Test Automation Engineer Coding Challenge

Candidates are expected to design and implement a system integration test for a RESTful API. Estimated time required to complete this task is 4 hours.

Task Description

The 8x8 product team has requested a RESTful API to enable our clients to send messages to their end users. User story for the feature is as follows.

- Our client makes an HTTP request to our API specifying the sender, recipient and the text message content.
- API sends a success response (HTTP status code 200) with a unique message identifier and the status of the message as "accepted" to our client. The unique message identifier is a GUID formatted as a string without the hyphens. You can generate sample GUIDs here.
- API forwards the message request containing the sender, recipient and the text to a message carrier's RESTful API.
- When the carrier responds with success status, our API inserts a record to a SQL database with the unique identifier of the message we sent to our client and the status "sent"

We have already implemented the API and the carrier for you. Define and implement an integration test to ensure that our API behaves according to the requirements of the product team. Ignore error behaviors and edge cases and only test the API for the success scenario.

You have full access to the API configuration (you can change the settings as you like in appsettings.json file) and the SQL database used by the API. Feel free to use them when implementing your tests. However, you should not change the settings for the carrier API (except if you really need to change the port because the port 8090 is already taken).

API Contracts

When a client wants to send a message using our API, it should make a HTTP request like the one below.

```
curl --location --request POST 'http://localhost:8080/api/messages' \
--header 'Content-Type: application/json' \
--data-raw '{
    "sender": "1234",
    "recipient": "5678",
    "message": "hello, world!"
}'
```

for which our API responds with HTTP status code 200 and the following response body of which the id is a random string from a GUID.

```
{
    "id": "8e083d487932411eb143903b5938a9e7",
    "status": "accepted"
}
```

Our API makes HTTP requests to the carrier API for the same request above like the one below.

```
curl --location --request POST 'http://localhost:8090/api/carrier' \
--header 'Content-Type: application/json' \
--data-raw '{
    "sender": "1234",
    "recipient": "5678",
    "message": "hello, world!"
}'
```

for which the carrier API responds with HTTP status code 200 with an empty response body.

Running the APIs

The task contains 3 archives: project.zip, app.zip and carrier.zip. project.zip contains the source files for the APIs as well as build definitions. app.zip and carrier.zip contain pre-built binaries for our API and the carrier API respectively for Windows, MacOS and Unix systems with support for dotnet.

You can run the APIs using docker compose, docker or using dotnet runtime without using containers. We recommend docker compose. You will need an internet connection when building the projects to pull the docker image for SQL Server Express from Docker Hub.

Run using docker compose

1. Make sure you have docker engine and docker compose installed (latest installations of docker comes with docker compose).
2. Extract the contents of project.zip.
3. Navigate to the Application directory in the project folder.
4. If you like to change any application settings for our API, navigate to WebApi directory and update the appsettings.json file. By default, the application is configured to listen on port 8080. If you are not familiar with aspnet core configuration, the appsettings.json file contains a database connection string which points to the SQL Express Server (which is bundled in the docker compose build, so you don't need to worry about this) and the carrier endpoint. Those are the only settings you need to worry about, if at all.

5. Run command `docker-compose up` from Application directory. This will bring up the carrier API, SQL Express Server and the web API.

Remark: Sometimes, the application may fail to start up on the first run if the database takes too long to initialise. If you run into an error like the web API exiting with a non-zero exit code or if the api responds with 500 http status code, stop the containers and rerun docker-compose up command. On the second run the start up is much faster and should run successfully.

Run using docker without docker compose

1. If you prefer to run docker containers without docker compose, make sure you have docker engine installed.
2. Run Microsoft SQL Server Express using the command `docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=ggqHN2HBXYLpq1y76cbs' -e 'MSSQL_PID=Express' -p 1433:1433 mcr.microsoft.com/mssql/server:2017-latest-ubuntu`
3. Extract project.zip and navigate to the Application folder. Build the carrier API using the command `docker build . -f ./CarrierApi/Dockerfile -t carrier:1.0.0`
4. Run the carrier application using the command `docker run -d -p 8090:80 carrier:1.0.0`
5. If you change the port of the carrier API or connection parameters to the database, you must update the appsettings.json in the WebApi project in the Application directory.
6. Build the web API using the command `docker build . -f ./WebApi/Dockerfile -t app:1.0.0`
7. Run the API using the command `docker run -d -p 8090:80 app:1.0.0`

Run without using docker or docker compose

1. Install ASP.NET Core runtime for your platform from the downloads. You only need the runtime as we have already compiled the applications for you.
2. Install and run Microsoft SQL Server Express from the downloads.
3. Extract the contents of carrier.zip, navigate to the extracted folder and run the command `dotnet CarrierApi.dll --urls="http://localhost:8090"` to run the carrier API.
4. Extract the contents of app.zip, navigate to the extracted folder and change all the `host.docker.internal` in the appsettings.json to `localhost`.
5. Run the command `dotnet WebApi.dll --urls="http://localhost:8080"` to run the API. If you want to change any other application settings, update the appsettings.json in the folder before running the application.

You can test the API using the curl request below:

```
curl --location --request POST 'http://localhost:8080/api/messages' \
--header 'Content-Type: application/json' \
--data-raw '{
    "sender": "1234",
    "recipient": "5678",
    "message": "hello, world!"
}'
```

If everything went well, you should get the HTTP response code 200.

Guidelines

As you are working through the task, keep in mind the following guidelines to assist you.

- You can use any technology stack and tools you are familiar with.
- We like integration test code to be clean, simple and maintainable. We hold tests to the same coding standards as our production software.
- Integration test suites must be extendable and scalable as we need to add more test cases or completely new features to the same API. Take this into consideration even though you only need to develop one test for this task.
- We like effective use of existing, industry standard tools. Don't reinvent the wheel unless your wheel is going to be significantly better.
- We have a soft spot for BDD tests because they are easier for the product teams and non-technical stakeholders to understand and validate but we are open to better approaches.
- Include a document explaining how to build and run your tests and how to view the test results.