Vwaza Multimedia: Full Stack Engineer Challenge

Project: The "Vwaza Release Manager" MVP

Role: Full Stack Engineer

Timeline: 1 Week (Flexible based on start date)

Deliverable: GitHub Repository + Documentation

The Mission

At Vwaza, we don't just store files; we manage complex relationships between Artists, Releases (Albums/EPs), and Tracks, all while handling high-volume media ingestion. We are looking for an engineer who can architect a system, not just write functions.

Your Goal: Build the **Vwaza Release Manager MVP**.

Since this is a one-week project, we expect **depth**, **architectural maturity**, and **robustness**. We want to see how you research solutions for real-world problems like file uploads, concurrency, database integrity, and role-based access control.

Tech Stack Requirements

You must use our core stack to demonstrate familiarity.

Layer	Requirement	Constraint
Language	TypeScript	**Strict Mode** is required. No any.
Backend	Node.js (**Fastify**)	Must use Fastify. Express is not accepted.
Database	PostgreSQL	**Raw SQL is REQUIRED.** ❌ Do **not** use an ORM (Prisma, TypeORM, Drizzle). ✅ You may use a query builder (Knex) or a driver

		(pg), but we want to see you writing and managing your own SQL queries.
Frontend	React	Must use **React Router v7**.

Project Scope: "The Ingestion Pipeline"

You are building a system that allows Artists to upload music releases and Admins to approve them.

1. The Data & Role Model

Implement a relational schema using **Raw SQL** supporting:

- **Users:** Two roles: ARTIST and ADMIN.
- **Releases:** Represents an Album or EP.
  - **Metadata:** Title, Genre, Cover Art URL, Status.
  - **Status Flow:** DRAFT $\rightarrow$ PROCESSING $\rightarrow$ PENDING_REVIEW $\rightarrow$ PUBLISHED (or REJECTED).
- **Tracks:** Belong to a Release.
  - **Metadata:** Title, ISRC, Audio File URL, Duration.

2. The Backend & Media Pipeline (The Core Challenge)

We want to see how you handle state and file management without blocking the main thread.

- **Authentication:** Implement secure JWT or Session-based auth.
- **Cloud Storage Integration:**
  - Implement an endpoint to upload "Audio Files" and "Cover Art".
  - **Requirement:** Do not store files on the local disk permanently. Integrate with a cloud storage provider (AWS S3, Cloudinary, DigitalOcean Spaces, or Vercel Blob).
- **Asynchronous Processing Simulation:**
  - When a Release is submitted, it enters a PROCESSING state.
  - Simulate a background process (using setTimeout or an in-memory mechanism) that mimics "Transcoding" (wait 5-10s) and "Metadata Extraction".
  - Once simulated processing is complete for all tracks, the Release status must automatically update to PENDING_REVIEW.
  - *Note:* You do not need a full message queue (like RabbitMQ) for this challenge, but your code structure should separate the API layer from the processing layer.

3. The Frontend (Two Views)

Build a polished UI with a focus on UX.

A. Artist Dashboard

- **Release Wizard:** A multi-step form to create a Release:
  1. Album Details.
  2. Track Uploads (Handle file selection and upload progress).
  3. Review & Submit.
- **Status Tracker:** Real-time view of releases. Users should see the status flip from PROCESSING $\rightarrow$ PENDING_REVIEW without refreshing the page (Polling is acceptable).

B. Admin Dashboard

- **Review Queue:** List releases waiting for review.
- **Action:** Admin can **Play a track** (streaming the file from the cloud provider) and **Approve/Reject** the release.

Advanced Requirements (The "Senior" Filter)

To distinguish yourself as a Senior candidate, we expect these production-standard features:

1. **Rate Limiting:** Protect your API endpoints from abuse.
2. **Database Performance:** Ensure you have appropriate indexes on foreign keys and status columns.
3. **SQL Proficiency:** Leverage the database for data manipulation. We prefer complex logic (like aggregations, sorting, or efficient filtering) to be handled in SQL rather than application code.
4. **Error Handling:** A global error handling strategy. The app should not crash on bad input.

Testing Strategy

We expect a mix of testing strategies to prove reliability:

- **Unit Tests:** For your core logic (e.g., status transitions, validation).
- **Integration Test:** At least one end-to-end flow test (e.g., A user logs in and creates a release).

Deliverables

Please provide a link to a **public GitHub Repository** containing:

1. **Source Code:** With a clean commit history showing your progression.
2. **README.md:** This is crucial. It must include:
   - **Setup:** Instructions on how to run your project locally (including how to set up environment variables for the cloud provider).
   - **Architecture Decisions:** Why did you choose this folder structure? Why did you choose that specific Auth strategy?
   - **Trade-offs:** What shortcuts did you take due to the 1-week limit?

💡 Evaluation Criteria

We are not just looking for "working code." We are looking for:

1. **System Design:** Separation of concerns.
2. **Code Quality:** Strict TypeScript typing, no any, clean interfaces.
3. **SQL Skills:** Quality and efficiency of your raw SQL queries.
4. **UX/UI:** It doesn't need to be award-winning, but it must be intuitive and not "broken".
5. **Resilience:** What happens if an upload fails halfway through? Does the DB state remain consistent?