1. Calculator Function

Write a calculator function that has this signature:

```
function calculateNextState(jsonState: string, input: string): string
```

- The function will initially accept a jsonState which is null.
- The input is the next character that the user inputs.
- For example, for the calculation 12*4=, the inputs this function will be called will be:
    - 1
    - 2
    - +
    - 4
    - 3
    - =
- What will be the jsonState? It will initially (the first time the function is called), be null.
- The function returns a string which is also a jsonState.
- The next time the function is called, the jsonState passed will be the previous jsonState that it returned.
- The jsonState will always be a legal JSON object (e.g. '{...}'), which will always contain the field "display". This field's will be the string that the calculator will display. Other stuff can be in the jsonState, but that is up to the implementor (you…) to decide.

Example

```
let s = null

s = calculateNextState(s, "1")

console.log(JSON.parse(s).display) // 1

s = calculateNextState(s, "2")

console.log(JSON.parse(s).display) // 12

s = calculateNextState(s, "+")

console.log(JSON.parse(s).display) // 12

s = calculateNextState(s, "4")

console.log(JSON.parse(s).display) // 4

s = calculateNextState(s, "3")

console.log(JSON.parse(s).display) // 43
```

```
s = calculateNextState(s, “=”)

console.log(JSON.parse(s).display) // 55

s = calculateNextState(s, “+”)

console.log(JSON.parse(s).display) // 55

s = calculateNextState(s, “1”)

console.log(JSON.parse(s).display) // 1

s = calculateNextState(s, “=”)

console.log(JSON.parse(s).display) // 56

s = calculateNextState(s, “5”)

console.log(JSON.parse(s).display) // 5
```

Notes

- The calculator does not support operator precedence. All operators (+, -, *, /) have the same precedence.
- The logic of what to display? Use your own calculator to see what that logic is.

2. Unit Tests

Write unit tests that fully check this function.

3. Git

Create a Github account, and a Git repository. Add the code above to the repo. All the rest of the code you will write below, will be in this git repo.

The readme of the repo will include specific instructions in how to build and test the code in it.

4. Web Server

Write a web server that accepts a POST /calculate, where the body is a JSON with the fields:

- `calculatorState`
- `input`

The web server will pass the calculatorState (as a JSON) and input to the function in step #1 (if you need to serialize it to a string, please do), and the state returned from that function (as JSON) will be the response that the web server returns

If the calculatorState is not passed in the body (i.e. only input), then you should assume it is the initial state. You can check it manually using the curl below:

```
curl http://localhost:3000/calculate -X POST -H 'content-type: application/json' -d '{"calculatorState": null, "input": "1"}'
```

What should be returned is this:

```
{"display":"1", /* ...other stuff you need for state */}
```

The calculator state you get back can be used as the next calculatorState in the next curl.

The code should be in the same Git repository you created for the unit tests.

The README.md file in the root should describe how to run the server (i.e. what needs to be installed, and which commands need to be run to make it work)

I will be following the instructions and testing it manually using `curl`.

5. Integration Tests

Write integration tests for the web server above. The code should be in the git repository you created.

The README.md file in the root of the repository should also describe how to run the tests.

I will be following the instructions and checking that all the tests pass (besides looking at the code…)

6. Docker

Create a docker container from the web server you created in step 4. Use docker run to test it manually.

The Dockerfile should be in the git repository.

The README.md file in the root of the repository should also describe how to build the dockerfile and run it.

I will be following the instructions and testing it manually using `curl`.

7. Docker-compose

Use docker-compose to create a docker environment that runs all the microservices given by the lecturer to create a full app.

These are the docker images you need for all the microservices:

Redis

The image needed is redis:alpine. Look on the internet to figure out which port needs to be exposed for Redis.

This database is used for storing user information and session authentication information (it is used by the user-service and by the currency-backend)

user-service

The image needed is webdevtoolsandtech/user-service.

To find out where the redis is, it needs the environment variable REDIS_ADDRESS, which should be set to the address and port (e.g. something like foo:3782) of the Redis database.

currency-calculator

That is the service you wrote.

currency-frontend

The frontend of the application. The image used is webdevtoolsandtech/currency-frontend

currency-backend

The backend of the application. This is the entry of the web app. It needs the following environment variables:

- REDIS_ADDRESS: the address of the redis database, in the form addr:port
- SESSION_SECRET: a secret. Can be any string (e.g ‘lalala’)
- USER_SERVICE_ADDRESS: The address of the user-service microservice, in the form addr:port
- FRONTEND_ADDRESS: The address of the currency-frontend microservice, in the form addr:port
- CALCULATOR_ADDRESS: The address of the calculator-service microservice, in the form addr:port

8. E2E tests (optional)

Write e2e tests for the docker-compose you wrote in step 7.