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

Due Date: Sunday, February 15th, 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 library providing the core logic underlying a videogame. In this game, the survivors of a zombie apocalypse are attempting to enter bunkers to shelter from the zombie hoards. Whether they can enter a bunker depends on their health status and their equipment.

This logic library has a REST API that surfaces the following functionality:

Route	Method	Example URL	Description
/survivor	GET	http://127.0.0.1:5000/survivor	Get a list of all survivors, with their IDs.
/survivor/{id}	GET	http://127.0.0.1:5000/survivor/4	Get data for a single survivor.
/register_survivor	POST	http://127.0.0.1:5000/register_survivor	Create a new survivor in the database, if a record does not already exist for the field supervisor_id.
/update_survivor/{id}	PUT	http://127.0.0.1:5000/update_survivor/4	Update a survivor record. Note that changes to the field survivor_id are not allowed.
/kill_survivor/{id}	DELETE	http://127.0.0.1:5000/kill_survivor/5	Kill a survivor, removing their record from the database.
/bunker	GET	http://127.0.0.1:5000/bunker	Get a list of all bunkers, with their IDs.

/bunker/{bunker_id}	GET	http://127.0.0.1:5000/bunker/3	Get a list of required supplies to enter a particular bunker.
/can_enter/{id}/{bunker_id}	GET	http://127.0.0.1:5000/can_enter/2/1	Checks whether a particular survivor is allowed to enter a particular bunker.

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

- name (string) - The name of the survivor. Note that multiple survivors are allowed to have the same name.
- survivor_id (string, format YYMMDD-NNNN) - A unique permanent identifier representing a particular survivor, based on the Swedish personnummer format. Allows the same character to persist across multiple instances of the game.
- infection_rate (integer, range of [0, 100]) - The extent to which a survivor has been infected by the zombie virus.
- inventory (a list of supplies, each referred to using an ID formatted as a string “XXX-NNN”, where “XXX” is a three-letter item code and NNN is a three-number code) - Represents the current supplies carried by that survivor.

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

- id (integer) - Represents a unique in-game identifier for the character. Specific to this instance of the game, in contrast to the survivor_id, which persists across instances.

An id is not needed as input for the register_survivor method, as it is assigned by the system for this instance of the game.

For example: **{“id”: 12, "name": "Leon Kennedy", "survivor_id": "830914-1234", "infection_rate": 0, "inventory": ["WPN-001", "MED-101"]}**

A bunker is represented by:

- id (integer) - A unique in-game identifier for the bunker.
- name (string) - The name of the bunker.
- required supplies (a list where - again - each item has an item ID).

For example: **{"id": 1, "name": "Hilltop", "supplies_required": ["WPN-001", "MED-101", "FOD-005"]}**

To be granted entry, a survivor must possess every item listed in the bunker's requirements. Bunkers also have specific health regulations regarding infection rates - the “Hospital” bunker allows entry for survivors with an infection rate below 50, whereas other bunkers strictly require a rate of 0.

All input should be 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 simplified 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/EsmeYi/dit636_examples/blob/main/as1-zombiesurvival/app.py

To deploy the system locally:

- Check out the repository: https://github.com/EsmeYi/dit636_examples.git
- In a terminal:
    - Enter the directory src/as1-zombiesurvival/
    - 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 the GET requests to /survivor/ and /bunker/, 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 id value used as input for /survivor/{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 id > 0 and < the total number of survivors” would be a representative value for the choice “value of 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.
        - An IF constraint can be used to state that Representative Value A for Choice X can only be selected if Representative Value B is chosen for Choice Y.
        - An ERROR constraint indicates that, if this representative value is chosen, an error is expected from the function.
        - A SINGLE constraint 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 “id < 0” for the choice “id value” would receive an [ERROR] constraint because a negative id should lead to an error message (or some other special program outcome) regardless of the values of any other choices made for the /survivor/{id} endpoint)

You should base your test specifications on the intended functional described above, rather than on the source code. The actual source code may not fully or correctly implement the functionality.

Your test specifications should aim to thoroughly test the behavior of each endpoint. They should cover different outcomes of the functionality, different ways of achieving each outcome, and handling of malformed, invalid, or otherwise erroneous input.

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 1.**
- **Open the Postman desktop agent.**
- **Use Postman to create tests for the system from problem 1, based on your choices, values, and constraints from answering that problem.**
    - **Create at least 12 test cases, with at least one test case for each API function tested in Problem 1.**

    - **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.**
- **Explain each test case - describe the goal/purpose, as well as the assertions you used to verify the behavior. Explain how the test case maps back to Problem 1.**
- **You should also submit the .postman_collection file as well so that we can re-execute your test cases.**

Many of you have learned some basics about REST and testing of APIs in Web Development, 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).