Transaction Cordinator: Design and Planning

Joshua Lee, Damon McCormick, Kim Ly, Chris Orimoto,
John Wang, and Daniel LeCheminant

October 4, 2004
## Contents

1 Overview 2

2 Document Revision History 2

3 System Architecture 2
   3.1 Overview 2
   3.2 Database Design 3
   3.3 Table Descriptions 3
   3.4 Variable Names 4
   3.5 API Design 6

4 Design Details 6
   4.1 Protocols 8
   4.2 Algorithms 8
   4.3 Documents for Download 9

5 Testing Plan 9
   5.1 Unit Testing 9
   5.2 Integration Testing 9
   5.3 System Testing 9
   5.4 Regression Testing 9

6 Plan 10
   6.1 First Build 10
   6.2 Second Build 12
   6.3 Third Build 13
   6.4 Full project 13
   6.5 Goals for Testing Plan 13
1 Overview

One of the design decisions made was that our system would be composed of three main modules: The database, the website, and the data processing modules. The website and data processing modules will run on logic written in PHP. The database will be written in MySQL. The interface between these three modules will be clearly defined in an API created and distributed by the database group. This API will specify the functions which we, as the data processing and user interface (website) group, can expect to use. This allows for well-informed parallel development.

2 Document Revision History

Revision 1.2.1 Minor modifications, released to public

Revision 1.2.0 Incorporated updated database info, released for internal review only

Revision 1.1.0 Incorporated updated GUI info, released for internal review only

Revision 1.0.0 Initial version, released for internal review only

3 System Architecture

3.1 Overview

The Transaction coordinator Assistant (TheTA) is implemented via a PHP web-client front-end interacting with a backend MySQL database. A PHP-enabled Apache web server provides the end user with an interface to TheTA. Therefore, the only requirement placed upon the end users of TheTA is that they have a web-browser. Both the PHP User Interface and the server side Data Processing query data from and update to the MySQL database via a custom API. The API is founded upon the MySQL client libraries and guarantees certain invariants necessary for features provided by TheTA.

Figure 1: Information Exchange Diagram
3.2 Database Design

The tables in the database act as the central data storage area for the transaction coordinator. Replicating data between tables allows them to act as data structures in addition to providing the necessary links among data for relational logic essential to the transaction coordinator’s performance. The below diagram illustrates each table definition in the database and the links between each table. Each table is represented by a split rectangle with the table name in the upper half, and the column names in the lower half. Arrows represent links between tables. Solid and dotted lines are used to indicate a lack of intersection between links. A link indicates that same column names between different tables represent the same data.

Figure 2: Database Tables

3.3 Table Descriptions

There are five main types of tables: permission, description, type, comment, and backup.
Permission tables can be identified by the suffix \_Table on the end of their name. The session table, however, is not a permission table in the strict sense. The session table keeps track of which users are currently logged in, and for how long. In this manner, it can be seen as a table granting permission to login to the system.

Tables with the suffix \_Type describe a re-usable class of transaction, document, or user. For instance, if all transactions that involve selling a home will at least require the same set of some documents then a transaction type for home sales can be defined as a template that knows which documents are necessary. Future home sales can be easily setup by instantiating the home sale transaction type.

Tables describing and maintaining information about a specific user, transaction, or document can be identified by the suffix \_Desc attached to their name. Information contained in description tables can be viewed as instantiations of type tables.

The comment table stores end user comments about a document or transaction and links the comment to the appropriate document or transaction.

The backup type is not shown above. Each individual table has a backup table that has the exact same definition. Backup tables can be identified by the suffix \_Backup in their table name. The backup tables store the original version of a row from the table they mirror whenever the mirrored table’s data changes. Backup information is kept until deleted by an admin, thereby providing the ability to restore a transaction to any previous point.

### 3.4 Variable Names

**Index** a unique integer identifying each row of the database table.

**Transaction_id** a unique integer associated with a single transaction.

**Doc_id** a unique integer associated with a single document.

**User_id** a unique integer associated with a single user.

**Status** a string indicating the state a document, user, or transaction is in. Possible states for users include:

- pending deletion
- pending approval

Possible states for documents and transactions include:

- not started
- in progress
- pending approval
- done/approved
- cancelled
• pending deletion
• past due.

**Timestamp** string representing the latest date that the information in this row has been accessed in any way.

**Comment** string to store user comment.

**Session_id** an integer signature/hash unique to each logged in user.

**Login_time** the latest time that the user logged in to TheTA.

**Login** a string containing the end user’s login name.

**Password** a string containing a unique hash of the end user’s password.

**Email** a string containing the end user’s email address.

**Address** a string containing the end user’s mail address.

**Title** a string containing an easily recognizable title for a transaction or document type.

**Filename** a unique string containing the filename of the appropriate document or document type.

**Trans_type_id** a unique integer representing a type of a transaction.

**Doc_type_id** a unique integer representing a type of document.

**User_type_id** a unique integer representing a type of user.

**Read** a Boolean granting or preventing read access to transactions and documents.

**Write** a Boolean granting or preventing write access to transactions and documents.

**Create** a Boolean granting or preventing the ability to create documents, transactions, or users.

**Update** a Boolean granting or preventing the ability to update documents, transactions, or the status of either.

**Delete** a Boolean granting or preventing the ability to delete documents, transactions, users, or their types.

**Add_permissions** a Boolean granting the ability to give other users permissions.

**Remove_permissions** a Boolean granting the ability to revoke the permissions of other users.

**Accessed_by** the last user_id to access this user, document, or transaction.
Modified_by the last user_id to modify this user, document, or transaction.

Created_by the user_id that created this user, document, or transaction.

Access_time the last time this row was accessed.

Create_time the time this row was created.

Modify_time the last time this row was modified.

Parent_type the type of a specific row in one of the description tables.

Due_date the date that a document or transaction is due.

Date_completed the date that a document or transaction was finished.

3.5 API Design

An API built on top of the MySQL client libraries provides the interface to these tables. The API both removes the need for the User Interface and Data Processing programmers to know SQL syntax and guarantees the following two invariants: for every modification of a row, a backup is created and stored in the corresponding backup table or the modification does not continue; for every access to a row the timestamp on that row is adjusted to the current time. In the below diagram, methods provided by the API that interact with a table are listed in the bottom half of the table rectangle. Methods beginning with add or delete are the constructor and deconstructor methods for rows in that table. Methods beginning with get or set are the access and set methods for the information in a specific row of the table. In each case, a transaction_id, document_id, or user_id is provided by the calling function, and the requested information is returned to the calling function.

4 Design Details

Our web service’s homepage will branch out to 5 different sections (shown in 10.1 Specifications of the Proj3 document): Introduction, Personal (ForMe), Support, Search, and Controller.

The Personal section will include documents to download for the user such as HUD-1 RESPA (settlement statement) forms. The user could either print those documents provided or e-mail his own documents to some other user (perhaps his client). A calendar would also be provided for in this section. Users will be able to write simple memos on the calendar. We will most likely store calendar dates on the server, if this is too time consuming, then an algorithm might be written that verifies the accuracy of calendar dates.

We will try to implement a bulletin board system in the Support section where any active user using our web services can post questions or comments to each other, or the system operator/administrator. An e-mail function will also be available in Support. We might
create an algorithm that makes sure e-mail was received/sent. This algorithm at the most basic level will alert system administration of any mailer-daemons (bounced messages). At a more advanced level, for the convenience of the system administrator, we could also have the algorithm make suggestions of what the correct e-mail address should have been for the bounced message.

In the Search section, we will write an algorithm that searches through document so that the user can easily access and check document statuses.

The most important design will be for the Permissions aspect of our web service. We will need to design a secure method of creating new accounts and verifying users signing in with their corresponding account stored in our databases. Different permissions will be granted to different users depending on their account types. We will try to use the most secure kind of encryption codes available. We are currently considering using RSA
encryption/decryption. Those of us working on the database (or data processing) will need a way verify accounts and eliminate inconsistencies to prevent random pranksters from impersonating the real transaction coordinators or real estate agents. Different accounts will need to be coordinated to verify authenticities.

4.1 Protocols

- correctness of calendar dates
- no double-instances of documents (might be provided by database already)
- security
  - encryption of password
  - verification of authenticity (eliminate impersonators)
- permissions (4 levels of access)
  - transaction coordinator (full access)
  - agent (read/write access to own deal)
  - client (read access to own deal)
  - public (main page only, login, advertisements.)
  - login process
- send-mail protocol
  - should make sure e-mail was received/sent
  - should be able to catch mailer-daemons
- bulletin board system (public/private forum)
  - reliant on permissions protocol
- interaction between user interface and database
  - protocols for the user interface will follow the protocols developed for the database

4.2 Algorithms

- encryption algorithm (RSA)
- decryption algorithm (RSA)
- simple scheduling algorithm (enumerate all documents and deadlines and place in order)
- calendar checking algorithm
4.3 Documents for Download

- HUD-1 (RESPA, Real Estate Settlement Procedures Act)
- Miscellaneous (any that we haven’t considered yet, but surely exist)

5 Testing Plan

5.1 Unit Testing

Each programmer writes tests for his own functions/programs; he/she should know his own function/program best.

Communication:

- Keep in contact with team members whose functions/programs use your own
- Make sure testing functions simulate teammate’s function/program

5.2 Integration Testing

No order for the testing of each category (GUI, Data Processing, Database) No need for order, we are only using one kind of test per area

5.3 System Testing

Testbenches

- This will be done for simpler testing purposes
- More brute force than manual testing

Manual Testing

- This will be done for more complex testing procedures
- Testing GUI would be one example
- More extensive

5.4 Regression Testing

Retesting using testbenches

- Simple and quick
- Makefiles, etc

Retest ourselves

- More assurance
• Takes much longer time
• Have a teammate (who’s not busy) recheck then test

Much of our system will not be readily testable in an automatic manner. The GUI/Website side of our ThETA (The Transaction Assistant) service is not something that a program can test without the creation of an autominer/bot. Therefore, we will be manually testing processes that require the use of the GUI such as logging in. Since the administrator, client, and agent have different access permissions, we must also make sure that each version of their calendars is correctly displayed.

The Data Processing module will be tested in a black box manner. We have all of the important use cases listed in the Specification, section 6. Before programming, we will create a test for each of these not covered by the aforementioned manual testing. For example, submitting a document with relevant due date and permission settings; also searching for that document (after submission) via a document ID number, or due date, or keyword search. These tests will be built into a module that we will run for each daily/weekly version of our data procession module. We will use this set of tests as a measure of completion of the data processing module.

As a test interface for the data processing module, we will have a testing mode. This will allow for commands and printouts using an input file (text document) of pre-defined command line-like code to instruct the data processing module to run queries or manipulations on (documents, dates, titles, names, accounts) the database. Basically, a scripting capability.

White box testing will be done by each programmer on an individual, on-going basis.

6 Plan

For our software engineering model we decided to use the iterative model. That is, to design, prototype and evaluate, and then iterate.

6.1 First Build

Because the database backend is fundamental to the rest of the project, it is expected that we will need to at least have a subset of all of the database’s features functional before the first build can be tested.

Incorporate specs and reqs enumerated in first design document [2 weeks]

• Login
• Set up new accounts
• Update information (mostly used by the TC)
• Build calendar
• File sharing
• Finalize page layout designs [1 week]
• Make page skeletons using html [1/2 week]
• Tie pages together by inserting links [1/2 week]

The first build will focus on incorporating the must have cases specified in the first design documents which enumerates the requirements and specifications. This is estimated to take about 2 weeks. The GUI team depends on the server team to have a database running which will enable the GUI to implement the features below. These tasks are designated to Josh, Chris, John and Kim.

Most of these tasks for the GUI are independent of each other. Most of the dependency is interfacing with the server to make sure the data the GUI displays is correct.

The ordering of the GUI screens will be by importance:

1. update screen
   • This will be the central screen for the first build.
   • This screen is where the transaction coordinator updates all the information she has acquired.
   • This screen must communicate with the Sql server to update information for an individual transaction.
   • The reflected changes must be stored in the sql server and be accessed by the sendmail protocol to notify relevant parties.

2. calendar
   • This is our primary medium for conveying information to the user.
   • It depends on the ability to have a correct calendar object which can calculate dates correctly.

3. login
   • This enables all users to log in.
   • It is estimated to take 2 days to get a fully working login.
   • It should be able to recognize users and make sure each user has the correct privileges.
   • The front end has no dependency.
   • This just means creating a php page which is aesthetically appealing and intuitive for users to log into.
The back end requires SQL to recognize the user and confirm the submitted password matches and to load the correct profile to the user.

4. template creation

5. new accounts

- This task creates accounts.
- It should give a brief introduction to the customer of what the system can do.
- Then the user will be asked to create a profile.
- Information included will be the user name which will be unique, and a password.
- The SQL server must have the functionality to see if a duplicate name does not exist and if the characters entered are valid.
- A form must be created for the user to enter in their contact information: email, address, phone number.

6. file uploading/sharing.

- This will be the last task on the first build.
- A screen must be made where users can easily download files.
- The files must also be linked and stored on the server.
- We already have a server.
- This task is for the GUI team to figure out.

We will show the first build to the customer and create tasks once we receive feedback.

6.2 Second Build

This build will be to streamline operations and make sure the customer is satisfied with the bare minimum functionality specified in the spec documents. The database interface will be expanded and improved to provide full access to all planned features. Any modifications the customer proposes should be implemented. These should be minor changes in usability of the program.

- Write PHP to gather input from the pages. Details below: [2 weeks]

  home page  no input, only navigation
  login page  2 textfields
  new account  11 textfields
  today’s schedule  no input, only navigation
  calendar  1 textbox for each day of the month
announcements  no input, only navigation
agent checklist  1 textfield, 7 choices, 7 checkboxes, 1 textbox
transaction detail  1 textfield, 1 textbox, 7 choices, 6 checkboxes
e-mail forms to client
client view
  • Check inputs for errors (eg. malformed email addresses) [1 week]
  • Write MySQL queries to insert inputs collected in 4) to input into database [1/4 week]
  • Re-implement specs and reqs after getting client feedback and suggestions

6.3 Third Build
  • Write MySQL queries to fetch data from database and display on appropriate pages. [1/4 week]
  • Put data fetched from database into appropriate places on the pages. [1/4 week]
  • Optional cases/features [variable time]

6.4 Full project
This stage will be to implement optional cases. One case is the Address book estimated at 3 days and requires Sql server to store names and data. The GUI must also be built for the address book module. A useful case is Memo posting estimated to take 3 days and requires storage of message object on the server to store messages. The GUI must also add a memo module onto the existing pages where appropriate.

6.5 Goals for Testing Plan
Build 1  All pages look like final version. Navigation links working.
  1. User different browsers and verify that layouts are right on each one.
  2. Follow all links to check navigation paths are working correctly.

Build 2  All input can be collected from user and inserted into the database.
  1. Verify that each input area can accept data.
  2. Verify data checks are complete by putting in incorrect data.
  3. Verify that data made it to the database by inspecting database.

Build 3  Each page displays appropriate data in the appropriate place
  1. Make up several transactions.
2. Check all pages for appropriate data.

3. Change data in transactions and verify changed data is updated and displayed correctly.