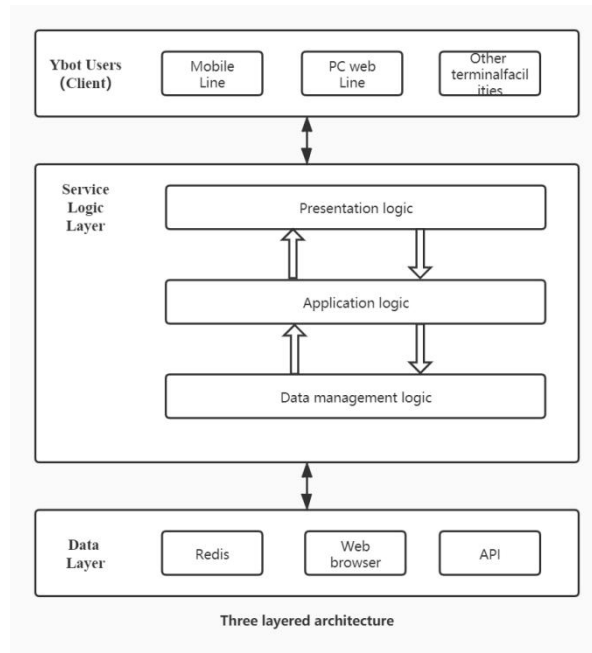
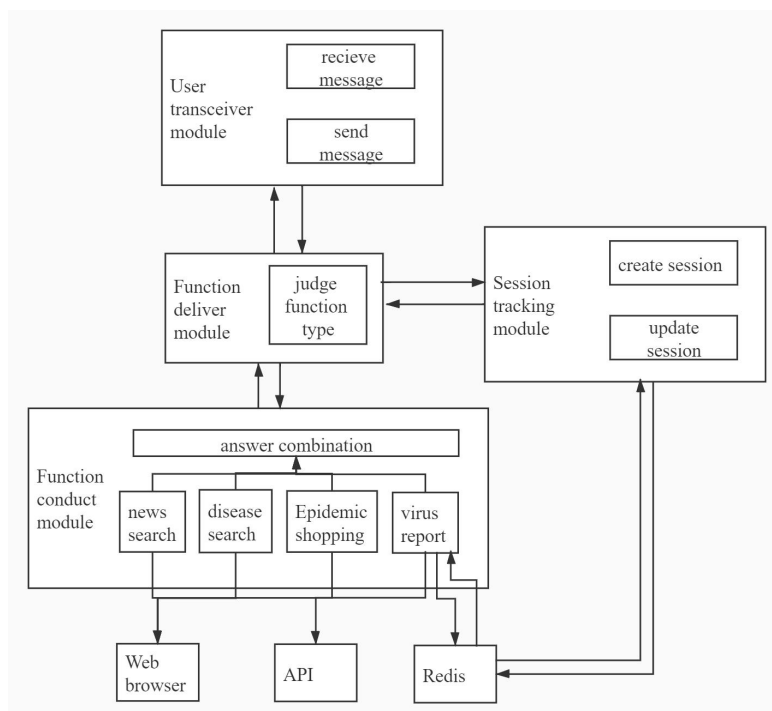


## Architecture:

Use three-tier architecture, divide it into three tiers, the key tier is service logic layer, which is composed of presentation logic, application logic, and data management logic. Each of them take charge of their needed functions.

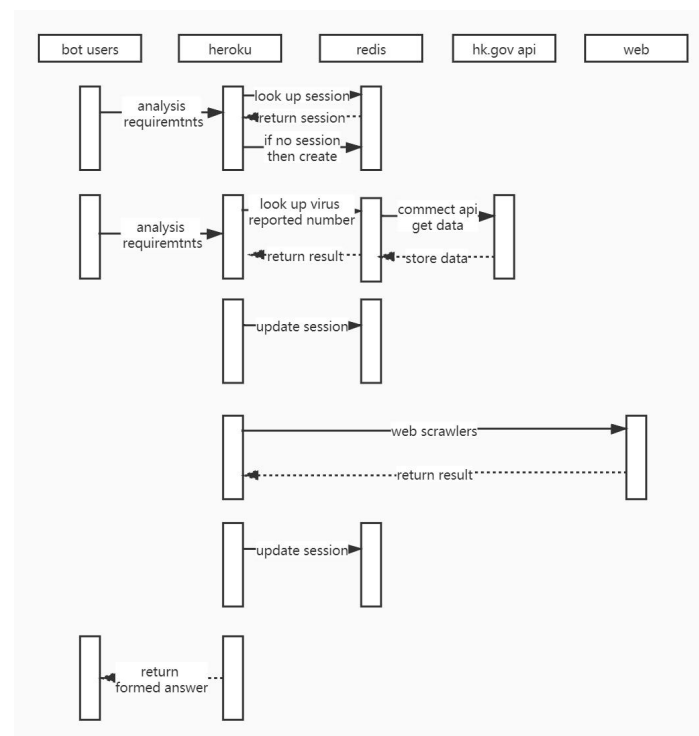


To be detailed, the presentation logic of logic layer is to receive user's message, abstract useful information and translate them to the next logic part, as well as to receive the prepared answer from the logic below and send them to its upper layer. It would be connected with application logic, and then connected with a much important one, data management logic. It would help deal with data acquire functions, including getting and updating data in redis and so on. Data management logic helps to connect the logic layer and the data layer.



For health care, here we provide function for them to search disease related news, search information about a certain disease or symptom, general situation of the reported virus, nearest hospital location, and online shop item search recommend for those who want to buy something for healing or protect. The dialog status of different users would be recorded in redis in their own session, this contributes to give result based on their past word , just when it's still valid, with information stored on the cloud we can continue to provide service for users without waiting continuously.

With these above, the three tiers are divided clearly and each has their well-defined role. In this way, it's easy to add some significant tag for them to solve data request conflict problem. And when we design the usage of limited redis, it is treated as a cache tool where we would like to put some frequent used data, store sessions and so on. Furthermore, many processes may visit data of the same function in redis. This may cause a deadlock, but here we use special key value suffix and take out user id from line message head, store them in redis so that one user would only use the information stored in his or her own session.



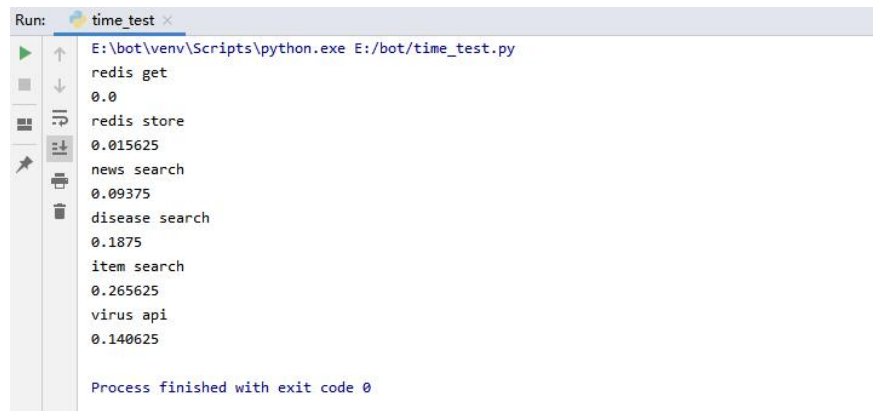
### Way to increase capacity of chat bot service:

First we consider the problem of QPS, from the scree shot below, we find actions that cost most time are news search, disease search, item search mainly because it's hard to find an appropriate API and we have to get data through crawlers. The time cost of API seems faster than search actions but still much slower than normal redis storage actions.

Virus api here would provide total reported covid 19 cases in Hong Kong, it wouldn't change in a single day so here we only get it through api once a day and then store it in redis. Its key is the date and value is the number. This would shrink the time cost in getting virus data frequently in a single day.

The problem of disease search is similar as virus api. The disease information we use in the website is unchanged for a long time so we would get some frequent gotten diseases ahead of time during spare time when there's not too much users. And for further use, only get data from where they store. Though get them from database also cost lots of time, it would be obviously better than by crawlers.

Compare with these two, the news and item details are not certain. They may change everyday, but we could also save time by get them by crawlers ahead of time before we need them. These data need to be updated in database frequently when system spare as well.



```
Run: time_test
E:\bot\venv\Scripts\python.exe E:/bot/time_test.py
redis get
0.0
redis store
0.015625
news search
0.09375
disease search
0.1875
item search
0.265625
virus api
0.140625
Process finished with exit code 0
```

In this way, the cost of once dialog would be quicker and in the same period it could react to more users and response quicker.

Second, our usage of redis is limited. Here because there's only 30MB, we only store sessions and some frequently used cache on it. To use the limited storage space in redis better, we set an overtime for these data according to their function and supposing valid period. These useless overtime data in redis world be deleted. Redis' default strategy is LRU, when the storage has been occupied up to a maximum percent, firstly those least recently used key would be deleted. Thus when user number increases, those less active would be knocked out.

For now we use redis to do these things, when the user amount increases greatly, we can put those overflow users in other storage platform other than radis, such as put them in a remote MySQL database in Google Cloud in a query table. To be more specific, we can put those active users in redis' limited space and other less active users in database because io in redis is much quicker than databases.

Besides users, other information such as item details, news contents, could also use this way to provide more space for redis by removing those less frequently used data into database and use redis to count how often they're used and when system spare we can move them back to redis to provide quicker read action. The code below is an example of overtime set and put frequent visited data into redis.

```
def virus_check():
    date = datetime.datetime.now().strftime('%F')

    session = redis1.get(f'{date}#virus')
    if session is not None:
        data = int.from_bytes(session, 'big')
        return data
    else:
        return ''

def virus_store(num):
    date = datetime.datetime.now().strftime('%F')
    redis1.set(f'{date}#virus', num.to_bytes(4, 'big'))
    redis1.expire(f'{date}#virus', 86400)
```

```
def shopping_get(item_name):
    item_content = redis1.get(f'{item_name}#itemcontent')
    if item_content is not None:
        return item_content.decode()
    item_num = redis1.get(f'{item_name}#itemname')
    if item_num is None:
        item_num = 1
        redis1.set(f'{item_name}#itemname', item_num.to_bytes(4, 'big'))
    else:
        item_num = int.from_bytes(item_num, 'big')+1
    return item_num

def shopping_record(item_name, content):
    if redis1.exists(f'{item_name}#itemname'):
        redis1.delete(f'{item_name}#itemname')
    redis1.set(f'{item_name}#itemcontent', content)
    return 1
```

### Ours is an example of PaaS:

Our bot is a PaaS. We only need to deploy our code written by python to Heroku, merely manage our applications and data, just as application hosting. We can use the pre-defined environment comprised of already deployed and configured IT resources. We just do coding and put packages and libraries in the file called requirement.txt and let Heroku do things left.

Furthermore, our bot has limited administrative control privilege, can deploy code but cannot operate some bottom thing in the service. This also justify it's a PaaS because heroku permits moderate level of administrative control over IT resources relevant to cloud consumer's usage of platform. We can develop, test and deploy our python project, on the pre-configures platform without managing them all by ourselves or use completed service.