(err.response?.data?.message || err.message);
18. } finally {
19. setLoading(false);
20. }
21. };
    * e.preventDefault() → default form submit behavior (page reload) rokta hai.
    * setLoading(true) → button disable kar dete hain aur message "Creating user..." show karte hain.
    * API call:
    * await API.post("/api/auth/signup", form);

Ye backend ke **signup route** pe form data bhejta hai.

* + Agar success:
    - User ko alert show hota hai (User created)
    - nav("/login") ke through **Login page pe redirect** ho jata hai.
  + Agar fail:
    - Error message alert hota hai (err.response.data.message agar available hai).
  + Finally block:
    - setLoading(false) → request complete hone ke baad loading flag reset hota hai.

1. **Input Fields**
2. <input
3. name="username"
4. value={form.username}
5. onChange={onChange}
6. required
7. />
   * name → decide karta hai form object ka kaunsa property update hoga.
   * value={form.username} → input ko React state ke saath sync rakhta hai.
   * onChange={onChange} → har type karne par form update hota hai.
   * required → field ko empty chhodkar submit nahi kar sakte.
8. **Submit Button**
9. <button type="submit" disabled={loading}>
10. {loading ? "Creating user..." : "Signup"}
11. </button>
    * Agar API call chal rahi hai (loading = true) → button disable + text change.
    * Warna normal "Signup" button show hota hai.

👉 Net result:

* User inputs → React state me store hota hai.
* Submit pe API call → success pe login page redirect.
* Button aur alerts se user ko clear feedback milta hai.

**🛠 Workflow (login.jsx)**

**🔄 Workflow samajhne ke liye**

1. User **form ke input (email + password)** fill karega.
2. Form submit hone par submit() function chalega.
3. Ye API ko request bhejega (/api/auth/login) backend pe.
4. Agar success → user data ko localStorage me save karenge → navigate karega dashboard pe.
5. Agar error → alert me dikha dega.

**🔍 Code Explanation with "Why"**

**1. Imports**

import React, { useState } from "react";

import { useNavigate } from "react-router-dom";

import API from "../api/axios";

* **useState**: form ke andar user jo type karega (email/password), usse track karne ke liye.
* **useNavigate**: react-router ka hook jo hume programmatically dusre page pe bhejne ke kaam aata hai (yaha login ke baad /dashboard).
* **API**: axios instance jo backend se baat karne ke liye banaya gya.

👉 Ye teen import isliye kiye gaye taki **state maintain ho, navigation ho, aur backend se communication ho sake**.

**2. State and navigate**

const [form, setForm] = useState({ email: "", password: "" });

const nav = useNavigate();

* **form** ek object hai jisme email aur password rakhe ja rahe hain.
* Initially dono empty hain.
* **nav** ek function hai jo navigate karega dashboard pe.

👉 Reason: **form input values track karna aur login ke baad redirect karna**.

**3. Input change handler**

**Code:**

const onChange = (e) => setForm({ ...form, [e.target.name]: e.target.value });

**1️ ...form ka use**

* form ek **state object** hai jo abhi ki sari input values hold karta hai:
* { email: "", password: "" }
* Jab user koi ek field type kare (jaise email), hum **existing state ko preserve** karna chahte hain.
* ...form matlab **spread operator**: "form ke sabhi current key-value pairs ko copy karo".
* Agar hum ye nahi lagate, aur sirf { [e.target.name]: e.target.value } likhte → baki fields (jaise password) reset ho jayenge.

**Example:**

form = { email: "abc@gmail.com", password: "12345" }

User type kare email field me: "xyz@gmail.com"

setForm({ [e.target.name]: e.target.value })

// result: { email: "xyz@gmail.com" } → password lost ho gaya

**Isliye hum use karte hain:**

setForm({ ...form, [e.target.name]: e.target.value })

* Result: { email: "xyz@gmail.com", password: "12345" } → baki values safe hain.

**2️ [e.target.name] ka matlab**

* e.target.name → input field ka **name attribute** leta hai (jaise "email" ya "password").
* [e.target.name]: e.target.value ka matlab hai:
  + **dynamic key update**: jis field me user type kar raha hai, uski value update karo.

**Example:**

* Input field:

<input name="email" />

* User type → e.target.name = "email"
* [e.target.name]: e.target.value → { email: "new value" }

**3️ : kyun use hua hai**

* JS me object me key-value define karte hain:

{ key: value }

* [e.target.name]: e.target.value → dynamic key ka syntax
  + Agar key fixed hoti: email: e.target.value → sirf email update hota
  + [e.target.name] → variable ke through key decide hoti (email/password/username)

**4. Submit handler**

const submit = async (e) => {

e.preventDefault();

try {

const res = await API.post("/api/auth/login", form);

const user = res.data.user;

if (user?.password) delete user.password;

localStorage.setItem("user", JSON.stringify(user));

nav("/dashboard");

} catch (err) {

alert(err.response?.data?.message || err.message);

}

};

* **e.preventDefault()**: form submit hone par page reload na ho.
* **API.post("/api/auth/login", form)**: backend ko POST request bhej rahe hain → body me email + password.
* **res.data.user**: backend se user ka data aata hai.
* **Delete password**: backend temporarily password bhej raha hai (learning purpose). Security ke liye localStorage me save karne se pehle delete kar diya.
* **localStorage.setItem**: user data ko browser me save kar diya taaki login hone ka proof rahe.
* **nav("/dashboard")**: agar login success ho gya to dashboard page pe redirect.
* **catch block**: agar error aaya (galat email/password), to alert me dikha do.

👉 Reason: **form ko backend pe bhejna, response handle karna, aur user ko redirect karna**.

**5. UI (JSX Form)**

<form onSubmit={submit}>

<input name="email" ... />

<input name="password" ... />

<button type="submit">Login</button>

</form>

* **onSubmit={submit}**: jab user login button click karega → submit() chalega.
* **name="email" / name="password"**: ye important hai kyunki onChange me humne [e.target.name] use kiya hai.
* **value={form.email} / value={form.password}**: ye state ke saath input ko sync me rakhta hai (controlled component).
* **Button**: submit karne ke liye.

👉 Reason: **inputs ko state se connect karna aur ek proper form ready karna**.

**📝 Simple Summary**

* **useState** → input values track karne ke liye.
* **onChange** → user jo likhe usse update karne ke liye.
* **submit** → backend pe request bhejne, localStorage me user save karne aur redirect karne ke liye.
* **localStorage** → login session maintain karne ke liye.
* **useNavigate** → login ke baad dusre page pe bhejne ke liye.

**🛠 Workflow (Dashboard.jsx)**

**🔄 Workflow**

1. Dashboard component load hote hi **localStorage se user data** fetch hota hai.
2. Agar user login nahi hai → message dikha ke login page pe redirect ka suggestion.
3. Agar user logged in hai → dashboard ke upar welcome message + user ID dikha.
4. PortfolioManager component ko render karte hain, jisme user ka portfolio manage hoga.

**🔍 Code Explanation**

**1️ Fetch user from localStorage**

const raw = localStorage.getItem("user");

* localStorage.getItem("user") → browser ke localStorage se previously save kiya gya user data fetch karta hai.
* **Why**: frontend ko pata hona chahiye ki kaun login hai, tabhi personalized dashboard show hoga.

**2️ Check if user is logged in**

if (!raw) {

return (

<div>

<h2>You are not logged in</h2>

<p>Go to <a href="/login">Login</a></p>

</div>

);

}

* Agar raw null hai → iska matlab user login nahi hai.
* **Return early** → dashboard ke rest components render nahi honge.
* **Why**: Unauthorized access rokna aur user ko login prompt dikhana.

**3️ Parse user object**

const user = JSON.parse(raw);

* localStorage me jo string store hoti hai → JSON.parse se object me convert karte hain.
* **Why**: tab hum user.username, user.\_id etc access kar sakte hain.

**4️ Render welcome message**

<h1>Welcome, {user.username || user.email}</h1>

<p>User ID: {user.\_id}</p>

* **user.username || user.email** → agar username available nahi hai to email show karenge.
* **Why**: Personalized greeting dene ke liye.

**5️ Render PortfolioManager**

<PortfolioManager userId={user.\_id} />

* PortfolioManager component me userId pass kar rahe hain.
* **Why**: PortfolioManager ko backend se portfolio fetch karne ke liye userId chahiye.
* PortfolioManager me CRUD operations (add/remove stocks) user ke liye specific honge.

**✅ Summary**

| **Part** | **Reason** |
| --- | --- |
| localStorage.getItem("user") | Check kaun login hai |
| if (!raw) | Unauthorized user ke liye message/redirect |
| JSON.parse(raw) | String → JS object conversion, username/email access ke liye |
| Welcome message | Personalized UI |
| <PortfolioManager userId={user.\_id} /> | Portfolio management ke liye backend se data fetch aur update |

**🛠 Workflow (PortfolioManager.jsx)**

**🔄 Workflow**

1. Component load hote hi userId ke basis pe portfolio fetch hota hai backend se.
2. Portfolio data state me store hota hai → UI me render hota hai.
3. User naya stock add kare → add function → backend call → state update → input clear.
4. User existing stock remove kare → remove function → backend call → state update.

**🔍 Code Explanation**

**1️ State declarations**

const [portfolio, setPortfolio] = useState([]);

const [symbol, setSymbol] = useState("");

const [loading, setLoading] = useState(false);

* **portfolio** → user ke stocks ka list
* **symbol** → input field ka value
* **loading** → add operation ke time button disable karne ke liye

**Why**: React state se hi UI dynamically update hoti hai.

**2️ Fetch portfolio function**

const fetchPortfolio = async () => {

try {

const res = await API.get(`/api/portfolio/${userId}`);

setPortfolio(res.data.portfolio || []);

} catch (err) {

alert(err.response?.data?.message || err.message);

}

};

* API.get → backend ke endpoint se portfolio fetch karta hai.
* setPortfolio → fetched portfolio ko state me save karta hai.
* **Why**: component mount hote hi user ka latest portfolio dikhe.

**3️ useEffect hook**

useEffect(() => { fetchPortfolio(); }, [userId]);

* **useEffect** → component render ke baad fetchPortfolio call karta hai.
* [userId] → dependency array, agar userId change hua → portfolio refetch hoga.
* **Why**: ensure latest portfolio data fetch ho har user ke liye.

**4️ Add stock function**

const add = async (e) => {

e?.preventDefault();

if (!symbol.trim()) return;

setLoading(true);

try {

const res = await API.post(`/api/portfolio/${userId}/add`, { symbol });

setPortfolio(res.data.portfolio);

setSymbol("");

} catch (err) {

alert(err.response?.data?.message || err.message);

} finally { setLoading(false); }

};

* **e.preventDefault()** → form submit hone par page reload na ho
* **Validation** → empty string check
* **setLoading(true)** → button disable / loading indicator
* **API call** → new symbol backend me add
* **setPortfolio** → state update → UI me reflect
* **setSymbol("")** → input clear

**Reason**: CRUD operation ka frontend flow maintain karne ke liye.

**5️ Remove stock function**

const remove = async (sym) => {

if (!confirm(`Remove ${sym} from portfolio?`)) return;

try {

const res = await API.delete(`/api/portfolio/${userId}/remove/${sym}`);

setPortfolio(res.data.portfolio);

} catch (err) {

alert(err.response?.data?.message || err.message);

}

};

* **Confirm dialog** → user accidentally delete na kare
* **API.delete** → backend se stock remove
* **setPortfolio** → state update → UI refresh

**Reason**: safe deletion and real-time UI update.

**6️ Return / JSX**

<form onSubmit={add}> ... </form>

<ul>

{portfolio.map((s) => (

<li key={s}>

{s} <button onClick={() => remove(s)}>Remove</button>

</li>

))}

</ul>

* Form → naya stock add karne ke liye
* List → portfolio items show
* Remove button → remove function call

**Why**: user interactive UI ke through apna portfolio manage kare.

**✅ Summary**

| **Part** | **Reason** |
| --- | --- |
| useState | Dynamic UI update ke liye |
| fetchPortfolio | Initial aur latest portfolio fetch |
| useEffect | Component mount ke baad API call |
| add function | Naya stock add, input clear aur loading handle |
| remove function | Stock remove + confirm + state update |
| JSX | Form aur list render, button actions bind |

**🛠 Workflow (api/axios.js)**

**🔹 Code Breakdown**

import axios from "axios";

* **axios** → ek popular HTTP client library jo browser aur NodeJS me kaam karta hai.
* **Why**: Ye fetch se better hai because easy syntax, automatic JSON parsing, interceptors, aur error handling provide karta hai.

const BASE = import.meta.env.VITE\_API\_URL || "http://localhost:8000";

* **import.meta.env.VITE\_API\_URL** → Vite framework me environment variable ko access karne ka tarika.
  + .env file me defined VITE\_API\_URL=http://localhost:4000
  + Ye backend ka base URL store karta hai.
* **Fallback** → agar .env variable missing hai, tab http://localhost:8000 default use hoga.
* **Reason**: Project deploy hone par easy switching between local, dev, production backend.

**Example**:  
VITE\_API\_URL = http://localhost:4000  
=> BASE = "http://localhost:4000"

const API = axios.create({

baseURL: BASE,

headers: { "Content-Type": "application/json" },

// withCredentials: true, // keep for later if you enable cookies/sessions

});

**1️ axios.create()**

* Ye ek **custom instance** banata hai jisse hum baar-baar base URL aur headers set kar sake.
* Har API call me ye automatically use hoga.

**2️ baseURL**

* Har request ke liye **base URL** define karta hai.
* API.get("/api/portfolio/123") → full URL banega:
* http://localhost:4000/api/portfolio/123
* **Reason**: Har jagah full URL likhne ki zarurat nahi, DRY principle maintain hota hai.

**3️ headers**

headers: { "Content-Type": "application/json" }

* Har request me ye header automatically include hoga.
* **Why**: Backend ko pata chal sake ki request JSON format me hai.

**4️ withCredentials (commented)**

* Agar backend cookies/session use karega → withCredentials: true enable karna hoga.
* Filhal hum JWT/cookies nahi use kar rahe, isliye comment me rakha hai.

export default API;

* Is custom axios instance ko baaki frontend files me import karenge:

import API from "../api/axios";

* Phir use karenge:

API.get("/api/portfolio/123")

API.post("/api/auth/login", form)

API.delete("/api/portfolio/123/remove/AAPL")

**🔹 Example Full URL Calculation**

1. **Frontend call:**

API.get("/api/portfolio/123")

1. **Base URL**: http://localhost:4000
2. **Combined URL**: http://localhost:4000/api/portfolio/123

Similarly:

API.post("/api/auth/login", { email, password })

* Full URL: http://localhost:4000/api/auth/login
* Body: { email, password }
* Header: Content-Type: application/json

**✅ Key Points / Reasoning**

| **Concept** | **Why Needed** |
| --- | --- |
| axios.create | DRY, reusable instance with baseURL & headers |
| baseURL | Har request ke liye full URL likhne ki zarurat nahi |
| headers | Backend ko pata chale ki request JSON hai |
| withCredentials | Future me cookies/session ke liye optional |
| export default | Ek hi centralized instance har component me use kar sake |

**CONNECTING FRONTEND AND BACKEND (AXIOS.JS)**

**Overview — axios file mein hum kya implement kar rahe hain**

Hum ek centralized HTTP client bana rahe hain jo: backend ki base URL set kare, har request ko JSON headers de, JWT token automatically add kare, global 401 handling kare, aur token ko manage karne ke helper functions provide kare.  
(Ye sab isliye taaki frontend code clean rahe aur auth logic ek jagah ho.)

**1) Base URL banaya gaya**

**Code (one line):**

const BASE = import.meta.env.VITE\_API\_URL || "http://localhost:8000";

**Kyun:** agar environment variable set ho to use karo, warna local dev server use karenge — easy switching between dev/prod.

**2) Axios instance create kiya (config)**

**Code (one line):**

const API = axios.create({ baseURL: BASE, timeout: 10000, headers: { "Content-Type": "application/json" } });

**Kyun:** ek shared client jisme base URL, timeout aur default JSON header set ho — baar-baar same config likhne ki zaroorat nahi.

**3) withCredentials comment (note)**

**Code (one line, commented):**

// withCredentials: true, // keep for later if you enable cookies/sessions

**Kyun:** agar future mein httpOnly cookie based auth use karoge to ye enable karna padega; filhal localStorage JWT ke saath use nahi karte.

**4) Request interceptor — Authorization header automatically add**

**Code (one line core):**

const token = localStorage.getItem("token"); if (token) config.headers.Authorization = `Bearer ${token}`;

**Kyun:** har request ke saath JWT bhejna automatically — har component mein manually header na lagana pade.

**5) Request interceptor — try/catch around localStorage**

**Code (one line):**

try { const token = localStorage.getItem("token"); ... } catch (e) { /\* ignore \*/ }

**Kyun:** SSR ya restricted browser contexts mein localStorage throw kar sakta hai — app crash se bachane ke liye safe guard.

**6) Response interceptor — global 401 handling**

**Code (one line core):**

if (error?.response?.status === 401) { localStorage.removeItem("token"); window.location.replace("/login"); }

**Kyun:** agar token invalid/expired ho to centrally token clear karke user ko login page pe bhej denge — har component mein same logic duplicate na karein.

**7) Response interceptor — rethrow error**

**Code (one line):**

return Promise.reject(error);

**Kyun:** specific components ko bhi error handle karne ka mauka chahiye (e.g., toast messages), isliye error ko propagate karte hain.

**8) setAuthToken helper**

**Code (one line):**

export function setAuthToken(token) { if (!token) return; localStorage.setItem("token", token); }

**Kyun:** login ke baad token store karne ka single place — consistency aur testability badhti hai.

**9) clearAuthToken helper**

**Code (one line):**

export function clearAuthToken() { localStorage.removeItem("token"); }

**Kyun:** logout ya global 401 handling mein token clear karne ka single function — DRY principle.

**10) getAuthToken helper**

**Code (one line):**

export function getAuthToken() { return localStorage.getItem("token"); }

**Kyun:** kahin token chahiye ho (debug/conditional logic) to direct localStorage access na karein — wrapper use karo.

**11) export default API**

**Code (one line):**

export default API;

**Kyun:** baki components sirf import API from "src/api/axios" karke use kar sakte — centralized client se sab requests jayengi.

**----------------------------NewsFecther fixes---------------------------------**

Good — I understand. Below I’ll explain the *current flow* of the code you pasted, endpoint by endpoint and file-by-file, showing the step-by-step behaviour, what DB collections are touched, what is deduped/updated, and then I’ll list likely issues and practical, low-risk fixes you can apply quickly. I’ll *not* assume anything you didn’t provide — if you later want the Article / Portfolio model files or the provider file I can analyze them too.

**High-level picture (one-sentence)**

When the server starts it connects to MongoDB, mounts three route groups (/user, /user/portfolio, /user/news), starts a cron scheduler that periodically calls runNewsFetch() which reads portfolio tickers, queries configured providers (marketaux), dedupes results and upserts Article documents. There is also a manual endpoint to trigger fetching immediately and endpoints to summarize/analyze saved articles (those require auth).

**server.js — startup flow (step-by-step)**

1. import "dotenv/config" loads environment variables.
2. Create an Express app: const app = express().
3. Middlewares applied:
   * cors() — enables cross-origin requests (currently wide open).
   * express.json() — parses JSON bodies.
4. Route modules imported and mounted:
   * userAuthRoutes at /user
   * portfolioRoutes at /user/portfolio
   * newsRouter at /user/news
5. Connect to MongoDB: await connectDB() inside an IIFE — server waits for DB connection before listening.
6. Start HTTP server: app.listen(PORT, ...). Then:
   * server.setTimeout(5 \* 60 \* 1000) — extends socket timeout to 5 minutes (so long requests don't immediately close).
7. startNewsScheduler() is called — this schedules the periodic background fetch (cron).
   * Note: startNewsScheduler() only sets up scheduling; it does not block the server.

**DB collections used here:** implicitly Portfolio and Article (see controllers).

**routes/newsRoutes.js — endpoints & flow**

Paths are mounted under /user/news due to server.js.

**1) GET /user/news?stock=RELIANCE&limit=10**

**Purpose:** fetch list view (headlines) of saved articles for a given stock.

Step-by-step:

1. Read req.query.stock and req.query.limit.
   * stock is coerced to uppercase.
   * limit sanitized to integer between 1 and 50 (default 10).
2. If no stock provided → 400 { error: "stock (query) required" }.
3. Query MongoDB:
4. const docs = await Article.find({ stockName: stock })
5. .sort({ publishedAt: -1 })
6. .limit(limit)
7. .lean();
   * Returns most recent limit articles for that stockName.
8. Map docs to a slim list with fields: id, title, source, url, publishedAt, provider.
9. Return 200 { ok: true, items: [...] }.
10. On error, 500 { error: err.message }.

**DB touched:** Article collection (read only).

**Notes:** No auth required. Pagination is by limit only (no offset/page token).

**2) POST /user/news/fetch-now**

**Purpose:** manual trigger to run the news fetcher on-demand.

Step-by-step:

1. fetchNow controller receives req.body (options).
2. Calls await runNewsFetch(opts).
3. If success → 200 { ok: true, message: "News fetch started/completed" }.
4. On error → 500 { ok: false, error }.

**DB touched:** Portfolio (read) and Article (write/upsert) inside runNewsFetch.

**Security:** **UNPROTECTED** — anyone who can hit this endpoint can trigger a run. (I’ll suggest fixes below.)

**3) POST /user/news/:articleId/summerize and POST /user/news/:articleId/analyze**

**Purpose:** perform analysis/summarization on an existing Article.

Step-by-step (router-level):

1. authMiddleware runs first — user must be authenticated.
2. Route handler (summerizeArticle / analyzeArticleController) executes (implementation not provided here).
   * Likely reads the Article by articleId, runs some NLP/AI analysis, then updates the Article doc with summary, key\_facts, impact\_label, etc.
3. Responds with analysis result or error.

**DB touched:** Article (read + update). Possibly audit fields under raw\_payload.analysis per your project notes.

**controller/newsFetcher.js — runNewsFetch() detailed flow**

This is the core of fetch work. I’ll go line-by-line conceptually.

1. **Setup**
   * PROVIDERS is an array of provider adapters (e.g., marketaux). Each entry has enabled set based on presence of provider API key env var (MARKETAUX\_API\_KEY).
   * TOP\_N reads default limit per ticker from env FETCH\_TOP\_N\_PER\_TICKER.
2. **upsertArticle(stockName, normalized)**
   * Build a doc object with normalized fields: stockName, title, url, source, provider, providerId, publishedAt, fetchedAt, summary (empty), raw\_text, entities, raw\_payload, country, language, etc.
   * Determine filter for dedupe/upsert:
     + prefer { provider, providerId } if present
     + else { url } if present
     + else { stockName, title, publishedAt }
   * Calls:
   * const up = await Article.updateOne(filter, { $setOnInsert: doc }, { upsert: true });
     + Only sets the doc on insert — *existing docs are not updated* (only new docs may be inserted).
   * Heuristically tests upsert result properties (up.upsertedId || up.upsertedCount || up.nUpserted) to decide inserted boolean and returns { inserted, doc }.
3. **runNewsFetch({ limitPerTicker = TOP\_N })**
   * Initialize stats object to capture metrics and errors.
   * Read portfolio items:
   * portfolioItems = await Portfolio.find({}, { stockName: 1, country: 1 }).lean();
     + If read fails -> log error + return early stats.
     + If no items -> return stats (no work).
   * Group unique (stockName, country) pairs into groups.
   * For each group:
     + Initialize seenUrls (Set) and seenProviderIds (Set) — these dedupe within a single group execution to avoid duplicates across providers this run.
     + fetchedCountForGroup = 0
     + For each provider in PROVIDERS (order matters — provider priority):
       - Skip provider if p.enabled is false.
       - Call const list = await p.module.search(stockName, country, limitPerTicker) — provider must return normalized list of items.
       - stats.totalFetched += list.length
       - For each item in list:
         * Deduplicate using provider+providerId or url using seen\* sets; skip duplicates.
         * Call await upsertArticle(stockName, item).
         * If inserted is true → stats.totalSaved += 1.
         * Increment fetchedCount and break if reached limitPerTicker.
       - Provider errors are caught, logged to stats.errors and the loop continues to next provider.
     + End providers loop.
   * After all groups processed, log stats and return it.

**DB touched:** Portfolio (read), Article (updateOne upserts).

**controller/newsScheduler.js — scheduler flow**

1. startNewsScheduler():
   * Reads CRON\_SCHEDULE env (default "0 0 \* \* \*" — daily at 00:00 UTC).
   * Prevents double-scheduling if task already exists.
   * Calls cron.schedule(schedule, async () => { await runNewsFetch(); }).
   * Returns the task reference.
2. stopNewsScheduler() stops and clears task.

**Note:** The cron job calls runNewsFetch() and waits for it to finish. There is no persistence of job status — the result is only logged.

**provider/marketauxProvider.js**

You indicated this exists but did not share it. Based on usage we expect search(stockName, country, limit) to return normalized items with fields used in upsertArticle:

* title, url, source, provider, providerId, publishedAt, raw\_text, entities, raw\_payload, country, language.

If you want I can inspect that provider file next to confirm fields and error modes.

**Which endpoints touch which DB collections**

* GET /user/news → reads Article
* POST /user/news/fetch-now → reads Portfolio, writes/upserts Article
* Cron job → same as fetch-now
* POST /user/news/:id/summerize and /:id/analyze → reads + updates Article (auth required)

**Response shapes (examples)**

**GET list**

200 {

"ok": true,

"items": [

{

"id": "64f...abc",

"title": "Reliance opens new store",

"source": "EconomicTimes",

"url": "https://.../article",

"publishedAt": "2025-09-30T12:34:56.000Z",

"provider": "marketaux"

}

]

}

**POST /fetch-now**

200 { "ok": true, "message": "News fetch started/completed" }

500 { "ok": false, "error": "Fetch failed" }

**POST /:articleId/summerize** (example)

200 {

"ok": true,

"articleId": "64f...abc",

"summary": "Short summary...",

"key\_facts": [...]

}

**Potential issues / edge cases / gotchas (prioritized)**

1. **Unprotected POST /fetch-now**
   * Anyone can trigger a full fetch. That could lead to abuse (DoS, excessive API costs for providers) if exposed publicly.
   * **Fix:** Protect with authMiddleware + role check (admin) or add an API token. Quick snippet:
   * router.post("/fetch-now", authMiddleware, adminOnly, fetchNow);

Or require a secret in body/headers and verify process.env.FETCH\_NOW\_SECRET.

1. **Synchronous / blocking behavior on manual fetch**
   * fetchNow awaits runNewsFetch and returns only after it completes. If fetching is slow, clients will wait a long time (but server.setTimeout=5min mitigates socket timeout). Better approach: enqueue background job and return immediately (202 Accepted + job id). If you want to keep current behavior, that's OK but consider rate-limiting.
2. **updateOne(..., { $setOnInsert: doc }, { upsert: true }) semantics**
   * This **only sets fields on insert**. If provider returns enriched info later, existing documents will *not be updated* with new content. That might be intended, but often you want to update some fields (e.g., raw\_payload, fetchedAt, or entities) even if doc exists.
   * **Fix:** Use findOneAndUpdate(filter, { $set: <fields-to-update>, $setOnInsert: <fields-for-new-doc> }, { upsert:true, returnDocument:'after' }) to both insert and update chosen fields.
3. **Upsert result detection is fragile**
   * The code checks several driver-specific properties (up.upsertedId || up.upsertedCount || up.nUpserted) — these vary by MongoDB driver version. Safer to check up.matchedCount / up.modifiedCount or use findOneAndUpdate and inspect returned document presence.
4. **Deduplication scope is per-run-per-group only**
   * seenUrls and seenProviderIds prevent duplicates within the same runNewsFetch() execution for a group, but do not prevent duplicates across runs if the same article appears later. The upsert filter mitigates duplicates across runs if providerId or url present — good. But when using fallback stockName+title+publishedAt filter, slight variations in title or timestamp could create duplicates.
5. **No DB indexes mentioned**
   * To make Article.find({stockName}).sort({publishedAt:-1}) fast, ensure index on { stockName: 1, publishedAt: -1 }. Also add unique indexes for dedupe: { provider:1, providerId:1 } unique partial, and { url:1 } unique partial.
6. **Provider errors swallowed but only logged**
   * Provider errors are added to stats.errors but there is no backoff or retry. If a provider fails transiently, you may want to retry a bounded number of times.
7. **Timezone & publishedAt**
   * You rely on publishedAt from provider. Ensure consistent UTC storage (Date objects). Cron default is "0 0 \* \* \*" (interpreted by node-cron as server local? node-cron docs: schedule uses system timezone unless configured). If your team expects 00:00 IST, note difference.
8. **Large portfolio size / rate limits**
   * If Portfolio contains many tickers, this will make many provider calls and possibly exceed provider rate limits or take long. Consider batching, rate limiting, or spreading work over multiple cron invocations.
9. **CORS wide-open**
   * app.use(cors()) permits all origins. If you want to restrict frontend, configure allowed origin(s).
10. **Exposed internal details in logs**
    * The fetcher prints full error messages. That’s fine for debugging but sanitize before shipping to a public log system.
11. **No pagination support for GET**
    * Only limit exists; no page or cursor. If there are many articles, you may want cursor/pagination.

**Quick suggested improvements & minimal code snippets**

**1) Protect fetch-now (quick)**

// routes/newsRoutes.js

import { authMiddleware } from "../middleware/auth.js";

import { requireRole } from "../middleware/roles.js"; // simple function that checks user role

router.post("/fetch-now", authMiddleware, requireRole("admin"), fetchNow);

Or header secret:

router.post("/fetch-now", (req,res,next)=>{

if (req.headers["x-fetch-secret"] !== process.env.FETCH\_NOW\_SECRET) return res.status(401).json({error:'unauth'});

next();

}, fetchNow);

**2) Make upsert update some fields when article exists (safer)**

// replace updateOne with findOneAndUpdate so we can both update and insert

const up = await Article.findOneAndUpdate(

filter,

{

$set: {

fetchedAt: new Date(),

// maybe update source/entities if you want latest enrichment:

source: doc.source,

raw\_text: doc.raw\_text,

},

$setOnInsert: doc

},

{ upsert: true, returnDocument: 'after' } // return the document after upsert

);

const inserted = up.createdAt && up.createdAt.getTime() === up.fetchedAt.getTime(); // or use flags

(You can adjust which fields to update.)

**3) Add indexes (Mongo shell / mongoose)**

// ensure in Article model definition (one-time)

articleSchema.index({ provider: 1, providerId: 1 }, { unique: true, partialFilterExpression: { provider: { $exists: true }, providerId: { $exists: true } } });

articleSchema.index({ url: 1 }, { unique: true, partialFilterExpression: { url: { $type: "string" } } });

articleSchema.index({ stockName: 1, publishedAt: -1 });

**4) Make fetch-now respond fast (background)**

If you prefer immediate response:

export const fetchNow = async (req, res) => {

try {

runNewsFetch(req.body || {}).then(stats => console.log('fetch completed', stats)).catch(err => console.error(err));

return res.status(202).json({ ok: true, message: "Fetch queued" });

} catch (err) {

return res.status(500).json({ ok: false, error: err.message });

}

};

**Performance considerations & limits**

* If Portfolio is large, the fetch could do N providers × M tickers requests — consider:
  + Capping tickers per run.
  + Spreading over multiple cron runs (rotate subsets).
  + Use a job queue (Bull, RabbitMQ) to parallelize with concurrency control.
* Monitor provider quota & add retry/backoff for 429s.

**Security checklist (quick)**

* Protect /fetch-now.
* Validate and sanitize stock query in GET /user/news (currently uppercase but OK).
* Rate limit public endpoints to prevent abuse.
* Restrict CORS to known origins in production.
* Limit exports of raw text if that contains sensitive info (PII).

**Where you’ll likely change if you extend features**

* Provider normalization: ensure all providers return a consistent schema, especially provider and providerId to rely on provider dedupe.
* Upsert behavior: decide whether to update or only insert once. Currently it inserts once and never modifies existing Article fields except by other controllers (summarize/analyze).
* Scheduler options: expose endpoint to toggle scheduler or to run incremental fetching for a subset of tickers.

**Short prioritized action list for small convincing changes**

1. **Protect /fetch-now** (high impact, small change).
2. **Add indexes** for Article (fast DB migration).
3. **Change updateOne to findOneAndUpdate** to allow updating selected fields on repeat fetches (safer data quality).
4. **Return 202 for manual fetch** (if you want UI not to hang).
5. **Add rate limiting** / throttle to the manual trigger.

If you want, next I can:

* Inspect provider/marketauxProvider.js to confirm normalization and error modes.
* Inspect models/Article.js and models/Portfolio.js to recommend exact index definitions and show precise migration steps.
* Provide a patch diff for the upsert change and example mongoose index creation script.

Tell me which of the next steps above you want me to do and I’ll apply them to the code you gave (or produce code snippets / diffs).

**------------------------------------------------------------------------------------**

Great — I read providers/marketauxProvider.js and rechecked the earlier fixes. Below I give a **concise, tested plan** plus exact, copy-paste code replacements for the two files you need to change. For each change I explain **what your code did before**, **what it will do after**, and **why that fixes your two issues** (1: new news not saved, 2: old news being saved).

Important: Marketaux supports a published\_after query parameter so the provider can ask the upstream API for only recent articles — we'll use that (docs/examples). ([Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com))

**Summary of the plan (short)**

1. Tell the provider to return only recent articles (use published\_after) — reduces old 2022 items. (change providers/marketauxProvider.js) ([Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com))
2. Make the orchestrator pass that since date (configurable via FETCH\_RECENT\_DAYS) and keep a defensive check client-side. (change controller/newsFetcher.js)
3. Make the upsert behavior more robust: insert new articles; if an article exists, update key metadata **only if** the incoming publishedAt is newer. This prevents silent no-ops that stop new articles from being saved. (change controller/newsFetcher.js)

Do these three and your two problems will be solved in most normal cases.

**1) Change provider to accept published\_after (file: providers/marketauxProvider.js)**

**Why:** Marketaux supports published\_after so we should ask the API for recent articles instead of fetching everything then filtering later. That reduces both bandwidth and likelihood of old 2022 articles being returned. Example usage of published\_after is shown in Marketaux docs / examples. ([Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com))

**What to replace** — full new file content (replace the current providers/marketauxProvider.js):

// providers/marketauxProvider.js

import axios from "axios";

import dotenv from "dotenv";

dotenv.config();

const MARKET\_TOKEN = process.env.MARKETAUX\_API\_KEY;

const MARKET\_BASE = "https://api.marketaux.com/v1";

const DEFAULT\_LIMIT = parseInt(process.env.FETCH\_TOP\_N\_PER\_TICKER || "10", 10);

if (!MARKET\_TOKEN) {

console.warn("MARKETAUX\_API\_KEY not set in .env || Please set it");

}

// rawSearch now accepts an optional publishedAfter (ISO string) to pass to Marketaux

async function rawSearch(q, limit = DEFAULT\_LIMIT, countries = "in", publishedAfter = null) {

const url = `${MARKET\_BASE}/news/all`;

const params = {

api\_token: MARKET\_TOKEN,

search: q,

limit,

countries,

filter\_entities: false,

group\_similar: true,

};

// add published\_after only if provided (Marketaux expects ISO-ish string / timestamp)

if (publishedAfter) {

params.published\_after = publishedAfter;

}

const res = await axios.get(url, { params, timeout: 15000 });

return res.data?.data ?? res.data ?? [];

}

/\*\*

\* Normalize Marketaux article object to our internal shape:

\* {

\* title, url, source, provider: "marketaux", providerId, publishedAt,

\* raw\_text, entities, raw\_payload, country, language

\* }

\*

\* New param `publishedAfter` is optional ISO string passed down to rawSearch so API returns recent articles only.

\*/

export async function search(stockName, country = "IN", limit = DEFAULT\_LIMIT, publishedAfter = null) {

if (!MARKET\_TOKEN) return [];

const countryParam = (country || "IN").toLowerCase();

const raw = await rawSearch(stockName, limit, countryParam, publishedAfter);

return (Array.isArray(raw) ? raw : []).map(a => {

const providerId = a.uuid || a.id || a.uid || null;

return {

title: a.title || a.description || a.headline || "untitled",

url: a.url || null,

source: a.source || a.source\_name || "marketaux",

provider: "marketaux",

providerId,

// keep publishedAt as a Date when present (upstream uses `published\_at`)

publishedAt: a.published\_at ? new Date(a.published\_at) : null,

raw\_text: a.description || a.snippet || "",

entities: Array.isArray(a.entities) ? a.entities.map(e => (e.name || e)).filter(Boolean) : [],

raw\_payload: a,

country: (a.country || country || "").toString().toUpperCase(),

language: a.language || ""

};

});

}

**Before:** your provider always called /news/all with search, limit, countries and returned whatever Marketaux returned. No upstream date filter was used.

**After:** provider can accept publishedAfter (ISO string) and adds published\_after to the Marketaux query when present — so Marketaux returns only articles published after that timestamp (supported by Marketaux examples). This will dramatically reduce old results returned by the provider. ([Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com))

**2) Change orchestrator to compute sinceDate, pass to provider and defensively filter (file: controller/newsFetcher.js)**

You will replace the provider call block and the upsertArticle function in newsFetcher.js. I give two precise replacements: (A) compute sinceDate (configurable via FETCH\_RECENT\_DAYS) and pass it to p.module.search(...); (B) replace upsertArticle with the robust variant that inserts or updates when incoming publishedAt is newer.

**Add / compute sinceDate and change provider call (inside runNewsFetch loop)**

**Find** the code block where you call const list = await p.module.search(stockName, country, limitPerTicker); — replace that block with the new one below.

**Replace the old provider call code (approx):**

// module.search returns normalized list

const list = await p.module.search(stockName, country, limitPerTicker);

stats.totalFetched += list.length;

for (const item of list) {

// ...

**With this new block:**

// --- compute the optional since cutoff (only if FETCH\_RECENT\_DAYS > 0) ---

const lookbackDays = parseInt(process.env.FETCH\_RECENT\_DAYS || "3", 10);

// treat <=0 as "no date filter" — pass null to provider to request everything

const sinceDate = lookbackDays > 0 ? new Date(Date.now() - lookbackDays \* 24 \* 60 \* 60 \* 1000) : null;

const publishedAfterParam = sinceDate ? sinceDate.toISOString() : null;

// module.search now accepts (stockName, country, limit, publishedAfter)

let list = await p.module.search(stockName, country, limitPerTicker, publishedAfterParam);

// defensive: ensure list is an array

if (!Array.isArray(list)) list = [];

// If we did pass a sinceDate, additionally filter client-side for safety

const originalCount = list.length;

if (sinceDate) {

list = list.filter(it => {

try {

const date = it && it.publishedAt ? new Date(it.publishedAt) : null;

return date instanceof Date && !isNaN(date) && date >= sinceDate;

} catch (e) {

return false;

}

});

if (originalCount !== list.length) {

console.log(`Provider ${p.name} returned ${originalCount} items, ${originalCount - list.length} dropped (older than ${sinceDate.toISOString()})`);

}

}

stats.totalFetched += list.length;

for (const item of list) {

// ...

**What changed (explanation):**

* We use FETCH\_RECENT\_DAYS env var (default 3) to create sinceDate.
* If FETCH\_RECENT\_DAYS <= 0 we treat that as "no filter" and pass null, so provider returns everything (same behavior as before).
* We pass sinceDate.toISOString() to marketauxProvider.search(...) so the provider queries Marketaux with published\_after. This reduces old items returned by the upstream API. ([Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com))
* We still do a defensive client-side filter to handle providers that may return slightly malformed dates.

**Why this helps:**

* If Marketaux supports published\_after (it does), we ask it for only recent articles — less old noise and fewer 2022 items will be returned. ([Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com))

**Replace upsertArticle with robust insert-or-update (same file)**

**Find old upsertArticle and replace it with the new function below.** This is the version I gave earlier but improved for clarity: it **inserts** if no existing doc is found; if an article exists, it **updates selected fields only if the incoming publishedAt is newer** (so you don’t clobber analysis fields like summary), and otherwise does nothing.

**Replace the old function with:**

// helper: upsert single normalized article doc (robust insert-or-update)

async function upsertArticle(stockName, normalized) {

const doc = {

stockName,

title: normalized.title,

url: normalized.url || null,

source: normalized.source,

provider: normalized.provider,

providerId: normalized.providerId || null,

publishedAt: normalized.publishedAt ? new Date(normalized.publishedAt) : new Date(),

fetchedAt: new Date(),

summary: "",

key\_facts: [],

rationale: "",

impact\_label: null,

impact\_confidence: 0,

raw\_text: normalized.raw\_text || "",

entities: normalized.entities || [],

raw\_payload: normalized.raw\_payload || {},

country: (normalized.country || "").toString().toUpperCase(),

language: normalized.language || ""

};

// dedupe order: provider+providerId -> url -> stockName+title+publishedAt

let filter = null;

if (doc.provider && doc.providerId) filter = { provider: doc.provider, providerId: doc.providerId };

else if (doc.url) filter = { url: doc.url };

else filter = { stockName: doc.stockName, title: doc.title, publishedAt: doc.publishedAt };

// First, check if an existing doc matches the filter

const existing = await Article.findOne(filter).lean();

if (!existing) {

// No existing doc — insert

const res = await Article.updateOne(filter, { $setOnInsert: doc }, { upsert: true });

const inserted = (res.upsertedId || res.upsertedCount || res.nUpserted) ? true : false;

return { inserted, doc };

} else {

// existing doc found — only update if incoming publishedAt is newer

const incomingPublished = doc.publishedAt instanceof Date ? doc.publishedAt : new Date(doc.publishedAt);

const existingPublished = existing.publishedAt ? new Date(existing.publishedAt) : null;

const incomingIsNewer = !existingPublished || (incomingPublished && incomingPublished > existingPublished);

if (incomingIsNewer) {

const fieldsToUpdate = {

publishedAt: incomingPublished,

fetchedAt: new Date(),

// update metadata fields (but intentionally do NOT overwrite summary/key\_facts)

...(doc.raw\_text ? { raw\_text: doc.raw\_text } : {}),

...(Array.isArray(doc.entities) && doc.entities.length ? { entities: doc.entities } : {}),

source: doc.source,

provider: doc.provider,

providerId: doc.providerId || null

};

await Article.updateOne(filter, { $set: fieldsToUpdate });

return { inserted: false, updated: true, doc: fieldsToUpdate };

} else {

// nothing to do

return { inserted: false, updated: false, doc: existing };

}

}

}

**Before:** updateOne(filter, { $setOnInsert: doc }, { upsert: true }) only sets data on insert, never updates existing documents. If the provider returned a newer publishedAt or different providerId/url that should have resulted in an insert, the old logic could still fail to add or update correctly — producing the symptom "new news not saved".

**After:** the function:

* inserts when no document matches the dedupe filter, and
* updates selected metadata only if incoming publishedAt is newer.  
  This ensures truly new items get inserted and that newer provider metadata replaces older metadata when appropriate — solving the "new news not saved" issue.

**.env and behavioral notes**

Add/tune this variable in your .env:

# number of days back to fetch; set to 3 for last 3 days, set to 0 to disable upstream date filtering

FETCH\_RECENT\_DAYS=3

* FETCH\_RECENT\_DAYS=3 means provider is asked published\_after=ISO( now - 3 days ). That avoids old 2022 items.
* FETCH\_RECENT\_DAYS=0 is treated here as **no upstream date filtering** (keeps previous behavior). Use >0 to enable recency filtering.

Also keep FETCH\_TOP\_N\_PER\_TICKER as you already have.

**Why these fix your reported problems (explicit)**

* Problem **(2) old news being fetched**: Marketaux can return old articles unless told otherwise. By passing published\_after (via provider change) we ask Marketaux to return only recent articles (e.g., last 3 days). Citations show Marketaux supports published\_after. ([Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com))
* Problem **(1) new news not being fetched/saved**: your old upsert used $setOnInsert only and never updated existing docs; if the dedupe filter matched an old doc or if provider results had different metadata, new items could be skipped. The new upsert logic inserts when missing and updates metadata only when incoming article is newer — so new articles will be persisted. This removes the "silent no-op" scenario.

**How to apply & test (quick checklist)**

1. Replace providers/marketauxProvider.js with the provider snippet above.
2. Replace the provider-call block and upsertArticle function in controller/newsFetcher.js with the snippets above. (If you prefer, send me the exact newsFetcher.js file and I’ll return a full file patch.)
3. Add FETCH\_RECENT\_DAYS=3 to .env and restart your server.
4. Watch server logs for lines like:
   * Provider marketaux returned X items, Y dropped (older than 2025-10-02T...) — confirms filtering worked.
   * Orchestrated news fetcher finished. stats: — check totalSaved.
5. Trigger manual fetch (if you allow manual POST /user/news/fetch-now) and check Article collection for new publishedAt values and new docs.

**Extra checks & recommended next steps (optional)**

* If you want maximum freshness, add a sort param (if Marketaux supports it) to request newest-first; docs suggest date filters are available; sorting defaults are usually newest-first but I kept changes minimal and robust (we do client-side filtering & use published\_after). If you want, I can add sort param after checking exact Marketaux param name.
* Add indexes on Article for { provider, providerId } and { url } and { stockName, publishedAt: -1 } to improve query/upsert performance.
* If you use Marketaux free plan, note the free plan returns fewer articles per request; you may need higher plan or increase limit.

**Citations (Marketaux docs / examples)**

* Marketaux API supports published\_after parameter and published\_at response field — examples in docs. ([Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com))

If you want I will:

* produce a **full patched newsFetcher.js** file with the new runNewsFetch() containing both the provider-call changes and the upsertArticle replacement (so you can copy-paste the whole file), or
* produce a **git-style patch** with exact line replacements if you paste the current newsFetcher.js (so I can match line numbers precisely).

Which would you prefer?

**------------------------------------------------------------------------------------**

**2) Change orchestrator to compute sinceDate, pass to provider and defensively filter (file: controller/newsFetcher.js)**

You will replace the provider call block and the upsertArticle function in newsFetcher.js. I give two precise replacements: (A) compute sinceDate (configurable via FETCH\_RECENT\_DAYS) and pass it to p.module.search(...); (B) replace upsertArticle with the robust variant that inserts or updates when incoming publishedAt is newer.

**Add / compute sinceDate and change provider call (inside runNewsFetch loop)**

**Find** the code block where you call const list = await p.module.search(stockName, country, limitPerTicker); — replace that block with the new one below.

**Replace the old provider call code (approx):**

// module.search returns normalized list

const list = await p.module.search(stockName, country, limitPerTicker);

stats.totalFetched += list.length;

for (const item of list) {

// ...

**With this new block:**

// --- compute the optional since cutoff (only if FETCH\_RECENT\_DAYS > 0) ---

const lookbackDays = parseInt(process.env.FETCH\_RECENT\_DAYS || "3", 10);

// treat <=0 as "no date filter" — pass null to provider to request everything

const sinceDate = lookbackDays > 0 ? new Date(Date.now() - lookbackDays \* 24 \* 60 \* 60 \* 1000) : null;

const publishedAfterParam = sinceDate ? sinceDate.toISOString() : null;

// module.search now accepts (stockName, country, limit, publishedAfter)

let list = await p.module.search(stockName, country, limitPerTicker, publishedAfterParam);

// defensive: ensure list is an array

if (!Array.isArray(list)) list = [];

// If we did pass a sinceDate, additionally filter client-side for safety

const originalCount = list.length;

if (sinceDate) {

list = list.filter(it => {

try {

const date = it && it.publishedAt ? new Date(it.publishedAt) : null;

return date instanceof Date && !isNaN(date) && date >= sinceDate;

} catch (e) {

return false;

}

});

if (originalCount !== list.length) {

console.log(`Provider ${p.name} returned ${originalCount} items, ${originalCount - list.length} dropped (older than ${sinceDate.toISOString()})`);

}

}

stats.totalFetched += list.length;

for (const item of list) {

// ...

**What changed (explanation):**

* We use FETCH\_RECENT\_DAYS env var (default 3) to create sinceDate.
* If FETCH\_RECENT\_DAYS <= 0 we treat that as "no filter" and pass null, so provider returns everything (same behavior as before).
* We pass sinceDate.toISOString() to marketauxProvider.search(...) so the provider queries Marketaux with published\_after. This reduces old items returned by the upstream API. [Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com)
* We still do a defensive client-side filter to handle providers that may return slightly malformed dates.

**Why this helps:**

* If Marketaux supports published\_after (it does), we ask it for only recent articles — less old noise and fewer 2022 items will be returned. [Free API Hub](https://freeapihub.com/apis/marketaux?utm_source=chatgpt.com)

**Replace upsertArticle with robust insert-or-update (same file)**

**Find old upsertArticle and replace it with the new function below.** This is the version I gave earlier but improved for clarity: it **inserts** if no existing doc is found; if an article exists, it **updates selected fields only if the incoming publishedAt is newer** (so you don’t clobber analysis fields like summary), and otherwise does nothing.

**Replace the old function with:**

// helper: upsert single normalized article doc (robust insert-or-update)

async function upsertArticle(stockName, normalized) {

const doc = {

stockName,

title: normalized.title,

url: normalized.url || null,

source: normalized.source,

provider: normalized.provider,

providerId: normalized.providerId || null,

publishedAt: normalized.publishedAt ? new Date(normalized.publishedAt) : new Date(),

fetchedAt: new Date(),

summary: "",

key\_facts: [],

rationale: "",

impact\_label: null,

impact\_confidence: 0,

raw\_text: normalized.raw\_text || "",

entities: normalized.entities || [],

raw\_payload: normalized.raw\_payload || {},

country: (normalized.country || "").toString().toUpperCase(),

language: normalized.language || ""

};

// dedupe order: provider+providerId -> url -> stockName+title+publishedAt

let filter = null;

if (doc.provider && doc.providerId) filter = { provider: doc.provider, providerId: doc.providerId };

else if (doc.url) filter = { url: doc.url };

else filter = { stockName: doc.stockName, title: doc.title, publishedAt: doc.publishedAt };

// First, check if an existing doc matches the filter

const existing = await Article.findOne(filter).lean();

if (!existing) {

// No existing doc — insert

const res = await Article.updateOne(filter, { $setOnInsert: doc }, { upsert: true });

const inserted = (res.upsertedId || res.upsertedCount || res.nUpserted) ? true : false;

return { inserted, doc };

} else {

// existing doc found — only update if incoming publishedAt is newer

const incomingPublished = doc.publishedAt instanceof Date ? doc.publishedAt : new Date(doc.publishedAt);

const existingPublished = existing.publishedAt ? new Date(existing.publishedAt) : null;

const incomingIsNewer = !existingPublished || (incomingPublished && incomingPublished > existingPublished);

if (incomingIsNewer) {

const fieldsToUpdate = {

publishedAt: incomingPublished,

fetchedAt: new Date(),

// update metadata fields (but intentionally do NOT overwrite summary/key\_facts)

...(doc.raw\_text ? { raw\_text: doc.raw\_text } : {}),

...(Array.isArray(doc.entities) && doc.entities.length ? { entities: doc.entities } : {}),

source: doc.source,

provider: doc.provider,

providerId: doc.providerId || null

};

await Article.updateOne(filter, { $set: fieldsToUpdate });

return { inserted: false, updated: true, doc: fieldsToUpdate };

} else {

// nothing to do

return { inserted: false, updated: false, doc: existing };

}

}

}

**Before:** updateOne(filter, { $setOnInsert: doc }, { upsert: true }) only sets data on insert, never updates existing documents. If the provider returned a newer publishedAt or different providerId/url that should have resulted in an insert, the old logic could still fail to add or update correctly — producing the symptom "new news not saved".

**After:** the function:

* inserts when no document matches the dedupe filter, and
* updates selected metadata only if incoming publishedAt is newer.  
  This ensures truly new items get inserted and that newer provider metadata replaces older metadata when appropriate — solving the "new news not saved" issue.