CAS CS 411

Software Engineering

Lab 1 - Concentration MVC

Goal

The goal of this lab is to build a decoupled model-view-controller (MVC) application. We will be building the single-player version of the card game concentration (sometimes called memory).

For the front-end, we will be using vanilla Javascript. For the back-end, we will be using Python/Flask.

Feel free to use different technologies if you desire. However, these lab instructions will be for the technologies outlined above.

Lab Virtual Environment

The first order of business is getting a Python virtual environment set up for the lab. A Python virtual environment is a separate environment for the Python interpreter into which you can install packages locally without impacting the system-level Python installation.

To create a new virtual environment, we will use the `virtualenv` package. Create a new directory and `cd` into it, then run

```
virtualenv lab1
```

to create a Python virtual environment named `lab1`. To start the virtual environment, run

```
source lab1/bin/activate
```

Note, to close the virtual environment, run the command

```
deactivate
```

Now we will install the libraries necessary to complete the lab. The package manager for python is PIP. It is common to keep the package requirements in a file called `requirements.txt`. To install the libraries (and their appropriate versions) needed for this lab from `requirements.txt` using PIP, make sure you are in the correct directory and in the `lab1` venv and run the command

```
pip install -r requirements.txt
```

Model

Overview

The model is responsible for:

- Keeping track of whether or not a card has been matched.

- Keeping track of card states (face down or face up).
- Determining if a game is over.

The model implementation can be found in `concentration/model.py`.

Representing Cards

Cards are represented as strings containing the value and suit. These strings follow the same convention as the file names in `concentration/images/`.

For example, the card *10 of diamonds* can be represented as "`10d`".

Representing Matches

The list `matched` stores card match information. It contains 52 booleans. Card `i` is matched if `matched[i]=True` and unmatched if `matched[i]=False`.

Representing States

The list `state` stores card state information. It contains 52 strings. Card `i` is face down if `state[i]="down"` and face up if `state[i]="up"`.

Implementation

Part 1: Game over

Implement part 1. `game_over` should return `True` if the game is over or `False` otherwise.

Hint: When a game is over, what should `matches` *look like?*

Controller

Overview

The controller is responsible for:

- Initializing and resetting the model.
- Tracking guessed cards.
- Tracking the number of guesses.
- Tracking when the game ends.

The controller will implement an API that the front end (view) can access. This API has multiple endpoints:

- POST: Reset the game.
- GET: Get card information (string, state, and match).
- POST: Select a card.
- GET: Get the number of guesses.

The controller implementation can be found in `concentration/controller.py`.

Understanding Flask API

The API functionality is implemented using the Flask Python library. The main Flask application `app` is initialized by calling:

```
app = flask.Flask(__name__)
```

All API endpoints are routed to `app`. When `controller.py` is ran, the Flask application is launched by calling:

```
app.run(debug=True)
```

Note, for this lab we are specifically running the Flask application in debug mode. The main benefit of debug mode is it allows the Flask application to automatically update when changes are made to `controller.py`.

Two endpoints have already been implemented: `/health` and `/reset`

`/health` accepts GET requests and returns a JSON of the following structure:

```
{
      "message": "OK"
}
```

`/reset` accepts POST requests and resets the current game. It returns the same JSON as `/health`.

To see these endpoints in action, lets start the API. Run `controller.py`. You should get an output similar to:

```
* Serving Flask app 'controller'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on [API ADDRESS]
Press CTRL+C to quit
```

The `[API ADDRESS]` is where Flask is hosting our API. Let's test out the `/health` endpoint. In a new terminal, run the `curl` command:

```
curl [API ADDRESS]/health
```

Let's test out the `/reset` endpoint. In a new terminal, run the `curl` command:

```
curl [API ADDRESS]/reset -X POST
```

Implementation

Part 2: Card information

Let's implement the `/card` endpoint. This endpoint gets the card information and should return a JSON of the following form:

```
{
      "card": [Appropriate card string],
      "match": [Appropriate boolean value],
      "state": [Appropriate state value]
}
```

A card's `index` is specified as a parameter to the GET request. This `index` can be used to find the appropriate information in the `model`.

To test this endpoint, run the `curl` command:

```
curl [API ADDRESS]/card/[index]
```

Part 3: Card selection

Let's implement the `/select` endpoint. This endpoint selects a card to play. Two edge cases are already implemented for you: selecting a matched card or selecting an already selected card.

The return data is also already implemented.

The task here is to implement the main logic when selecting a card. We need to:

- Change the card's `state` to `"up"`
- If two cards are selected, check to see if the two cards' values match. If they do, we need to update `match`.

To test this endpoint, run the `curl` command:

```
curl [API ADDRESS]/select/[index] -X POST
```

Previous curl commands will also come in handy to test this endpoint.

Part 4: Guesses counter

Let's implement the `/guesses` endpoint. This endpoint gets the number of guesses and should return a JSON of the following form:

```
{
      "guesses": [Appropriate guesses value]
```

```
}
```

We need to implement a guesses counter:

- This counter should be initialized globally (like `model`) with an appropriate value.
- It should be reset when the `/reset` endpoint is called.
- It should be updated when the `/select` endpoint is called.

To test this endpoint, run the `curl` command:

```
curl [API ADDRESS]/guesses
```

Upon completion, all API functionality is complete. At this point, a game of concentration could be played with only API calls (give it a try).

This is not an easy way to play the game so we will build a simple front-end view that uses this API.

View

The view is responsible for:

- Displaying the cards in a grid (face up or face down). Cards are clickable.
- Displaying the number of guesses.
- Displaying when duplicate or invalid cards are selected.
- Reports number of guesses when you win
- A reset game button.

The view implementation can be found in `concentration/view.html` and `concentration/view.js`.

Communicate with Flask API

The view will communiscate with the controller using the Flask API we finished implementing.

A function for each API call has been made. The functions `apiReset()` and `apiGuesses()` are already implemented for you. These send POST and GET requests to the controller and call the `/reset` and `/guesses` endpoints respectively.

Implementation

Part 5: Communicate with Flask API.

You may have noticed that the `apiReset()` and `apiGuesses()` rely on a global variable `API_URL` that has been set to `null`. First, replace this with the URL of your Flask application.

Now, we need to finish implementing `apiSelect()` and `apiCard()`. These implementation are almost identical to `apiReset()` and `apiGuesses()`. However, we must incorporate the card index into the API call.

Part 6: Select logic.

We need to implement the logic for `select()`. This function is called whenever a card is clicked.

The API call using apiSelect has been provided to you. For this task we must do the following:

1. Update the message text if there is an error in the selection. This can be done by calling `$('#message').text(TEXT)` where `TEXT` is the text you want to display.
2. Update the guesses label. This can be done by calling `$('#label').text(TEXT)` where `TEXT` is the value you want to display.

Once this implemented. You should be able to see the guesses value increase, and error messages displayed. Next, we will implement changing the card images.

Part 7: Update cards logic.

We will now implement the `updateCards()` function which changes the card images. Currently, the function loops over all the cards and sets their images to the face down image.

This logic is incomplete. A card's front image should be displayed if

1. The card is currently selected.
2. The card has been matched.

You will first want to make an API call for each card to get its information. With that information, you will need to set the appropriate image for the card.

Part 8: Update reset logic.

Our final task is to implement the `reset()` function which contains the reset logic.

In this task, we must:

1. Make an API call to reset the game.
2. Reset the displayed values in `$('#message')` and `$('#guesses')`.