ChatGPT

Issue: Finalizing Last Set of Fixes for Finanzas SD (Prefacturas & Budget Modules)

We have three remaining fix areas to address before **Finanzas Service Delivery** can go live. This issue details each item with context, instructions, and quality requirements. All work should follow our project **guardrails** and include thorough testing with evidence of success.

Important Guardrails & Standards

- **CI/CD Security:** Use OIDC for CI; **no static credentials** in pipelines.
- **Preserve Prod Behavior:** Do not alter existing `/finanzas/*` production paths or break SPA deep-links. The Single Page Application must handle deep links (404/403 fallback to index).
- **Least-Privilege IAM:** Ensure all AWS roles and permissions follow least-privilege principles. No broad or admin access in deployment roles.
- **Coding & Quality:** All code changes must pass linting and **automated tests**. **Tests must pass before merge** to main. Follow coding standards for clarity and maintainability.
- **Documentation:** Update relevant docs (API contract, README, runbooks) if behavior changes. Keep OpenAPI spec in sync with implementation (update `openapi/finanzas.yaml` as needed).

Item 1: Finalize Back-End and API Functionality

Context: The backend API needs final verification and tweaks for all planned endpoints and features. Key handlers include projects, handoff, **catalogo rubros**, allocations, plan reports, payroll ingestion, adjustments, alerts, providers, and the **prefactura** webhook. At this stage, all endpoints should be implemented per the API contract and return expected results.

Tasks:
- **Complete Pending Endpoints:** Ensure any remaining endpoints (e.g. `adjustments.create`, prefactura webhook) are fully implemented and integrated. The prefactura webhook should correctly impact the corresponding “rubro” (budget item) as designed. Verify that business logic (e.g. pro-rated calculations for adjustments) is correct and matches requirements.
- **OpenAPI & Contracts:** Review and update the OpenAPI spec (`finanzas.yaml`) to reflect the final implementation. This spec should include all schemas and example responses. Once updated, re-import or deploy it to API Gateway (if using SAM/CDK, ensure the template reflects the latest spec) so that documentation and integrations are up-to-date.
- **Catalogo Rubros Data:** Integrate the `catalogo_rubros_ikusi.xlsx` data into the system. If this Excel contains reference data (e.g. list of budget categories), transform and load it into the appropriate data store (e.g. a DynamoDB table or seed data script). Ensure the `/catalogo/rubros` API endpoint (or similar) is serving this data correctly in dev and will do so in prod.
- **Testing Endpoints:** For each critical endpoint (projects, allocations, close-month, etc.), perform functional tests: - Use Postman or cURL to send requests (including authenticated calls with a valid JWT) and confirm correct responses (HTTP 200 for success, proper error codes for edge cases). For example, test the health

check and a protected endpoint via the CloudFront URL in dev to ensure routing works [1] [2] . - Verify the prefactura approval flow end-to-end: simulate a prefactura webhook call and check that the related budget entry updates (e.g., a “rubro” value changes) and that any notifications or follow-on processes trigger. - Write unit tests for any new logic introduced (e.g., calculation functions, data transformations). Augment integration tests for end-to-end flows if possible.
- **Evidence:** Document the results of testing each endpoint. Include relevant snippets or files: - Log or screenshot of a successful prefactura webhook call (e.g., HTTP 202 Accepted and the resulting data change). - Sample API responses (JSON) for key endpoints to show correctness. - Updated Postman collection (if maintained) with all endpoints passing.
- **DoD:** All backend tests (unit/integration) should be green and endpoints return expected JSON outputs. Peers should review code changes for these fixes.

Item 2: Implement Security and Infrastructure Guardrails

Context: We must ensure the deployment and infrastructure configurations meet our security and reliability standards. This includes CI/CD pipeline security, AWS infrastructure (CloudFront, API Gateway, Cognito) settings, and monitoring/alerts. The goal is a robust production setup with no security loopholes, aligned with the guardrails above.

Tasks:
- **CI/CD OIDC Setup:** Verify that the GitHub Actions (or CI tool) uses OpenID Connect for AWS authentication (no AWS keys in repo). If not already done, implement OIDC auth and remove any hard-coded secrets. Ensure the IAM role for GitHub has a trust policy for the OIDC provider and only the necessary permissions (e.g. deployment to specific resources).
- **Infrastructure Least Privilege:** Audit AWS IAM roles and policies for the backend (Lambda, DynamoDB, etc.) and CloudFront update workflow. Lock them down to only required actions. For example, the role updating CloudFront should only allow CloudFront-related actions on the specific distribution, not wildcard access.
- **CloudFront API Integration (Prod):** Extend the CloudFront distribution to proxy API traffic in production as was done for dev. Run the `update-cloudfront` workflow for **prod** environment to add the `/finanzas/api/*` behavior pointing to the prod API Gateway stage [3] [4] . This should create/update: - The CloudFront Function for path rewriting (strip `/finanzas/api` prefix) as per design [5] . - A cache behavior for `'/finanzas/api/*'` with origin = prod API Gateway domain and origin path = `/prod` (or appropriate stage) [3] . - Ensure the SPA fallback (for `/finanzas/*` errors) remains in place so deep links still load the app [6] . - **CORS Configuration:** Double-check that CORS is restricted to the CloudFront domain. The API should only accept requests from `https://{our-cloudfront-domain}/finanzas` origin in production [7] . Update the AllowedCorsOrigin parameter in the API deployment (SAM template) if the CloudFront domain changed for prod. Test an OPTIONS preflight from the browser domain to confirm a 200 OK with proper CORS headers [8] .
- **JWT Authentication:** Ensure the Authorization header is being forwarded by CloudFront and JWT validation is enforced. In production, test a protected endpoint by invoking through CloudFront with a valid Cognito JWT to verify that unauthorized requests are blocked and authorized requests succeed [9] [2] .
- **Monitoring & Alarms:** Set up CloudWatch alarms and synthetic canaries: - Create alarms for critical metrics (e.g. API Gateway 5XX error count, Lambda errors, CloudFront 4XX/5XX rates). As per DoD, a simulated 5xx in dev should trigger an alarm – configure similarly for prod. - Configure CloudWatch Synthetics canaries to periodically hit key endpoints (e.g. GET `/finanzas/` and a deep link, GET `/finanzas/api/health`) to monitor uptime. Ensure they run in prod and alert on failures. - Verify access

logs for CloudFront and API Gateway are enabled (for auditability).
- **Evidence:** Provide proof of these guardrails: - Screenshot or CLI output showing CloudFront distribution with both UI and API origins/behaviors configured (post-update). - Excerpt of IAM policy documents or roles to demonstrate least privilege (or a summary of changes made). - Results of a CORS preflight test (e.g., curl output) showing the `Access-Control-Allow-Origin` is correct [8]. - Alarm testing evidence: screenshot of an alarm in CloudWatch going off (from a dev test) or a canary report showing successful runs. - Confirmation that no secrets are present in the repo or CI configs (could be a screenshot of repository secrets settings or a code snippet from the workflow showing OIDC usage).

DoD: All security checks should pass (e.g., no critical findings in a vulnerability scan, all secrets removed). CloudFront should seamlessly serve both the app and API (tested via the same URL). Alarms and logs should be actively monitoring the system.

Item 3: End-to-End Testing and UI/UX Polish

Context: With backend and infrastructure solidified, perform comprehensive end-to-end testing of the **Aprobación Pre-facturas** and **Gestión Presupuesto** modules. This is to catch any remaining bugs and ensure a smooth user experience. Additionally, refine the UI/UX for clarity and wow-factor — the interface should be intuitive and responsive for end users.

Tasks:
- **Functional QA Testing:** Execute full end-to-end scenarios in a staging/dev environment: - **Pre-factura Approval Flow:** Simulate the lifecycle of a pre-invoice: creation or intake (if applicable), approval steps, and the triggering of the prefactura webhook. Verify the UI updates (e.g. a “Pre-factura approved” status or similar) and that corresponding budget adjustments occur behind the scenes. Check edge cases, such as rejecting a pre-factura or receiving malformed data via the webhook, and ensure graceful handling (no crashes, user gets error feedback).
- **Budget Management (Presupuesto):** Walk through creating or updating budgets, allocating funds (rubros), and the end-of-month closing process. Confirm calculations (totals, remaining budget) are correct in the UI and API responses. If the UI allows exporting reports (PDF/CSV as noted in requirements), test those downloads for correctness and formatting.
- **Multi-User/Role Testing:** If different roles (e.g., requester, approver, manager) exist, test the app with users of each role to ensure role-based access and UI element visibility work as expected.
- **UI/UX Improvements:** Audit the frontend for any usability issues: - Fix any layout bugs, alignment issues, or Spanish translations that are missing or incorrect. - Ensure consistency in styling across the app (check that new components match the design system). - Add enhancements for user experience where possible: for example, loading indicators while data fetches, confirmation messages when actions succeed or fail, etc. Aim to delight the user in the final touches. - Verify the app is responsive (if it will be used on different screen sizes) and works on the supported browsers.
- **Performance Optimization:** Evaluate if any part of the UI or API is slow with real data sizes. Optimize API queries or front-end rendering as needed (e.g., use pagination or lazy loading for large lists of budget items). The goal is a snappy UI for a better impression on users.
- **Environment Config for Prod:** Prepare the final production build of the UI. Ensure `VITE_API_BASE_URL` (and any other env vars) are correctly set to use the CloudFront proxy path in production [10]. Build the app for production and verify that it calls the `/finanzas/api` endpoints (no direct dev URLs).
- **Regression Test:** Run the full automated test suite (unit, integration tests in the repo) to ensure nothing else broke. Particularly, rerun any core integration tests and the golden test cases from QA. The app should

have **zero regressions** – all previously working features should still work.
- **User Acceptance & Training:** (If applicable) conduct a final review with stakeholders or product owners. Gather any feedback and fix minor issues. Ensure any user documentation or training material is updated reflecting the current UI (screenshots, instructions).
- **Evidence:** For final sign-off, assemble an **evidence package** for this item: - Screenshots or screen recordings of the UI during key flows (e.g., a successful pre-factura approval, budget update screen showing correct calculations). - A summary of manual test cases run and their outcomes (pass/fail). Include this in the issue or as an attached QA report. - Output of the final build/test pipeline showing 100% tests passed. - Any user feedback or UAT sign-off notes, if available.

DoD: All critical user journeys are verified to work in a production-like environment. The UI is polished and free of obvious issues. Stakeholders accept the functionality. The application is ready for production deployment with high confidence in quality.

Quality Assurance & Evidence for Closure

Each fix above must be delivered with robust testing and proof. Before closing this issue:
- Attach or link a **test evidence report** for each item (can be consolidated). This should include CI pipeline results, Postman test results, screenshots of successful operations, and any relevant CloudWatch logs or metrics.
- The team should **iterate and re-test** until every acceptance criterion is met and no new bugs are introduced. If any test fails or new issue arises, address it and repeat the tests.
- Ensure a final smoke test is done on the live (production) environment once deployed. This includes hitting the health check, logging in, and navigating through a couple of primary screens to confirm everything is truly functional in prod. Only then should we mark the issue as resolved.

By following these instructions and guardrails, we aim to achieve a smooth go-live with a high-quality product that delivers a wow factor to our customers. The focus is on **efficiency with no compromise on quality** – do it right the first time, but verify every step. Let’s push this across the finish line!

---

1 2 3 4 5 6 7 8 9 10 cdn-proxy.md
file://file_000000007a78722fbc164e44b6dbd13e