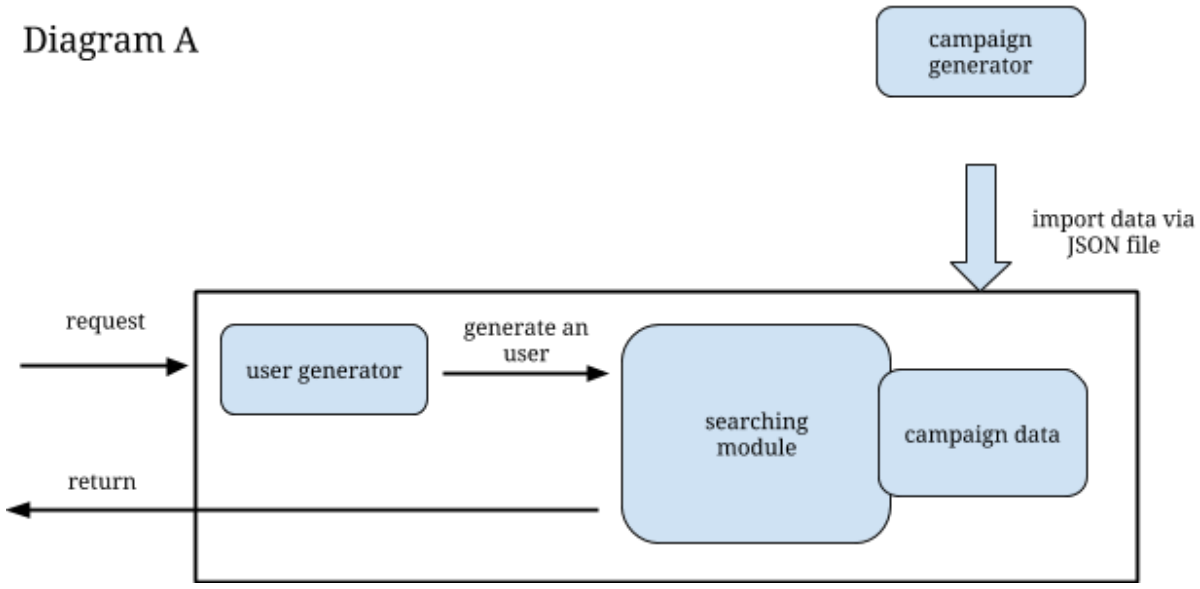
Test Project: Campaign

Brief

Preparation stage
- Generate the campaign data
- Import the campaign data to server app

Test stage
- Send request to the server app (=> generate a user => search the campaign)
- Return the campaign name meeting the criteria

Diagram A



In the test stage, the response is suppose to be done in less than 100ms, the faster the better.

The purpose is test the campaign searching performance.

You can use any language for them.

==========

Here's the each part details.

1. Campaign Data Generator

GET http://myhost:3000/campaign?x={number}&y={number}&z={number}

The generator has parameters: X (X <= 100) , Y (Y =< 26), Z (Z <= 10000)

Generate Z number of campaigns, each campaign has a target list (length is random and less than Y).
The target has an attribution list (length is random and less than X).
Each campaign has an offer price, which is randomly generated.

Here is a sample output JSON file:

```
[
	{
		"compaign_name": "campaign870",
		"price":3.25
		"target_list" : [
			{"target":"attr_A", "attr_list":["A0",...,"A99"]},
			{"target":"attr_B", "attr_list":["B0",...,"B36"]},
			...
		]
	},
	...
]
```

2. API and user data generator

Create a API for user generation.

GET http://myhost:3000/user

This endpoint will return a user info defined as following.

There's a counter for request (starts with 0), each request will increase the counter by 1.

Create a function to generate the following user info when it receive a request.

```
"user": "u"+counter,
"profile":[
	"attr_A" : "A" + random(200)
	"attr_B" : "B" + random(200)
```

```
        ...
    ]
```

The length of the profile will be increased by 1 everytime when a new user is generated. The attribute name and value contained in the profile need to follow the alphabet order, such as [“attr_A” : “A1”], [“attr_B” : “B23”], etc. When the profile length reaches its maximum length of 26, it gets reset to 1 again, which means it only contains “attr_A”.

for example:

The 1st user

```
{
    "user": “u1”,
    "profile":[
        "attr_A" : "A1"
    ]
}
```

The 3rd user

```
{
    "user": “u3”,
    "profile":[
        "attr_A" : "A23"
        "attr_B" : "B132"
        "attr_C" : "C45"
    ]
}
```

The 27th user

```
{
    "user": “u27”,
    "profile":[
        "attr_A" : "A109"
    ]
}
```

The 28th user

```
{
    "user": “u28”,
    "profile":[
        "attr_A" : "A109"
```

```
                "attr_B" : "B87"
        ]
}
```

3. Matching

Create a API to import the campaign data (JSON file) generated by step 1 , then save them in some data struct or database.
POST http://myhost:3000/import_camp

Definition

- Targeted User: If all targets of the campaign can be found in an user's profile, and the user’s profile attribute value can be found in the list of the campaign target attri_list.

Create a API for campaign searching with user info
POST http://myhost:3000/search

The request will POST a user info, and return the searching result.

Create a API for campaign searching
GET http://myhost:3000/search_auto

The request for the search API will trigger the function to create a user in step 2 and pass the user to the search function, if the user is a “Targeted User”, then Find the the campaign that has the HIGHEST price, otherwise return “none”.

Return:

```
        {
                "winner": {campaign name},
                "counter": {counter}
        }
```

For example:

Assume there’s 2 campaigns

```
        {
                "compaign_name": "campaign1",
                "price":0.25
                "target_list" : [
```

```
                {"target":"attr_A", "attr_list":["A0",...,"A9"]},
                {"target":"attr_B", "attr_list":["B0",...,"B16"]},
        ]
}

{
        "compaign_name": "campaign2",
        "price":0.35
        "target_list" : [
                {"target":"attr_A", "attr_list":["A0",...,"A19"]},
        ]
}
```

To help you understanding the abstract model, let's assume some campaign targeting {attr_A = "country", attr_list=["US","UK",...]}, {attr_B= "hobby", attr_list = ["football","fishing","sports"]}.

Cases

user	response	note
{ "user": "u1000", "profile":["attr_A" : "A5"] }	{ "winner": "campaign2", "counter": 1 }	only campaign2 matched
{ "user": "u2000", "profile":["attr_A" : "A5" "attr_B" : "B15"] }	{ "winner": "campaign2", "counter": 2 }	campaign1 and campaign2 matched. campaign2's price is higher
{ "user": "u3000", "profile":["attr_A" : "A15"] }	{ "winner": "campaign2", "counter": 3 }	only campaign2 matched
{ "user": "u4000", "profile":["attr_A" : "A52"]	{ "winner": "none", "counter": 4 }	no campaign matched

}		

==========

The Test Process

Preparation

1. Generate 1000 campaigns with the “Campaign generator”, X=50,Y=10,Z=1000, the output file is downloadable.
2. Import the above data to the server app with data struct or database to make the searching provision ready.

Case A

3. POST an user to “http://localhost:3000/search” to verify the search result

Case B

4. Use “wrk” (github.com/wg/wrk) to do the performance test
   `wrk -c 64 -d 10s http://localhost:3000/search_auto`