Filtered endpoint

@app.route('/api/reits/advanced-filter', methods=['GET'])

def get\_advanced\_filtered\_reits():

    """

    DEFINITIVE ENDPOINT V3: Reconstructs a full time-series for each REIT before

    calculation. This ensures YoY growth is always calculated over a true one-year

    period, correctly handling sparse data.

    """

    app.logger.info(f"Request received for DEFINITIVE PANDAS-BASED filter with args: {request.args}")

    args = request.args

    property\_type = args.get('property\_type')

    min\_revenue\_growth = args.get('min\_revenue\_growth', type=float)

    min\_ffo\_growth = args.get('min\_ffo\_growth', type=float)

    min\_operating\_margin = args.get('min\_operating\_margin', type=float)

    max\_revenue\_growth = args.get('max\_revenue\_growth', type=float)

    max\_ffo\_growth = args.get('max\_ffo\_growth', type=float)

    max\_operating\_margin = args.get('max\_operating\_margin', type=float)

    try:

        with db.engine.connect() as conn:

            # Step 1 & 2: Data fetching remains the same

            params = {}

            sql\_tickers = "SELECT Ticker, Company\_Name, Business\_Description, Website FROM reit\_business\_data WHERE 1=1"

            if property\_type:

                sql\_tickers += " AND Property\_Type LIKE :property\_type"

                params['property\_type'] = f"%{property\_type}%"

            candidate\_df = pd.read\_sql(text(sql\_tickers), conn, params=params)

            if candidate\_df.empty:

                return jsonify({"reits": []})

            candidate\_tickers = tuple(candidate\_df['Ticker'].tolist())

            sql\_financials = text("""

                SELECT ticker, TRIM(line\_item) as line\_item, fiscal\_year, fiscal\_quarter, value

                FROM reit\_income\_statement

                WHERE TRIM(line\_item) IN ('Total Revenue', 'Operating Income', 'FFO')

                AND ticker IN :tickers

                AND fiscal\_quarter IS NOT NULL         -- ADD THIS

            """)

            financials\_df = pd.read\_sql(sql\_financials, conn, params={"tickers": candidate\_tickers})

            financials\_df['value'] = financials\_df['value'].replace(0, np.nan)

        # --- Step 3: Calculate Metrics with Time-Aware Logic ---

        results = []

        app.logger.info("--- STARTING METRIC CALCULATION ---")

        for ticker, group in financials\_df.groupby('ticker'):

            group = group.sort\_values(by=['fiscal\_year', 'fiscal\_quarter'], ascending=True)

            def \_period\_index(df):

                if df.empty:

                    return pd.PeriodIndex([], freq='Q-DEC')

                return pd.PeriodIndex(year=df['fiscal\_year'].astype(int),

                                    quarter=df['fiscal\_quarter'].astype(int),

                                    freq='Q-DEC')

            # Split by line item

            revenue\_data   = group[group['line\_item'] == 'Total Revenue'].copy()

            op\_income\_data = group[group['line\_item'] == 'Operating Income'].copy()

            ffo\_data       = group[group['line\_item'] == 'FFO'].copy()

            # Build a MASTER index spanning the union of all quarters seen for this ticker

            idxs = []

            for df\_part in (revenue\_data, op\_income\_data, ffo\_data):

                pi = \_period\_index(df\_part)

                if len(pi) > 0:

                    idxs.append(pi)

            if not idxs:

                # No usable data at all; skip cleanly

                results.append({

                    'Ticker': ticker,

                    'operating\_margin': None,

                    'avg\_revenue\_yoy\_growth': None,

                    'avg\_ffo\_yoy\_growth': None

                })

                continue

            start = min(pi.min() for pi in idxs)

            end   = max(pi.max() for pi in idxs)

            master\_index = pd.period\_range(start=start, end=end, freq='Q-DEC')

            def \_series\_on\_master(df):

                if df.empty:

                    return pd.Series(index=master\_index, dtype='float64')

                s = pd.Series(df['value'].values, index=\_period\_index(df))

                # Ensure numeric and keep NaNs (no filling)

                s = pd.to\_numeric(s, errors='coerce')

                return s.reindex(master\_index)

            # Aligned series on the same master timeline

            rev\_series = \_series\_on\_master(revenue\_data)

            op\_series  = \_series\_on\_master(op\_income\_data)

            ffo\_series = \_series\_on\_master(ffo\_data)

            app.logger.info(f"[{ticker}] Master tail(8) periods: {list(master\_index[-8:])}")

            app.logger.info(f"[{ticker}] FFO tail(8) values: {ffo\_series.tail(8).tolist()}")

            def \_ttm\_sum\_strict(s):

                last4 = s.tail(4)

                # If any quarter missing, return None (avoid partial sums)

                if last4.isna().any() or len(last4) < 4:

                    return None

                return float(last4.sum())

            # Compute TTM & Op margin strictly

            ttm\_revenue = \_ttm\_sum\_strict(rev\_series)

            ttm\_op\_inc  = \_ttm\_sum\_strict(op\_series)

            operating\_margin = (ttm\_op\_inc / ttm\_revenue) if (ttm\_revenue not in (None, 0) and ttm\_op\_inc is not None) else None

            def get\_strict\_yoy\_growth\_metrics\_on\_master(s):

                # Need at least 8 quarters on the \*master\* axis

                if len(s.dropna()) == 0 or len(s) < 8:

                    return None, pd.Series([np.nan] \* 4, index=s.tail(4).index)

                recent = s.tail(8)

                yoy = recent.pct\_change(periods=4, fill\_method=None)  # no filling

                last4 = yoy.tail(4)

                if last4.isnull().any():

                    return None, last4

                return float(last4.mean()), last4

            avg\_revenue\_yoy\_growth, rev\_growths\_for\_log = get\_strict\_yoy\_growth\_metrics\_on\_master(rev\_series)

            avg\_ffo\_yoy\_growth,     ffo\_growths\_for\_log = get\_strict\_yoy\_growth\_metrics\_on\_master(ffo\_series)

            # --- Logging stays the same but now uses the aligned series ---

            app.logger.info(f"--- Processing Ticker: {ticker} ---")

            app.logger.info(f"[{ticker}] Raw Revenue points for TTM: {rev\_series.tail(4).tolist()}")

            app.logger.info(f"[{ticker}] Raw OpIncome points for TTM: {op\_series.tail(4).tolist()}")

            app.logger.info(f"[{ticker}] Individual Revenue YoY Growths for Avg: "

                            f"{[f'{x:.2%}' if pd.notna(x) else 'N/A' for x in rev\_growths\_for\_log]}")

            app.logger.info(f"[{ticker}] Individual FFO YoY Growths for Avg: "

                            f"{[f'{x:.2%}' if pd.notna(x) else 'N/A' for x in ffo\_growths\_for\_log]}")

            results.append({

                'Ticker': ticker,

                'operating\_margin': operating\_margin,

                'avg\_revenue\_yoy\_growth': avg\_revenue\_yoy\_growth,

                'avg\_ffo\_yoy\_growth': avg\_ffo\_yoy\_growth

            })

        app.logger.info("--- FINISHED METRIC CALCULATION ---")

        if not results:

             return jsonify({"reits": []})

        # Step 4, 5, & 6: Merging, Filtering, and Final Logging (No changes needed)

        metrics\_df = pd.DataFrame(results)

        final\_df = pd.merge(candidate\_df, metrics\_df, on='Ticker')

        final\_df = final\_df.astype(object).where(pd.notna(final\_df), None)

        filtered\_df = final\_df.copy()

        if min\_operating\_margin is not None:

            filtered\_df = filtered\_df[filtered\_df['operating\_margin'].notna() & (filtered\_df['operating\_margin'] >= min\_operating\_margin)]

        if min\_revenue\_growth is not None:

            filtered\_df = filtered\_df[filtered\_df['avg\_revenue\_yoy\_growth'].notna() & (filtered\_df['avg\_revenue\_yoy\_growth'] >= min\_revenue\_growth)]

        if min\_ffo\_growth is not None:

            filtered\_df = filtered\_df[filtered\_df['avg\_ffo\_yoy\_growth'].notna() & (filtered\_df['avg\_ffo\_yoy\_growth'] >= min\_ffo\_growth)]

        if max\_operating\_margin is not None:

            filtered\_df = filtered\_df[filtered\_df['operating\_margin'].notna() & (filtered\_df['operating\_margin'] <= max\_operating\_margin)]

        if max\_revenue\_growth is not None:

            filtered\_df = filtered\_df[filtered\_df['avg\_revenue\_yoy\_growth'].notna() & (filtered\_df['avg\_revenue\_yoy\_growth'] <= max\_revenue\_growth)]

        if max\_ffo\_growth is not None:

            filtered\_df = filtered\_df[filtered\_df['avg\_ffo\_yoy\_growth'].notna() & (filtered\_df['avg\_ffo\_yoy\_growth'] <= max\_ffo\_growth)]

        app.logger.info("--- VERIFICATION LOG (FINAL) ---")

        if filtered\_df.empty:

            app.logger.info("No REITs matched the final criteria.")

        else:

            for index, row in filtered\_df.iterrows():

                op\_margin\_str = f"{row['operating\_margin']:.2%}" if row['operating\_margin'] is not None else "N/A"

                rev\_growth\_str = f"{row['avg\_revenue\_yoy\_growth']:.2%}" if row['avg\_revenue\_yoy\_growth'] is not None else "N/A"

                ffo\_growth\_str = f"{row['avg\_ffo\_yoy\_growth']:.2%}" if row['avg\_ffo\_yoy\_growth'] is not None else "N/A"

                log\_message = (

                    f"Ticker: {row['Ticker']:<8} | "

                    f"OpMargin: {op\_margin\_str:<10} | "

                    f"AvgRevenueGrowth: {rev\_growth\_str:<10} | "

                    f"AvgFFOGrowth: {ffo\_growth\_str:<10}"

                )

                app.logger.info(log\_message)

        app.logger.info("-----------------------------")

        reits\_json = filtered\_df.to\_dict('records')

        return jsonify({"reits": reits\_json})

    except Exception as e:

        app.logger.error(f"Error in final pandas-based filter logic: {e}")

        traceback.print\_exc()

        return jsonify({"error": "A database error occurred."}), 500

Header

// Header.js – corrected and complete

import React, { useState, useEffect } from "react";

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

import axios from "axios";

import { auth } from "../firebase.js";

import { signOut } from "firebase/auth";

import { db } from "../firebase.js";

import { collection, query, where, getDocs } from "firebase/firestore";

import Sidebar from "./Sidebar.js";

const API\_BASE\_URL =

  process.env.REACT\_APP\_BACKEND\_URL || "http://127.0.0.1:5000";

// Tweak just this if the auth buttons ever shift left/right

const AUTH\_GROUP\_STYLE = { marginRight: "55px" };

const Header = ({ currentUser, userPlan, setUserPlan }) => {

  const navigate = useNavigate();

  const location = useLocation();

  /\* ─────────────────────────  Firebase / user  ───────────────────────── \*/

  const [username, setUsername] = useState("");

  const [loginHovered, setLoginHovered] = useState(false);

  useEffect(() => {

    if (!currentUser || !currentUser.emailVerified) {

      setUsername("");

      setUserPlan(null);

      return;

    }

    const fetchUserData = async () => {

      const q = query(collection(db, "users"), where("email", "==", currentUser.email));

      try {

        const querySnapshot = await getDocs(q);

        if (!querySnapshot.empty) {

          const userData = querySnapshot.docs[0].data();

          setUsername(userData.username || "");

          setUserPlan(userData.plan);

        } else {

           // This handles cases where user exists in Firebase Auth but not in your database yet

           // You might want to sign them out if their DB record is missing

           // signOut(auth); // <-- Comment this out for now

        }

      } catch (error) {

        console.error("Error fetching user data in Header:", error);

        signOut(auth);

      }

    };

    fetchUserData();

  }, [currentUser, setUserPlan, location]);

  /\* ─────────────────────────  sidebar  ───────────────────────── \*/

  const [isSidebarOpen, setIsSidebarOpen] = useState(false);

  /\* ─────────────────────────  search box  ────────────────────── \*/

  const [searchQuery, setSearchQuery] = useState("");

  const [suggestions, setSuggestions] = useState([]);

  const [isFetching, setIsFetching] = useState(false);

  // recent searches (array of {Ticker, Company\_Name})

  const [recentSearches, setRecentSearches] = useState([]);

  const [isFocused, setIsFocused] = useState(false);

  // load recents from localStorage

  useEffect(() => {

    const stored = JSON.parse(localStorage.getItem("recentSearches") || "[]");

    setRecentSearches(stored);

  }, []);

  // save a new recent (object) at top, dedupe, keep max 5

  const saveRecent = (item) => {

    const filtered = recentSearches.filter((r) => r.Ticker !== item.Ticker);

    const updated = [item, ...filtered].slice(0, 5);

    setRecentSearches(updated);

    localStorage.setItem("recentSearches", JSON.stringify(updated));

  };

  // fetch from API when user types

  useEffect(() => {

    if (!searchQuery) {

      setSuggestions([]);

      return;

    }

    let active = true;

    (async () => {

      setIsFetching(true);

      try {

        const res = await axios.get(`${API\_BASE\_URL}/api/reits`, {

          params: { search: searchQuery },

        });

        if (active) setSuggestions(res.data?.reits || []);

      } catch {

        active && setSuggestions([]);

      } finally {

        active && setIsFetching(false);

      }

    })();

    return () => {

      active = false;

    };

  }, [searchQuery]);

  // when user selects, record and navigate

  const handleSelectTicker = (item) => {

    saveRecent(item);

    setSearchQuery("");

    setSuggestions([]);

    navigate(`/reits/${item.Ticker}`);

  };

  /\* ─────────────────────────  render  ───────────────────────── \*/

  return (

    <>

      {/\* Sidebar overlay \*/}

      {isSidebarOpen && (

        <div

          onClick={() => setIsSidebarOpen(false)}

          style={{

            position: "fixed",

            top: 0,

            left: 0,

            width: "100vw",

            height: "100vh",

            backdropFilter: "blur(6px)",

            WebkitBackdropFilter: "blur(6px)",

            backgroundColor: "rgba(0,0,0,0.25)",

            zIndex: 1200,

          }}

        />

      )}

      <Sidebar isOpen={isSidebarOpen} onClose={() => setIsSidebarOpen(false)} />

      {/\* Header bar \*/}

      <nav

        style={{

          position: "fixed",

          top: 0,

          left: 0,

          width: "100%",

          height: 80,

          display: "flex",

          alignItems: "center",

          justifyContent: "space-between",

          padding: "0 20px",

          background: "#fff",

          color: "#333",

          boxShadow: "0 4px 6px rgba(0,0,0,.1)",

          zIndex: 1100,

        }}

      >

        {/\* LEFT:  hamburger + logo + search \*/}

        <div style={{ display: "flex", alignItems: "center", gap: 10 }}>

          {/\* ☰ hamburger \*/}

          <div

            onClick={() => setIsSidebarOpen((o) => !o)}

            style={{

              marginLeft: 10,

              fontSize: 26,

              lineHeight: 0,

              cursor: "pointer",

              userSelect: "none",

              color: "#5A153D",

            }}

          >

            &#9776;

          </div>

          {/\* logo \*/}

          <img

            src="/logo-crop.PNG"

            alt="Viserra Logo"

            style={{ height: 60, cursor: "pointer" }}

            onClick={() => navigate("/")}

          />

          {/\* search box \*/}

          <div style={{ width: 320, position: "relative" }}>

            <input

              value={searchQuery}

              onChange={(e) => setSearchQuery(e.target.value)}

              onFocus={() => setIsFocused(true)}

              onBlur={() => setTimeout(() => setIsFocused(false), 200)}

              placeholder="Search REIT ticker…"

              style={{

                width: "100%",

                padding: "10px 14px",

                fontSize: "1rem",

                borderRadius: 4,

                border: "1px solid #ccc",

              }}

            />

            {(

              // show API suggestions if typing

              searchQuery ||

              // otherwise, if focused and have recents, show them

              (!searchQuery && isFocused && recentSearches.length > 0)

            ) && (

              <div

                style={{

                  position: "absolute",

                  top: 46,

                  left: 0,

                  width: "108.5%",

                  maxHeight: 260,

                  overflowY: "auto",

                  background: "#fff",

                  border: "1px solid #ccc",

                  borderRadius: 4,

                  zIndex: 1300,

                }}

              >

                {searchQuery && isFetching && (

                  <p style={{ margin: 8, fontSize: ".9rem", color: "#555" }}>

                    Loading…

                  </p>

                )}

                {searchQuery &&

                  !isFetching &&

                  suggestions.length === 0 && (

                    <p style={{ margin: 8, fontSize: ".9rem" }}>

                      No match for <strong>{searchQuery}</strong>

                    </p>

                  )}

                {/\* unified list: either API results or recents (filtered) \*/}

                {!searchQuery && isFocused && recentSearches.filter(r => r.Ticker && r.Company\_Name).length > 0 && (

                  <div style={{ padding: "8px 12px", color: "#000", fontWeight: 600 }}>

                    Recent

                  </div>

                )}

                {(searchQuery

                  ? suggestions

                  : recentSearches.filter((r) => r.Ticker && r.Company\_Name)

                ).map((r) => (

                  <div

                    key={r.Ticker}

                    onClick={() => handleSelectTicker(r)}

                    style={{

                      padding: "8px 12px",

                      cursor: "pointer",

                      borderBottom: "1px solid #eee",

                      display: "flex",

                      alignItems: "center",

                    }}

                    onMouseEnter={(e) =>

                      (e.currentTarget.style.background = "#faf0fb")

                    }

                    onMouseLeave={(e) =>

                      (e.currentTarget.style.background = "transparent")

                    }

                  >

                    {/\* Ticker in purple \*/}

                    <span

                      style={{

                        color: "#5A153D",

                        fontWeight: 600,

                      }}

                    >

                      {r.Ticker}

                    </span>

                    {/\* Company name in black, safe-split \*/}

                    <span style={{ color: "#000", marginLeft: 8 }}>

                      {(r.Company\_Name || "").split(" (")[0]}

                    </span>

                  </div>

                ))}

              </div>

            )}

          </div>

        </div>

        {/\* RIGHT:  auth buttons / greeting \*/}

        <div

          style={{

            display: "flex",

            alignItems: "center",

            gap: 25,

            ...AUTH\_GROUP\_STYLE,

          }}

        >

          {currentUser && currentUser.emailVerified && location.pathname !== '/signup' ? (

            <>

              {/\* greeting dropdown \*/}

              <div

                className="nav-link dropdown-trigger"

                onMouseEnter={(e) =>

                  e.currentTarget

                    .querySelector(".acct-dd")

                    .classList.add("show")

                }

                onMouseLeave={(e) =>

                  e.currentTarget

                    .querySelector(".acct-dd")

                    .classList.remove("show")

                }

                style={{ cursor: "pointer" }}

              >

                Hello, {username || currentUser.email}

                <div className="acct-dd dropdown-menu">

                  <div

                    className="dropdown-item"

                    onClick={() => navigate("/user")}

                  >

                    My Account

                  </div>

                </div>

              </div>

              {/\* logout \*/}

              <button

                onClick={() => {

                  setUsername("");

                  signOut(auth);

                }}

                onMouseEnter={(e) => {

                  e.currentTarget.style.backgroundColor = "#faf0fb";

                  e.currentTarget.style.color = "#5A153D";

                }}

                onMouseLeave={(e) => {

                  e.currentTarget.style.backgroundColor = "#5A153D";

                  e.currentTarget.style.color = "#fff";

                }}

                style={{

                  padding: "8px 16px",

                  fontSize: "1rem",

                  border: "none",

                  color: "#fff",

                  backgroundColor: "#5A153D",

                  borderRadius: "4px",

                  cursor: "pointer",

                }}

              >

                Logout

              </button>

            </>

          ) : (

            <button

              onClick={() => navigate("/login")}

              onMouseEnter={(e) => {

                e.currentTarget.style.backgroundColor = "#faf0fb";

              }}

              onMouseLeave={(e) => {

                e.currentTarget.style.backgroundColor = "#fff";

              }}

              style={{

                padding: "8px 16px",

                fontSize: "1rem",

                border: "2px solid #5A153D",

                borderRadius: "4px",

                cursor: "pointer",

                color: "#5A153D",

                backgroundColor: "transparent",

              }}

            >

              Sign In

            </button>

          )}

        </div>

      </nav>

    </>

  );

};

export default Header;

Login

import React, { useState, useEffect } from "react";

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

import {

  signInWithEmailAndPassword,

  signOut,

  GoogleAuthProvider,

  signInWithPopup,

  sendEmailVerification,

  applyActionCode,

} from "firebase/auth";

import { auth } from "../firebase.js";

import { db } from "../firebase.js";

import { collection, query, where, getDocs } from "firebase/firestore";

import BottomBanner from "../components/BottomBanner.js";

import Loading from "../components/Loading.js";

const Login = ({ setCurrentUser }) => {

  const navigate = useNavigate();

  const [isLoading, setIsLoading] = useState(false);

  const [email, setEmail] = useState("");

  const [password, setPassword] = useState("");

  const [error, setError] = useState("");

  const [successMessage, setSuccessMessage] = useState("");

  const [showResendLink, setShowResendLink] = useState(false);

  const [resendCooldown, setResendCooldown] = useState(0);

  const inputStyle = {

    width: "95%",

    padding: "0.75rem",

    fontSize: "1rem",

    borderRadius: "6px",

    marginBottom: "1.3rem",

    border: "1px solid #ccc",

  };

  useEffect(() => {

      const searchParams = new URLSearchParams(window.location.search);

      if (searchParams.get('status') === 'created') {

        setSuccessMessage("Account created! We've sent a link to your email. Please verify before logging in.");

        setShowResendLink(true);

        setResendCooldown(60); // <-- ADD THIS LINE

        window.history.replaceState(null, '', window.location.pathname);

      } else if (searchParams.get('verified') === 'true') {

        setSuccessMessage('Success! Your email has been verified. You can now log in.');

        window.history.replaceState(null, '', window.location.pathname);

      }

  }, []);

  useEffect(() => {

    let timer;

    if (resendCooldown > 0) {

      timer = setTimeout(() => setResendCooldown(resendCooldown - 1), 1000);

    }

    return () => clearTimeout(timer);

  }, [resendCooldown]);

  const handleResendVerification = async () => {

    if (!email || !password) {

      setError("For security purpose, please enter your email and password to resend the link.");

      return;

    }

    try {

      setError("");

      setSuccessMessage("");

      const userCred = await signInWithEmailAndPassword(auth, email, password);

      if (userCred.user && !userCred.user.emailVerified) {

        const actionCodeSettings = {

          url: `${window.location.origin}/login?verified=true`,

        };

        await sendEmailVerification(userCred.user, actionCodeSettings);

        setSuccessMessage("A new verification link has been sent to your email.");

        setResendCooldown(60);

      }

      await signOut(auth);

    } catch (err) {

      if (err.code === 'auth/too-many-requests') {

        setError("Too many requests. Please wait a moment before trying again.");

        setResendCooldown(60);

      } else {

        setError("Failed to resend email. Please check your credentials.");

      }

      console.error("Resend error:", err);

    }

  };

  const handleLogin = async () => {

      setIsLoading(true); // <-- TURN ON LOADING

      try {

        setError("");

        setSuccessMessage("");

        setShowResendLink(false);

        const userCred = await signInWithEmailAndPassword(auth, email, password);

        await userCred.user.reload();

        if (!userCred.user.emailVerified) {

          setError("Please verify your email before logging in.");

          setShowResendLink(true);

          await signOut(auth);

          setIsLoading(false); // <-- TURN OFF LOADING

          return;

        }

        const q = query(collection(db, "users"), where("email", "==", email));

        const snapshot = await getDocs(q);

        if (snapshot.empty) {

          setError("Your account setup is not complete. Please sign up again.");

          await signOut(auth);

          setIsLoading(false); // <-- TURN OFF LOADING

          return;

        }

        setCurrentUser(userCred.user);

        setIsLoading(false); // <-- TURN OFF LOADING

        navigate("/");

      } catch (err) {

        setIsLoading(false); // <-- TURN OFF LOADING ON ERROR

        setError("Invalid email or password");

        setShowResendLink(false);

        console.error("Login error:", err);

      }

  };

  const googleProvider = new GoogleAuthProvider();

  const handleGoogleLogin = async () => {

      setIsLoading(true); // <-- TURN ON LOADING

      try {

        setError("");

        const result = await signInWithPopup(auth, googleProvider);

        const user = result.user;

        const q = query(collection(db, "users"), where("email", "==", user.email));

        const snap = await getDocs(q);

        if (snap.empty) {

          setError("Your account is not active yet. Please complete the signup process.");

          await signOut(auth);

          setIsLoading(false); // <-- TURN OFF LOADING

          return;

        }

        setCurrentUser(user);

        setIsLoading(false); // <-- TURN OFF LOADING

        navigate("/");

      } catch (err) {

        setIsLoading(false); // <-- TURN OFF LOADING ON ERROR

        console.error("Google login error:", err);

        setError("Failed to log in with Google. Please try again or use email/password.");

      }

  };

  return (

    <>

      <div style={{ backgroundColor: "#fff", minHeight: "100vh" }}>

        <div

          style={{

            width: "clamp(320px, 40%, 600px)",

            margin: "2rem auto",

            borderRadius: "12px",

            boxShadow: "0 4px 10px rgba(0,0,0,0.15)",

            display: "flex",

            flexDirection: "column",

            justifyContent: "center",

            alignItems: "center",

            padding: "3rem",

            backgroundColor: "#fff",

          }}

        >

          <h2 style={{ marginBottom: "0.5rem", fontSize: "1.8rem", color: "#333" }}>

            Sign In

          </h2>

          <p style={{ marginBottom: "1.5rem", color: "#666" }}>

            Access personalized content and exclusive features

          </p>

          <input

            type="email"

            placeholder="Email"

            value={email}

            onChange={(e) => setEmail(e.target.value)}

            style={inputStyle}

          />

          <input

            type="password"

            placeholder="Password"

            value={password}

            onChange={(e) => setPassword(e.target.value)}

            style={inputStyle}

          />

          {successMessage && (

            <p style={{ color: "green", marginBottom: "1rem" }}>{successMessage}</p>

          )}

          {error && (

            <p style={{ color: "red", marginBottom: "1rem" }}>{error}</p>

          )}

          {showResendLink && (

            <div style={{ marginBottom: "1rem", fontSize: "0.9rem", textAlign: "center" }}>

              <span>Didn't receive an email? </span>

              <button

                onClick={handleResendVerification}

                disabled={resendCooldown > 0}

                onMouseEnter={(e) => {

                  if (resendCooldown === 0) {

                    e.currentTarget.style.color = "#B12D78";

                  }

                }}

                onMouseLeave={(e) => {

                  // Only change color back if the button is active (not counting down)

                  if (resendCooldown === 0) {

                    e.currentTarget.style.color = "#5A153D";

                  }

                }}

                style={{

                  color: resendCooldown > 0 ? "#999" : "#5A153D",

                  background: "none",

                  border: "none",

                  textDecoration: "underline",

                  cursor: resendCooldown > 0 ? "default" : "pointer",

                  padding: 0,

                  fontSize: "0.9rem",

                }}

              >

                {resendCooldown > 0 ? `Resend in ${resendCooldown}s` : "Resend verification link"}

              </button>

            </div>

          )}

          <button

            onClick={handleLogin}

            onMouseEnter={(e) => {

              e.currentTarget.style.backgroundColor = "#faf0fb";

              e.currentTarget.style.color = "#5A153D";

            }}

            onMouseLeave={(e) => {

              e.currentTarget.style.backgroundColor = "#5A153D";

              e.currentTarget.style.color = "#fff";

            }}

            style={{

              width: "100%",

              padding: "0.75rem",

              backgroundColor: "#5A153D",

              color: "#fff",

              border: "2px solid #5A153D", // Added for consistency

              borderRadius: "6px",

              fontSize: "1rem",

              cursor: "pointer",

            }}

          >

            Login

          </button>

          <div

            style={{

              display: "flex",

              alignItems: "center",

              width: "100%",

              margin: "1.5rem 0",

            }}

          >

            <div style={{ flex: 1, height: "1px", backgroundColor: "#ccc" }} />

            <span style={{ margin: "0 10px", color: "#666", fontSize: "0.9rem" }}>

              Or sign in using

            </span>

            <div style={{ flex: 1, height: "1px", backgroundColor: "#ccc" }} />

          </div>

          <button

            onClick={handleGoogleLogin}

            onMouseEnter={(e) => {

              e.currentTarget.style.backgroundColor = "#faf0fb";

            }}

            onMouseLeave={(e) => {

              e.currentTarget.style.backgroundColor = "#fff";

            }}

            style={{

              width: "100%",

              padding: "0.75rem",

              backgroundColor: "#fff",

              color: "#5A153D",

              border: "2px solid #5A153D",

              borderRadius: "6px",

              fontSize: "1rem",

              cursor: "pointer",

              marginBottom: "1rem",

            }}

          >

            Google Account

          </button>

          <button

            onClick={() => navigate("/")}

            onMouseEnter={(e) => {

              e.currentTarget.style.backgroundColor = "#ccc"; // Darken on hover

            }}

            onMouseLeave={(e) => {

              e.currentTarget.style.backgroundColor = "#ddd"; // Revert on leave

            }}

            style={{

              width: "100%",

              padding: "0.75rem",

              backgroundColor: "#ddd",

              color: "#333",

              border: "none",

              borderRadius: "6px",

              fontSize: "1rem",

              cursor: "pointer",

            }}

          >

            Return Home

          </button>

          <div

            style={{

              marginTop: "1.2rem",

              fontSize: "0.9rem",

              textAlign: "center",

              color: "#333",

            }}

          >

            Don't have an account?

            <span

              onMouseEnter={(e) => {

                e.currentTarget.style.color = "#B12D78";

              }}

              onMouseLeave={(e) => {

                e.currentTarget.style.color = "#5A153D";

              }}

              style={{

                color: "#5A153D",

                cursor: "pointer",

                fontWeight: "bold",

                marginLeft: "4px",

              }}

              onClick={() => navigate("/signup")}

            >

              Sign Up

            </span>

          </div>

        </div>

      </div>

      {isLoading && <Loading />}

      <BottomBanner />

    </>

  );

};

export default Login;

Signup

import React, { useState, useEffect } from "react";

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

import {

  createUserWithEmailAndPassword,

  sendEmailVerification,

  signOut,

  GoogleAuthProvider,

  signInWithPopup,

} from "firebase/auth";

import {

  collection,

  addDoc,

  getDocs,

  query,

  where,

} from "firebase/firestore";

import { auth, db } from "../firebase.js";

import BottomBanner from "../components/BottomBanner.js";

import Loading from "../components/Loading.js";

const API\_BASE\_URL = process.env.REACT\_APP\_BACKEND\_URL || "http://127.0.0.1:5000";

const passwordRegex = /^(?=.\*[A-Za-z])(?=.\*\d)(?=.\*[^A-Za-z0-9]).{8,}$/;

function Signup({ currentUser }) {

  const navigate = useNavigate();

  const [isLoading, setIsLoading] = useState(false);

  // -------------- Form Fields --------------

  const [username, setUsername] = useState("");

  const [email, setEmail] = useState("");

  const [password, setPassword] = useState("");

  // -------------- Validation & Error States --------------

  const [error, setError] = useState("");

  const [successMessage, setSuccessMessage] = useState("");

  const [usernameError, setUsernameError] = useState("");

  const [emailError, setEmailError] = useState("");

  const [passMinLength, setPassMinLength] = useState(false);

  const [passHasLetter, setPassHasLetter] = useState(false);

  const [passHasNumber, setPassHasNumber] = useState(false);

  const [passHasSpecial, setPassHasSpecial] = useState(false);

  const [passwordTouched, setPasswordTouched] = useState(false);

  // -------------- Track if user is from Google --------------

  const [isGoogleUser, setIsGoogleUser] = useState(false);

  // --------------------------------------

  //  Username check (This function is unchanged)

  // --------------------------------------

  async function checkUsernameInUse() {

    setUsernameError("");

    if (!username) return;

    try {

      const usersRef = collection(db, "users");

      const qUsername = query(usersRef, where("username", "==", username));

      const snap = await getDocs(qUsername);

      if (!snap.empty) {

        setUsernameError("This username is already taken. Please choose a different one.");

      }

    } catch (err) {

      console.error("Error checking username:", err);

      setUsernameError("Unable to check username right now.");

    }

  }

  // --------------------------------------

  //  Email check (This function is unchanged)

  // --------------------------------------

  async function checkEmailInUse() {

    setEmailError("");

    if (!email) return;

    try {

      const usersRef = collection(db, "users");

      const qEmail = query(usersRef, where("email", "==", email));

      const snap = await getDocs(qEmail);

      if (!snap.empty) {

        setEmailError("An account with this email already exists.");

        return;

      }

    } catch (err) {

      console.error("Error checking email:", err);

      setEmailError("Unable to check email right now.");

    }

  }

  // --------------------------------------

  //  Validate password format (This function is unchanged)

  // --------------------------------------

  function validatePasswordFormat(newPass) {

    setPassMinLength(newPass.length >= 8);

    setPassHasLetter(/[A-Za-z]/.test(newPass));

    setPassHasNumber(/\d/.test(newPass));

    setPassHasSpecial(/[^A-Za-z0-9]/.test(newPass));

  }

  // --------------------------------------

  //  Google sign-up logic (This function is unchanged)

  // --------------------------------------

  const googleProvider = new GoogleAuthProvider();

  const handleGoogleSignup = async () => {

    try {

      setError("");

      const result = await signInWithPopup(auth, googleProvider);

      const usersRef = collection(db, "users");

      const emailQuery = query(usersRef, where("email", "==", result.user.email));

      const emailSnap = await getDocs(emailQuery);

      if (!emailSnap.empty) {

        setError("This email is already associated with an account. Please log in instead.");

        await signOut(auth);

        return;

      }

      setIsGoogleUser(true);

      if (result.user.displayName) {

        setUsername(result.user.displayName.replace(/\s+/g, ""));

      }

      setEmail(result.user.email);

      setSuccessMessage(

        "Google sign-up successful. Please confirm your username and click 'Create Free Account'."

      );

    } catch (err) {

      console.error("Google signup error:", err);

      setError("Failed to sign up with Google. Please try again or use email/password.");

    }

  };

  // --------------------------------------

  //  Final signup logic (MODIFIED for the new flow)

  // --------------------------------------

  const handleSignup = async () => {

      if (!isGoogleUser) {

        if (

          usernameError ||

          emailError ||

          !passMinLength ||

          !passHasLetter ||

          !passHasNumber ||

          !passHasSpecial

        ) {

          setError("Please fix the errors above before continuing.");

          return;

        }

        if (!username || !email || !password) {

          setError("All fields are required.");

          return;

        }

      } else {

        if (!username) {

          setError("Please set a username before continuing.");

          return;

        }

      }

      setIsLoading(true); // <-- TURN ON LOADING

      try {

        setError("");

        if (!isGoogleUser) {

          const userCred = await createUserWithEmailAndPassword(auth, email, password);

          const actionCodeSettings = {

            url: `${window.location.origin}/login?verified=true`,

          };

          await sendEmailVerification(userCred.user, actionCodeSettings);

          await signOut(auth);

        }

        const usersRef = collection(db, "users");

        const emailQuery = query(usersRef, where("email", "==", email));

        const emailSnap = await getDocs(emailQuery);

        if (!emailSnap.empty) {

          setError("An account with this email already exists. Please log in.");

          if (isGoogleUser) await signOut(auth);

          setIsLoading(false); // <-- TURN OFF LOADING

          return;

        }

        await addDoc(usersRef, {

          username,

          email,

          plan: "free",

          createdAt: new Date().toISOString(),

        });

        setIsLoading(false); // <-- TURN OFF LOADING

        if (isGoogleUser) {

          navigate("/");

        } else {

          navigate("/login?status=created");

        }

      } catch (err) {

        setIsLoading(false); // <-- TURN OFF LOADING ON ERROR

        if (err.code === 'auth/email-already-in-use') {

          setError("This email is already registered. Please log in or use a different email.");

        } else {

          console.error("Signup error:", err);

          setError(err.message);

        }

      }

  };

  return (

    <>

      <div style={{ backgroundColor: "#fff", minHeight: "100vh" }}>

        <div

          style={{

            width: "clamp(320px, 40%, 600px)",

            margin: "2rem auto",

            borderRadius: "12px",

            boxShadow: "0 4px 10px rgba(0, 0, 0, 0.15)",

            display: "flex",

            flexDirection: "column",

            alignItems: "center",

            padding: "3rem",

            backgroundColor: "#fff",

          }}

        >

          <h2>Sign Up</h2>

          {successMessage && <p style={{ color: "green" }}>{successMessage}</p>}

          {error && <p style={{ color: "red" }}>{error}</p>}

          <input

            type="text"

            placeholder="Username"

            value={username}

            onChange={(e) => {

              setUsername(e.target.value);

              setUsernameError("");

            }}

            onBlur={checkUsernameInUse}

            style={inputStyle}

          />

          {usernameError && (

            <p style={{ color: "red", marginTop: "-0.5rem" }}>{usernameError}</p>

          )}

          {!isGoogleUser && (

            <>

              <input

                type="password"

                placeholder="Password"

                value={password}

                onChange={(e) => {

                  const newPass = e.target.value;

                  setPassword(newPass);

                  setPasswordTouched(true);

                  validatePasswordFormat(newPass);

                }}

                style={inputStyle}

              />

              {passwordTouched && (

                <div style={{ textAlign: "left", marginBottom: "1.3rem" }}>

                  <p style={{ margin: "0 0 0.25rem", fontWeight: "bold" }}>

                    Password Requirements:

                  </p>

                  <ul style={{ listStyleType: "disc", paddingLeft: "1.25rem", margin: 0 }}>

                    <li style={{ color: passMinLength ? "green" : "red" }}>

                      At least 8 characters

                    </li>

                    <li style={{ color: passHasLetter ? "green" : "red" }}>

                      At least one letter

                    </li>

                    <li style={{ color: passHasNumber ? "green" : "red" }}>

                      At least one digit

                    </li>

                    <li style={{ color: passHasSpecial ? "green" : "red" }}>

                      At least one special character

                    </li>

                  </ul>

                </div>

              )}

            </>

          )}

          <input

            type="email"

            placeholder="Email"

            value={email}

            onChange={(e) => {

              setEmail(e.target.value);

              setEmailError("");

            }}

            onBlur={checkEmailInUse}

            style={inputStyle}

            disabled={isGoogleUser}

          />

          {emailError && (

            <p style={{ color: "red", marginTop: "-0.5rem" }}>{emailError}</p>

          )}

          {!isGoogleUser && (

            <>

              <div

                style={{

                  display: "flex",

                  alignItems: "center",

                  width: "100%",

                  margin: "0.5rem 0 0.5rem 0",

                }}

              >

                <div style={{ flex: 1, height: "1px", backgroundColor: "#ccc" }} />

                <span style={{ margin: "0 10px", color: "#666", fontSize: "0.9rem" }}>

                  Or sign up using

                </span>

                <div style={{ flex: 1, height: "1px", backgroundColor: "#ccc" }} />

              </div>

              <button

                onClick={handleGoogleSignup}

                onMouseEnter={(e) => {

                  e.currentTarget.style.backgroundColor = "#faf0fb";

                  e.currentTarget.style.color = "#5A153D";

                }}

                onMouseLeave={(e) => {

                  e.currentTarget.style.backgroundColor = "#fff";

                  e.currentTarget.style.color = "#5A153D";

                }}

                style={{

                  margin: "1rem 0",

                  padding: "0.75rem 1.5rem",

                  backgroundColor: "#fff",

                  color: "#5A153D",

                  border: "2px solid #5A153D",

                  borderRadius: "6px",

                  fontSize: "1rem",

                  width: "105%",

                  height: "45px",

                  cursor: "pointer",

                }}

              >

                Google Account

              </button>

            </>

          )}

          <button

            onClick={handleSignup}

            onMouseEnter={(e) => {

              e.currentTarget.style.backgroundColor = "#faf0fb";

              e.currentTarget.style.color = "#5A153D";

            }}

            onMouseLeave={(e) => {

              e.currentTarget.style.backgroundColor = "#5A153D";

              e.currentTarget.style.color = "#fff";

            }}

            style={signupBtn}

          >

            Create Free Account

          </button>

          <div

            style={{

              marginTop: "1.5rem",

              fontSize: "0.9rem",

              textAlign: "center",

              color: "#333",

            }}

          >

            Already have an account?

            <span

              onClick={() => navigate("/login")}

              onMouseEnter={(e) => {

                e.currentTarget.style.color = "#B12D78";

              }}

              onMouseLeave={(e) => {

                e.currentTarget.style.color = "#5A153D";

              }}

              style={{

                color: "#5A153D",

                cursor: "pointer",

                fontWeight: "bold",

                marginLeft: "4px",

              }}

            >

              Log In

            </span>

          </div>

        </div>

      </div>

      {isLoading && <Loading />}

      <BottomBanner />

    </>

  );

  }

// styling

const inputStyle = {

  width: "100%",

  padding: "0.75rem",

  marginBottom: "1.3rem",

  fontSize: "1rem",

  borderRadius: "6px",

  border: "1px solid #ccc",

};

const signupBtn = {

  marginTop: "1.5rem",

  padding: "0.75rem 1.5rem",

  backgroundColor: "#5A153D",

  color: "#fff",

  border: "none",

  width: "105%",

  height: "45px",

  borderRadius: "6px",

  fontSize: "1rem",

  cursor: "pointer",

};

export default Signup;

Pricing

import React, { useEffect, useState } from "react";

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

import BottomBanner from "../components/BottomBanner.js";

import Loading from "../components/Loading.js";

const API\_BASE\_URL = process.env.REACT\_APP\_BACKEND\_URL || "http://127.0.0.1:5000";

// The component now accepts props from App.js

function PricingPage({ currentUser, userPlan }) {

  const navigate = useNavigate();

  const [showSuccessPopup, setShowSuccessPopup] = useState(false);

  const [showCancelPopup, setShowCancelPopup] = useState(false);

  const [isLoading, setIsLoading] = useState(false);

  // This useEffect is now MUCH simpler. It only handles the redirect from Stripe.

  useEffect(() => {

    if (performance.getEntriesByType("navigation")[0]?.type === "back\_forward") {

      setIsLoading(false);

    }

    const params = new URLSearchParams(window.location.search);

    if (params.get("status") === "success") {

      setShowSuccessPopup(true);

      // Clean up the URL so the popup doesn't reappear on refresh

      window.history.replaceState({}, document.title, "/pricing");

    } else if (params.get("status") === "cancel") {

      setShowCancelPopup(true);

      window.history.replaceState({}, document.title, "/pricing");

    }

  }, []); // The dependency array is empty as it only needs to run once.

  // This function now uses the currentUser prop.

  const handleSubscribe = async () => {

    // If no user is logged in, redirect them to the login page.

    if (!currentUser) {

      navigate('/login');

      return;

    }

    setIsLoading(true);

    try {

      const response = await fetch(`${API\_BASE\_URL}/api/create-checkout-session`, {

        method: "POST",

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

        // Send the current user's email to the backend to identify them

        body: JSON.stringify({ email: currentUser.email })

      });

      const data = await response.json();

      if (data.url) {

        window.location.href = data.url;

      } else {

        alert("Unable to create Stripe session: " + (data.error || "Unknown error"));

        setIsLoading(false);

      }

    } catch (err) {

      console.error("Error:", err);

      setIsLoading(false);

    }

  };

  return (

    <div>

      <div style={{ padding: "80px 20px", textAlign: "center" }}>

        <h1 style={{ color: "#5A153D", fontSize: "2.5rem" }}>Unlock Premium Access</h1>

        <p style={{ fontSize: "1.2rem", color: "#333" }}>

          Upgrade to Premium for full access to all our analytical tools and data.

        </p>

        {/\* This button now uses the userPlan prop \*/}

        <button

          onClick={handleSubscribe}

          disabled={isLoading || userPlan === 'premium'}

          onMouseEnter={(e) => {

            if (userPlan !== 'premium') {

              e.currentTarget.style.backgroundColor = "#faf0fb";

              e.currentTarget.style.color = "#5A153D";

            }

          }}

          onMouseLeave={(e) => {

             if (userPlan !== 'premium') {

               e.currentTarget.style.backgroundColor = "#5A153D";

               e.currentTarget.style.color = "#fff";

             }

          }}

          style={{

            backgroundColor: userPlan === 'premium' ? "#ccc" : "#5A153D",

            color: userPlan === 'premium' ? "#666" : "#fff",

            padding: "12px 28px",

            fontSize: "1.1rem",

            border: "none",

            borderRadius: "8px",

            cursor: userPlan === 'premium' ? "not-allowed" : "pointer",

            marginTop: "30px",

            transition: "background-color 0.2s, color 0.2s"

          }}

        >

          {userPlan === 'premium' ? 'You Are a Premium Member' : 'Subscribe Now'}

        </button>

      </div>

      {/\* SUCCESS POPUP (This section is unchanged) \*/}

      {showSuccessPopup && (

        <div

          style={{

            position: "fixed",

            top: 0,

            left: 0,

            width: "100%",

            height: "100%",

            backgroundColor: "rgba(0, 0, 0, 0.5)",

            display: "flex",

            alignItems: "center",

            justifyContent: "center",

            zIndex: 1000,

          }}

        >

          <div

            style={{

              backgroundColor: "#fff",

              padding: "25px",

              borderRadius: "10px",

              textAlign: "center",

              width: "320px",

              boxShadow: "0 4px 8px rgba(0,0,0,0.2)",

            }}

          >

            <h3 style={{ color: "#5A153D" }}>Payment Successful!</h3>

            <p style={{ color: "#333" }}>

              You are now a premium subscriber. Thank you!

            </p>

            <button

              onClick={() => setShowSuccessPopup(false)}

              style={{

                backgroundColor: "#5A153D",

                color: "white",

                padding: "10px 16px",

                border: "none",

                borderRadius: "6px",

                marginTop: "10px",

                cursor: "pointer",

              }}

            >

              Close

            </button>

          </div>

        </div>

      )}

      {/\* CANCEL POPUP (This section is unchanged) \*/}

      {showCancelPopup && (

        <div

          style={{

            position: "fixed",

            top: 0,

            left: 0,

            width: "100%",

            height: "100%",

            backgroundColor: "rgba(0, 0, 0, 0.5)",

            display: "flex",

            alignItems: "center",

            justifyContent: "center",

            zIndex: 1000,

          }}

        >

          <div

            style={{

              backgroundColor: "#fff",

              padding: "25px",

              borderRadius: "10px",

              textAlign: "center",

              width: "320px",

              boxShadow: "0 4px 8px rgba(0,0,0,0.2)",

            }}

          >

            <h3 style={{ color: "#5A153D" }}>Payment Canceled</h3>

            <p style={{ color: "#333" }}>

              Your subscription was not completed.

            </p>

            <button

              onClick={() => setShowCancelPopup(false)}

              style={{

                backgroundColor: "#5A153D",

                color: "white",

                padding: "10px 16px",

                border: "none",

                borderRadius: "6px",

                marginTop: "10px",

                cursor: "pointer",

              }}

            >

              Close

            </button>

          </div>

        </div>

      )}

      {isLoading && <Loading />}

      <BottomBanner />

    </div>

  );

}

export default PricingPage;

Usseraccount

import React, { useState, useEffect } from "react";

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

import { auth, db } from "../firebase.js";

import { onAuthStateChanged } from "firebase/auth";

import { collection, query, where, getDocs } from "firebase/firestore";

import BottomBanner from "../components/BottomBanner.js";

const Useraccount = () => {

  const navigate = useNavigate();

  const [userData, setUserData] = useState(null);

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

  useEffect(() => {

    const unsubscribe = onAuthStateChanged(auth, async (user) => {

      if (user) {

        const usersRef = collection(db, "users");

        const q = query(usersRef, where("email", "==", user.email));

        const snapshot = await getDocs(q);

        if (!snapshot.empty) {

          const data = snapshot.docs[0].data();

          setUserData(data);

        } else {

          console.error("No user document found");

          navigate("/signup");

        }

      } else {

        navigate("/login");

      }

      setLoading(false);

    });

    return () => unsubscribe();

  }, [navigate]);

  if (loading) {

    return (

      <div className="App" style={{ padding: "2rem", textAlign: "center" }}>

        Loading...

      </div>

    );

  }

  if (!userData) {

    return (

      <div className="App" style={{ padding: "2rem", textAlign: "center" }}>

        User data not available.

      </div>

    );

  }

  // --- Style definitions for the new look ---

  const containerStyle = {

    maxWidth: "700px",

    margin: "0 30px",

    padding: "0 20px",

    textAlign: "left", // This will now correctly align everything inside

  };

  const sectionStyle = {

    border: "1px solid #e0e0e0",

    borderRadius: "8px",

    padding: "24px",

    marginBottom: "20px",

  };

  const infoRowStyle = {

    display: "flex",

    justifyContent: "space-between", // Changed back to space-between for clean columns

    alignItems: "center",

    padding: "12px 0",

    borderBottom: "1px solid #f0f0f0",

    fontSize: "1rem"

  };

  const buttonContainerStyle = {

    display: "flex",

    justifyContent: "flex-start",

    gap: "15px",

    marginTop: "30px",

  };

  const buttonStyle = {

    padding: "10px 20px",

    fontSize: "16px",

    borderRadius: "5px",

    backgroundColor: "#5A153D",

    color: "#fff",

    border: "2px solid #5A153D",

    cursor: "pointer",

    transition: "background-color 0.2s, color 0.2s"

  };

  return (

    // We leave the outer "App" div alone as it's part of the global layout

    <div className="App" style={{ paddingTop: "1rem" }}>

      {/\* We add a NEW container div inside to control this page's specific layout \*/}

      <div style={containerStyle}>

        <h2 style={{ marginBottom: "1rem", fontSize: "1.75rem" }}>My Account</h2>

        <hr style={{ border: "none", borderBottom: "1px solid #e0e0e0", marginBottom: "2rem" }} />

        <div style={sectionStyle}>

            <h2 style={{marginTop: 0, borderBottom: '1px solid #eee', paddingBottom: '15px', fontSize: '1.5rem'}}>User Information</h2>

            <div style={infoRowStyle}>

                <strong style={{color: '#555'}}>Username:</strong>

                <span>{userData.username}</span>

            </div>

            <div style={infoRowStyle}>

                <strong style={{color: '#555'}}>Email:</strong>

                <span>{userData.email}</span>

            </div>

             <div style={{...infoRowStyle, borderBottom: 'none'}}>

                <strong style={{color: '#555'}}>Registered Date:</strong>

                <span>{userData.createdAt ? new Date(userData.createdAt).toLocaleDateString() : "N/A"}</span>

            </div>

        </div>

        <div style={sectionStyle}>

            <h2 style={{marginTop: 0, borderBottom: '1px solid #eee', paddingBottom: '15px', fontSize: '1.5rem'}}>Subscription</h2>

             <div style={{...infoRowStyle, borderBottom: 'none'}}>

                <strong style={{color: '#555'}}>Current Plan:</strong>

                <span style={{textTransform: 'capitalize', fontWeight: 'bold'}}>{userData.plan}</span>

            </div>

        </div>

        <div style={buttonContainerStyle}>

          <button

            onClick={() => navigate("/")}

            onMouseEnter={(e) => {

              e.currentTarget.style.backgroundColor = "#faf0fb";

              e.currentTarget.style.color = "#5A153D";

            }}

            onMouseLeave={(e) => {

              e.currentTarget.style.backgroundColor = "#5A153D";

              e.currentTarget.style.color = "#fff";

            }}

            style={buttonStyle}

          >

            Back to Home

          </button>

          {userData.plan === 'free' && (

            <button

              onClick={() => navigate('/pricing')}

               onMouseEnter={(e) => {

                e.currentTarget.style.backgroundColor = "#faf0fb";

                e.currentTarget.style.color = "#5A153D";

              }}

              onMouseLeave={(e) => {

                e.currentTarget.style.backgroundColor = "#5A153D";

                e.currentTarget.style.color = "#fff";

              }}

              style={buttonStyle}

            >

              Upgrade to Premium

            </button>

          )}

        </div>

      </div>

      <BottomBanner />

    </div>

  );

};

export default Useraccount;

App.py

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

import pandas as pd

from sqlalchemy import text

from flask\_cors import CORS

from datetime import datetime

import os

from dotenv import load\_dotenv

import stripe

import bcrypt

import jwt

from datetime import timedelta

import json

import requests

import traceback

from worker import generate\_stability\_analysis\_task

from celery.result import AsyncResult

from google.cloud import firestore

import firebase\_admin

from firebase\_admin import credentials, firestore as admin\_firestore

# Explicitly load environment variables from the Credentials.env file

dotenv\_path = os.path.abspath(

    "C:/Users/wsche/OneDrive/桌面/Investment Research/Startup Project/Python Run/Credentials.env"

)

load\_dotenv(dotenv\_path)

app = Flask(\_\_name\_\_)

CORS(app, resources={r"/api/\*": {"origins": ["http://localhost:3000", "https://www.viserra-group.com"]}})

# get the stripe secret key from the environment variables

stripe.api\_key = os.getenv("STRIPE\_SECRET\_KEY")

STRIPE\_WEBHOOK\_SECRET = os.getenv("STRIPE\_WEBHOOK\_SECRET")

# Load secret key for JWT auth (Log in and Sign up)

app.config['SECRET\_KEY'] = os.getenv("APP\_SECRET\_KEY")

# Database credentials from environment variables

DB\_USERNAME = os.getenv("DB\_USERNAME")

DB\_PASSWORD = os.getenv("DB\_PASSWORD")

DB\_HOST = os.getenv("DB\_HOST")

DB\_PORT = os.getenv("DB\_PORT")

DB\_NAME = os.getenv("DB\_NAME")

# Construct the database connection string

DB\_URL = f"mysql+pymysql://{DB\_USERNAME}:{DB\_PASSWORD}@{DB\_HOST}:{DB\_PORT}/{DB\_NAME}"

# Apply the same SSL forced connection logic

app.config['SQLALCHEMY\_DATABASE\_URI'] = DB\_URL

app.config['SQLALCHEMY\_ENGINE\_OPTIONS'] = {

    "connect\_args": {

        "ssl": {

            "fake\_flag\_to\_enable": True  # Ensures SSL connection

        }

    }

}

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

# Initialize SQLAlchemy with the updated configuration

db = SQLAlchemy(app)

# -------------------------------------------------------------------------

# =========================== REIT ENDPOINTS ==============================

# -------------------------------------------------------------------------

@app.route('/')

def index():

    return "REIT Screener API is running!"

# -------------------------------------------------------------------------

# =========================== REIT ENDPOINTS ==============================

# -------------------------------------------------------------------------

@app.route('/api/reits', methods=['GET'])

def get\_reits():

    """

    Filters REITs based on user-selected preferences:

    - Country (from 'Country\_Region' in reit\_business\_data)

    - Property Type (from 'Property\_Type' in reit\_business\_data; supports multiple categories)

    - Ticker (if ticker=?)

    - min\_avg\_return (for Average Annual Return)

    - search (partial ticker match for real-time suggestions)

    Merges with scoring analysis data from reit\_scoring\_analysis.

    Returns relevant business data plus new fields (Numbers\_Employee, Year\_Founded, etc.).

    """

    # Get user selections from request parameters

    selected\_country = request.args.get('country', default=None, type=str)

    selected\_property\_type = request.args.get('property\_type', default=None, type=str)

    selected\_ticker = request.args.get('ticker', default=None, type=str)

    min\_avg\_return = request.args.get('min\_avg\_return', default=None, type=float)

    # NEW: Real-time search parameter

    search\_term = request.args.get('search', default=None, type=str)

    app.logger.info("Search term received: %s", search\_term)

    # Load REIT business data from MySQL

    try:

        with db.engine.connect() as conn:

            query = "SELECT \* FROM reit\_business\_data"

            business\_data = pd.read\_sql(query, conn)

            app.logger.info(f"Total REITs loaded from business data: {business\_data.shape[0]}")

    except Exception as e:

        app.logger.error(f"Error loading REIT business data: {e}")

        return jsonify({"error": "Failed to load REIT business data"}), 500

    # Apply filters if present

    if selected\_country:

        business\_data = business\_data[business\_data['Country\_Region'] == selected\_country]

    if selected\_property\_type:

        business\_data = business\_data[

            business\_data['Property\_Type'].str.contains(selected\_property\_type, case=False, na=False)

        ]

    if selected\_ticker:

        business\_data = business\_data[business\_data['Ticker'] == selected\_ticker]

    # NEW: If a search term is provided, filter by Ticker startswith (case-insensitive)

    if search\_term:

        if 'Ticker' in business\_data.columns:

            app.logger.info("Ticker column sample: %s", business\_data['Ticker'].head().to\_dict())

        else:

            app.logger.error("Ticker column missing in business\_data")

        try:

            business\_data = business\_data[

                business\_data['Ticker'].notna() &

                business\_data['Ticker'].astype(str).str.lower().str.startswith(search\_term.lower(), na=False)

            ]

            app.logger.info("After search filter, business\_data shape: %s", business\_data.shape)

        except Exception as e:

            app.logger.error("Error filtering by search term: %s", e)

            return jsonify({"error": "Error filtering by search term"}), 500

    app.logger.info(

        f"Filtered REITs after country/property/ticker/search selection: {business\_data.shape[0]}"

    )

    if business\_data.empty:

        return jsonify({"explanation": "No REITs match the selected criteria.", "reits": []})

    # Load scoring analysis data from MySQL

    try:

        with db.engine.connect() as conn:

            risk\_query = "SELECT \* FROM reit\_scoring\_analysis"

            risk\_data = pd.read\_sql(risk\_query, conn)

            app.logger.info(f"Total REITs loaded from scoring analysis: {risk\_data.shape[0]}")

    except Exception as e:

        app.logger.error(f"Error loading scoring analysis data: {e}")

        return jsonify({"error": "Failed to load scoring analysis data"}), 500

    # Merge business data with scoring analysis data using the 'Ticker' column

    merged\_data = pd.merge(business\_data, risk\_data, on="Ticker", how="inner")

    app.logger.info(

        f"Total REITs after merging business and scoring analysis data: {merged\_data.shape[0]}"

    )

    # Apply Average Annual Return filter

    if min\_avg\_return is not None:

        merged\_data = merged\_data[merged\_data['Average Annual Return'] > min\_avg\_return]

        app.logger.info(

            f"Filtered REITs with Average Annual Return greater than {min\_avg\_return}: {merged\_data.shape[0]}"

        )

    # Replace NaN values with None for better JSON serialization

    merged\_data = merged\_data.astype(object).where(pd.notna(merged\_data), None)

    # We won't sort; display in original order

    data\_to\_display = merged\_data

    explanation = (

        f"Filtered REITs: Minimum Annual Annual Return - {min\_avg\_return}, "

        f"Filtered REITs: Country - {selected\_country}, "

        f"Property Type - {selected\_property\_type}, "

        f"Ticker - {selected\_ticker}."

    )

    response = {

        "explanation": explanation,

        "reits": data\_to\_display[

            [

                "Ticker",

                "Company\_Name",

                "Business\_Description",

                "Website",

                "Numbers\_Employee",

                "Target\_Price",

                "Year\_Founded",

                "US\_Investment\_Regions",

                "Overseas\_Investment",

                "Property\_Type",

                "Total\_Real\_Estate\_Assets\_M\_",

                "5yr\_FFO\_Growth",

            ]

        ].to\_dict(orient='records')

    }

    return jsonify(response)

# -------------------------------------------------------------------------

# QUARTERLY STATEMENTS ENDPOINT (Income Statement, Balance Sheet, Cash Flow)

# -------------------------------------------------------------------------

@app.route("/api/reits/<string:ticker>/statements/quarterly", methods=['GET'])

def get\_quarterly\_statements(ticker):

    """

    Fetches quarterly financial statements for a given ticker from one of:

      reit\_income\_statement (Income Statement)

      reit\_balance\_sheet   (Balance Sheet)

      reit\_cash\_flow       (Cash Flow)

      reit\_industry\_metrics (Industry Specific)

    Usage example:

      GET /api/reits/WPC/statements/quarterly?type=is

        => returns Income Statement rows for WPC

      Optional query params:

        limit      -> # of rows to limit (e.g. ?limit=100)

        from\_year  -> min year to filter

        to\_year    -> max year to filter

    """

    statement\_type = request.args.get("type", "is").lower()

    limit = request.args.get("limit", default=None, type=int)

    from\_year = request.args.get("from\_year", default=None, type=int)

    to\_year = request.args.get("to\_year", default=None, type=int)

    # Map type -> table name

    table\_map = {

        "is": "reit\_income\_statement",

        "bs": "reit\_balance\_sheet",

        "cf": "reit\_cash\_flow",

        "industry": "reit\_industry\_metrics",

    }

    table\_name = table\_map.get(statement\_type)

    if not table\_name:

        return jsonify({"error": "Invalid 'type' parameter. Must be one of is|bs|cf|industry."}), 400

    # Build the base SELECT and WHERE

    sql = f"""

        SELECT

            line\_item,

            fiscal\_year,

            fiscal\_quarter,

            value,

            excel\_row\_index

        FROM {table\_name}

        WHERE ticker = :ticker

    """

    params = {"ticker": ticker}

    # Dynamically add any filters

    if from\_year is not None:

        sql += " AND fiscal\_year >= :from\_year"

        params["from\_year"] = from\_year

    if to\_year is not None:

        sql += " AND fiscal\_year <= :to\_year"

        params["to\_year"] = to\_year

    # Add ORDER BY last (after WHERE conditions)

    sql += " ORDER BY excel\_row\_index ASC, fiscal\_year ASC, fiscal\_quarter ASC"

    # Optionally limit the number of rows

    if limit is not None:

        sql += " LIMIT :limit"

        params["limit"] = limit

    try:

        with db.engine.connect() as conn:

            df = pd.read\_sql(text(sql), conn, params=params)

    except Exception as e:

        app.logger.error(f"Error fetching quarterly statements for {ticker}: {e}")

        return jsonify({"error": "Failed to load statements"}), 500

    if df.empty:

        return jsonify({"message": f"No {statement\_type.upper()} data found for ticker '{ticker}'"}), 200

    # Convert the 'fiscal\_quarter' column to None where blank

    df["fiscal\_quarter"] = df["fiscal\_quarter"].astype(object).where(pd.notna(df["fiscal\_quarter"]), None)

    records = df.to\_dict(orient="records")

    return jsonify({

        "ticker": ticker,

        "statement\_type": statement\_type,

        "rows": records

    })

# -------------------------------------------------------------------------

# PORTFOLIO ANALYSIS ENDPOINT

# -------------------------------------------------------------------------

@app.route("/api/reits/<string:ticker>/breakdowns", methods=['GET'])

def get\_portfolio\_breakdowns(ticker):

    """

    Returns portfolio breakdowns by property\_type, secondary\_type, US state, and country.

    Each entry has: category, rba\_gla, pct (fraction of total), data source, and calc basis.

    """

    try:

        with db.engine.connect() as conn:

            df = pd.read\_sql(text("""

                SELECT breakdown\_type

                     , category

                     , rba\_gla

                     , pct

                     , source

                     , basis

                  FROM reit\_portfolio\_analysis

                 WHERE ticker = :ticker

                 ORDER BY

                   FIELD(breakdown\_type,

                         'property\_type',

                         'secondary\_type',

                         'state',

                         'country'),

                   pct DESC

            """), conn, params={"ticker": ticker})

    except Exception as e:

        app.logger.error(f"Error loading portfolio breakdowns for {ticker}: {e}")

        return jsonify({"error": "Failed to load breakdowns"}), 500

    if df.empty:

        return jsonify({"message": f"No breakdowns found for ticker '{ticker}'"}), 200

    # pivot into four lists, now including source & basis

    result = {}

    for btype in ["property\_type", "secondary\_type", "state", "country"]:

        sub = df[df["breakdown\_type"] == btype][

            ["category", "rba\_gla", "pct", "source", "basis"]

        ]

        result[btype] = sub.to\_dict(orient="records")

    return jsonify({"ticker": ticker, "breakdowns": result}), 200

# -------------------------------------------------------------------------

# OVERVIEW FINANCIAL DATA ENDPOINT (Last 6 quarters)

# -------------------------------------------------------------------------

def convert\_date\_to\_quarter(date\_obj):

    """

    Convert a datetime (e.g., 2024-03-31) to a string like Q1 '24.

    """

    if pd.isna(date\_obj) or not isinstance(date\_obj, (datetime, pd.Timestamp)):

        return None

    quarter = (date\_obj.month - 1) // 3 + 1

    year\_short = str(date\_obj.year)[-2:]

    return f"Q{quarter} '{year\_short}"

def build\_col\_name(ticker\_prefix, metric):

    """

    Build a column name like "GIPR\_US\_Equity\_FFO\_PS".

    """

    return f"{ticker\_prefix}\_{metric}"

@app.route("/api/reits/<ticker>/financials", methods=['GET'])

def get\_financials(ticker):

    """

    Returns up to 6 most recent quarterly data points for FFO\_PS, DVD, and NOI\_PS.

    Optionally (if include\_scores=true is passed), also returns

    stability\_percentile and fundamental\_percentile.

    """

    include\_scores = request.args.get('include\_scores', 'false').lower() == 'true'

    # 1) Define the financial line items we want to fetch

    line\_items\_to\_fetch = [

        'Dividends per Share',  # From Income Statement

        'FFO',                  # From Industry Metrics

        'FFO / Total Revenue %' # From Industry Metrics

    ]

    # 2) Build and execute the SQL query to fetch the data in long format

    # We use UNION ALL to combine results from two different tables efficiently.

    sql\_query = text("""

        SELECT fiscal\_year, fiscal\_quarter, line\_item, value

        FROM reit\_income\_statement

        WHERE ticker = :ticker AND line\_item = 'Dividends per Share' AND fiscal\_quarter IS NOT NULL

        UNION ALL

        SELECT fiscal\_year, fiscal\_quarter, line\_item, value

        FROM reit\_industry\_metrics

        WHERE ticker = :ticker AND line\_item IN ('FFO', 'FFO / Total Revenue %') AND fiscal\_quarter IS NOT NULL

        ORDER BY fiscal\_year, fiscal\_quarter

    """)

    try:

        with db.engine.connect() as conn:

            df = pd.read\_sql(sql\_query, conn, params={"ticker": ticker})

        if df.empty:

            # If no data, prepare an empty response but still fetch scores later

            results = []

        else:

            # 3) Pivot the data from long to wide format

            # This makes it easier to create the JSON object for each time period.

            pivoted\_df = df.pivot\_table(

                index=['fiscal\_year', 'fiscal\_quarter'],

                columns='line\_item',

                values='value'

            ).reset\_index()

            # 4) Take the last 26 quarters for the overview chart

            pivoted\_df = pivoted\_df.tail(26)

            # Sanitize column names for JSON compatibility (replace spaces and %)

            pivoted\_df.rename(columns={

                'Dividends per Share': 'dividends\_per\_share',

                'FFO': 'ffo',

                'FFO / Total Revenue %': 'ffo\_per\_revenue\_pct'

            }, inplace=True)

            # 5) Format the data into the JSON structure the frontend expects

            results = []

            for \_, row in pivoted\_df.iterrows():

                # Re-create the "Q1 '23" style quarter label

                year\_short = str(int(row['fiscal\_year']))[-2:]

                quarter\_label = f"Q{int(row['fiscal\_quarter'])} '{year\_short}"

                row\_obj = {

                    "quarter": quarter\_label,

                }

                # Add each metric if it exists in the row, otherwise add None

                row\_obj["dividends\_per\_share"] = float(row['dividends\_per\_share']) if pd.notna(row.get('dividends\_per\_share')) else None

                row\_obj["ffo"] = float(row['ffo']) if pd.notna(row.get('ffo')) else None

                row\_obj["ffo\_per\_revenue\_pct"] = float(row['ffo\_per\_revenue\_pct']) if pd.notna(row.get('ffo\_per\_revenue\_pct')) else None

                results.append(row\_obj)

    except Exception as e:

        app.logger.error(f"Error fetching real-time financial data for {ticker}: {e}")

        return jsonify({"error": "Failed to load financial overview data"}), 500

    if include\_scores:

        # Look up scoring analysis

        try:

            with db.engine.connect() as conn:

                scoring\_query = f"SELECT \* FROM reit\_scoring\_analysis WHERE Ticker = '{ticker}'"

                scoring\_data = pd.read\_sql(scoring\_query, conn)

        except Exception as e:

            app.logger.error(f"Error loading scoring analysis for ticker {ticker}: {e}")

            scoring\_data = pd.DataFrame()

        if not scoring\_data.empty:

            scoring\_row = scoring\_data.iloc[0]

            stability\_score = scoring\_row.get("Stability Percentile", None)

            fundamental\_score = scoring\_row.get("Fundamental\_Percentile", None)

        else:

            stability\_score = None

            fundamental\_score = None

        response = {

            "quarterly\_data": results,

            "stability\_percentile": stability\_score,

            "fundamental\_percentile": fundamental\_score

        }

        return jsonify(response), 200

    else:

        # Return only the array of quarterly data for backward compatibility

        return jsonify(results), 200

class EmailSignup(db.Model):

    \_\_tablename\_\_ = "email\_signups"

    id = db.Column(db.Integer, primary\_key=True, autoincrement=True)

    email = db.Column(db.String(255), unique=True, nullable=False)

    interest = db.Column(db.Enum("REITs", "Crowdfunding", "Both"), nullable=False)

    feedback = db.Column(db.Text, nullable=True)

    created\_at = db.Column(db.TIMESTAMP, server\_default=db.func.current\_timestamp())

    def \_\_init\_\_(self, email, interest, feedback=None):

        self.email = email

        self.interest = interest

        self.feedback = feedback

@app.route("/api/signup", methods=["POST"])

def signup():

    """Handles new email signups and stores them in the MySQL database."""

    data = request.json

    email = data.get("email")

    interest = data.get("interest")

    feedback = data.get("feedback", None)  # Optional field

    if not email or not interest:

        return jsonify({"error": "Missing required fields"}), 400

    # Check if email already exists

    existing\_entry = db.session.execute(

        db.select(EmailSignup).filter\_by(email=email)

    ).scalar\_one\_or\_none()

    if existing\_entry:

        return jsonify({"error": "Email already exists in database"}), 409

    try:

        # Insert new record using SQLAlchemy ORM

        new\_signup = EmailSignup(email=email, interest=interest, feedback=feedback)

        db.session.add(new\_signup)

        db.session.commit()

        return jsonify({"message": "Signup successful!"}), 201

    except Exception as e:

        db.session.rollback()

        return jsonify({"error": f"Database error: {str(e)}"}), 500

class ContactMessage(db.Model):

    \_\_tablename\_\_ = "contact\_messages"

    id = db.Column(db.Integer, primary\_key=True, autoincrement=True)

    first\_name = db.Column(db.String(100), nullable=False)

    last\_name = db.Column(db.String(100), nullable=False)

    email = db.Column(db.String(255), nullable=False)

    message = db.Column(db.Text, nullable=False)

    created\_at = db.Column(db.TIMESTAMP, server\_default=db.func.current\_timestamp())

    def \_\_init\_\_(self, first\_name, last\_name, email, message):

        self.first\_name = first\_name

        self.last\_name = last\_name

        self.email = email

        self.message = message

@app.route("/api/contact", methods=["POST"])

def contact():

    """

    Handles new contact form submissions and stores them in the contact\_messages table.

    """

    data = request.json

    first\_name = data.get("firstName")

    last\_name = data.get("lastName")

    email = data.get("email")

    message = data.get("message")

    # Basic validation

    if not all([first\_name, last\_name, email, message]):

        return jsonify({"error": "Missing required fields"}), 400

    try:

        new\_contact = ContactMessage(

            first\_name=first\_name,

            last\_name=last\_name,

            email=email,

            message=message

        )

        db.session.add(new\_contact)

        db.session.commit()

        return jsonify({"message": "Contact message received!"}), 201

    except Exception as e:

        db.session.rollback()

        return jsonify({"error": f"Database error: {str(e)}"}), 500

@app.route("/api/reits/<string:ticker>/price", methods=['GET'])

def get\_price\_data(ticker):

    """

    Returns all historical close\_price and volume for the specified ticker.

    """

    try:

        with db.engine.connect() as conn:

            sql\_query = f"""

                SELECT date, close\_price, volume

                FROM reit\_price\_data

                WHERE ticker = '{ticker}'

                ORDER BY date ASC

            """

            df\_price = pd.read\_sql(sql\_query, conn)

        if df\_price.empty:

            return jsonify({"message": f"No price data found for ticker '{ticker}'"}), 200

        # Convert to JSON-safe types

        df\_price["date"] = df\_price["date"].astype(str)

        df\_price["close\_price"] = df\_price["close\_price"].astype(float)

        df\_price["volume"] = df\_price["volume"].astype(float)

        price\_records = df\_price.to\_dict(orient='records')

        return jsonify({

            "ticker": ticker,

            "price\_data": price\_records

        }), 200

    except Exception as e:

        app.logger.error(f"Error fetching price data for {ticker}: {e}")

        return jsonify({"error": "Failed to load price data"}), 500

# -------------------------------------------------------------------------

# ====================== SCORING AND LLM ENDPOINTS ===============================

# -------------------------------------------------------------------------

# endpoint to START the analysis job

@app.route("/api/reits/<string:ticker>/start-analysis", methods=['POST'])

def start\_stability\_analysis(ticker):

    """

    Starts the stability analysis task in the background.

    Immediately returns a task ID.

    """

    task = generate\_stability\_analysis\_task.delay(ticker)

    return jsonify({"task\_id": task.id}), 202

# endpoint to CHECK THE STATUS and GET THE RESULT of the analysis job

@app.route("/api/reits/analysis-result/<string:task\_id>", methods=['GET'])

def get\_analysis\_result(task\_id):

    """

    Checks the status of a background task.

    Returns the result if the task is complete.

    """

    task\_result = AsyncResult(task\_id, app=generate\_stability\_analysis\_task.app)

    if task\_result.successful():

        result = task\_result.get()

        # NEW: Check for our custom "DELISTED" status from the worker

        if result.get("status") == "DELISTED":

            return jsonify(result), 200

        # Existing check for other internal errors

        if result.get("error"):

            return jsonify({"status": "FAILURE", "error": result["error"]}), 200

        # If no errors, it's a success

        return jsonify({

            "status": "SUCCESS",

            "result": result

        }), 200

    elif task\_result.failed():

        return jsonify({

            "status": "FAILURE",

            "error": str(task\_result.info) # Get the exception info

        }), 200

    else:

        # Task is still pending or in another state

        return jsonify({"status": "PENDING"}), 202

# -------------------------------------------------------------------------

# ====================== Stripe ENDPOINTS ===============================

# -------------------------------------------------------------------------

@app.route('/api/create-checkout-session', methods=['POST'])

def create\_checkout\_session():

    # Determine the domain dynamically based on the environment

    if os.environ.get('FLASK\_ENV') == 'production':

        YOUR\_DOMAIN = 'https://www.viserra-group.com'

    else:

        YOUR\_DOMAIN = 'http://localhost:3000'

    data = request.json or {}

    user\_email = data.get("email")

    if not user\_email:

        return jsonify({'error': 'User email is required to create a session.'}), 400

    try:

        session = stripe.checkout.Session.create(

            payment\_method\_types=['card'],

            mode='subscription',

            line\_items=[{

                'price': 'price\_1R5WryL1vfYfs767GYSqHKn0',  # Your Premium Plan Price ID

                'quantity': 1,

            }],

            client\_reference\_id=user\_email,

            # Use an f-string to build the URL with the correct domain

            success\_url=f"{YOUR\_DOMAIN}/pricing?status=success",

            cancel\_url=f"{YOUR\_DOMAIN}/pricing?status=cancel"

        )

        return jsonify({'url': session.url})

    except Exception as e:

        print("Stripe Error:", str(e))

        return jsonify({'error': str(e)}), 500

@app.route('/api/stripe-webhook', methods=['POST'])

def stripe\_webhook():

    payload = request.data

    sig\_header = request.headers.get('Stripe-Signature')

    try:

        event = stripe.Webhook.construct\_event(

            payload, sig\_header, STRIPE\_WEBHOOK\_SECRET

        )

    except ValueError as e:

        return 'Invalid payload', 400

    except stripe.error.SignatureVerificationError as e:

        return 'Invalid signature', 400

    if event['type'] == 'checkout.session.completed':

        session = event['data']['object']

        user\_email = session.get('client\_reference\_id')

        if not user\_email:

            print("🔥 Webhook error: No client\_reference\_id in session.")

            return "Webhook Error: Missing user identifier", 400

        try:

            if not firebase\_admin.\_apps:

                raw\_cred = os.getenv("FIREBASE\_SERVICE\_ACCOUNT")

                cred\_json = json.loads(raw\_cred)

                cred = credentials.Certificate(cred\_json)

                firebase\_admin.initialize\_app(cred)

            db\_fs = admin\_firestore.client()

            users\_ref = db\_fs.collection("users")

            query = users\_ref.where("email", "==", user\_email).limit(1)

            docs = query.stream()

            user\_doc = next(docs, None)

            if user\_doc:

                user\_doc.reference.update({"plan": "premium"})

                print(f"✅ Successfully upgraded user {user\_email} to premium.")

            else:

                print(f"🔥 Webhook error: User not found with email {user\_email}.")

        except Exception as e:

            print(f"🔥 Firebase update error in webhook: {e}")

            return "Server error during user update", 500

    return 'Success', 200

# -------------------------------------------------------------------------

# =========================== Peer Scatter ENDPOINTS ==============================

# -------------------------------------------------------------------------

@app.route("/api/peer-scatter", methods=["GET"])

def get\_peer\_scatter():

    """

    Returns peer scatter data for all REITs whose Property\_Type includes the requested property type.

    The endpoint expects a query parameter 'property\_type'.

    It returns an array of objects: { "ticker": <Ticker>, "x": <Stability Percentile>, "y": <Fundamental Percentile> }

    """

    property\_type = request.args.get("property\_type")

    if not property\_type:

        return jsonify({"error": "property\_type parameter is required"}), 400

    try:

        with db.engine.connect() as conn:

            # Query business data to get tickers matching the property type (using a LIKE query)

            query\_business = text("""

                SELECT Ticker

                FROM reit\_business\_data

                WHERE Property\_Type LIKE :prop

            """)

            business\_df = pd.read\_sql(query\_business, conn, params={"prop": f"%{property\_type}%"})

            if business\_df.empty:

                return jsonify([])  # No REITs found for this property type

            # Get unique tickers

            tickers = tuple(business\_df["Ticker"].unique())

            # Ensure tickers is a tuple (if only one, force a tuple with a trailing comma)

            if len(tickers) == 1:

                tickers = (tickers[0],)

            # Query scoring analysis data for these tickers

            query\_scoring = text("""

                SELECT Ticker, `Stability Percentile` AS stability, `Fundamental\_Percentile` AS fundamental

                FROM reit\_scoring\_analysis

                WHERE Ticker IN :tickers

            """)

            scoring\_df = pd.read\_sql(query\_scoring, conn, params={"tickers": tickers})

            # Build the output list

            result = []

            for \_, row in scoring\_df.iterrows():

                if row["stability"] is not None and row["fundamental"] is not None:

                    result.append({

                        "ticker": row["Ticker"],

                        "x": float(row["stability"]),

                        "y": float(row["fundamental"])

                    })

            return jsonify(result)

    except Exception as e:

        app.logger.error("Error in get\_peer\_scatter: " + str(e))

        return jsonify({"error": str(e)}), 500

# -------------------------------------------------------------------------

# ====================== REC ENDPOINTS ===============================

# -------------------------------------------------------------------------

@app.route('/api/rec/universe', methods=['GET'])

def get\_rec\_universe():

    """

    Returns a list of all Real Estate Crowdfunding vehicles

    with basic info from the 'rec\_universe' table.

    """

    try:

        with db.engine.connect() as conn:

            query = "SELECT \* FROM rec\_universe"

            universe\_df = pd.read\_sql(query, conn)

    except Exception as e:

        app.logger.error(f"Error loading REC universe data: {e}")

        return jsonify({"error": "Failed to load REC Universe data"}), 500

    if universe\_df.empty:

        return jsonify({"message": "No REC vehicles found.", "rec\_universe": []}), 200

    # Replace NaN values with None for safe JSON serialization

    universe\_df = universe\_df.astype(object).where(pd.notna(universe\_df), None)

    # Convert DataFrame to a list of dicts

    rec\_universe\_list = universe\_df.to\_dict(orient='records')

    return jsonify({"rec\_universe": rec\_universe\_list}), 200

@app.route("/api/rec/<string:investment\_vehicle>/performance", methods=['GET'])

def get\_rec\_performance(investment\_vehicle):

    """

    Returns time-series data (e.g., total return, NAV growth, distribution yield)

    for the specified REC vehicle. The actual DB columns may have underscores

    instead of spaces, so we automatically replace spaces with underscores

    before looking for the column.

    """

    # 1) Convert spaces to underscores to match your DB column naming convention

    col\_name = investment\_vehicle.replace(' ', '\_')

    try:

        with db.engine.connect() as conn:

            # Load each table

            df\_return = pd.read\_sql("SELECT \* FROM rec\_total\_return", conn)

            df\_distribution = pd.read\_sql("SELECT \* FROM rec\_distribution\_yield", conn)

            df\_nav = pd.read\_sql("SELECT \* FROM rec\_nav\_growth", conn)

    except Exception as e:

        app.logger.error(f"Error loading REC time-series tables: {e}")

        return jsonify({"error": "Failed to load one or more REC tables"}), 500

    if df\_return.empty and df\_distribution.empty and df\_nav.empty:

        return jsonify({"message": "No time-series data available for any vehicle."}), 200

    data\_out = {

        "vehicle": investment\_vehicle,

        "total\_return": [],

        "distribution\_yield": [],

        "nav\_growth": []

    }

    def extract\_series(df\_wide, column):

        """

        Convert wide-format DF into a list of {date, value},

        stripping '%' if found and converting to float.

        """

        if df\_wide.empty or column not in df\_wide.columns:

            return []

        df\_wide = df\_wide.copy()

        # Convert 'Dates' to datetime

        df\_wide['Dates'] = pd.to\_datetime(df\_wide['Dates'], errors="coerce")

        # Keep only date + the single vehicle column, drop NA

        df\_wide = df\_wide[['Dates', column]].dropna(subset=[column])

        # Strip '%' and convert to float

        df\_wide[column] = (

            df\_wide[column]

            .astype(str)

            .apply(pd.to\_numeric, errors='coerce')

        )

        df\_wide.dropna(subset=[column], inplace=True)

        # Sort by date ascending

        df\_wide.sort\_values(by='Dates', inplace=True)

        results = []

        for \_, row in df\_wide.iterrows():

            results.append({

                "date": row['Dates'].strftime('%Y-%m-%d') if not pd.isna(row['Dates']) else None,

                "value": row[column]

            })

        return results

    # Extract from each table

    data\_out["total\_return"] = extract\_series(df\_return, col\_name)

    data\_out["distribution\_yield"] = extract\_series(df\_distribution, col\_name)

    data\_out["nav\_growth"] = extract\_series(df\_nav, col\_name)

    # If all are empty, no match

    if not data\_out["total\_return"] and not data\_out["distribution\_yield"] and not data\_out["nav\_growth"]:

        return jsonify({"message": f"No timeseries data found for vehicle '{investment\_vehicle}'"}), 200

    return jsonify(data\_out), 200

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

App.js

// App.js

import React, { useState, useEffect } from "react";

import { BrowserRouter as Router, Routes, Route, useLocation } from "react-router-dom";

import \* as ReactGaModule from "react-ga4";

import HomePage from "./pages/HomePage.js";

import FilterPage from "./pages/FilterPage.js";

import DetailPage from "./pages/DetailPage.js";

import CrowdfundingPage from "./pages/Crowdfunding.js";

import RecDetailPage from "./pages/RecDetailPage.js";

import AboutUs from "./pages/AboutUs.js";

import ContactUs from "./pages/ContactUs.js";

import PricingPage from "./pages/Pricing.js";

import Login from "./pages/Login.js";

import Signup from "./pages/Signup.js";

import Useraccount from "./pages/Useraccount.js";

import Header from "./components/Header.js";

import { auth } from "./firebase.js";

import { onAuthStateChanged } from "firebase/auth";

import ClerkSignInPage from "./pages/ClerkSignInPage.js";

import ClerkSignUpPage from "./pages/ClerkSignUpPage.js";

import "./App.css";

// Log to see exactly what's in ReactGaModule

console.log("ReactGaModule is:", ReactGaModule);

console.log("ReactGaModule.default is:", ReactGaModule.default);

// The second .default is your GA4 instance

const realGA = ReactGaModule.default.default;

// Now you can call .initialize on the real GA object

const TRACKING\_ID = "G-HH9G61RW3G"; // Your GA4 measurement ID

realGA.initialize(TRACKING\_ID);

// Track pageviews on route change

function AnalyticsTracker() {

  const location = useLocation();

  useEffect(() => {

    realGA.send({ hitType: "pageview", page: location.pathname });

  }, [location]);

  return null;

}

function App() {

  // Track the user's plan here in App, so Header can fill it, and DetailPage can use it

  const [userPlan, setUserPlan] = useState(null);

  const [currentUser, setCurrentUser] = useState(null);

  // Listen for auth changes at the top level of the app

  useEffect(() => {

    const unsubscribe = onAuthStateChanged(auth, (user) => {

      console.log("%cAuth state changed in App.js!", "color: blue; font-weight: bold;", user); //

      setCurrentUser(user); // Set the user object when auth state changes

    });

    // Cleanup subscription on unmount

    return () => unsubscribe();

  }, []);

  return (

    <div className="App">

      <Router>

        <AnalyticsTracker />

        {/\* 1) Render Header, passing userPlan & setUserPlan so it can update the plan \*/}

        <Header currentUser={currentUser} userPlan={userPlan} setUserPlan={setUserPlan} />

        <Routes>

          <Route path="/" element={<HomePage />} />

          <Route path="/filter" element={<FilterPage />} />

          {/\* 2) Pass userPlan to DetailPage for content gating \*/}

          <Route path="/reits/:ticker" element={<DetailPage userPlan={userPlan} />} />

          <Route path="/Crowdfunding" element={<CrowdfundingPage />} />

          <Route path="/Crowdfunding/:vehicle" element={<RecDetailPage userPlan={userPlan} />} />

          <Route path="/about" element={<AboutUs />} />

          <Route path="/contact" element={<ContactUs />} />

          <Route path="/pricing" element={<PricingPage currentUser={currentUser} userPlan={userPlan} />} />

          <Route path="/user" element={<Useraccount />} />

          <Route path="/login" element={<Login currentUser={currentUser} setCurrentUser={setCurrentUser} />} />

          <Route path="/signup" element={<Signup currentUser={currentUser} />} />

          <Route path="/clerk-signin" element={<ClerkSignInPage />} />

          <Route path="/clerk-signup" element={<ClerkSignUpPage />} />

        </Routes>

      </Router>

    </div>

  );

}

export default App;