# Design summary – Employee Search API

## Approach

For designing an employee search API for up to 1M users, I have built a fast.api MVP solution to start with. Everything is written with core Python without any 3rd party dependencies.

* I care a lot about teamwork, so I spent some time clearing out the README.md guide with good documentation to get started for others BE fellow developers.
* Also included in is a Swagger doc at `/docs` endpoint where FE engineer can find it easily to implement the API
* For better readability, I spent time refactoring the code into a well-structured project so that new features can be easily added to the project in the future.
* There is also a tests folder where I put some basic test for the search api and the rate-limiting
* The project is also well containerized so it can be ready for deployment via either docker-compose /docker-swarm or k8s microservices

## Architectural decisions

* Type safety and security: we usually use fast-api with Pydantic for type-safety checking for the API endpoint (null exception and code linting inside the python project, which is very helpful for developers to help debug faster). However, as I could not use 3rd party libs, I have at least implemented a minimum safety check via models (which u can find in the models.py) and a lot of checking on the database service (to avoid sql injection)
* Rate-limit: To protect the API, using fastapi middleware, I have implemented a basic version of the famous **Sliding window** algorithm to track via IP which receive two parameters: window\_size (length of rate limit in second) and max\_requests (maximum number of requests within the window\_size timebound)

Idea to improve (with assumptions / trade-offs)

There are several points that we can consider improving

* For the searching algorithm, it is now using SQL’s LIKE basic searching, but in the future when the scale up to 1M user, the search will start slowing down. We can use other opensource solution to speed up the search like Solr or ElasticSearch (which search via vector database) -> trade off of this is we have to handle more logic between two database (SQL database and vector database, which put more effort within writing the database, to trade back the speed of the search)
* As the filter can be duplicated a lot among user’s request while filter via department, company...therefore we can implement a caching strategy (ram caching via Redis) to cache the database expensive search result. Therefore, saving the resource to run each SQL's request. -> Tradeoff is we will definitely use more RAM resource in the cluster, ec2
* Final point, we will need to have a system to monitoring and tracking:
  + query performance by time (track on the hardware resource whether we can fulfill the current amount of search daily)
  + search patterns (to optimize the search index)
  + cache hitting rate (caching effectiveness)

In my opinion, to have a good observation tool is a good key to understand user’s demand and adapt our system with fastpace