Lab Tracking Web Application Development Plan - Refined

This refined plan integrates the detailed requirements for both external customers and internal lab personnel, leveraging your chosen tech stack: React 19, React Router 7, Tailwind CSS for the frontend, and Express.js with TypeScript, PostgreSQL, and Prisma for the backend, authenticated with JWT and deployed on Google Cloud.

I. Core Requirements & Features (Minimum Viable Product - MVP)

The MVP is expanded to include comprehensive customer-facing and lab-internal workflows.

A. User Management & Authentication

1. **Customer Registration & Profile:**
   - **Information Capture:** Company name, ID card/tax ID, primary address, shipping address for receiving samples/results.
   - **File Attachments:** Allow customers to upload relevant documents during registration (e.g., business license, tax documents).
   - **Email Confirmation:** Mandatory email verification for new customer registrations.
   - **Login/Logout:** Secure authentication for customers.
2. **Internal Lab User Management (Admin):**
   - **Roles:** Admin, Lab Technician, Doctor/Approver, Lab Manager.
   - **Secure Authentication:** Login/Logout for internal staff.
   - **User Administration:** Admins can add, edit, delete internal users and assign their roles.
   - **Authorization (RBAC):** Implement granular role-based access control to restrict features and data based on user roles.

B. Customer Portal Features

1. **Document Request List:**
   - **Listing & Search:** Customers can view and search all their submitted test requests.
   - **Key Information:** Display Request Date, Request Number, Company Name, Requester, Document Status.
   - **Document Status Lifecycle:** "Submitted" -> "Acknowledged and Received Sample" -> "Paid" -> "Approved" -> "Rejected".
   - **Actions:** View, Edit (if status allows), Delete (if status allows), Print Request Summary.
2. **Add/Edit Requesting Test:**

    - **Auto-Generated Request No.:** Automatically generate a unique request number based on company and sender.
    - **Sample List Management:** A dynamic list where customers can add/edit/delete individual samples within a request.
        - **Sample Details:** For each sample: Sample ID, Sent Sample Date, Animal Type, Sample Specimen, Panel, Method, Sample Quantity.
    - **Saving Options:**
        - **Save Draft:** Allow customers to save incomplete requests to continue later.
        - **Submit for Approval:** Submit the request to the lab for review and processing.
3. **Invoice Page:**
    - **Detailed Invoice Display:** Show Lab's Company Info, Customer's Tax Company Info, Invoice Number.
    - **Itemized Details:** Running Item ID, Detail (description of service/test), Quantity, Unit Price, Total for item.
    - **Summary:** Calculate Sub-total, Tax (7%), and Net Total.
    - **Actions:** Print Invoice, Attach Payment Slip (for customer to upload proof of payment).

C. Lab Internal Operations (Admin / Lab Technician / Doctor)

1. **Admin Dashboard & Request Overview:**
    - **Search & Filtering:** Powerful search capabilities by Request Number, Document Status, Objective, Requester, Company, Request Date Range.
    - **Request Listing:** Display all customer requests with: Request Date, Request No., Company, Objective, Requester, Document Status.
    - **Document Status (Internal View):** "Waiting Approval Lab" -> "Submitted" -> "Acknowledged" -> "Lab Result Entry" -> "Waiting Doctor Approval" -> "Approved" -> "Paid" -> "Rejected".
    - **Actions:** Acknowledge (Receive Sample), View Lab Result, Approve/Reject (for Admin/Doctor), Mark as Paid.
2. **Receive Sample & Acknowledge Request (Lab Technician):**
    - **Request Details:** Display the full request information from the customer.
    - **Quantity Adjustment:** Ability to adjust the received quantity of each sample if it differs from the requested quantity.
    - **Acknowledgement:** Confirm sample reception, update status, and move to "Lab Result Entry" step.
3. **Lab Result Entry (Lab Technician):**
    - **Request & Case Info:** Show Request No., Case No., Case Date, Company,

    Sender, Admin (who received sample) Name.
  - **Result Input:** Fields to enter test results for each sample/panel.
  - **Attachment:** Ability to attach final lab result documents (e.g., PDF reports, raw data files).
  - **Result Status:** Set status (e.g., "Pending Review," "Completed").
  - **Request Approval Button:** Submit the entered results for Doctor's approval.

D. Doctor Approval Workflow (Doctor / Approver)

1. **Document Approval Page:**
   - **List Pending Documents:** Doctors see a list of lab result documents awaiting their approval.
   - **Review:** Ability to review all associated lab results and attachments.
   - **Actions:**
     - **Approve:** Mark the lab document as "Approved," triggering notification to customer and allowing invoice generation/release.
     - **Reject:** Mark the lab document as "Rejected," providing a reason, and potentially sending it back for re-testing/re-entry.

II. Full-Stack Architecture Suggestion

Your tech stack choices are excellent and well-suited for these requirements.

A. Frontend (React with React Router & Tailwind CSS)

- **Framework:** React 19+
- **Routing & Data Layer:** React Router 7+ will manage navigation and data flow effectively.
  - **Loader & Action Functions:** Crucial for server-side rendering and efficient data fetching/mutations within routes, reducing boilerplate in components.
  - **Protected Routes:** Implement loader functions or custom route components to guard routes based on user authentication status and role (e.g., /customer/* for customers, /admin/* for lab staff, /doctor/* for doctors).
- **UI Components & Styling:**
  - **Tailwind CSS:** For all styling.
  - **Shadcn UI / Radix UI:** Strongly recommend using Shadcn UI for pre-built, accessible, and customizable UI components like tables, forms, modals, date pickers, and file upload areas. This will save significant development time and ensure consistency.
  - **Icons:** Use Font Awesome (you already have it) or Lucide React for consistent iconography.
- **State Management:**

  - **React Query (TanStack Query):** Essential for managing all data fetching, caching, and synchronization with your backend API. It simplifies complex data flows (e.g., polling for status updates, optimistic UI).
  - **Context API / Zustand:** For global, non-data-related state (e.g., user context, theme settings, notification messages).
- **Form Management & Validation:**
  - **React Hook Form:** For efficient form handling, especially with complex forms like Add Requesting Test and Lab Result Entry.
  - **Zod:** For schema-based validation. You can define validation schemas once and use them on both the frontend and backend, ensuring consistency.
- **API Interaction:** Use fetch API or axios for making HTTP requests to your Express backend, integrated with React Query.

B. Backend (Express.js with TypeScript, PostgreSQL, Prisma)

- **Framework:** Express.js (TypeScript)
- **Database:** PostgreSQL (with JSONB support for flexible fields like details in audit_trail or value in results).
- **ORM: Prisma** is an excellent choice. It provides type-safe queries, powerful migrations, and a clean way to interact with your PostgreSQL database.
  - You'll define your schema in schema.prisma and use prisma generate to create type-safe client.
  - Prisma Migrate will handle database schema changes.
- **Authentication:**
  - **JWT (JSON Web Tokens):** For stateless authentication. After successful login, issue a JWT containing userId and role. Clients store this token (preferably in an HttpOnly cookie for browser-based apps).
  - **Password Hashing:** Use bcrypt for securely hashing passwords before storing them.
- **Authorization (RBAC):**
  - Implement Express middleware that verifies the JWT and extracts the user's role.
  - Create granular middleware functions (e.g., isAdmin, isLabTech, isDoctor, isCustomerOrAdmin) to protect specific routes based on the required role.
- **API Design:** Build a clear and consistent **RESTful API**.
  - **Versioning:** Use /api/v1/... for your endpoints.
  - **Resources:** Define clear endpoints for customers (/api/v1/customers, /api/v1/customer-requests, /api/v1/invoices), and lab internal operations (/api/v1/lab/requests, /api/v1/lab/samples, /api/v1/lab/results, /api/v1/lab/users).
- **Input Validation:** Use **Zod** (or Joi) on the backend to validate all incoming

request bodies and query parameters. This prevents invalid or malicious data from reaching your database.

- **Error Handling:** Implement a centralized error handling middleware in Express to catch all errors and send consistent, user-friendly error responses (e.g., 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error).
- **Environment Variables:** Use dotenv for all configurations.
- **File Uploads:** Use multer middleware for handling multipart/form-data uploads (customer registration attachments, lab result attachments, payment slips). For secure and scalable storage, **integrate with Google Cloud Storage** (or equivalent object storage) to store files and save the file URLs/paths in your database.

C. Database Schema (Conceptual - PostgreSQL with Prisma-like structure)

This schema reflects the relationships and new entities required.

```
-- `User` table (for both internal lab staff and customer accounts)
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(), -- Use UUIDs for IDs
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  role VARCHAR(50) NOT NULL, -- 'customer', 'admin', 'lab_technician', 'doctor',
'lab_manager'
  is_email_confirmed BOOLEAN DEFAULT FALSE,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `Customer` table (linked to `User` for customer-specific details)
CREATE TABLE customers (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID UNIQUE NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  company_name VARCHAR(255) NOT NULL,
  tax_id_or_id_card VARCHAR(100) UNIQUE,
  address_line1 VARCHAR(255) NOT NULL,
  address_line2 VARCHAR(255),
  city VARCHAR(100) NOT NULL,
  state VARCHAR(100),
  zip_code VARCHAR(20),
  country VARCHAR(100) NOT NULL,
  shipping_address_line1 VARCHAR(255),
  shipping_address_line2 VARCHAR(255),
  shipping_city VARCHAR(100),
  shipping_state VARCHAR(100),
  shipping_zip_code VARCHAR(20),
  shipping_country VARCHAR(100),
  registration_attachments JSONB, -- Store array of file paths/URLs for attached
documents
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `Project` table (Optional, but good for organizing test requests)
CREATE TABLE projects (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(255) UNIQUE NOT NULL,
  description TEXT,
  created_by_id UUID REFERENCES users(id), -- Creator could be internal staff or
customer
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `TestRequest` table (The main customer request for tests)
CREATE TABLE test_requests (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  request_no VARCHAR(255) UNIQUE NOT NULL, -- Auto-generated: e.g.,
"COMP-SENDER-YYMMDD-001"
  customer_id UUID NOT NULL REFERENCES customers(id) ON DELETE CASCADE,
  requester_name VARCHAR(255) NOT NULL, -- Name of contact person at customer
company
  objective TEXT,
  request_date DATE NOT NULL DEFAULT CURRENT_DATE,
  document_status VARCHAR(50) NOT NULL DEFAULT 'submitted', -- 'submitted',
'acknowledged_sample_received', 'paid', 'approved', 'rejected', 'draft'
  lab_internal_status VARCHAR(50) NOT NULL DEFAULT 'waiting_approval_lab', --
'waiting_approval_lab', 'lab_result_entry', 'waiting_doctor_approval', 'completed'
  project_id UUID REFERENCES projects(id),
  notes TEXT,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `TestRequestSample` (Samples within a specific test request)
CREATE TABLE test_request_samples (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  test_request_id UUID NOT NULL REFERENCES test_requests(id) ON DELETE
CASCADE,
  customer_sample_id VARCHAR(255) NOT NULL, -- ID provided by customer
  sent_sample_date DATE,
  animal_type VARCHAR(100),
  sample_specimen VARCHAR(100), -- e.g., 'blood', 'tissue', 'urine'
  panel VARCHAR(255), -- Test panel requested
  method VARCHAR(255), -- Test method requested
  requested_qty DECIMAL(10, 3) NOT NULL,
  received_qty DECIMAL(10, 3), -- Quantity actually received by lab
  unit VARCHAR(50), -- e.g., 'ml', 'g', 'cells'
  current_status VARCHAR(50) NOT NULL DEFAULT 'received', -- Matches
test_request status for individual sample tracking
  storage_location_id UUID REFERENCES storage_locations(id), -- Direct link to
storage
  notes TEXT,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  UNIQUE(test_request_id, customer_sample_id) -- Ensures unique samples within a
request
);

-- `StorageLocation` table (Hierarchical structure for lab storage)
CREATE TABLE storage_locations (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(255) NOT NULL, -- E.g., "Freezer -80C 1", "Rack A", "Box 1"
  type VARCHAR(50) NOT NULL, -- E.g., 'building', 'room', 'freezer', 'rack', 'box', 'well'
  parent_id UUID REFERENCES storage_locations(id), -- For hierarchical relationships
  capacity INTEGER, -- Max number of items/sub-locations
  current_occupancy INTEGER DEFAULT 0, -- Track current usage
  description TEXT,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `LabTest` (Represents an actual lab test performed, linked to TestRequestSample)
CREATE TABLE lab_tests (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  test_request_sample_id UUID NOT NULL REFERENCES test_request_samples(id) ON
DELETE CASCADE,
  case_no VARCHAR(255) UNIQUE, -- Lab internal case number
  case_date DATE,
  assigned_lab_technician_id UUID REFERENCES users(id),
  test_panel VARCHAR(255), -- Actual panel performed
  test_method VARCHAR(255), -- Actual method used
  lab_result_status VARCHAR(50) NOT NULL DEFAULT 'pending', -- 'pending',
'completed', 'approved', 'rejected'
  notes TEXT,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `LabResult` (Detailed results for a specific `LabTest`)
CREATE TABLE lab_results (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  lab_test_id UUID NOT NULL REFERENCES lab_tests(id) ON DELETE CASCADE,
  parameter VARCHAR(255),
  value TEXT, -- Can be JSONB for complex or structured results
  unit VARCHAR(50),
  reference_range TEXT, -- e.g., "Normal: 10-20"
  is_abnormal BOOLEAN DEFAULT FALSE,
  notes TEXT,
  recorded_by_id UUID REFERENCES users(id), -- Lab technician who recorded
  recorded_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `Invoice` table
CREATE TABLE invoices (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  invoice_no VARCHAR(255) UNIQUE NOT NULL,
  test_request_id UUID NOT NULL REFERENCES test_requests(id) ON DELETE
CASCADE,
  customer_id UUID NOT NULL REFERENCES customers(id) ON DELETE CASCADE,
  invoice_date DATE NOT NULL DEFAULT CURRENT_DATE,
  due_date DATE,
  lab_tax_info JSONB, -- JSON for lab's tax details (name, address, tax_id)
  sub_total DECIMAL(12, 2) NOT NULL,
  tax_rate DECIMAL(5, 2) NOT NULL DEFAULT 0.07, -- 7% tax
  tax_amount DECIMAL(12, 2) NOT NULL,
  net_total DECIMAL(12, 2) NOT NULL,
  payment_status VARCHAR(50) NOT NULL DEFAULT 'pending', -- 'pending', 'paid',
'overdue', 'refunded'
  payment_slip_attachment_url TEXT, -- URL to uploaded payment slip
  issued_by_id UUID REFERENCES users(id), -- Admin who issued
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `InvoiceLineItem` (Details for each item on an invoice)
CREATE TABLE invoice_line_items (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  invoice_id UUID NOT NULL REFERENCES invoices(id) ON DELETE CASCADE,
  description TEXT NOT NULL,
  quantity INTEGER NOT NULL,
  unit_price DECIMAL(12, 2) NOT NULL,
  line_total DECIMAL(12, 2) NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `DocumentAttachment` (Generic table for all file attachments)
CREATE TABLE document_attachments (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  file_name VARCHAR(255) NOT NULL,
  file_url TEXT NOT NULL, -- URL to file in Google Cloud Storage
  mime_type VARCHAR(100),
  entity_type VARCHAR(100) NOT NULL, -- 'customer_registration', 'test_request',
'lab_result', 'payment_slip'
  entity_id UUID NOT NULL, -- ID of the associated entity (e.g., customer.id,
test_request.id, lab_test.id, invoice.id)
  uploaded_by_id UUID REFERENCES users(id),
  uploaded_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- `AuditTrail` (Comprehensive activity log)
CREATE TABLE audit_trail (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES users(id),
  action VARCHAR(255) NOT NULL, -- e.g., 'customer_registered',
'request_submitted', 'sample_received', 'result_approved'
  entity_type VARCHAR(100) NOT NULL, -- 'user', 'customer', 'test_request',
'test_request_sample', 'lab_test', 'invoice'
  entity_id UUID, -- ID of the affected entity
  details JSONB, -- Store old/new values, specific changes, IP address, etc.
  timestamp TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

III. Development Workflow & Best Practices

1. **Project Structure:**
    - **Monorepo (pnpm workspaces):** This is highly recommended. It allows for a single node_modules structure, easy sharing of types (shared/types.ts) between frontend and backend, and simplified tooling.
    - Example structure:
```
your-repo/
├── apps/
│   ├── frontend/      # React app
│   └── backend/       # Express app
├── packages/
│   └── shared/        # Shared types, utility functions
└── pnpm-workspace.yaml
```

2. **Shared Types:** Crucially, define common TypeScript interfaces for all data models (e.g., Customer, TestRequest, Sample, Invoice) in a packages/shared/types.ts file within your monorepo. This ensures type consistency between your frontend API calls and backend responses/database interactions.

3. **API Contract:** Prioritize defining your API endpoints, request/response bodies, and error formats (e.g., using OpenAPI/Swagger or just clear TypeScript interfaces).
4. **Version Control:** Git, with a feature-branching workflow and pull requests for code reviews.
5. **Environment Variables:** Use .env files for secrets and environment-specific configs. **Never commit them to Git.**
6. **Testing Strategy:**
    - **Frontend:** Unit/Component tests with Jest/Vitest and React Testing Library. E2E tests with Cypress/Playwright for full user flows (customer registration, submitting a request, admin review, doctor approval).
    - **Backend:** Unit tests for services and utilities. Integration tests with Supertest for API endpoints, including authentication and authorization checks. Database integration tests using a test database.
7. **Linting & Formatting:** Ensure ESlint and Prettier are configured consistently across both frontend and backend to maintain code quality.
8. **Logging:** Implement structured logging on the backend (e.g., Winston or Pino) for debugging, monitoring, and auditing. Log relevant actions, errors, and system events.
9. **Deployment (Google Cloud):**
    - **Backend:**
        - **Compute Engine / Cloud Run / Kubernetes Engine:** For hosting your Express application. Cloud Run is excellent for serverless containers, scaling automatically.
        - **Cloud SQL (PostgreSQL):** A managed PostgreSQL service.
        - **Cloud Storage:** For storing file attachments (customer documents, lab results, payment slips).
        - **Cloud Build:** For CI/CD to automate building Docker images and deploying to chosen services.
    - **Frontend:**
        - **Cloud Storage (for static assets):** Serve your built React application's static files from a Cloud Storage bucket, exposed via a Load Balancer or Cloud CDN.
        - **Firebase Hosting:** Another simple option for hosting static React apps.
    - **IAM:** Configure fine-grained Identity and Access Management (IAM) roles for service accounts and users.
10. **Security Measures:**
    - **Input Validation:** Essential on both frontend and backend.
    - **Authentication & Authorization:** Robust JWT implementation, HttpOnly

    cookies for tokens, and thorough RBAC middleware.
    - **Password Hashing:** Always use bcrypt for password storage.
    - **CORS:** Properly configure the cors middleware in Express, limiting origin to your frontend domain.
    - **Rate Limiting:** Protect API endpoints (especially login and registration) from abuse.
    - **SQL Injection / XSS Protection:** Use Prisma (ORM) to prevent SQL injection. Sanitize and escape all user-generated content rendered on the frontend to prevent XSS.
    - **Dependency Management:** Regularly scan for and update vulnerable dependencies.

IV. Frontend Specific Suggestions (React Router)

- **Route Structure:**

```
app/
├── src/
│   ├── App.tsx        # Main router configuration
│   ├── routes/
│   │   ├── auth/      # Login, Register, Forgot Password
│   │   │   ├── login.tsx
│   │   │   └── register-customer.tsx
│   │   ├── customer/   # Customer-specific routes
│   │   │   ├── dashboard.tsx
│   │   │   ├── requests/
│   │   │   │   ├── index.tsx     # Document Request List
│   │   │   │   ├── add.tsx       # Add Requesting Test
│   │   │   │   └── $requestId.tsx # View/Edit Request
│   │   │   └── invoices/
│   │   │   │   ├── index.tsx     # Invoice List
│   │   │   │   └── $invoiceId.tsx # View Invoice
│   │   ├── lab/       # Lab internal routes
│   │   │   ├── dashboard.tsx
│   │   │   ├── requests/        # Admin Dashboard, List Requests
│   │   │   │   ├── index.tsx
│   │   │   │   └── $requestId/   # Request Details, Receive Sample
│   │   │   │      ├── index.tsx
│   │   │   │      └── receive-sample.tsx
│   │   │   ├── lab-results/      # Lab Result Entry
│   │   │   │   ├── $labTestId.tsx
│   │   │   │   └── entry.tsx
│   │   │   ├── users/           # Admin User Management
│   │   │   │   ├── index.tsx
│   │   │   │   └── $userId.tsx
│   │   │   └── storage/          # Storage Management
│   │   ├── doctor/      # Doctor-specific routes
│   │   │   └── approval.tsx       # Document Approval Page
│   │   ├── layouts/     # Main Layout, Auth Layout, Customer Layout, Lab Layout
│   │   ├── components/  # Reusable UI components (tables, forms, buttons,
modals, alerts)
│   │   ├── hooks/       # Custom React hooks (e.g., `useAuth`, `usePermissions`)
│   │   ├── api/        # Functions for interacting with backend (using React Query)
│   │   ├── styles/      # Tailwind config, global CSS
│   │   └── types/       # Frontend-specific and shared types
```

- **Dynamic Forms:** Utilize React Hook Form's array fields (e.g., useFieldArray) for dynamically adding/removing samples in the Add Requesting Test page.
- **File Upload Components:** Implement dedicated components for file uploads, showing progress and handling errors.
- **Notifications:** Use a toast notification system (e.g., from Shadcn UI or React Hot Toast) to provide user feedback for successful actions or errors.

V. Backend Specific Suggestions (Express & TypeScript)

- **Modular Structure:** Continue with the suggested modular structure to promote clean code and separation of concerns.

```
backend/
├── src/
│   ├── app.ts
│   ├── server.ts
│   ├── config/          # Database config, environment loading, JWT config
│   ├── routes/          # index.ts, auth.ts, customers.ts, testRequest.ts, lab.ts,
doctor.ts, etc.
│   ├── controllers/      # AuthController.ts, CustomerController.ts,
LabController.ts, DoctorController.ts
│   ├── services/        # UserService.ts, CustomerService.ts,
TestRequestService.ts, LabService.ts, InvoiceService.ts, FileService.ts
│   ├── middleware/       # authMiddleware.ts (JWT verification),
roleMiddleware.ts (RBAC), validateMiddleware.ts (Zod validation)
│   ├── utils/           # jwt.ts, password.ts (bcrypt), requestNoGenerator.ts,
errorHandler.ts
│   ├── models/         # (Prisma schema defined in prisma/schema.prisma)
│   ├── types/          # Shared types/interfaces
│   └── common/         # Common constants, enums
├── prisma/             # schema.prisma, migrations
├── package.json
├── tsconfig.json
├── .env
├── .gitignore
```

- **Auto-Generating Request Numbers:** Implement a service/utility function that generates unique request numbers based on company ID/name and date, potentially with a daily sequence reset. Ensure atomicity for this operation (e.g., using database transactions or sequences).
- **Complex Queries:** Leverage Prisma's powerful querying capabilities for complex searches (e.g., by multiple fields for the admin dashboard).
- **Transactional Operations:** Use database transactions for multi-step operations (e.g., receiving samples where multiple database updates occur) to ensure data consistency.
- **Background Tasks (Optional):** For email sending (registration confirmation, lab result notification), consider using a dedicated email service or a simple background job queue if the volume is high.

VI. High-Level Development Timeline (Iterative Approach)

This revised timeline breaks down the development into more focused modules, keeping your specific needs in mind.

- **Phase 1: Core Setup & Authentication (3-4 weeks)**
  - Set up monorepo (pnpm workspaces).
  - Basic Express server with TypeScript.
  - PostgreSQL & Prisma setup, initial users and customers schema.
  - JWT authentication implementation (login, registration, password hashing).
  - Email confirmation for customer registration.
  - Frontend: Basic login/registration forms, protected routes based on role.
  - Role-Based Access Control (RBAC) middleware for admin role.
- **Phase 2: Customer Request & Sample Management (4-5 weeks)**
  - Database schema for test_requests, test_request_samples, projects, storage_locations.
  - Backend API for CRUD operations on customers, test_requests,

test_request_samples.
    - Implement auto-generation of request numbers.
    - Frontend:
        - Customer "Document Request List" page (list, search, view).
        - "Add Requesting Test" page with dynamic sample list, Save Draft/Submit functionality.
        - Customer Profile management (address, attachments).
    - File upload integration with Google Cloud Storage for customer registration attachments.
- **Phase 3: Lab Receiving & Result Entry (4-5 weeks)**
    - Database schema for lab_tests, lab_results, document_attachments.
    - Backend API for:
        - Admin dashboard search and listing of all requests.
        - "Receive Sample & Acknowledge" functionality (update quantities, change status).
        - "Lab Result Entry" for technicians (add lab tests, results, attach final reports).
    - Frontend:
        - Admin Dashboard page.
        - "Receive Sample" page for lab technicians.
        - "Lab Result Entry" page for technicians, including result input fields and file attachment.
    - Implement audit_trail logging for key actions.
- **Phase 4: Doctor Approval & Invoicing (3-4 weeks)**
    - Backend API for Doctor Approval (approve/reject lab results).
    - Database schema for invoices, invoice_line_items.
    - Backend API for Invoice generation, listing, updating payment status, and payment slip attachment.
    - Frontend:
        - "Doctor Document Approval" page.
        - "Invoice List" and "View Invoice" pages for customers.
        - "Attach Payment Slip" functionality for customers.
        - Internal lab UI for marking invoices as paid.
- **Phase 5: Refinements, Reporting & Deployment (Ongoing)**
    - Comprehensive UI/UX refinements across all modules.
    - Implement advanced search/filtering/sorting on all lists.
    - Develop detailed dashboard views and basic reporting features.
    - Automate notifications (email for status changes, result availability).
    - Thorough testing (unit, integration, E2E).

    - Set up Dockerization for both frontend and backend.
    - Configure CI/CD pipelines (Google Cloud Build).
    - Security hardening and performance optimization.
    - Documentation.

This plan provides a solid framework for your Lab Tracking Web Application. By breaking it down into manageable phases, you can tackle the complexity incrementally and ensure a robust and user-friendly system.