Seneca | FACULTY OF CONTINUING EDUCATION AND TRAINING

SENECA WEB PROGRAMMING PROGRAM
SUMMER 2024 SESSION

Module Number: WEB 302
Assignment #3

Instructor: Tim Lai
Title: Flask CRUD Application
Assigned: Thursday June 20
Due date: 11:59pm Friday July 12
Value: 25%

Submission details: This assignment MUST be submitted in at least one of 2 ways, as specified below. If the assignment does not follow the specified instructions, it will be considered incomplete. *Failure to upload this assignment before the due date, without negotiating an extension prior to the due date, will result in a grade of zero*.

1. This assignment should be uploaded to a **GitHub repository** which I will invite you to. When I download this GitHub repository, I must be able to access all your source files so that I can see your Python code, HTML/Jinja code and connect your site to a database.
2. If you have difficulty uploading to GitHub, then you may alternatively upload your assignment as a ZIP file to Blackboard. *DO NOT EMAIL ME A ZIP FILE AS AN ATTACHMENT. I WILL NOT RECEIVE IT.*

Instructions: You will be creating a fully-functional **MVC CRUD Flask application** which allows users to create, read, update and delete **objects** which can belong to a **class of your choice** (i.e. Spaceship, Dog, Ninja etc.) *as long as it is NOT a Knight or Clown – be creative*. This application will store the objects in an **SQL database** using **Flask WTF** and the **MySQL Connector** module.

Your application should make a clear distinction between the **Models**, **Views** and **Controllers**, and should follow proper **MVC** and **object-oriented** principles to the best of your ability. It should use logical folder structure and naming conventions.

Follow these steps to create the Flask application:

1. **Environment:** Set up your project's environment and install any required packages.
    a. **Virtual Environment:** Update PIP. Then install **virtualenv** and **virtualenvwrapper/virtualenvwrapper-win** using **PIP**. Create a virtual environment using virtualenvwrapper/virtualenvwrapper-win and work on that environment. Remember that if you are using Windows, you must install **virtualenvwrapper-win** and you must use a Windows CLI such as Command Prompt or Powershell, or run the **cmd** command in Git Bash in order for the virtualenvwrapper-win commands to

be recognized. Using an alternate virtual environment package such as **venv** is also acceptable.

   b. **PIP Packages:** Once you are working on the virtual environment, install **Flask-WTF**, **mysql-connector** and **python-dotenv** and save your project requirements in a **requirements.txt** file using the **pip freeze** command.
   c. **Environment Variables:** Store your **database credentials** and other environment variables in a **.env** file and then import them into a dedicated module and save them as constants using **python-dotenv**. You should also randomly generate a **SECRET_KEY** constant in this file which is to be used for **CSRF tokens**.

2. **Database:** Create your database, connect to it and create your database table.
   a. **Create Database:** Create a Python script to connect to your **MySQL user**, create a **cursor** and create your **database**, if it does not already exist, using **mysql-connector**.
   b. **Connect to Database:** In another module, create another Python script to connect to the **database**, create a new **cursor**, which has the dictionary parameter set to true to return rows as dictionaries.
   c. **Create Table:** In another module, create a database table for your class of object, if it does not already exist, using **mysql-connector**.
3. **Utility Functions:** Create a utilities module for storing any **utility functions**. This should include any functions which you need to **filter** or **sanitize** data. At a minimum, this should include a function to **filter strings** which strips whitespace, HTML tags and any special characters and a function to **format a name** or title with a capital first letter. You can also include a function to escape HTML characters, but it is not required if you use Jinja filters for HTML escaping.
4. **Models:** You will be making two Models – a generic Model class which can act as a parent class to any other Models, and which handles all database operations, and a specific Model, which represents your object's class and is a child class of the generic Model class.
   a. **Parent Model:** The generic parent Model should have a constructor method which defines a table name property, which will be a string and a columns property, which will be a tuple of strings.
   b. **Insert Method:** This class should also have an **insert()** method, which inserts a new row into a database table using the value of the **table** property, and using all of the columns listed in the **columns** tuple.
   c. **Update Method:** It should also have an **update()** method, which takes in an ID as a parameter, and updates a row with an ID that matches the ID which was passed in, using the table name property and the columns listed in the columns tuple.
   d. **Save Method:** To simplify the interface, you may want to include a save() method which checks for an ID and calls the **update()** method if one is found and the **insert()** method otherwise. The insert(), update() and save() methods should work regardless of the amount, names and data types of the columns which a child Model has.
   e. **Delete, Get All and Get Methods:** This class should also have a **delete()** method, which takes in an ID as a parameter, and deletes a row from the database table with an ID which matches the ID which was passed in, a **get_all()** method, which gets all of the rows from the database table and a **get()** method which takes in an ID as a parameter and which gets one row from the database with an ID which matches the ID which was passed in as a parameter.
   f. **Child Model:** The specific Model which represents your object's class, must be defined as a **child class** of the Model class.

    g. The child Model class should also have a **constructor method** which uses the **super()** function to call the parent Model class' **constructor method** inside of it. The first argument of the parent class' constructor method should be the name of the database table associated with the child Model and the second argument should be a tuple which lists the names of the columns in that table. The child Model class's constructor method should also take in a form parameter and use it to define at least **3 properties** – a **name**, an **image** and something **unique to your class** which can be of a data type of your choice.
    h. Both Model classes should only make use of properties which are accessed using **@property** and **@property_name.setter** decorator methods. Setter methods should perform necessary validation and sanitization. Any strings must be **filtered and validated** to prevent XSS attacks and hacking. Names should be formatted with **capital first letters**.
5. **Forms:** There should also be a forms module which defines classes for **adding**, **editing** and **deleting** the objects using **WTForms** and the **FlaskForm** class. Image files should be checked against a **collection of approved files** (this can be done with **Flask WTF**).
6. **Controllers:** Create a separate module to hold your **Controllers**.
    a. **Flask Instance:** In this module, create a new instance of the Flask class and save it in a variable.
    b. **Secret Key:** Store the value of the SECRET_KEY constant in the SECRET_KEY key of the Flask instance's config attribute.
    c. **Routes:** Use the Flask instance's route decorator to create distinct **add()**, **create()**, **read()**, **show()**, **edit()**, **update()** and **delete()** functions. These functions should be used to transfer data from the Model to the View and from the View to the Model.
    d. **Add Function:** The add() function should point to a template with a form to add a new object to the database and should pass the **args** attribute of the **request** object and an instance of the add form that you created using FlaskWTF.
    e. **Create Function:** The create() function should accept the POST method, create a new instance of the add form, pass the **form** prop of the request object into it as an argument, and set the **csrf_enabled** argument to True. Check if the **validate_on_submit()** method of the form object returns True, and if so create a new instance of your child Model class, pass the form object into it as the argument of its constructor method and then call the Model class' insert() or save() method to save the data. Then redirect to the read() route with a success message if validate_on_submit() is True and redirect back to the add() route with an error message if it is False.
    f. **Read Function:** The read() function should create a new instance of the child Model class and call its **get_all()** method to store a list of all of the rows from the database in a variable and point to a template which loops through and displays the data from all of the rows, pass the **args** attribute of the **request** object and the rows from the database to the template.
    g. **Show Function:** The show() function should take in a row's ID as a parameter, create a new instance of the child Model class, call its **get()** method and pass in the ID as an argument, to store a row with a matching ID from the database in a variable. It should also create a new instance of the delete form that you created using FlaskWTF. It should then point to a template which displays the row's data with a form to delete an object from the database, pass the **args** attribute of the **request** object, the row from the database and the delete form object to the template.

h. **Edit Function:** The edit() function should take in a row's ID as a parameter, create a new instance of the child Model class, call its **get()** method and pass in the ID as an argument, to store a row with a matching ID from the database in a variable. It should also create a new instance of the edit form that you created using FlaskWTF. It should then set the properties of the form object to the values of the returned row's columns. It should then point to a template with a form to edit an object from the database, pass the **args** attribute of the **request** object, the row from the database and the edit form object to the template.
i. **Update Function:** The update() function should accept the POST method, create a new instance of the edit form and set the **csrf_enabled** argument to True. Collect the ID of the row from the form data and save it in a variable. Check if the **validate_on_submit()** method of the form object returns True, and if so create a new instance of your child Model class, pass the form object into it as the argument of its constructor method and then call the Model class' update() or save() method and pass the row's ID into it to save the data. Then redirect to the read() route with a success message if validate_on_submit() is True and redirect back to the edit() route with an error message if it is False.
j. **Delete Function:** The delete() function should accept the POST method, create a new instance of the delete form and set the **csrf_enabled** argument to True. Collect the ID of the row from the form data and save it in a variable. Check if the **validate_on_submit()** method of the form object returns True, and if so create a new instance of your child Model class and then call the Model class' delete() method and pass the row's ID into it to delete the data. Then redirect to the read() route with a success message if validate_on_submit() is True and redirect back to the read() route with an error message if it is False.

7. **Views:** The views will be made up of the following Jinja HTML templates:
   a. **View Objects Template:** The application should start on a **View Objects** page which displays the **name** of each of the objects which have been added to the database. The user should also be able to view an **image** which is associated with each object.
   b. **Add Object Template:** There should be a clear **navigation item** or **button** that links to an **Add Object** page which displays a **form** where users can enter **properties** for a new object, including a **name** and an **image**. If the user does not fill out all the required fields, then they should be presented with an **error message** and should not be able to proceed. If the user correctly fills out all the required fields and the data gets sent to the database then they should be taken back to the View Objects page where they are presented with a **success message** and can see all the objects in the database, including the one they just added.
   c. **Show Object Template:** The Show Object page should display all the object's data and have an **Edit button** and a **Delete button**. When the Delete button is clicked the selected object should be **deleted** from the database. If the object is successfully deleted, they should be presented with a success message. If there is an error, they should be presented with an error message.
   d. **Edit Object Template:** When the Edit button on the Show Object page is clicked, the user should be taken to an **Edit Object** page where there is a **form** which is **already filled out** with that **object's properties**. If the user edits any of the properties, then they should be taken back to the View Objects page where the changes take effect. If the user leaves one of the required fields blank, they should be presented with an error message and should not be able to proceed.
   e. *I MUST be able to add objects using the form on the Add Object page, edit objects using the form on the Edit Object page, and delete objects using the Deletion*

buttons. Any strings must have HTML **escaped** before being displayed to prevent XSS attacks and hacking, which can be done using a utility function or using Jinja's e filter.

8. **UI/UX Design:** Create the user interface for your application using Jinja, HTML and CSS.
    a. **HTML/Jinja Templates:** The application should have an intuitive and functional **user interface** which uses **Jinja templates**, including a parent template with child templates for the pages. Nothing complicated or fancy is required for the user interface design, but the user interface should be easy to navigate and use. It must make use of appropriate **headings**, a **header**, a **footer** and **forms**. You should make use of additional semantic markup where appropriate such as section and paragraph tags. You should demonstrate a basic competency with HTML, and you *MUST* have valid HTML according to the W3C validator (https://validator.w3.org/) or there could be deductions. Accessibility considerations should be made for non-standard users.
    b. **CSS:** You must also apply a minimal amount of CSS to the page (i.e. updating the font family, background color, margin, padding etc. of elements) by linking to an **external CSS file**. Use of CSS libraries such as Bootstrap, or Tailwind is *NOT* required but is strongly encouraged. If you would rather use one or more of those libraries to apply your CSS styling instead of writing your CSS code from scratch, then that is okay. Responsiveness is also *NOT* required but is strongly encouraged.
    c. **JavaScript:** JavaScript is *NOT* required and will not be evaluated. However, you may use JavaScript if you *DO NOT* use any JavaScript for your form validation and *DO NOT* use any JavaScript libraries or frameworks which use HTML templates, such as React, Vue or Angular. If you use JavaScript for either of these purposes, then there may be deductions.

Grading breakdown:

- Creation of database with mysql-connector and python-dotenv **(10%)**
- Filtering and sanitization with utility functions **(5%)**
- Parent and child Model classes with database methods **(20%)**
- Add, edit and delete form classes with Flask-WTF **(10%)**
- Controller with Flask routes and functions **(20%)**
- Displaying data and forms in Views using Jinja templates **(20%)**
- UI/UX design with HTML, CSS and Jinja templating **(10%)**
- Good coding/file organization practices **(5%)**

Your files should contain only valid, as determined by the W3C validator, (https://validator.w3.org/) HTML code. You can use invalid code in the HTML but if you do, you must include a comment beside the invalid code explaining your decision. To properly validate the HTML in your Jinja templates, you MUST do a "view source" of the page in a web browser. The W3C validator does not understand Jinja code. *You will lose 2% per type of uncommented validation error*. You should not have any Python or Flask errors. *You will lose 2% per Python or Flask error which could have been fixed*.

Late Policy

All late assignments will be given a grade of zero.

Plagiarism

There are serious penalties for cheating and plagiarism offences and you are expected to be aware of our Academic Honesty Policy. Please refer to the Academic Policy at http://www.senecacollege.ca/ academic-policy/acpol-09.html for more information.