Assignment 04 - Application Server - Using a database.

Due 25th of Mar
Points: 400
This is a bigger assignment - don't leave it to the last minute. The assignment has been split into 3 parts. Part 1 and 2 are 400pts as Assignment 04.

Part 1. Setup - 100pts (Mar 11) - get existing code to run, connect to database. Use 'curl' or 'wget' to pull back data from the /api/v1/status API endpoint. Verity that you are talking to the database with /api/v1/get-state. Run the web pages and verity that they will paint.

Part 2. Mar 25 - 300pts - Create the tables in the model. Populate them. Look in ./a04-server.py (or for Go a04-goserver.go) and implement ALL of the functions that have comments mentioning "Assignment 04". Build automated test for the API that test each of these functions. Run the web pages and verify that the application is working.

Part 3. (Assignment 05 - Authentication / Encryption) Implement a login/security layer for this application (Due Next Month)

Storing data in a database for the sake of storing it in a database is not much fun. Let's actually use this set of database skills to build something.

Let's build an issue tracker. Issues are what you have when something is wrong in some set of software. They could be anything from miss spelling 'stofware' instead of 'software' to annoying things like 'it makes the computer crash' to really horrible things like 'it kills people'. These are the defects. There are commercial defect tracking systems like JIRA that are multi-million dollar a year businesses. We will build some features that JIRA is missing.

I will give you the front end - web pages and what they expect. Our issue tracker is simplified - it has no user accounts (Accounts and login stuff is actually very complicated). Also a real tracker would include file upload. File upload requires special work on the server side - so we will leave that out.

The nifty features that it will have.

1. Search based on words. This is taking Assignment 03 - Key word search and applying it.
2. Ability to have a dashboard of issues.
3. Ability to attache notes to issues.
4. A severity and a frequency of issues.

In this assignment you get to take the ERD for the model and implement the tables for it and then implement the server that uses SELECTs, UPDATEs, DELETEs and INSERTs to connect this model to an outside application.

The application server is in Python using the `bottle` package for handing web requests.

The ERD is in `./assignment-04.erd.pdf` . First take the ERD and build the tables, constraints and triggers for the model.

The sample server is in `./a04-server.py` . There will be a set of videos detailing adding an endpoint to the server and how that works.

First - the 1st line in the file `#!/usr/bin/python3` is for Mac and Linux. You man need to change that if you are running on Windows. On a Mac or Linux system a `#!` at the beginning of a script tells the shell how to run the script. The rest of the line is the path to an executable that this file will be passed to. In this case it is passed to `/usr/bin/python3` . You will note that the 2nd

line in the file is my install of Anaconda Python. Find the location of your python and change this to have the correct location for your python.

You can find the location of an excitable with

```
$ which python3
```

Second - Normally script files need to be executable (this is Mac and Linux again) to run. If you do `ls -l a04-server.py` and it has

```
-rwxr-xr-x  1 philip  staff  18042 Feb 20 19:53 a04-server.py
```

something with `-rwx` at the beginning then the executable bit is set. That is the `x`. If it has `-rw-` then it is not set as an executable. To set the `x` so that yo can run it you need to use the `chmod` program.

```
$ chmod +x a04-server.py
```

Now you need some environment variable set to be able to connect in the code. It is not a good idea to hard code usernames and passwords into programs. Very often this information is moved into environment variables. (Amazon Web Services, AWS, has all of its coding examples work this way!)

If you look in the file `./database.ini` there are configuration items to allow the python code to connect to your database.

```
[postgresql]
host=127.0.0.1
database=ENV$DATABASE
user=ENV$DBUSER
password=ENV$PGPASSWORD
```

These items are, `host=` - this is the IP address of the host. Under the assumption that you are running this on your Linux virtual machine then the IP address of the local machine, 127.0.0.1 is probably correct.

The other items start with an `ENV$`. The python code will see this and then use the 2nd part to pull this from the environment. In my case I set and export these values.

```
$ export DATABASE=philip
$ export DBUSER=philip
$ export PGPASSWORD=not-going-to-tell-you-this
```

In the shell you can set local variables with just `NAME=Value`. To make that into an environment variable you use `export`. This is true on Mac and Linux shells. On Windows you have to go into some system configuration and find it and set it. If you are doing this on windows let me know and we will walk through the ugly details.

Now you can test this by running the python code that reads in and creates the connection to the database.

```
$ python3 config.py
Connecting to the PostgreSQL database...
```

With the correct configuration of you environment variables it should connect.

When you get the Python code to connect to the database you should have a little celebration. This is an important step.

For Go look in ./cfg.json

The configuration file ./cfg.json is the config for the a04-goserver.go program. It is read in near the top of the program.

```
{
        "static_files":"./www",
        "host":"0.0.0.0",
        "port":"12138",
        "db_flags": [ "debug-log", "get-issue-detail.01"]
}
```

and the ./setup.sh script for environment variables - you will need to “source” this file in the shell to set the environment variable.

```
# urlExample := "postgres://username:password@localhost:5432/database_name"
export DATABASE_URL="postgres://philip:my_passwrod_is_not_in_this_file@127.0.0.1:5432/philip"
```

Edit it to have your database connection information.

Implementation

No comes the fun/hard part. The server is not complete. In a number of places it has

```
    return "{"+"\"status\":\"TODO\",\"n_rows\":0,\"data\":[]"+"}"
```

These need to be replace with actual code that implements the server.

You need to implement each of the following end points in the server.

Endpoint	Method	Description
/api/v1/issue-list	GET	use `run_select ( "SELECT * FROM i_issue_st_sv", {})`
		to select back the set of issues in the database
		that are not `Deleted` . Create the view i_issue_st_sv
		to join from i_issue to i_state and i_severity so that
		both the state_id and the state are returned. Sort the
		data into descending severity_id, and descending creation
		and update dates. The view i_issue_st_sv should be
		added to your data model that you turn in.

As soon as you have `/api/v1/issue-list` implemented use the test code to verify that it works.

First load the sample data, `./sample-issues.sql` into PostgreSQL. Using your cleanup script, `./delete-issues-note.sql` :

```
$ psql
philip=# \i delete-issue-note.sql
philip=# \i sample-issues.sql
```

With a correct model it should load without errors.

In one terminal window:

```
$ ./a04-server.py
```

In an 2nd terminal window:

```
$ python3 test_api.py /api/v1/status /api/v1/db-version /api/v1/issue-list
```

It should run and tell you if

1. The server is running,
2. The database is ok,
3. The API end point worked and returned correct data.

Endpoint	Method	Description
/api/v1/create-issue	GET, POST	
		perfom an insert into i_issue with parameters from the GET or POST.
		The paramters should require ‘body’ and ‘title’ but allow for
		defaults for “severity_id” and “issue_id” . These should default
		to ‘1’ for the first ID in the set of ids.
		For “issue_id” it should default to a new UUID if not specified.
		Use `run_insert` do to the insert and remember to commit the
		change to the database. The return from `run_insert` will
		have the status/success and ID to return to the client
		application.

Endpoint	Method	Description
/api/v1/delete-issue	GET, POST	
		Use a passed ‘issue_id’ to do a delete from it i_issue table.

Endpoint	Method	Description
/api/v1/update-issue	GET, POST	

Endpoint	Method	Description
		Use a passed 'issue_id' the primary key to update the body and or the
		title of an issue.

Endpoint	Method	Description
/api/v1/get-issue-detail	GET	
		The default issue get allows returning a list of issues. This will take
		the issue_id and return the same format for the data but for a single issue
		with all of the associated notes for this issue in order of when the notes
		where created. This is sorted by i_note.seq.

Endpoint	Method	Description
/api/v1/add-note-to-issue, POST	GET	
		Insert into i_note with issue_id so as to assocaite the issue with a note.

Endpoint	Method	Description
/api/v1/delete-note	GET, POST	
		Delete the specified note from the issue based on note_id and issue_id

Endpoint	Method	Description
/api/v1/update-severity	GET, POST	
		Update the severity_id in i_issue given the issue_id and the severity_id.

Turn In

1. Your data model. This is the .sql file with the create table statements in it that you built from the ERD.
2. Your modified version of a04-server.py with your implementation of the server in it.
3. Your data cleanup script.
4. Your tests in the test_api.py program for these endpoints: `/api/v1/add-note-to-issue`, `/api/v1/delete-note`, `/api/v1/update-severity`
5. Your automated test programs to verify the API.