Design and Planning Document

10/09/2014, version 1.0

Document Revision History

Rev 1.0 2014-10-9 Initial version

System Architecture

Our backend will be composed of a server created using the Django framework and a PostgreSQL database.

Database: We will be using a single PostgreSQL database to store the all of our data for this application.

Tables (Columns in bold are used as keys):

- User
  - Columns: **username** (string), password (string), cards (Card Table)
- Card
  - Columns: **companyName** (string), **jobTitle** (string), contacts (Contact Table), tasks (Task Table), notes (string), status (int)
- Task
  - Columns: **companyName** (string), **jobTitle** (string), type (string), deadline (datetime)
- Contact
  - Columns: **companyName** (string), **jobTitle** (string), contactName (string), contactTitle (string), contactEmail (email), contactPhone (int), contactLinkedIn (url),
- Document
  - docName (string), type (string),
- Tag

User Table

id (int)	username (string)	password (string)	email (string)

Card Table

id (int)	companyName (string)	jobTitle (string)	contacts (Contact Table)	tasks (Task Table)	status (int)	Tag (string)

User_Card Table

userId (int)	**cardId** (int)

Company Table

id (int)	companyName (string)

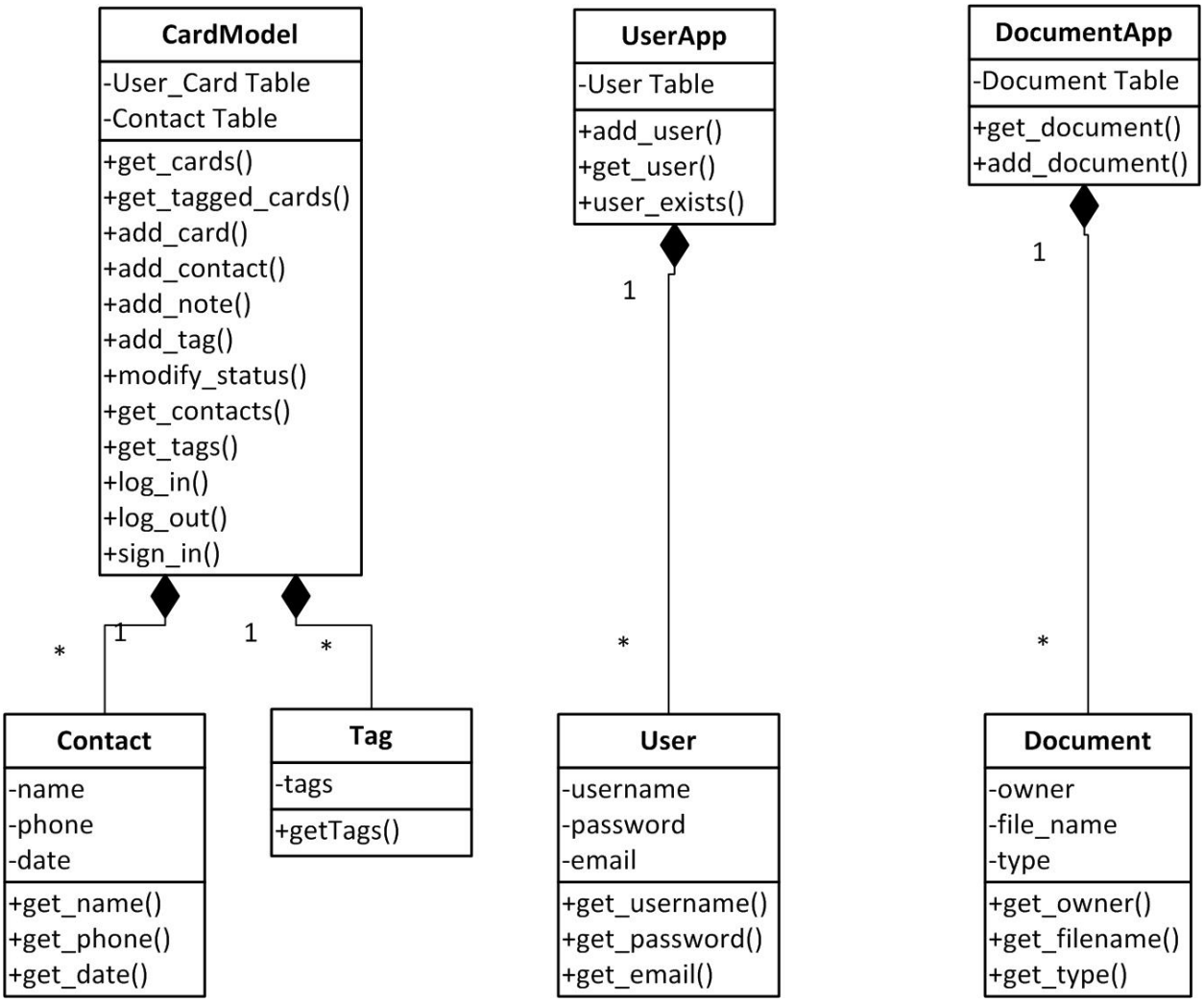
Contact Table

id (int)	companyName (string)	jobTitle (string)	contactName (string)	contactTitle (string)	contactEmail (email)	contactPhone (int)	contactLinkedIn (url)

Document Table

id (int)	docName (string)	type (string)	url (string)

Class Diagram



Design Details

Jobdex will be a web app.

Django views will set up most of the endpoints for the frontend to interact with. Using AngularJS models and services, we will be interacting with the REST API to query database data. Interactions that don't require database transactions will be handled by AngularJS. Routing will be handled through Django's routing system.

We might use some Django plugins to do things like easily set up a REST API (e.g. TastyPie, DjangoRestFramework). We will be using Gulp for build tasks and Git for version control.

Our frontend will be styled using LESS in development and compiled to a minified CSS for production, and AngularJS to split the Javascript logic into models, views, and controllers. Any hardcoded text in the template will be filled in by AngularJS and any dynamic text (such as the username) will be filled in by the Django templating system.

Routes (urls.py)
/card/<id>

CardApp

- **Model (models.py)**
  - User.objects().all().filter(id= 1)
    - Returns a User object (row of User)
  - Cards.objects().all().filter(owned=User)
    - Returns all Cards owned by user (can be many rows)
  - Tags.objects().all().filter(card=Card)
- **Controller (views.py)**
  - tags
    - add_tag(int card_id, String tag_name)
      - return int (1 for success, =<0 error)
    - modify_tag(int card_id, String current_tag_name, String new_tag_name)
      - return int (1 for success, =<0 error)
    - get_tags(int card_id)
      - return list of tags (Strings) for that card
    - remove_tag(int card_id, String tag_name)
      - return int (1 for success, =<0 error)
  - cards
    - create_card(String company_name, String job_title, int status)
      - return int (1 for success, =<0 error)
    - modify_card_status(int card_id, int updated_status - 0 is “considering”, 1 is “in progress”, 2 is “completed”)
      - return int (1 for success, =<0 error)
    - get_user_cards()
      - user_id = request.POST[user]
        - User.objects().all().filter(id= 1)
        - Cards.objects().all().filter(owned=User)
      - return JSON defined in API
- **View (template)**
  - my-cards.html extends dashboard.html extends base.html
    - {{list of card objects}}
    - {{tags given card}}

UserApp

- **Model (models.py)**
  - User.objects().all().filter(id= 1)

        - Returns a User object (row of User)
    - User.objects().all().filter(id=1)
        - Returns a User object (row of User)
- **Controller (views.py)**
    - sign_up(String username, String password, String password_confirm, String email)
        - return int (1 for success, =<0 error)
    - login(String username, String password)
        - return int (1 for success, =<0 error)
    - logout()
        - return int (1 for success, =<0 error)
- **View (template)**
    - sign-up.html extends message.html extends base.html
    - sign-in.html extends message.html extends base.html

DocumentApp

- **Model (models.py)**
    - Documents.objects().all().filter(owned=User)
        - Returns all Documents owned by user (can be many rows)
- **Controller (views.py)**
    - upload_document(PDF document)
        - return int (1 for success, =<0 error)
    - remove_document(String document_id)
        - return int (1 for success, =<0 error)
    - get_documents(String user_id)
        - return list of Document objects
- **View (template)**
    - profile.html extends base.html

Implementation Plan

Iteration 1

When we load the dashboard, the cookies will be checked for existence of a signed-in user. If they are signed in, we take them to their dashboard; otherwise, they are redirected to the Register page. On the dashboard, we would need to query our server for the items to be displayed. When the user wants to add a card, we query for the list of companies based on his filter. When the user wants to edit or view details of a card, the server will query for that card as well.

Note: For Iteration 1, we are no longer implementing the "Save company to 'interested'" and "Click 'Bookmark Company'" user stories, even though we indicated that we would in the Requirements and Specification. We decided these two user stories were not crucial enough for the basic functionality of our app. We also decided that it would be more time

efficient to implement “filter” in the second iteration rather than the first.

- The previously mentioned databases must be created in order for many of the user stories and functions defined below to work and be tested. Many of the functions are dependent on these databases and other functions, as mentioned below. For this reason, the teams will be split up so that we have **three teams** each working on one of the “apps” defined above:
    - Team 1 (**Yaxin and Bhavini**): The UserApp
    - Team 2 (**Seth and Alex**): The CardApp (This teams seems to have more work than the other two. This work may be redistributed to solve this.)
    - Team 3 (**Paulina and Paul**): the DocumentApp
- There will be two people per team. Person A will peer-review Person B’s code and vice versa in each team.

Name	sign up
Developer Days	1
Team Member(s)	Team 1
Dependencies	Dependent on a User database - creates user in User database
Actors	any user
Triggers	visit page
Preconditions	no cookie
Actions	open registration page, save user to database and redirect to dashboard
Postconditions	user is taken to the dashboard
Acceptance test	user name and password validation, make sure the user is in the database

Name	sign in
Developer Days	1
Team Member(s)	Team 1
Dependencies	Dependent on User database and the sign_up function - database must be set up and user must be in database
Actors	user with an existing username and password
Triggers	visit page

Preconditions	has cookie or existing login of cookies disabled
Actions	opens registration page for that specific user
Postconditions	user is taken to the dashboard
Acceptance test	user name and password validation, make sure the user is in the database

Name	log out
Developer Days	1
Team Member(s)	Team 1
Dependencies	Dependent on User database and the sing_in function - User must be created and be active in database
Actors	logged in user
Triggers	select “logout” button
Preconditions	already logged in
Actions	takes user back to sign in page, saves all information to the user’s profile
Postconditions	display the login page
Acceptance test	ensure data is saved to user’s profile, not still logged in upon refresh

Name	documents (resume/cover letter) upload
Developer Days	1
Team Member(s)	Team 3
Dependencies	Dependent on the sign_in function, and the Documents database - a user must be in session and the document must be added to the Documents database
Actors	logged in user
Triggers	click upload
Preconditions	must be signed in

Actions	save document to database and updates the documents view
Postconditions	documents page updated with new documents
Acceptance test	documents saved in database and the documents page updated

Name	view dashboard
Developer Days	1
Team Member(s)	Team 1
Dependencies	Dependent on the sign_in function - User must be active and in the user database; sends user to their dashboard
Actors	logged in user
Triggers	click dashboard
Preconditions	must be signed in
Actions	show all the user's cards (i.e. companies applied to)
Postconditions	user is taken to the dashboard page
Acceptance test	make sure dashboard page is shown, make sure all of user's corresponding cards from databases loaded into the view

Name	remove documents
Developer Days	1
Team Member(s)	Team 3
Dependencies	Dependent on the sign_in function and the document upload - a logged in user must have uploaded a document (must be in the documents database) for the document to be removed
Actors	logged in user
Triggers	click "remove" button
Preconditions	must be signed in, must be on the documents page, must have documents to remove
Actions	remove specific document from database and refresh the page

Postconditions	user is taken to the documents page
Acceptance test	make sure that the appropriate document(s) are deleted once the user gets taken back to the documents page, make sure deleted document(s) corresponding to appropriate user are removed from the database

Name	Create card for each company you're interested in
Developer Days	1
Team Member(s)	Team 2
Dependencies	Dependent on the sign_in function, the Cards database and the User_card join table must exist - a valid logged in user creates a card by adding it to the two databases
Actors	Any registered user
Triggers	Click create card for company
Preconditions	Company exists in your library of companies
Actions	Reads in information and saves it to specific fields of card. Checks for validity of these fields. Saves it to database, update associated count.
Postconditions	Card saved in database and view in Dashboard updated to reflect that.
Acceptance test	Check for existence of card. Check for update of card count. Check that card doesn't already exist (or a card too similar).

Name	Add tags
Developer Days	1
Team Member(s)	Team 2
Dependencies	Dependent on the sign_in function and the create_card function - A valid user must be logged in and he card must be in the Cards database
Actors	Any registered user
Triggers	Click "add tags" for a card
Preconditions	Card exists and doesn't have max number of allowed tags (20)

Actions	Tag associated with card and database updated to reflect association. Update view immediately with a tag added to card
Postconditions	New tag associated with card in database such that filtering will catch it.

Name	remove tags
Developer Days	1
Team Member(s)	Team 2
Dependencies	Dependent on the sign_in function and the create_card function and add_tag function - A valid user must be logged in and he card must be in the Cards database and the removed tag must exist in the database
Actors	Any registered user
Triggers	Click “remove tags” for a card
Preconditions	Card exists and tag exists
Actions	Tag associated with card and database removed from database
Postconditions	Tag associated with the card is removed

Name	modify tags
Developer Days	1
Team Member(s)	Team 2
Dependencies	Dependent on the sign_in function and the create_card function and add_tag function - A valid user must be logged in and he card must be in the Cards database and the modified tag must exist in the database
Actors	Any registered user
Triggers	Click “modify tags” for a card
Preconditions	Card exists and tag exists
Actions	Tag associated with card and database changed
Postconditions	Tag associated with the card is changed in the database

Name	Card view
Developer Days	1
Team Member(s)	Team 2
Dependencies	Dependent on the sign_in function and the create_card function - A user in the user database must be logged in and the card must exist in the Card database
Actors	Logged in user
Triggers	User clicks on a specific job card on the Dashboard
Preconditions	User already created a card
Actions	Takes the user from a view of multiple cards on the dashboard to a detailed view of the specific card
Postconditions	User's view is centered on the selected card, conditions/tabs of the card are clearly displayed. Back is button visible
Acceptance test	Opening card view doesn't crash the dashboard, make sure selected card is the one that comes into the larger view

Name	Modified application stage to card (modify_card_status)
Developer Days	1
Team Member(s)	Team 2
Dependencies	Dependent on the sign_in function and the create_card function and the add_application_stage function - A user in the user database must be logged in and the card must exist in the Card database with the stage of the application for it to be modified
Actors	Logged in user
Triggers	User clicks on the "application stage" section on the card view
Preconditions	Card is already created, user has selected a specific card's view, a status already exists
Actions	Selection menu appears to allow the user to choose between three stages: 0. considering, 1. in progress, 2. completed. The selection is then saved
Postconditions	User's selection becomes saved and displayed on the card; the tag on the card (to be used for filtering) will update to reflect that stage change

Acceptance test	Make sure user selection corresponds with what is displayed afterwards, make sure it appears when filtering for its corresponding category

- The testing for each unit test is dependent on the method it is testing. Unit Tests for the different apps can be rotated by the teams:
    - Team 1: DocumentApp unit tests
    - Team 2: UserApp unit tests
    - Team 3: CardApp unit tests

Project Risks

- Can't finish features in time within a particular iteration
    - Mitigation: prioritization (focus on essential features, then add on), weekly reality checks
- Security risks - user accounts may be compromised, upload documents feature may be exploited
    - Mitigations: hash passwords in database, built-in input sanitization provided by Django
- Teammate might become malicious or team might not work cohesively together
    - Mitigation: weekly meetings, get to know each other on a personal level, consult TA if situations escalate
- Mess up git (merging)
    - Mitigations: read git documentation thoroughly, commit and push often, defer to expertise of teammate with more git experience. ALWAYS USE GIT REBASE INSTEAD.

Testing Plan

Our units tests will use Python's built-in unit testing framework. The tests will be as follows:

CardApp Unit Test

- `testCreate_card()` - test that the company_name and job_title must be between 1 and 128 characters and return appropriate error code
    - create a card with company_name and job_title that are both between 1 and 128 characters and status = 0, 1, or 2. make sure it returns 1
    - create a card with a company_name that is 129 characters and make sure it returns =<0
    - create a card with empty company_name. make sure it returns =<0
    - create a card with None as company_name. make sure it returns =<0
    - create a card with a job_title that is 129 characters. make sure it returns =<0
    - create a card with empty job_title. make sure it returns =<0
    - create a card with None as job_title. make sure it returns =<0
    - create a card with status = 3. make sure it returns =<0
    - create a card with status = 0, 1, or 2. make sure it returns 1
- `testModify_card_status()` - test that status $\in$ {0, 1, 2}

    - modify card status to be 0, 1, or 2. make sure it returns 1
    - modify card status to be 3. make sure it returns =<0
- testGet_user_cards() - test that it returns the correct JSON of the user's cards
- testAdd_tag() - test that tag_name has length 1-128 characters and card_id is valid
    - create a tag with tag_name that is between 1 and 128 characters. make sure it returns 1
    - create a tag with tag_name that is 129 characters. make sure it returns =<0
    - create a tag with empty tag_name. make sure it returns =<0
    - create a tag with None as tag_name. make sure it returns =<0
    - create a tag with empty card_id. make sure it returns =<0
    - create a tag with None as card_id. make sure it returns =<0
- testModify_tag() - test that modify_tag return the correct error code for different cases
    - pass in an existing card_id and existing current_tag_name. make sure the tag_name is updated in database and it returns 1
    - pass in a new_tag_name that is 129 characters. make sure it returns =<0
    - pass in a card_id that is 129 characters. make sure it returns =<0
    - pass in a current_tag_name that is 129 characters. make sure it returns =<0
    - pass in a card_id that does not exist. make sure it returns =<0
    - pass in a current_tag_name that does not exist. make sure it returns =<0
    - pass in an existing new_tag_name. make sure it returns =<0
- testGet_tags() - test that card_id exists in order to return tag list
    - check that it doesn't return a list of tags associated with an invalid card_id
    - check that it returns a list of tags associated with a valid card_id
- testRemove_tag() - test the card_id or tag_name we want to remove exist
    - remove the tag of an invalid card_id. make sure it returns =<0
    - remove the tag_name that does not exist. make sure it returns =<0
    - remove an existing tag_name of a valid card_id and make sure it returns 1

UserApp Unit Test

- testSign_up() - test that user input information is correct
    - sign up with a username and a password, both between 1 and 128 characters, a password_confirm matching with password, and a valid email address(xxxx@xxxx.xx). make sure it returns 1
    - sign up with empty username. make sure it returns =<0
    - sign up with None username. make sure it returns =<0
    - sign up with a username that is 129 characters. make sure it returns =<0
    - sign up with empty password. make sure it returns =<0
    - sign up with None password. make sure it returns =<0
    - sign up with a password that is 129 characters. make sure it returns =<0
    - sign up with a password_confirm that does not match the password. make sure it returns =<0

    - sign up with an invalid email address format (i.e. Doesn’t have ‘@’ and ‘.’). make sure it returns =<0
    - sign up with empty email address. make sure it returns =<0
    - sign up with None email address. make sure it returns =<0
- testLogin() - test that username and password are valid
    - login in with a username and a password that are both between 1 and 128 characters. make sure it returns 1
    - login with a username that does not exist. make sure it returns =<0
    - login with an existing username that does not match the password in database. make sure it returns =<0
    - login with empty username. make sure it returns =<0
    - login with empty password. make sure it returns =<0
    - login with None username. make sure it returns =<0
    - login with None password. make sure it returns =<0
    - login with a username that is over 128 characters. make sure it returns =<0
    - login with a password that is over 128 characters. make sure it returns =<0
- testLogout() - test that user has successfully logged out (session ended)

DocumentApp Unit Test

- testUpload_document() - test that valid document is uploaded
    - upload a new document and check that document is added to database. make sure it returns 1
    - upload a document that is not in PDF format. make sure it returns =<0
    - upload a document that already exists in database. make sure it returns =<0
- testRemove_document() - test that documents are being removed correctly
    - remove a document with existing document_id and check that the document is not in the database any more. make sure it returns 1
    - remove a document with document_id that does not exist. make sure it returns =<0
    - pass in a document_id that is not between 1 and 128 characters. make sure it returns =<0
- testGet_documents() - test that it returns list of document if user exists
    - pass in an existing user_id. make sure it returns the list of document objects belonging to user_id
    - pass in a user_id that does not exist. make sure it does not return any list
    - pass in a user_id that is over 128 characters. make sure it does not return any list