DIGITAL ENVISION CODING TEST

BACKEND DEVELOPER

Write a simple node.js application to send a happy birthday message to users on their birthday at exactly 9am on their local time. For example, if one user is in New York and the second user is in Melbourne, they should be getting a birthday message in their own timezone.

Requirements

- Javascript / Typescript (preferred)
- Simple API to create or delete users only:
    - POST /user
    - DELETE /user
- User has a first name and last name, birthday date and location (locations could be in any format of your choice)
- The system needs to send the following message at 9am on users' local time via call to https://hookbin.com endpoint (create a new one for yourself): "Hey, {full_name} it's your birthday"
- The system needs to be able to recover and send all unsent messages if the service was down for a period of time (say a day).
- You may use any database technology you'd like, and you are allowed to take advantage of the database's internal mechanisms.
- You may use 3rd party libs such as express.js, moment.js, ORM etc to save development time.

Things to consider

- Make sure your code is scalable and has a good level of abstraction. For example, in the future we may want to add a happy anniversary message as well.
- Make sure your code is tested and testable
- Be mindful of race conditions, duplicate messages are unacceptable
- Think about scalability (with the limits of localhost), will the system be able to handle thousands of birthdays a day?

Bonus

For extra brownie points, add PUT /user for the user to edit their details. Make sure the birthday message will still be delivered on the correct day.

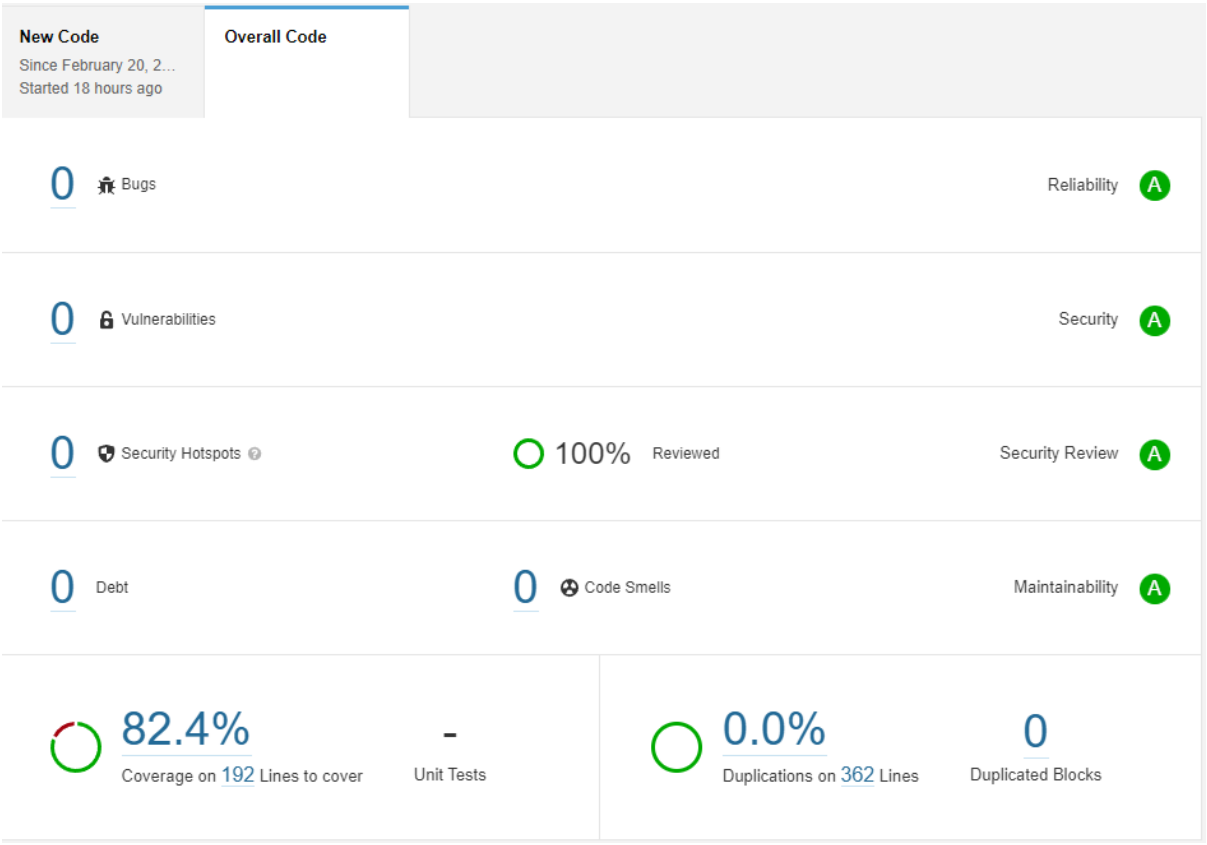
Code Approach and Explanation

https://github.com/AnangM/birthday-app-backend

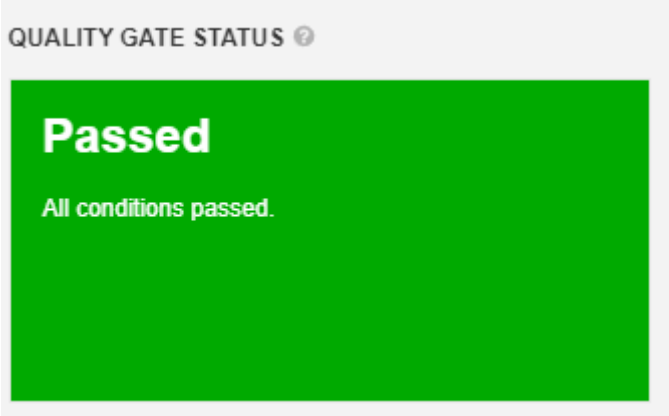
This simple node.js app is simple but challenging. I approach this problem by breaking it into 3 separate problems: User data, Worker, Job. User data basically just create, update and delete user data. When new user is created, the corresponding job will be created, therefore with update and delete that will change user's corresponding job. The Job and queue driven by postgres database. Then every half an hour the worker (cron) will get all jobs that is reserved for current or past hour to be delivered. Using Redis may improve queue performance but I don't think it is needed for this use case. All jobs are stored in UTC time so job will be delivered right around 9 o'clock user's time. For example if user are located in New York (UTC -5) the message will be delivered at 14 o'clock UTC. The worker will send current and past undelivered job so each user will get their message even when the system went down for a moment of time. After the job delivered, then setup new job for next year.

I am using nest.js as they make development easier and faster but still a robust system with wide community support. Also choosing postgres as they have many advantages than mysql especially when handling a lots of data.

The code quality monitored and quality checked by sonarqube.



Sonarqube can be part of our workflow (CI/CD) before deployment so code quality always be up to standard and able detect bug, vulnerabilities and security hotspot before deployed to server. Only code that pass quality gate can proceeds to next step in pipeline. As we can see from picture above, my code is free of bugs, vulnerability, code smells and 82% lines already covered by test. Only 1 security hotspot and it is because unsecured (weak password) local instance of postgres. Unfortunately the unit test is not picked up by sonarqube due to some errors in my local instance of sonarqube.



Testing done in unit testing, end-to-end testing and load testing. Unit and end-to-end testing done by implementing jest as jest natively supported by nest.js and load testing done using local installation of k6. Load testing done by simulating multiple user accessing the api ramping up from 0 to 100 users in the first 5 minutes, then hold 100 users for 10 minutes and ramp-down to 0 in the last 5 minutes.

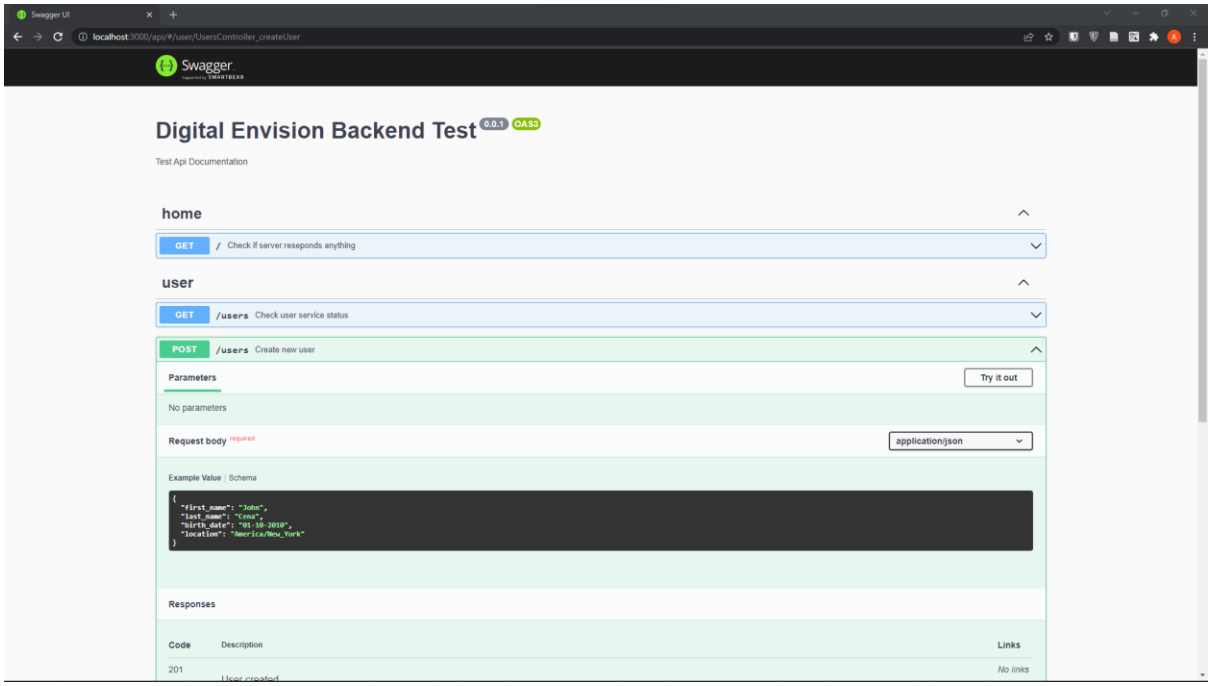
```
running (20m00.7s), 000/100 VUs, 87375 complete and 0 interrupted iterations
default ✓ [======================================] 000/100 VUs  20m0s

    ✓ User created succesfully

    checks.........................: 100.00% ✓ 87375     ✗ 0
    data_received..................: 33 MB   28 kB/s
    data_sent......................: 21 MB   17 kB/s
    http_req_blocked...............: avg=5.16µs  min=0s med=0s      max=7.11ms   p(90)=0s      p(95)=0s
    http_req_connecting............: avg=932ns   min=0s med=0s      max=2.86ms   p(90)=0s      p(95)=0s
    http_req_duration..............: avg=26.09ms min=0s med=23.86ms max=477.89ms p(90)=41.84ms p(95)=50.86ms
      { expected_response:true }...: avg=26.09ms min=0s med=23.86ms max=477.89ms p(90)=41.84ms p(95)=50.86ms
    http_req_failed................: 0.00%   ✓ 0         ✗ 87375
    http_req_receiving.............: avg=88.65µs min=0s med=0s      max=15.78ms  p(90)=82.26µs p(95)=525.29µs
    http_req_sending...............: avg=20.25µs min=0s med=0s      max=14.52ms  p(90)=0s      p(95)=61µs
    http_req_tls_handshaking.......: avg=0s      min=0s med=0s      max=0s       p(90)=0s      p(95)=0s
    http_req_waiting...............: avg=25.98ms min=0s med=23.74ms max=477.71ms p(90)=41.75ms p(95)=50.75ms
    http_reqs......................: 87375   72.771541/s
    iteration_duration.............: avg=1.03s   min=1s med=1.03s   max=1.47s    p(90)=1.04s   p(95)=1.05s
    iterations.....................: 87375   72.771541/s
    vus............................: 1       min=1       max=100
    vus_max........................: 100     min=100     max=100
```

As we can see from picture above. The code are able to handle 100 users easily in my local instance (ryzen 7 5700u, 8gb ddr4) while also running other task such as sonarqube container, visual studio code, the k6 instance it self, and many more. This prove that we may simulate real world usage in our QA workflow.

Documentation done by implementing openapi (swagger) so other department can look through api documentation in almost real time.



Last but not least I also attached dockerfile so the code can be easily deployed and scaled according to need.