Homework Assignment #6

Estimated time: 240 – 360 minutes

Objectives

- Gain some familiarity setting up environments for cloud developing using an AWS.
- Start to become familiar with at least one AWS SDK.
- Gain some familiarity with the AWS Command-Line Interface.

Overview

In this assignment, you will write a program similar to the Consumer program used in a previous homework in CS5270. Specifically, this Consumer program will read objects (Widget Requests) from an S3 bucket (namely, Bucket 2) and then process those requests. Each request specifies results in a single Widget creation, update, or deletion in either another S3 bucket (Bucket 3) or in a DynamoDB table. You may write this program using any language and development stack that you choose, as long as the language supports an AWS SDK. You may even use a remote VS Code server from HW5, if you wish. The instructor will provide an executable Producer program that will allow you to do some system testing.

Instructions

Step 0 – Preliminaries

Be sure that your AWS workspace includes the VPC, EC2 instances, S3 buckets, and DynamoDB table described in previous assignments. If you are unsure of the state of these resources,

- Test your connectivity to the EC2 instances using SSH and your key pair,
- Test your Bucket 3 website, and
- Test your DynamoDB table by adding some widgets to it manually through the DynamoDB console.
- Test everything by running the instructor provided Producer on Host A and Consumer on Host B.

Step 1 – Design your Consumer program

As you should remember, the Consumer program processes Widget Requests to create, update, or delete Widgets. Your Consumer program should be able to store the widgets either in the Bucket 3 or in the widgets DynamoDB table.

Your Consumer program needs to allow the user to specify command-line arguments for the storage strategy and resources to use, like the instructor provided Consumer program. The syntax for the arguments is up to you. Look for a good open-source 3rd-party library that can help you the implement command-line arguments with minimal coding effort.
You Consumer program will need to periodically try to a read a single read Widget Requests from Bucket 2 in key order (e.g., smallest key first). Don't try to read all existing requests are one time for two reasons. First, the Producer program may be running concurrently and adding to the set of requests. Second, in later homework assignments, you will be extending this system so multiple Consumer programs can run at the same time without getting in each other's way. If read only one Widget Request at a time, your application will be more scalable than if you tried to read them all at once. Third, repeatedly retrieving a list of all keys in an S3 bucket may not as efficiently as retrieving a small number of keys (like just one) at one time.

If the Consumer succeeds in reading a request, then it should delete the request, process the request, and then immediate go back to trying to read another request.

If there is no available request, then the Consumer should wait for a little while (e.g., 100ms) before trying again. In other words, the reads should be done a typical polling loop:

```
Loop until some stop condition met
      Try to get request If
      got request
            Process request
      Else
            Wait a while (100ms)
End loop
```

The Widget Requests are in a JSON format according to the schema given in Appendix A. This schema is available as a download with the assignment if needed. Some sample Widget Requests are also available for download with the assignment. Note: Do not try to write your own JSON parser. Please find an appropriate library that can parse JSON text into objects, if your development stack does not already include one. There are multiple open-source JSON parses for most common languages.

Requests come in three flavors (create, update, and delete):

Widget Create Request
When the Consumer processes a Widget Create Request, it will create the specified widget and store it in either Bucket 3 or the DynamoDB table.

Widget Delete Request
When the Consumer processes a Widget Delete Request, it needs to make sure that the specified object does not exist. If it does not currently exist, the Consumer should **not** throw an error. Instead, it should simply log a warning and move on to the next request.

Note: Although, you need to consider handling delete requests in your design, you do not need implement this requirement until HW7.

Widget Change Request
When the Consumer processes a Widget Update Request, it will first retrieve the specified widget, and then change all the attributes mentioned in the request. If a property is not included in an update request or if its value is null, its current value should not be change. If value of a property in an update request is the empty string, that means the corresponding property of the widget should be set to null or deleted (if it is one of the other attributes). A widget's id and owner cannot be changed. If the specified widget does not exit, it should **not** throw an error. Instead, it should simply log a warning.

Note: Although, you need to consider handling change requests in your design, you do not need implement this requirement until HW7.

As mentioned in above, your HW6 implementation only needs to process Widget Create Requests. However, your design should anticipate the handling of Widget Delete and Update Request in the near future.

A widget needs to contain all the data found in a Widget Create Request. When a Widget needs to be stored in Bucket 3, you should serialize it into a JSON string and store that string data. Its key should be based on the following pattern:

widgets/{owner}/{widget id}

where {owner} is derived from the widget's owner and {widget id} is derived from the widget's id. The {owner} part of the key should be computed from the Owner property by 1) replacing spaces with dashes and converting the whole string to lower case.

When a widget needs to be stored in the DynamoDB table, place every widget attribute in the request its own attribute in the DynamoDB object. In other words, in addition to the *widget_id*, *owner*, *label*, and *description*, all the properties listed in the *otherAttributes* properties need to be stored as attributes in the DynamoDB object and not as single map or list.

In a future homework assignment, requests will come from a source other than an S3 bucket, like a queue. Therefore, design your Consumer so the logic for retrieving requests can be easily swapped out at runtime with a different algorithm. At a minimum, encapsulate the request retrieve logic into a method or function, but try to go a step in ensuring low couple, high cohesion, and good modularization.

Step 5 – Implement components of the Consumer program with test cases

In this step, you need to implement your design and test each component (e.g., class) using executable unit test cases.

It is important to note that testing is not an optional part of this assignment. It is a critical part of any software engineering project industry and should be part of any software programming assignment in academy. In fact, the department establish this a goal in response by the Industry Advisory Committee's overwhelming request to do so.

If you are not familiar with unit tests and the testing tools for your chosen environment, you may need to do some extra reading on this subject. **See the lecture about "Unit Testing" in this module**. TA and instructor can provide further help in this area if needed.

If you find that method or function is hard to test, then it is very likely that your method or function is trying to do too much. Break it up into smaller methods or functions so each one focus on doing one thing. This is consistent with the principles of good modularization or the "Single Responsibility Principle" of the SOLID.

Finally, your Consumer should also produce a log file that records what is does. There are many open- source logging libraries available for every popular programming language. Choose a good one and use it. Don't try to do all the log by brute-force file I/O.

Step 6 – Complete some system testing

Once you have completed your implementation and unit testing, do some ad hoc system testing by running the instructor-provided Producer with your Consumer. You may run the Producer on Host A or on your own machine. If you have setup the AWS credential file for AWS CLI properly, this should work just fine because the Producer will use this file when it tries to create an S3 client or DynamoDB client. To see, the possible parameters and examples, execute the following:

```
java -jar producer.jar --help
```

Similarly, you can run your Consumer on Host B or on your own machine. During debugging, it will be considerably easier to run it on your own machine, and probably directly in your IDE.

Keep logs from Producer and Consumer for at least one successful run and submit them as part of the homework assignment.

Step 6 – Commit your work to a git repository

Manage your all project's artifacts with Git, including project files, build instructions, etc. The only things you don't need to keep in the Git repository are artifacts that are generated during the build process or at runtime (i.e., the log files, request, and Widgets). Commit to your Git repository frequently. Your Git commit log will be examined during grading and must show meaningful workflow. A single commit when you are all finished is not acceptable.

Also, you will need to take a snapshot of your Git repository's commit log and submit it with your assignment.

Hints and Other Thoughts

Review how to create executable test cases. If you are uncertain of how to set up and program executable test cases in your chosen language for this assignment, do some searching online for testing tutorials. The "Unit Testing Examples" in this module provide some examples in Python. Virtually every modern programming language has at least one readily available testing framework and harness. The framework is a library of macros, objects, or classes that allow you to define test cases and mocks (if needed), and to compare expected results with observed results. The harness allows you to execute the test cases and often includes other features like coverage reports. For some development stacks, the testing framework and harness are combined into one tool.

Make sure the executable test cases are meaningful. Each test case should adhere to the following pattern:

1. Set up an initial state, if needed
2. Stimulate the thing to be tested (i.e., run the method or function with the desired parameters
3. Compare expected results to predicted results. Be sure to include comparisons for all relevant portions of the resultant state.

Not doing sufficient meaningful comparisons is the primary reason for why test case might fail to provide real value.

Use Path and Input Validation Testing Techniques. Reasonable coverage of your code can be achieved by using a combination of Path Testing and Input Validation Testing. With Path Testing, each test case exercises a different path through a target method or function. With path testing, a thorough suite for a target method or function is one that ensures that every statement is executed at least once, every conditional statement is tried for each possible outcome, each loop boundary is checked, and the throwing of possible exceptions are exercised.

With Input Validation Testing, the input domain for a method or function (which can include the state of associated object or even the state of the whole system) is partitioned into meaningful subsets. Reasonably cover involves testing an example input from each subset. For example, to test a method that takes a "Long" integer as a parameter and does not rely on anything else, the input domain is the set of all long integers. Meaningful partitioning of this domain would be {{Positive Long integers}, {0}, {Negative Long integers}, {the Minimum Long Integer value}, {the Maximum Long Integer value}, and {null or undefined}}.

Keep it simple. Don't over design, i.e., do not invent requirements or try to anticipate future requirements except those that explicitly mentioned. Also, don't over implement, i.e., do not build components that have no purpose relative to your design. Use open-source software where possible, e.g., for logging.

Be flexible. If you get into the coding and testing, and discover a problem with your design, go back and fix it.

Don't cheat. Don't even think about decompiling and code any code that has been provide with this or any previous assignment. There are some idiosyncrasies that I will recognize. I don't expect your design to be the same as mine. Also, do not copy each other's designs or code. Do you own thing. That's the only way you will learn.

You may use publicly available design ideas (such as design patterns) and code snippets. However, if you include anything that is not your creation, you must give credit to the source; otherwise, it is considered plagiarism. Plagiarism and all forms of cheating will be severely penalized and reported.

Submission

Your submission must include the following:

- All screen snapshots mentioned in the above steps
- An archive file of your entire project (including hidden project files and build instructions, but not necessarily any compiled artifacts).
- The log files from at least one successful system test.

Appendix A

Below is the JSON schema for Widget Requests. Note that the owner property can be any string of upper- and lower-case letters and spaces.

```
{
  "$schema":  "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "type": {
      "type": "string",
      "pattern": "create|delete|update"
    },
    "requestId": {
      "type": "string"
    },
    "widgetId": {
      "type": "string"
    },
    "owner": {
      "type": "string",
      "pattern": "[A-Za-z ]+"
    },
    "label": {
      "type": "string"
    },
    "description": {
      "type": "string"
    },
    "otherAttributes": {
      "type": "array",
      "items": [
        {
          "type": "object",
          "properties": {
            "name": {
              "type": "string"
            },
            "value": {
              "type": "string"
            }
          },
          "required": [
            "name",
            "value"
          ]
        }
      ]
    }
  },
  "required": [
    "type",
    "requestId",
    "widgetId",
    "owner"
  ]
}
```