Candy Queen

Lecture assignment

In this assignment we are going to learn more about the **Django** framework. We will start at the basics and gradually learn more concepts within this fantastic framework! We will be using the fictional candy store called **Candy Queen** as a project theme 🍭. Feel free to go wild! 🍫🍩

Assignment description

Here below is the functionality this assignment should provide:

Project setup

- A project setup using **PyCharm**
  - Makes use of a virtual environment
  - Django setup
- Examine the files within the scaffolded project
- Three apps should be setup
  - Candy
  - User
  - Manufacturer

Views

- Let's add three views
  - candies
  - candies/{id}
  - manufacturers
- Let's add URL mappings for these new views
- The candies view should be implemented as follows:
  - The route to that view is /candies
  - It should return a model which consists of all candies
- The candies/{id} view should be implemented as follows:
  - The route to that view is /candies/{id}
  - It should return a single candy model which is associated with the provided id
  - If the model is not found, it should return a status code of 404
- The manufacturers view should be implemented as follows:
  - The route to that view is /manufacturers
  - It should return a model which consists of all manufacturers

Templates

- Let's start by creating three templated HTML documents
  - candies.html
  - candy_detail.html
  - manufacturers.html
- Instead of letting the views return a raw HTTP response, let's make them return a response which populates a templated view
- candies.html
  - Render the list of candy models in a fashionable matter
  - Each candy should consist of:
    - Name
    - Category
    - Price
    - On sale
    - Manufacturers logo
    - The first image of candy
  - Each candy should be a link where you can click it to get to the details site
- candy_detail.html
  - All the fields above
  - Description

        - All images associated with candy
- manufacturers.html
    - Render the list of manufacturer models in a fashionable matter
    - Each manufacturer should consist of:
        - Name
        - Logo
        - Year of start
        - Number of associated candies
- A layout site should be setup which is rendered for every page within the application
- A CSS file should be a part of the layout site and residing within a /static folder

Database

- A database **Postgres** should be used for this assignment. The database can be setup online via https://www.elephantsql.com/.
    - All models should be mapped to the database.
    - All data coming from the database should make use of the Model API
- Setup the connection string to the database
- These models should be setup
    - Candy
        - name
        - description
        - category
        - price
        - on_sale
        - manufacturer
    - CandyImage
        - image
        - candy
    - CandyCategory
        - name
    - Manufacturer
        - name
        - year_of_start
        - logo
- When these models have been setup, let's create a migration script
- Take a look at that migration script and then execute it
- Let's open the database viewer in PyCharm and examine the table structure
- Let's add some dummy data to the database using a predefined population script called population_script.sql
- Open up the command line and play around with the Model API
- Now let's replace all this static data within the views with actual data provided from a database
- Let's rerun the application and see the data flowing!
- Let's add a view for getting a detailed page for a candy and a template
- Rerun the application and check it out!

Data from and to the server

- So far we have only been working with GET requests, but we would like to be able to send data to the server
- Let's take a look at different methods to send data to the server
    - Via a form
    - Via AJAX and JSON formatted
    - Via URL parameters
    - Via Query parameters
- With these methods in mind, let's create views:
    - candies [POST]
        - Should make use of FormModel for the candy model
        - Add validation to the model
        - Validate the model server-side and return a suitable status code if the model is not valid
        - If valid, add the candy to the database
    - candies/{id} [DELETE]
        - Should delete a candy based on the URL parameter id provided
        - If the model is not found, should return a suitable status code
    - candies/{id} [PUT]
        - Should make use of FormModel for the candy model
        - Add validation to the model
        - Validate the model server-side and return a suitable status code if the model is not valid
        - If valid, update the candy within the database
- Let's add a query parameter to the candies [GET] method which has the name search_filter and if provided should filter the candies based on their name
- Let's add a search input box within the candy template and a button which when pressed should issue a GET request and filter out the data based on the search string entered by the user

Authentication

- A user should be able to authenticate to the application
    - Register
    - Login
    - Logout
- Let's start by creating three new views
    - register
        - GET
            - Retrieve the register.html template
        - POST
            - Accept the information needed when a user is registering
            - Should be validated
    - login
        - GET
            - Retrieve the login.html template
        - POST
            - Accept the information needed when a user is logging in
            - Should be validated
            - Should return a suitable status code if the user does not have access to the application
    - logout
        - Should log out the user and redirect to main site
- The user has a couple of fields associated with it. Let's say we would like to add more fields associated with the user
    - Add a profile image to the user
    - Add a favorite candy associated with the user
    - Display the profile image in the navigation bar when the user is logged in

Error handling

- It is important to handle errors correctly within our application, so that the application doesn’t suddenly crash when thousands of concurrent users are buying candy at the same time.
- We are going to introduce a global exception handler that all exceptions end up in. Within this exception handler we can determine how to handle different types of exception