ChatGPT

GlowBot Rebuild – Trend-Aware Affiliate Content Engine

GlowBot is an AI-powered affiliate marketing content engine for skincare and beauty products. We will rebuild GlowBot with a **modular, trend-aware architecture** that injects real-time social data into GPT-4 prompts. The project uses a **monorepo** (via pnpm workspaces) containing a Node.js/Express backend (with OpenAI integration and scrapers), a Next.js + Tailwind CSS frontend (an admin dashboard), a set of platform scrapers, and shared utilities. This scaffold emphasizes scalability (SaaS readiness), clean code structure, and the ability to evolve with new features. Below is the complete project structure, with each file's purpose and content:

Project Structure

```
glowbot/                # Monorepo root (pnpm workspace)
├── package.json        # Root package for workspaces and shared scripts
├── pnpm-workspace.yaml # Defines workspace folders
├── Dockerfile          # Container setup for deployment
├── .env.example        # Example environment variables (for reference)
├── backend/            # Backend Express app (OpenAI, routes, scrapers usage)
│   ├── package.json
│   ├── config.js       # Configuration (API keys, settings) using Replit
secrets
│   ├── index.js        # Express server setup, routing, middleware
│   ├── routes/         # Express route handlers
│   │   ├── generate.js     # POST /api/generate (content generation)
│   │   └── trending.js     # GET /api/trending and /api/scraper-health (trend
data)
│   └── services/
│       └── contentGenerator.js  # GPT-4 prompt templates and OpenAI calls
├── scrapers/           # Modular scrapers for various platforms
│   ├── package.json
│   ├── index.js        # Aggregates all scraper results into one trending list
│   ├── tiktok.js       # Placeholder TikTok scraper
│   ├── reddit.js       # Placeholder Reddit scraper
│   ├── youtube.js      # Placeholder YouTube scraper
│   ├── instagram.js    # Placeholder Instagram scraper
│   └── amazon.js       # Placeholder Amazon scraper
├── shared/             # Shared utilities (constants, types, etc.)
│   ├── package.json
│   └── constants.js    # Shared constants (e.g. tone options, template types)
└── frontend/           # Next.js frontend for the admin dashboard
```

```
    ├── package.json
    ├── next.config.js      # Next.js configuration (for static export)
    ├── tailwind.config.js  # Tailwind CSS configuration
    ├── postcss.config.js   # PostCSS configuration for Tailwind
    ├── styles/
    │   └── globals.css     # Global CSS (Tailwind base imports)
    ├── pages/
    │   ├── _app.js         # Custom App to include global styles
    │   └── index.js        # Dashboard UI: trending list, generation controls
    └── components/
        └── ProductItem.js  # (Optional) UI component for a product entry
```

Below we detail each part of the project with code scaffolding and explanations.

Root Configuration and Scripts

File: `package.json` **(root)** – This defines the monorepo workspaces and common scripts. It uses **pnpm workspaces** to include the `backend`, `frontend`, `scrapers`, and `shared` folders. We also define convenient scripts for development, build, and start. The dev script uses `concurrently` to run frontend and backend together for local development. The build script runs each package's build (frontend will compile and export the Next.js app to static files, backend has no build step but we include a placeholder). The start script in production runs the backend server (which also serves the static frontend).

```
{
  "name": "glowbot-monorepo",
  "private": true,
  "version": "1.0.0",
  "workspaces": [ "backend", "frontend", "scrapers", "shared" ],
  "scripts": {
    "dev": "concurrently -n backend,frontend \"pnpm --filter backend dev\"
\"pnpm --filter frontend dev\"",
    "build": "pnpm --filter frontend build && pnpm --filter backend build",
    "start": "pnpm --filter backend start"
  },
  "devDependencies": {
    "concurrently": "^8.0.0"
  }
}
```

File: `pnpm-workspace.yaml` – Lists the workspace folders for pnpm to recognize. This is another way to declare workspaces (in addition to package.json). We include all four subdirectories:

```
packages:
  - backend
```

```
  - frontend
  - scrapers
  - shared
```

File: `.env.example` – A sample environment configuration. In Replit, actual secrets are set via the Secrets tab (as environment variables). This file documents the required env vars like OpenAI API key and any others (e.g., if we had API keys for scrapers or Stripe). It should **not** contain real secrets, just placeholders:

```
# .env.example (for reference only - use Replit Secrets for actual values)
OPENAI_API_KEY="sk-XXXXXXXX..."    # OpenAI API Key (stored securely in Replit)
# Add other API keys or configuration values as needed, e.g.:
# TIKTOK_API_KEY="..."
# REDDIT_API_TOKEN="..."
# etc.
```

Backend – Node.js Express Server and API

The backend is a **Node.js + Express** application that handles API requests, integrates with scrapers and OpenAI, and implements caching and error handling. It is structured into route modules and a service for content generation. The backend reads configuration from environment variables (using Replit secrets for sensitive data), and is designed with future SaaS features in mind (e.g. easy integration of auth, metering, etc.).

File: `backend/package.json` – Defines the backend package and its dependencies. Key dependencies include Express for the server, OpenAI for AI integration, and Morgan for logging HTTP requests. We also include `nodemon` as a dev dependency for hot-reloading during development. The scripts include `dev` (which runs the app with nodemon for development) and `start` (which runs the server in production). We list `@glowbot/scrapers` and `@glowbot/shared` as dependencies so we can import the scrapers and shared constants via the workspace.

```
{
  "name": "@glowbot/backend",
  "version": "1.0.0",
  "private": true,
  "type": "commonjs",
  "main": "index.js",
  "dependencies": {
    "express": "^4.18.2",
    "openai": "^4.0.0",          // OpenAI Node SDK for GPT-4
    "morgan": "^1.10.0",         // HTTP request logging middleware
    "@glowbot/scrapers": "workspace:*",
    "@glowbot/shared": "workspace:*"
  },
  "devDependencies": {
```

```
    "nodemon": "^2.0.22"
  },
  "scripts": {
    "dev": "nodemon index.js",
    "start": "node index.js",
    "build": "echo \"No build step for backend\""
  }
}
```

File: `backend/config.js` – Central configuration file for the backend. It loads environment variables (from Replit's Secrets or a .env file in other environments) and exports configuration constants for use in the app. This makes it easy to adjust settings or switch out API keys without changing code. In a future SaaS scenario, this can be extended to include configuration per environment (development vs production) or user-specific settings. For now, it just pulls in the OpenAI API key and any other needed variables:

```
// backend/config.js
require('dotenv').config();  // Load .env if present (mainly for non-Replit
local env)

module.exports = {
  OPENAI_API_KEY: process.env.OPENAI_API_KEY || "",
  PORT: process.env.PORT || 3000,
  // Future: include other config like DB connection strings, Stripe keys, etc.
};
```

File: `backend/index.js` – The main entry point of the Express server. This sets up the Express app, applies middleware, and mounts the API routes. Key points: - We use `morgan('dev')` logging middleware for request logging (useful for debugging and monitoring). - We enable `express.json()` to parse JSON request bodies. - The routes are mounted under `/api` (for example, the generate route will be `/api/generate`). - After API routes, we serve the built frontend static files (from the Next.js export) so that the admin dashboard is available. We use `express.static` to serve the `frontend/out` directory, which contains the static HTML, JS, and CSS from Next.js. - We include basic error handling: a catch-all for unknown routes returning 404, and a generic error handler to catch any server errors and respond with a 500 status. - The server listens on the configured port (default 3000). In Replit, this port is automatically used for the web view.

```
// backend/index.js
const express = require('express');
const path = require('path');
const morgan = require('morgan');
const { PORT } = require('./config');

// Import routes
const generateRouter = require('./routes/generate');
const trendingRouter = require('./routes/trending');
```

```
const app = express();

// Middleware
app.use(morgan('dev'));
app.use(express.json());

// Mount API routes under /api
app.use('/api/generate', generateRouter);
app.use('/api/trending', trendingRouter);
// (Note: trendingRouter will also handle /api/scraper-health as defined within
it)

// Serve frontend static files (Next.js exported app)
const frontendDir = path.join(__dirname, '..', 'frontend', 'out');
app.use(express.static(frontendDir));
// Serve index.html for any unknown route (so refreshing on client-side routes
works)
app.get('*', (req, res) => {
  res.sendFile(path.join(frontendDir, 'index.html'));
});

// Basic error handling middleware
app.use((err, req, res, next) => {
  console.error("Server Error:", err);
  res.status(500).json({ error: 'Internal Server Error', details:
err.message });
});

// Start the server
app.listen(PORT, () => {
  console.log(`  GlowBot backend running on port ${PORT}`);
});
```

File: `backend/routes/generate.js` – Defines the `/api/generate` POST endpoint. This route receives a JSON body like `{ "product": "...", "templateType": "...", "tone": "..." }`. It uses the content generation service to produce AI content based on the requested template type (original, comparison, or caption) and tone. We integrate real-time data by fetching the latest trending data for the specified product (so that the prompt has current context). The route: - Validates the request body (checks that product name is provided; if templateType or tone are missing, we use defaults). - Uses the `contentGenerator` service to produce the content by calling the appropriate function (`generateOriginal`, `generateComparison`, or `generateCaption`). - Implements caching: if we recently generated content for the same product/template/tone combination, we can return the cached result instead of calling OpenAI again (to save time and API quota). - Returns the AI-generated content as JSON.

```
// backend/routes/generate.js
const express = require('express');
const router = express.Router();
const scrapers = require('@glowbot/scrapers');
const { generateOriginal, generateComparison, generateCaption } = require('../
services/contentGenerator');
const { TEMPLATE_TYPES, TONE_OPTIONS } = require('@glowbot/shared');

// Simple in-memory cache for generation results (to avoid duplicate OpenAI
calls)
const generationCache = new Map();  // Key: `${product}|${templateType}|${tone}
`, Value: generated text

router.post('/', async (req, res) => {
  try {
    let { product, templateType, tone } = req.body;
    if (!product) {
      return res.status(400).json({ error: "Missing 'product' in request
body" });
    }
    product = product.trim();
    // Default templateType and tone if not provided
    templateType = templateType ? templateType.toLowerCase() : 'original';
    tone = tone ? tone.toLowerCase() : 'friendly';

    // Validate or adjust tone/templateType against allowed values
    if (!TEMPLATE_TYPES.includes(templateType)) {
      return res.status(400).json({ error: `Invalid templateType. Must be one
of: ${TEMPLATE_TYPES.join(', ')}` });
    }
    // For tone, we allow any string (OpenAI can handle variety), but we have
some suggested options
    if (!TONE_OPTIONS.map(t => t.toLowerCase()).includes(tone)) {
      console.warn(`Tone '${tone}' is not one of default options, but
proceeding.`);
    }

    const cacheKey = `${product}|${templateType}|${tone}`;
    if (generationCache.has(cacheKey)) {
      // Return cached content to save API calls (cache could be cleared
periodically in real use)
      return res.json({ product, templateType, tone, content:
generationCache.get(cacheKey), cached: true });
    }

    // Get current trending data for this product to inject into prompt (if
available)
```

```
    let trendData = null;
    const allTrends = await scrapers.getTrending();
    const found = allTrends.find(item => item.name.toLowerCase() ===
product.toLowerCase());
    if (found) {
      trendData = found.sources;  // object containing data from various
platforms
    }

    // Generate content using the appropriate template
    let content;
    if (templateType === 'original') {
      content = await generateOriginal(product, tone, trendData);
    } else if (templateType === 'comparison') {
      content = await generateComparison(product, tone, trendData, allTrends);
    } else if (templateType === 'caption') {
      content = await generateCaption(product, tone, trendData);
    }

    // Cache and return the result
    generationCache.set(cacheKey, content);
    return res.json({ product, templateType, tone, content });
  } catch (err) {
    console.error("Generation error:", err);
    res.status(500).json({ error: 'Failed to generate content', details:
err.message });
  }
});

module.exports = router;
```

File: `backend/routes/trending.js` – Defines two endpoints: **GET** `/api/trending` and **GET** `/api/scraper-health`. - `/api/trending` gathers trending product data from all scrapers and returns a deduplicated list of trending products across platforms. We implement **caching** here as well: the scrapers can be slow or have rate limits, so we cache the combined trending results for a short time (e.g., a few minutes) to avoid hitting the scrapers on every request. The combined data structure for each product includes the product name and a `sources` object with per-platform info. - `/api/scraper-health` provides a quick check on each scraper's status. It calls each scraper's `getTrending()` in a try/catch and reports which succeeded or failed. In a real scenario, this could be optimized to not fully scrape but rather perform a lightweight check (or use a cached result) – but here we demonstrate the structure.

```
// backend/routes/trending.js
const express = require('express');
const router = express.Router();
const scrapers = require('@glowbot/scrapers');
```

```
// Simple in-memory cache for trending results
let trendingCache = {
  timestamp: 0,
  data: []
};
const TRENDING_CACHE_TTL = 5 * 60 * 1000; // cache for 5 minutes

// GET /api/trending - get combined trending product list
router.get('/', async (req, res) => {
  try {
    const now = Date.now();
    if (now - trendingCache.timestamp < TRENDING_CACHE_TTL &&
trendingCache.data.length > 0) {
      return res.json({ cached: true, trends: trendingCache.data });
    }
    const combinedTrends = await scrapers.getTrending();  // fetch fresh data
from all scrapers
    trendingCache = { timestamp: Date.now(), data: combinedTrends };
    return res.json({ cached: false, trends: combinedTrends });
  } catch (err) {
    console.error("Error fetching trending data:", err);
    res.status(500).json({ error: 'Failed to fetch trending products', details:
err.message });
  }
});

// GET /api/scraper-health - check health of each scraper
router.get('/scraper-health', async (req, res) => {
  const health = {};
  // We attempt to fetch a small set of trending data from each scraper
individually
  const scraperList = ['tiktok', 'reddit', 'youtube', 'instagram', 'amazon'];
  for (let name of scraperList) {
    try {
      const module = require(`@glowbot/scrapers/${name}`);  // require each
scraper module
      // Attempt to get trending (could limit to quick check if implemented)
      await module.getTrending(1);  // we could pass a limit parameter if
supported
      health[name] = 'ok';
    } catch (e) {
      console.error(`${name} scraper health check failed:`, e.message);
      health[name] = 'error';
    }
  }
  return res.json(health);
});
```

```
module.exports = router;
```

File: `backend/services/contentGenerator.js` – This module implements the **AI content generation** using OpenAI's API. It exports three async functions: `generateOriginal`, `generateComparison`, and `generateCaption`. Each function crafts a **prompt (or chat messages)** tailored to that content type and includes any available real-time data about the product: - We initialize the OpenAI API client using the API key from config. - For each content type, we create a prompt with the requested tone and incorporate `trendData` (the aggregated info from scrapers) if provided. For example, we might mention the product's Amazon rating or a trending TikTok stat in the prompt to guide GPT-4. - We use the ChatGPT model (GPT-4) via `createChatCompletion`, with a system message to set the style/tone guidelines and a user message to specify the task and data. - The OpenAI responses are returned as plain text content.

These are **basic prompt templates**; in a real system, you might refine these prompts or even use few-shot examples for better outputs. Also, error handling and retry logic is included: if the OpenAI API call fails or times out, we catch it and (optionally) could retry once.

```
// backend/services/contentGenerator.js
const { OPENAI_API_KEY } = require('../config');
const { Configuration, OpenAIApi } = require('openai');

// Initialize OpenAI API client
const openaiConfig = new Configuration({ apiKey: OPENAI_API_KEY });
const openai = new OpenAIApi(openaiConfig);

// Helper: extract some key data from trendData for use in prompts
function formatTrendData(trendData) {
  if (!trendData) return "";
  const lines = [];
  if (trendData.Amazon) {
    if (trendData.Amazon.rating) lines.push(`Amazon rating: $
{trendData.Amazon.rating}/5`);
    if (trendData.Amazon.topReview) lines.push(`Top Amazon review: "$
{trendData.Amazon.topReview}"`);
  }
  if (trendData.TikTok) {
    if (trendData.TikTok.views) lines.push(`Trending on TikTok with $
{trendData.TikTok.views} views`);
  }
  if (trendData.Reddit) {
    if (trendData.Reddit.mentions) lines.push(`Mentioned in skincare Reddit $
{trendData.Reddit.mentions} times`);
  }
  // ... include other platform data similarly
  return lines.join("\n");
}
```

```
async function generateOriginal(product, tone, trendData) {
  // Prepare prompt for an original content piece (e.g., a UGC-style review or
script)
  const trendInfo = formatTrendData(trendData);
  const systemPrompt = `You are a creative copywriter specializing in beauty
products. Write an original, engaging content piece about the product "$
{product}" in a ${tone} tone. The content should feel authentic and user-
generated.`;
  let userPrompt = `Product: ${product}\nTone: ${tone}\n`;
  if (trendInfo) {
    userPrompt += `Context:\n${trendInfo}\n`;
  }
  userPrompt += "Write a detailed review or script highlighting the product's
features and benefits, and why it's trending.";

  try {
    const response = await openai.createChatCompletion({
      model: 'gpt-4',
      messages: [
        { role: 'system', content: systemPrompt },
        { role: 'user', content: userPrompt }
      ]
    });
    const content = response.data.choices[0].message.content.trim();
    return content;
  } catch (err) {
    console.error("OpenAI API error (original):", err);
    throw err;
  }
}

async function generateComparison(product, tone, trendData, allTrends) {
  // Choose another product to compare with (e.g., another trending product)
  let compareWith = "another similar product";
  if (allTrends && allTrends.length > 1) {
    const other = allTrends.find(p => p.name.toLowerCase() !==
product.toLowerCase());
    if (other) compareWith = other.name;
  }
  const trendInfo = formatTrendData(trendData);
  const systemPrompt = `You are an expert at writing product comparisons.
Compare "${product}" with "${compareWith}" in a ${tone} tone. Use casual,
informative language as if written by a skincare enthusiast.`;
  let userPrompt = `Compare the product "${product}" with "${compareWith}".\n`;
  if (trendInfo) {
    userPrompt += `Relevant data:\n${trendInfo}\n`;
  }
```

```
  userPrompt += "Discuss how they differ in terms of benefits, ingredients,
price, and user feedback.";

  try {
    const response = await openai.createChatCompletion({
      model: 'gpt-4',
      messages: [
        { role: 'system', content: systemPrompt },
        { role: 'user', content: userPrompt }
      ]
    });
    return response.data.choices[0].message.content.trim();
  } catch (err) {
    console.error("OpenAI API error (comparison):", err);
    throw err;
  }
}

async function generateCaption(product, tone, trendData) {
  // Generate a short social media caption
  const trendInfo = formatTrendData(trendData);
  const systemPrompt = `You are a social media copywriter. Create a short,
catchy caption for "${product}" in a ${tone} tone. It should sound like an
influencer wrote it, with an engaging style.`;
  let userPrompt = `Write an Instagram/TikTok caption for "${product}" in a $
{tone} tone.\n`;
  if (trendInfo) {
    userPrompt += `Context:\n${trendInfo}\n`;
  }
  userPrompt += "Include a question or call-to-action, and use a couple of
emojis or hashtags relevant to skincare.";

  try {
    const response = await openai.createChatCompletion({
      model: 'gpt-4',
      messages: [
        { role: 'system', content: systemPrompt },
        { role: 'user', content: userPrompt }
      ]
    });
    return response.data.choices[0].message.content.trim();
  } catch (err) {
    console.error("OpenAI API error (caption):", err);
    throw err;
  }
}

module.exports = { generateOriginal, generateComparison, generateCaption };
```

Note: The OpenAI API key is pulled from `config.js`. In a Replit environment, ensure you add `OPENAI_API_KEY` in the Secrets. The prompt templates here are simple; they can be fine-tuned over time or replaced with prompt engineering best practices or few-shot examples to improve output quality. We included each content type as a separate function for clarity and easy maintenance.

Scrapers – Modular Trending Data Gatherers

The **scrapers** module provides an interface to gather trending product data from various platforms (TikTok, Reddit, YouTube, Instagram, Amazon). Each platform's scraper is designed to be independent and exchangeable, exposing a uniform function `getTrending()`. In this scaffold, we provide placeholder implementations – in a real system, these would use platform APIs or web scraping to get actual data (e.g., top trending skincare products on TikTok, popular threads on Reddit's skincare subreddit, etc.).

All scrapers return data in a **common format**: an array of objects, each representing a trending product with some details:

```
{ name: "Product Name", platform: "PlatformName", url: "...", metric: 123, ... }
```

The fields can vary by platform (for instance, TikTok might return view count, Reddit might return mention count, Amazon might return rating or rank). The **scrapers/index.js** aggregator will merge these into a unified list per product.

File: `scrapers/package.json` – Defines the scrapers package. It has no dependencies except perhaps some HTTP library for real fetching (we include `axios` as a placeholder for making API calls to illustrate). The package name is `@glowbot/scrapers` so it can be imported in the backend. We also set `"type": "commonjs"` since we use CommonJS require in this context.

```
{
  "name": "@glowbot/scrapers",
  "version": "1.0.0",
  "private": true,
  "type": "commonjs",
  "main": "index.js",
  "dependencies": {
    "axios": "^1.4.0"
  }
}
```

File: `scrapers/index.js` – The **aggregator** that combines all platform scrapers. It imports each scraper module and calls their `getTrending()` functions (in parallel) to retrieve data. It then **deduplicates and merges** the results by product name: if a product appears in multiple sources, it consolidates those entries into one object with a `sources` map. For example, a result might look like:

```
{
  name: "CoolSkin Serum",
  sources: {
    TikTok: { views: 200000, url: "https://..."},
    Reddit: { mentions: 50, url: "https://..." },
    Amazon: { rating: 4.5, url: "https://..." }
  }
}
```

This unified structure makes it easy for the backend or AI prompts to see all relevant info about a trending product. The aggregator limits the number of items from each scraper (for example, top 5 from each) to avoid an excessive list.

```
// scrapers/index.js
const TikTok = require('./tiktok');
const Reddit = require('./reddit');
const YouTube = require('./youtube');
const Instagram = require('./instagram');
const Amazon = require('./amazon');

async function getTrending() {
  // Fetch trending data from all platforms in parallel
  const results = await Promise.allSettled([
    TikTok.getTrending(),
    Reddit.getTrending(),
    YouTube.getTrending(),
    Instagram.getTrending(),
    Amazon.getTrending()
  ]);
  const allData = [];
  // Collect fulfilled results
  results.forEach((res, idx) => {
    if (res.status === 'fulfilled') {
      allData.push(...res.value);
    } else {
      console.error(`Scraper ${idx} failed:`, res.reason);
    }
  });
  // Merge data by product name
  const merged = {};
  for (let entry of allData) {
    const nameKey = entry.name.trim();
    if (!merged[nameKey]) {
      merged[nameKey] = { name: nameKey, sources: {} };
    }
    // Use platform name as key in sources, include all other fields of entry
```

```
except name
    const { platform, ...details } = entry;
    merged[nameKey].sources[platform] = details;
  }
  // Convert merged object to array
  const mergedList = Object.values(merged);
  return mergedList;
}

// Optionally, export individual scrapers for direct use or health checks
module.exports = {
  getTrending,
  tiktok: TikTok,
  reddit: Reddit,
  youtube: YouTube,
  instagram: Instagram,
  amazon: Amazon
};
```

Next, we have each platform scraper. These are **placeholder implementations** – they return dummy data to simulate trending results. In practice, each would contain logic to fetch real data (for example, using `axios` or other libraries, possibly with official APIs or HTML parsing): - **TikTok scraper**: could call a TikTok trending hashtag API or scrape TikTok for trending beauty product mentions. - **Reddit scraper**: could use Reddit's API to search subreddits like r/SkincareAddiction for frequently mentioned products. - **YouTube scraper**: could use YouTube Data API to find popular videos about certain beauty products. - **Instagram scraper**: might use the Instagram Basic Display API or third-party service to find trending posts/tags related to products. - **Amazon scraper**: could scrape Amazon best sellers or new hot products in Beauty category, and fetch ratings or top reviews for context.

Each `getTrending(limit)` function returns an array of up to `limit` items (default some number like 5). We include a parameter `limit` in the function signature in case we want to limit results (for health checks or partial queries). In these stubs, we simply return a few hard-coded examples.

File: `scrapers/tiktok.js` – Placeholder TikTok scraper.

```
// scrapers/tiktok.js
// Simulate fetching trending beauty products on TikTok (e.g., by hashtag views)
async function getTrending(limit = 5) {
  // In a real implementation, use TikTok API or web scraping here.
  // For now, return dummy data:
  const trending = [
    { name: "Glow Serum X", platform: "TikTok", views: 1200000, url: "https://
tiktok.com/.../glow-serum-x" },
    { name: "HydraMoist Cream", platform: "TikTok", views: 950000, url:
"https://tiktok.com/.../hydramoist-cream" },
    { name: "SunSafe SPF 50", platform: "TikTok", views: 500000, url: "https://
```

```
tiktok.com/.../sunsafe-spf50" }
  ];
  return trending.slice(0, limit);
}

module.exports = { getTrending };
```

File: `scrapers/reddit.js` – Placeholder Reddit scraper.

```
// scrapers/reddit.js
// Simulate fetching trending beauty products from Reddit (e.g., mentions on
skincare forums)
async function getTrending(limit = 5) {
  // Real implementation might use Reddit API to find frequently mentioned
products in certain subreddits.
  // Dummy data:
  const trending = [
    { name: "Glow Serum X", platform: "Reddit", mentions: 34, url: "https://
reddit.com/r/SkincareAddiction/..." },
    { name: "Aloe Gel 99", platform: "Reddit", mentions: 20, url: "https://
reddit.com/r/SkincareAddiction/..." },
    { name: "HydraMoist Cream", platform: "Reddit", mentions: 15, url: "https://
reddit.com/r/SkincareAddiction/..." }
  ];
  return trending.slice(0, limit);
}

module.exports = { getTrending };
```

File: `scrapers/youtube.js` – Placeholder YouTube scraper.

```
// scrapers/youtube.js
// Simulate fetching trending beauty products on YouTube (e.g., via video
titles/tags)
async function getTrending(limit = 5) {
  // Real implementation might use YouTube Data API for popular videos in
beauty.
  const trending = [
    { name: "Radiant Foundation", platform: "YouTube", videos: 10, url:
"https://youtube.com/results?search_query=Radiant+Foundation" },
    { name: "Glow Serum X", platform: "YouTube", videos: 7, url: "https://
youtube.com/results?search_query=Glow+Serum+X" },
    { name: "Mega Mascara", platform: "YouTube", videos: 5, url: "https://
youtube.com/results?search_query=Mega+Mascara" }
  ];
```

```
  return trending.slice(0, limit);
}

module.exports = { getTrending };
```

File: `scrapers/instagram.js` – Placeholder Instagram scraper.

```
// scrapers/instagram.js
// Simulate fetching trending beauty products on Instagram (e.g., via hashtags)
async function getTrending(limit = 5) {
  // Real implementation might use Instagram API or scrape popular hashtags.
  const trending = [
    { name: "HydraMoist Cream", platform: "Instagram", posts: 800, url:
"https://instagram.com/explore/tags/hydramoistcream" },
    { name: "Vitamin C Serum", platform: "Instagram", posts: 1200, url:
"https://instagram.com/explore/tags/vitaminCserum" },
    { name: "Glow Serum X", platform: "Instagram", posts: 1500, url: "https://
instagram.com/explore/tags/glowserumx" }
  ];
  return trending.slice(0, limit);
}

module.exports = { getTrending };
```

File: `scrapers/amazon.js` – Placeholder Amazon scraper.

```
// scrapers/amazon.js
// Simulate fetching trending beauty products on Amazon (e.g., best sellers or
new releases)
async function getTrending(limit = 5) {
  // Real implementation might scrape Amazon's best sellers or use an API if
available.
  const trending = [
    { name: "SunSafe SPF 50", platform: "Amazon", rating: 4.8, url: "https://
amazon.com/dp/B00...SPF50", topReview: "No white cast and great protection!" },
    { name: "Aloe Gel 99", platform: "Amazon", rating: 4.5, url: "https://
amazon.com/dp/B00...ALOE99", topReview: "Soothing and absorbs quickly." },
    { name: "Glow Serum X", platform: "Amazon", rating: 4.7, url: "https://
amazon.com/dp/B00...GLOWX", topReview: "My skin has never looked better!" }
  ];
  return trending.slice(0, limit);
}

module.exports = { getTrending };
```

Each of these scrapers focuses on one source. The combined trending output (via the aggregator) picks up overlapping product names and merges their details. For example, `"Glow Serum X"` appears in TikTok, Reddit, YouTube, Instagram, and Amazon dummy data; the aggregator will merge these into one entry with all five sources listed. This rich dataset can be fed into the AI prompts to produce more informed content. **Error resilience**: Each scraper's errors are caught in the aggregator (`Promise.allSettled` usage ensures one failing scraper won't break the whole trend fetch). We could further improve resilience by adding retry logic within each scraper (e.g., if a network request fails, try again after a short delay), but that is omitted here for brevity.

Shared Utilities (Shared Package)

The `shared` directory contains common utilities or constants that are used by both backend and frontend, avoiding duplication. This is especially useful for things like defining allowed tone options or template types in one place, so the front-end UI and back-end validation stay in sync. We use a small `constants.js` module for this purpose.

File: `shared/package.json` – Defines the shared package. It has no external dependencies. We give it the name `@glowbot/shared` so it can be referenced by other packages.

```
{
  "name": "@glowbot/shared",
  "version": "1.0.0",
  "private": true,
  "type": "commonjs",
  "main": "constants.js"
}
```

File: `shared/constants.js` – Exports shared constant values. We define: - `TONE_OPTIONS`: a list of tone presets that the user can choose from (and that our backend recognizes). These could be styles like friendly, professional, luxury, enthusiastic, etc. - `TEMPLATE_TYPES`: the list of content template types our system supports (currently "original", "comparison", "caption").

These constants can be imported by the backend (for input validation or default values) and by the frontend (to populate dropdowns or ensure consistency). This avoids hard-coding these strings in multiple places.

```
// shared/constants.js
module.exports = {
  TONE_OPTIONS: [ "Friendly", "Professional", "Luxury", "Enthusiastic" ],
  TEMPLATE_TYPES: [ "original", "comparison", "caption" ]
};
```

(In the future, shared could also hold things like TypeScript types/interfaces, utility functions, or even a unified client for calling the backend from the frontend. For now, our focus is on simple constants.)

Frontend – Next.js Admin Dashboard (with Tailwind CSS)

The frontend is a **Next.js** application using **Tailwind CSS** for styling. It serves as a lightweight **admin dashboard** where a content creator or marketer can: - View the list of currently trending skincare/beauty products (aggregated from all sources). - Select a content template type (Original, Comparison, Caption) and a tone for the AI generation. - Trigger content generation for a selected product (or even batch-generate for multiple products). - View the generated content and easily copy it to clipboard or export it.

The UI is kept simple and responsive. We utilize Tailwind utility classes for quick styling. The Next.js app is configured to be exported as a static site for simplicity (since our data is fetched via the backend API dynamically).

File: `frontend/package.json` – Defines the frontend package and its dependencies. We include `next`, `react`, and `react-dom` for the Next.js framework. For styling, we have `tailwindcss`, `postcss`, and `autoprefixer`. We also depend on our shared package for constants. The scripts include the standard Next.js commands. The `build` script runs Next's build and export to generate static files in the `out` directory. (We use static export to simplify integration with our Express backend; the pages will fetch data at runtime via the API.)

```
{
  "name": "@glowbot/frontend",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build && next export",
    "start": "next start"
  },
  "dependencies": {
    "next": "13.x",   // (use the latest Next 13 or 14 version)
    "react": "18.x",
    "react-dom": "18.x",
    "@glowbot/shared": "workspace:*"
  },
  "devDependencies": {
    "tailwindcss": "^3.3.0",
    "postcss": "^8.4.21",
    "autoprefixer": "^10.4.13"
  }
}
```

File: `frontend/next.config.js` – Next.js configuration. We specify `output: 'export'` to enable static HTML export. (This means we cannot use server-side rendering or dynamic routes that aren't known at build time. Our use-case is fine since we will fetch dynamic data via API calls on the client side.) We also disable ESLint and Typescript checks during builds for simplicity (assuming a pure JS project).

```
// frontend/next.config.js
module.exports = {
  output: 'export',
  eslint: {
    ignoreDuringBuilds: true
  },
  typescript: {
    ignoreBuildErrors: true
  }
};
```

File: `frontend/tailwind.config.js` – Tailwind configuration file. It defines where Tailwind should look for class names (our pages and components directories). It also extends the default theme or plugins if needed (we leave it mostly default). Ensure this is present so that Tailwind can tree-shake unused styles.

```
// frontend/tailwind.config.js
module.exports = {
  content: [
    "./pages/**/*.{js,jsx,ts,tsx}",
    "./components/**/*.{js,jsx,ts,tsx}"
  ],
  theme: {
    extend: {
      // You can extend theme here (colors, fonts, etc.)
    }
  },
  plugins: []
};
```

File: `frontend/postcss.config.js` – Configures PostCSS with Tailwind CSS and Autoprefixer plugins, which is the standard setup for Tailwind.

```
// frontend/postcss.config.js
module.exports = {
  plugins: {
    tailwindcss: {},
    autoprefixer: {}
  }
};
```

File: `frontend/styles/globals.css` – A global stylesheet that imports Tailwind's base styles, components, and utilities. This is required for Tailwind to work. You can also add any global custom CSS here.

```
/* frontend/styles/globals.css */
@tailwind base;
@tailwind components;
@tailwind utilities;

/* Custom global styles (if any) can go here. For now, we rely on Tailwind
utilities only. */
```

File: `frontend/pages/_app.js` – Custom App component for Next.js. This is where we import the global CSS (Tailwind) so that styles are applied across the app. We also could set up any context providers or layout here if needed. Currently, we just ensure Tailwind's styles are loaded.

```
// frontend/pages/_app.js
import '../styles/globals.css';
export default function App({ Component, pageProps }) {
  return <Component {...pageProps} />;
}
```

File: `frontend/pages/index.js` – The main dashboard page. This page displays the trending products and provides controls for content generation. Key features: - On mount, it fetches the trending products from our backend (`/api/trending` endpoint). This is done on the client side using `useEffect` to ensure we have fresh data whenever the page is loaded. The trending data is stored in state. - The page includes UI controls: a dropdown to select the content template (Original, Comparison, Caption), another dropdown to select tone (using the options from `shared/constants.js`), and an optional "Generate All" button for batch generation. - Each trending product is listed, with a "Generate" button. When clicked, it will call the `/api/generate` endpoint for that product with the selected template and tone. The result is then displayed under that product. - We maintain state for generated content results (mapping product name -> generated text) and loading states. - We include a "Copy" button next to each generated output to copy the content to clipboard for easy use. - Basic styling is done with Tailwind classes (e.g., using flex, padding, margins, text styles, etc.) to keep the UI clean and responsive.

```
// frontend/pages/index.js
import { useState, useEffect } from 'react';
import { TONE_OPTIONS, TEMPLATE_TYPES } from '@glowbot/shared';

export default function Dashboard() {
  const [trending, setTrending] = useState([]);
  const [loadingTrends, setLoadingTrends] = useState(true);
  const [templateType, setTemplateType] = useState('original');
  const [tone, setTone] = useState('Friendly');
  const [generating, setGenerating] = useState({});  // track loading state per
product
  const [generatedContent, setGeneratedContent] = useState({}); // store
generated text per product
```

```
useEffect(() => {
  // Fetch trending products on component mount
  async function fetchTrending() {
    try {
      setLoadingTrends(true);
      const res = await fetch('/api/trending');
      const data = await res.json();
      const trends = data.trends || [];
      setTrending(trends);
    } catch (err) {
      console.error("Failed to fetch trending products:", err);
    } finally {
      setLoadingTrends(false);
    }
  }
  fetchTrending();
}, []);

const handleGenerate = async (productName) => {
  try {
    setGenerating(prev => ({ ...prev, [productName]: true }));
    // Call generate API for the given product
    const res = await fetch('/api/generate', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ product: productName, templateType, tone })
    });
    const data = await res.json();
    if (data.error) {
      alert(`Error: ${data.error}`);
    } else if (data.content) {
      setGeneratedContent(prev => ({ ...prev, [productName]: data.content }));
    }
  } catch (err) {
    console.error("Generation failed:", err);
    alert("Failed to generate content. Check console for details.");
  } finally {
    setGenerating(prev => ({ ...prev, [productName]: false }));
  }
};

const handleCopy = (productName) => {
  const text = generatedContent[productName];
  if (!text) return;
  navigator.clipboard.writeText(text).then(() => {
    alert('Copied content for "' + productName + '" to clipboard');
  }).catch(err => {
```

```
      console.error("Clipboard copy failed:", err);
    });
  };

  const handleGenerateAll = async () => {
    // Trigger generation for all trending products (batch generation)
    for (let item of trending) {
      // We could optimize by doing these in parallel, but doing sequentially to
avoid rate limits
      await handleGenerate(item.name);
    }
    // Note: In a real app, consider rate limiting or queueing if many items,
and better UI feedback.
  };

  return (
    <div className="min-h-screen bg-gray-50 p-4">
      <h1 className="text-2xl font-bold mb-4">□ GlowBot Dashboard</h1>

      {/* Controls */}
      <div className="flex flex-col sm:flex-row sm:items-center mb-6 gap-2">
        <div>
          <label className="mr-2 font-medium">Template:</label>
          <select
            value={templateType}
            onChange={(e) => setTemplateType(e.target.value)}
            className="border px - 2 py-1 rounded"
          >
            {TEMPLATE_TYPES.map(type => (
              <option key={type} value={type}>{type.charAt(0).toUpperCase() +
type.slice(1)}</option>
            ))}
          </select>
        </div>
        <div>
          <label className="mr-2 font-medium">Tone:</label>
          <select
            value={tone}
            onChange={(e) => setTone(e.target.value)}
            className="border px-2 py-1 rounded"
          >
            {TONE_OPTIONS.map(option => (
              <option key={option} value={option}>
                {option}
              </option>
            ))}
          </select>
        </div>
```

```
        <button
          onClick={handleGenerateAll}
          className="bg-blue-600 text-white px-4 py-2 rounded shadow hover:bg-
blue-700 disabled:opacity-50 mt-2 sm:mt-0"
          disabled={loadingTrends || trending.length === 0}
        >
          Generate All
        </button>
      </div>

      {/* Trending Products List */}
      {loadingTrends ? (
        <p>Loading trending products...</p>
      ) : trending.length === 0 ? (
        <p>No trending products found.</p>
      ) : (
        <div className="space-y-4">
          {trending.map(item => (
            <div key={item.name} className="bg-white p-4 rounded shadow">
              <div className="flex justify-between items-center">
                <div>
                  <h2 className="text-lg font-semibold">{item.name}</h2>
                  {/* Display a summary of sources for context */}
                  <p className="text-sm text-gray-600">
                    {Object.keys(item.sources).join(', ')} {/* e.g., "TikTok,
Reddit, Amazon" */}
                  </p>
                </div>
                <button
                  onClick={() => handleGenerate(item.name)}
                  className="bg-green-600 text-white px-3 py-1 rounded hover:bg-
green-700 disabled:opacity-50"
                  disabled={generating[item.name]}
                >
                  {generating[item.name] ? 'Generating...' : 'Generate'}
                </button>
              </div>
              {/* Generated content output */}
              {generatedContent[item.name] && (
                <div className="mt-3 border-t pt-3">
                  <p className="whitespace-pre-line text-
gray-800">{generatedContent[item.name]}</p>
                  <button
                    onClick={() => handleCopy(item.name)}
                    className="text-blue-600 text-sm underline mt-1"
                  >
                    Copy to clipboard
                  </button>
```

```
                </div>
              )}
            </div>
          ))}
        </div>
      )}
    </div>
  );
}
```

Explanation: The dashboard starts by loading trending products from the backend. It then presents each product with a Generate button. The user selects which template and tone they want from the dropdowns at the top. Clicking **Generate** will call the backend API and on success, display the content. The content appears below the product name, and a **Copy to clipboard** link allows quick copying. We also included a **Generate All** button that will iterate through all products and generate content for each (useful for batch generation tasks, though in practice one might implement this with proper queuing or background processing if the list is long).

We use basic Tailwind classes for styling: - The outer `div` has a gray background and padding to make the app look clean. - Each product card is a white, rounded box with a shadow (to stand out from the background). - Buttons are styled with colors (blue for global actions, green for generate) and have hover states. - The generated text is shown in a `<p>` with `whitespace-pre-line` to preserve newlines from the AI output (which often is formatted in paragraphs or lists).

This is a minimal UI for demonstration. In a production scenario, you might enhance it with modals, toast notifications, better error displays, pagination for trending list if large, etc. The focus here is to have a working interface to trigger and view the AI content generation.

Dockerfile – Containerization for Deployment

Finally, we provide a **Dockerfile** to containerize the application for scalable deployment. This Dockerfile will: - Use a Node.js base image. - Copy the monorepo code into the image. - Install dependencies using pnpm. - Build the frontend (Next.js static export). - Start the Express server (which serves the static files and APIs).

File: `Dockerfile` – A multi-stage Docker setup: 1. **Builder stage**: Installs dependencies and builds the frontend. 2. **Runner stage**: Copies the needed files and runs the backend server.

We separate build and run stages to keep the final image slim (dev dependencies not included). The static files from the Next.js export are copied into the final stage. The container runs the backend start command, which will launch the Express server (serving both the API and the static frontend).

```
# Dockerfile for GlowBot

# Stage 1: Build stage
FROM node:18-alpine AS builder
```

```
# Install pnpm
RUN npm install -g pnpm

# Set working directory
WORKDIR /app

# Copy all files (note: .dockerignore can be used to exclude dev files)
COPY . .

# Install dependencies for all workspaces
RUN pnpm install

# Build frontend (Next.js static export)
RUN pnpm --filter @glowbot/frontend run build

# Stage 2: Runner stage
FROM node:18-alpine

# Set working directory
WORKDIR /app

# Copy only necessary files from builder stage
COPY --from=builder /app/backend /app/backend
COPY --from=builder /app/frontend/out /app/frontend/out
COPY --from=builder /app/package.json /app/package.json
COPY --from=builder /app/backend/package.json /app/backend/package.json
COPY --from=builder /app/node_modules /app/node_modules

# Expose port (if needed; Replit might handle this implicitly)
EXPOSE 3000

# Start the backend server
CMD ["node", "backend/index.js"]
```

In this configuration, the entire app runs in one container on one process (Express). The static files are served by Express. For a larger-scale SaaS deployment, one might split the frontend and backend into separate services or use a more advanced process manager, but this setup is sufficient for an MVP and easier to deploy on platforms like Replit or a single VPS.

Conclusion and Next Steps

We have constructed a modular, trend-aware content engine with GlowBot. The **monorepo architecture** (using pnpm workspaces) keeps the code for backend, frontend, scrapers, and shared utilities neatly separated while still allowing them to interoperate. This makes the project scalable and maintainable as it grows: - **Dynamic Data Injection**: Our content generation uses real-time data from scrapers to make AI outputs more relevant. This can be expanded by improving scraper logic or adding new data sources. -

Admin Dashboard: The Next.js/Tailwind frontend provides an elegant interface for content creators to leverage the AI – it can be further refined with user authentication, saving of generated content, etc. - **Scraping Infrastructure**: We set up a pattern for scrapers that can be extended. As GlowBot grows, each scraper can be fleshed out (or moved to serverless functions or separate services if needed for scale). - **SaaS Readiness**: The code is structured to allow multi-tenant or user-specific features in the future (e.g., easily plugging in an Auth middleware in Express, or adding Stripe subscription checks before allowing generation). Configuration and secrets are abstracted to environment variables, which is critical for security and flexibility. - **Error Handling & Resilience**: We included caching layers, basic retry logic placeholders, and health-check endpoints, which are important for reliability in production.

With this scaffold in place, Replit's AI Agent (or any developer) can quickly spin up the MVP and then focus on the core logic: enhancing prompts, refining the scrapers, and polishing the UI/UX. GlowBot's rebuild is well-structured for iterative improvement and scaling into a full-fledged commercial SaaS product. Enjoy building and happy coding!
