DIT636 / DAT560 -
Assignment 2: Test Design and API Testing

Due Date: Sunday, February 16th, 23:59 (PDF, Via Canvas)

There are two questions worth a total of 100 points. You may discuss these problems in your teams and turn in a single submission for the team (zipped archive) on Canvas. Answers must be original and not copied from online sources.

Cover Page: On the cover page of your assignment, include the name of the course, the date, your group name, and a list of your group members.

Peer Evaluation: All students must also submit a peer evaluation form. This is a separate, individual submission on Canvas.

Problem 1 - Functional Test Design (60 Points)

In this problem, you will design abstract test specifications for a student management system. The primary purpose of this system is to check whether students are ready to graduate from a particular degree program.

This system has a REST API that surfaces the following functionality:

Route	Method	Example URL	Description
/student	GET	http://127.0.0.1:5000/student	Get a list of all students, with their IDs.
/student/{student_id}	GET	http://127.0.0.1:5000/student/6	Get data for a single student.
/create	POST	http://127.0.0.1:5000/create	Create a new student record in the database, if a record does not already exist for that personnummer.
/update/{student_id}	PUT	http://127.0.0.1:5000/update/2	Update a student record. Note that changes to personnummer are not allowed.
/delete/{student_id}	DELETE	http://127.0.0.1:5000/delete/5	Delete a student record.
/program	GET	http://127.0.0.1:5000/program	Get a list of all programs, with their IDs.
/program/{program_id}	GET	http://127.0.0.1:5000/program/1	Get a list of required courses for a particular program.
/finished/{student_id}/{program_id}	GET	http://127.0.0.1:5000/finished/1/1	Checks whether a particular student is ready to graduate from a particular program.

The POST and PUT methods require, as input, a JSON structure representing a student. The allowed records in this structure include:

- name (string)
- personnummer (string, format YYMMDD-NNNN)
- courses_passed (a list of courses, each an ID represented by a string of format “XXXNNN”, where “XXX” is a three letter department ID and NNN is a three number course ID).

The student records stored in the app also include the field:

- student_id (integer).

A student_id is not needed for the create method, as it is assigned by the system.

A degree program is represented by a list of courses, where - again - each has a course_id.

All input is validated by the system. If you provide invalid or malformed input - either in the endpoint URL or the JSON bodies for the create and update methods - you should expect an appropriate error.

Note that the POST/PUT/DELETE methods do not actually make permanent changes in this example. You will get an appropriate response, but the record will not actually be created, updated, or deleted. You can use the result body to verify the results of running these functions.

You can find the source code of this system at https://github.com/Greg4cr/dit636_examples/blob/main/src/as1-webapi/app.py.

To deploy the system locally:

- Check out the repository: https://github.com/Greg4cr/dit636_examples.git
- In a terminal:
  - Enter the directory src/as1-webapi/
  - Install the Python package flask: python -m pip install flask
  - Set the following environmental variables:
    - export FLASK_APP=app.py
    - export FLASK_ENV=development
    - (on Windows, “set” instead of “export”)
  - Start flask: flask run
- Once the system is deployed, you can interact with the system using curl, Postman, or other utilities that can send requests to the endpoints defined above.
- See the following tutorial for an example of how to deploy this type of application: https://realpython.com/api-integration-in-python/#rest-and-python-tools-of-the-trade

For each endpoint, except for /student/ and /program/, Identify the choices, representative values, and constraints that you would use to create test specifications.

- **Based on the input parameters or other environmental factors under your control, identify the choices you control when testing this endpoint.**
  - These are aspects of the execution of that endpoint that you control and can affect the outcome of executing the function at that endpoint.
    - (e.g., the student_id value used as input for /student/{student_id})
- **For each choice, identify representative input values.**
  - These are the options that you can select for that choice that could change the outcome of executing the function.
    - (e.g., “a valid student_id > 0 and < the total number of students” would be a representative value for the choice “value of student_id”)
- **For each representative value, if applicable, identify constraints**

- Constraints: IF, ERROR, SINGLE
- Constraints limit the combinations of representative values that will be tried when testing that endpoint.
    - IF states that Representative Value A for Choice X can only be selected if Representative Value B is chosen for Choice Y.
    - ERROR indicates that, if this representative value is chosen, an error is expected from the function.
    - SINGLE indicates that the chosen representative value should result in a normal outcome of the function, but it should be tried one time because it is an unusual value.
- (e.g., a representative value “student_id < 0” for the choice “student_id value” would receive an [ERROR] constraint because a negative student_id is invalid, regardless of the values of any other choices made for that function being tested)

You should base your test specifications on the functional description above, rather than on the source code. The actual source code may not fully or correctly implement the functionality. Your tests should be based on the intended functionality (as described above), rather than on the implementation.

See page 187 in the Software Testing and Analysis textbook for an example solution to a similar problem. See Exercise Session 2 for another example of this process.

Note that you do not need to create the full list of test specifications for this problem, just identify the choices, representative values, and constraints.

Problem 2 - API Testing (with Postman) (40 Points)

Based on your work in the previous problem, you will now develop a set of concrete test cases that can be executed using the Postman tool for testing the system through its REST API.

- **If you do not have one already, create a free account at https://www.postman.com/ and download the Postman desktop agent.**
- **Deploy the system locally, following the instructions in Problem 2.**
- **Open the Postman desktop agent.**
- **Use Postman to create tests for the system from problem 1, based on your choices, values, and constraints.**
    - **Create at least 12 test cases, with at least one test case for each API function tested in Problem 2.**
    - **Each test case has its own input (URL + request body) and one or more assertions on the output (called "tests" in Postman). Do not simply submit 12 assertions!**
    - **Your set of test cases should test both normal functionality as well as handling of erroneous input.**
- **In your report, include the request type, URL, body, assertions ("tests"), and any other information that you used as part of your test cases. Please explain each test case - describe the goal/purpose, as well as the assertions you used to verify the behavior.**
    - **You are welcome to submit the .postman_collection file as well (as a separate zip, in addition to the PDF), but it is not required as long as the tests are shown and explained in your report.**

Many of you have learned some basics about REST and testing of APIs in DIT341, but please talk to us if you have questions. For a starting place on testing in Postman, see https://learning.postman.com/docs/writing-scripts/test-scripts/

Note: The test cases do not have to pass! The tests should reflect how you think the system *should* work, so if they fail, that indicates the system is faulty (in your view).