

File Edit Selection View Go Run Terminal Help

assets.ts - mms-server - Visual Studio Code

EXPLORER

MMS-SERVER

coverage

node\_modules

src

assets.ts

enums.ts

jobs.ts

properties.ts

spares.ts

users.ts

database

helpers

middleware

models

routes

types

index.ts

vitest

.env

.gitignore

package-lock.json

package.json

prettier.config.js

README

tsconfig.json

vitest.config.ts

assets.ts

src > controllers > assets.ts > getAssetTree > propertyId

1 import { Request, Response } from 'express';

2 import \* as Assets from '../models/assets';

3 import \* as AssetRelations from '../models/assetRelations';

4 import \* as Jobs from '../models/jobs';

5 import makeAssetTree from '../helpers/assets/makeAssetTree';

6 import makeIdList from '../helpers/makeIdList';

7

8 export async function getAssetTree(req: Request, res: Response) {

9 try {

10 const propertyId = req.params.propertyid;

11 const getAssetTree = await Assets.getAssetTree(parseInt(propertyId), 0);

12 const tree = makeAssetTree(getAssetTree)

13 res.status(200).json(tree);

14 } catch (err) {

15 console.log(err);

16 res.status(500).json({ message: 'Request failed' });

17 }

18 }

19

20 export async function getAsset(req: Request, res: Response) {

21 const assetId = parseInt(req.params.assetid);

22

23 try {

24 const assetDetails = await Assets.getAssetById(assetId);

25 if (assetDetails.length > 0) {

26 const propertyId = assetDetails[0].property\_id;

27 const getChildren = await AssetRelations.getChildren(assetId);

28 const idsForRecents = makeIdList(getChildren, 'descendant\_id');

29 const recentJobs = await Jobs.getRecentJobs(idsForRecents);

30 const children = await Assets.getAssetTree(propertyId, assetId);

31 const tree = makeAssetTree(children, assetId)

32 res.status(200).json({ assetDetails, recentJobs, tree });

33 } else {

34 res.status(500).json({ message: 'Request failed' });

35 }

## Maintenance Management System

# Technical Documentation

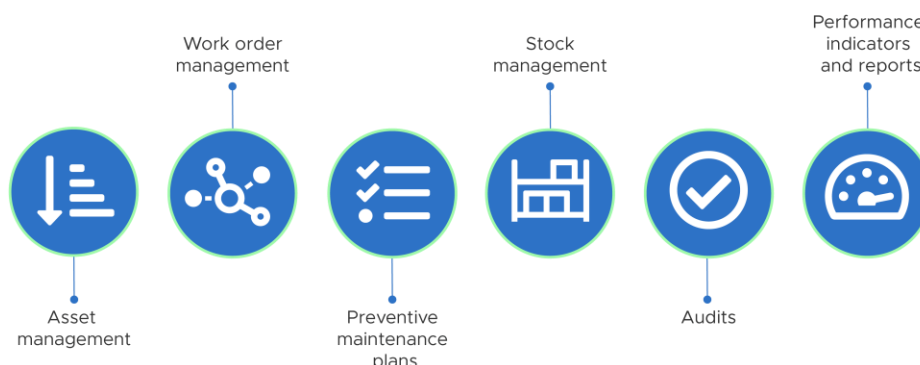
This Document outlines the **key challenges** faced when developing this **full stack cloud-based** system, along with the information and **solutions** required to solve these problems.

By Jake Gallagher

A computerized maintenance management system or CMMS is software that centralizes maintenance information and facilitates the processes of maintenance operations. It helps optimize the utilization and availability of physical equipment like vehicles, machinery, communications, plant infrastructures and other assets.

- IBM

### MAIN FUNCTIONS OF A CMMS



# Hierarchical Data

One of the Main problems with designing this type of system is the requirement for hierarchical data, take for example a Car, it has many child components e.g., engine, transmission, suspension, interior. Each of these components can be made up of many child components and so on for 'n' number of generations.

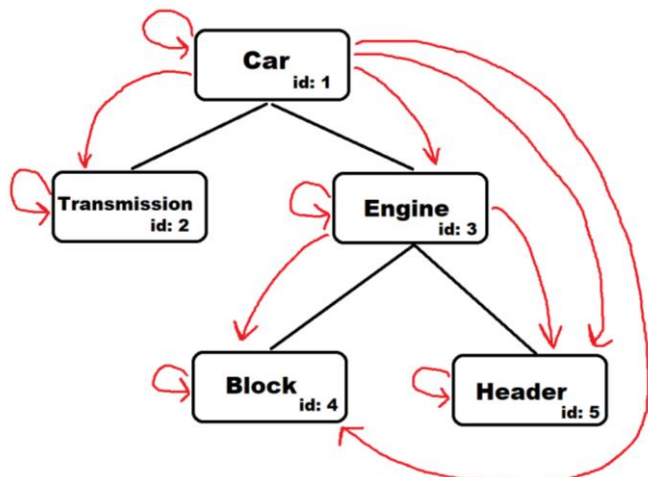
Due to the large number of potential components as well as the unspecified tree depth it becomes quite dangerous to handle this data with traditional SQL methods such as only using a parentID column or Nested Sets.

During my research I came across a presentation delivered by a consultant from Percona (a database that extends MySQL and adds some interesting features for scalability, such as single row locking). One of the methods mentioned in this presentation was the SQL anti-pattern "**Closure Tables**", this method relies on an additional table of data that holds information on the relationship between 'ancestors' and 'descendants'.

For each relationship between a descendant and every one of its ancestors there is an individual row in this table holding the ancestor\_id, descendant\_id and depth (difference in generations between ancestor and descendant), importantly there is also a row with depth '0' where the ancestor\_id and descendant\_id are both pointing at the descendant, this allows for easy insertion of new children components.

The main benefit of this method is the **fantastic efficiency of querying sub-trees**, there is no need to traverse the tree to find all components that are descended from the one you have chosen, simply query the closure table to find all rows that have your item as an ancestor, take the descendant id and find all component details using that.

Each red arrow represents one row in the Closure-Table with the ancestor\_id being where the arrow originates and the descendant\_id being where the arrow ends, with the depth being the number of generations difference e.g.,  
Car -> Car = depth 0  
Car -> Header = depth 2



## Tabulated Data – Filtering and Sorting

One of the key features of any Maintenance Management System is the large volume of data that needs to be shown in a clear and concise manner. This obviously calls for the use of tables, however building individual tables for each section that requires them is inefficient due to the large variety of field types in the application and so it is not possible to use a premade component. To solve this issue, I created my own Filterable & Sortable table component, this allows for complete control over the data that can be shown and allows for fast development in the future if a new field type is required.

Some of the field types include:

**Date** – displayed as '25/04/23' whilst remaining 2023-04-25 00:00:00 under the hood to ensure correct and easy sorting.


**Stock count** – displaying a tick, warning, or cross next to the numerical value for easy recognition of low and empty stock counts.

**Links** – one displaying the ID and the other displaying a chosen name but both linking to the ID link.

**Edit / Delete** – both calling the Modal reusable component for their respective functions.

The method I have chosen to approach this problem is sending a configuration object along with the data to the table component, this object dictates what columns of the table can be sorted/filtered as well as the type of the column and passes through some helper functions as well. The most useful part of this method is due to the way it standardizes the config object it opens up the possibility of using custom fields in the future, allowing the client to create/choose their own column names as well as the type of data making the whole system far more configurable, this is something I intend to look at in the future.

The following screenshot shows the data filtered where **location** includes the string “consumables” and **sorted** by “remaining stock”



Cardboard Co. ▾

Properties  
Jobs  
Assets  
Spares  
Settings

Spares Management + Add Spares Item

Location ▾  
consumables

✓ Greater than 1 Months supply ⚠ Less than 1 Months supply ✗ Nil stock remaining

Part Number	Manufacturers Part Number	Name	Manufacturers Part Name	Location	Remaining Stock ↓	Usage (AVG per Month)	Adjust Stock	Delete
<u>WS-10mm-Drillbit</u>	10mm Drillbit HSS	10mm Drillbit HSS	10mm Drillbit HSS	Consumables Cupboard	✗ 0	2.33		✗
<u>15580</u>	N/A	Non-leaded solder	Non-leaded solder	Consumables Cupboard	⚠ 2	3.33		✗
<u>22195</u>	N/A	AA Batteries	AA Bateries	Consumables Cupboard	✓ 16	1		✗



