AI-04009 Inventree Database Design

Owner	Ⓡ Rigel Arcayan
Tags	Project
Created time	July 9, 2026 4:32 AM

I’m trying to understand warehousing in a smaller scale for homelab use. The project is to learn inventory organization using the open source Inventree Db, understanding Postgres design and implementation, disaster recovery etc. Explain what is a part, stock, bom, supplier. I’m tying to understand the Inventree database so that I can build my own personal inventory database for home lab use with my real inventory and which are related so I can premade a csv table for import and on the side understand modern database. Are the parts, stock, bom and supplier dataset the basic pillar of my warehousing needs. Create a realistic sample csv dataset for the Inventree db of parts, stock, bom, location and supplier

Building an inventory system is a great way to map real-world relational database design to physical objects. In systems like InvenTree, everything hinges on a clean separation between **definitions** (the idea of an item) and **instances** (the physical items on your shelf).

Yes, **Parts, Stock, BOM, and Suppliers** (alongside **Locations**) are absolutely the core pillars of any warehouse or inventory database. Here is how they break down conceptually and relationally.

The Core Pillars Explained

- **Part (The Definition):** A Part is an entry in your catalog. It defines *what* an item is, its specifications, manufacturer part number (MPN), and category, but it does **not** represent physical inventory. For example, "Cat6a RJ45 Keystone Jack" is a part. You can have a part entry even if you currently own zero units.

- **Stock (The Physical Reality):** Stock represents the *actual physical instances* of a Part sitting in a specific location. While the Part table says *what* it is, the Stock table tracks *how many* you have, their serial numbers, batch codes, and exactly where they are sitting right now (e.g., "Box A, Shelf 2").
- **BOM (Bill of Materials - The Recipe):** A BOM defines hierarchical relationships between parts. It specifies which child parts (components) are required to build a parent part (an assembly). For example, a "Homelab Cluster Node Assembly" (Parent Part) has a BOM that lists 1x Mini-PC, 2× 32GB RAM Sticks, and 1x NVMe SSD.
- **Supplier (The Source):** A Supplier is the external entity from whom you purchase parts. In a relational DB, this usually links to a SupplierPart table, which bridges your internal Part ID to the supplier's specific SKU and pricing structure.
- **Location (The Coordinate):** A hierarchical tree defining where physical things live (e.g., Room → Rack 1 → Shelf B). Stock entries must point to a specific Location ID.

Relational Database Dependency & Import Order

When building or importing into a PostgreSQL database, you must respect **foreign key constraints**. You cannot import a Stock item pointing to location_id = 5 or part_id = 101 if those IDs do not exist yet.

To prevent import errors, you must populate your CSV data in this specific order:

1. **Locations** & **Suppliers** (They have no dependencies on parts).
2. **Parts** (Can link to suppliers, but independent of stock).
3. **BOM** & **Stock** (Dependent on both Parts and Locations existing first).

Realistic Sample CSV Datasets

Here are realistic, normalized sample datasets structured exactly how a relational backend or InvenTree import engine handles them.

1. Locations (locations.csv)

2. Suppliers (suppliers.csv)

3. Parts (parts.csv)

Note: is_assembly flags whether this part can have a BOM recipe.

4. Bill of Materials (bom.csv)

Links parent parts to component parts using foreign keys (parent_part_id → parts.id).

5. Stock Items (stock.csv)

Tracks actual instances. Notice how we can have multiple distinct stock batches of the exact same part_id in different locations.

Database Design Insight for Your Journey

As you dig into PostgreSQL for this project, pay close attention to how InvenTree implements **Adjacency Lists** for hierarchies. Both the Part table (Categories) and the Location table use a parent_id pointing back to their own table's id.

This self-referencing relationship allows you to create infinitely deep nested locations (Home → Office → Desk → Drawer 1 → Component Organizer → Bin A) using a single simple table structure. Looking at how PostgreSQL uses recursive Common Table Expressions (CTEs) to query these trees is highly recommended as you build your own system.

Locations.csv

```
id,name,parent_id,description
1,Server Closet,,Main network closet in hallway
2,Rack A,1,42U Primary Server Rack
```

```
3,Parts Bin 01,,Storage bin on workbench
b
```

Suppliers.csv

```
id,name,url,contact_email
10,DigiKey,https://www.digikey.com,sales@digikey.com
11,Mouser,https://www.mouser.com,sales@mouser.com
```

Parts.csv

```
id,name,IPN,description,is_assembly,default_location_id
101,Ubuntu Micro Node,HW-NODE-01,Mini PC node for virtualizat
ion cluster,True,2
102,DDR4 32GB SODIMM,COMP-RAM-01,3200MHz Non-ECC Memory Modul
e,False,3
103,1TB NVMe SSD,COMP-SSD-01,PCIe Gen4 M.2 2280 Storage,Fals
e,3
104,Cat6a Patch Cable 1ft,CBL-C6A-01,Slim run ethernet cable,
False,1
```

Bom.csv

```
id,parent_part_id,component_part_id,quantity,note
1,101,102,2,Requires two sticks for dual-channel
2,101,103,1,Primary boot and VM storage drive
```

Stock.csv

```
id,part_id,quantity,location_id,status,serial_number
501,102,4,3,In Stock,
502,103,2,3,In Stock,
503,101,3,2,Operational,NODE-S_N-01
504,104,10,1,In Stock,
```