ChatGPT

Invoice PDF Generator: Assignment Summary

Plan

The goal is to build a **proof-of-concept invoice PDF generator** (frontend in React/Next.js + Tailwind, backend in Node/Express, database in MongoDB) that allows authenticated users to create invoices and download them as PDFs. The system must handle **resource-intensive PDF creation** in the background so the UI stays responsive, ensuring a smooth user experience and a stable server. Core requirements include user signup/login (protecting the dashboard from unauthenticated users), an invoice dashboard with a form to add line items, and a **"Generate PDF"** button to start PDF creation [1] [2]. When the button is clicked, the invoice appears immediately in a list marked **"Processing..."**. The server generates the PDF asynchronously and, once ready, updates the invoice status to **"Ready"** and shows a **download link** that only the owner can use [3] [4].

Deliverables (due within 24 hours) are explicitly specified:
- **Live Demo:** a working deployment (e.g. frontend on Vercel, backend with MongoDB Atlas) showing the full flow [5].
- **Source Code:** a public GitHub repo with all frontend and backend code [6].
- **README.md:** must include an overview, setup/run instructions, and an **"Architectural Decisions"** section answering the evaluation questions [7].
- **Video Walkthrough (≈5 min):** a demo of signing up, creating an invoice, and seeing the automatic status update; plus a code tour highlighting key parts (background job logic, rate limiter, download security) and verbal reasoning for major choices [8].

The **evaluation criteria** inform the plan: top priority is **System Design & Architecture** (we must clearly describe the end-to-end async flow) [2], followed by **Security & Robustness** (show how User A cannot access User B's data and how rate-limiting is enforced) [4]. Code organization (frontend vs backend separation) and smooth user experience (asynchronous flow without UI freezes) are also assessed [9]. These points will be integrated into the README and video explanations as part of the plan.

Workflow and Architecture

The system uses a **modern MERN stack**: a Next.js/React frontend (with Tailwind CSS for styling) talking via REST (or similar) to a Node.js/Express backend, with MongoDB as the database. The **end-to-end flow** works as follows:

- **User Action (Frontend):** A logged-in user fills out an invoice form (client info, line items) on the dashboard and clicks *"Generate PDF."*
- **API Request:** The frontend immediately sends an HTTP request to the backend to create the invoice. The UI does **not** freeze – it shows the new invoice in the list with status *"Processing..."* right away [3].

- **Backend – Enqueue Job:** The backend route inserts a new invoice record in MongoDB with status "pending/processing" and enqueues a PDF-generation job in a background queue (e.g. using BullMQ with Redis). This **background job** approach offloads heavy work from the request cycle [10].
- **Background Worker:** A separate worker process picks up the job, generates the PDF (e.g. using a library like Puppeteer or pdfkit), and saves the file (locally or cloud storage). Once done, it updates the invoice record's status to "ready" and may store a file URL or blob in the database.
- **Real-time Update:** The worker notifies the frontend that the invoice is ready. This can be done via WebSockets (e.g. Socket.IO) or Server-Sent Events: the frontend listens for a "status update" event and automatically changes the UI from *"Processing..."* to *"Ready"* without a page reload [3]. A **Download** link then appears next to that invoice.
- **Secure Download:** When the user clicks **Download**, the frontend calls a secure backend endpoint. The server checks the user's identity (e.g. JWT or session) and ensures the invoice belongs to them before streaming the PDF. This enforces **data tenancy** – User A can never download User B's invoice [11] [4].

This architecture addresses the *three focus areas*:

- **Scalable Task Handling:** By using a background job queue, long-running PDF generation is performed asynchronously in separate worker processes. This prevents one large job from blocking other requests. In practice, this might use a library like Bull/BullMQ or Agenda with Redis, so the Node event loop stays free [10]. This design choice directly satisfies the System Design & Architecture criterion: the flow is clearly asynchronous and scalable [2]. Using separate workers means the main server remains responsive.

- **System Stability & Abuse Prevention:** To prevent abuse (e.g. someone spamming PDF requests), we implement a **rate limiter** on the PDF-generation endpoint (e.g. using express-rate-limit). The assignment suggests "5 PDF generations per user per minute" (which we will enforce) [2]. Rate limiting throttles excessive requests and prevents overloading the service. Industry best practices confirm this: rate limiting improves scalability and reliability by stopping clients from overwhelming your resources [12]. We would store rate-limit counters in a shared store (e.g. Redis) so it scales across multiple server instances. This directly meets the Security & Robustness criterion by handling malicious or buggy use.

- **Data Security & Tenancy:** Each invoice record in MongoDB will include the creator's user ID. All queries (listing invoices, downloading a PDF) will filter by the authenticated user's ID. The download endpoint checks ownership before serving the file. This ensures **isolation**: no user can access another's data. If the app scaled to multiple servers, the MongoDB database itself is already multi-tenant via user IDs. We may also secure the PDF storage (private folders or signed URLs) to prevent direct link sharing. This design solves the Data Security & Tenancy concern from the brief [4]. It also directly addresses the Security & Robustness evaluation question ("User A cannot download User B's invoice").

Each focus area ties into the evaluation criteria: the asynchronous, queued design satisfies *System Design & Architecture* by showing a non-blocking flow [10] [2]. Rate limiting and strict ownership checks cover *Security & Robustness* [12] [4]. Cleanly separating frontend, backend, and worker code (and writing modular, documented code) addresses *Code Quality & Best Practices*. Finally, ensuring the user sees immediate

feedback and smooth updates fulfills *User Experience* – the flow should “work as expected” with no manual page refresh needed [3] [12].

To-Do List

A comprehensive task list to build and submit the project is as follows:

- **Frontend Setup (React/Next.js):** Initialize a Next.js app with Tailwind CSS. Create pages/components for:
- **Authentication:** Signup and login pages (with form validation). After login, redirect to dashboard. Use JWT or NextAuth to manage auth tokens.
- **Dashboard:** A protected page that lists the user’s invoices. Implement a form to input invoice details (client name, line items with description and price).
- **Generate Button:** Add a “Generate PDF” button that calls the backend API to create a new invoice. Upon clicking, optimistically display the invoice in the list with status **Processing...**.
- **Status Updates:** Integrate real-time updates (e.g. with Socket.IO or polling). When the server signals an invoice is ready, update its status to **Ready** in the UI and enable a “Download” link.
- **Backend Setup (Node.js/Express):** Initialize an Express app connected to MongoDB (hosted on Atlas). Implement:
- **User Model & Auth Routes:** Mongo schema for User (with hashed passwords). Routes for signup and login that issue JWTs or sessions. Protect all dashboard/invoice routes to only allow authenticated access.
- **Invoice Model:** Mongo schema for Invoice (fields: userId, client info, line items, status flag, maybe file path).
- **Create Invoice API:** POST route that validates input, creates an invoice with status “processing”, and enqueues a job to generate the PDF. Returns the new invoice ID. Apply an Express rate-limit middleware here (limit e.g. 5 requests/min/user).
- **List Invoices API:** GET route that returns all invoices for the logged-in user.
- **Download Invoice API:** GET route that takes an invoice ID, checks that the requesting user owns it, and streams back the PDF file (with appropriate HTTP headers). Return 403 if the invoice does not belong to the user.
- **Background Job System:**
- **Queue Setup:** Integrate a queue library (e.g. Bull or BullMQ) with Redis. Ensure the queue is connected and workers can run alongside the server or separately.
- **Worker Process:** Write a separate script or service that listens for new invoice jobs. When a job arrives: generate the PDF (e.g. using Puppeteer to render an HTML invoice, or use pdfkit/React PDF), save the file (e.g. to a `/pdfs` directory or cloud storage), then update the MongoDB invoice document: set `status = "ready"` and store the file path/URL.
- **Notify Frontend:** After updating the DB, emit a WebSocket event (if using Socket.IO) or similar message so the frontend knows to refresh that invoice’s status.

- **Real-time Status Updates:** Implement a mechanism for pushing status changes to the client. For example, use **Socket.IO**: upon user login, connect to a socket and join a room for that user. The worker emits an event like `invoice_ready` with the invoice ID when done. The client listens and updates the UI. Ensure fallback (e.g. long-polling) if needed, but WebSockets is ideal for a smooth UX.

- **Security and Tenancy:**

- **Ownership Checks:** In every invoice-related endpoint, filter by the authenticated user ID. In the download route, explicitly verify `invoice.userId === req.user.id` before sending the PDF.
- **File Protection:** Store PDFs in a secure location (not publicly accessible). If using cloud storage (e.g. S3), generate temporary signed URLs. If local, ensure the download route checks access.

- **Rate Limiting:** Use a rate-limiter middleware (e.g. `express-rate-limit`). Configure it to allow only **5 PDF-generation requests per user per minute** (as specified). Use a store like Redis for the limiter so it works across multiple server instances.

- **System Stability:** Consider task queuing details: limit the number of concurrent worker threads for PDF jobs, so one user's very large invoice cannot consume all resources. Optionally use a cluster of workers for scale.

- **Deployment:**

- **Frontend:** Deploy the Next.js app to Vercel (free tier). Configure any environment variables (e.g. API base URL).
- **Backend & Database:** Deploy the Express app on a free Node host (Heroku, Render, etc.) or as a serverless function. Use MongoDB Atlas for the database (free tier). Point the backend to the Atlas URI. Also ensure Redis (for Bull queues and rate-limit store) is available (e.g. RedisToGo or Upstash).

- **CORS and Security:** If frontend/backend are on different domains, set up CORS. Use HTTPS in deployment.

- **Documentation and Demo Prep:**

- **README.md:** Write clear setup instructions (install, environment variables, run commands). Include a **project overview** and note how to run locally. Add an **"Architectural Decisions"** section answering the evaluation questions (e.g. explain the async flow vs sync choice, describe rate-limiter implementation and scaling plan, and how data isolation is enforced) [2] [4].
- **Testing:** Test the full flow end-to-end (signup, invoice creation, status update, download). Verify rate-limiting is enforced (e.g. by firing >5 requests).

- **Walkthrough Video:** Script and record a ~5-minute video: first demo the app (signup, create invoice, see processing → ready, and download PDF). Then show key parts of the code (e.g. the job queue setup, rate-limit config, download route with auth check) and explain the reasoning behind them.

- **Final Review:** Ensure code quality (use linters, clear file structure), update UI (user-friendly messages, loading indicators), and make sure all evaluation points are covered in the README and video.

Throughout, every detail from the assignment brief is addressed. For example, the README will explicitly answer how the end-to-end **Generate→Ready** flow is asynchronous (meeting the System Design criterion) and why that choice is critical [2] [10]. It will also explain the step-by-step security (User A vs B) checks and rate limiting strategy (meeting Security & Robustness) [4] [12]. Each line item from the brief is implemented and documented, ensuring no requirements are missed.

Sources: The above plan and design draw directly on the assignment instructions (provided PDF) for requirements and deliverables [1] [4], as well as industry knowledge on async job queues [10] and API rate limiting [12] to justify architectural choices. These citations demonstrate why background processing and rate limiting are standard best practices that align with the brief's criteria.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [11] Assignment - Invoice PDF Generator.pdf
file://file-6WYbUvxA2tcCAe6f3WvaGW

[10] How to Handle Background Jobs & Queues in Node.js with BullMQ and Redis | NextGen Dev Labs
https://medium.com/nextgen-dev-labs/async-like-a-boss-background-jobs-email-queues-bullmq-in-node-js-579cf6a298a1

[12] Securing APIs: Express rate limit and slow down | MDN Blog
https://developer.mozilla.org/en-US/blog/securing-apis-express-rate-limit-and-slow-down/