System design

What does the system do?

Build a web application to show the weather forecast of a city using https://www.metaweather.com/api/.

The goal of the system

- Users should be able to choose a city from the list easily
- Users should be able to see the five days forecast for one city.

System APIs

searchCity(cityName)

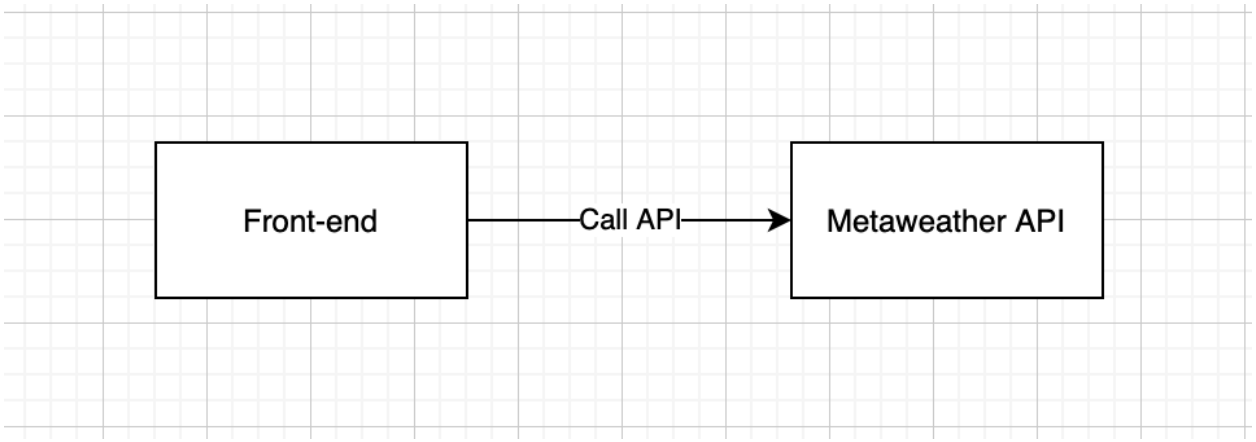
Parameter:
cityName: This should be the city name we want to search for
Output:
A successful search should return a list of cities which are matching the query

searchWeathersForCity(cityId)

Parameter:
cityId: the id of the city we want to search for. Since the meta weather uses the city id in their API contract, using cityId is a direct way instead of using the name of the city.
Output:
A successful search should return a weather forecast for six days.

High-level design

Naive approach

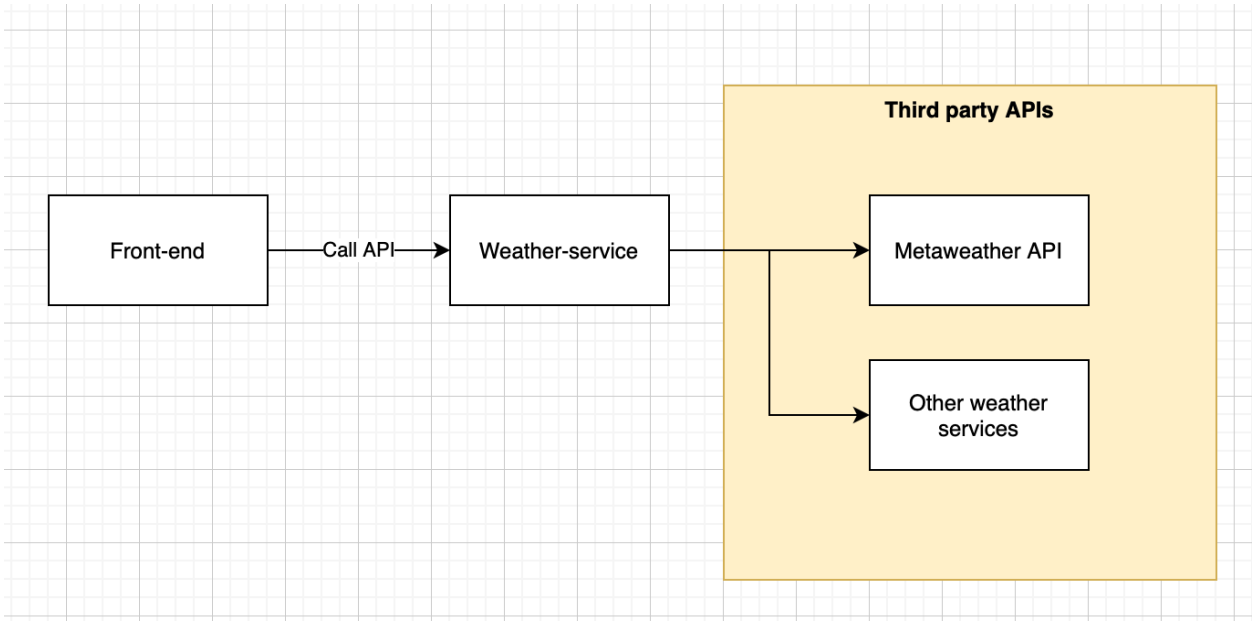


At a very basic level, we can just design a front-end client to call Metaweather directly and take the data from their end. However, this approach comes which a lot of disadvantages:

- What if Metaweather API is broken and we want to change to another API? Our client is tightly coupling with Metaweather API now.
- What if we want our system to scale and serve more requests?

Backend service as a middle man (Current implementation)

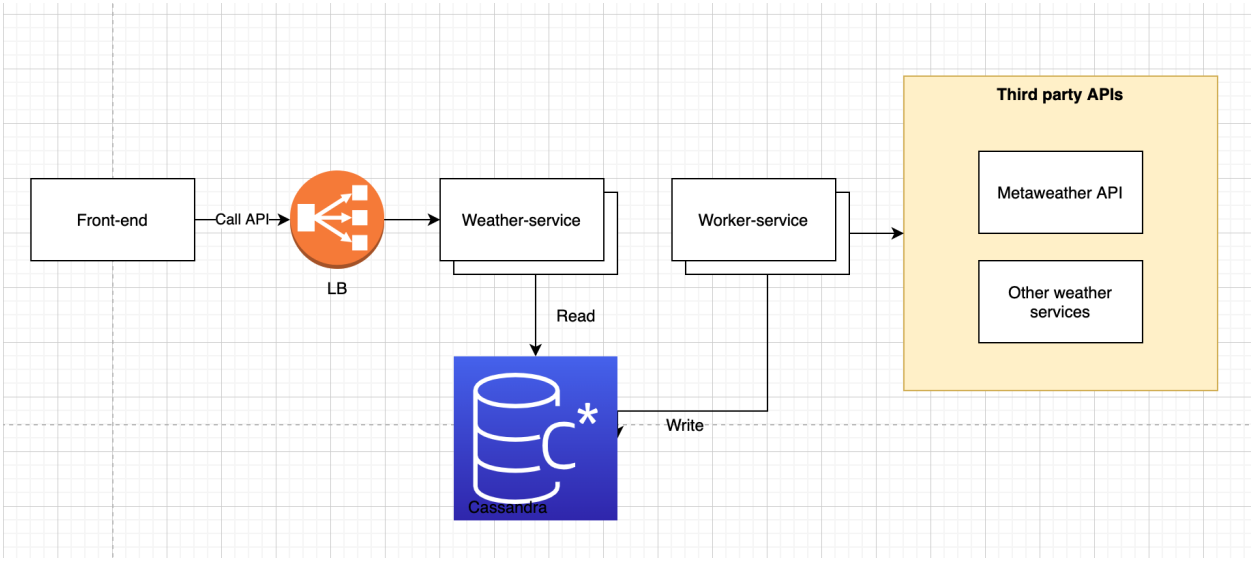
For the first problem, we can introduce more flexibility by adding a layer of backend service to get data from Metaweather API.



With this approach, our Client will no longer need to know which weather services we are using. We can use Metaweather API for now, and in the future, if needed, we can integrate more services into our system, and our client will only talk via our API contract at weather service. The system will stay the same way as our logic at the backend may develop in a different way too.

Scale for high load

As for the scaling problem, assuming that our system is getting bigger and the traffic becomes insane. How should we scale our application? One of the disadvantages of our system is that we have to depend on third-party APIs. Therefore, if we want to scale up our system, we need to contact them for scaling up at their infrastructure first. One potential solution we could use in the future is to maintain a database for our system so that we can own up the scaling by our side.

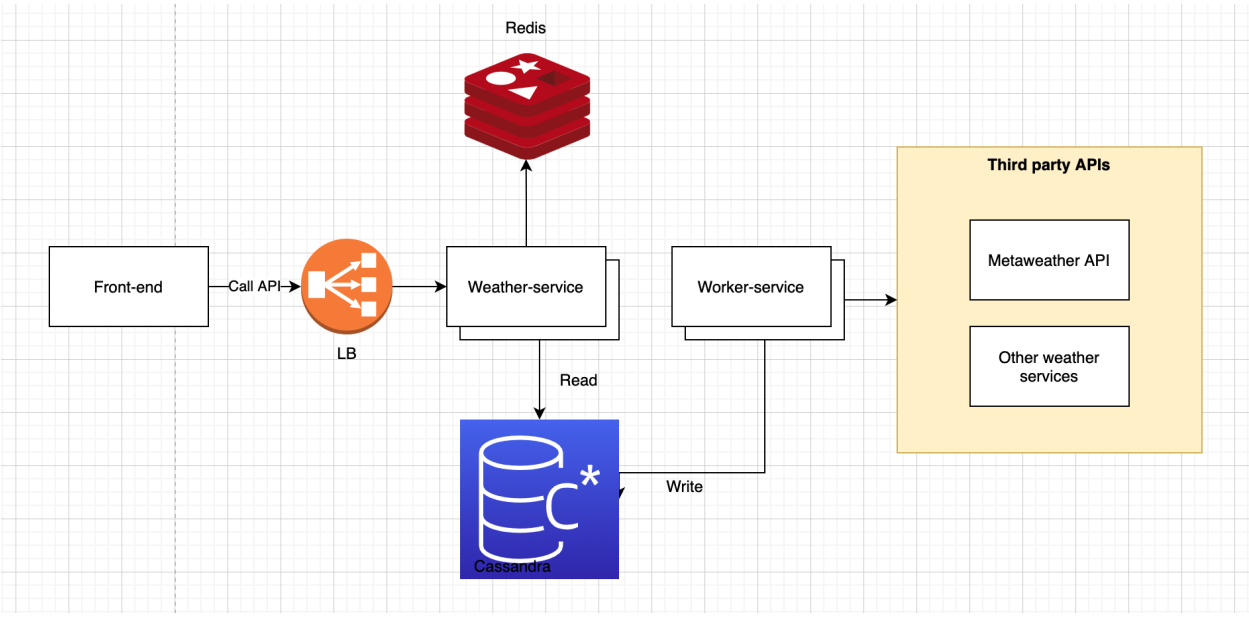


In this architecture, we introduce a Database storage Cassandra to handle all data collected for all possible sources and platforms. In our system, we should not need to use any of the ACID property, so we can utilize a NoSQL DB to store our data. Using Cassandra as our main DB, we can ensure the availability and scalability as it supports these things very well. A worker service will be in charge of getting all the data and putting it into our storage.

With this, we should be able to scale our system easily, with our own DB and our own services. A **Load balancer** of course will be introduced to dispatch requests evenly between our **weather-services.** We can also have **one or multiple workers for our sources.** With this, We will no longer need to worry about third-party API not able to handle big traffic.

Cache

Finally, we can use Cache to cache out the content of our weather. Since the weather data rarely change, we can cache the data as long as we can using the LRU cache.



CDN

Last but not least, we can use CDN to cache static content.