ChatGPT

Solving Next.js Build Errors & Finalizing Project Deployment

Diagnosing the Vercel Build Error (Module Resolution)

The **"Module not found"** errors during the Vercel build were primarily caused by improper module path resolution. In our Next.js frontend, we used the `@/` alias to reference files under `src/`, but this alias wasn't configured correctly for the build. Additionally, **case-sensitivity issues** can trigger such errors on Vercel's Linux environment (file paths that work on Windows might fail on Vercel due to letter-case mismatches [1]).

To confirm the cause, we reviewed similar cases and our project's history. The project log shows that after adjusting the TypeScript config to define the alias properly, the build issues were nearly resolved [2]. Specifically, adding a `paths` **mapping in** `tsconfig.json` for `@/*` fixed the alias resolution (with `"@/*": ["./src/*"]` mapping) [3]. This ensures Next.js can locate modules referenced with `@/`. We also verified that **filename casing** in imports exactly matches the actual files (since Vercel's filesystem is case-sensitive). For example, if our file is `errorMessages.ts`, all imports must use the same capitalization.

Solution Steps:

1. **Fix the TSConfig Path Alias:** In `frontend/tsconfig.json`, make sure we have the proper alias settings. Based on Next.js defaults, we set:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": { "@/*": ["./src/*"] }
  }
  // ...other options...
}
```

This change (already implemented) allows imports like `@/lib/...` to resolve to the `frontend/src/lib` folder. Our project log confirms that after adding this, aliases started resolving correctly in local builds [3].

1. **Revert to Using Aliases in Imports:** We had temporarily switched some imports to relative paths to troubleshoot. Now that the alias is configured, return to using `@/` imports everywhere for consistency. This was done for key files (e.g. pages and hooks), which **eliminated nearly all module-not-found errors** in the build [2].

2. **Correct Any Case Mismatch:** Double-check every import path versus the actual filename. On Vercel, `import { createError } from '@/lib/errorMessages'` will fail if the file is named `errormessages.ts` or `ErrorMessages.ts` (any casing difference). Ensure

`errorMessages.ts` exists and is spelled exactly as imported. Vercel's guidance emphasizes matching letter-casing in filenames and import statements [1]. If you renamed files or directories, update git index (e.g. via `git mv`) so the case change is committed.

3. **Fix the Remaining Problem File (`useAuth.ts`):** Our build log indicated one remaining module error related to the `useAuth` hook (complaining it couldn't resolve `../lib/errorMessages`). This suggests that file still had a wrong import path. Open `frontend/src/hooks/useAuth.ts` and make sure imports use the alias, e.g.:

   ```
   import { createError } from '@/lib/errorMessages';
   import { CSRF_HEADER } from '@/lib/csrf';
   ```

   If it was using a relative path (`../lib/errorMessages`), change it to the alias. Save and commit this change. (This step was a blocker earlier due to editor limitations, but it must be fixed now for a successful build.)

4. **Local Test:** Before pushing, run the frontend build locally: `cd frontend && npm run build`. This will catch any remaining "Module not found" errors or typographic mistakes in imports. With the alias and imports corrected, the build should pass locally. If an error still occurs, it likely points to a specific file – double-check that file's import statements and existence.

5. **Deploy via Vercel (Git Push):** Once the build is passing locally, push the changes to GitHub. This will trigger Vercel to attempt a new build. With the fixes above, the build should succeed. (We no longer need the temporary workarounds like commenting out `useAuth` usage – those were just to bypass errors. Remove any such hacks now that the underlying issue is fixed.)

   Note: In our troubleshooting, adjusting `next.config.mjs` or adding a `jsconfig.json` did not resolve the issue [4]. The definitive fix was updating `tsconfig.json` as described. If the Vercel build still fails after these steps, re-check the build logs for any new clues (e.g. another file path issue). Most commonly, any remaining "module not found" on Vercel after alias fixes would be due to an unnoticed casing discrepancy or an import pointing to a file that was moved or deleted.

Preparing the Production Environment Variables

For the deployed application to function end-to-end, we must configure environment variables on each platform (Vercel, Northflank, GitHub) correctly:

- **Backend (Northflank) Env Vars:** Ensure the AWS S3 credentials and settings are set in Northflank for the backend service. According to our `.env.example`, the backend expects `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_REGION` (default `us-east-1`), and `S3_BUCKET_NAME` to connect to S3 [5]. Double-check that these are provided in Northflank's environment config. Also set `JWT_SECRET` (as per README) and any other needed vars (e.g. if `DEBUG` or CORS origins need to be overridden for production). Since our backend is already running on Northflank, it likely has these configured, but it's worth verifying.

- **Frontend (Vercel) Env Vars:** Set the `NEXT_PUBLIC_API_URL` on Vercel to point to the backend's public URL [6]. In our case, the backend is at `https://p01--emendas-backend--`

`c7pb86lv22r8.code.run/` – use that (or a custom domain if you have one) as the value for `NEXT_PUBLIC_API_URL` in Vercel's project settings. This ensures the frontend knows where to send API requests in production. Our env template shows this variable is used to differentiate local vs production API endpoints. For example, in development it's `http://localhost:8000` [7], but in production we need to override it with the Northflank URL.

- **CORS Configuration:** The backend is currently allowing all origins (`allow_origins=["*"]` in FastAPI middleware). This means it will accept requests from the Vercel app by default. For now, this is okay to get things working quickly. (In a polished production, we'd restrict `allow_origins` to our Vercel domain and any other client domains. Our env file even suggests setting `CORS_ORIGINS=https://your-frontend.vercel.app` in production [8]. You can update that later; with `*` already in code, it's not blocking us.)

- **Frontend Variables (Optional):** Review any other `NEXT_PUBLIC_*` settings. For instance, things like app name, theme color, etc., are defined in env (see `NEXT_PUBLIC_APP_NAME`, etc. in env.example [9]). These can be set in Vercel as needed, but they are not critical for functionality. The main one is the API URL. Also ensure the frontend build has `NODE_ENV=production` (Vercel sets this automatically).

- **GitHub Secrets (for Automation):** We will set up a GitHub Actions workflow to run the SIOP bot. For that, you'll need to add the AWS credentials (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and possibly `AWS_REGION` and `S3_BUCKET_NAME`) as **repository secrets** in GitHub. The workflow will pull these in to upload to S3. This prevents us from hardcoding sensitive keys in the code or workflow file.

Automating Daily SIOP Data Updates (GitHub Actions)

To keep the data pipeline up-to-date, we will automate the **SIOP download + S3 upload** step using GitHub Actions. The goal is to run the `automation` bot every night, so that new SIOP data is fetched and pushed to S3, and then signal the backend to refresh its cache.

1. Create the Workflow File: In the repository, add a workflow (e.g. `.github/workflows/daily-siop-sync.yml`). Configure it to run on a schedule, for example:

```
name: Daily SIOP Data Sync

on:
  schedule:
    - cron: "0 3 * * *"   # Runs every day at 03:00 UTC (adjust as needed)
```

This uses GitHub's Cron to trigger daily. You can adjust the time. (Ensure the repository has GitHub Actions enabled and the schedule triggers are active.)

2. Workflow Job Setup: Use an Ubuntu runner. The job steps will roughly be:

- **Checkout code:**

```
- uses: actions/checkout@v3
```

- **Set up Python:** (Our automation is Python-based)

```
- uses: actions/setup-python@v4
  with:
    python-version: "3.10"   # or 3.9+, to match our requirements
```

- **Install dependencies:**

```
- name: Install Automation Dependencies
  run: pip install -r automation/requirements.txt
```

This will install Selenium, boto3, etc., as listed in `automation/requirements.txt`.

- **Install Chrome (for Selenium):** Since our bot uses Chrome WebDriver, we need a Chrome browser in the runner. We can add a step to install Google Chrome. For example:

```
- name: Install Chrome
  run: |
    sudo apt-get update
    sudo apt-get install -y wget libappindicator3-1 fonts-liberation
    wget -q -O /tmp/chrome.deb https://dl.google.com/linux/direct/
google-chrome-stable_current_amd64.deb
    sudo dpkg -i /tmp/chrome.deb || sudo apt-get install -yf
```

This downloads and installs Chrome stable on the runner. (There are also GitHub Action actions for this, or one could use Chromium. But the above is a straightforward approach.)

- **Run the SIOP bot:** Finally, execute our automation script:

```
- name: Run SIOP Automation
  env:
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    AWS_REGION: ${{ secrets.AWS_REGION }}
    S3_BUCKET_NAME: ${{ secrets.S3_BUCKET_NAME }}
  run: |
    cd automation
    python main.py --mode enhanced --headless
```

Let's break down this command:

- We `cd automation` to run the script in the automation folder.

- We run `python main.py` with `--mode enhanced` and `--headless`. In **enhanced mode**, our automation script will not only download the SIOP spreadsheet but also **upload it to S3 automatically** via our S3Service [10] [11]. (The code appends the backend `services` path and uses `S3Service.upload_file` for each CSV [12] [11].) The `--headless` flag ensures Chrome runs in headless mode (no UI), which is necessary on a CI runner.
- We pass the AWS credentials and bucket name as environment variables. The automation script will pick these up (it uses `os.getenv` in `S3Manager` to configure AWS [13] [14]). This allows the bot to authenticate to AWS and upload the file.

Monitor this step's output in the Actions logs. It should log steps like launching the browser, downloading the file, then "⬆ Iniciando upload" and " Upload concluído" messages from the script. Any errors here (like a failure to open Chrome or to find elements on the SIOP site) will need addressing, but assuming it's been working manually, it should run on the runner as well.

- **Trigger Backend Refresh:** Once the data is uploaded to S3, we want the backend to update its cache immediately. We have a special API endpoint for this: `/api/trigger-etl`. The backend code indicates this endpoint kicks off a background task to process new S3 data [15]. We'll call it as the final step:

```
- name: Trigger Backend ETL Refresh
  run: curl -X POST "https://p01--emendas-backend--
c7pb86lv22r8.code.run/api/trigger-etl"
```

  This sends a POST request to the backend's trigger URL. The backend doesn't require auth for this route (it's considered an internal trigger and explicitly meant to be called by GitHub Actions after upload [15]), so no token is needed. The response is a JSON message indicating the ETL was started in background. We don't necessarily need to wait for completion; the backend will load the latest CSV from S3, process it, and update its in-memory cache. By the time the next user accesses the frontend, the new data should be available via the API.

Put all these steps in the YAML workflow. With this in place, each night the sequence will be: **GitHub Action -> run bot -> upload S3 -> call backend -> backend loads new data**. This automates the manual process you described (downloading the spreadsheet and uploading it daily).

> **Note:** The **timing** of the cron job can be adjusted to whenever new SIOP data is expected to be available. If SIOP updates daily by a certain time, schedule the job after that. Also, monitor the first few runs of this Action – if Chrome fails to run due to missing dependencies or display, we might need to tweak (e.g., use Xvfb or a different approach). The above installation of Chrome usually suffices for headless operation.

Final Checks and Deployment Steps

To ensure everything works by tomorrow morning, here is a checklist of what to verify and do (in order):

1. **Apply Code Fixes for Frontend Build:** Update `tsconfig.json` and fix all import paths as described (especially in `useAuth.ts`). Remove any temporary comments/hacks. Commit these changes. Verify `npm run build` passes locally.

2. **Push to Trigger Vercel:** Push the commits. On Vercel, watch the deployment logs. We expect a clean build now (no "module not found" errors). If any error remains, address it immediately (the

error message will indicate the file/path). Commonly it would be a file naming issue or a forgotten import.

3. **Configure Vercel Env:** In the Vercel dashboard for the project, set `NEXT_PUBLIC_API_URL` = `https://p01--emendas-backend--...code.run` (or the correct backend URL). Also set any other `NEXT_PUBLIC_*` variables as needed (optional). Redeploy if you added env vars (though Vercel might auto-redeploy after env changes).

4. **Backend (Northflank) Check:** Ensure the backend at Northflank is running with the latest code and proper env vars. Test the backend's health endpoint (e.g., GET `/health` on the Northflank URL) to see if it responds. Since it's already deployed successfully, this is just a sanity check.

5. **Test Frontend ↔ Backend Integration:** Once the frontend is deployed on Vercel, open the app in a browser:

6. Try logging in (if you have user credentials setup) to test the auth flow. The network calls (`/api/login`, etc.) should ultimately hit the backend. If login succeeds or fails as expected, it means frontend can talk to backend.
7. Test a data view: the dashboard should fetch data from the backend's `/api/opportunities` or `/api/summary`. Since we might still be using sample data until a real S3 upload happens, ensure the frontend at least displays something (the sample dataset, if `USE_SAMPLE_DATA=True`, will allow the backend to serve some demo data).

8. If any API calls from the frontend are failing (check the browser console network tab), verify the URL it's hitting. It should be the Northflank URL (as set by `NEXT_PUBLIC_API_URL`). If it's trying to call localhost or something, then the env var isn't set correctly on Vercel.

9. **Run the Automation Manually (if needed):** If time permits, you might run the SIOP automation once manually now to populate the S3 with the latest file. This can be done by running the bot on your machine or triggering the GitHub Action workflow manually (you can dispatch it on the Actions page). Getting one fresh data file in S3 and refreshing the backend will ensure the frontend shows real up-to-date data. Otherwise, the system will update overnight as scheduled.

10. **Monitor the First Scheduled Run:** Since you'll be working overnight, you can manually watch the GitHub Action at the scheduled time (or trigger it manually for a trial run). Check the logs for any errors (especially any Selenium issues or AWS permission problems). If it fails, you can debug and re-run. If it succeeds, you should see the new file in the S3 bucket (check AWS S3 console for the object under `siop-data/YYYY/MM/DD/` path) and the backend's logs should indicate it processed new data (Northflank logs or by hitting the `/health` or `/api/summary` endpoint to see updated timestamps/counts).

By following the above steps, we cover the entire pipeline: **fixing the frontend build**, **configuring all services**, and **automating the data refresh cycle**. This integrated approach aligns with the project's architecture from SIOP to Dashboard:

📊 SIOP → Bot (GitHub Action) → ☁ S3 → ⚙ Backend ETL → 🖥 Frontend (Vercel)

Each component is now set up to succeed: - The **frontend** will build and run on Vercel without module errors. - The **backend** is running on Northflank and ready to serve data (and is callable by the frontend).

- The **automation bot** is wired into a scheduled GitHub Action for daily updates. - AWS S3 is the bridge storing the data file, with both the bot and backend configured to use it.

Finally, once the system is up tomorrow, perform a full end-to-end test in production: verify that new data (if any) from the morning's SIOP run appears on the dashboard. If something is missing, you can manually trigger the refresh or check logs, but the setup above should minimize any manual intervention.

Good luck – by implementing these solutions, you should be on track to have the entire project running smoothly by morning!

Sources:

- Project troubleshooting log – path alias fix and results [3] [2]
- Vercel guide on ModuleNotFound causes (case-sensitivity) [1]
- Environment variable reference (`NEXT_PUBLIC_API_URL`, AWS keys, etc.) [5] [6]
- Backend trigger endpoint for GitHub Actions [15]

---

[1] How do I resolve a 'module not found' error?
https://vercel.com/guides/how-do-i-resolve-a-module-not-found-error

[2] [3] [4] GitHub
https://github.com/v1nometrics/saep_tentativa/blob/788087869ccf2fbee6533386591a5ffb24549c07/.MD's/tentativas_contorno_erro_build_frontend.md

[5] [6] [7] [8] [9] GitHub
https://github.com/v1nometrics/saep_tentativa/blob/788087869ccf2fbee6533386591a5ffb24549c07/env.example

[10] [11] [12] GitHub
https://github.com/v1nometrics/saep_tentativa/blob/788087869ccf2fbee6533386591a5ffb24549c07/automation/main.py

[13] [14] GitHub
https://github.com/v1nometrics/saep_tentativa/blob/788087869ccf2fbee6533386591a5ffb24549c07/automation/siop_core.py

[15] GitHub
https://github.com/v1nometrics/saep_tentativa/blob/788087869ccf2fbee6533386591a5ffb24549c07/backend/main.py