Iteration 2 Project Document

Question 1:

What functionalities were implemented? What were added that were not planned? What functionalities were removed? Such information should be provided in this document.

What functionalities were implemented?

1. Sign up
2. Log in
3. Deposit money into account
4. Withdraw money from account
5. View transaction history
6. View deposit history
7. View withdrawal history
8. Send someone money
9. Pay a bill
10. Edit User's profile
11. Forgot username or email? Retrieve your username and email using your card number

What were added that were not planned?

1. Forgot username or email? Retrieve your username and email using your card number

What functionalities were removed?

1. Monthly charge deduction/ interest addition
2. interest addition(only on saving accounts): 0.2% interest is added on whatever the balance of the account in a month was.

2. MODELS

Question 2:
The document must have a section named Models where you document your models and what you can do with them (methods). Why are they necessary for your application?

Model 1: User

The user model has the following fields:

```
 username: {
   type: String,
   unique: true,
   required: [true, "invalid username"],
  },

  email: {
   type: String,
   required: [true, "email required"],
   unique: true,
   validate: [isEmail, "invalid email"],
  },

  password: {
   type: String,
   required: [true, "invalid password"],
   minlength: [8, "password has to be minimum 8 characters long"],
  },

  card: { type: String },
  balance: { type: String },
  depositHistory: { type: Array },
  withdrawHistory: { type: Array },
  moneyTransferHistory: { type: Array },
  receivedMoneyHistory: { type: Array },
```

Methods:

2. login: retrieves the user with the given email and password from the database. It is important because this user object is shown to the client once a user logs in

3. transection: It adds to subtracts money from the user's account balance and also updates the corresponding depositHistory or withdrawHistory array. It is important because it simulates deposit or withdrawal.

4.. transferMoneyTo: deducts the given amount from the user's balance and adds that amount to the recipients balance. It is important because it simulates money transfer from one user to another user.

5. putUserInfo: this updates the username and/or password and/or email of the user as requested from the client side. This method is important for updating user's account

Model 2: Company

A company can have an account with the bank, and the users can pay a bill to the company. This is the purpose of having a company model. It has the following schema

```
  companyname: {
    type: String,
    unique: true,
    lowercase: true,
    required: [true, "invalid company name"],
  },
  email: {
    type: String,
    required: [true, "email required"],
    lowercase: true,
    unique: true,
    validate: [isEmail, "invalid email"],
  },
  password: {
    type: String,
    required: [true, "invalid password"],
    minlength: [8, "password has to be minimum 8 characters long"],
  },
  card: { type: String },
  balance: { type: String },
  depositHistory: { type: Array },
  withdrawHistory: { type: Array },
  moneyTransferHistory: { type: Array },
  receivedMoneyHistory: { type: Array },
};
```

Methods:

1. log in: retrieves the company with the given email and password from the database. It is important because this company object is shown to the client once a company employee logs in

2. transferMoneyTo: deducts the given amount from the user’s balance and adds that amount to the recipients balance. This is important to receive a bill or send money to someone else

3. Routes and Controllers

Question 3
The document must have a section named Routes and Controllers that should report the paths used in your API. What each one of them does? How to use them? (Short code-snippets on how to access these paths are welcome in your report)

We have attached all the screenshots in the test section. See the test section for screenshots

1. signup a user/company

Test using postman

url: "/user/signup",

method: POST

json: {

"username": "",

"email": "",

"password": ""

}

if a company

url: user/signup/company

method: POST

json: {

"companyname": "",

"email": "",

"password": ""

}

}

2. user/company login

Test using postman url: "/user/login/company"

url: "/user/login",

method: POST

json: {

"email": "",

"password": ""

}

3. user deposit an amount when a user is logged in

Test using postman

url: "/user/deposit",

method: POST

json: {

"email": "",

"amount": ""

}

4. user withdraw an amount when a user is logged in

Test using postman method: POST

url: "/user/withdraw",

json: {

"email": "",

"amount": ""

}

// withdraw amount 0 validation taken care from the front-end

5. user get a deposit history when a user is logged in

Test using postman
method: GET
url: "/user/deposit/history”

[see details in Array named "depositHistory" in response Object]

6. user get a withdraw history when a user is logged in

Test using postman

url: "/user/withdraw/history"

method: GET

7. user get a withdraw and deposit history when a user is logged in

Test using postman url: "/user/deposit_withdraw/history"

method: GET

8. send some money to a user/person using their id

Test using postman

url: "/user/money/transfer/user",

method: POST

json: {

"recipientType": "", // value has to be user

"email": "", // money sender email

"amount": "",

"receiverID": "" // receiver(person) id

}

9. Pay a bill

Test using postman

url: "/user/money/transfer/company",

method: POST

json: {

"recipientType": "", // value has to be "company"

"email": "", // money sender email

"amount": "",

```
    "receiverID": "" // receiver(company) id
}
```

10. update a user profile when a user is logged in

Test using postman

```
  url: "/user/info/update",
  method: POST
  json: {
    "username": "updated value", // must contain an empty value
    "email": "updated value", // must contain an empty value
    "password": "updated value" // must contain an empty value
  }
```

11. api test: show some information of a person using their id

Test using postman

```
  url: "/api/user/id/<8 digit bank unique card number>"
  method: POST
```

Question 4.

You should show how you designed your Data Model collection(s)? Which pattern is it following (embedded or normalized)? Why you chose this particular pattern?

We used normalized data model as it

- Reduces redundant data
- Provides data consistency within the database
- More flexible database design
- Higher database security
- Better and quicker execution
- Greater overall database organization

Question 5

The document must have a section named Tests documenting the tests performed. Which tools were used? Which tests were performed? Why? How complete are your tests? Did they cover most success and failure cases?

We performed tests using mocha and postman. Type “npm test” from the terminal to run mocha test. We also tested using postman, and these tests include all the success and failure scenario for every functionality. See the Test section below. We have included all the screenshots of every test performed

Tests

Functionality 1: Sign up

url: localhost:3000/user/signup

request method: POST

post json example (All 3 key-value pairs of the following json must be present in the post data):

```
{
    "username": "test1",
    "email": "test1@gmail.com",
    "password": "test1test1"
}
```

expected response(success case): `"message": "succesfully signed up. Please login to access account"`

failure cases:

User enters

a. Username that is
   i. not unique

   expected response:

```
{
    "username": "username already exists",
    "email": "",
    "password": "",
    "card": ""
}
```

   ii. Enters an empty string for “username”

   example:

```
{
    "username": "",
    "email": "test1@gmail.com",
    "password": "test1test1"
}
```

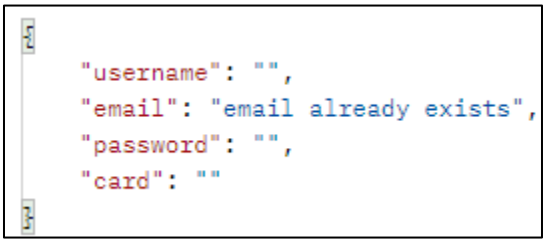
   Expected Response

```
{
    "username": "invalid username",
    "email": "",
    "password": "",
    "card": ""
}
```

b. email that is:

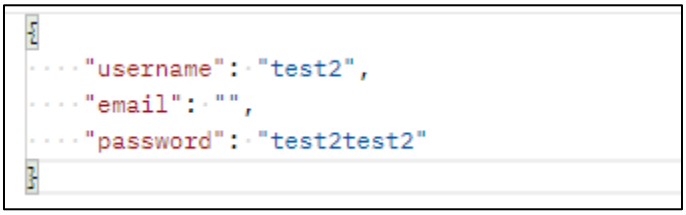
i. Not unique

Expected response:



ii. Enters an empty string for “email”

example:



expected response:

iii. Not the format of an email address

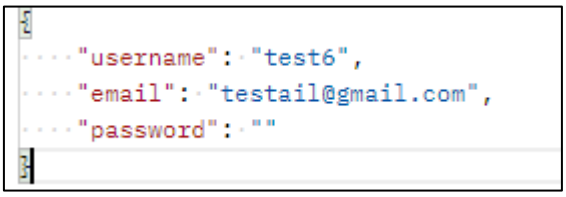
example:

expected response:

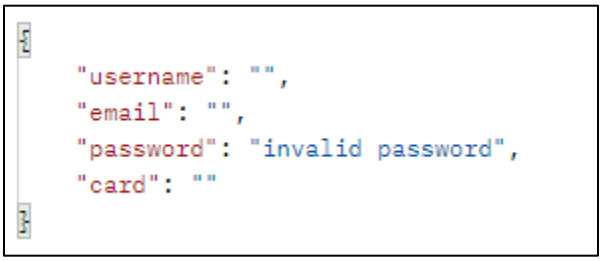
c. Password that is:

i. Enters an empty string for password

example:



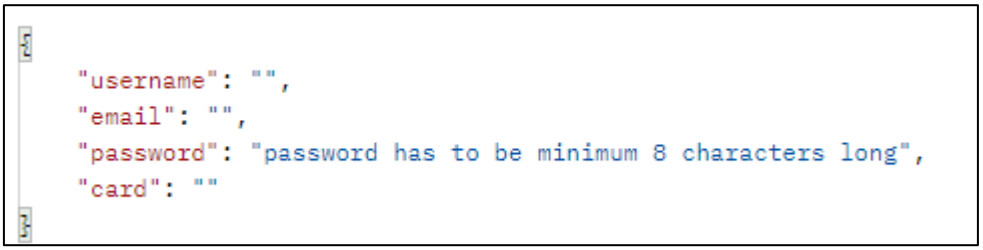
expected response:



ii. Not 8 characters or more

example:

expected response:



Functionality 2: Login

url: localhost:3000/user/login

request method: POST

post json example (All 3 key-value pairs of the following json must be present in the post data):

```
{
    "email": "test1@gmail.com",
    "password": "test1test1"
}
```

expected response example(success case):

```
{
    "user": {
        "_id": "62367297235ba47353249958",
        "username": "test1",
        "email": "test1@gmail.com",
        "password": "$2b$10$JYgaMuxd7jBb8hMSfscPa.SxAKd98QKsCea0.U3FIG6W2nMN0TN7e",
        "card": "2af984cd",
        "balance": "0",
        "depositHistory": [],
        "withdrawHistory": [],
        "moneyTransferHistory": [],
        "receivedMoneyHistory": [],
        "__v": 0
    }
}
```

Failure Cases:

1. User enters invalid email (an email that has not been used to signup)

   Example:

   ```
   {
       "email": "test2@gmail.com",
       "password": "test364test1"
   }
   ```

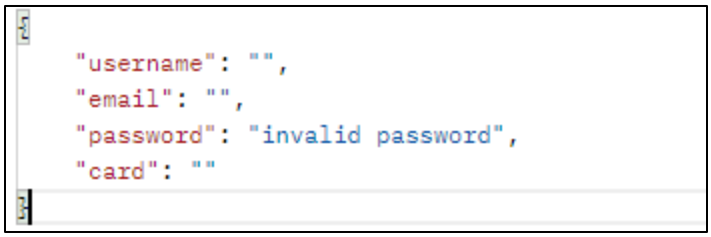
   Expected Response:

   ```
   {
       "username": "",
       "email": "invalid email",
       "password": "",
       "card": ""
   }
   ```

2. User enter invalid password (password doesn't match with the corresponding email)

Example:

Expected Response:



Important note: To test all other functionalities starting from functionality 3 inclusive, the user has to be logged in. If you try to test the endpoints without logging in i.e., without sending a correct post request to localhost:3000/user/login, you will receive the following response: "`Please log in to resume". We have used cookie to track whether a user is logged in. If while testing other endpoints, you get this message as a response, it will mean that the cookie has expired and you have been logged out. However, we have kept the expiry age of the cookie really long to give you the convenience of testing functionalities for a long time without getting logged out.` You can also manually delete the cookie from postman which simulates the expiration of a cookie.

Functionality 3: Deposit money in users account

url: localhost:3000/user/deposit

request method: POST

post json example:

```
{
    "email": "test1@gmail.com",
    "amount": "50"
}
```

Note: the email has to be the email associated with this user's account.

expected response example(success case): the app sends the "user" object as a response. See the increase in balance and the deposit history to ensure that the deposit has been done


```
{
    "user": {
        "_id": "62367297235ba47353249958",
        "username": "test1",
        "email": "test1@gmail.com",
        "password": "$2b$10$JYgaMuxd7jBb8hMSfscPa.SxAKd98QKsCea0.U3FIG6W2nMN0TN7e",
        "card": "2af984cd",
        "balance": "50",
        "depositHistory": [
            {
                "amount": "50",
                "date": "2022-03-20T01:30:17.603Z"
            }
        ],
        "withdrawHistory": [],
        "moneyTransferHistory": [],
        "receivedMoneyHistory": [],
        "__v": 0
    }
}
```

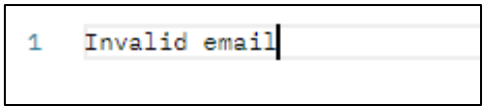
Failure Cases:

ser enters invalid email (an email that is not associated with this account)

Example:

Note: here test1@gmail.com is not the email associated with this account

Expected response:



Note: the case where the user enters “0” for amount has not been validated in the backend, as it will be done from the front-end.

Functionality 4: Withdraw money from users account

url: localhost:3000/user/withdraw

request method: POST

post json example:

```
{
    "email": "test3@gmail.com",
    "amount": 40
}
```

Note: the email has to be the email associated with this user's account.

expected response example(success case): the app sends the "user" object as a response. See the decrease in balance and the withdraw history to ensure that the withdrawal has been done


```
{
    "user": {
        "_id": "62375d64a6b4cf024488243a",
        "username": "test3@gmail.com",
        "email": "test3@gmail.com",
        "password": "$2b$10$9XRDnvyZ1IisGMB8HgrD4.W6hCO/k0ixyJnNQgSA2WUrFYG30xVIa",
        "card": "0e494537",
        "balance": "60",
        "depositHistory": [
            {
                "amount": 100,
                "date": "2022-03-20T16:59:34.253Z"
            }
        ],
        "withdrawHistory": [
            {
                "amount": 40,
                "date": "2022-03-20T16:59:45.695Z"
            }
        ],
        "moneyTransferHistory": [],
        "receivedMoneyHistory": [],
        "__v": 0
    }
}
```

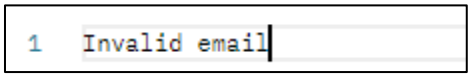
Failure Cases:

1. User enters invalid email (an email that is not associated with this account)

Example:

Note: here test1@gmail.com is not the email associated with this account

Expected Response:



2. User enters (tries to withdraw) an amount greater than the account balance

Example

Note: This user's balance is $60 but is trying to withdraw $100

Expected Response:

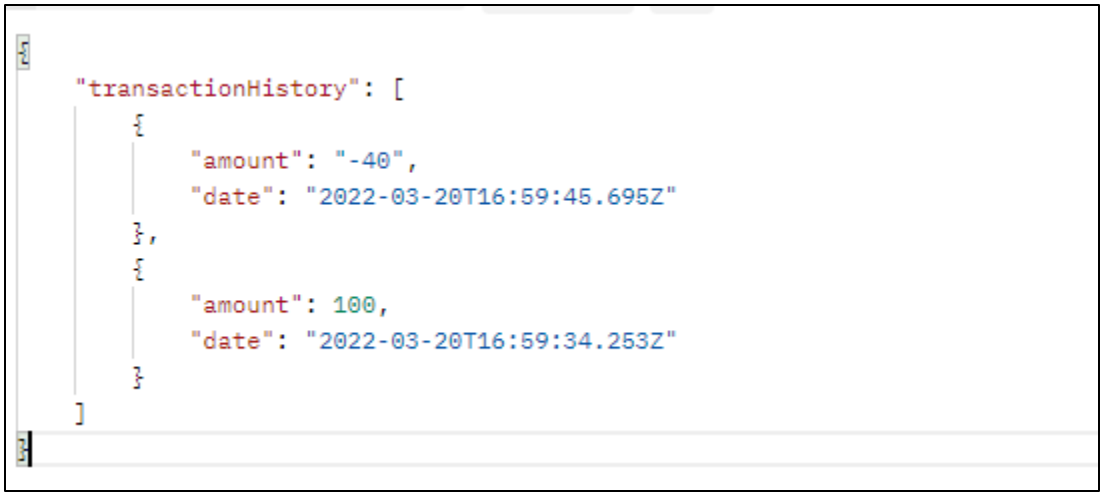
Note: the case where the user enters "0" for amount has not been validated in the backend, as it will be done from the front-end.

Functionality 5: View transaction history in users account

url: localhost:3000/user/deposit_withdraw/history

request method: GET

expected response example(success case):



Failure Cases:

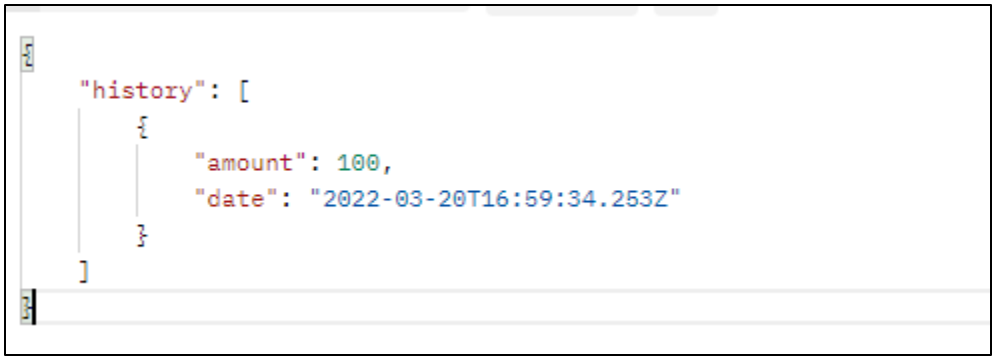
the only failure case for this endpoint is that if the user tries to send a request to this endpoint without logging in first. In this case, the user will be asked to “log in to resume”, as mentioned above in this document.

Functionality 6: View deposit history in users account

url: localhost:3000/user/deposit/history

request method: GET

expected response example(success case):



Failure Cases:

the only failure case for this endpoint is that if the user tries to send a request to this endpoint without logging in first. In this case, the user will be asked to “log in to resume”, as mentioned above in this document.

Functionality 7: View withdraw history in users account

url: localhost:3000/user/withdraw/history

request method: GET

expected response example(success case):

Failure Cases:

the only failure case for this endpoint is that if the user tries to send a request to this endpoint without logging in first. In this case, the user will be asked to “log in to resume”, as mentioned above in this document.

Functionality 8: Send money to another user

url: localhost:3000 /user/money/transfer/user

request method: POST

post json example

```
{
    "recipientType": "user",
    "email": "test3@gmail.com",
    "amount": "10",
    "receiverID": "90a67f02"
}
```

Note:

- enter an “user” for “recipientType”
- email is the sender’s/this account’s email
- amount is the amount user wants to transfer
- receiverID is receiver’s card number. This is the value for the key, “card” in the database. You can also find this value associated to a particular account by logging into that account.

expected response example(success case):

```
1   money transfer successful. make a GET request to localhost:3000/user/login to see the corresponding decrease in balance
```

To ensure transfer has been done, do the following

1. 1. make a GET request to localhost:3000/user/login to see the corresponding decrease in balance in sender’s account
2. 2. either check the database to see the increase in balance in the receivers account by logging in to the receivers account or you can also check in this in the database

Important note: sending and receiving money doesn’t not appear in the transaction or deposit or withdraw history, as this is not considered a transaction. When a user sends money, their account balance decreases by that amount, and the recipient’s balance increases by that amount.

Failure Cases:

1. User enters invalid email (an email that is not associated with this account)

Example:

```
{
    "recipientType": "user",
    "email": "test3@gmail.com",
    "amount": "10",
    "receiverID": "90a67f02"
}
```

Expected Response:

```
1   Invalid email
```

2. User enters invalid receiver Id (an card number that does not belong to any user)
Example:

```
{
    "recipientType": "user",
    "email": "test3@gmail.com",
    "amount": "10",
    "receiverID": "90a67f02"
}
```

Expected Response:

```
1   invalid receiver ID
```

3. User tries to send more money than what is available in their account (enter an amount bigger than their account balance)

Example:

```
{
    "recipientType": "user",
    "email": "test3@gmail.com",
    "amount": "10000000000000000",
    "receiverID": "90a67f02"
}
```

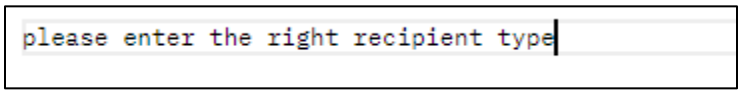
Expected Response:

```
not enoungh money to send
```

4. User does not enter “user” as a recipient type. Note: If the user enters “company” as the recipientType, then the app will try to send the money to a company, and since there is no company with the receiver id of a user, it will say “Invalid receiverID as a response”

Example:

Expected Response:



Functionality 9: Pay Bill

Note: to use this functionality, a company has to be signed up by sending a post request to “user/signup/company” with the following json.

```
{
   "companyname":  "",
    "email": "",
    "password": ""
 }
```

Now trying paying bill to that company in the following way:

url: localhost:3000 /user/money/transfer/company

request method: POST

post json example:

Note:

- enter “company” for “recipientType”
- email is the sender’s/this account’s email
- amount is the amount user wants to pay as bill
- receiverID is the company’s card number. This is the value for the key, “card” in the database. You can also find this value associated to a particular company account by logging into that account.

expected response example(success case):

```
money transfer successful. make a GET request to localhost:3000/user/login to see the corresponding decrease in balance
```

To ensure transfer has been done, do the following

1. make a GET request to localhost:3000/user/login to see the corresponding decrease in balance in sender’s account

2. either check the database to see the increase in balance in the receivers account by logging in to the receivers account or you can also check in this in the database

Failure Cases:

1. User enters invalid email (an email that is not associated with this account)

Example:

```
{
    "method": "",
    "email": "test1@gmail.com",
    "amount": "9",
    "receiverID": "f9051371"
}
```

Expected Response:

```
1   Invalid email
```

2. User enters invalid receiver Id (an card number that does not belong to any company)

Example:

```
{
    "method": "",
    "email": "test3@gmail.com",
    "amount": "9",
    "receiverID": "f9051372"
}
```

Expected Response:

```
1   invalid receiver ID
```

3. User tries to pay more money than what is available in their account (enter an amount bigger than their account balance)

Example:

Expected Response:

```
not enoungh money to send
```

3. User does not enter “company” as a recipient type.

Example:

Expected Response:

```
invalid receiver ID
```

Functionality 10: Update User Profile Info

url: localhost:3000/user/info/update

request method: POST

post json example:

Note:

- Enter the value of those fields that you want to update. For other fields, provide an empty string

expected response example(success case):

Failure Cases:

The only failure cases are if the user tries to do this without logging in, or is not posting the correct json required.

Functionality 11: Forgot Email

If the user forgot the email they used to sign in, and can no longer log in, they can retrieve their email and username using their card number

url: localhost:3000/ /api/user/id/<8 digit bank unique card number>

request method: GET

expected response example(success case):


```
{
    "username": "test1",
    "email": "test1@gmail.com"
}
```


Failure Cases:

1. User enters invalid card number:
Expected Response:


```
{
    "error": "user not found"
}
```