Project Design

CMSC 495 6380

6/5/2021

Group 5

Summer Smith, Sean Dwyer, Keith Combs

**Revision History**

|  |  |  |
| --- | --- | --- |
| **Date** | **Name** | **Description** |
| 6/13 | Summer | Created document with cover page, revision history table, and outline.  Initial pseudocode for interest accrual subsystem. |
| 6/14 | Summer | Create event trace diagram for interest accrual subsyem |
| 6/14 | Sean | Added event trace diagrams and pseudocode for authentication subsystem |
| 6/14 | Keith | Added event trace diagrams, scenarios and pseudocode for transaction subsystem |
| 6/15 | Keith | Added database schema and library functions |
| 6/15 | Sean | Added additional risk/risk mitigation information |
| 7/11 | Keith | Updated transactions to reflect changes in programming |

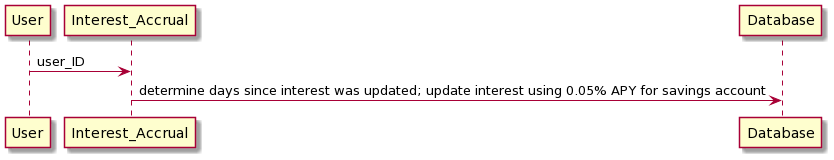
**Event Trace Diagram**

*Scenario 1:*

Description: the user logs in and the interest accrual system updates the interest value if it hasn’t been updated that day.

Pre-Condition: user is logged in.

Post-Condition: interest value is updated to reflect total interest (as of today).

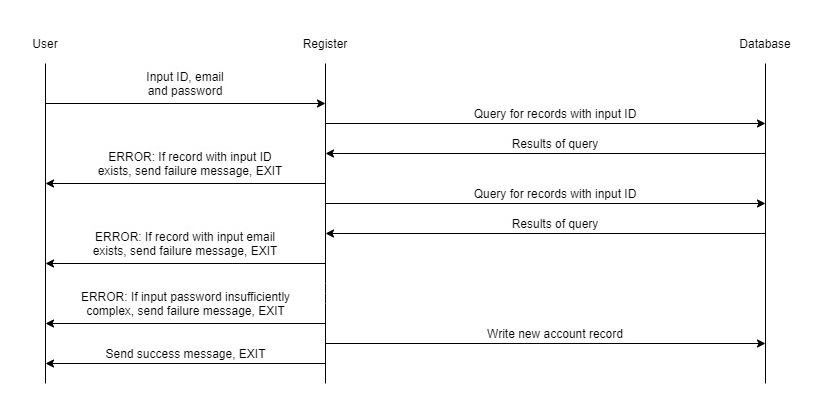


*Registration Scenario:*

Description: User enters user ID, password, and email address and is registered by the app as a new account.

Precondition: The web app is running and accessible.

Post-condition: A new database record has been made with the entered user ID, password, and email address, and account values have been initialized to zero; the user is notified of registration success.

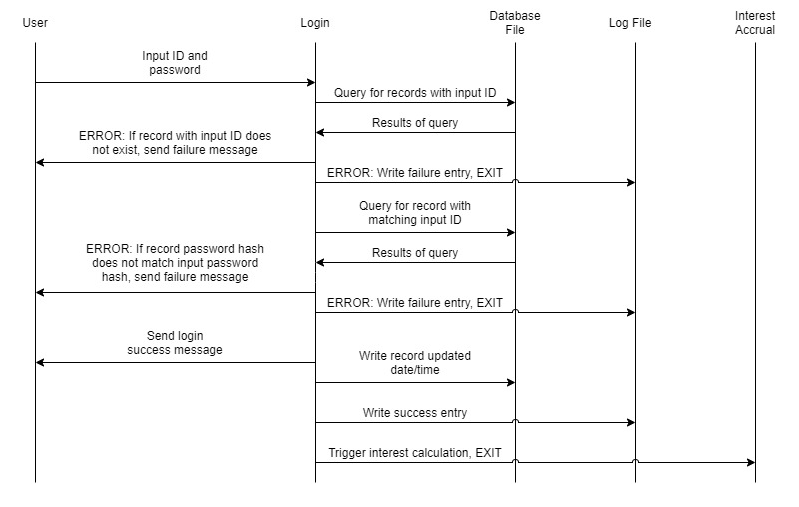
****

*Login Scenario:*

Description: User enters user ID and password, and is logged on if matched to an existing account.

Precondition: The web app is running and accessible; the user has previously been registered in web app.

Post-condition: User is notified of logon success and redirected to home; user session is recorded as logged in; logon attempt is logged; interest accrual is triggered.

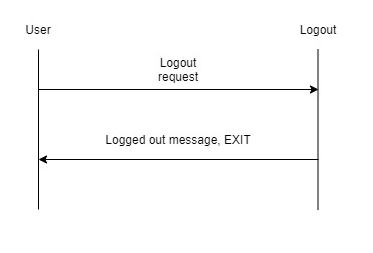


*Logout Scenario:*

Description: User triggers logout.

Precondition: The web app is running and accessible; the user is logged in.

Post-condition: User is notified of logout; user session is cleared.

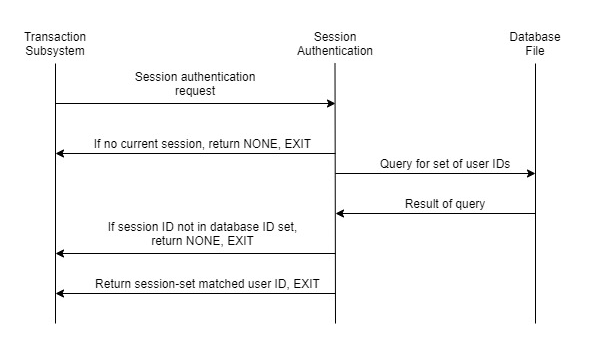


*Session Scenario:*

Description: Current session is verified as logged in.

Precondition: The web app is running and accessible; the user has logged in successfully.

Post-condition: Session logon status is returned.

****

*Withdrawal Transaction Scenario*

Description: User is logged in and chooses to withdraw money.

Pre-Condition: User is logged in and wants to withdraw money.

Post-Condition: Reduce money in the account and show balance.

1. Evaluate Login
2. Show Bank Account Balance
3. Select withdraw
   1. Enter amount to withdraw
   2. Confirm
4. Check if there is enough money to withdraw from the balance
   1. If not, show alert, withdraw and add $20 fee
   2. Else reduce money to back acount
5. Show Bank account balance
6. Back to home screen

Graphical user interface, application

Description automatically generated

*Deposit Transaction Scenario*

Description: User is logged in, chooses deposit in the account.

Pre-Condition: User is logged in and wants to deposit money.

Post-Condition: Add money to the account and show balance.

1. Evaluate Login Status
2. Show bank account balance
3. Select deposit
   1. Enter Amount to deposit
   2. Confirm
4. Show bank account balance
5. Back to home screen

Graphical user interface, diagram, application

Description automatically generated

*Transfer Transaction Scenario*

Description: User is logged in chooses accounts and transfers money between accounts.

Pre-Condition: User is logged in and wants to transfer money from account A to account B.

Post condition: Balance of Account A decreases while Account B’s increase.

1. Evaluate Login Status
2. Show account balances
3. Select Transfer
4. Choose accounts and direction of transfer
5. Show Account balance of Account A and Account B
6. Enter amount to transfer
   1. Confirm
7. Check if there is enough money to transfer from the balance of account A
   1. If not, show alert, transfer money and add $20 fee
   2. Else reduce money in Account A and Add money to Account B
8. Show bank account balances
9. Back to home screen

Diagram

Description automatically generated with medium confidence

**Class Design**

1. Authentication Subsystems

Class register {

database.connect() // standard library database API

Void register() {

user\_ID = request.form(“User ID”) // framework feature

password = request.form(“Password”) // framework feature

email = request.form(“Email”) // framework feature

user\_set = database.fetchall(“User ID”) // standard library database API

if user\_ID in user\_set {

render(“Registration failure”) // framework feature

return

}

email\_set = database.fetchall(“Email”) // standard library database API

if email in email\_set {

render(“Registration failure”) // framework feature

return

}

if not check\_password(password) {

render(“Registration failure”) // framework feature

return

}

database.execute(user\_ID, hash(password), email, checking=0, savings=0, interest=0) // standard library database API

database.commit() // standard library database API

database.close() // standard library database API

render(“Registration success) // framework feature

redirect(“Login page”) // framework feature

return

}

bool check\_password(password) {

if(password length > 12 AND

password uppercase >= 1 AND

password lowercase >= 1 AND

password numbers >= 1 AND

password special characters >= 1) {

return true

}

return false

}

string hash(password) {

return sha256\_crypt(password) // framework feature

}

}

NOTE: Logout Scenario/functionality is trivial and is included in this class as a simple method; a logged in user will simply have their session cleared and be notified of logging out. An authenticated user will be unable to access the logout page in the first place.

Class logon {

Void login() {

session.clear() // framework feature global

user\_ID = request.form(“User ID”) // framework feature

password = request.form(“Password”) // framework feature

user\_set = database.fetchall(“User ID”) // standard library database API

if user\_ID not in user\_set {

render(“Login failure”) // framework feature

log(IP, user\_ID, hash(password), time, “failure”)

return

}

stored\_hash = database.fetch(user\_ID)[“password”] // standard library database API

if hash(password) != stored\_hash {

render(“Login failure”) // framework feature

log(IP, user\_ID, hash(password), time, “failure”)

return

}

session.set(user\_id) // framework feature global

render(“Login Success”) // framework feature

redirect(“Home page”)

log(IP, user\_id, hash(password), time, “success”)

return

}

Void logout() {

Session.clear() // framework feature global

Render(“logged out”) // framework feature

Redirect(“Login page”) // framework feature

Return

}

string hash(password) {

return sha256\_crypt(password) // framework feature

}

Void log(ip, id, hash, time, status) {

Open(log\_file)

Writeline(ip, id, hash, time, status)

Close(log\_file)

}

}

Class session\_authentication {

String verify() {

user\_id = session.get(“user\_id”) // frame feature global

if user\_id = None {

return None

}

user\_set = database.fetchall(“User ID”) // standard library database API

if user\_ID not in user\_set {

return None

}

return user\_id

}

}

1. Transaction Subsystems
2. Session\_authentication.verify() // Framework Feature
   1. Evaluates login status
   2. Returns accountID
3. withDrawAmount(float amount, accountType, userID){
   1. if(amount is <= to balance in account){

 i.     reduce account by amount

* 1. else

i.     overDraftAlert()

* + - 1. Display alert to user
      2. Subtract $20 from amount

 ii.     Reduce account by amount

 iii.     UpdateDB( accountID, amount)

* + - 1. If(accountID exists)
         1. Update balance
      2. Else
         1. Error
  1. Render page

1. depositAmount(float amount, accountType, userID){

i.     Add amount to balance

ii.     UpdateDB(int accountID, float balance)

* + - 1. If(accountID exists)
         1. Update balance
      2. Else
         1. Error
    1. Render page

1. transferAmount(float amount, accountType, userID){

i.     If accountID1 has >= amount

* + - 1. UpdateDB(accountID1, amount)
      2. UpdateDB(accountID2, amount)
    1. Else
       1. overDraftAlert()
       2. Return Error
    2. Render page

1. Interest Accrual Subsystem

Class Interest {

database.connect() //standard library database API

APY = .005

APD = APY/365

Void interest(user\_ID){

delta = date.today() – database.fetchall(user\_ID)[“last\_updated”]

if delta.days > 0{

collect\_interest = (APD \* delta.days) \* database.fetchall(user\_ID)[“savings”]

new\_interest = database.fetchall(user\_ID)[“interest”] + collect\_interest

database.execute(user\_ID, interest = new\_interest, last\_updated = date.today()) //standard library database API

log(IP, user\_ID, time, “interest updated”)

database.commit() //standard library database API

}

database.close() //standard library database API

return

}

1. Database System

NOTE: This is not a subsystem for development under this project. Rather these are the built-in or library database APIs natively available for Python and will be integrated throughout the other subsystems when implemented. The database itself is merely the database file; interactions are handled by the API calls made in the project-developed subsystems.

sqllite3/sqlalchemy library

Database schema{

user\_ID: string

password: hashed string

accountID: int

email: string

checking: float

saving: float

}

Database.fetchall()

Database.commit()

Database.execute()

Database.UpdateDB()

**Risk and Risk Mitigation**

For the transaction server, error handling on the input is caught in the browser input text box. For the withdrawal, deposit, and transfer the float input will be caught in the browser and a second check on the input before it is put in the database. The account selection is returned from the database and errors are handled by limiting the ability to select pre-created accounts.

As a general note, the HTML form object built in to the webpage can itself be set to prevent null inputs; as such, error/risk from null inputs is handled at the HTML request level, before it reaches the project-developed code base.

Many of the risks, and potential mitigations, identified in the Project Analysis are not addressed here. The mitigations for those risks are largely either implementation features of the web application framework, and thus not visible in a class design context, utilize external security tools, or are implemented as information displayed to the user, which will be implemented in HTML templates and not in these classes.

There are two particular sets of risks worth noting here, however. First, password complexity and storage are addressed in the registration class in the authentication module. Password complexity is enforced by the logic in the register() method. Password storage is similarly addressed through the hash() method and logic in the register() method that makes it clear that the password information written to the database file only occurs after it is hashed. Additionally, note that comparisons in the login functionality are made between user input passwords that are hashed and the stored hashes themselves; there is no cleartext-to-cleartext comparison possible.

The second risk category raised is for SQL injections. This is mitigated primarily through the use of proper design patterns in implementing the database access and write calls through the database API. While the crudest design patterns could allow for code injections into the SQL statements, appropriate variable substitution handling is trivial to implement and immediately verifiable as providing input sanitization prior to executing the SQL statement. These mitigations are not visible here since, again, they are executed in implementation, but are noted for their significance and will be specifically annotated in code once implemented.