# Full Stack Developer Assessment Documentation

Project: CSV Import & Management Tool using MERN + OpenSaaS  
Frameworks: Next.js, Node.js, PostgreSQL, Prisma, OpenSaaS  
AI Integration: Cursor AI (Chat history embedded below)

## Technical Approach

### Key Architectural Decisions:

- Tech Stack: Chose the MERN stack with Next.js (App Router) for frontend flexibility and PostgreSQL + Prisma for relational data modeling.  
- Framework: Integrated the OpenSaaS SDK for authentication and multi-tenancy, which simplified boilerplate code and enhanced modularity.  
- Data Model: Two main models:  
 - CsvFile: Stores file-level metadata like name, upload date, and owner.  
 - CsvRow: Stores row-level CSV content as JSON, indexed by file.  
- Directory Structure: Followed modular structure with isolated API routes (/api/csv/...), components (CsvManager.tsx), and services (prisma, lib).

### Challenges Faced & Solutions:

- Challenge 1: Environment setup and auth config.  
 - Solution: Used .env with Cursor AI guidance. Integrated OpenSaaS and validated sign-up/login flow early.  
- Challenge 2: Handling dynamic CSV data.  
 - Solution: Parsed using csv-parse/sync, mapped headers/rows dynamically, stored in Prisma with JSON fields.  
- Challenge 3: Efficient UI rendering for large CSVs.  
 - Solution: Used useEffect and pagination logic to load subsets of rows. Implemented sorting and editable cells using conditional rendering.  
- Challenge 4: Restricting data per user.  
 - Solution: Scoped all Prisma queries using userId from session context.  
- Challenge 5: Displaying dynamic, editable tables.  
 - Solution: Derived headers from row data and rendered conditionally editable cells using state logic in React.

### Libraries/Tools Chosen:

- Next.js: For its App Router and seamless SSR.  
- Prisma: Schema modeling and migration support with PostgreSQL.  
- csv-parse: Reliable CSV parsing library for server-side processing.  
- shadcn/ui: Component library used for clean, responsive UI.  
- OpenSaaS: Provided auth and workspace management out of the box.  
- Cursor AI: Used heavily for debugging, schema modeling, code generation, and architectural validation.

# Simulated Cursor AI Chat Log

You:

I'm setting up the OpenSaaS project using Next.js, Node.js, and PostgreSQL. Can you help me configure `.env` and ensure auth is working?

AI:

Sure! Here's a sample `.env` file based on the OpenSaaS template:  
  
DATABASE\_URL="postgresql://postgres:admin123@localhost:5432/csvmanager"  
AUTH\_SECRET="generate-a-secure-secret-key"  
BASE\_URL="http://localhost:3000"  
NODE\_ENV="development"

AI:

Ensure PostgreSQL is running locally. Then run:  
  
npx prisma generate  
npx prisma migrate dev  
npm run dev

You:

Done — I see the login/signup UI and was able to sign up. Auth flow is working ✅

You:

I need a Prisma schema for storing uploaded CSV files and their row data. Here’s what the assignment expects:  
- CsvFile: id, userId, fileName, originalName, uploadedAt, columnHeaders, rowCount  
- CsvRow: id, csvFileId, rowData (JSON), rowIndex

AI:

Here’s the schema:  
  
model CsvFile {  
 id String @id @default(uuid())  
 userId String  
 user User @relation(fields: [userId], references: [id])  
 fileName String  
 originalName String  
 uploadedAt DateTime @default(now())  
 rowCount Int  
 rows CsvRow[]  
}  
  
model CsvRow {  
 id String @id @default(uuid())  
 csvFileId String  
 csvFile CsvFile @relation(fields: [csvFileId], references: [id])  
 rowData Json  
 rowIndex Int  
}

You:

Great — implemented and ran migrations successfully. Now working on the upload API route.

You:

Can you help me write an API route to parse a CSV file, store metadata in `CsvFile`, and rows in `CsvRow`?

AI:

Sure! Use `csv-parse/sync` to parse the uploaded file, then save it like this:  
  
```ts  
// /app/api/csv/upload/route.ts  
import { parse } from 'csv-parse/sync';  
import { prisma } from '@/lib/prisma';  
  
export async function POST(req: NextRequest) {  
 const formData = await req.formData();  
 const file = formData.get('file') as File;  
 const bytes = await file.arrayBuffer();  
 const text = new TextDecoder().decode(bytes);  
  
 const records = parse(text, { columns: true, skip\_empty\_lines: true });  
  
 const csvFile = await prisma.csvFile.create({  
 data: {  
 userId: user.id,  
 fileName: file.name,  
 originalName: file.name,  
 rowCount: records.length,  
 rows: {  
 create: records.map((row, index) => ({  
 rowData: row,  
 rowIndex: index,  
 })),  
 },  
 },  
 });  
  
 return NextResponse.json({ id: csvFile.id });  
}

You:

That worked. I tested with a sample CSV and confirmed that the data is stored correctly.

You:

I need help displaying a list of uploaded CSV files in the UI. Each file should show name, upload date, and row count.

AI:

You can fetch the list via an API like `/api/csv/list`, and render it in your `CsvManager.tsx` page:  
  
```ts  
useEffect(() => {  
 fetch('/api/csv/list')  
 .then((res) => res.json())  
 .then(setCsvFiles);  
}, []);  
```

AI:

Use `Card`, `Badge`, or `Table` components from shadcn/ui for a clean UI.

You:

Done! The list loads perfectly. Shows file names, uploaded date, and row count. ✅

You:

I'm building the Notion-like table. Need help rendering dynamic columns from `rowData` and making them sortable/editable.

AI:

For dynamic headers:  
  
```ts  
const headers = Object.keys(rows[0]?.rowData || {});  
```

AI:

To show editable cells, you can render an input when a cell is active:  
  
```tsx  
<td onClick={() => setEditing({ row: i, col: key })}>  
 {editing?.row === i && editing?.col === key ? (  
 <input value={value} onChange={...} />  
 ) : (  
 value  
 )}  
</td>  
```

AI:

For pagination:  
  
```ts  
const PAGE\_SIZE = 10;  
const paginatedRows = rows.slice(page \* PAGE\_SIZE, (page + 1) \* PAGE\_SIZE);  
```

You:

I’ve implemented:  
- Dynamic table headers ✅  
- Basic pagination ✅  
- Sorting by column header click ✅  
- Cell editing (in progress) 🟡  
- Filtering and column reordering (not started) ❌

You:

How do I restrict CSV access per user?

AI:

In all API routes, filter by `userId`:  
  
```ts  
const files = await prisma.csvFile.findMany({  
 where: { userId: session.user.id },  
});  
```

You:

Done. All files now scoped to the logged-in user. Also added loading spinners and empty state messages.

You:

What should I include in the README?

AI:

A simple `README.md` should include:  
- Setup instructions (`npm install`, `.env`)  
- DB setup (`npx prisma migrate dev`)  
- Demo video (3-5 min screen capture)  
- Summary of features completed  
- Note about Cursor AI issues (optional)