# GMS Vessel Tracker

## Plan

**Quick principles before we start**

1. **MVP-first** — ship something that saves your time (Phase 1) before polishing UX or automation.
2. **Keep it simple & testable** — prefer readable code and small modules over clever hacks.
3. **Iterate in phases** — each phase should be a working product you can demo.
4. **Document everything** — commit messages, README, architecture diagrams and short dev logs — this becomes your job-winning story.

**Phase 0 — Choose & validate API (planning + small tests)**

**Goal:** Confirm Datalastic (or alternative) gives reliable IMO → destination info for the vessels you care about, and get an API key.

**Why:** avoid building on a provider with gaps in coverage or a price that surprises you.

**Tasks**

* Get a Datalastic API key and test 10–20 real IMOs from your company lists. Verify returned fields: vessel\_name, vessel\_type, destination\_port, destination\_country, eta. Use their docs to find the endpoints. ([Datalastic](https://datalastic.com/api-reference/?utm_source=chatgpt.com" \o "API Reference - Datalastic))
* Evaluate alternatives if any IMOs are missing: MarineTraffic, VesselFinder, FleetMon — compare coverage / pricing.
* Decide rate-limit / cost strategy (calls per IMO, batching support, per-request pricing).

**Deliverable / Acceptance criteria**

* A short test script (Node) that accepts sample IMOs and returns normalized JSON for each IMO (name, type, destination port, country, eta). If >90% of test IMOs return usable destination info, accept Datalastic.

**Tools / libs**

* axios or node-fetch for calls.
* Local .env for API key.

**Phase 1 — MVP: Bulk IMOs → Table (manual send / immediate ops help)**

**Goal:** Paste/upload bulk IMOs → fetch data → show a user-friendly table. Save tons of manual time.

**Features**

* Frontend: input box / CSV upload for IMOs (newline or comma-separated).
* Backend endpoint: POST /api/resolve-imos → returns normalized list.
* UI: tabular display with columns: IMO, Vessel Name, Type, Destination Port, Country, ETA, link to vessel on MarineTraffic.
* Export options: copy HTML table, download CSV, or copy markdown.
* Lightweight client-side caching of last results (optional).

**Implementation steps**

1. **Backend (Express)**
   * Route POST /api/resolve-imos receives array of IMOs.
   * Check Redis cache for IMO (if available); else call Datalastic (batched if possible).
   * Normalize response and return JSON.
   * Simple rate limit / queue per-IP to avoid accidental cost spikes.
2. **Frontend (React)**
   * Simple page: paste list or upload CSV; call endpoint; display table.
   * Provide export buttons (CSV & Copy HTML).
3. **Local testing**
   * Use Postman / Insomnia to test backend endpoints.
   * Manual QA: test with 10–50 IMOs.

**Tech / libs**

* Backend: Express, axios/node-fetch, pg or Prisma (optional), dotenv.
* Frontend: React + Tailwind/CSS (or DaisyUI if you like).
* Dev DB: None required yet — purely ephemeral; or SQLite for fast local dev.

**Testing**

* Unit test for normalization function (Jest).
* Integration test: mock Datalastic responses and call resolve-imos.

**Deliverable**

* A working UI where you paste IMOs and get a clean table you can copy/paste into mail.

**Time estimate (beginner)**: 3–7 days of focused work.

**Phase 2 — Add relational DB + company model + retention logic**

**Goal:** Persist company & vessel info so you don’t need to re-enter IMOs for each company. Add expiry logic (e.g., delete old voyage rows after 7 days).

**Features**

* Database schema for Companies, Vessels, and Voyages/PortCalls.
* Admin UI: create company, add list of vessel IMOs (bulk) and contacts.
* Query: Click a company → show its vessels & latest destination (no IMO input needed).
* Background job: refresh voyage info for stored IMOs every X hours/days (configurable).
* Data retention: automatically delete voyage/destination rows older than 7 days (or soft-delete flag).
* Basic caching (Redis) for live API hits to control costs.

**Suggested DB schema (simplified)**

-- company

CREATE TABLE companies (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL,

contact\_email TEXT,

contact\_person TEXT,

created\_at TIMESTAMP DEFAULT now()

);

-- vessel

CREATE TABLE vessels (

imo BIGINT PRIMARY KEY,

name TEXT,

type TEXT,

company\_id INT REFERENCES companies(id),

last\_seen TIMESTAMP,

created\_at TIMESTAMP DEFAULT now()

);

-- voyage / portcall

CREATE TABLE voyages (

id SERIAL PRIMARY KEY,

imo BIGINT REFERENCES vessels(imo),

destination\_port TEXT,

destination\_country TEXT,

eta TIMESTAMP,

fetched\_at TIMESTAMP DEFAULT now()

);

**Implementation steps**

* Add Postgres (remote dev env like Railway, Render or local Docker).
* Add pg (or Prisma) integration. Prisma helps if you want typed models; otherwise pg is fine.
* Build API endpoints:
  + POST /api/companies (create)
  + GET /api/companies/:id/vessels
  + POST /api/companies/:id/import-vessels (bulk IMOs)
  + POST /api/vessels/:imo/refresh
* Background worker (simple): a cron (node-cron) that reads vessels needing refresh and calls Datalastic; write results to voyages.
* Add retention: DELETE FROM voyages WHERE fetched\_at < now() - interval '7 days' (cron job).

**Caching & Redis**

* Cache per-IMO API responses for short TTL (10–30 minutes if you need freshness, or 24h if not).
* Use Redis for quick cache lookups and to power your queue (Bull/BullMQ).

**Testing**

* DB migration tests.
* End-to-end: import vessel list → verify DB rows created and can be viewed.

**Deliverable**

* A UI where you select a company and see its vessels & latest destinations without re-entering IMOs.

**Time estimate (beginner)**: 1–2 weeks.

**Notes**

* TTL of 7 days is fine; implement as a configurable env var so you can tweak later.

**Phase 3 — Email generation, preview & send (manual approval)**

**Goal:** Generate full branded HTML cold-mails automatically with the vessel table inserted; let you preview and edit before sending. Integrate transactional mail provider.

**Features**

* Email template engine using **MJML** or **Handlebars** → compile to inline CSS HTML (MJML is great for responsive marketing emails).
* Store templates in DB and allow template editing via UI (simple editor).
* Preview pane that renders compiled HTML; allow editing the table in the preview.
* One-click: Open draft in Outlook (mailto with body? For complex HTML, better: send via provider OR open a window with the raw HTML to copy into Outlook).
* Send via SendGrid (API) or SendGrid SMTP (recommended transactional). ([Twilio](https://www.twilio.com/docs/sendgrid/for-developers/sending-email/getting-started-with-transactional-emails?utm_source=chatgpt.com))
* Track sends: store entries in mail\_queue table (status: draft, queued, sent, failed) and store response info.

**Implementation steps**

* Add template model:
  + templates(id, name, mjml\_text, compiled\_html, created\_at).
* Build server-side renderer:
  + Render MJML + insert vessel\_table\_html (sanitize before inserting).
* Build preview & edit UI.
* Implement send:
  + Use SendGrid Node SDK (@sendgrid/mail) or send via SMTP using nodemailer.
  + For SendGrid dynamic templates you can use template IDs and substitution variables.
* Add logging + success/failure states.

**Testing**

* Email previews across clients: test in Gmail, Outlook webmail, and a phone. Use Litmus/Email on Acid if you want premium testing.
* Sandbox send mode in SendGrid to avoid actually emailing during dev.

**Deliverable**

* A workflow: choose company → preview email with auto-populated vessel table → edit if needed → send or save draft.

**Time estimate:** 1–2 weeks.

**Phase 4 — Dashboard & analytics (professional polish)**

**Goal:** Build a dashboard showing operational metrics, API usage, mail metrics and basic campaign analytics.

**Features**

* Dashboard panels:
  + Total companies, vessels tracked, mails sent today/this month.
  + API usage and estimated cost (calls this month vs monthly budget).
  + Per-company mail stats: mails sent, open rate (if you enable tracking), bounce rate.
  + Queue health (jobs pending, failed).
* Alerts & thresholds: if API calls > X per month, send email alert or Slack webhook.
* Role-based UI: ops vs admin.

**Implementation steps**

* Store metrics in DB or use lightweight analytics store (e.g., timeseries in Postgres or small ElasticSearch).
* Instrument mail sends with provider webhooks for bounces/opens; save events.
* Build charts in React (Recharts or Chart.js).
* Add export/report CSV.

**Testing**

* Simulate events to verify dashboard updates and alerts.

**Deliverable**

* A single-page dashboard summarizing health and activity.

**Time estimate:** 1–2 weeks.

**Phase 5 — Full automation, reliability & ops**

**Goal:** Automate scheduled sends, robust error handling, queuing, CI/CD, monitoring and make the system production-ready.

**Features**

* Daily scheduled runs:
  + For each company, select vessels with arrivals in the next N days -> compose email -> queue send batch (with rate limits).
  + Optional approval step before send for selected companies.
* Queue & workers:
  + Use BullMQ (Redis) to process email sends and API refresh jobs.
* Error handling & retries:
  + Exponential backoff for API calls; dead-letter queue for failed jobs.
* CI/CD:
  + GitHub Actions to run tests, lint, build and deploy (Vercel for frontend; Render/Heroku or Docker on VM for backend).
* Monitoring & logging:
  + Sentry for errors, PostHog/Google Analytics for product usage, Prometheus/Grafana for infra metrics (optional).
* Backups:
  + DB backups (automatic daily), and Redis persistence plan.
* Security:
  + Secrets in environment variables / secret manager; secure access to production DB and redis; HTTPS only.

**Implementation steps**

* Add worker processes that read mail queue and send in rate-limited batches.
* Add CI pipeline: tests → build → deploy to staging → deploy to production.
* Add Sentry and basic health endpoints.

**Acceptance criteria**

* System can automatically send daily batched emails without manual intervention (with retries) for a week without failures.
* Alerting triggers when API usage exceeds defined threshold.

**Time estimate:** 2–4 weeks (depending on depth of automation).

**Recommended libraries & tools (summary)**

* Backend: Node + Express (or Fastify), axios, pg or Prisma (for DB), bullmq + ioredis (queue), node-cron (simple scheduling).
* Frontend: React (create-react-app or Vite), Tailwind CSS (or DaisyUI), Recharts for charts.
* Email/template: MJML (author), @sendgrid/mail (send), or nodemailer for SMTP.
* Caching: Redis (for per-IMO short TTL).
* Tests: Jest + Supertest (backend), React Testing Library (frontend), Cypress (optional E2E).
* CI/CD: GitHub Actions; host frontend on Vercel, backend on Render/Railway/Heroku, Postgres on Railway/ElephantSQL/Render.
* Monitoring: Sentry (errors), simple logs to files or Papertrail/Logflare.

(If you want URLs/documentation for any of these, I can paste them.)

**Security & cost tips (important)**

* Keep API keys in .env and never commit them.
* Implement per-user/company rate limits to prevent accidental cost spikes.
* Start with the smallest API plan from Datalastic to test coverage (then scale). ([Datarade](https://datarade.ai/data-providers/datalastic/profile?utm_source=chatgpt.com" \o "Datalastic - Pricing, Reviews, Data & APIs - Datarade))
* Use SendGrid sandbox mode during development. ([SendGrid](https://sendgrid.com/en-us/use-cases/transactional-email?utm_source=chatgpt.com))

**Minimal API design (examples)**

**POST /api/resolve-imos**  
Request:

{ "imos": ["9381234","9312345"] }

Response:

[

{ "imo":"9381234","name":"Vessel A","type":"Bulk Carrier","destination\_port":"Jebel Ali","destination\_country":"UAE","eta":"2025-09-20T10:00:00Z" },

...

]

**POST /api/companies/:id/generate-email**

* Produces compiled HTML, returns preview and mail\_queue id.

**POST /api/mail/send/:queueId**

* Sends queued mail (used by worker).

**Testing & quality checklist**

* Unit tests for normalization/parsing logic.
* Integration tests for API endpoints (mock provider).
* Manual cross-client email tests for templates.
* Load test for API calls (to estimate cost).
* Security scan for dependencies (npm audit).

**Documentation & portfolio (how to present this project for hiring)**

1. **README**: problem statement, architecture diagram, tech stack, demo link, setup instructions.
2. **Case study blog** (LinkedIn / Medium): 3–5 posts:
   * Day 1: Problem & MVP
   * Mid-build: Challenges (rate limits, template rendering)
   * Final: Metrics & impact (e.g., “Saved X hours / week”)
3. **GitHub repo**: clean commits, issue tracker, separate branches (feature/).
4. **Live demo**: host a small staging instance; include screenshots/GIFs.
5. **Interview talking points**: be ready to discuss trade-offs (API cost vs scraping), caching TTL, queue design and how system would scale.