Technical Assessment : Interview Questions & Answers

Introduction

Technical assessment is a simple login page which you should test with automation using Robot Framework for UI and Python for API.

The demo app can be found from: https://github.com/Interview-demoapp/Flasky

Basic instructions can be found from github page. Please create tests against the acceptance criteria.

The following is needed:

Q1) Write UI automation with Robot Framework?

A1) UI automation test suites can be found here. You can run the UI automation tests via run command. The instructions on how to install the virtual environment and run the tests are provided in README.md

Q2) Write API automation – requests with Python (we know there are RF libraries for APIs, but we want to see your Python skills)?

A2) API automation test suites can be found here. I implemented a custom library called CRUD_Library.py for RF. You can run the API automation tests via run command. The instructions on how to install the virtual environment and run the tests are provided in README.md

Q3) When using Python also write unit tests for Python implementation (=test the test code)

A3) The given time was not enough to implement unit tests for CRUD_Library.py. So, this task is left undone. But main idea is to use unittest python module to create unit tests. And use mockito to mock all the dependencies of CRUD_library, such as requests & logger. Then use coverage python module to get the coverage report for the unit tests. For another demo project, a year ago I implemented a similar “test the test code” task, here is the code for it.

Q4) Explain your approach to the implementation

A4) First of all, for the sake of clarity, I call the given UI & API acceptance criteria s as epics. They are epics, because they state high level goals in which we can define many concrete use cases. For example, if we break down the epic “If authenticated I can update personal information of users”, you will see many use cases as stated in Table 6 of RequiremensVsTestResults.pdf.

I also noticed that Login as Epic is missing, so my Epic list was the following:

For WebUI:

- 1-Registration.robot
- 2-Login.robot
- 3-Review-Own-Info.robot

For API:

- 1-Reviewing-Users.robot
- 2-Getting-Personal-Information.robot
- 3-Updating-Personal-Information.robot

Note that for each EPIC, I decided to have own test suite. After that I asked myself, which epic should I start to focusing? Referring to Flasky documentation, I decided to start with registration epic, because:

- You can't login unless you are registered
- You can't request API unless you are registered & have a valid API key (*)

So, I focused on the registration form on Web UI. I did some exploratory testing to understand the design principles and checked Flasky's code on how (incorrectly) registration is handled.
Then, I decided that requirements specification for the registration form is missing. So, I wrote one as stated in Section 2 of RequiremensVsTestResults.pdf. I sent the spec to Signant Health in mail to get green light. Once I got the green light, I decided to create test data for each field in the registration form:

- Usernames.json
- Passwords.json
- Firstnames.json
- Lastnames.json
- PhoneNumbers.json

So, in short, first I clarified the requirements for registration form and then I created test data based on that specification.

I specified that all 5 fields in the registration form are mandatory. So, I needed to implement my first case, which fetches randomly username, password, first name, last name and phone number and fills in the form and expects a success / failure based on the fetched values' validity. So, when I thought about the design of test architecture to achieve this objective, I have decided to implement in TestProject/CustomLibs folder the following:

- RegistrationFormDataUtils.py : implements generators combining the fields to create registration form data
- RegistrationFormDataReader.py : a custom RF library utilizing the generators exposing keywords to DataManager, which itself is a thin wrapper for test suites.

Basically, the test suite's for the epics (e.g. 1-Registration.robot) utilizes DataManager to read a **registration_form_data**, an example of which is listed as follows:


```
{
   "username": {
      "value": "# 123abc",
      "isValid": false,
      "expected_error": "Minimum 8 characters required. Space character is not allowed"
    },
   "password": {
      "value": "ABC!?.abABC!?.abABC!?.",
      "isValid": false,
```

```
      "expected_error": "Min 8 characters. Must contain at least one character from [a-z], [A-Z], [0-9] and [!?.]"
    },
  "first_name": {
      "value": "Helena!.?",
      "isValid": false,
      "expected_error": "Each first name must contain only characters from the set [a-zA-Z]. First names must be seperated by a a single space. First names must have at least 2 characters"
  },
  "last_name": {
      "value": "W Xi",
      "isValid": false,
      "expected_error": "Each last name must contain only characters from the set [a-zA-Z]. Last names must be seperated by a a single space. Last names must have at least 2 characters"
  },
  "phone_number": {
      "value": "",
      "isValid": false,
      "expected_error": "Please fill out this field."
  }
}
```


As you see, **registration_form_data** is a dictionary containing several **field_data** items, each of which is a dictionary themselves. Each field_data (e.g. last_name) contains what “value” (e.g. “W Xi”) should be written to the form and if it has invalid data (e.g. "isValid": false) then the “expected_error” we should see while taking into account **Figure 1** of RequiremensVsTestResults.pdf.

Now that I implemented DataManager and its dependencies, I managed to receive **registration_form_data** instances each having its **field_data** items randomly picked from:

- Usernames.json
- Passwords.json
- Firstnames.json
- Lastnames.json
- PhoneNumbers.json

This has been the most challenging part of the project. Once that is ready, I implemented my first test case “Registering With Variety Of Registration Form Data”. It was failing (as expected), taking into account **Figure 1** of RequiremensVsTestResults.pdf and the **registration_form_data** instances read in each iteration from DataManager.

So far, I implemented only 1 test case, proving that the Signant approved requirements specs (i.e. **Figure 1** of RequiremensVsTestResults.pdf) was not working.

Referring to the user stories in Table 1, Table 2, Table 3, Table 4 and Table 5 of RequiremensVsTestResults.pdf, I then implemented the test cases for each user story.

Having implemented 1-Registration.robot (the first & the most important epic), I then easily implemented the remaining UI epics

- 2-Login.robot

- 3-Review-Own-Info.robot

Once I was done with UI tests, I tagged them acc.to the RequiremensVsTestResults.pdf, Section 2, Tables 1-5. And I re-run the tests to see the status per UI epic & UI user story.

Once I was done with UI tests, I turned my attention to API epics:

For API:

- 1-Reviewing-Users.robot
- 2-Getting-Personal-Information.robot
- 3-Updating-Personal-Information.robot

There was one common issue that all these epics share. They all needed users with tokens in the system database file and system must be started with that database file (which I will talk about more in test-ability question **Q8** later). So, how do get users with valid tokens into the system?

Luckily, there was a solution. For 2-Login.robot epic's tests, I had implemented the keyword Re-Start Web Application With Many Registered Users. This keyword was already doing the registration of 3 valid users for me. So, I already had a database file with 3 valid users. They were lacking the tokens each, so I followed the instructions to create a token for each manually:

```
(.venv) (base) hakan@hakan-VirtualBox:~/Python/Robot/CRF_Demo_Solution/TestProject$ curl -u SuperDuperUser1
http://localhost:8080/api/auth/token
Enter host password for user 'SuperDuperUser1':
{"status":"SUCCESS","token":"MTczNzAyNjc5NDg4Mzc1MzcwNDIwODAyOTM1NTExMjU4NzkzNDM2"}

(.venv) (base) hakan@hakan-VirtualBox:~/Python/Robot/CRF_Demo_Solution/TestProject$ curl -u SuperDuperUser2
http://localhost:8080/api/auth/token
Enter host password for user 'SuperDuperUser2':
{"status":"SUCCESS","token":"MzI2MTg2NjIyMDY0OTk3NTc1ODcxMjEwNDQ2OTUxMTUzMTQxOTY4"}

(.venv) (base) hakan@hakan-VirtualBox:~/Python/Robot/CRF_Demo_Solution/TestProject$ curl -u SuperDuperUser3
http://localhost:8080/api/auth/token
Enter host password for user 'SuperDuperUser3':
{"status":"SUCCESS","token":"MTgxMjcxNDM1OTIzNTUzNjMyMzIxNTgzNjY0MTg1MzU4MTg0MTA3"}
```

Volaa! I had the **test database file** with 3 users with valid tokens. I stored it to TestProject/TestData/ demo_app_with_users.sqlite. Lets take a pause on this one, and re-focus the API epics.

For API:

- 1-Reviewing-Users.robot
- 2-Getting-Personal-Information.robot
- 3-Updating-Personal-Information.robot

All these epics have test cases, which are to make GET/PUT requests to the API. With each request API returns is supposed to manipulate the database content or not. So, we need to verify (using RF's Database Library) that the database content is indeed modified the right way or kept intact. So I implemented DbManager, which enabled my test suites to access the Flasky's local database in real time. So, I could locally do the following:

- Kill all Flasky processes if any
- Copy **test database file** into Flasky/instance
- Start Flasky only with `flask run --host=0.0.0.0 --port=8080`

And I can repeat the above process in a keyword as many times needed. This became especially handy in Test Setup phase. Now, before each API test call, Flasky was ready with 3 users with tokens (**). This made the API epics become testable; I can make the API calls with/without tokens and verify that database has/has not changed.

Note also that DataManager was already designed to return the users in the **test database file;** DataManager will read ManyValidUsers.json one user at a time into a **registration_form_data** instance. This capability is proven useful in API test suites later (for example, in this test case)

While (**) is great news for locally running Flask,what happens if Flasky was running on a remote server?? (which I will talk about more in test-ability question soon).

Now, I was ready to implement the test cases. I managed to implement

- 1-Reviewing-Users.robot
- 2-Getting-Personal-Information.robot

They were easy, because we were reading users/personal information from database and verifying that they match with what we received from API.

But for the third one:

- 3-Updating-Personal-Information.robot

I needed to specify how API should work, when it received the following

```
/api/users/{username}          PUT    Token
```

So, I specified Section 3 (Figure 2) of RequiremensVsTestResults.pdf,which then allowed me to breakdown the epic into smaller use cases. For each use case, it was easier to implement test case(s) and or do exploratory testing with POSTMAN. (BTW, you can find my POSTMAN collection for the project here). Once the test cases were ready, I tagged them according to Table 6 of RequiremensVsTestResults.pdf.

Now all API test cases were ready as well. Then using the run command, I re-run all Web UI test suites against chromium, firefox and webkit browsers. With the same run command, I ran also the API tests. The results are stored under Results/ folder respectively.

Finally, if I do a retrospective on the overall implementation procedure (***):

1. Define the UI/API specification
2. Implement test data based on (1)
3. Repeat (1) and (2), until test data covers all possible use cases
4. Implement code utilizing test data/database
5. Implement test cases based on (1)
6. Repeat (1)-(5) in iterations if needed

Q5) How do you review code?

A5) If the code I am reviewing is the System Under Test (SUT)'s code, then I take one use case at a time, I first define/refer to the specs of the use case having flowcharts (as in Figure 1 & Figure 2 of RequiremensVsTestResults.pdf). While doing so, I also review the SUT's code for the use case in question to see if there are any test cases and functionality I missed. In this process, I may update a the requirements' spec. In short, I mostly I go use case by use case and analyze relevant functionality in the SUT's code.

If the code I am reviewing is the test code itself, then I repeat a similar process taking into account one use case at a time, but this time I utilize system diagram's (as in Figure 1 & Figure 2 of RequiremensVsTestResults.pdf) to see if the existing test cases cover all the functionality.

Q6) How do you enforce coding standards?

In general (but not for this demo), I use automated tools such as robotframework-robocop and robotframework-tidy tools to enforce the coding standards. Also peer to peer reviews can be utilized.

Q7) How do you plan what kind of approach you take for test automation - what libraries to use, how does it work in couple of years, how to make it easy to maintain, etc? What are the main points to consider?

A7) I usually do a research on StackOverflow etc for an answer to solve a particular problem. Let's say that the thread there said to use particular library X to solve my problem. Then I check the project page in GitHub as well as the pip repository for the following questions for Project X:

- Does it have an active community?
  - When was the last time an issue was fixed?
  - When was the last time when a new release was made?
  - What does the release history look like?
  - What the current release plan?
- Does it have a stable & tested code base?
  - How many open issues does currently have?
  - How many critical issues does it currently have?
  - Does it have a test code that is running green?
- Developer community characteristics
  - How many contributors are there?
  - How many of them have been in the last 3 months?

One can create an excel checklist to collect answers for the above questions. If must of them are green, then we can consider using library X into use. On top of that I try to use the latest version of the libraries when possible.

Q8) Code test-ability, how do you enforce it? How do you make sure that the product is testable?

A8) Considering FlaskyIn **A4** (**), I explained how to make Flasky start with **test database file** having 3 users with tokens. However, when Flasky runs on its own web server, what then? We need a similar mechanism, where can start Flasky remotely with the **test database file.** I am not sure how to achive this. My guess is to use docker container having the Flasky with the test database file & some script/tool to (re-)start the Flasky remotely. I might be wrong here.

If I was to provide a more general answer to this question, then I would check this blog. I am not fully competent on the subject yet.

Q9) Provide the following back as freely distributable git repository.
A9) It can be found here. It is freely distributable with Apache 2.0 licence.

Q10) Report of executed tests
A10) Robot FW's Test Logs can be found here

Q11) Report of found issues/bugs
A11) Please refer to RequirementsVsTestResults.pdf

Q12) Tell us what improvement would you propose for the app?
A12) Regarding the API's epic (3-Updating-Personal-Information.robot), I wonder is it really intended for user X to be able to update firstname, lastname and phone of another user Y ? Should updating personal information be restricted to one's own firstname, lastname and phone only?

Q13) If you would be given a week to do quality assurance for this product, briefly plan the tasks based on your skills, knowledge and expertise
A13) Referring to the development process I outlined in **A4 (***),** here are the tasks:

1. Define the UI/API specification
2. Implement test data based on (1)
3. Repeat (1) and (2), until test data covers all possible use cases
4. Implement code utilizing test data/database
5. Implement test cases based on (1)
6. Repeat (1)-(5) in iterations if needed

1 week would not be enough. I spent a total of 3.5 weeks with weekends, and I can still implement more test cases.

Q14) Instructions how to run it and short description of components, including external libraries
Q14) README.md at the root of the project describes all the required aspects

Q15) Description about taken approach and potential gaps in application
A15) This question is not understood.

Q16) How much time it took, there is no time limit as such.
A16) Here is a time report. Altogether, I spent 155 hours.