SDD - eBanking Web App

Group Four
Anthony DeDominic <dedominica@my.easternct.edu>
Courtney Combs <combsco@my.easternct.edu>
Jeremy Drexler <drexlerj@my.easternct.edu>
Kevin Bailey <baileyk@my.easternct.edu>

May 6, 2016

Contents

i. Revisions

Since the documents for this group are version controlled and written in a plaintext, human readable, format, changes to the SRS can be found at https://github.com/adedomin/CSC385-Documents/commits/master

Commits clearly show the sections that were added, changed, or even removed from documentation.

1. Overview

1.1. Scope

This SDD is to define most of the high level patterns for the ebanking project and designs to be used in our banking application. The specification will show the designs of the user interface, objects, architectural and database components.

2. Definitions

The following definitions may be strewn across the document.

- 2.1. DAO: Data Access Object. An object pattern that abstracts database calls so that other databases could be used without changing other code.
- 2.2. MVC: Model View Control, Design pattern that allows for object abstraction
- 2.3. Node.js: Javascript for backend servers.
- 2.4. express.js: a Node.js framework to simplify the creation of HTTP driven web applications.
- 2.5. MongoDB: A NoSQL database that is highly scalable and performant.
- 2.6. LevelDB: A basic, embedded key:value, database that is entirely implemented in Javascript
- 2.7. Redis: A client - server key:value database that is distrubatble.
- 2.8. Factory: pattern that defines objects that return new objects. Likely setting some default parameters.

- 2.9. middleware: Business logic that is broken into discrete steps. Such as login, getting data and otherwise.
- 2.10. controllers: Business logic to handle changes to underlying models; in the context of the application, middleware and controllers are very similar.
- 2.11. Dependency Injection: Requires client objects to find their needed object dependencies.

3. Object Design

3.1. DAOs

Data is a huge part of the application. In order to abstract data access from the code, the Data Access Object design pattern should be used.

Below is a list of needed objects for accessing the various datas for the application.

3.1.1. UserDAO

The UserDAO should implement the following interface.

- getUser(String name, function)
- saveUser(User user, function)
- deleteUser(String name, function)

Please see definitions for what the verbs (get, save, delete) mean in this context.

All of the above should take a callback as an argument.

The callback arguments must take the following.

- getUser() -> function (err, user)
- saveUser() -> function (err)
- deleteUser() -> function (err, user)

err is null if no error, as is common in most JavaScript (Node.js) applications. Otherwise it is an object which contains error information.

user is the User object that was retrieved or removed.

The UserDAO should include member variables that make sense for the object.

- db -> some kind of database connector

Other variables may be required, for instance for Relational, SQL like databases, it may be necessary to create Query Factories.

in that case:

- selectUserQueryFactory()
- updateUserQueryFactory()
- deleteUserQueryFactory()

These factories should return query objects that will properly handle sanitizing user inputs by using some prepared statements.

3.1.2. AccountDAO

This DAO should implement the following interface.

- getAccount(String accountId, function)
- saveAccount(Account account, function)
- deleteAccount(String accountId, function)

Please see definitions for what the verbs (get, save, delete) mean in this context.

The callback arguments must take the following.

- getAccount() -> function (err, account)
- saveAccount() -> function (err)
- deleteAccount() -> function (err, account)

err is null if no error, as is common in most JavaScript (Node.js) applications. Otherwise it is an object which contains error information.

account is the Account object that was retrieved or removed.

The AccountDAO should include member variables that make sense for the object.

- db -> some kind of database connector

Other variables may be required, for instance for Relational, SQL like databases, it may be necessary to create Query Factories.

in that case, these query factory objects MUST implement the following functions:

- selectAccountQueryFactory()
- updateAccountQueryFactory()
- deleteAccountQueryFactory()

These factories should return query objects that will properly handle sanitizing user inputs by using some prepared statements.

3.1.3. HistoryDAO

This DAO shall access user's transaction and other account changes.

- getHistory(String accountId, function)
- saveHistory(History history, function)
- deleteHistory(String historyId, function)

Please see definitions for what the verbs (get, save, delete) mean in this context.

The callback arguments must take the following.

- getHistory() -> function (err, history)
- saveHistory() -> function (err)
- deleteHistory() -> function (err, history)

err is null if no error, as is common in most JavaScript (Node.js) applications. Otherwise it is an object which contains error information.

History is the History object that was retrieved or removed.

The HistoryDAO should include member variables that make sense for the object.

- db -> some kind of database connector

Other variables may be required, for instance for Relational, SQL like databases, it may be necessary to create Query Factories.

In that case, these query factory objects MUST implement the following functions:

- selectHistoryQueryFactory()
- updateHistoryQueryFactory()
- deleteHistoryQueryFactory()

These factories should return query objects that will properly handle sanitizing user inputs by using some prepared statements.

3.1.5. TransactionsDAO

This DAO should only be used by core logic to persist pending transactions.

- getAllTransactions(function)
- setNewTransaction(Transaction transaction, function)
- deleteTransaction(String transactionid, function)

This object is special, as it is intended to persist potentially interrupted transactions.

The callback arguments must take the following:

- getAllTransactions() -> function (err, transactions)

- saveNewTransaction() -> function (err, transaction)
- deleteTransaction() -> function (err)

The callback for getAllTransactions should be expected to register all the transactions to be executed in the future.

3.2. Data Objects

3.2.1 User

The user object is the object that holds all the pertinent user information.

It should include all the following data:

- String username
- String email <- may be same as above
- String hashedPassword <- includes salt, rounds and algorithms used.
- [String] accountIds

The array merely points to id's of the owned account objects, this is practical since accounts can be shared and have no set owner.

3.2.2. Account

The account object defines an account with money and other such objects:

- String AccountId
- Boolean isInternal
- String bankNumber
- String routeNumber
- String type
- ACL acl <- access control list
- DebitCard debitCard
- Decimal balance

The "isInternal" field should determine the behavior of the object. For instance, only internal accounts should have a debit card or a balance. Balance for external accounts may be possible but also may not be; because of this, they should not be set. Only in external accounts is the bankNumber truly relevant, but for consistency, all internal accounts should include out bank number.

The Decimal object should implement Binary Coded Decimals to allow for high accuracy, arbitrary precision calculations. Types, like double, simply can't give the

granularity for accurate withdraws or deposits from an account. decimal.js look's promising as it does just that.

This object should make no logic for such functionality

The ACL should be some kind of list of users and their roles on the account. They should be in a form like:

```
{
    <username>: [role1, role2]
}
```

roles should be one of the following:

- read, they can see the account's details
- send, they can send money to the account, but not necessarily see the details.
- charge, they can charge up to a limit, should be reversible
- owner, they can completely control the account, by default the user who created the account has this role.

Debit card should be an object that includes an indeterminate member variable/s that describe an ATM card for the account. It should at least have a card number.

3.2.3. Transactions

The transaction object shall allow a user to defer or to set a recurring transfer from one account to another.

In the bill pay type, the job is a recurrent transfer to an organization's account.

- String TransactionId
- Boolean isRecurring
- Date startAt <- when to fire off
- String to <- an account id
- String from <- an account id
- Decimal transferAmount
- String notes <- optional

3.2.4. History

- String accountId <- account effected
- Date date <- date of transaction
- String notes <- what happened
- Decimal amount <- changed amount

The notes field should have a message that describes what happened. For instance:

```
%TYPE - %USER_ACCOUNT to %OTHER_ACCOUNT
```

3.3. Controllers and Middlewares

3.3.1 Helper Controller

This controller shall offer helpful middlewares that are shared among all the controller. This section shall discuss the necessary design and functions that should be in this object.

- generateToken(next)
  - This function shall make use of the node.js crypto primitives to generate a 48 bytes of random data. The next() shall be a callback which takes the parameters of err and token. If err is not null, assume that something went wrong. For ease of use, the function will convert the random bytes into a hexadecimal string.
- getUser: function (username, next) {
  - Refer to the DAO's in section 3.1.1. This function is more for abstracting dependencies.
- updateUser: function (username, changes, next)
  - Refer to DAO 3.1.1. This function gets the user using the DAO and then saves changes passed in as a parameter.
- addAccountToUser: function (username, accountid, next)
  - Refer to DAO 3.1.1. This function does similar to above but it adds an account specifically.
- removeUser: function (username, next)
  - Refer to DAO 3.1.1.
- getSession: function (sessionid, next)
  - This function is used to verify session information the user passed in. Specifically it will make a database call to the session db. If a value is returned, the session is determined to be accurate.
- setUser: function (user, next)
  - Refer to DAO 3.1.1. This is just a saveUser() abstraction for dependency reasons.
- setSession: function (session, next)
  - Creates a new session by storing the user's name in association with a randomly generated token.
- removeSession: function (sessionid, next)
  - Remove a session by Id.
- getAccount: function (accountid, next)
  - Abstraction of DAO 3.1.2. Fetches an account by its id.
- checkAccountAccess: function (username, account)

  - This function will read the ACL and return if the user has access or not.

- transferAccountBalance: function (to, from, change, next)
  - Transfers amount by manipulating the two accounts and saving them using the account DAO (3.1.2). It also creates two new histories to save using the History DAO (3.1.3).
- createAccount: function (account, next)
- updateAccount: function (account, next)
- getTransactions: function (owner, next)
- addTransaction: function (transaction, next)
- removeTransaction: function (transactionId, owner, next)
- checkPass: function (password, hash, next)
- hashPass: function (password, next)
- getHistory: function(accountId, next)

4. Object Interactions

To be implemented

5. Architecture

5.1 Architecture Overview

The architecture of the e-banking system needs to allow a user to communicate through a web interface, authenticate and manage customer data, and allow transactions in real time (while putting a stop to any transaction in which an error occurs such as insufficient funds, or a hold being placed on the account). For those reasons we decided to use a three tier online transaction processing architecture, which is often used for online banking architecture. By using this model we allow ourselves to avoid potential bottlenecks that may arise if we were instead using a client-server architecture. [1]

5.2 Presentation layer

The front end presentation layer (the e-banking website) is detailed in section 4, and describes the functionality and various actions that the customer can take while

[1] See Figure 1:2.1.1 in the SRS

interacting with the e-banking system, highlighting the usability of the service. It also shows what the interface may end up looking like.

5.3 Domain Layer

This is where authentication of user information will take place as well as data processing/management.

5.4 Technical Services Layer

A back-end MongoDB database which will store user and transaction data. This will be accessed by the user when a request has been made and their information verified. The entities and their relationships are detailed in section 3.