EECS 485 Midterm Exam Spring 2023

This is a 120 minute exam. You will be allowed to access any online resources, including the lecture slides, project code, Google, Stack Overflow, and ChatGPT during the exam. You will not be allowed to collaborate or communicate with any human

For free response questions, type your answers in the space provided in each problem.

You are to abide by the University of Michigan/Engineering honor code. To receive a grade, please sign below to signify that you have kept the honor code pledge.

I have neither given nor received aid on this exam, nor have I concealed any violations of the Honor Code.

Signature: ____

Name: ____

Uniqname: ____

UMID: ____

NOTE: In the Spring semester we elected to only include Free Response questions on exams, but this semester we may include different question formats, such as Multiple Choice. In addition, due to the in-person format of the Fall 2023 exams, we will **not** be allowing access to online resources.

Free Response 1: Bank485

Implement Bank485, a financial transaction platform similar to Venmo / PayPal.

Users on the platform can "connect" and become **friends** with each other.
Users on the platform are able to send and receive financial transactions **only** amongst their friends.
Users have the option to make these transactions **public or private**.
Users have a feed, where they can view the public and private transactions made by themselves, and the public transactions made amongst their friends.

We have provided the following database schema:

```
CREATE TABLE users (
    username VARCHAR(20),
    profile_picture VARCHAR(64) NOT NULL,
    account_balance REAL,
    PRIMARY KEY (username)
);
```

```
CREATE TABLE friends (
    username1 VARCHAR(20),
    username2 VARCHAR(20),
    PRIMARY KEY (username1, username2)
);
```

```
CREATE TABLE transactions (
    transaction_id INTEGER AUTOINCREMENT,
    send_username VARCHAR(20),
    recieve_username VARCHAR(20),
    transaction_amount REAL,
    is_public INTEGER, # 1 if public
    PRIMARY KEY (transaction_id)
    FOREIGN KEY (send_username) REFERENCES users.username
    FOREIGN KEY (receive_username) REFERENCES users.username
);
```

Users are friends if there is a row where one of the users is `username1` and the other is `username2`. Which user is `username1` and which is `username2` does not matter. You can assume that there is only one row in `friends` for any pair of users.

Helper Functions

- `get_upload(filename) # returns flask.send_from_directory for 'filename'`

GET /

This route renders the ‘homepage.html’ template. The rendered template includes:

- the username and profile picture for logged in user and all their friends
- Links to ‘/transaction_form/’ web page
- the transaction history for the logged in user and all of their friends
    - Display transaction if the **logged in** user **or** a **friend** is the sender or receiver
    - Only display the **public** transactions that were completed by logged in users friends
    - Display both the **public** and **private** transactions the logged in user partakes in
    - Transactions must be displayed in descending order (newest transactions first), newer transactions will have a higher id

You can assume that there is always a logged-in user and can access the uniqname of the logged-in user via `flask.session[“logname”]`.

At the end of the route, you must render the `homepage.html` template with a context dictionary. The context dictionary must only contain the following keys:

- `users`: A list of user information for the logged in user and all their friends. Each dictionary element in the list must contain “username” and “profile_picture” keys.
- `transactions`: A list of all public transactions the logged in user, or one of their friends participated in. Each element in the list must contain “send_username”, “receive_username”, “transaction_amount” and “transaction_id” keys. A transaction must not appear in this list twice.

Here is an example context dictionary:

```
{
   "users": [ { "username": "user1", "profile_picture": "user1.png" } ],
   "transactions": [ {"send_username": "user1", "receive_username": "user2",
                       "transaction_amount": 12.50, "transaction_id": 1 } ]
}
```

```
@bank485.app.route('/')
def show_index():
    connection = bank485.model.get_db()

    # Get relevant usernames for homepage
    users = []
```

```
# users contains current usernames and profile pictures for logged in user and
their friends
```

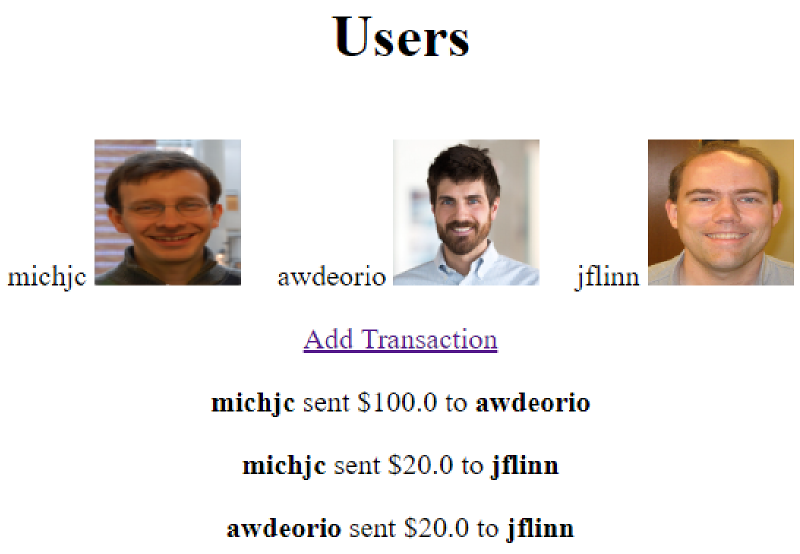
```
# Get relevant transactions, return rendered “homepage” template
 transactions = []
```

```
context = {"users":users, "transactions": transactions}

return flask.render_template("homepage.html", **context)
```

`homepage.html`

Using Jinja2 and HTML, write code to display the homepage of bank485. Refer to the below screenshot. You do NOT need to add any styling. The context is passed from from *show_index()*



Users

michjc awdeorio jflinn

Add Transaction

michjc sent $100.0 to **awdeorio**

michjc sent $20.0 to **jflinn**

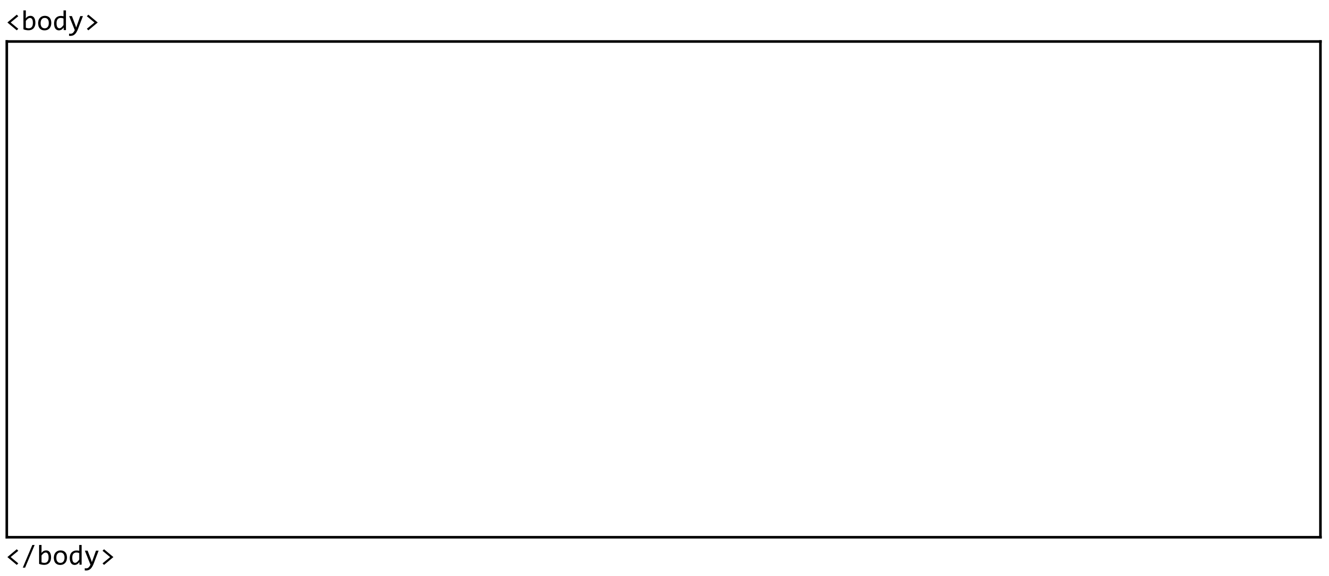
awdeorio sent $20.0 to **jflinn**

For each account that is either the logged in user or a friend, the page must display their username along with their profile picture.

Include a link to '/transaction_form/' to display the Transaction Form web page

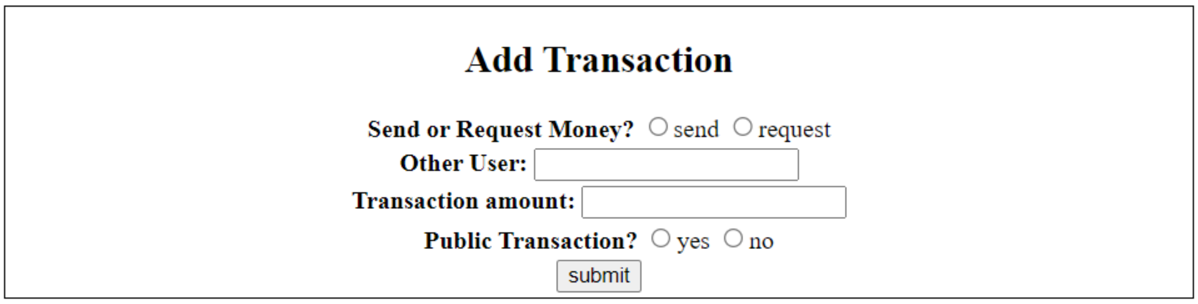
For each public transaction, include both the sending and receiving usernames, as well as the transaction amount. The amount must start with a $ but the number of cents can be displayed in any way you choose.

Profile pictures are stored as filenames that need to be passed to the `get_upload` Flask route in order to be converted to a URL, similar to the images in Project 2.



transaction_form.html

Using Jinja2 and HTML, write code to display a form for initiating transactions that sends a POST request to '/transaction/'. Refer to the below screenshot. You do NOT need to add any styling.


Add Transaction
Send or Request Money? ○ send ○ request
Other User:
Transaction amount:
Public Transaction? ○ yes ○ no
submit

The form must include the following fields:

- Send or Request 'radio' input - named "send_or_request"
- Text box to enter other username - named "other_user"
- Text box to enter transaction amount - named "amount" - type = "text"
- Public or Private 'radio' input - named "is_public"
- Submit button - named "submit"
- Hidden input(s) to pass any additional information

On submission, a POST request must be sent to the `/transaction/` route, which attempts to insert a new transaction into the database.

POST /transaction/

This route handles potential transactions

For a given incoming transaction request, add the transaction to the database if

- the users are friends AND
- the transaction can be completed with the monetary values in the users respective accounts

If these two conditions are met, add the transaction to the 'transactions' SQLite table. You do not need to worry about updating the monetary values in the users accounts after a transaction occurs - assume this is implemented when a transaction is added to the 'transactions' table. If either condition is not met, immediately redirect the logged in user to the homepage.

This route will have to determine whether the incoming transaction is of type send or receive, with respect to the logged in user.

This route must redirect the logged in user back to the homepage, where they can see the updated transactions.

```
@bank485.app.route('/transaction/', methods=['POST'])
def post_transaction():
    connection = bank485.model.get_db()

    # extract relevant form information
# check if transaction requirements met, add to DB if so
```

Free Response 2: Market485

Implement an online market, Market485, using React functional components for buyers and sellers to exchange goods.

We have provided the following database schema on the server:

```
CREATE TABLE users (
    username VARCHAR(20),
    profile_picture VARCHAR(64),
    PRIMARY KEY (username)
);
```

```
CREATE TABLE goods (
    id INTEGER AUTOINCREMENT,
    title VARCHAR(32),
    description VARCHAR(1024),
    seller_username VARCHAR(20),
    price REAL,
    available INTEGER,  # 1 if good is available
    PRIMARY KEY (id),
    FOREIGN KEY (seller_username) REFERENCES users.username
);
```

GET api/goods/

The API route *'/api/goods/'* accepts a GET request and returns a list of **all available** goods in the database. Each element in the list contains **exactly** the fields below:

```
GET /api/goods/
Returns
[
      {
            "title": string,
            "id": integer,
            "description": string,
            "price": float
      },
      …
]
```

```
@market485.app.route('/api/goods/')
def get_goods():
    # Connect to database
    connection = market485.model.get_db()
```

POST /api/add_good/

The API route *'/api/add_good/'* accepts a POST request from an HTML form, adding an entity to the *goods* table with the provided fields. The good must be marked as 'available' when it is added to the database. You may assume multiple users are not inserting goods at the same time. If there is an error adding the entry to the database, abort with a 404 error code. The route returns the newly added 'good' table entry following this format: .

POST /api/add_good/
Returns
```
{
  "available": integer,
  "description": string,
  "id": integer,
  "price": real,
  "seller_username": string,
  "title": string,
  “resource_url”: string  # /api/good/<id>/
}
```

```
@market485.app.route('/api/add_good/', methods=['POST'])
def add_good():
    connection = market485.model.get_db()
    title = flask.request.form["title"]
    description = flask.request.form["description"]
    seller_username = flask.request.form["seller_username"]
    price = flask.request.form["price"]
```

POST api/buy_good/

The API route *'/api/buy_good/'* accepts a POST request, removing an entity from the *goods* table based on the provided 'good ID'. Passes "good_id" key value pair in request body. Returns nothing.

POST /api/buy_good/<good_id>
body: {"good_id": xxx}

```
@market485.app.route('/api/buy_good/', methods=['POST'])
def buy_good():
    connection = market485.model.get_db()
```

React UI:

The main component of the application must display all available goods as shown below, allow a user to submit a 'buy_good' request to the server when pressing the "buy" button, as well as conditionally render a form to submit an 'add_good' request to the server.

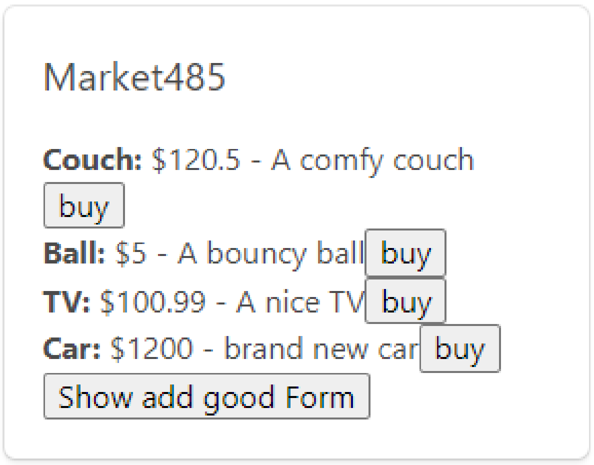
When a good is 'bought', the item must be removed from the UI without needing a page refresh. You can assume no users will try and buy the same item at the same time.

When a good is added, the item must appear in the UI without needing a page refresh. Additionally, the "add good" form must remain visible and the fields must be cleared.

The form must be included with the following specifications:

- Text box to enter title - named "title"
- Text box to enter price - named "price" - type = "text"
- Text box to enter description - named "description"
- Submit button - named "submit"

The following screenshot is the UI of the application. You must match the UI to the best of your ability. Styling is not required.



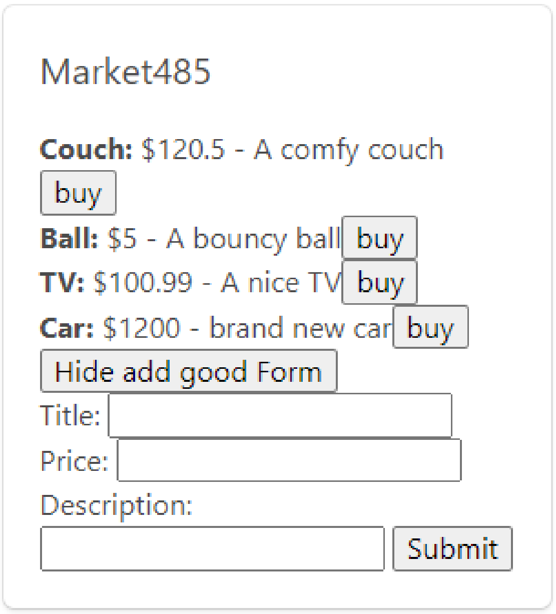
Market485

Couch: $120.5 - A comfy couch

Ball: $5 - A bouncy ball

TV: $100.99 - A nice TV

Car: $1200 - brand new car



Market485

Couch: $120.5 - A comfy couch

Ball: $5 - A bouncy ball

TV: $100.99 - A nice TV

Car: $1200 - brand new car

```
const BUY_GOOD_URL = 'http://localhost:8000/api/buy_good/'
const ADD_GOOD_URL = 'http://localhost:8000/api/add_good/'
const FETCH_GOODS_URL = 'http://localhost:8000/api/goods/'

const Good = (props) => {







}
const App = () => {
  var [goods, setGoods] = useState([]);
  var [showForm, setShowForm] = useState(false);

  let fetchGoods = function(){



  }

  let buyGood = function(id){







  }
  let handleAddGood = function(event){

  }

// Execute initial effects and return components, feel free to add any
// additional functions you may need in this box

}
```