**Problem:** Factories often face challenges with the real-time monitoring and management of production equipment, leading to downtime, inefficiencies, and costly repairs. Without an integrated system, managers struggle to quickly add new machinery, detect anomalies, or scale operations smoothly. This leads to delays in production and increased maintenance costs.

**Solution:** The factory management system provides real-time tracking of equipment status, output speed, and other critical metrics. It automatically alerts the maintenance team when anomalies occur, reducing downtime. Managers can easily add new equipment or production lines through an intuitive interface, ensuring scalability. Compared to other solutions, this system offers better data-driven insights by storing historical performance data, allowing factory managers to optimize operations over time.

1. **Real-time monitoring** ensures immediate response to operational issues, minimizing production delays.
2. **Automated alerts** for anomalies help reduce maintenance response time, lowering downtime.
3. **Easy scalability** allows managers to add machines or production lines effortlessly as the factory grows.
4. **Historical performance data** gives managers better insights for predictive maintenance and operational improvements, leading to optimized production efficiency.
5. The intuitive interface simplifies management, enabling even non-technical users to effectively handle factory operations.

**Nouns:**

Factory, production equipment, production lines, operational status, performance metrics, machine, managers, machines, anomaly, maintenance team, historical data, scalability, alert, predefined conditions, downtime, response time.

**Verbs:**

Contain, monitor, track, add, modify, alert, stored, analyzed, optimizing operations.

**Rules:**

1. **Factories** will contain **production equipment** organized into multiple **production lines**.
2. The system will **monitor** and **track** the **operational status** and **performance metrics** of each **machine** in real-time.
3. **Managers** can **add** or **modify** the attributes of **machines** and **production lines**.
4. **Anomalies** (e.g., output speed drops or power outages) will trigger **alerts** for the **maintenance team**.
5. **Historical data** will be **stored** and **analyzed** to improve **scalability** and assist in **optimizing** **factory operations**.
6. **Alerts** will be sent when **predefined conditions** are met, reducing **downtime** and improving **response times**.

**Concept Diagram:**

图示

描述已自动生成

**Logical Diagram for documental database:**

图示

描述已自动生成

**JSON Examples:**

文本

描述已自动生成文本

描述已自动生成

<< Factory

Machine >>

文本

描述已自动生成

<< Maintenance

Team

Things stored in cache:

Factories id and name: HASH(id: name)

Production lines id and belonging factory id: HASH(pl\_id: fac\_id)

Machines id and belonging production lines id: HASH(m\_id: pl\_id)

Status of machine (if any record is not fixed yet): STRING; mach:id:status == ‘ok’

Teams id and leader name: HASH(id: name)

Number of records solved by team: STRING; team:id:nrec == 3

The factories id and name is a tricky one, since I basically needs to save an array of hashes, which is hard and inelegant in Redis. I have to create a hash with {id: …, name: …} for each factory, and then push keys of these hashes into an array. In this case, getting them means getting all keys and manually get hashes from keys within a iteration in JS. However, since I don’t want to put too much data IO logic in JS, I did it in another way: create a big hash with keys are ids and values are names. As long as there’s no third attribute I need to insert into, this works well and I can HGETALL to get the whole array without too much postmodifying.

Similar method is used to prodlines, machines and teams. Although they actually can be stored in other structures, using same structure makes my code more robust, consistent and readable.

Status of machines and number of solved issues are similar, if consider Boolean as 0/1 number, or assume a ‘status number’ on each machine tracking number of unsolved issues on it, they are the just same thing but different name. Since there’s no number or integer in Redis, string is the best choice to them.

All cached values has a corresponding “xxx:cached” value, which is either true or deleted, depends on logic and/or expiration time. This part is inspired by the repo shared by professor, which allows me not deleting and re-creating everything when expire, but leave them alive but untrusted. A little bit more memory-expensive but much faster.

The first thing to do is FLUSHALL, which in my code is actually done when the user firstly tries to access some actual data. But in larger view, the first thing did after connecting to the database is FLUSHALL.

Then, in most cases when the user act causes data extraction from MongoDB, the code firstly checks if it’s cached in Redis, mostly by check if “xxx:cached” exists (not expired). If so, then get data from Redis, otherwise get data from MongoDB and corresponding data will be cached into Redis, and prepare for next time usage.