Project

TPC-C

INSTRUCTIONS

- This project is an individual project.
- Submit a ZIP file called "`project3-[student_number].zip`" (for example: "`project3-A012345L.zip`") containing a file "`completestock.sql`" (containing the SQL code for Question 2) and a directory "`Project3`" (containing the Python code for the project with the three subdirectories with the code for the three apps of Question 4, Question 5 and Question 6, respectively).
- Do not submit other SQL and Django files.
- Submit the ZIP file to the "`Project 3: Submission`" folder under Luminus "`Files > Project 3: TPC-C`" by **Friday 27 September 2019, 17:00**.
- Past this deadline and before **Wednesday 2 October 2019, 17:00**, you may submit to the "`Project 3: Late Submission`" folder (penalties apply).

The Transaction Processing Performance Council (TPC) proposes the TPC-C benchmark (`www.tpc.org/tpcc`) to measure the performance of online transaction processing systems.

In the TPC-C business model, a wholesale parts (items) supplier operates out of a number of warehouses. TPC-C simulates a complete environment where a population of terminal operators executes transactions against a database. The benchmark is centred around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, checking the status of orders, and updating and monitoring the level of stock at the warehouses.

In this project we consider a simplified version of the TPC-C database. The simplified schema consists of the three tables `item`, `warehouse` and `stock`.

The objective of the project is to develop a Django Web application with a PostgreSQL database that implements a simple order and stock management system for terminal operators.

1. (0 points) Setting-up PostgreSQL Database

   Create a PostgreSQL database called "`bt5110_p3`".

   Download the SQL files "`TPCCSchema.sql`", "`TPCCItem.sql`", "`TPCCWarehouse.sql`" and "`TPCCStock.sql`" from Luminus "`Files > Project 3: TPC-C > Code`" folder.

   In the "`bt5110_p3`" database, execute the SQL file "`TPCCSchema.sql`" to create the schema.

   Populate the three tables named `item`, `warehouse` and `stock` using the corresponding SQL files.

   Execute the SQL files "`TPCCItem.sql`", "`TPCCWarehouse.sql`" and "`TPCCStock.sql`" in this order.

2. (2 points) Completing the Stocks

   In the data we provided, the table `stocks` only contains records for items and warehouses such that the quantity is non-zero. The item is only listed for a warehouse if it is currently available in this warehouse.

   We now want to change this and have a record for every item in every warehouse, possibly with a zero quantity if the item is not in stock.

   Write and execute a single SQL statement that adds to the table stocks the missing records.

   Save the statement in a file called "`completestock.sql`"

3. (0 points) Setting-up Django

   In this project, you create a Django project. The project consists of three apps. Each subsequent question is an app.

   The name of the Django project should be **Project3**.

   The URL of the Project and the apps need to be set up in such a way that you access the apps from the browser address `http://127.0.0.1:8000/AppRaw`, `http://127.0.0.1:8000/AppORM` and `http://127.0.0.1:8000/AppLine`, for Question 4, Question 5 and Question 6, respectively.

   - Start the `use_djangostack` terminal.

     The script ("`use_djangostack.bat`" or "`use_djangostack.bash`") is in your Bitnami Django stack installation folder.

     For Windows, click the "`use_djangostack.bat`" file.

     For Mac, open `Terminal` from Launchpad, and execute `cd /Applications/djangostack-2.2.4-0` to go to the Bitnami-Django stack installation folder (assuming that you have installed the stack in `/Applications/djangostack-2.2.4-0`). If you have installed Bitnami Django stack in a different folder change the line accordingly. On the terminal, execute `./use_djangostack`.
   - In the Django stack terminal, go to the Django projects directory.

     For Mac, execute the following in the terminal.

     ```
     cd /Applications/djangostack-2.2.4-0/apps/django/django_projects
     ```

     For Windows, execute the following in the terminal.

     ```
     cd C:\Users\[your username]\Bitnami Django Stack projects
     ```

     Change "`[your username]`" with your Windows username.
   - Create a new Django project in the current directory by running:

     ```
     django-admin.py startproject Project3
     ```

     If you use older version of Django (e.g. Django 1), run the following command:

     ```
     django-admin startproject Project3
     ```

     Note on Django Project and Django Apps:

     – A Django project comprises several apps including a default app (with the same name as the project) which it creates automatically.

     – Although not necessary, it is a good practice to put apps that take care of different functions of your website (e.g. admin or user panel, forum etc.) inside the project directory. You can see the directory of the default app named "`Project3`" inside the project directory named "`Project3`" (be wary of confusion due to naming).

     – Each App directory acts as a python module.

     – The default app directory "`Project3`" contains

       ∗ "`settings.py`" file: Use this file to set up your PostgreSQL database connection, add new apps that are part of project3, set the directory of Static files (e.g. CSS or JavaScript used by your webpages).

       ∗ "`urls.py`" file: When you write your project3 url in browser, Django looks for a matching pattern in the "`urlpatterns`" list of this file. Each app under this project also contains its own "`urls.py`" file.
   - Go inside the "`Project3`" folder with this command:

     ```
     cd Project3
     ```

- Start the Django server to serve "`Project3`" by executing

  ```
  python3 manage.py runserver
  ```

  If you use older version of Django (e.g. Django 1), change `python3` to `python`.
- Go to the browser address `http://127.0.0.1:8000` to check the default Project3 app page.
- Press Ctrl+C to stop the server.

4. (0 points) Building a Search App with Raw SQL

   Build an app called "AppRaw" that allows a user to search items by their name with a regular expression.

   From the user's perspective the app is a Web page with a text box and a submit button. The user enters his/her search query as a regular expression. Learn more about pattern matching and POSIX regular expressions with PostgreSQL at `www.postgresql.org/docs/9.3/functions-matching.html`. The list of identifiers, image identifiers, names and prices of the items the name of which matches the regular expression is returned on a page with the initial search widgets.

   Follow the steps below in order to build the app. The code we are providing uses raw SQL statements in Django.

   - With the Django Stack terminal in the "Project3" folder, execute the following command:

     ```
         python3 manage.py startapp AppRaw
     ```

     If you use older version of Django (e.g. Django 1), change `python3` to `python`.
     The command creates a folder for the app named `AppRaw` inside the "`Project3`" folder. You should have the files "`__init__.py`", "`admin.py`", "`apps.py`", "`models.py`","`tests.py`" and "`views.py`" as well as a folder "`migrations`" inside the newly created folder.
     For "`AppRaw`" to be recognised as an app under the project "`Project3`", edit the "`settings.py`" inside the default app "`Project3`" file and add a new element in the Python list "`INSTALLED_APPS = [...]`". The list should look like the following:

     ```
     INSTALLED_APPS = [
         'django.contrib.admin',
         'django.contrib.auth',
         'django.contrib.contenttypes',
         'django.contrib.sessions',
         'django.contrib.messages',
         'django.contrib.staticfiles',
         'AppRaw'
     ]
     ```

   - Configure "Project3" Database settings in Django You have to specify the associated database containing the tables in the "`settings.py`" file under the default app directory "`Project3`". Replace the existing "`DATABASES = {...}`" lines with the following

     ```
     DATABASES = {
         'default': {
             'ENGINE': 'django.db.backends.postgresql_psycopg2',
             'NAME': 'bt5110_p3',
             'HOST': 'localhost',
             'PORT': '5432',
             'USER': 'postgres',
             'PASSWORD': 'bt5110'  ## replace with your password
         }
     }
     ```

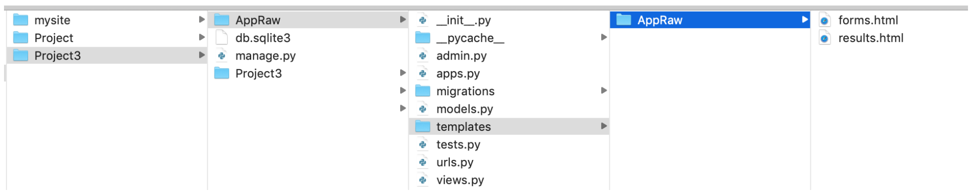
   - Create Views (Not to be confused with views in SQL).
     - Edit the file named views.py in the app directory "`AppRaw`".
     - Import certain libraries for establishing database connection and HTML page rendering and the models. Replace the content of the files with the following.

       ```
       from django.shortcuts import render
       from django.template import loader
       from django.http import HttpResponse
       from django.db import connection # connection module for Database operations
       ```

- When we visit the url `127.0.0.1:8000/AppRaw`, we want Django to call a view named "`AppRaw`" defined in the "`views.py`" file.
  One way to define a view in Django is to make it a function. The function "`AppRaw()`" should render the first page for us. Later in template we define how the page looks like in the "`forms.html`" file.
  Add the following function in "`views.py`":

  ```
  def AppRaw(request):
      # Renders 'AppRaw/forms.html' template with empty dictionary
      return render(request,'AppRaw/forms.html',{})
  ```

- Create Templates. Django looks for app templates under a folder named "`templates`". Create "`templates`" folder under "`AppRaw`" app directory. Django recommends to namespace all the templates associated to each app. Thus, we create another folder named "`AppRaw`" under "`templates`" and create empty "`forms.html`" file inside "`AppRaw`" folder. Refer to the figure below to avoid confusion.

  Add the following HTML code in "`forms.html`"

  ```
  <html>
      <body>
              <form action="result/" method = "GET">
                  SQL Reg ex:<br>
                  <input type="text" name="regex_id" id ="regex_id"><br>
                  <input type="submit" value="Submit">
              </form>
      </body>
  </html>
  ```

  We are adding an HTML "`form`" with a "`text`" input and "`submit`" button. The form uses the "`GET`" method. By mentioning `action="result/"` we are telling Django to jump from `127.0.0.1:8000/AppRaw/` to `127.0.0.1:8000/AppRaw/result/`. Although at this point we have not done the mappings from URLs to views.

- Configure URLs. Every time you create a new Django app, you have to configure two URLs file. One inside the "`Project3`" default app directory, and the other inside the app directory (in this case "`AppRaw`").
  Configure Project3 Urls. Open "`urls.py`" in "`Project3`" default app directory and edit it as follows:

  ```
  from django.contrib import admin
  from django.urls import path,re_path
  from django.urls import include

  urlpatterns = [
      path('AppRaw/',include('AppRaw.urls'))
  ]
  ```

  If you use older version of Django (e.g. Django 1), the file should be as follows:

  ```
  from django.contrib import admin
  from django.conf.urls import url,include
  ```

```
urlpatterns = [
    url('AppRaw/',include('AppRaw.urls'))
]
```

Configure App urls. Create a file name "urls.py" in the "AppRaw" directory. It should contain the following line -

```
from django.urls import path,re_path
from . import views # Import the views

urlpatterns = [
  re_path(r'^$', views.AppRaw),
  path('result/',views.getrows_db)
  ]
```

If you use older version of Django (e.g. Django 1), the file should be as follows:

```
from django.conf.urls import url
from . import views # Import the views

urlpatterns = [
  url(r'^$', views.AppRaw),
  url('result/',views.getrows_db)
  ]
```

In the list "urlpatterns" we are telling Django server to call "AppRaw" view from "views.py" file if the part in the url after 127.0.0.1:8000/AppRaw/ is empty. We are also specifying that if the part in the url after 127.0.0.1:8000/AppRaw/ is "result/" call "views.getrows_db" view.

- More views and templates. At this point, the "views.py" requires a view called "getrows_db" to deal with "GET" requests from rendering of "forms.html", database query and showing the records returned by the query. We edit "views.py" inside the "AppRaw" file to add the following function.

```
def getrows_db(request):
    if request.method == 'GET':
        # values sent via GET by the user
        form = request.GET.get('regex_id','')
        query = 'SELECT * FROM item WHERE i_name  ~ \'%s\'' % form
        # The connection object.
        c = connection.cursor()
        # Execute query by connection object
        c.execute(query)
        # Fetch all the rows. fetchall() returns a list of tuples.
        results = c.fetchall()
        context = {'records': results}
        return render(request,'AppRaw/results.html',context)
```

Create another template "AppRaw/results.html" to look like the following.

```
<html>
    <body>
    <table>
        {% for r in records %}
                <tr>
                  <td> {{ r.0 }}</td>
                  <td> {{ r.1 }}</td>
                  <td> {{ r.2 }}</td>
                  <td> {{ r.3 }}</td>
                  <td> {{ r.4 }}</td>
                </tr>
        {% endfor %}
        </table>
    </body>
</html>
```

You are encouraged to read about each of the topics `Models`, `Views` and `Templates` from the Django documentation (by topics) at `https://docs.djangoproject.com/en/2.2/topics/`

- Execute the following command from "`Project3`" Django project directory.

  ```
  python3 manage.py runserver
  ```

  If you use older version of Django (e.g. Django 1), change `python3` to `python`.
  Then go to `127.0.0.1:8000/AppRaw` in your browser.

5. (2 points) Building a Search App with Django's Object Relational Mapping

   Build a new app called "`AppORM`" that is the same app as in Question 4 but uses Django's QuerySet (`https://docs.djangoproject.com/en/2.2/ref/models/querysets/`) application programming interface leveraging Django's object relational mapping to query the database tables. Use the tables and the content that we gave you in PostgreSQL. Do neither create nor populate the tables from Django using the ORM. Use the ORM to map and query the existing tables. You are not allowed to use raw SQL queries in Django.

   Django uses models to define the mapping between its objects and objects in a relational database. This mapping is often refers to as the object relational mapping (ORM).

   A model is a Python class that contains the fields and integrity constraints corresponding to the data that you are storing in the database. Generally, each model is a class that maps to a single database table. Each attribute of the model represents a database column.

   You need to change the "`models.py`" file in your newly created "`AppORM`" folder. All models in the app are defined as a class in "`models.py`" as follows.

```
from django.db import models

class Item(models.Model):
    i_id = models.IntegerField(primary_key=True)
    i_im_id = models.CharField(unique=True, max_length=8)
    i_name = models.CharField(max_length=50)
    i_price = models.DecimalField(max_digits=5, decimal_places=2)

    class Meta:
        managed = False
        db_table = 'item'

class Stock(models.Model):
    w = models.ForeignKey('Warehouse', models.DO_NOTHING, primary_key=True)
    i = models.ForeignKey(Item, models.DO_NOTHING)
    s_qty = models.SmallIntegerField()

    class Meta:
        managed = False
        db_table = 'stock'
        unique_together = (('w', 'i'),)

class Warehouse(models.Model):
    w_id = models.IntegerField(primary_key=True)
    w_name = models.CharField(max_length=50, blank=True, null=True)
    w_street = models.CharField(max_length=50, blank=True, null=True)
    w_city = models.CharField(max_length=50, blank=True, null=True)
    w_country = models.CharField(max_length=50, blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'warehouse'
```

6. (6 points) Building a Line Order and Delivery App

   Build a new app called "`AppLine`" that allows a user to enter and place a line order or enter or to realise a line delivery. A line order or delivery is the updating of the stock quantity of one item in one warehouse. The quantity is decreased in the case of a line order. The quantity is increased in the case of a delivery.

   The user should be able to search and select the item, search and select the warehouse, indicate the quantity and choose whether it is an order or a delivery.

   If the line order or the line delivery is successful, the quantity of the respective item in the respective warehouse is updated accordingly to the line order or delivery.

   The message "`Your line order or delivery has been accepted`" is displayed.

   If the line order cannot be placed because the stock for the item in the warehouse is insufficient. The line order is not placed. The stock remains unchanged.

   The message "`Insufficient stock`" is displayed.

   Once the line order is placed or the line delivery is realised, the user can place a new line order or request a new line delivery.

   From the users perspective the app can be a single interactive and dynamic Web page or a sequence of several successive Web pages. More points are given for a more integrated and elegant solution.

   Prefer a solution that uses Django ORM with PostgreSQL predefined tables. Fewer points are given if you opt to use raw SQL. Be encouraged to push the logic of the application into SQL using integrity constraints, transactions, views, functions and triggers, if necessary. More points are given for using integrity constraints and catching the errors (for instance for the case of insufficient stock). More points are given for appropriate use of these tools. Do not use JQuery. Use JavaScript to select the item and the warehouse from the results of the search. Define the foreground, background and font colours using CSS. Although you are free to experiment, JavaScript and CSS styles should otherwise mostly contribute to the functionalities of the app. More points are given for such appropriate use of these tools. Provide stepper arrows for adjusting the quantity.