

RMM Service

A Remote Monitoring and Management (RMM) platform helps IT professionals manage a fleet of Devices with Services associated with them. This Web Service will fulfill the most basic requirements of an RMM by keeping a simple inventory of Devices and Services to calculate their total costs. The service will provide its functionality via a RESTful API.

Next, find a brief description of some system design choices. This does not pretend to be an exhaustive nor a complete system design document.

The data model

In the next diagram, you will find two entities: *device* and *service*. There is a many-to-many relationship represented with the table *device_service*.

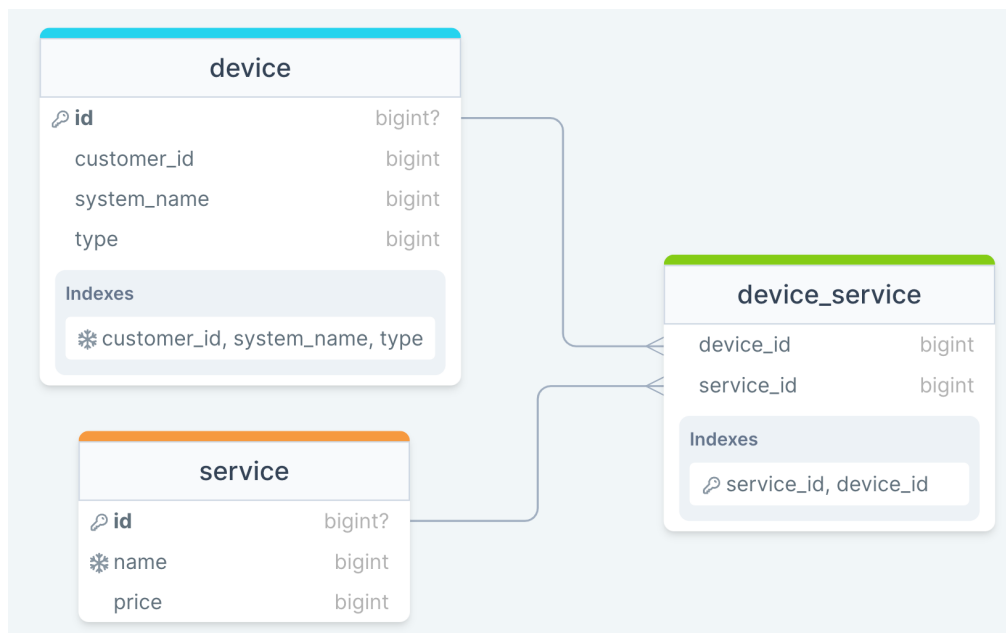


Fig 1 - ER diagram

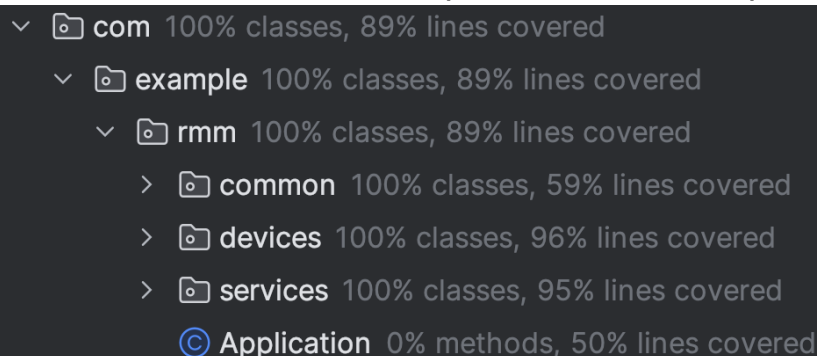
- A device cannot be duplicated for the same customer as a unique constraint comprising *customer_id*, *system_name*, and *type* in the *device* table guarantee that.
- A service cannot be duplicated as a unique constraint is enforced on the *name* column.
- In the joining table, a compound primary key (*device_id* and *service_id*) is implemented to prevent the assignment of a service to the same device more than once.
- Devices can be grouped by customer, for that, a *customer_id* column is present in the *device* table. This will support the generation of reports like the one shown in the example contained in the [Feature Requirements](#) section.

System design notes

- Devices are grouped by customer to better support reporting (provide any integer for the ID).
- All the devices are required to have, at least, one *base service*.
- Initially, the base service represent the device cost and it's automatically added on every new registered device.
- The cache will save the device ID and the sum cost of all the services associated to that device. A cache entry might be <1, 64>, being 1 the device ID and 64 the total cost of its services.
- The cache is updated when the associations between a device and its services is altered, the cases are:
 - Register a new device - a base service is associated to it.
 - Delete a device - services are associated to it get remove from the relationship. This is not strictly necessary because the device won't appear in any search, however, it will be unnecessarily using memory space.
 - Add/remove services to/from a device.
- The cache *put* and *remove* operations are executed asynchronously; it contributes to lower latency – faster responses.
- To modify (add/remove) the services assigned to a device a single endpoint was designed. It was developed to receive a set of the services IDs and the action (ADD/REMOVE) to be executed against the device. Implemented in that way, any additional action can be supported in the future (DISABLE, ENABLE, etc.) without breaking the contract.
- ConcurrentHashMap was used to implement the rudimentary cache, it's performant, thread safe and satisfy the requirements of the application. [Here](#) some details of its advantages against other solutions for local cache.
- Input/Request validations return an *errors* array in the response in case of failures.

Implementation notes

- The service is designed to have three main layers: Controller, Service, and Repository.
- The source code is structured by feature instead of by infrastructure type:



```

  ▾ com 100% classes, 89% lines covered
    ▾ example 100% classes, 89% lines covered
      ▾ rmm 100% classes, 89% lines covered
        > common 100% classes, 59% lines covered
        > devices 100% classes, 96% lines covered
        > services 100% classes, 95% lines covered
        © Application 0% methods, 50% lines covered

```

- There is a package for devices which contains the business logic related to devices. Inside that package, the regular controller, service, and repository packages will be found.
- Test coverage of 100% files and 89% lines.

Further optimizations

- A revisit on the used data structures (sets, maps, etc.) and the way they are initialized should be done checking that no more than the needed memory is allocated. Put special attention to the configs of the local cache and its defaults.
- The number database queries can be significantly reduced using caching in other parts of the code.
- Avoid using exceptions to control business flow, creating exceptions is more expensive than simply returning a generic message that can contain success and failed responses.

Further improvements

- Did you see the *ServiceService* class? What a name! Naming can be improved in a lot of different places of the applications.
- It's not safe to save money values in float variables, *BigDecimal* is a better option.
- Use *ResponseEntity* for greater flexibility on the returned types, headers, status codes, etc.
- *LocalCache* should implement an interface probably named *Cache* so it can be replaced by a different implementation – coding to interfaces, not implementation.
- When modifying devices services, don't add services if they are already in the device list – improve idempotency. Easy to implement if the device in the cache, apart from the cost, have also the services IDs.
- Tests can be simplified in a lot of different places.

The missing parts

- Code documentation.
- More exhaustive checks for the query params, path variables, headers and their sizes and format.
- More error details for duplicated errors – currently returning “data integrity” errors.
- Errors are ignored in many places; they should be gracefully handled.
- Make more use of interfaces than implementations, so we can have a more decoupled code.
- More logging is needed so we can trace and debug errors more easily.
- Metrics should be collected to monitor the service health.
- Move some configs to environment variables or config files.
- Work on possible improvements:
 - Cache by customer.
 - Customer cost report.
- Unit tests.

Building and Running

- Run **locally** (Java 21 JDK required – not tried with other Java versions):
 - `./gradlew bootRun`
- Run with **Docker** (first execution of `bootBuildImage` can take over 3 minutes):
 - `./gradlew bootBuildImage`
 - If want to see the logs: `docker run -p 8080:8080 normm`
 - If want to run in detached mode: `docker run -d -p 8080:8080 normm`

Open <http://localhost:8080/swagger-ui/index.html>

Testing

`./gradlew bootTestRun`