SVB BANK

Jakob Short — CS
Olivia Yee — CS
Samantha Bracellari — CS

User Manual

CONTENTS

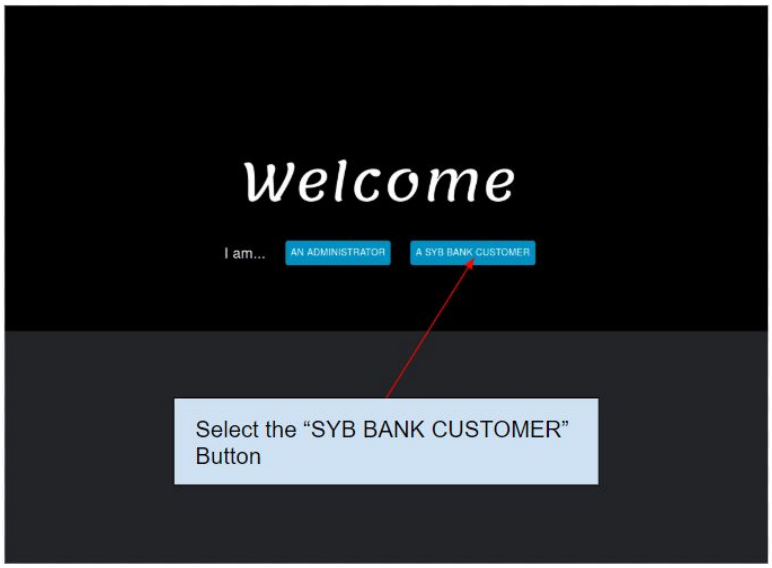
1. System Generation
2. Customer Functionality
    a. Create Account
    b. Login
    c. Initiate a Transaction
        i. Transfer
        ii. Deposit
        iii. Withdraw
    d. Transaction History
    e. Weekly Spending
    f. Create Bank Account
    g. Close Bank Account
3. Admin Functionality
    a. Login
    b. Modify Customer
    c. Review Transaction

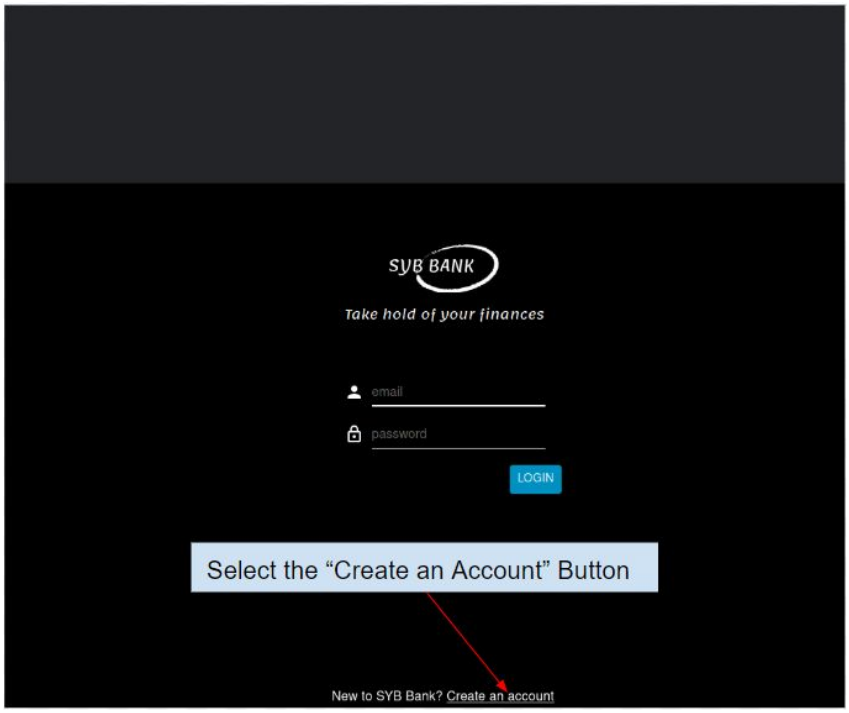
SYSTEM GENERATION

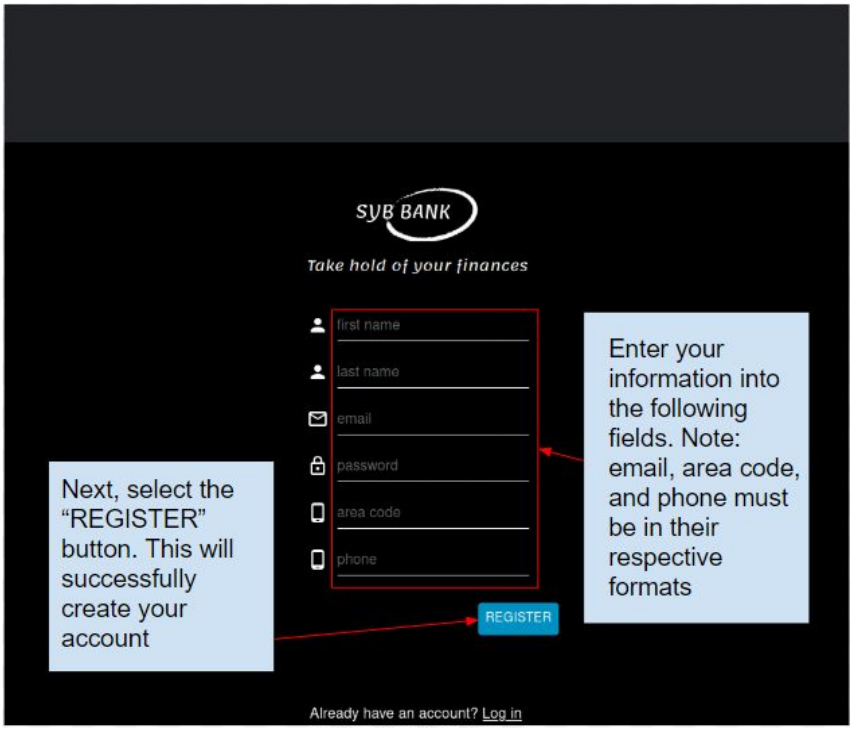
Instructions for generating the system can be found in our GitHub repository's README:
https://github.com/sbracellari/SYB-Bank/blob/master/README.md

CUSTOMER FUNCTIONALITY

a. **Create Account:**
   In order to access the services provided by SYB Bank, a customer must first create an account.

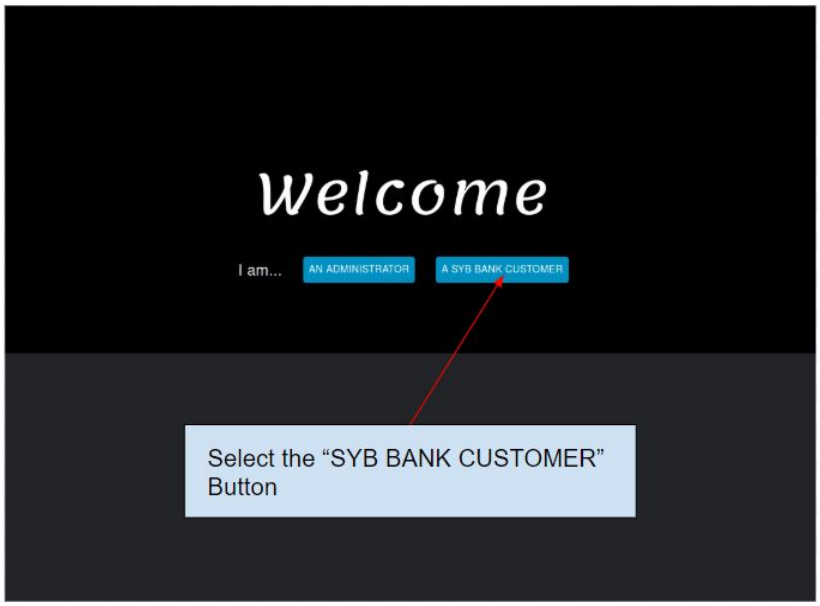

Welcome
I am...
AN ADMINISTRATOR
A SYB BANK CUSTOMER
Select the “SYB BANK CUSTOMER” Button

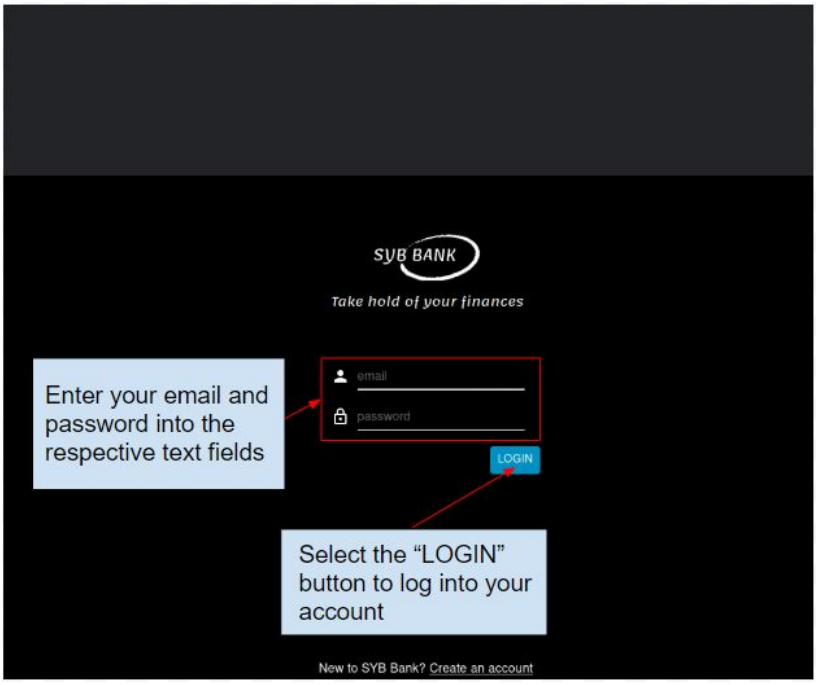

SYB BANK
Take hold of your finances
email
password
LOGIN
Select the “Create an Account” Button
New to SYB Bank? Create an account



b. Login:

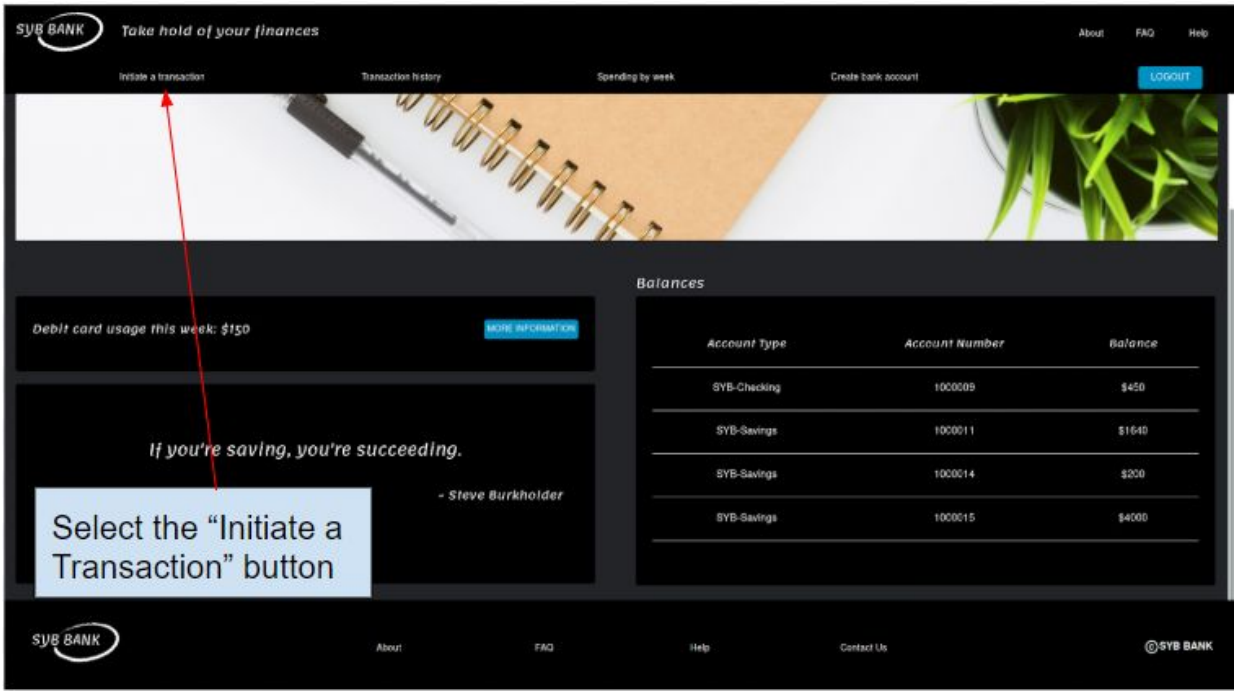
To view their accounts and access other functionality, the customer must first log into their account.





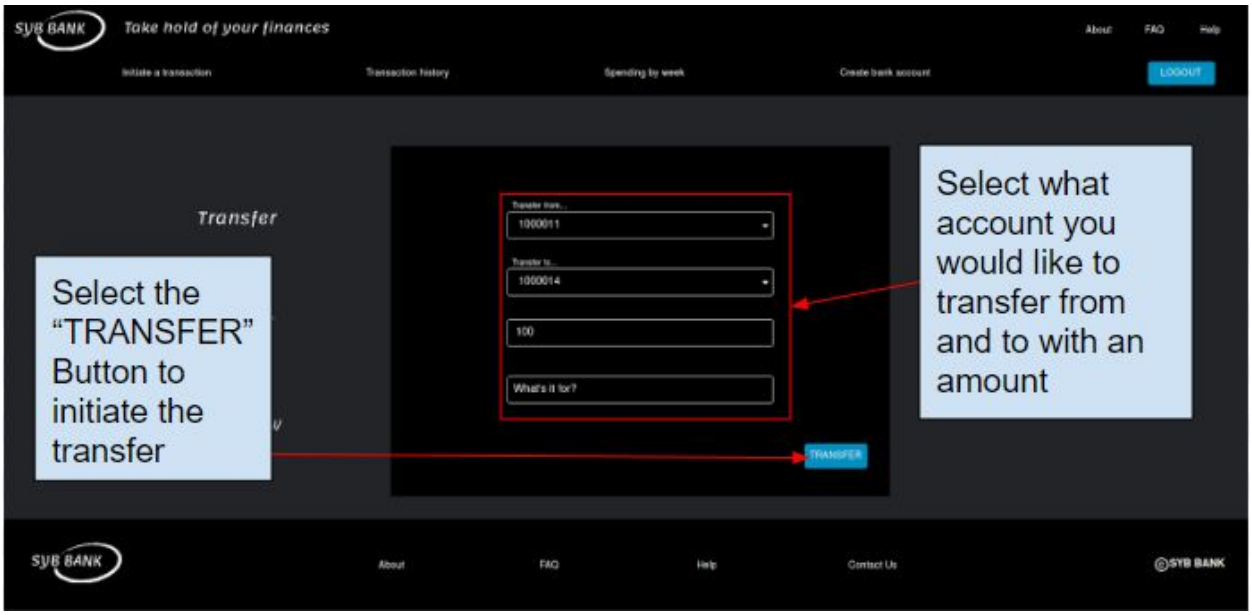
c. Initiate a Transaction:

In order to withdraw, deposit, or transfer money between accounts, the user must go to the “Initiate a Transaction” page from the home screen.



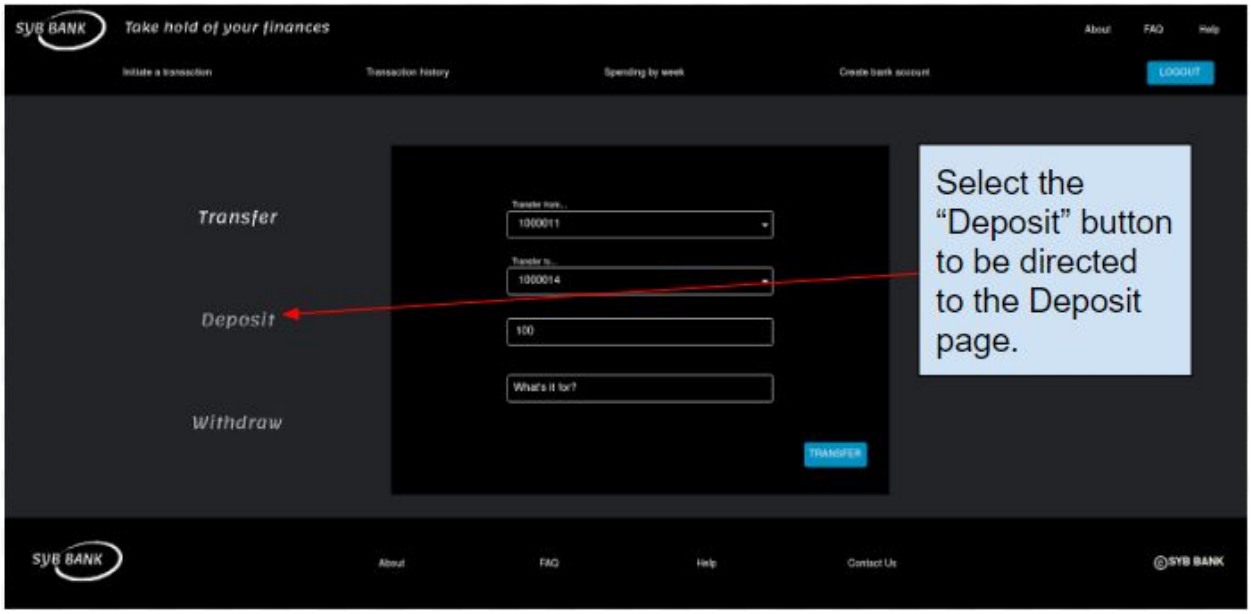
i. **Transfer:**

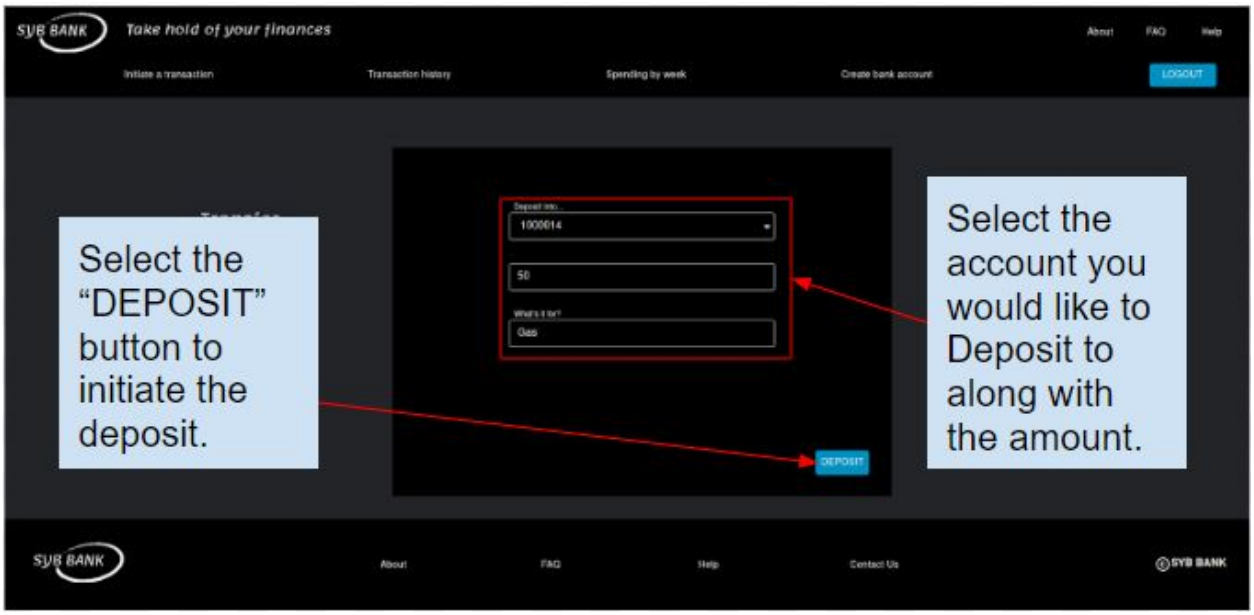
Upon selection of the “Initiate a Transaction” button, the customer will be brought to the transfer page by default.



ii. **Deposit:**

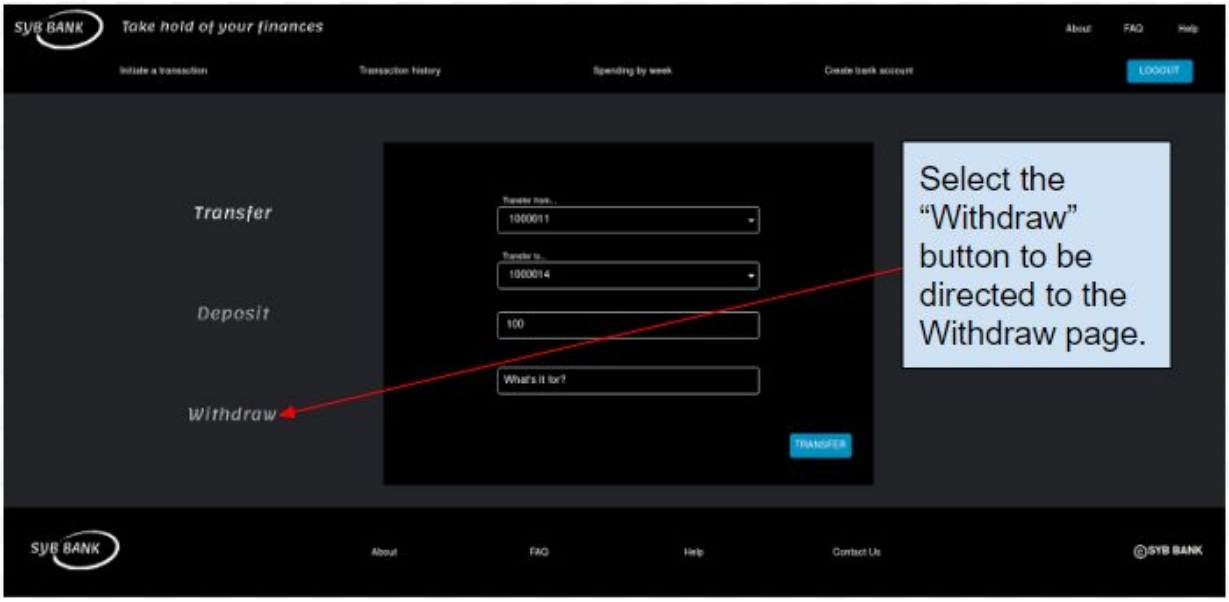
Upon selection of the “Initiate a Transaction” button, the customer will be brought to the transfer page by default. The user must select the “Deposit” button to make a deposit.

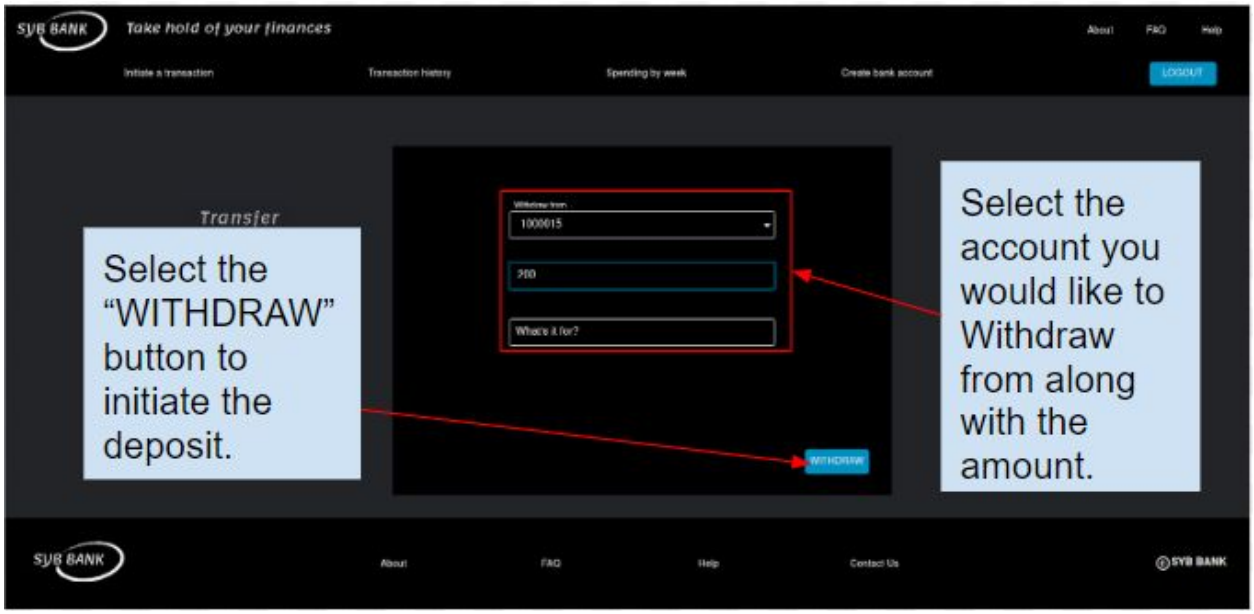




iii. Withdraw:

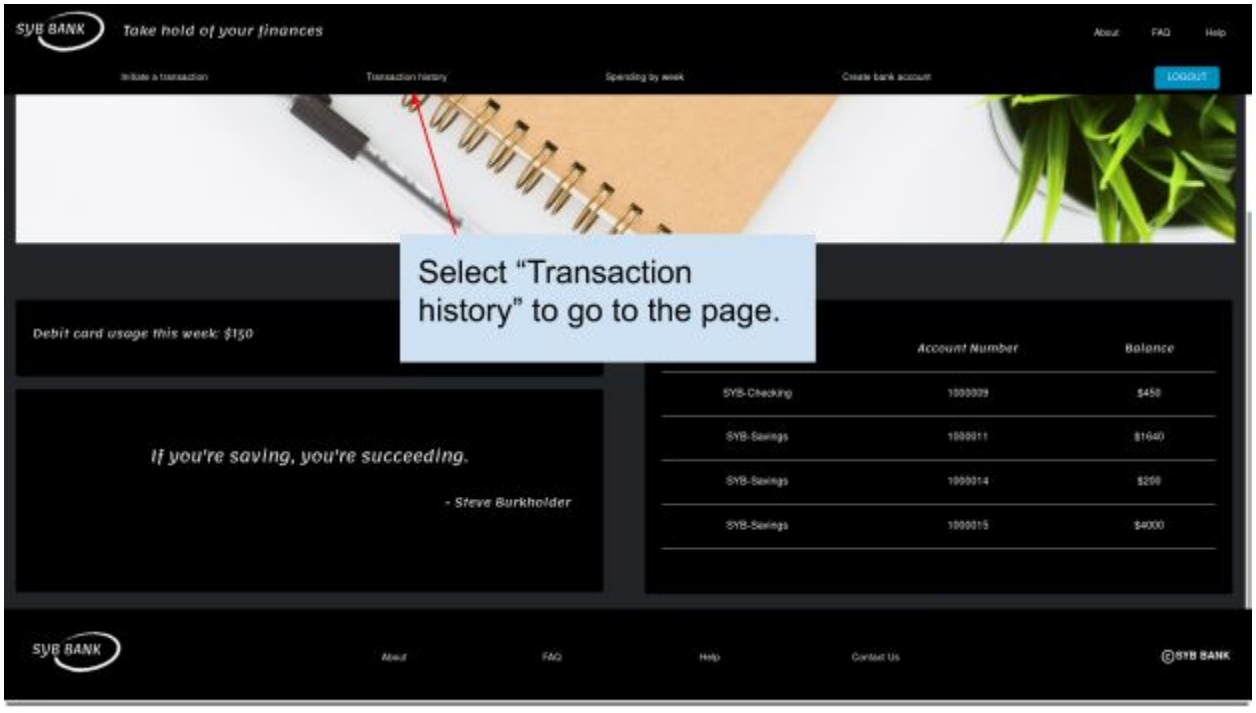
Upon selection of the "Initiate a Transaction" button, the customer will be brought to the transfer page by default. The user must select the "Withdraw" button to make a withdrawal.

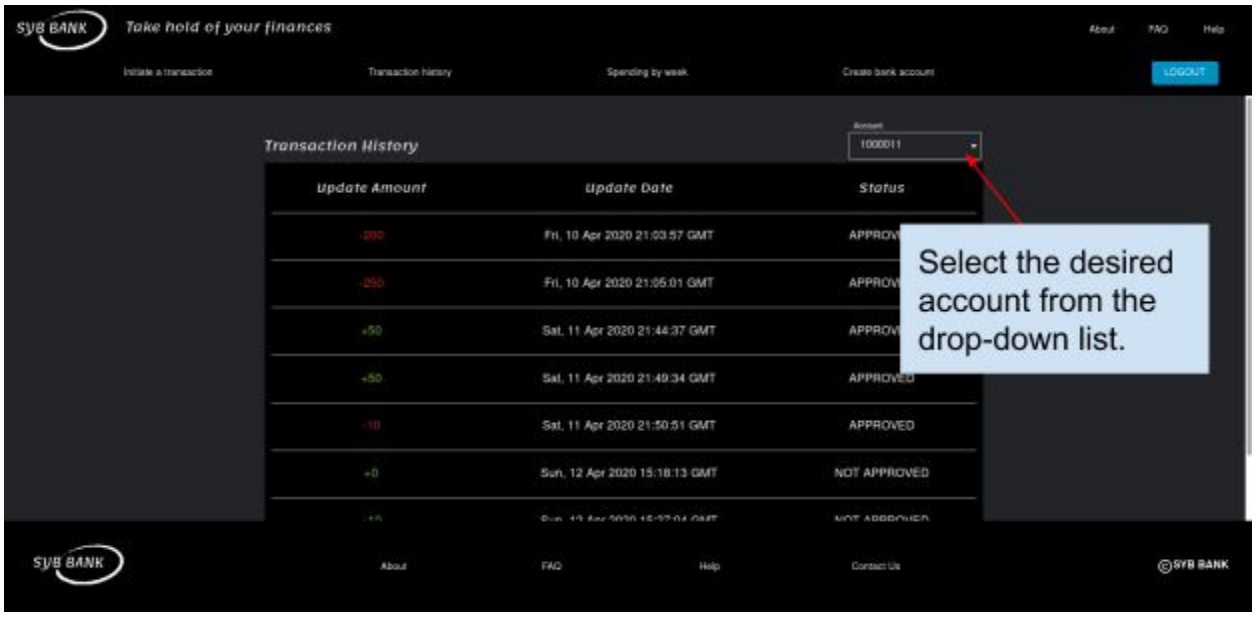




d. Transaction History:

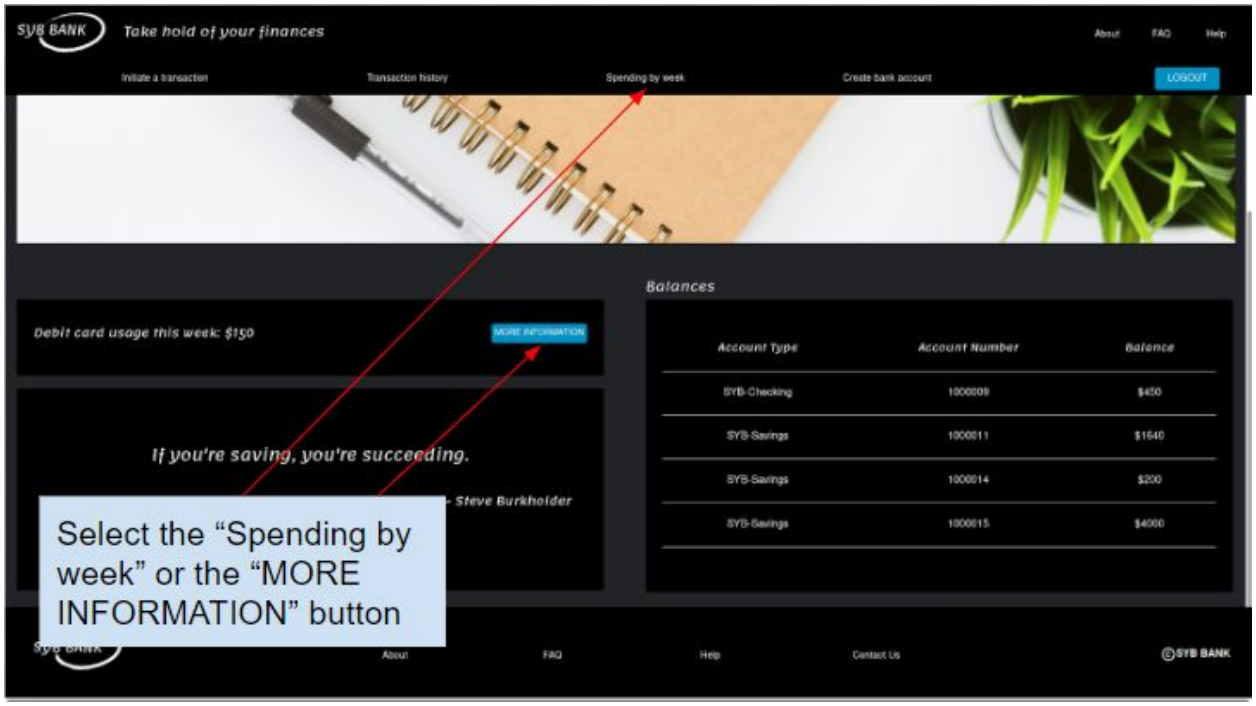
To view all transaction history for a bank account, the user must go to the "Transaction History" page.





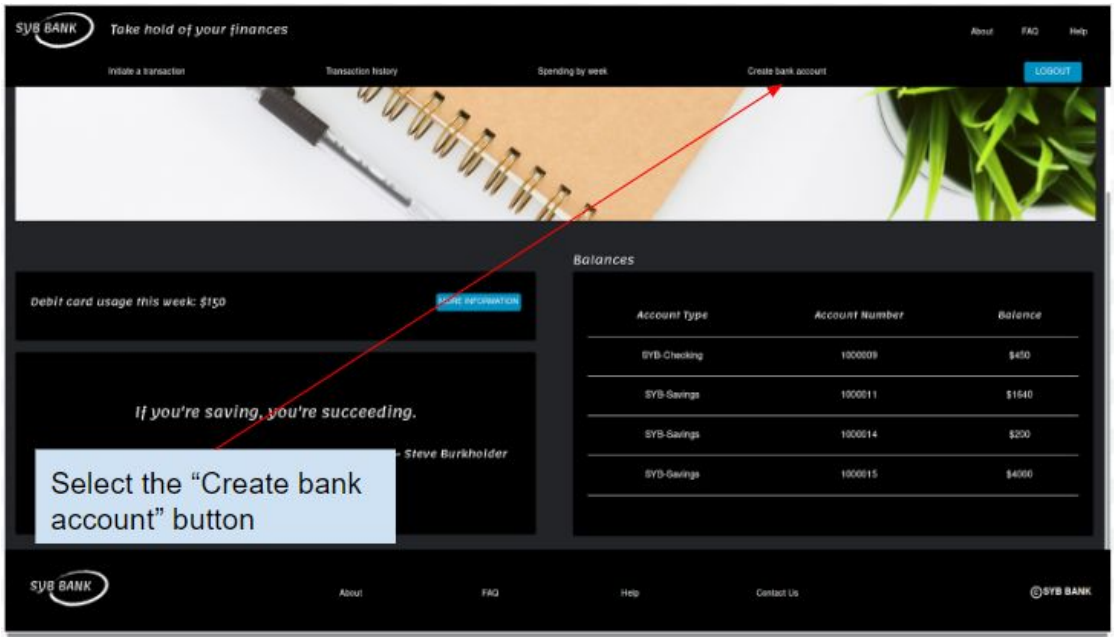
e. Weekly Spending:

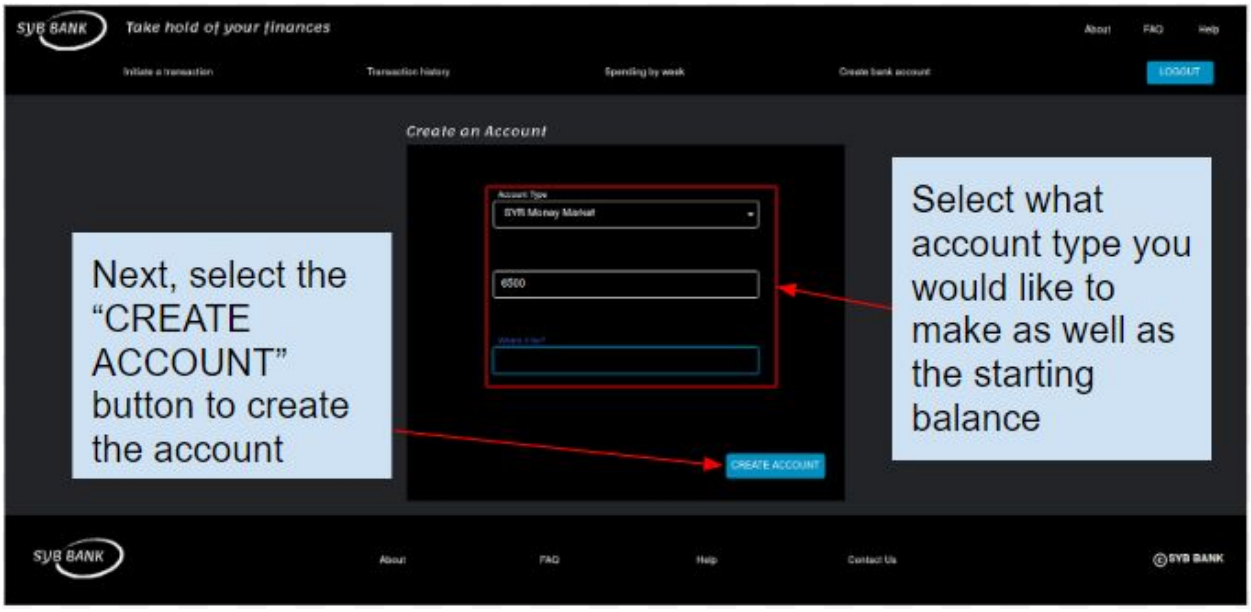
To view checking account activity for the week, the user must go to the "Spending by week" page.



f. Create Bank Account:

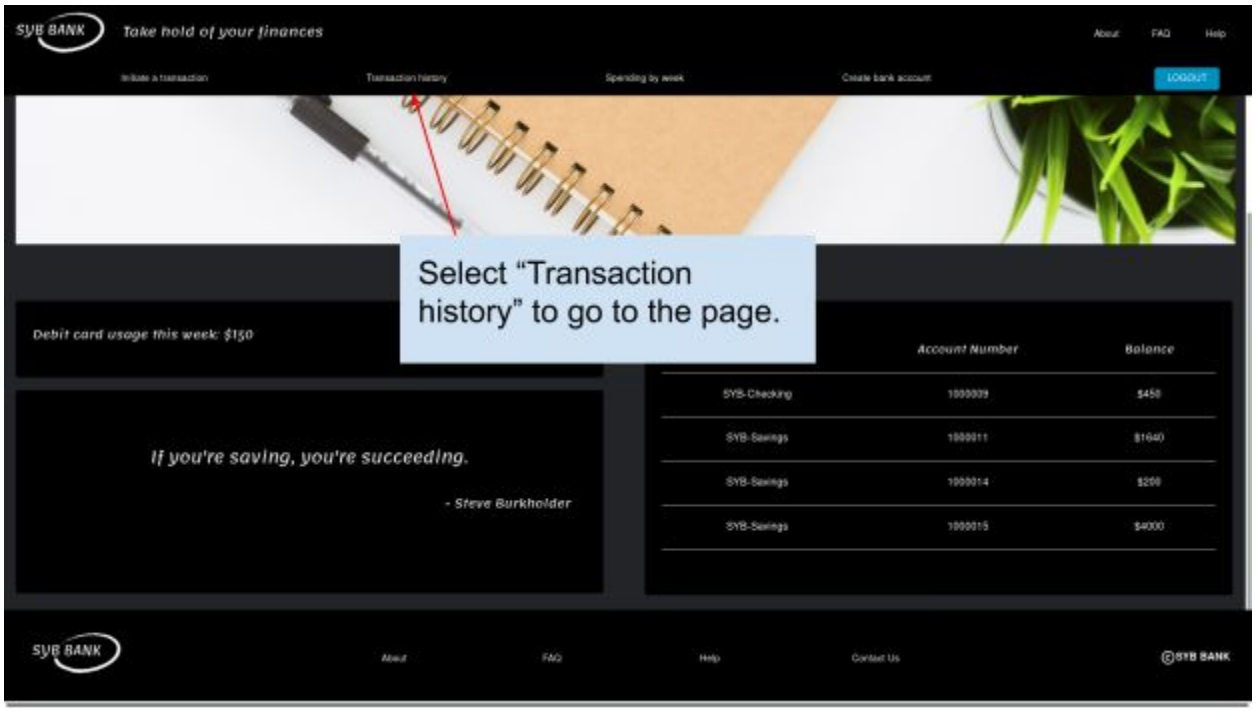
To create a Savings or Money Market account, the user must go to the “Create bank account” page. Note: When creating a Money Market account, your minimum balance must be 5000.

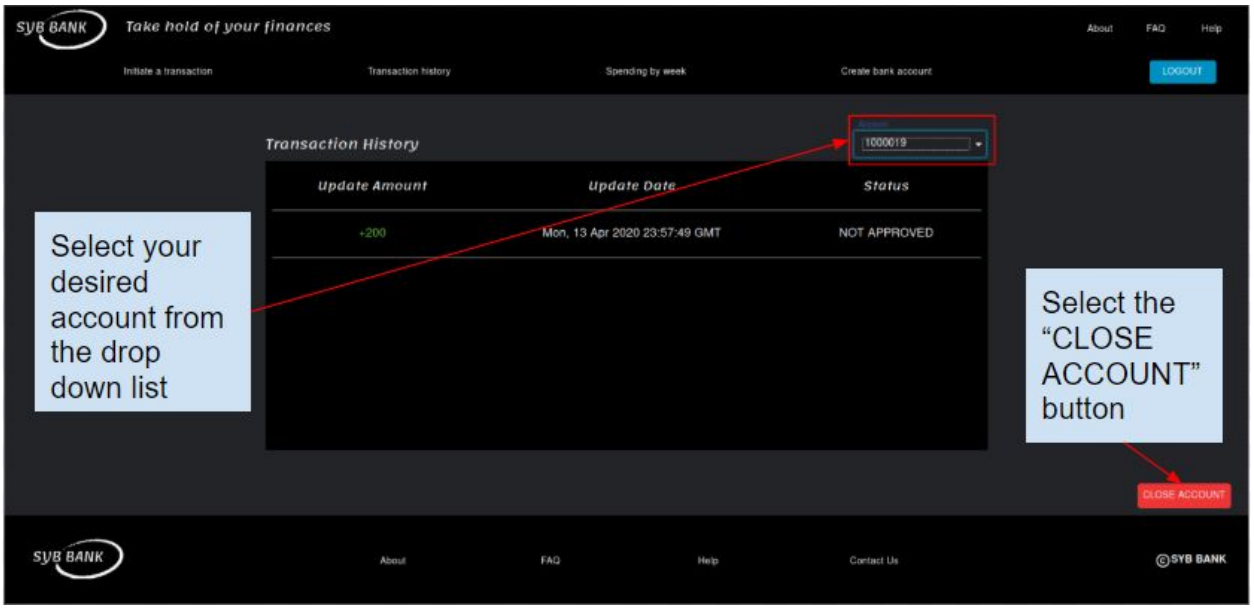


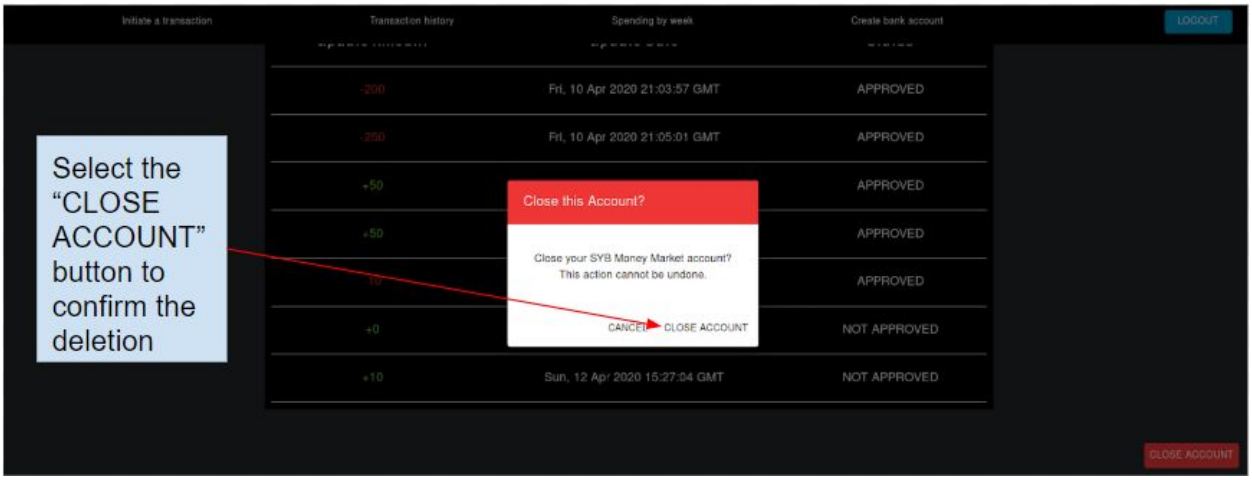


g. Close Bank Account:

In order to close a bank account (an account other than checking) you must go to the “Transaction history” page and select the account to be deleted.



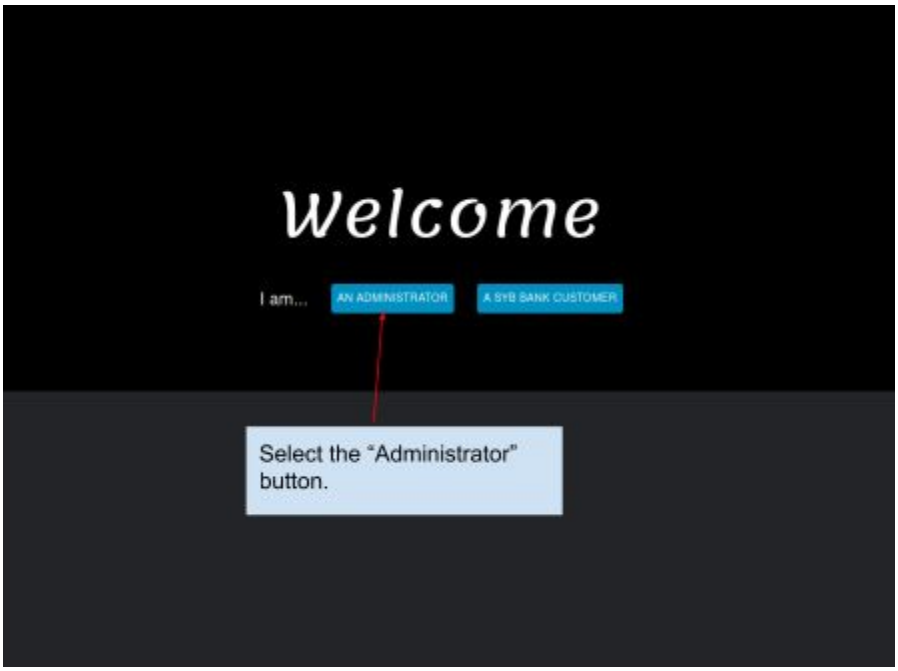


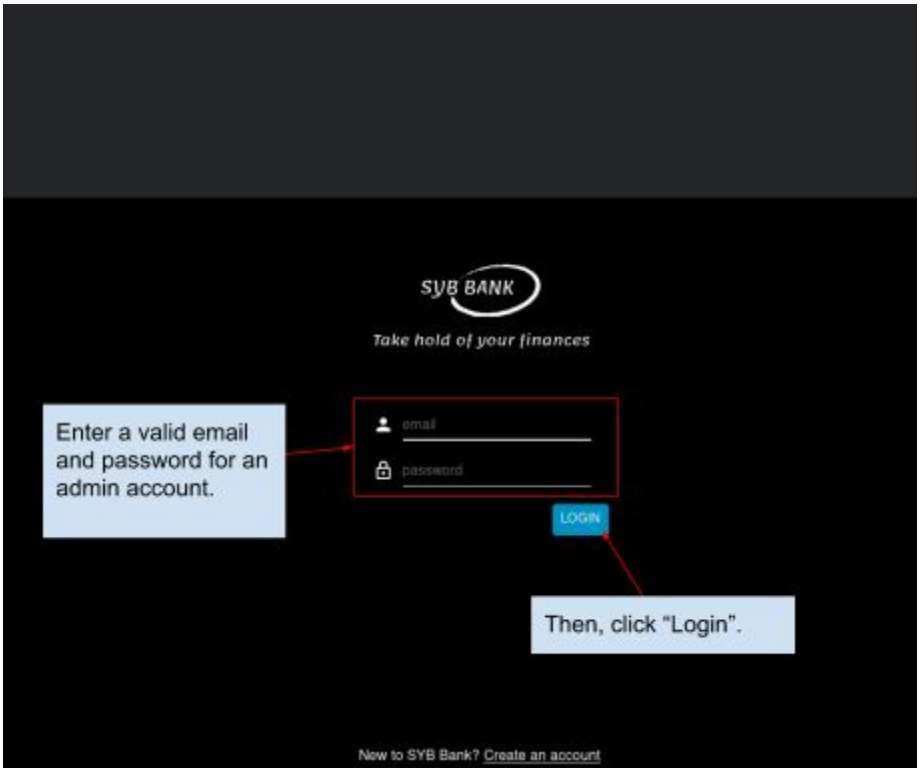


ADMIN FUNCTIONALITY

a. Login:

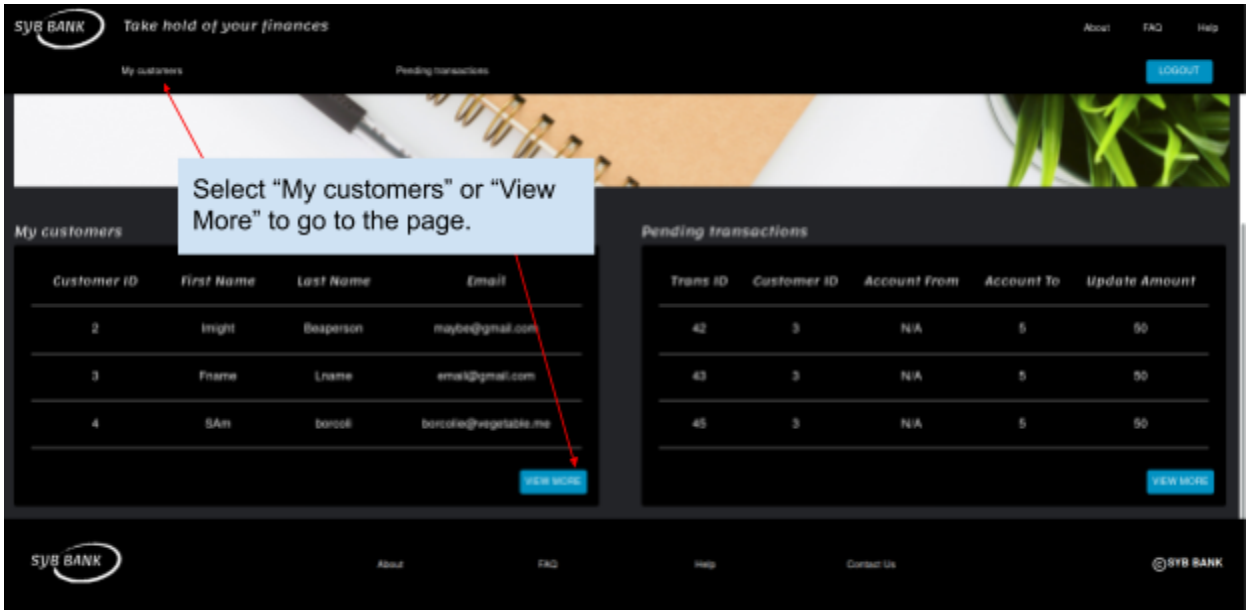
To view customer information and pending transactions, an employee must log into their account.

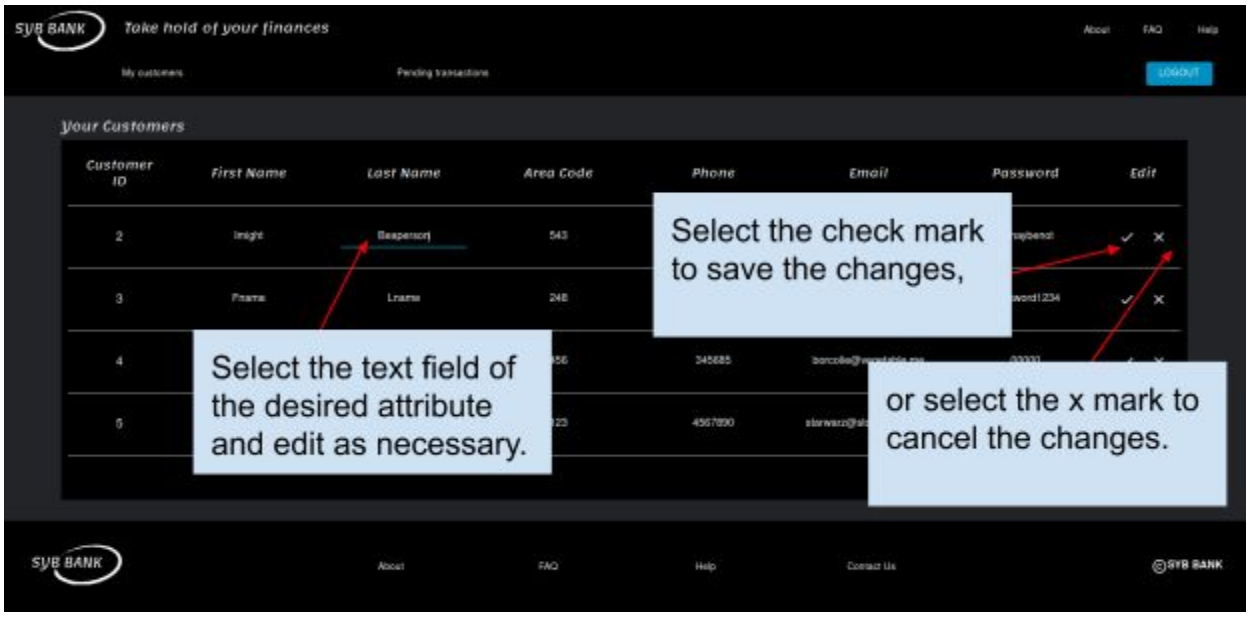




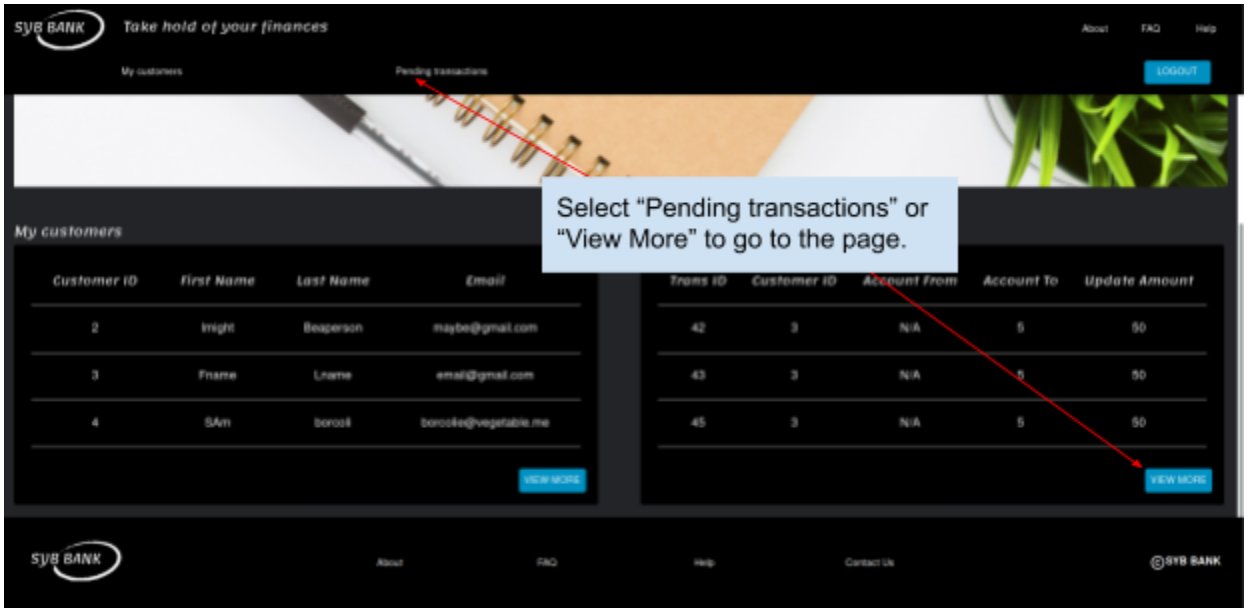
b. Modify Customer:

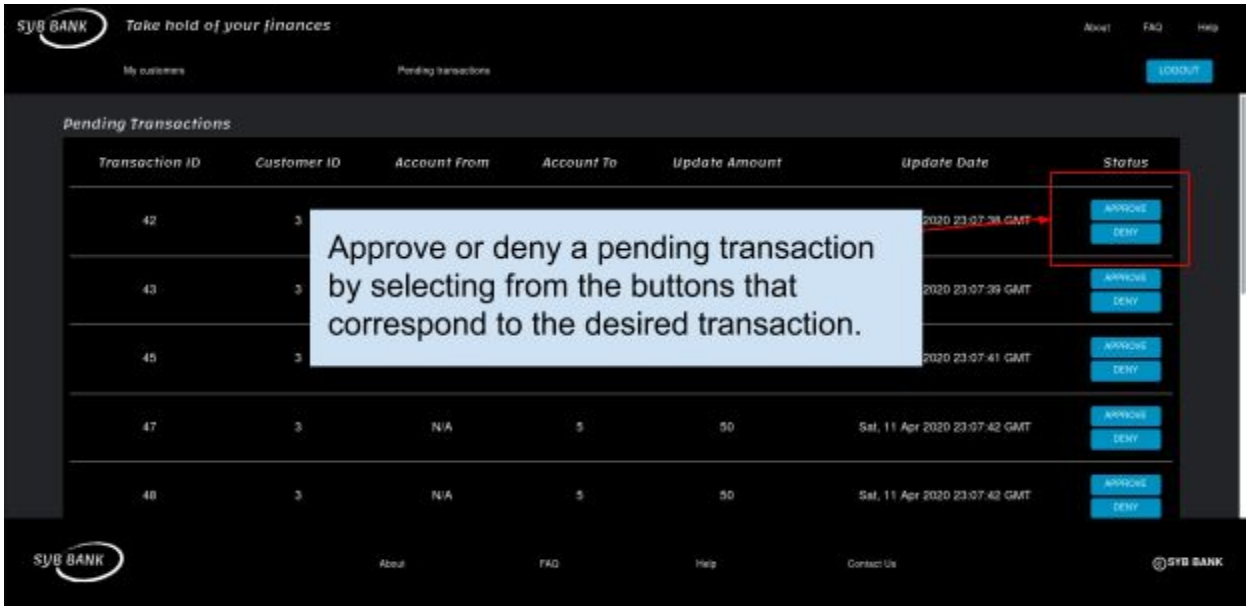
To modify customer information, such as first and last name, email address, or phone number, the employee must go to the "My customers" page.





c. **Review Transaction:**
To review pending transactions, the employee must go to the “Pending transactions” page.





Implementation Manual

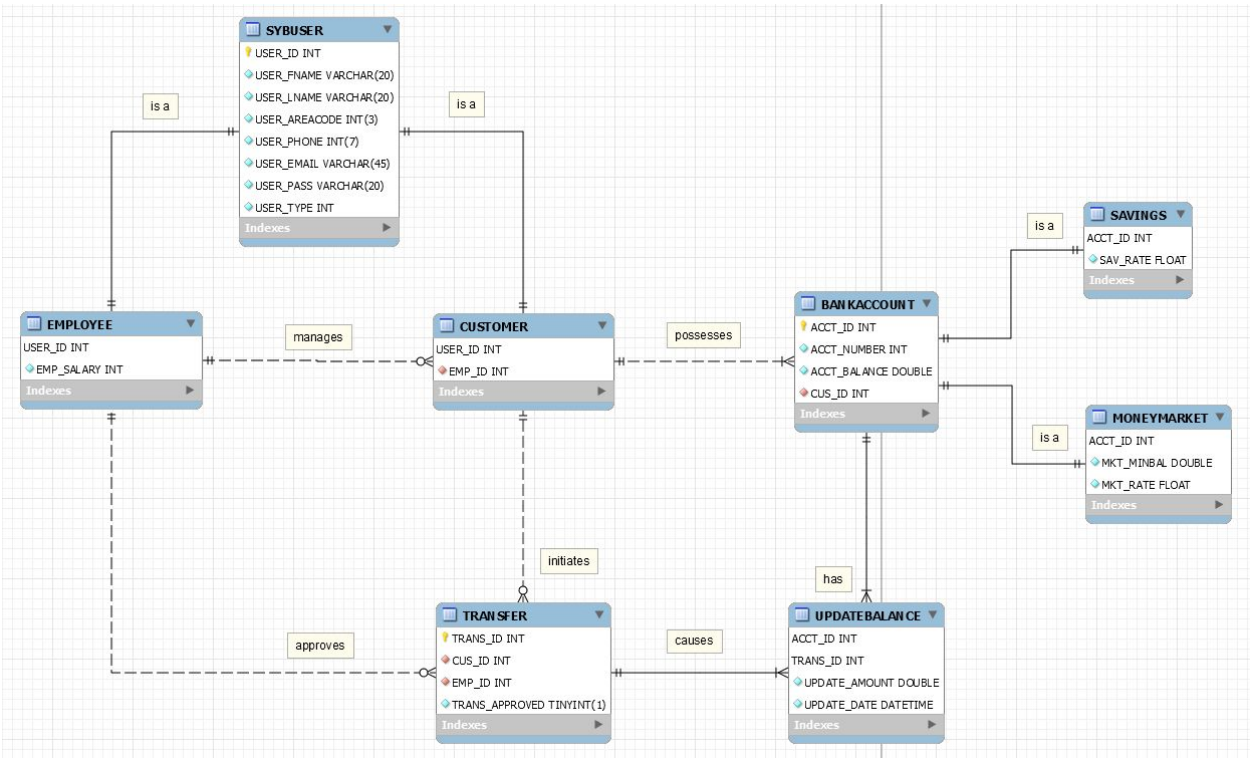
1. Problem Statement

A banking/accounting database system that allows customers to create accounts and make transactions such as deposits, withdrawals, and transfers, as well as allows accountants to manage customers and approve transactions.

2. System Requirements

The system will allow customers to create different types of bank accounts (checking, savings, etc.) and make transactions within those accounts (deposit, withdrawal, transfer, etc.). Additionally, the system will allow employees to manage customers (update/delete customer information) as well as approve or reject transactions.

3. Conceptual Database Design



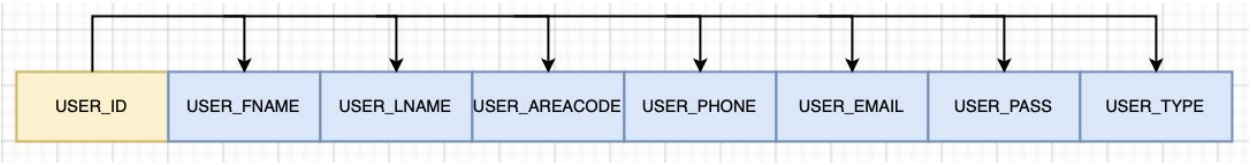
4. Functional Requirements

- The system shall allow customers to create a bank account
    - Tables: BANKACCOUNT, SAVINGS, MONEYMARKET
    - Input: account number, account balance, account type
    - Output: new bank account added to BANKACCOUNT table and SAVINGS table or MONEYMARKET table where applicable
- The system shall allow customers to delete a bank account
    - Tables: BANKACCOUNT, SAVINGS, MONEYMARKET
    - Input: account number
    - Output: bank account corresponding to account number will be deleted from BANKACCOUNT table and SAVINGS table or MONEYMARKET table where applicable
- The system shall allow customers to initiate transactions (update account balance)
    - Tables: TRANSFER, UPDATEBALANCE, BANKACCOUNT
    - Input: transaction amount, source account ID, target account ID, date
    - Output: new account balance
- The system shall allow customers to retrieve account balance
    - Tables: BANKACCOUNT

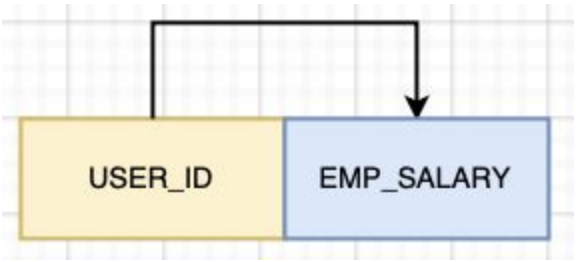
    - Input: account ID
    - Output: account balance
- The system shall allow customers to retrieve transaction information
    - Tables: UPDATEBALANCE
    - Input: account ID, transaction ID
    - Output: transaction type, transaction amount, source account ID, target account ID, date
- The system shall allow employees to manage customers (add, update, and delete customer information)
    - Tables: CUSTOMER
    - Input: customer first name, customer last name, customer area code, customer phone, customer email address, customer password
- The system shall allow employees to approve transactions (update transactions)
    - Tables: TRANSFER
    - Input: transaction ID, transaction approved

5. Logical Database Design

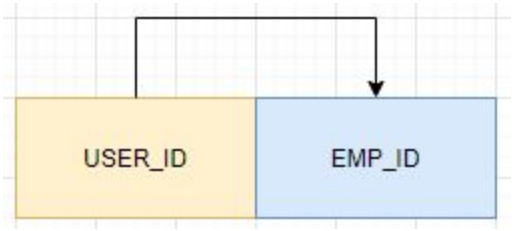
SYBUSER (
USER_ID INT (Auto Incremented, Not Null, Unique),
USER_FNAME VARCHAR(20) (First Name, Not Null),
USER_LNAME VARCHAR(20) (Last Name, Not Null),
USER_AREACODE INT(3) (Area Code for Phone Number, Not Null),
USER_PHONE INT(7) (Phone Number not including Area Code, Not Null),
USER_EMAIL VARCHAR(45) (User Email, Used for Login, Not Null, Unique),
USER_PASS VARCHAR(20) (User Password, Used for Login, Not Null), USER_TYPE INT (Subtype Discriminator, Not Null, 0=employee, 1=customer))
PRIMARY KEY: USER_ID



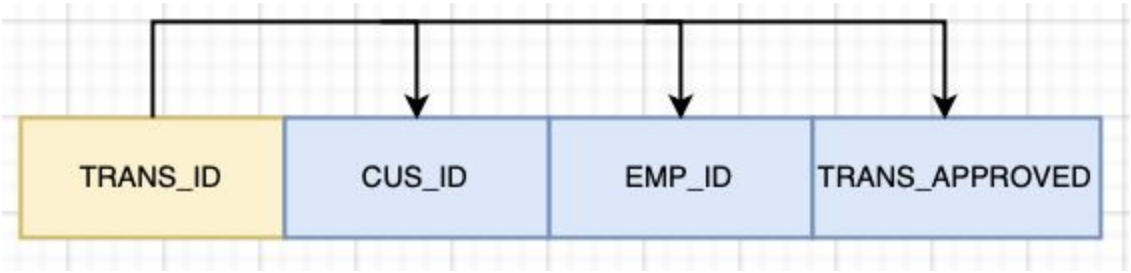
EMPLOYEE (
USER_ID INT (Not Null, Unique, From USER Table),
EMP_SALARY INT (Shows Employee Yearly Salary, Not Null))
PRIMARY KEY: USER_ID
FOREIGN KEY: USER_ID REFERENCES USER



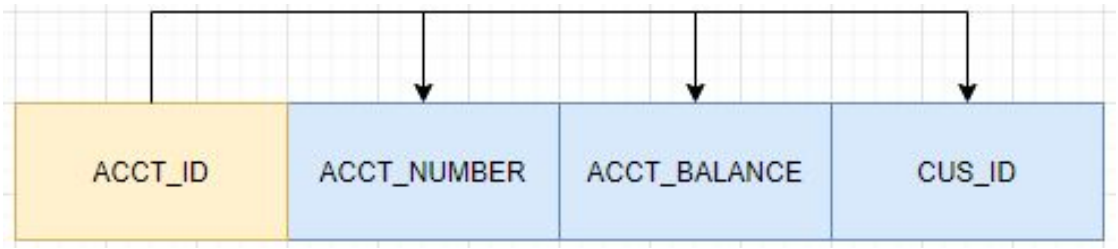
CUSTOMER (
USER_ID INT (Not Null, Unique, From USER Table),
EMP_ID INT (ID for Employee that manages this user, Not Null))
PRIMARY KEY: USER_ID
FOREIGN KEY: USER_ID REFERENCES USER
FOREIGN KEY: EMP_ID REFERENCES USER_ID IN EMPLOYEE



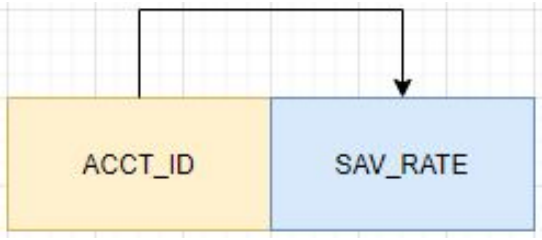
TRANSFER (
TRANS_ID INT (Auto Incrementing, Not Null, Unique),
CUS_ID INT (ID of customer that initiated the transfer, Not Null),
EMP_ID INT (ID of employee that reviews the transfer, Not Null),
TRANS_APPROVED TINYINT(1) (Status of the transfer, Default=0))
PRIMARY KEY: TRANS_ID
FOREIGN KEY: CUS_ID REFERENCES USER_ID IN CUSTOMER
FOREIGN KEY: EMP_ID REFERENCES USER_ID IN EMPLOYEE



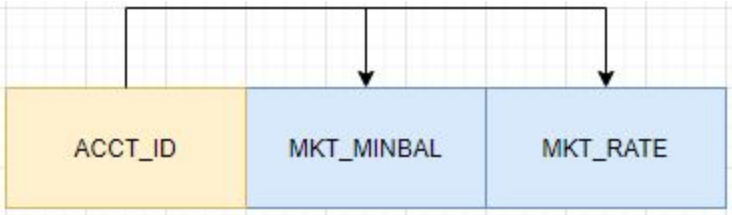
BANKACCOUNT (
ACCT_ID INT (Auto Incrementing, Not Null, Unique),
ACCT_NUMBER INT (Routing number for the account, Not Null, Unique),
ACCT_BALANCE DOUBLE (Current Account Balance, Not Null, Default = 0.00),
CUS_ID INT (ID of customer who owns account, Not Null))
PRIMARY KEY: ACCT_ID
FOREIGN KEY: CUS_ID REFERENCES USER_ID IN CUSTOMER



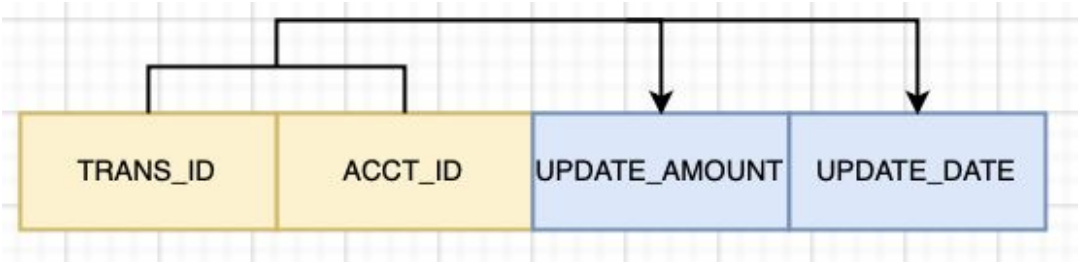
SAVINGS (
ACCT_ID INT (Not Null, Unique),
SAV_RATE FLOAT (Interest rate for savings account, Not Null, Default=2.0))
PRIMARY KEY: ACCT_ID
FOREIGN KEY: ACCT_ID REFERENCES ACCOUNT



MONEYMARKET (
ACCT_ID INT (Not Null, Unique),
MKT_MINBAL DOUBLE (Minimum balance for account, Not Null, Default=2500),
MKT_RATE FLOAT (Interest rate for mm account, Not Null, Default=2.5))
PRIMARY KEY: ACCT_ID
FOREIGN KEY: ACCT_ID REFERENCES ACCOUNT



UPDATEBALANCE (
ACCT_ID INT (Not Null, Unique),
TRANS_ID INT (Not Null, Unique),
UPDATE_AMOUNT DOUBLE (Amount deposited or withdrawn, Not Null),
UPDATE_DATE DATETIME (Date of transfer, Not Null))
PRIMARY KEY: ACCT_ID
PRIMARY KEY: TRANS_ID
FOREIGN KEY: ACCT_ID REFERENCES ACCOUNT
FOREIGN KEY: TRANS_ID REFERENCES TRANSFER



Entities	**Attributes**	**Key**
SYBUSER	USER_ID	PK
	USER_FNAME	
	USER_LNAME	
	USER_AREACODE	
	USER_PHONE	
	USER_EMAIL	
	USER_PASS	
	USER_TYPE	
EMPLOYEE	USER_ID	PK
	EMP_SALARY	
CUSTOMER	USER_ID	PK
	EMP_ID	FK
TRANSFER	TRANS_ID	PK
	CUS_ID	FK
	EMP_ID	FK
	TRANS_APPROVED	
BANKACCOUNT	ACCT_ID	PK
	ACCT_NUMBER	
	ACCT_BALANCE	
	CUS_ID	FK

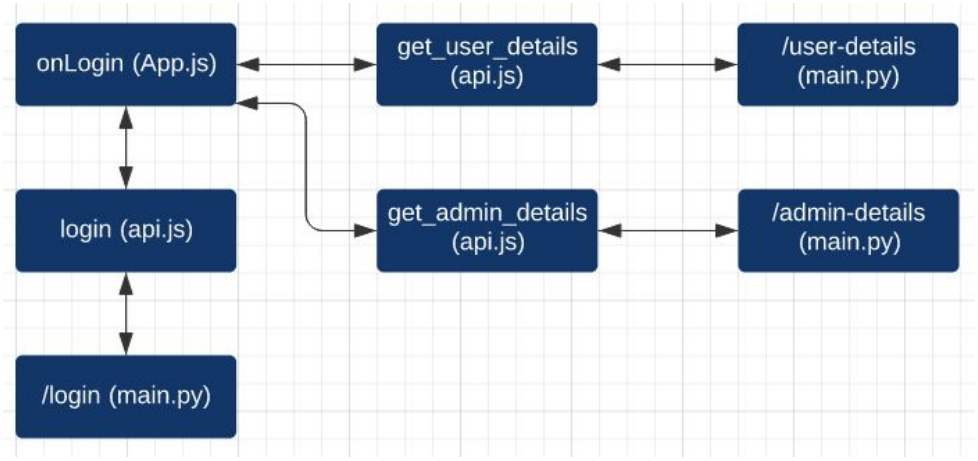
SAVINGS	ACCT_ID	PK, FK
	SAV_RATE	
MONEYMARKET	ACCT_ID	PK, FK
	MKT_MINBAL	
	MKT_RATE	
UPDATEBALANCE	ACCT_ID	PK, FK
	TRANS_ID	PK, FK
	UPDATE_AMOUNT	
	UPDATE_DATE	

6. Application Programs

The frontend of this project was written in ReactJS, while the backend was written in Python and used Flask.

Below will show a flow chart of each interaction that can be taken in the application, along with screenshots of the critical methods that are associated with those interactions.

1. Login (administrator or user)



```
// is called when a user hits the login button.
// will call the login method in the api.js with the email
onLogin = () => {
  login(this.state.email).then((data) => {
    // setting password and user ID to data in the fetch response
    let password = data[0].USER_PASS
    let user_id = data[0].USER_ID

    // checking that the returned password is the same as the entered password.
    // if they are the same, the user is logged in and directed to their
    // respective home page (administrator or customer)
    if (this.state.password === password) {
      this.setState({
        logged_in: true,
        login_err: false
      })

      // upon successful login, the user's ID is saved in local storage for the
      // backend to read and use later
      localStorage.setItem('user_id', user_id)

      // if the user is an administrator, call the get_admin_details method
      // in api.js which will retrieve the admin's 2 core pieces of information -
      // their managed customers and their pending transactions
      if (this.state.is_admin) {
        get_admin_details().then((admin_data) => {
          this.setState({
            pending_transactions: admin_data.PENDING_TRANSACTIONS,
            customers: admin_data.CUSTOMERS,
          })
        })
      } else {
        // if the user is a customer, call the get_user_details method in
        // api.js, which will retieve 4 of the 5 core pieces of the user's information -
        // their accounts, weekly transactions for their checking account, their account
        // balances, and their debit card usage for their checking account
        get_user_details().then((user_data) => {
          this.setState({
            accounts: user_data.ACCOUNTS,
            weekly_spending: user_data.WEEKLY_TRANSACTIONS,
            balances: user_data.BALANCES,
            debit_card_usage: user_data.DEBIT_CARD_USAGE.DEBIT
          })
        })
      }

      // if the password does not match, do not log them in. instead, set the login error to
      // true, which will notify them that either their email or password was incorrect
    } else {
      this.setState({ login_err: true })
    }
  })
}
```

```
export const login = async (email) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/login', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        email: email
      })
    })
    const data = await response.json()
    return data
  } catch (err) {
    console.log("ERR ", err)
    return err
  }
}
```

```
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL SELECT
# METHOD PURPOSE: login to account
# INPUT PARAMETERS: only params needed for the SQL query are email
# OUTPUT: will return the user's password and ID if the email was valid. If valid and the user
# entered in a correct password, they will be directed to their home page
@app.route('/syb-bank/login', methods=['POST']) # POST request since the email is sent in the body
def login():

    # getting the email from the fetch request in the frontend
    email = request.json.get('email')

    data = []

    # call the login procedure with the email as an argument
    cur.callproc('login', [email])

    # iterate through the result set
    for set in cur.stored_results():
        for row in set:
            data.append(dict(zip(set.column_names,row))) # append each row of the result set to an array
    return jsonify(data) # jsonify the array so the frontend will accept it
```

```
export const get_user_details = async () => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/user-details', {
      method: 'GET',
      headers: {
        'Accept': 'application/json'
      }
    })

    const data = await response.json()
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL SELECT
# METHOD PURPOSE: to grab the 4 main components of the user's information -
# their weekly spending, their account balances, their current debit card usage,
# and a list of all of their active accounts
# INPUT PARAMETERS: only param needed is user_id
# OUTPUT: displays user's balances, weekly transactions, debit card usage, and a list of their active accounts
@app.route('/syb-bank/user-details', methods=['GET']) # GET request since we are not manipulating the DB
def get_user_details():

    # read the user's ID from local storage using the scraper in scraper.py
    user_id = s.read_local_storage()

    balances, accounts, weekly_transactions = [], [], []
    debit_card_usage = 0

    # call getBalances procedure with user ID
    cur.callproc('getBalances', [user_id])

    # iterate through result set
    for set in cur.stored_results():
        for row in set:
            balances.append(dict(zip(set.column_names,row))) # append each row to an array

    # call the getAccounts procedure with user ID
    cur.callproc('getAccounts', [user_id])

    # iterate through the result set
    for set in cur.stored_results():
        for row in set:
            accounts.append(dict(zip(set.column_names,row))) # append each row to an array

    # call the weeklyTransactions procedure with user ID
    cur.callproc('weeklyTransactions', [user_id])

    # iterate through the result set
    for set in cur.stored_results():
        for row in set:
            weekly_transactions.append(dict(zip(set.column_names,row))) # append each row to an array

    # call the debitCardUsage procedure with user ID
    cur.callproc('debitCardUsage', [user_id])

    # iterate through the result set
    for set in cur.stored_results():
        for row in set:
            debit_card_usage = dict(zip(set.column_names,row)) # append each row to an array

    # jsonify the four arrays so the frontend will accept the return value
    return jsonify({
        'BALANCES': balances,
        'ACCOUNTS': accounts,
        'WEEKLY_TRANSACTIONS': weekly_transactions,
        'DEBIT_CARD_USAGE': debit_card_usage
    })
```

```javascript
export const get_admin_details = async () => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/admin-details', {
      method: 'GET',
      headers: {
        'Accept': 'application/json'
      }
    })

    const data = await response.json()
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: ADMINISTRATOR
# PREDICTED QUERIES TO BE USED: MySQL SELECT
# METHOD PURPOSE: to grab the 2 main components of the admin's information -
# their managed customers and their pending transactions
# INPUT PARAMETERS: only param needed is user_id
# OUTPUT: displays  admin's customers and pending transactions
@app.route('/syb-bank/admin-details', methods=['GET']) # GET request since we are not manipulating the DB
def get_admin_details():

    # read the user's ID from local storage using the scraper in scraper.py
    user_id = s.read_local_storage()

    customers, pending_transactions = [], []

    # call the getCustomers procedure with user ID
    cur.callproc('getCustomers', [user_id])

    # iterate through the result set
    for set in cur.stored_results():
        for row in set:
            customers.append(dict(zip(set.column_names,row))) # append each row to an array

    # call the viewTransactionsAdmin procedure with user ID
    cur.callproc('viewTransactionsAdmin', [user_id])

    # iterate through the result set
    for set in cur.stored_results():
        for row in set:
            pending_transactions.append(dict(zip(set.column_names,row))) # append each row to an array

    # jsonify the result so it is accepted by the frontend
    return jsonify({
        'CUSTOMERS': customers,
        'PENDING_TRANSACTIONS': pending_transactions
    })
```

2. Register (user)


```
// when the customer creates a new SYB Bank account, call the register method in api.js with
// first name, last name, area code, phone, email and password. the fetch request will return
// true upon successful registration and the user will be redirected to the login page, and
// false otherwise
onRegister = () => {
  register(
    this.state.first_name,
    this.state.last_name,
    this.state.email,
    this.state.password,
    this.state.area_code,
    this.state.phone
  ).then((data) => {

    if (!data) {
      this.setState({
        logged_in: false,
        register_err: true,
        is_user: false,
        is_admin: false
      })
      return
    }

    this.setState({
      register_err: false,
      is_user: true,
      is_admin: false
    })
  })
}
```

```javascript
export const register = async (first_name, last_name, email, password, area_code, phone) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/register', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        email: email,
        password: password,
        first_name: first_name,
        last_name: last_name,
        area_code: area_code,
        phone: phone
      }),
    })

    const data = response.ok
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL INSERT
# METHOD PURPOSE: create a user account
# INPUT PARAMETERS: params needed for the SQL query are first_name, last_name, area_code, phone, email, and password
# OUTPUT: a new row will be inserted in USER table, and user will be taken to the login page
@app.route('/syb-bank/register', methods=['POST']) # POST request since we are manipulating the DB
def register():

    # get request attributes from the frontend
    first_name = request.json.get('first_name')
    last_name = request.json.get('last_name')
    area_code = request.json.get('area_code')
    phone = request.json.get('phone')
    email = request.json.get('email')
    password = request.json.get('password')

    # since nothing is returned, there is no result set to iterate through
    # this will just return the network response (200, 404, 500, etd.)
    return cur.callproc('createUserAccount', [first_name, last_name, area_code, phone, email, password])
```

3. View transaction history (user)


```
// handle dropdown input for accounts (when viewing transaction history or
// choosing source/target accounts for transactions)
handleAccChange = event => {
  this.setState({ account_num: event.target.value })

  // on each change call the get_transaction_history method in api.js which
  // returns the transaction history for that account
  get_transaction_history(event.target.value).then(data => {
    this.setState({ transaction_history: data.TRANSACTION_HISTORY })
  })
}
```

```
export const get_transaction_history = async (account_num) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/transaction-history', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        account_num: account_num
      })
    })
    const data = await response.json()
    return data
  } catch (err) {
    return err
  }
}
```

```
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL SELECT
# METHOD PURPOSE: to grab the transaction history of a specified account
# INPUT PARAMETERS: only params needed are user_id and account_number
# OUTPUT: displays user's transaction history for a specified account
@app.route('/syb-bank/transaction-history', methods=['POST']) # POST request since we are sending a request body
def get_transaction_history():

    # read the user's ID from local storage using the scraper in scraper.py
    user_id = s.read_local_storage()

    # get request attributes from the frontend
    account_num = request.json.get('account_num')

    transaction_history = []

    # call the transactionHistory procedure with user ID and account number
    cur.callproc('transactionsHistory', [user_id, account_num])

    # iterate through the result set
    for set in cur.stored_results():
        for row in set:
            transaction_history.append(dict(zip(set.column_names,row))) # append each row to an array

    # jsonify the result so the frontend will accept it
    return jsonify({
        'TRANSACTION_HISTORY': transaction_history
    })
```

4. Create a bank account (user)


```
// on bank account creation, will check if the starting balance is negative. if its not,
// it will call the create_bank_account method in api.js with the account type and starting
// balance. if the starting balance is negative, the user will not be able to create the account
createBankAccount = () => {
  if (this.state.acc_type === '3' && this.state.starting_balance < 5000) {
      this.setState({
        acc_err: true,
        accSnackbar: true
      })
    return
  }

  if (!this.state.amt_err) {
    create_bank_account(
    this.state.acc_type,
    this.state.starting_balance
  ).then(data => {
    this.setState({
      acc_err: data.length === 0,
      accSnackbar: true,
      accounts: data.ACCOUNTS,
      balances: data.BALANCES
     })
  })
  }
}
```

```
export const create_bank_account = async (acc_type, starting_balance) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/create-bank-account', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        acc_type: acc_type,
        starting_balance: starting_balance
      })
    })

    const data = await response.json()
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL INSERT
# METHOD PURPOSE: create a new bank account
# INPUT PARAMETERS: only params needed for the SQL query are acc_type, starting balance, and user_id all other fields
# of the ACCOUNT table are default vals and will be generated on bank account creation
# OUTPUT: ACCOUNT table will be updated, i.e. a new row will be inserted
@app.route('/syb-bank/create-bank-account', methods=['POST']) # POST request since we are manipulating the DB
def create_bank_account():

    # read the user's ID from local storage using the scraper in scraper.py
    user_id = s.read_local_storage()

    # get request attributes from the frontend
    acc_type = request.json.get('acc_type')
    starting_balance = request.json.get('starting_balance')

    accounts, balances = [], []

    # call createBankAccount procedure
    cur.callproc('createBankAccount', [acc_type, starting_balance, user_id])

    # call getBalances procedure with user ID
    cur.callproc('getBalances', [user_id])

    # iterate through result set
    for set in cur.stored_results():
        for row in set:
            balances.append(dict(zip(set.column_names,row))) # append each row to an array

    # call the getAccounts procedure with user ID
    cur.callproc('getAccounts', [user_id])

    # iterate through the result set
    for set in cur.stored_results():
        for row in set:
            accounts.append(dict(zip(set.column_names,row))) # append each row to an array

    # jsonify the four arrays so the frontend will accept the return value
    return jsonify({
        'BALANCES': balances,
        'ACCOUNTS': accounts
    })
```

5. Delete a bank account (user)


```
// on bank account creation, will check if the starting balance is negative. if its not,
// it will call the create_bank_account method in api.js with the account type and starting
// balance. if the starting balance is negative, the user will not be able to create the account
createBankAccount = () => {
  if (this.state.acc_type === '3' && this.state.starting_balance < 5000) {
      this.setState({
        acc_err: true,
        accSnackbar: true
      })
    return
  }

  if (!this.state.amt_err) {
    create_bank_account(
    this.state.acc_type,
    this.state.starting_balance
  ).then(data => {
    this.setState({
      acc_err: data.length === 0,
      accSnackbar: true,
      accounts: data.ACCOUNTS,
      balances: data.BALANCES
     })
  })
  }
}
```

```
export const delete_account = async (account_num) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/delete-account', {
      method: 'POST',
      headers: {
        Accept: 'application/json',
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        account_num: account_num
      })
    })

    const data = await response.json()
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL DELETE
# METHOD PURPOSE: delete an existing account
# INPUT PARAMETERS: only param needed is account_number
# OUTPUT: removes the account and its associated updates from the db
@app.route('/syb-bank/delete-account', methods=['POST']) # POST request since we are manipulating the DB
def delete_account():

    # read the user's ID from local storage using the scraper in scraper.py
    user_id = s.read_local_storage()

    # get request attribute from the frontend
    account_number = request.json.get('account_num')

    balances, accounts, data = [], [], []

    # call isChecking prodecure to check if the account number belongs to a checking account
    cur.callproc('isChecking1', [account_number])

    # iterate through the result set
    for result in cur.stored_results():
        data = result.fetchone() # there's only one row, so we can just fetch it right away

    data = str(data)[1]

    # if the result is a 1, the account is a checking and cannot be deleted
    if data == '1':
        return '0'

    # else, the account is either a savings or a money market and can be deleted, so fetch
    # updated data
    else:
        # call deleteBankAccountProcedure
        cur.callproc('deleteBankAccount', [account_number])

        # call getBalances procedure with user ID
        cur.callproc('getBalances', [user_id])

        # iterate through result set
        for set in cur.stored_results():
            for row in set:
                balances.append(dict(zip(set.column_names,row))) # append each row to an array

        # call the getAccounts procedure with user ID
        cur.callproc('getAccounts', [user_id])

        # iterate through the result set
        for set in cur.stored_results():
            for row in set:
                accounts.append(dict(zip(set.column_names,row))) # append each row to an array

        # jsonify the two arrays so the frontend will accept the return value
        return jsonify({
            'BALANCES': balances,
            'ACCOUNTS': accounts
        })
```

6. Make a transfer (user)


```
// as long as the transfer amount is not negative, (i.e. the
// amt_err is true), this will call the transfer method in api.js with
// values of account from, account to, and amount, and, once the
// request returns, will set the transaction error to the opposite of the
// fetch request response (the fetch request returns a boolean)
onTransfer = () => {
  if (!this.state.amt_err) {
    transfer(
    this.state.acc_from,
    this.state.acc_to,
    this.state.amount
  ).then(data => {
    this.setState({
      transaction_error: !data,
      snackbar: true
    })
  })
  }
}
```

```
export const transfer = async (acc_from, acc_to, amount) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/transfer', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        acc_from: acc_from,
        acc_to: acc_to,
        amount: amount
      })
    })

    const data = response.ok
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL INSERT
# METHOD PURPOSE: initiate a transfer
# INPUT PARAMETERS: only params needed for the SQL query are acc_from, acc_to, amount, and user_id
# OUTPUT: TRANSFER and UPDATE records will be created while attributes not mentioned are default vals and
# will be generated on transfer creation
@app.route('/syb-bank/transfer', methods=['POST']) # POST request since we are manipulating the DB
def transfer():

    # read the user's ID from local storage using the scraper in scraper.py
    user_id = s.read_local_storage()

    # get request attributes from the frontend
    acc_from = request.json.get('acc_from')
    acc_to = request.json.get('acc_to')
    amt = request.json.get('amount')

    # since nothing is returned, there is no result set to iterate through
    # this will just return the network response (200, 404, 500, etd.)
    return cur.callproc('transfer', [acc_from, acc_to, amt, user_id])
```

7. Make a deposit (user)


```javascript
// as long as the deposit amount is not negative, (i.e. the
// amt_err is true), this will call the deposit method in api.js with
// values of account from, account to, and amount, and, once the
// request returns, will set the transaction error to the opposite of the
// fetch request response (the fetch request returns a boolean)
onDeposit = () => {
  if (!this.state.amt_err) {
  deposit(
    this.state.acc_to,
    this.state.amount
  ).then(data => {
    this.setState({
      transaction_error: !data,
      snackbar: true
    })
  })
  }

}
```

```javascript
export const deposit = async (acc_to, amount) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/deposit', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        acc_to: acc_to,
        amount: amount
      })
    })

    const data = response.ok
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL INSERT
# METHOD PURPOSE: initiate a deposit
# INPUT PARAMETERS: only params needed for the SQL query are acc_to, amount, and user_id
# OUTPUT: TRANSFER and UPDATE records will be created while attributes not mentioned are default vals and
# will be generated on transfer creation
@app.route('/syb-bank/deposit', methods=['POST']) # POST request since we are manipulating the DB
def deposit():

    # read the user's ID from local storage using the scraper in scraper.py
    user_id = s.read_local_storage()

    # get request attributes from the frontend
    acc_to = request.json.get('acc_to')
    amt = request.json.get('amount')

    # since nothing is returned, there is no result set to iterate through
    # this will just return the network response (200, 404, 500, etd.)
    return cur.callproc('deposit', [acc_to, amt, user_id])
```

8. Make a withdrawal (user)


```
// as long as the withdraw amount is not negative, (i.e. the
// amt_err is true), this will call the withdraw method in api.js with
// values of account from and amount, and, once the
// request returns, will set the transaction error to the opposite of the
// fetch request response (the fetch request returns a boolean)
onWithdraw = () => {
  if (!this.state.amt_err) {
  withdraw(
    this.state.acc_from,
    this.state.amount
  ).then(data => {
    this.setState({
      transaction_error: !data,
      snackbar: true
    })
  })
  }

}
```

```
export const withdraw = async (acc_from, amount) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/withdraw', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        acc_from: acc_from,
        amount: amount
      })
    })

    const data = response.ok
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: USER
# PREDICTED QUERIES TO BE USED: MySQL INSERT
# METHOD PURPOSE: initiate a withdrawal
# INPUT PARAMETERS: only params needed for the SQL query are acc_from, amount, and user_id
# OUTPUT: TRANSFER and UPDATE records will be created while attributes not mentioned are default vals and
# will be generated on transfer creation
@app.route('/syb-bank/withdraw', methods=['POST']) # POST request since we are manipulating the DB
def withdraw():

    # read the user's ID from local storage using the scraper in scraper.py
    user_id = s.read_local_storage()

    # get request attributes from the frontend
    acc_from = request.json.get('acc_from')
    amt = request.json.get('amount')

    # since nothing is returned, there is no result set to iterate through
    # this will just return the network response (200, 404, 500, etd.)
    return cur.callproc('withdraw', [acc_from, amt, user_id])
```

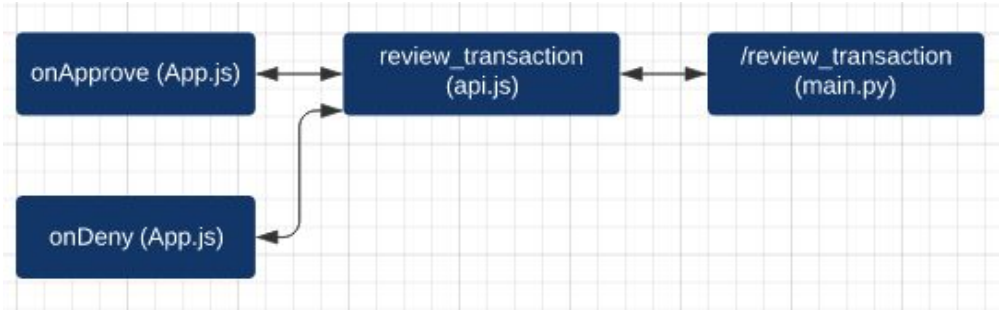
9. Modify customer information (administrator)


```javascript
// when an admin saves their changes, this method will be called, which calls the
// modify_customer method in api.js with the 7 variables below
onCustomerChange = () => {
    modify_customer(
      this.state.user_id,
      this.state.input_first_name,
      this.state.input_last_name,
      this.state.input_area_code,
      this.state.input_phone,
      this.state.input_email,
      this.state.input_password
  ).then(data => {
    this.setState({
      modify_err: data.CUSTOMERS.length === 0,
      customerSnackbar: true,
      disabled: true,
      customers: data.CUSTOMERS
    })
  })
}
```

```javascript
export const modify_customer = async (user_id, first_name, last_name, area_code, phone, email, password) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/modify-customer', {
      method: 'POST',
      headers: {
        Accept: 'application/json',
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        user_id: user_id,
        email: email,
        password: password,
        first_name: first_name,
        last_name: last_name,
        area_code: area_code,
        phone: phone
      }),
    })

    const data = await response.json()
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: ADMINISTRATOR
# PREDICTED QUERIES TO BE USED: MySQL UPDATE
# METHOD PURPOSE: modify customer info
# INPUT PARAMETERS: first_name, last_name, area_code, phone, email, password
# OUTPUT: the selected account will be updated in the USER table, and the query will
# return a new result set of the customers
@app.route('/syb-bank/modify-customer', methods=['POST']) # POST request since we are manipulating the DB
def modify_customer():

    try: # use a try/except to ensure that the user ID is not null since it is not being read from local storage

        # get request attributes from the frontend
        user_id = request.json.get('user_id')
        first_name = request.json.get('first_name')
        last_name = request.json.get('last_name')
        area_code = request.json.get('area_code')
        phone = request.json.get('phone')
        email = request.json.get('email')
        password = request.json.get('password')

        # call the modifyCustomer procedure with all of the above attributes
        cur.callproc('modifyCustomer', [user_id, first_name, last_name, area_code, phone, email, password])

        # call the getCustomers procedure with user ID
        cur.callproc('getCustomers', [user_id])

        customers = []

        # iterate through the result set
        for set in cur.stored_results():
            for row in set:
                customers.append(dict(zip(set.column_names,row))) # append each row to an array

        # jsonify the result so the frontend will accept it
        return jsonify({
            'CUSTOMERS': customers
        })

    # null check on the user ID
    except user_id is None:
        return jsonify([]) # return empty jsonified array if user ID is null
```

10. Review a transaction (administrator)




```
// when a transaction is approved, set the approved value to 1, and call the review_transaction
// method in api.js
onApprove = i => {

  const approved = 1

  review_transaction(
    this.state.pending_transactions[i].TRANS_ID,
    approved
  ).then(data => {
    // a new list of the pending transactions will be returned
    this.setState({ pending_transactions: data.PENDING_TRANSACTIONS })
  })
}
```

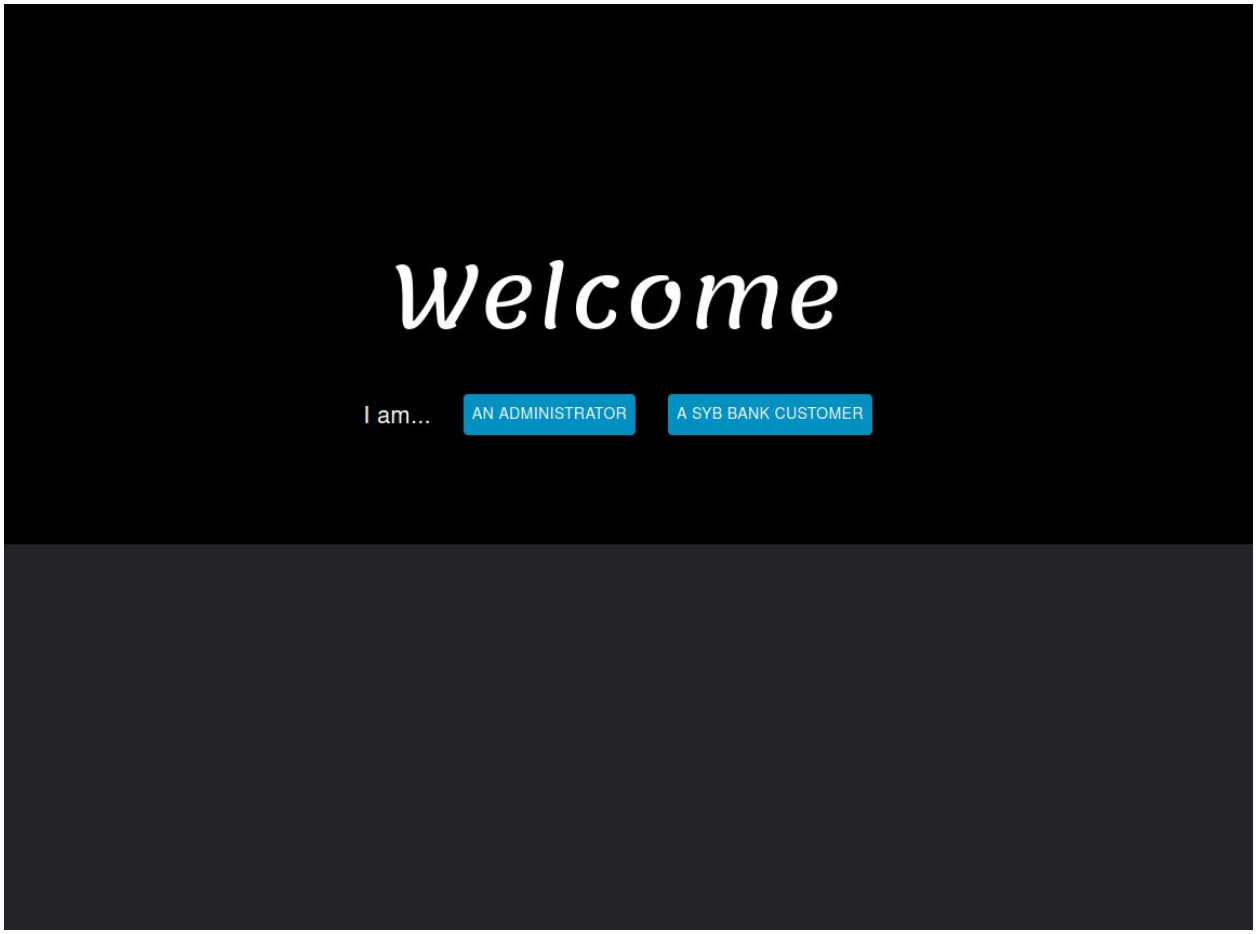
```
// when a transaction is denied, set the approved value to 0, and call the review_transaction
// method in api.js
onDeny = i => {

  const approved = 0

  review_transaction(
    this.state.pending_transactions[i].TRANS_ID,
    approved
  ).then(data => {
    // a new list of the pending transactions will be returned
    this.setState({ pending_transactions: data.PENDING_TRANSACTIONS })
  })
}
```

```javascript
export const review_transaction = async (transaction_id, approved) => {
  try {
    const response = await fetch('http://localhost:5000/syb-bank/review-transaction', {
      method: 'POST',
      headers: {
        Accept: 'application/json',
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        transaction_id: transaction_id,
        approved: approved
      })
    })

    const data = await response.json()
    console.log(data)
    return data
  } catch (err) {
    return err
  }
}
```

```python
# MAIN ACTOR: ADMINISTRATOR
# PREDICTED QUERIES TO BE USED: MySQL UPDATE
# METHOD PURPOSE: review a pending transaction
# INPUT PARAMETERS: SQL takes params transaction_id and a boolean value of approved, which determines
# whether or not the transactions was approved or not
# OUTPUT: TRANSACTION table will be updated, i.e. the approved field will be updated with a value of true or false
@app.route('/syb-bank/review-transaction', methods=['GET', 'POST']) # POST request since we are manipulating the DB, GET for getting transa
def review_transaction():
    try: # use a try/except since user ID is not used in one of the queries

        # read the user's ID from local storage using the scraper in scraper.py
        user_id = s.read_local_storage()

        # get request attributes from the frontend
        trans_id = request.json.get('transaction_id')
        approved = request.json.get('approved')

        pending_transactions = []

        # call the reviewTransaction procedure first, then call the viewTransactionsAdmin procedure to get
        # the new list of pending transactions after one has been approved
        cur.callproc('reviewTransaction', [trans_id, approved])
        cur.callproc('viewTransactionsAdmin', [user_id])

        # iterate through the result set
        for set in cur.stored_results():
            for row in set:
                pending_transactions.append(dict(zip(set.column_names,row))) # append each row to an array

        # jsonify the result so the frontend will accept it
        return jsonify({
            'PENDING_TRANSACTIONS': pending_transactions
        })

    # null check on the user ID
    except user_id is None:
        return jsonify([])
```

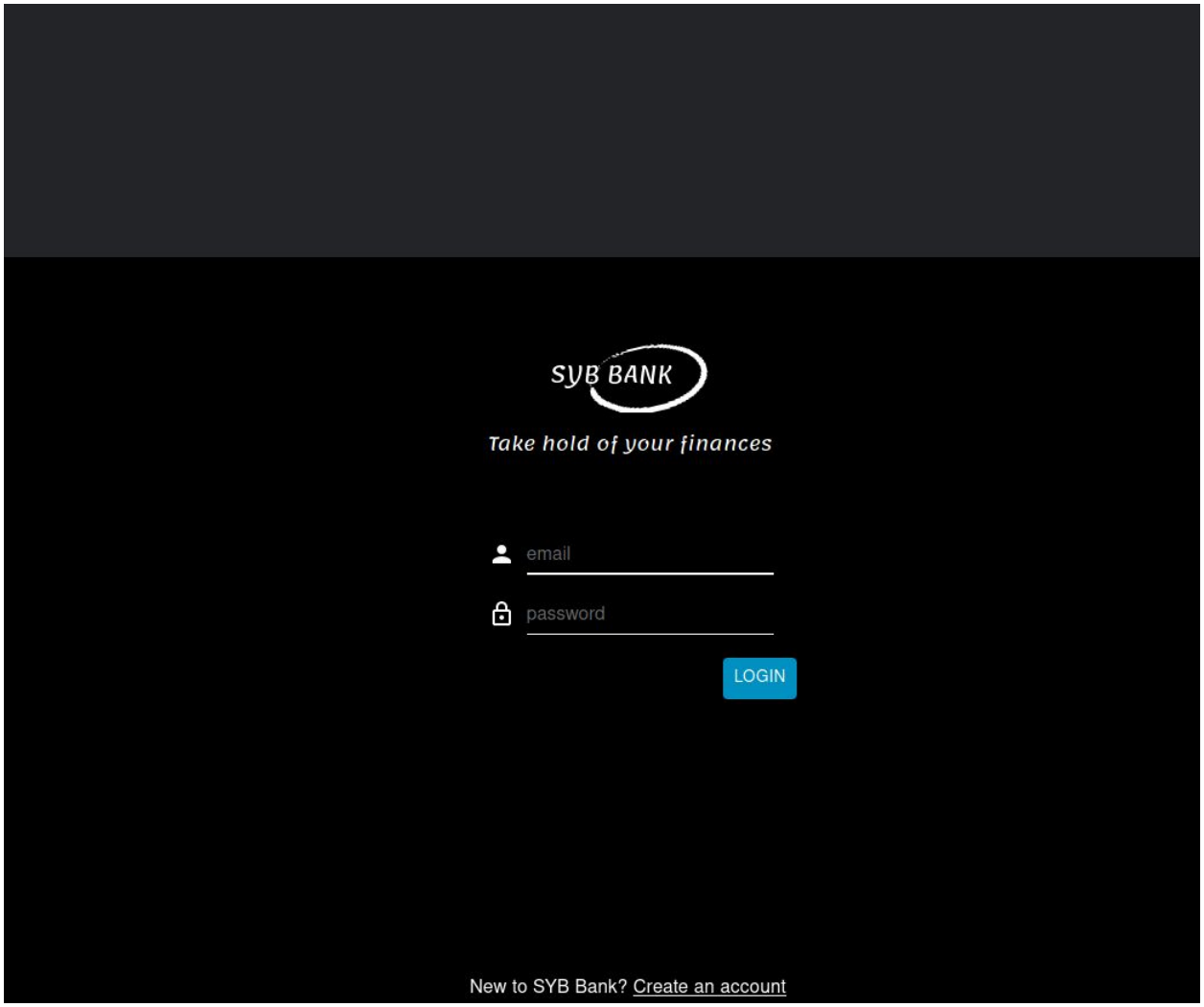
7. User Interface Design

WELCOME



Description: A user or administrator will see this page upon initial visit to the website. The user/administrator will choose their role by clicking their respective button.
Entity: None
Modified Attributes: None

LOGIN



Description: A user or administrator can reach this page either by clicking one of the two buttons on the welcome page, by clicking the login button at the bottom of the register page, or on successful registration. The user/administrator will enter their email and password and click log in.

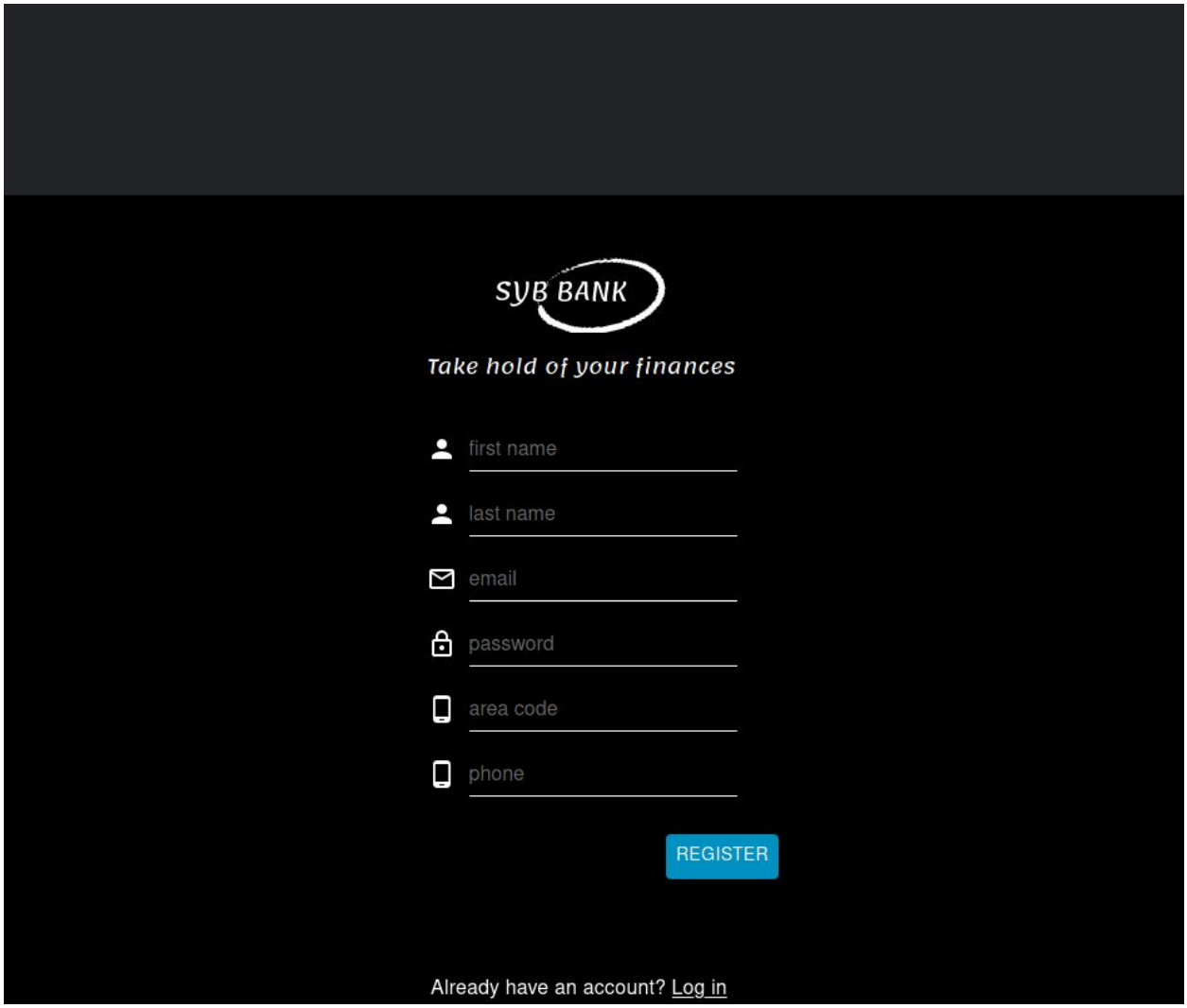
Entity:

USER (**USER_ID** INT, USER_FNAME VARCHAR(20), USER_LNAME VARCHAR(20), USER_AREACODE INT(3), USER_PHONE INT(7), USER_EMAIL VARCHAR(45), USER_PASS VARCHAR(20), USER_TYPE INT)

PRIMARY KEY: USER_ID

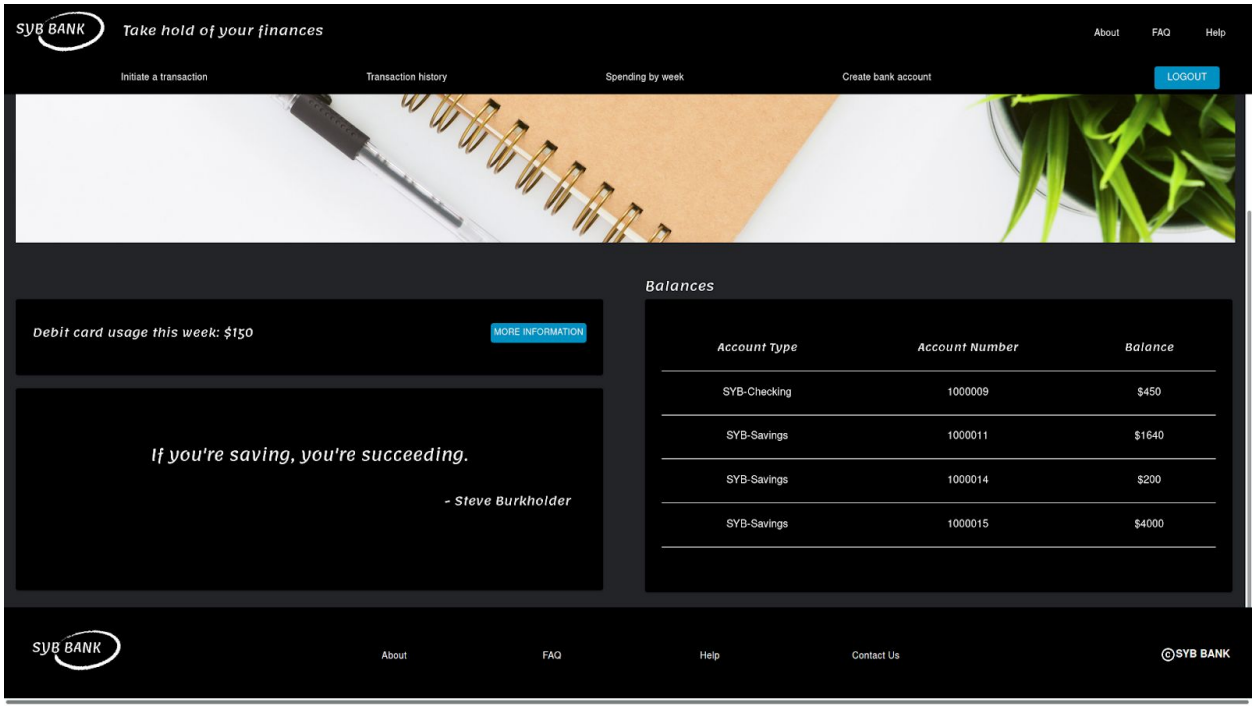
Modified Attributes: None

REGISTER



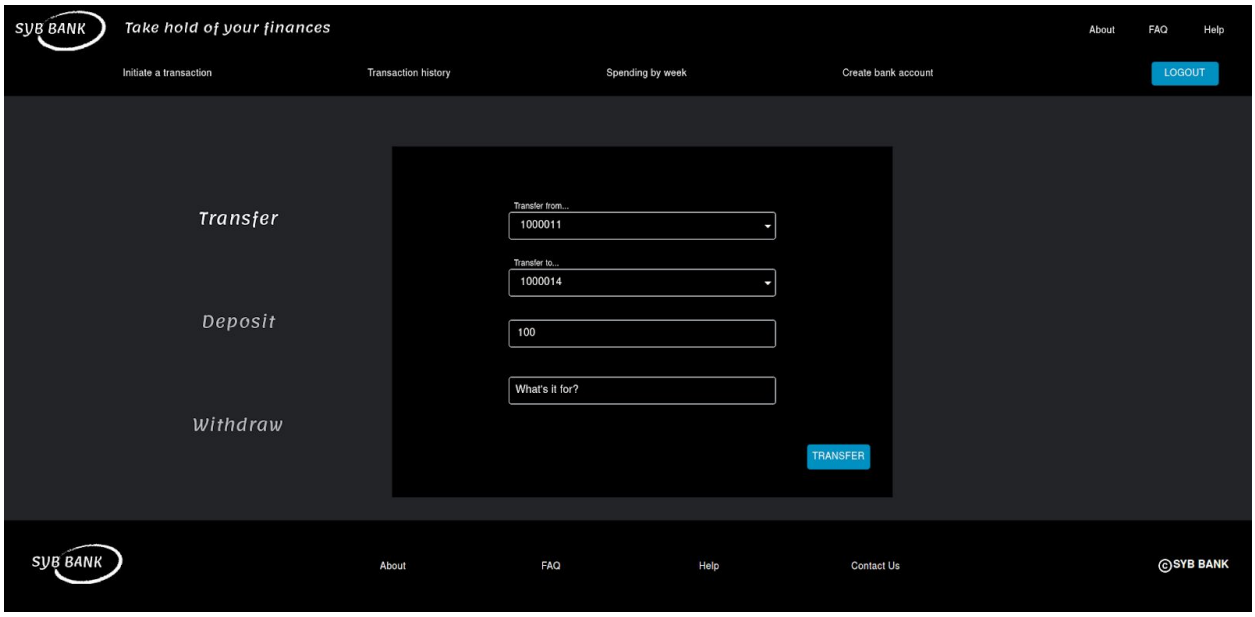
Description: The user can reach this page by clicking the “New to SYB Bank? Create an Account” button at the bottom of the login page. They will enter their first name, last name, area code, phone number, email, and password, and click register.
Entity:
USER (**USER_ID** INT, USER_FNAME VARCHAR(20), USER_LNAME VARCHAR(20), USER_AREACODE INT(3), USER_PHONE INT(7), USER_EMAIL VARCHAR(45), USER_PASS VARCHAR(20), USER_TYPE INT)
PRIMARY KEY: USER_ID
Modified Attributes: Registering creates a new row in the USER table.

USER HOME PAGE



Description: The user will reach the home page upon successful login. The main functionality of this page allows the user to initiate a transaction, view their transaction history, view their weekly spending, create a new bank account, and log out. Secondary actions include viewing a help page, an FAQ page, an about page, and providing a way to contact SYB Bank.
Entity: None
Modified Attributes: None

USER INITIATE TRANSFER



Description: The user can reach this page by clicking the “Initiate a transaction” button in the header of the home page, and then clicking the “Transfer” button on the left of the resulting page. This page allows a user to initiate a transfer, while also providing all of the same functionality as the home page (as the header and footer remain the same). To initiate a transfer, the user must input in which account they wish to transfer from, which account they wish to transfer to, an amount, and a reason for the transfer, if they’d like to. They will then click the “TRANSFER” button.

Entity:

TRANSFER (**TRANS_ID** INT, CUS_ID INT, EMP_ID INT, TRANS_APPROVED TINYINT(1))

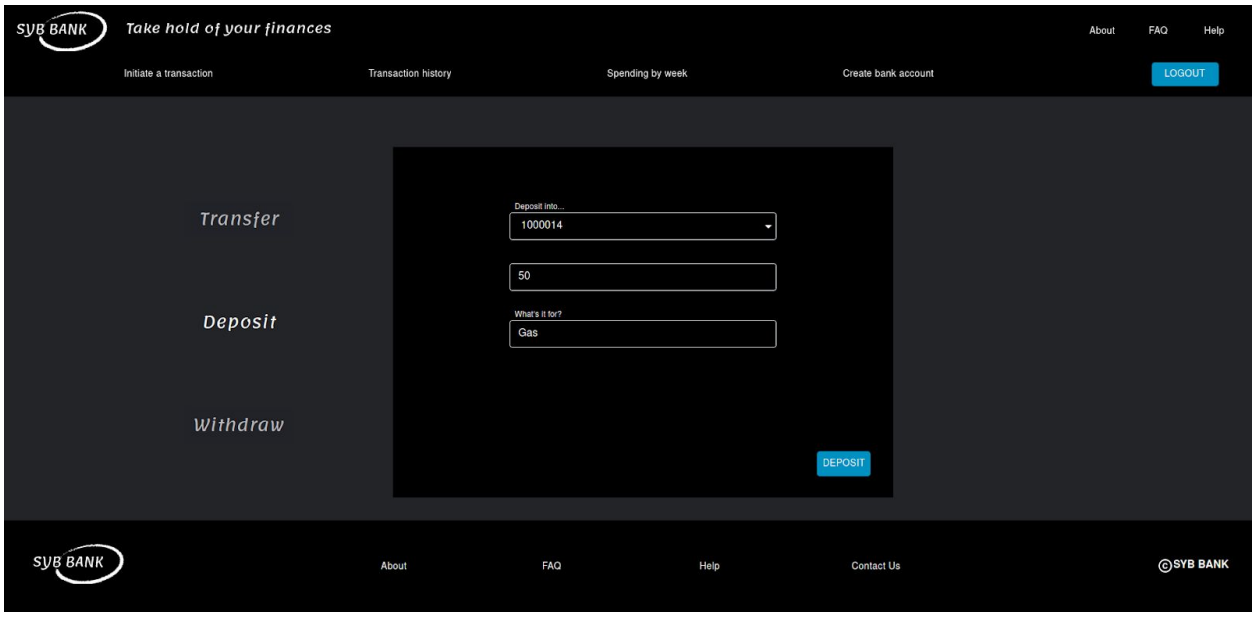
PRIMARY KEY: TRANS_ID

FOREIGN KEY: CUS_ID REFERENCES USER_ID IN CUSTOMER

FOREIGN KEY: EMP_ID REFERENCES USER_ID IN EMPLOYEE

Modified Attributes: A new row will be created in the TRANSFER table.

USER INITIATE DEPOSIT

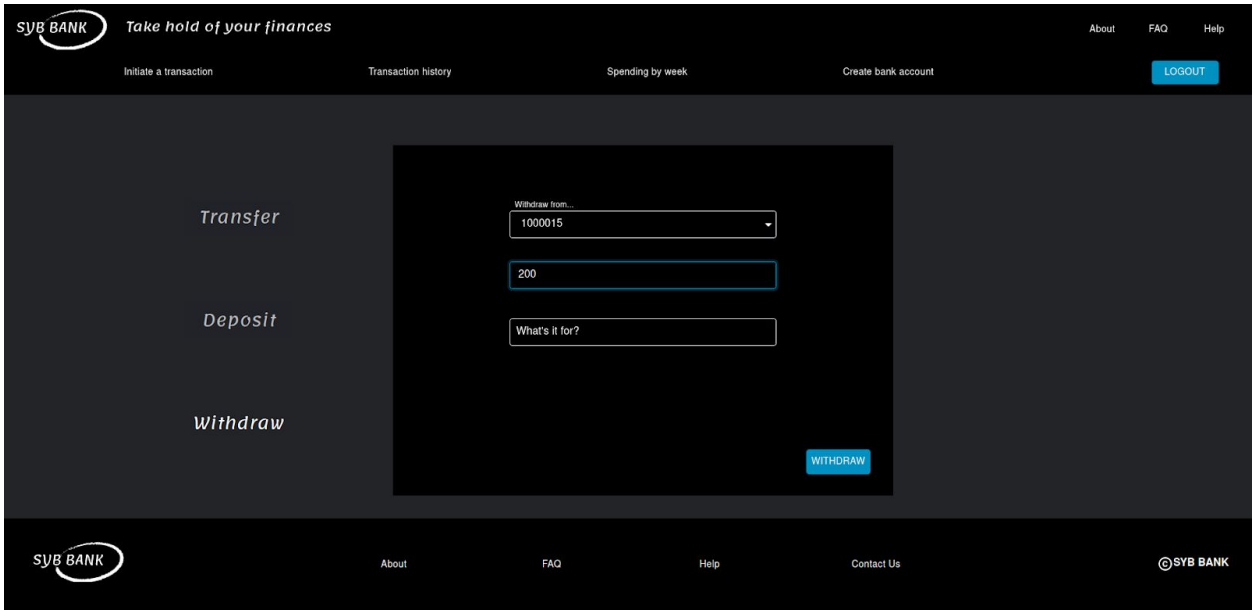


Description: The user can reach this page by clicking the “Initiate a transaction” button in the header of the home page, and then clicking the “Deposit” button on the left of the resulting page. This page allows a user to initiate a deposit, while also providing all of the same functionality as the home page (as the header and footer remain the same). To initiate a deposit, the user must input in which account they wish to deposit, an amount, and a reason for the deposit, if they’d like to. They will then click the “DEPOSIT” button.
Entity:
TRANSFER (**TRANS_ID** INT, CUS_ID INT, EMP_ID INT, TRANS_APPROVED TINYINT(1))

PRIMARY KEY: TRANS_ID
FOREIGN KEY: CUS_ID REFERENCES USER_ID IN CUSTOMER
FOREIGN KEY: EMP_ID REFERENCES USER_ID IN EMPLOYEE

Modified Attributes: A new row will be created in the TRANSFER table.

USER INITIATE WITHDRAWAL



Description: The user can reach this page by clicking the “Initiate a transaction” button in the header of the home page, and then clicking the “Withdraw” button on the left of the resulting page. This page allows a user to initiate a withdrawal, while also providing all of the same functionality as the home page (as the header and footer remain the same). To initiate a withdrawal, the user must input in which account they wish to withdraw, an amount, and a reason for the deposit, if they’d like to. They will then click the “WITHDRAW” button.

Entity:

TRANSFER (**TRANS_ID** INT, CUS_ID INT, EMP_ID INT, TRANS_APPROVED TINYINT(1))

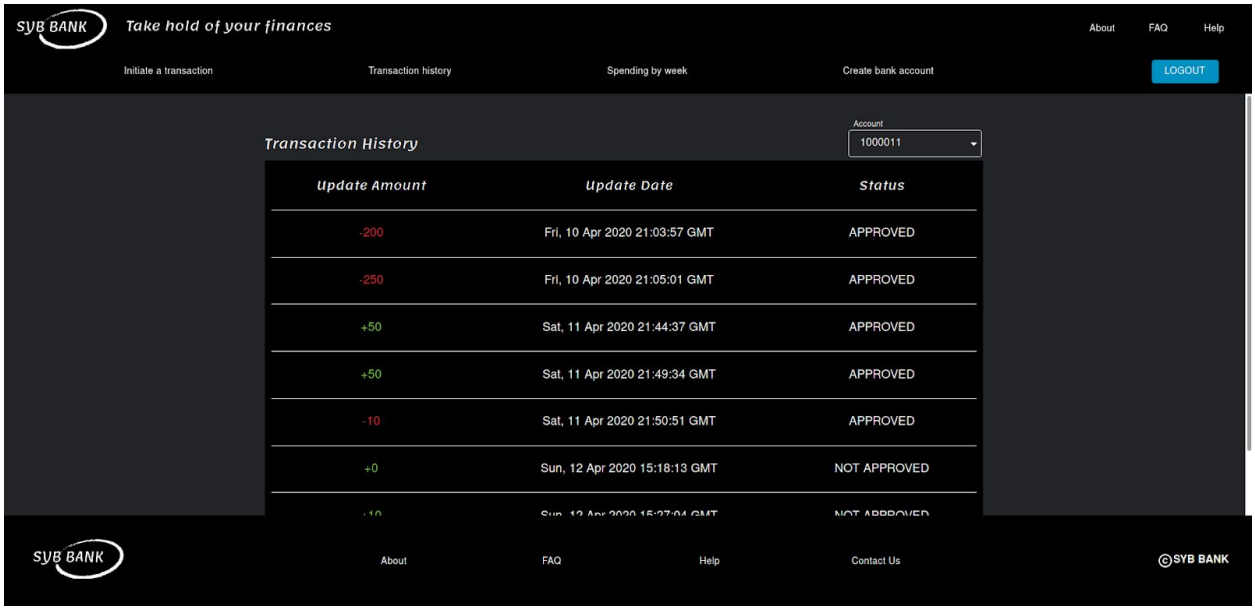
PRIMARY KEY: TRANS_ID

FOREIGN KEY: CUS_ID REFERENCES USER_ID IN CUSTOMER

FOREIGN KEY: EMP_ID REFERENCES USER_ID IN EMPLOYEE

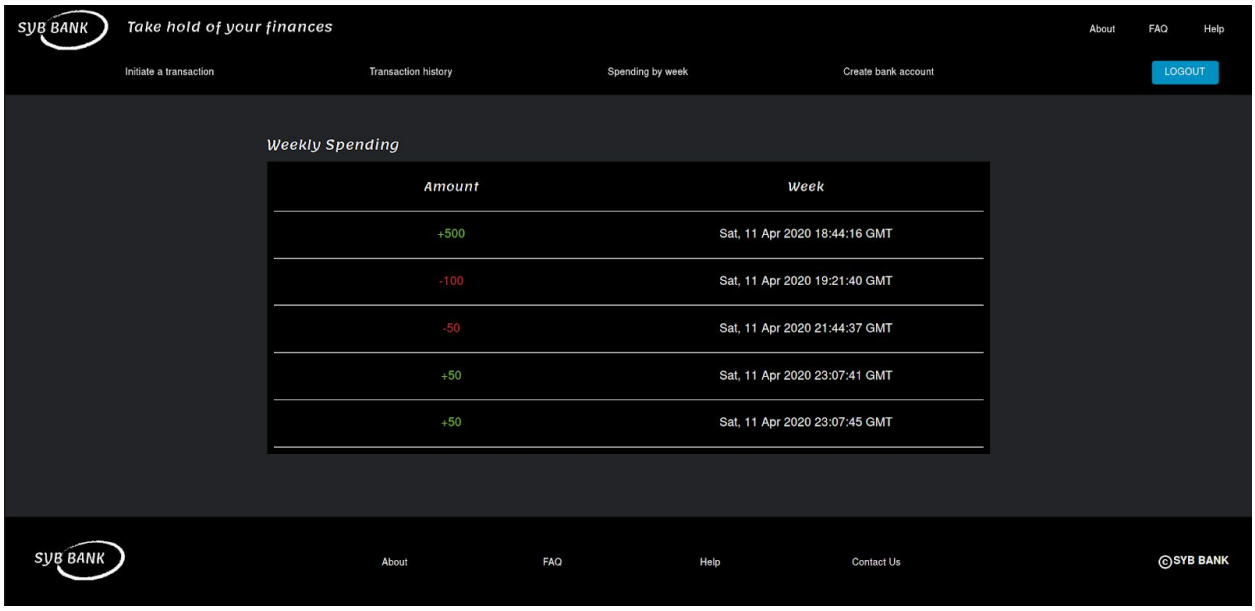
Modified Attributes: A new row will be created in the TRANSFER table.

USER VIEW TRANSACTION HISTORY



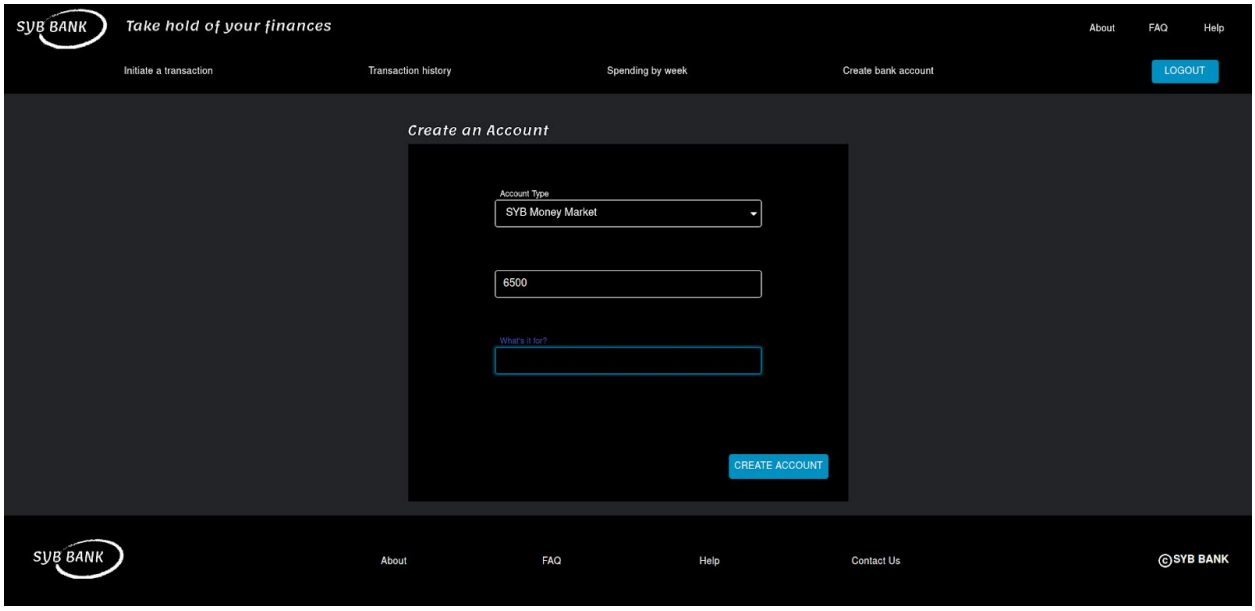
Description: The user can reach this page by clicking the “Transaction history” button in the header of the home page. This page allows a user to select applicable accounts to view transaction history of, as well as close the currently selected account, while also providing all of the same functionality as the home page (as the header and footer remain the same). To change which account’s transaction history the user would like to view, they can simply select the one you’d like to view in the dropdown.
Entity: None
Modified Attributes: None

USER VIEW WEEKLY TRANSACTIONS



Description: The user can reach this page by clicking the “Spending by week” button in the header of the home page. This page allows a user to select applicable accounts to view the weekly spending of, as well as close the currently selected account, while also providing all of the same functionality as the home page (as the header and footer remain the same). To change which account’s weekly spending the user wishes to view, they can simply select the one you’d like to view in the dropdown.
Entity: None
Modified Attributes: None

USER CREATE NEW BANK ACCOUNT



Description: The user can reach this page by clicking the “Create bank account” button in the header of the home page. This page allows a user to create a new account while also providing all of the same functionality as the home page (as the header and footer remain the same). To create the account, the user will specify the type of account they’d like to create, the starting balance, and an optional reason, and then click the “CREATE ACCOUNT” button.

Entity:

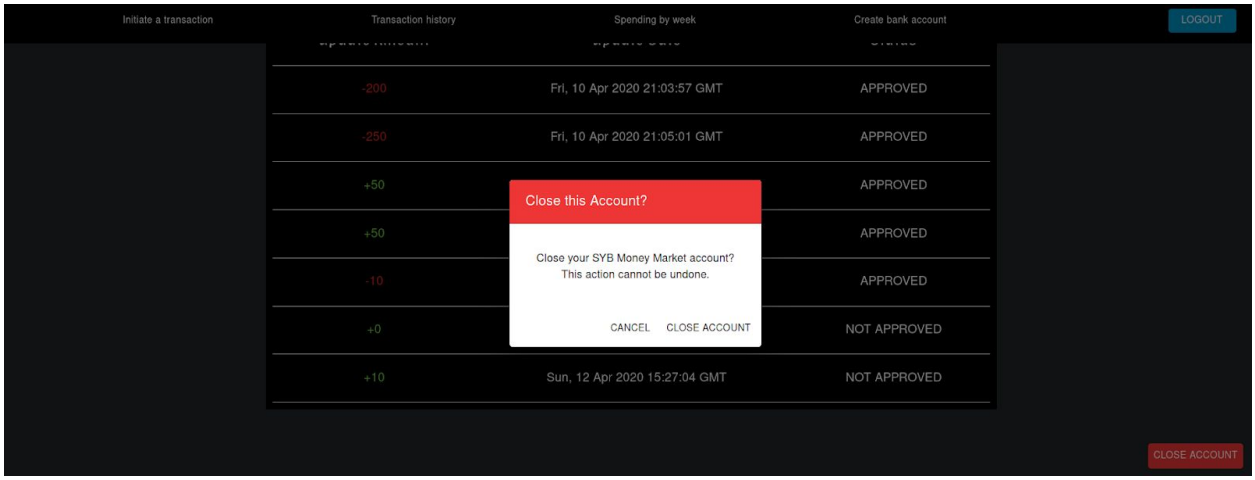
ACCOUNT (**ACCT_ID** INT, ACCT_NUMBER INT, ACCT_BALANCE DOUBLE, CUS_ID INT)

PRIMARY KEY: ACCT_ID

FOREIGN KEY: CUS_ID REFERENCES USER_ID IN CUSTOMER

Modified Attributes: A new row will be created in the ACCOUNT table.

USER CLOSE EXISTING BANK ACCOUNT



Description: The user will be able to view this dialog by clicking the “Close Account” button that appears on both the “Transaction history” and “Spending by week” pages. This dialog allows the user to close the account that they were currently viewing statistics on in either of those pages. To close the account, the user will simply click “Close Account”.

Entity:

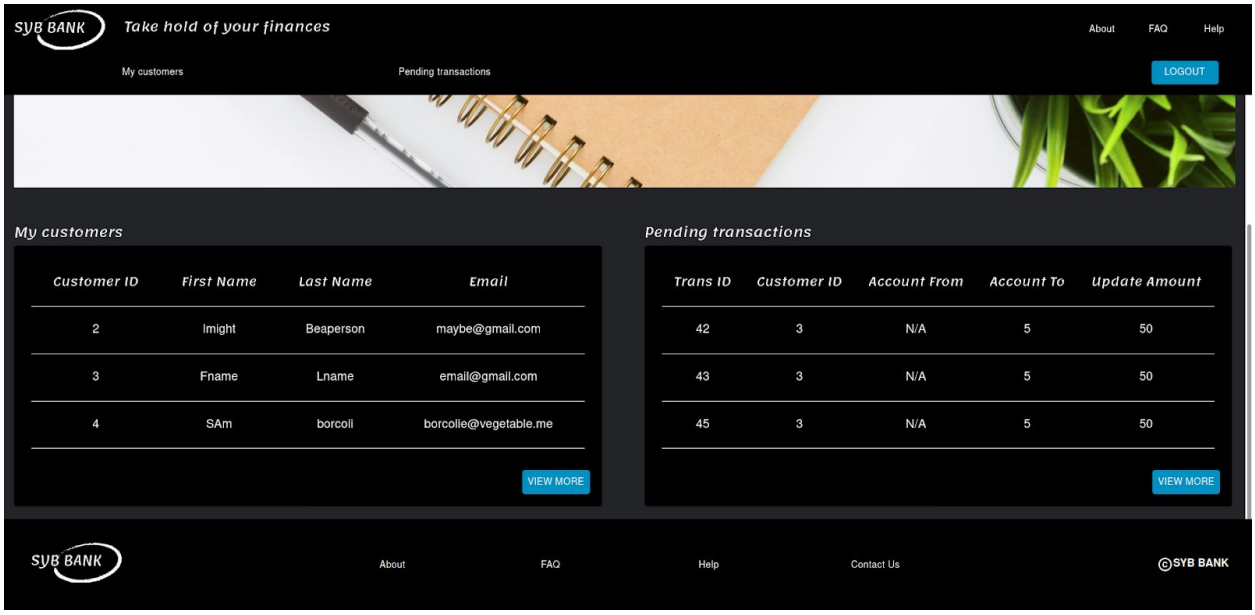
ACCOUNT (**ACCT_ID** INT, ACCT_NUMBER INT, ACCT_BALANCE DOUBLE, CUS_ID INT)

PRIMARY KEY: ACCT_ID

FOREIGN KEY: CUS_ID REFERENCES USER_ID IN CUSTOMER

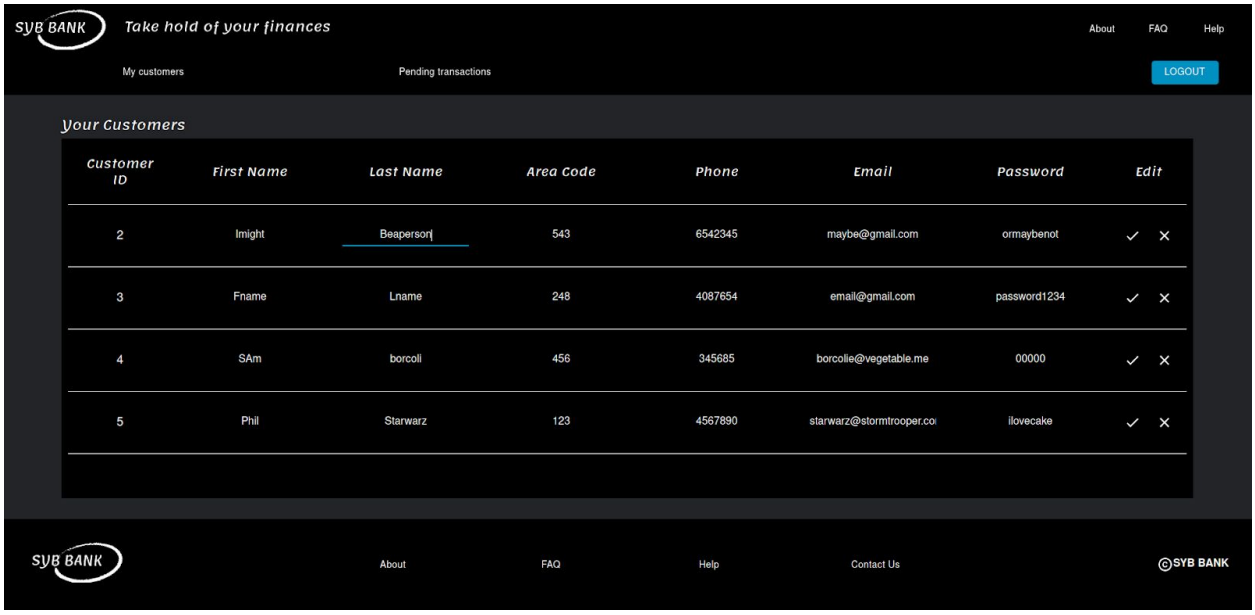
Modified Attributes: A row will be deleted in the ACCOUNT table.

ADMINISTRATOR HOME PAGE



Description: The administrator will reach the home page upon successful login. The main functionality of this page allows the administrator to view their managed customers, view their pending transactions, and log out. Secondary actions include viewing a help page, an FAQ page, an about page, and providing a way to contact SYB Bank.
Entity: None
Modified Attributes: None

ADMINISTRATOR VIEW MANAGED CUSTOMERS



Description: The administrator can reach this page by clicking the “My customers” button in the header of the home page. This page allows an administrator to view their managed customers and edit their information, while also providing all of the same functionality as the home page (as the header and footer remain the same). To edit a customer’s information, click the pencil icon button. Upon click, all fields except the customer ID field will change to editable text fields, and the pencil icon button will change to two icon buttons, a check (to submit newly edited information) and an X (to cancel editing).

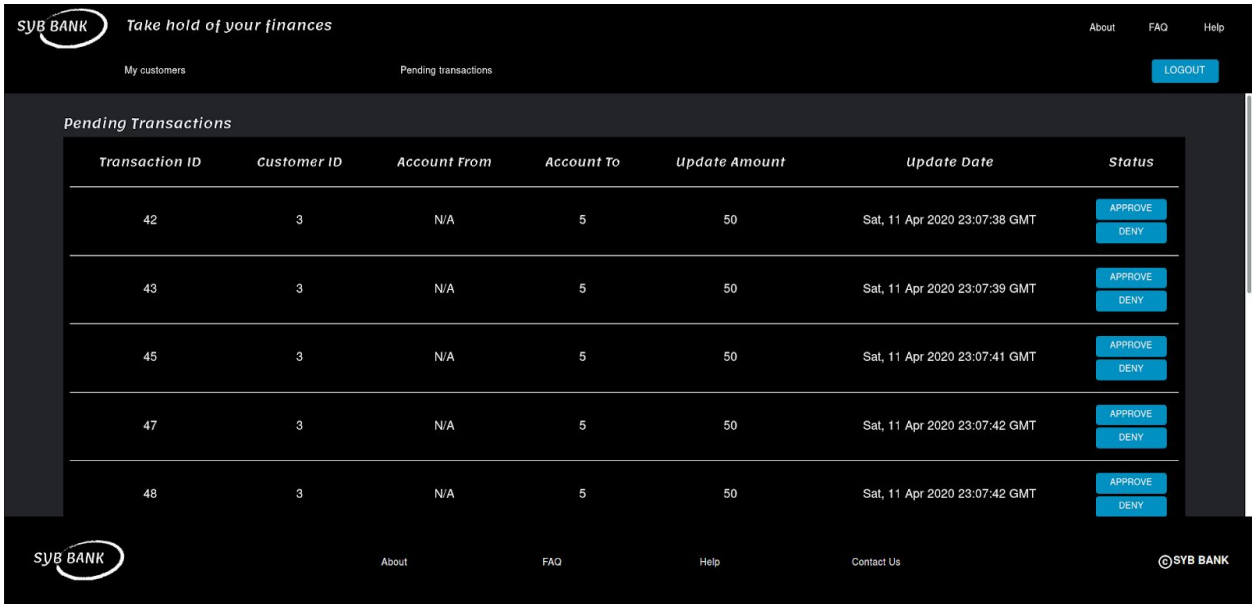
Entity:

USER (**USER_ID** INT, USER_FNAME VARCHAR(20), USER_LNAME VARCHAR(20), USER_AREACODE INT(3), USER_PHONE INT(7), USER_EMAIL VARCHAR(45), USER_PASS VARCHAR(20))

PRIMARY KEY: USER_ID

Modified Attributes: Some combination of USER_FNAME, USER_LNAME, USER_AREACODE, USER_PHONE, USER_EMAIL and USER_PASS will be updated in the USER table.

ADMINISTRATOR VIEW PENDING TRANSACTIONS



Description: The administrator can reach this page by clicking the “Pending transactions” button in the header of the home page. This page allows an administrator to view their pending transactions and approve/deny them, while also providing all of the same functionality as the home page (as the header and footer remain the same). To approve a transaction, click “Approve”, and to deny one, click “Deny”.

Entity:

TRANSFER (**TRANS_ID** INT, CUS_ID INT, EMP_ID INT, TRANS_APPROVED TINYINT(1))

PRIMARY KEY: TRANS_ID

FOREIGN KEY: CUS_ID REFERENCES USER_ID IN CUSTOMER

FOREIGN KEY: EMP_ID REFERENCES USER_ID IN EMPLOYEE

Modified Attributes: The TRANS_APPROVED field will be updated to either 1 or 0 in the TRANSFER table.

8. Implementation Plan

03/28/2020 - 04/03/2020	
Team Member	**Task**
Jakob	Set up DB and write SQL
Olivia	Set up DB and write SQL
Sam	Frontend development, help with SQL
04/04/2020 - 04/10/2020	
Team Member	**Task**
Jakob	Finish database, integrate into backend, implement the make transaction, weekly transactions, and delete account modules
Olivia	Finish database, integrate into backend, implement the create account, login, and modify customer modules
Sam	Finish frontend, integrate into backend, implement the create bank account, view transaction, and review transaction modules
04/11/2020 - 04/17/2020	
Team Member	**Task**
Jakob	Test integration, fix bugs
Olivia	Test integration, fix bugs
Sam	Test integration, fix bugs

9. Code Listing

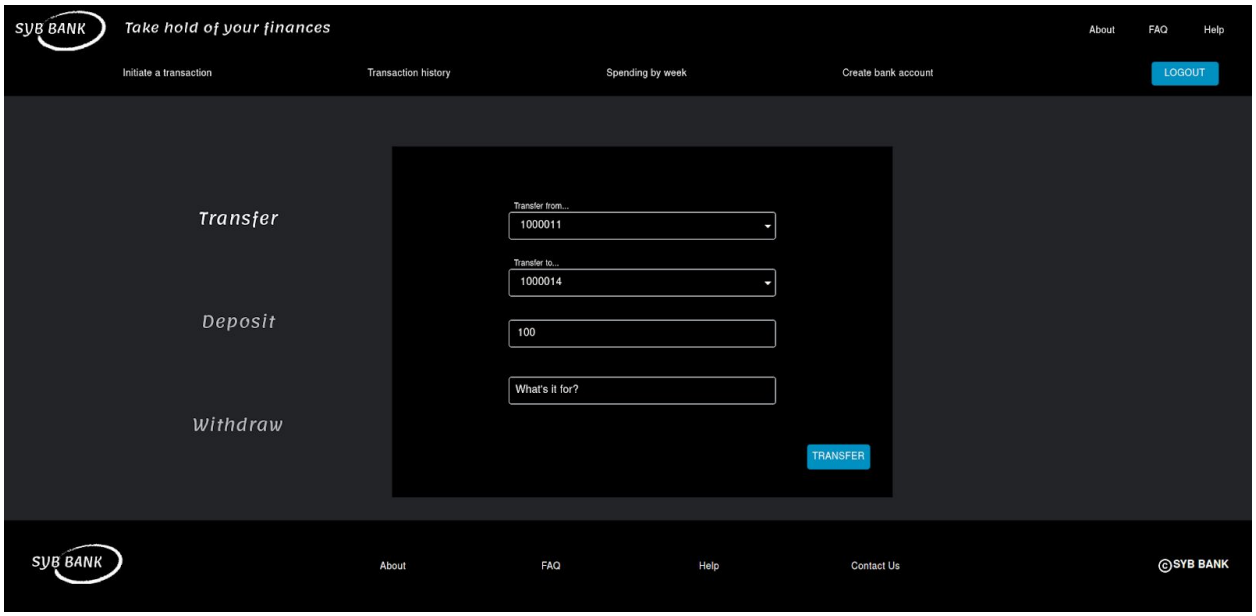
The code for SYB Bank is provided on this link: https://github.com/sbracellari/SYB-Bank

10. Sample output

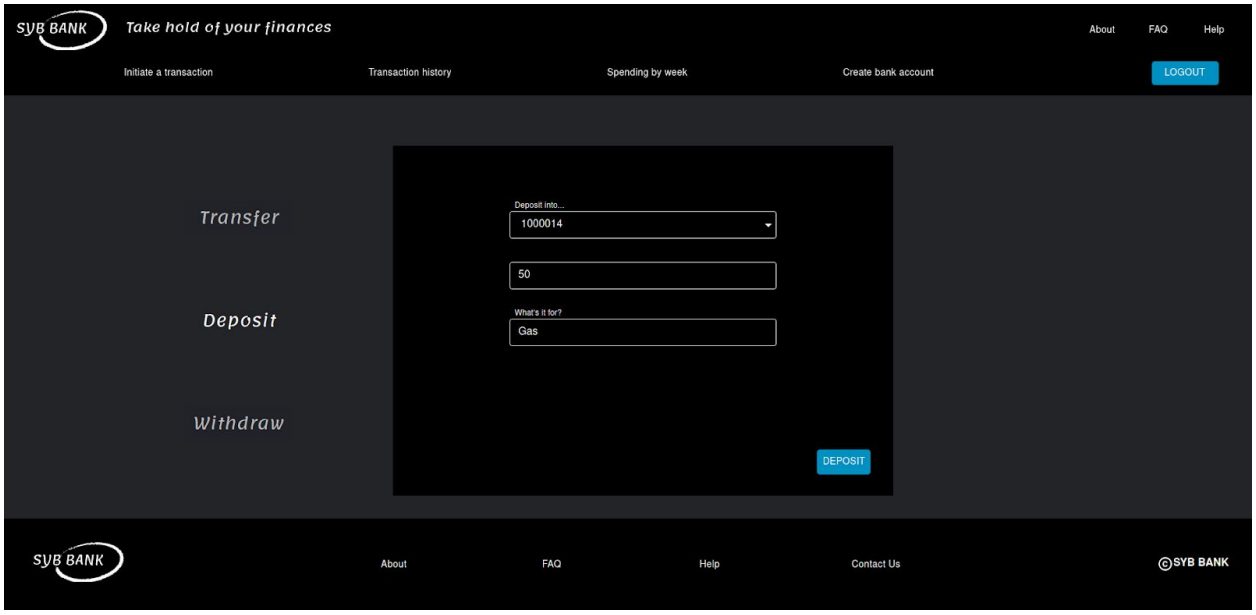
Screenshots are based on functional requirements.

INITIATE TRANSACTION

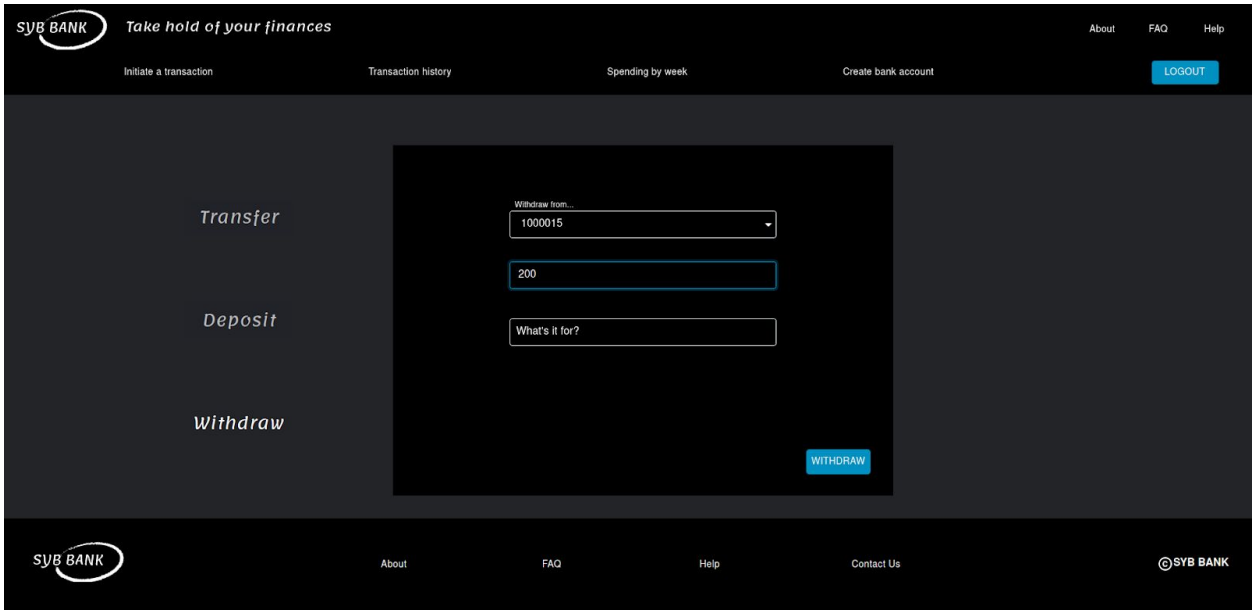
Transfer



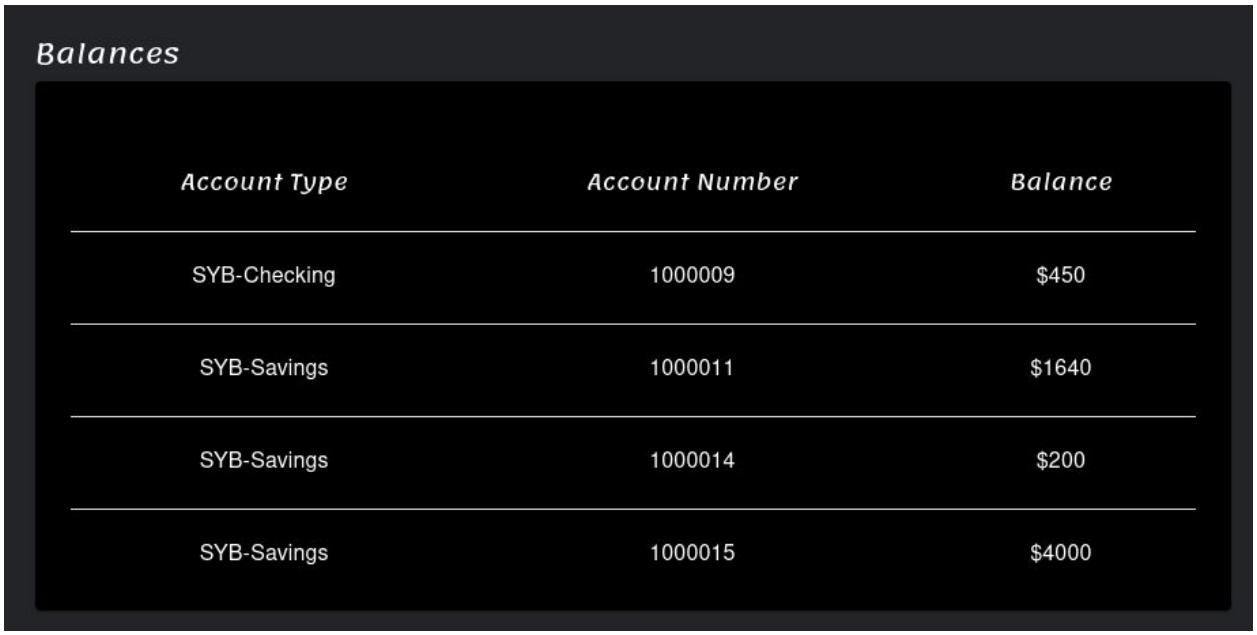
Deposit



Withdraw

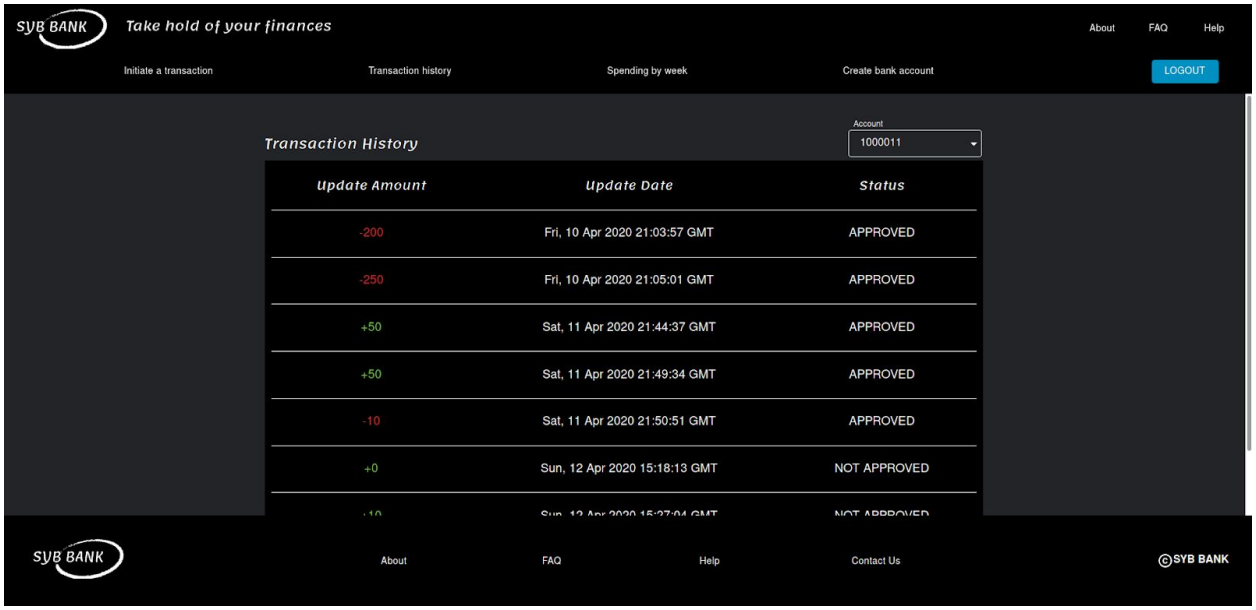


RETRIEVE ACCOUNT BALANCE

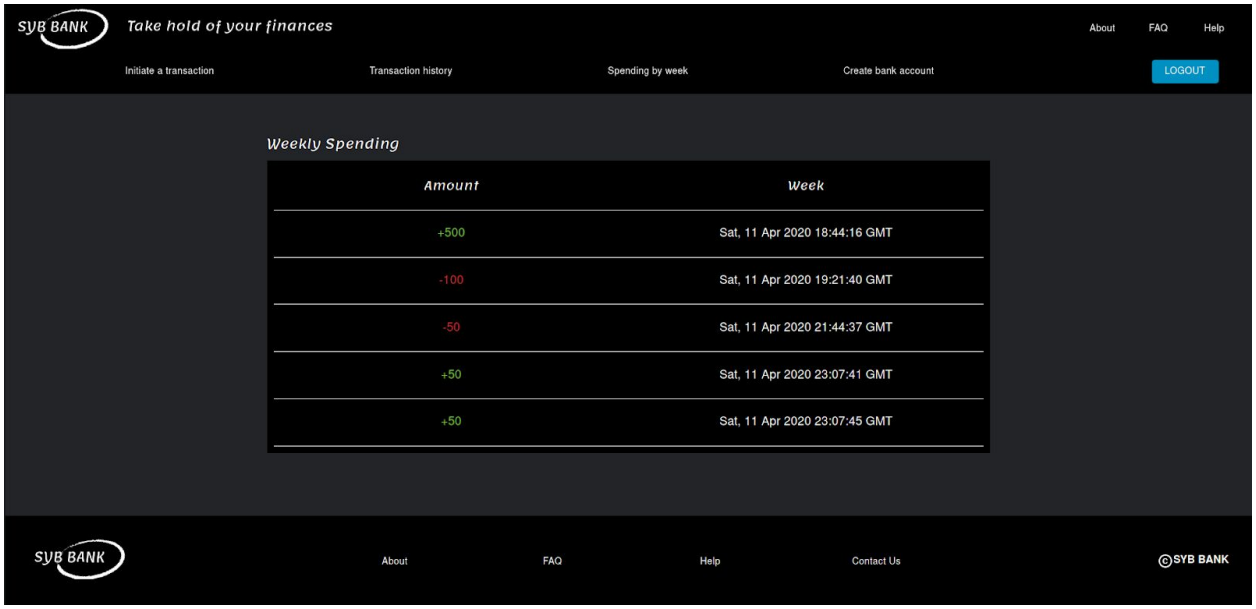


RETRIEVE TRANSACTION INFORMATION

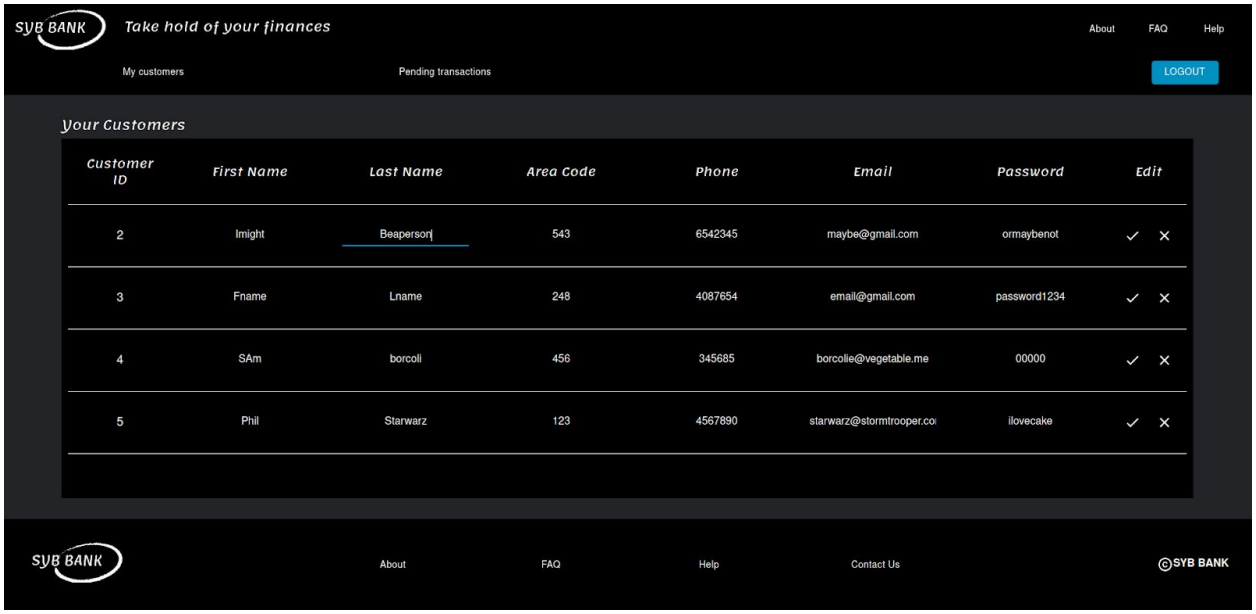
Transaction History



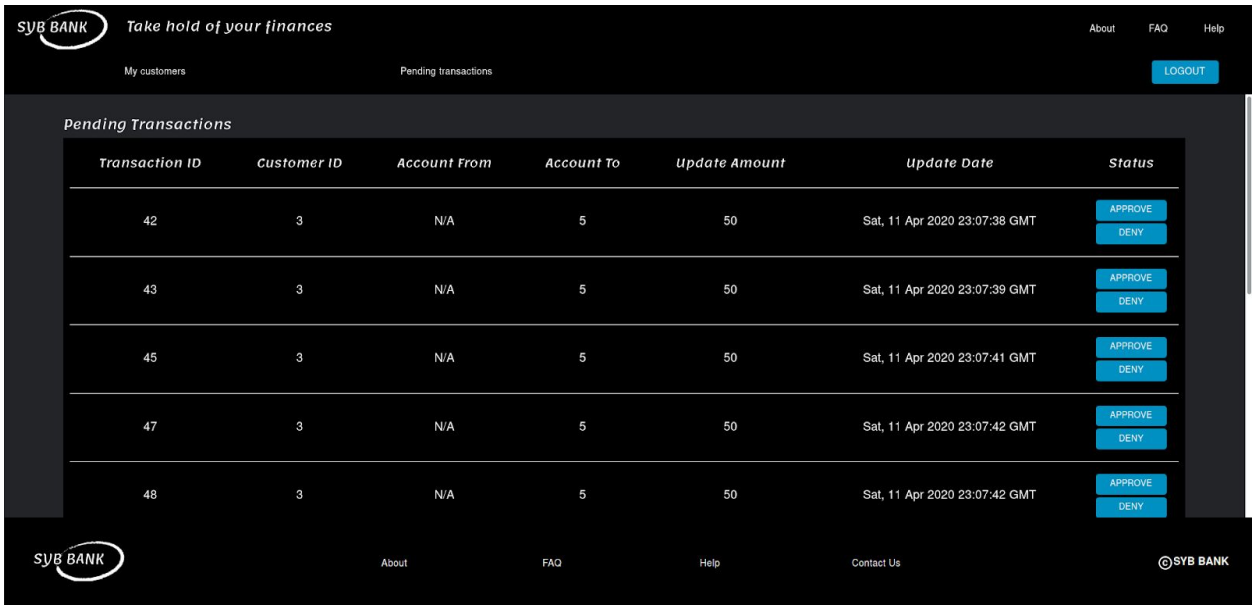
Weekly Transactions



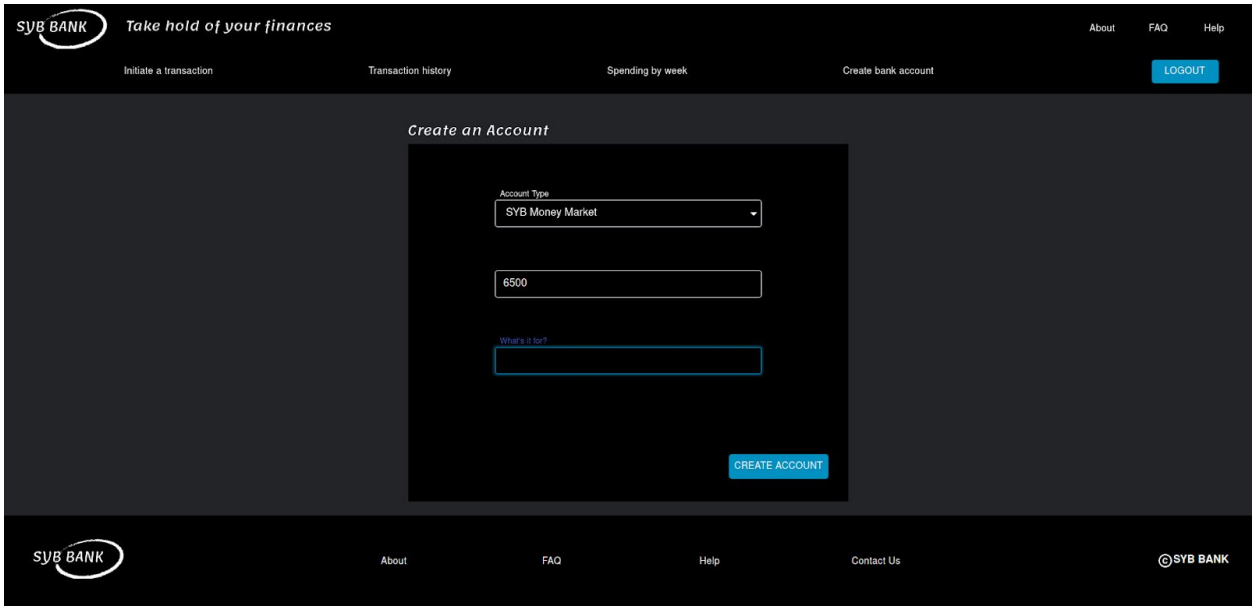
MODIFY CUSTOMER INFORMATION



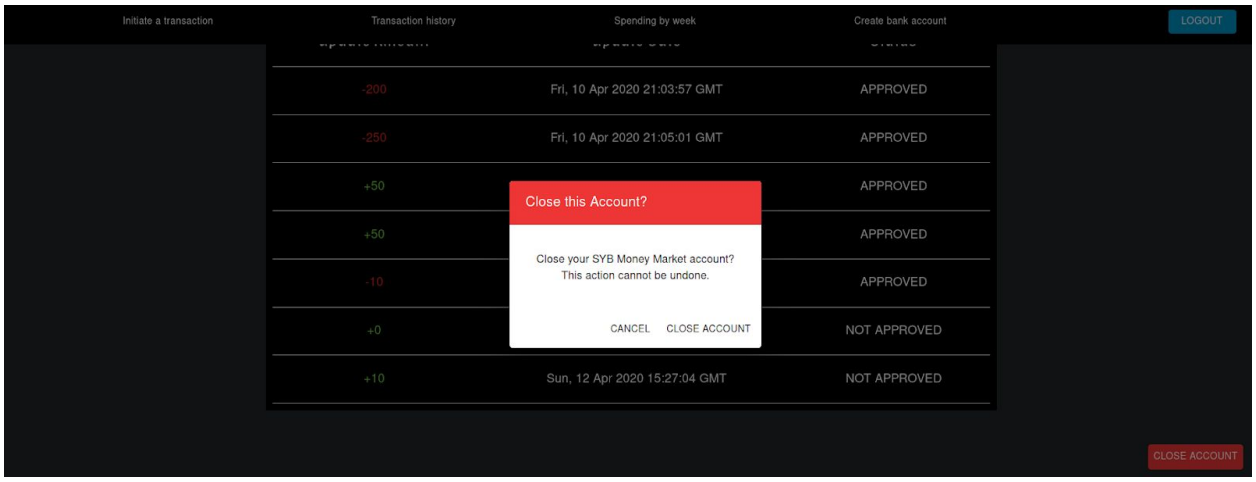
REVIEW PENDING TRANSACTIONS



CREATE BANK ACCOUNT



DELETE BANK ACCOUNT



References

https://reactjs.org/docs/getting-started.html
https://flask.palletsprojects.com/en/1.1.x/
https://www.selenium.dev/
https://www.stackoverflow.com/
https://dev.mysql.com/doc/

Special thanks to Tirachard Kumtanom, the photographer of the photo used on the admin and user home pages