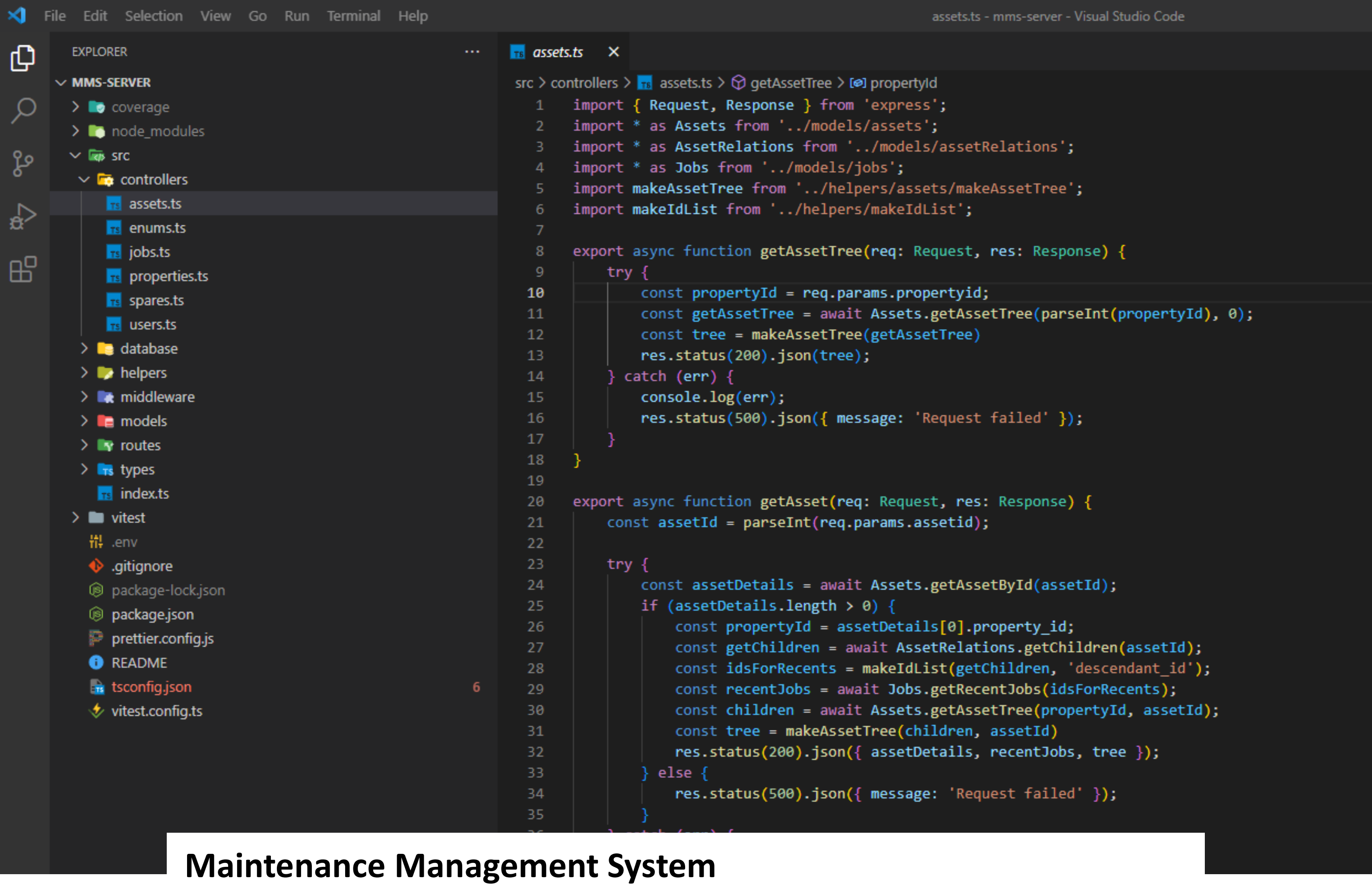


```
import { Request, Response } from 'express';
import * as Assets from '../models/assets';
import * as AssetRelations from '../models/assetRelations';
import * as Jobs from '../models/jobs';
import makeAssetTree from '../helpers/assets/makeAssetTree';
import makeIdList from '../helpers/makeIdList';

export async function getAssetTree(req: Request, res: Response) {
    try {
        const propertyId = req.params.propertyid;
        const getAssetTree = await Assets.getAssetTree(parseInt(propertyId), 0);
        const tree = makeAssetTree(getAssetTree)
        res.status(200).json(tree);
    } catch (err) {
        console.log(err);
        res.status(500).json({ message: 'Request failed' });
    }
}

export async function getAsset(req: Request, res: Response) {
    const assetId = parseInt(req.params.assetid);

    try {
        const assetDetails = await Assets.getAssetById(assetId);
        if (assetDetails.length > 0) {
            const propertyId = assetDetails[0].property_id;
            const getChildren = await AssetRelations.getChildren(assetId);
            const idsForRecents = makeIdList(getChildren, 'descendant_id');
            const recentJobs = await Jobs.getRecentJobs(idsForRecents);
            const children = await Assets.getAssetTree(propertyId, assetId);
            const tree = makeAssetTree(children, assetId)
            res.status(200).json({ assetDetails, recentJobs, tree });
        } else {
            res.status(500).json({ message: 'Request failed' });
        }
```

Maintenance Management System

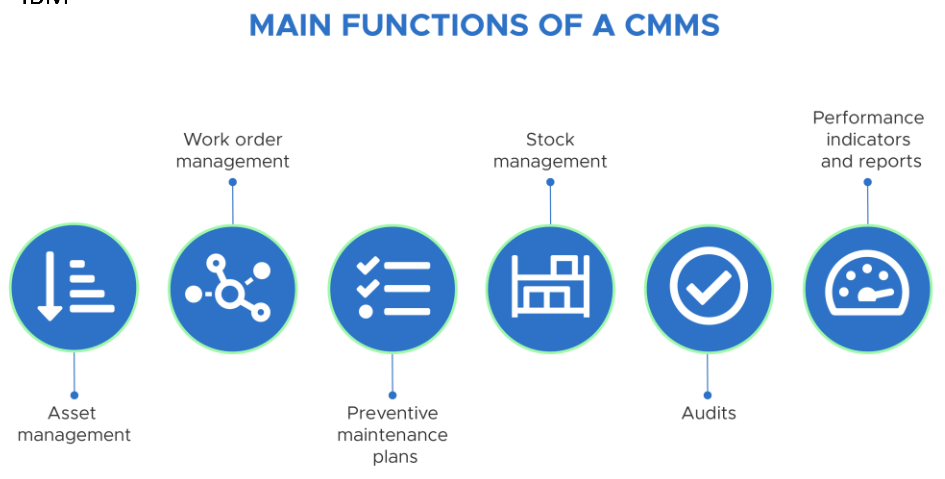
Technical Documentation

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



Hierarchical Data

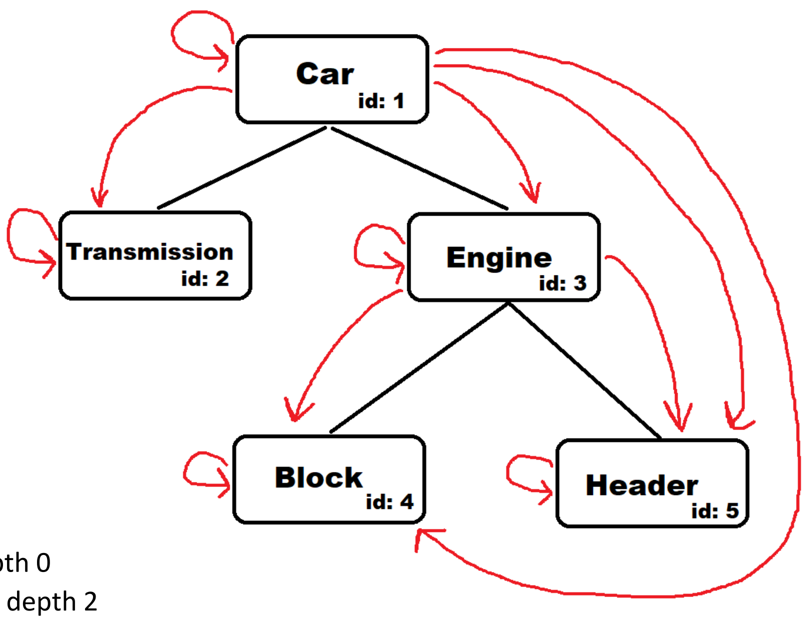
One of the Main problems with designing this type of system is the requirement for hierarchical data, take for example a Car, it has many child components e.g., engine, transmission, suspension, interior. Each of these components can made up of many child components and so on for ‘n’ number of generations.

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.



Car -> Car = depth 0
Car -> Header = depth 2
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.

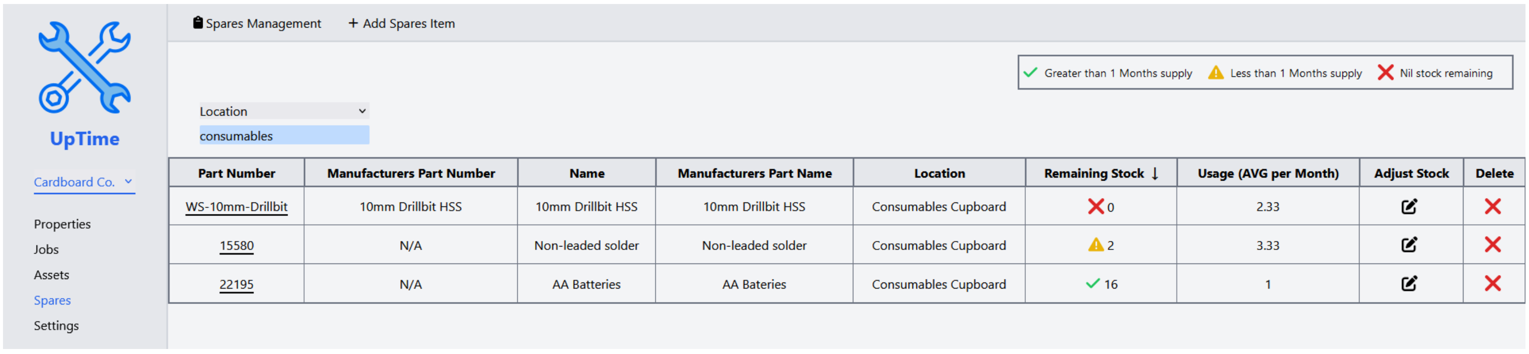
Tabulated Data – Filtering and Sorting

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"



Location
consumables

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

Azure Migration

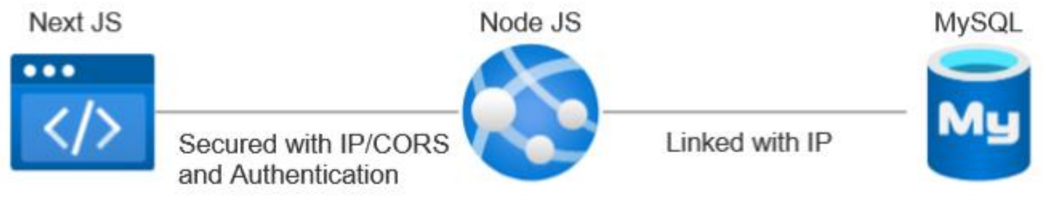
Migrating the project to the cloud had quite a few problems and difficulties to overcome. The first issue that I encountered was deploying the frontend to 'Azure static web app', due to this being a relatively new feature the documentation is not quite as helpful as some other areas. When the deployment failed it produced a warning for failing to remove a header from the git config, however this header turned out to be a red herring as it was not the error that caused it to fail, I had to open up the build log and managed to find a single line that stated, 'The limit for this Static Web App is 104857600 bytes.' At this point I carried out some further research and found that if you are trying to use SSR with a Next.js project then the bundle size is limited to 100MB. I then solved this issue by optimizing the dependencies, one thing that I hadn't noticed earlier was that when scaffolding the project with next it set typescript as a non-dev dependency, after fixing this the bundle size was small enough to deploy.

I Had some issues when deploying the NodeJS code to azure app service, these issues were primarily CORS and networking problems that I managed to solve by tinkering with the networking settings, for the CORS issues I found that the CORS settings area in Azure is not particularly useful as it overrides any CORS configuration in the codebase but is less configurable, because of these limitations I decided to use the CORS settings that are in my codebase in conjunction with ENV variables for further configuration.

Due to the way in which the App Service NodeJS app is deployed utilising GitHub Actions to allow for Continuous Deployment, I found that the most flexible way to configure and initialise PM2 is through the Azure configuration options 'Start Command', it can be done with the 'Start' command in the Package.json which is automatically run on every deployment with Azure App Service, however, in order to change your PM2 configuration this then requires a full redeployment of the code.

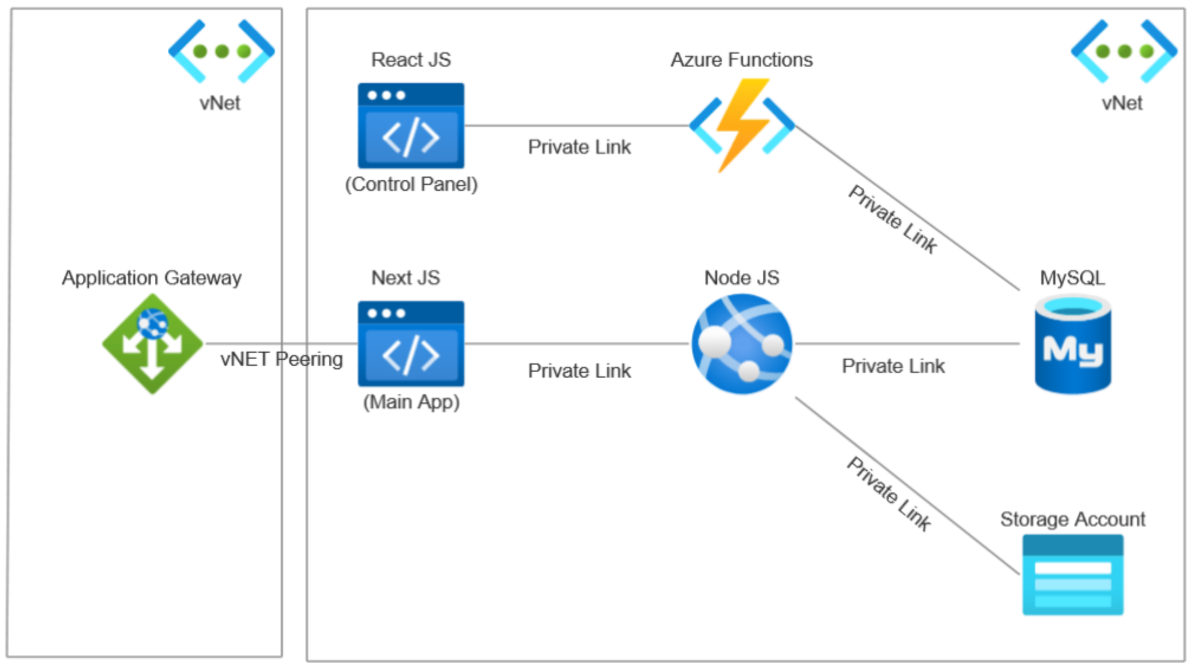
Deploying the Database was simple, and I encountered no major issues.

Current Cloud Architecture



Currently the set up consists of a MySQL database connected to a Node JS backend, the database is isolated by using the connection rules to allow only the server's IP to connect, this is similar to how the front end is connected to the backend, with the connections to the backend limited to the specific IP of the front end and inbuild authentication using JWT.

Planned Cloud Architecture



The current planned future development of the project includes Compartmentalising the project into Virtual Networks and connecting them allowing for a single point of entry which will allow me to lock down access to the backend and database further than is currently available.

The storage account will be necessary for handling files which is a feature which I plan to develop soon, this will allow for the storing of files against jobs as well as functionality to 'Mass Upload' data from CSV formats to reduce the hassle of creating large quantities of data in the system.

Another feature that I have identified as having future potential is the creation of a standalone 'Control Panel', this will allow me to create a layer of abstraction between the database and common functions which currently need to be actioned directly in the DB but which are not suitable to be housed within the main app. However, this may then require another application gateway to maintain the integrity of the vNET that is only accessible through vNET Peering, so this will come down to the cost involved as to whether it is worth it at this stage.