ECE 522 ASSIGNMENT 6

ZHAOYI WANG 1689747

Question 1

For Question a)

```
pub fn get_transaction_history(&self, u_name: &str) ->
Result<(), UBaseErr> {
        // Establish the connection
        let conn = sqlite::open(&self.fname).unwrap();

        // check the entry for selected "sender"
        let mut st_send = conn
            .prepare("SELECT * FROM transactions WHERE u_from =
?;").unwrap();
        st_send.bind(1, u_name);
        // print all results
        while let State::Row = st_send.next().unwrap() {
            let sender = st_send.read::<String>(0).unwrap();
            if u_name == sender {
                let u_from = sender;
                let u_to = st_send.read::<String>(1).unwrap();
                let time_stamp = st_send.read::<String>
(2).unwrap();
                let amount = st_send.read::<String>
(3).unwrap();
                println!("{} sent {} to {} on {}.", u_from,
amount, u_to, time_stamp);
            }
        };
```

```
        // check the entry for selected "receiver"
        let mut st_rec = conn
            .prepare("SELECT * FROM transactions WHERE u_to =
?;").unwrap();
        st_rec.bind(1, u_name);
        // print all results
        while let State::Row = st_rec.next().unwrap() {
            let receiver = st_rec.read::<String>(1).unwrap();
            if u_name == receiver {
                let u_from = st_rec.read::<String>(0).unwrap();
                let u_to = receiver;
                let time_stamp = st_rec.read::<String>
(2).unwrap();
                let amount = st_rec.read::<String>(3).unwrap();
                println!("{} received {} from {} on {}.", u_to,
amount, u_from, time_stamp);
            }
        };

        Ok(())
    }
```

Test Example: The information in database as shown below

	u_name	p_word
1	Mike	$2b$12$nh2yqlJh2h6u5Ta…
2	Tim	$2b$12$AgbIOTDmjB3Zpgj…
3	Sarah	$2b$12$LtEg5qvJsSfShsH…
4	Jason	$2b$12$DW90nGVVFnhqEoO…
5	Jack	$2b$12$9EV4dwHeBv0zkYq…

	u_from	u_to	t_date	t_amount
1	Mike	Tim	2021-11-02 23:28:48	100
2	Sarah	Jason	2021-11-02 23:28:48	150
3	Jack	Mike	2021-11-02 23:28:48	50

We would like to search information that contains `Mike` :

```
/Users/wangzhaoyi/.cargo/bin/cargo run --color=always --package
DBProject --bin DBProject

warning: `DBProject` (bin "DBProject") generated 6 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running `target/debug/DBProject`
Mike sent 100 to Tim on 2021-11-02 23:28:48.
Mike received 50 from Jack on 2021-11-02 23:28:48.

Process finished with exit code 0
```

For Question b)

We will set a new table called `balance` which contains the account name and its balance.

```
create table balance (account text, balance float);
```

For `pay_with_verify()` , `save_money()` and `get_balance()` function:

```
pub fn pay_with_verify(&self, u_from: &str, u_to: &str, amount:
f64) -> Result<(), UBaseErr> {
        let conn = sqlite::open("./data/users.db")?;
        // get the balance from sender and receiver
        let balance_sender = self.get_balance(u_from);
        let balance_receiver = self.get_balance(u_to);
        // whether the sender has enough balance to pay?
        if balance_sender < amount {
            println!("Not enough money for {}'s payment.",
u_from); // if no
        } else {
            // if yes
            // update the transaction table
            let mut st = conn.prepare("insert into transactions
(u_from, u_to, t_date, t_amount)
values(?,?,datetime(\"now\"),?);")?;
            st.bind(1, u_from)?;
            st.bind(2, u_to)?;
            st.bind(3, amount)?;
            st.next()?;
```

```
            // update the balance table
            let mut st_update_sender = conn.prepare("update
balance set balance=? where account=?;")?;
            st_update_sender.bind(1, balance_sender - amount)?;
            st_update_sender.bind(2, u_from)?;
            st_update_sender.next()?;

            let mut st_update_receiver = conn.prepare("update
balance set balance=? where account=?;")?;
            st_update_receiver.bind(1, balance_receiver +
amount)?;
            st_update_receiver.bind(2, u_to)?;
            st_update_receiver.next()?;
        };
        Ok(())
    }
```

```
pub fn save_money(&self, u_name: &str, amount: f64) ->
Result<(), UBaseErr> {
        let current_balance = self.get_balance(u_name);
        let new_balance = amount + current_balance;
        let conn = sqlite::open("./data/users.db")?;
        // add info to balance table
        let mut st2 = conn.prepare("update balance set
balance=? where account=?;")?;
        st2.bind(1, new_balance)?;
        st2.bind(2, u_name)?;
        st2.next()?;
        Ok(())
    }

pub fn get_balance(&self, account: &str) -> f64 {
        let mut balance = 0.0;
        let conn = sqlite::open("./data/users.db").unwrap();
        let mut st = conn
            .prepare("SELECT * FROM balance WHERE account =
?;").unwrap();
        st.bind(1, account);

        while let State::Row = st.next().unwrap() {
            let account_name = st.read::<String>(0).unwrap();
            if account == account_name {
```

```
                return st.read::<String>(1).unwrap().parse::
<f64>().unwrap();
            }
        }
        balance
    }
```

First we initialize our database in `main()` as:

```
let BankBase = UserBase {
        fname: String::from("./data/users.db"),
    };

BankBase.clear_database();

BankBase.add_user("Mike", "123456");
BankBase.add_user("Jason", "123456");
BankBase.add_user("Lin", "123456");

BankBase.save_money("Mike", 500.0);
BankBase.save_money("Jason", 100.0);
BankBase.save_money("Lin", 200.0);
```

Our database now looks like this:

	u_name	p_word
1	Mike	$2b$12$RKakujjyrN4aICqIM0…
2	Jason	$2b$12$Sgr.l2PxD35KZEGP9n…
3	Lin	$2b$12$7HTNnxmZWun0LPG3Sj…

	account	balance
1	Mike	500
2	Jason	100
3	Lin	200

Now, we make some payment in `main()`:

```
    BankBase.pay_with_verify("Mike", "Lin", 100.0);
    BankBase.pay_with_verify("Jason", "Mike", 150.0);
    BankBase.pay_with_verify("Lin", "Mike", 50.0);

    BankBase.get_transaction_history("Mike");
```

Since Jason doesn't have enough money to pay $150 to Mike, so, his transaction is rejected (failed).

```
Not enough money for Jason's payment.
Mike sent 100.0 to Lin on 2021-11-03 01:27:37.
Mike received 50.0 from Lin on 2021-11-03 01:27:37.
```

Finally, our `balance` table looks like this:

	account	balance
1	Mike	450
2	Jason	100
3	Lin	250

For Question c)

To test the function:

```
#[test]
pub fn test_pay() {
    use super::*;
    use sqlite::State;

    let init_db = UserBase {
        fname: String::from("./data/users.db")
    };
    let conn = sqlite::open(&init_db.fname).unwrap();

    init_db.clear_database(); // init the database
    // add user
    init_db.add_user(&String::from("Mike"),
&String::from("123456"));
```

```
    init_db.add_user(&String::from("Jason"),
&String::from("123456"));
    // add funds
    init_db.save_money("Mike", 500.0);
    init_db.save_money("Jason", 100.0);
    // make a payment
    init_db.pay_with_verify("Mike", "Jason", 100.0);
    // find the payment record
    let mut st = conn.prepare("select * from transactions where
u_from = ?;").unwrap();
    st.bind(1, "Mike").unwrap();
    // whether the payment record is the same as what we expect
    while let State::Row = st.next().unwrap() {
        assert_eq!(String::from("Mike"), st.read::<String>
(0).unwrap());
        assert_eq!(String::from("Jason"), st.read::<String>
(1).unwrap());
        assert_eq!(String::from("100.0"), st.read::<String>
(3).unwrap());
    };
}
```

The output:

```
running 1 test
test tests::test_pay ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
filtered out; finished in 4.08s
```

Question 2

For the `impl UserBase{}`:

```
impl UserBase {
    pub fn clear_database(&self) {
        // Use before thinking!!!
        let connection =
sqlite::open("./data/users.db").unwrap();
        connection
            .execute(
```

```
                "
            delete from users;
            delete from balance;
            delete from transactions;
            ",
                )
                .unwrap();
        }

        pub fn add_user(&self, u_name: &str, p_word: &str) ->
Result<(), UBaseErr> {
            let conn = sqlite::open("./data/users.db")?;
            let hpass = bcrypt::hash(p_word, DEFAULT_COST)?;
            // Add info to users
            let mut st = conn.prepare("insert into users(u_name,
p_word) values (?,?);")?;
            st.bind(1, u_name)?;
            st.bind(2, &hpass as &str)?;
            st.next()?;
            // init balance with 0
            let mut st2 = conn.prepare("insert into
balance(account, balance) values (?,?);")?;
            st2.bind(1, u_name)?;
            st2.bind(2, 0.0)?;
            st2.next()?;

            Ok(())
        }

        pub fn save_money(&self, u_name: &str, amount: f64) ->
Result<(), UBaseErr> {
            let current_balance = self.get_balance(u_name);
            let new_balance = amount + current_balance;
            let conn = sqlite::open("./data/users.db")?;
            // add info to balance table
            let mut st2 = conn.prepare("update balance set
balance=? where account=?;")?;
            st2.bind(1, new_balance)?;
            st2.bind(2, u_name)?;
            st2.next()?;
            Ok(())
        }
```

```
    pub fn pay_with_verify(&self, u_from: &str, u_to: &str,
amount: f64) -> Result<(), UBaseErr> {
        let conn = sqlite::open("./data/users.db")?;
        // get the balance from sender and receiver
        let balance_sender = self.get_balance(u_from);
        let balance_receiver = self.get_balance(u_to);
        // whether the sender has enough balance to pay?
        if balance_sender < amount {
            println!("Not enough money for {}'s payment.",
u_from); // if no
        } else {
            // if yes
            // update the transaction table
            let mut st = conn.prepare("insert into transactions
(u_from, u_to, t_date, t_amount)
values(?,?,datetime(\"now\"),?);")?;
            st.bind(1, u_from)?;
            st.bind(2, u_to)?;
            st.bind(3, amount)?;
            st.next()?;
            // update the balance table
            let mut st_update_sender = conn.prepare("update
balance set balance=? where account=?;")?;
            st_update_sender.bind(1, balance_sender - amount)?;
            st_update_sender.bind(2, u_from)?;
            st_update_sender.next()?;

            let mut st_update_receiver = conn.prepare("update
balance set balance=? where account=?;")?;
            st_update_receiver.bind(1, balance_receiver +
amount)?;
            st_update_receiver.bind(2, u_to)?;
            st_update_receiver.next()?;
        };
        Ok(())
    }

    pub fn get_balance(&self, account: &str) -> f64 {
        let mut balance = 0.0;
        let conn = sqlite::open("./data/users.db").unwrap();
        let mut st = conn
            .prepare("SELECT * FROM balance WHERE account =
?;").unwrap();
```

```
        st.bind(1, account);

        while let State::Row = st.next().unwrap() {
            let account_name = st.read::<String>(0).unwrap();
            if account == account_name {
                return st.read::<String>(1).unwrap().parse::
<f64>().unwrap();
            }
        }
        balance
    }

    pub fn get_transaction_history(&self, u_name: &str) ->
Result<(), UBaseErr> {
        // Establish the connection
        let conn = sqlite::open("./data/users.db").unwrap();

        // check the entry for selected "sender"
        let mut st_send = conn
            .prepare("SELECT * FROM transactions WHERE u_from =
?;").unwrap();
        st_send.bind(1, u_name);
        // print all results
        while let State::Row = st_send.next().unwrap() {
            let sender = st_send.read::<String>(0).unwrap();
            if u_name == sender {
                let u_from = sender;
                let u_to = st_send.read::<String>(1).unwrap();
                let time_stamp = st_send.read::<String>
(2).unwrap();
                let amount = st_send.read::<String>
(3).unwrap();
                println!("{} sent {} to {} on {}.", u_from,
amount, u_to, time_stamp);
            }
        };

        // check the entry for selected "receiver"
        let mut st_rec = conn
            .prepare("SELECT * FROM transactions WHERE u_to =
?;").unwrap();
        st_rec.bind(1, u_name);
        // print all results
```

```
        while let State::Row = st_rec.next().unwrap() {
            let receiver = st_rec.read::<String>(1).unwrap();
            if u_name == receiver {
                let u_from = st_rec.read::<String>(0).unwrap();
                let u_to = receiver;
                let time_stamp = st_rec.read::<String>
(2).unwrap();
                let amount = st_rec.read::<String>(3).unwrap();
                println!("{} received {} from {} on {}.", u_to,
amount, u_from, time_stamp);
            }
        };
        Ok(())
    }
    pub fn get_encrypt_pass(&self, u_name: &str) -> String {
        let connection = sqlite::open(&self.fname).unwrap();
        let mut st = connection.prepare("select * from users
where u_name=?").unwrap();
        st.bind(1, u_name).unwrap();

        let mut password = String::new();

        while let State::Row = st.next().unwrap() {
            //user_password(input) = user_password(db)?
            password = st.read::<String>(1).unwrap();
        }
        password
    }
    pub fn check_exist(&self, u_name: &str) -> bool {
        let connection = sqlite::open(&self.fname).unwrap();
        let mut st = connection.prepare("select * from users
where u_name=?").unwrap();
        st.bind(1, u_name).unwrap();

        let mut result = false;
        while let State::Row = st.next().unwrap() {
            let temp_name = st.read::<String>(0).unwrap();
            if temp_name == u_name {
                result = true;
            }
        }
        result
    }
```

```
}
```

For the `handle_input()` function:

```
fn handle_input(args: &String) -> String {
    println!("please input your({}) password", &args);
    let mut pass_input = String::new();
    std::io::stdin().read_line(&mut pass_input).expect("Cannot
read");
    pass_input.trim().to_string()
}
```

For the `run_command_line()` function: (Core Logic)

```
fn run_command_line() {
    let bank_base = UserBase {
        fname: String::from("./data/users.db"),
    };

    let args: Vec<String> = env::args().collect();
    let length = args.len(); // the first arg is in args[1]
    let key_word = &args[1];
    match key_word.as_str() {
        "new" => {
            if length != 4 {
                eprintln!("Wrong number of arguments. Follow
this: cargo run new [user] [password]");
                process::exit(1);
            } else {
                bank_base.add_user(&args[2], &args[3]);
                println!("Adding user {} with password {}",
&args[2], &args[3]);
            }
        }
        "transfer" => {
            if length != 5 {
                eprintln!("Wrong number of arguments. Follow
this: cargo run transfer [sender] [receiver] [amount]");
                process::exit(1);
            }
```

```
            if bank_base.check_exist(&args[2]) != true ||
bank_base.check_exist(&args[3]) != true {
                eprintln!("Name doesn't exits!");
                process::exit(1);
            } else {
                let pass_input = handle_input(&args[2]);
                if verify(&pass_input,
&bank_base.get_encrypt_pass(&args[2])).unwrap() {
                    println!("Sending money from {} to {}...",
&args[2], &args[3]);
                    bank_base.pay_with_verify(&args[2],
&args[3], args[4].parse::<f64>().unwrap());
                    println!("Operation done successfully!");
                } else {
                    eprintln!("Wrong Password!");
                    process::exit(1);
                };
            }
        }
        "balance" => {
            if length != 3 {
                eprintln!("Wrong number of arguments. Follow
this: cargo run balance [user]");
                process::exit(1);
            }
            if bank_base.check_exist(&args[2]) != true {
                eprintln!("Name doesn't exits!");
                process::exit(1);
            } else {
                let pass_input = handle_input(&args[2]);
                if verify(&pass_input,
&bank_base.get_encrypt_pass(&args[2])).unwrap() {
                    println!("Balance is {}",
bank_base.get_balance(&args[2]));
                    println!("Operation done successfully!");
                } else {
                    eprintln!("Wrong Password!");
                    process::exit(1);
                };
            }
        }
        // e.g. cargo run save Mike 1000
        "save" => {
```

```
            if length != 4 {
                eprintln!("Wrong number of arguments. Follow
this: cargo run save [user] [amount]");
                process::exit(1);
            }
            if bank_base.check_exist(&args[2]) != true {
                eprintln!("Name doesn't exits!");
                process::exit(1);
            } else {
                let pass_input = handle_input(&args[2]);
                if verify(&pass_input,
&bank_base.get_encrypt_pass(&args[2])).unwrap() {
                    println!("Adding ${} to account {}...",
args[3].parse::<f64>().unwrap(), &args[2]);
                    bank_base.save_money(&args[2],
args[3].parse::<f64>().unwrap());
                    println!("New balance is {}",
bank_base.get_balance(&args[2]));
                    println!("Operation done successfully!");
                } else {
                    eprintln!("Wrong Password!");
                    process::exit(1);
                };
            };
        }
        _ => {
            eprintln!("Wrong Operation, please try again...");
            process::exit(1);
        }
    }
}
```

For the `main()` function:

```
fn main() {
    let BankBase = UserBase {
        fname: String::from("./data/users.db"),
    };

    run_command_line();
}
```

We will do the following:

First, add some users:

```
> cargo run new Mike 123456
Adding user Mike with password 123456

> cargo run new Jack 123456
Adding user Jack with password 123456

> cargo run new Sarah 123456
Adding user Sarah with password 123456
```

	u_name	p_word
1	Mike	$2b$12$9LxQ5LLBhBlJ8AjbS2…
2	Jack	$2b$12$7cKYbAKyBrlkAe6nIS…
3	Sarah	$2b$12$zX4WzPlKN8nNFNXf6v…

Second, add funds to users' accounts: *(Test of wrong number of arguments is included here)*

```
> cargo run save 1
Wrong number of arguments. Follow this: cargo run save [user]
[amount]

> cargo run save Jack 50
please input your(Jack) password
123456
Adding $50 to account Jack...
New balance is 50
Operation done successfully!

> cargo run save Mike 500
please input your(Mike) password
123456
Adding $500 to account Mike...
New balance is 500
Operation done successfully!
```

	account	balance
1	Mike	500
2	Jack	50
3	Sarah	0

Third, we make some transfer (payment): *(Tests of wrong name, wrong amount are included here)*

```
> cargo run transfer Mike Lin 10
Name doesn't exits!

> cargo run transfer Sarah Mike 10
please input your(Sarah) password
123456
Sending money from Sarah to Mike...
Not enough money for Sarah's payment.
Operation done successfully!

> cargo run transfer Mike Sarah 100
please input your(Mike) password
123456
Sending money from Mike to Sarah...
Operation done successfully!
```

	u_from	u_to	t_date	t_amount
1	Mike	Sarah	2021-11-03 07:23:38	100.0

Finally, let's check the balance: *(Tests of wrong password and wrong arguments are included here)*

```
> cargo run balance Sarah
please input your(Sarah) password
123456
Balance is 100
Operation done successfully!

> cargo run balance Jack
please input your(Jack) password
345
Wrong Password!

> cargo run test Jim
Wrong Operation, please try again...
```

	account	balance
1	Mike	400
2	Jack	50
3	Sarah	100

Question 3

The key operations are replacing `bcrypt` with `rust-argon2`.

First, we need to remove all operations that handled by `bcrypt`. And then, import the new one.

```
[dependencies]
sqlite = "0.26.0"
rust-argon2 = "0.8"
```

```
use argon2::{self, Config};
```

Next, update the method for password encryption.

```
pub fn add_user(&self, u_name: &str, p_word: &str) ->
Result<(), UBaseErr> {
        let conn = sqlite::open("./data/users.db")?;
        let config = Config::default(); // load hash config
```

```
        let salt = "randomsalt".as_bytes();
        let hash = argon2::hash_encoded(p_word.as_bytes(),
salt, &config).unwrap(); // hash the password
        // Add info to users
        let mut st = conn.prepare("insert into users(u_name,
p_word) values (?,?);")?;
        st.bind(1, u_name)?;
        st.bind(2, &hash as &str)?;
        st.next()?;
        // init balance with 0
        let mut st2 = conn.prepare("insert into
balance(account, balance) values (?,?);")?;
        st2.bind(1, u_name)?;
        st2.bind(2, 0.0)?;
        st2.next()?;

        Ok(())
    }
```

Then, update the password verification method.

```
let pass_input = handle_input(&args[2]);

if
argon2::verify_encoded(&bank_base.get_encrypt_pass(&args[2]),
pass_input.as_bytes()).unwrap() {
    // Password Correct. Do the operation.
} else {
    // Password is wrong. Operation rejected.
};
```

Do the same command line tests as the Question 2, the results are same. All functions are working properly. For details, please refer to the souce code.