6.170 Software Studio:
Fall 2014
AmazeJobs: Problem Analysis

Catherine Liu, Elliott Marquez, Olga Shestopalova

Description:

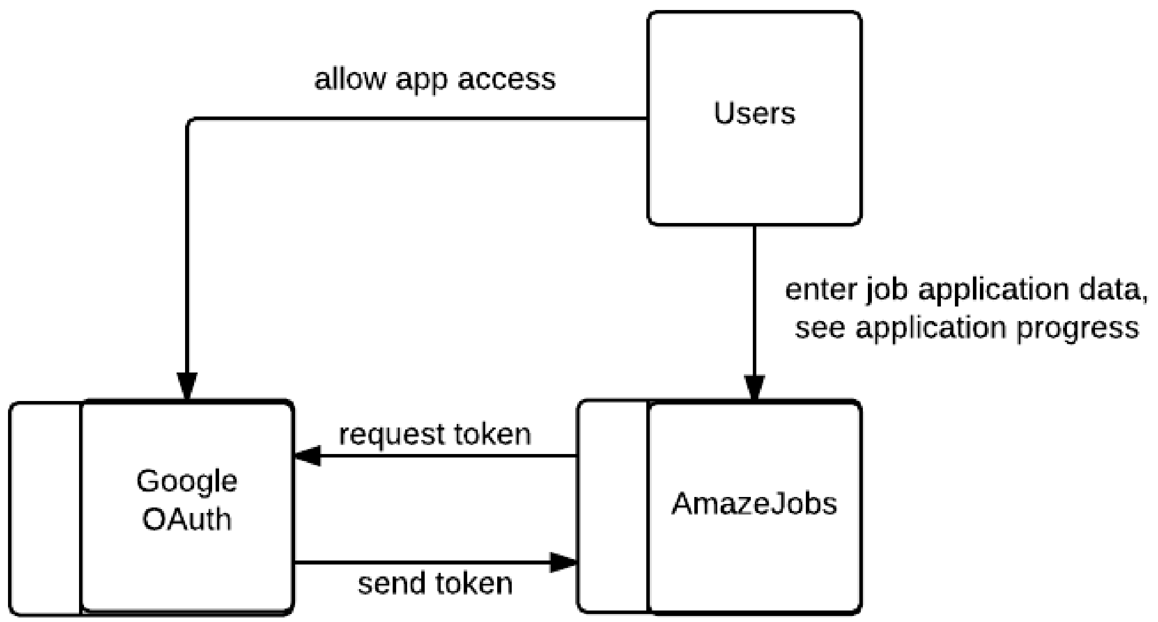
AmazeJobs keeps track of users' job applications: their progress, deadlines, and tasks associated with each application. It strives to alleviate the users' burden of remembering all the jobs they applied to, which of them responded, how long ago, and tasks with deadlines that must not be forgotten. When the user inputs the data from the different applications, AmazeJobs will aggregate it, and display it in an organized and user friendly interface.

A key user group we are targeting are students around career fair season, where each student will speak to many companies, but frequently will forget to apply on time, or follow-up. AmazeJobs has phases for each job application: Applying, Interviewing, Offered, Terminated; only one can be active at a time, and the completion of one will give rise to another (unless it is a terminal phase such as Offered or Terminated). Terminated can be entered after any of the other phases. Tasks are tied to a specific phase.

Purpose:

- **Help keep track of the status of many job applications.** AmazeJobs will gather the status of each job application in a single, easy-to-use interface to make it easier to determine which applications need further work, and what stage of applying you are in with each application. If an application needs further work, a user can assign a task to it.
- **Keep track of and remind the user of important events and deadlines.** For example, we would like to help the user schedule interview times/dates and remember when applications have due dates.
- **Have an aggregate task list of things to do.** AmazeJobs will gather all the assigned tasks from multiple job applications and organize them so that a user can view them all together and see what has been done and what needs to be completed.

Context Diagram:

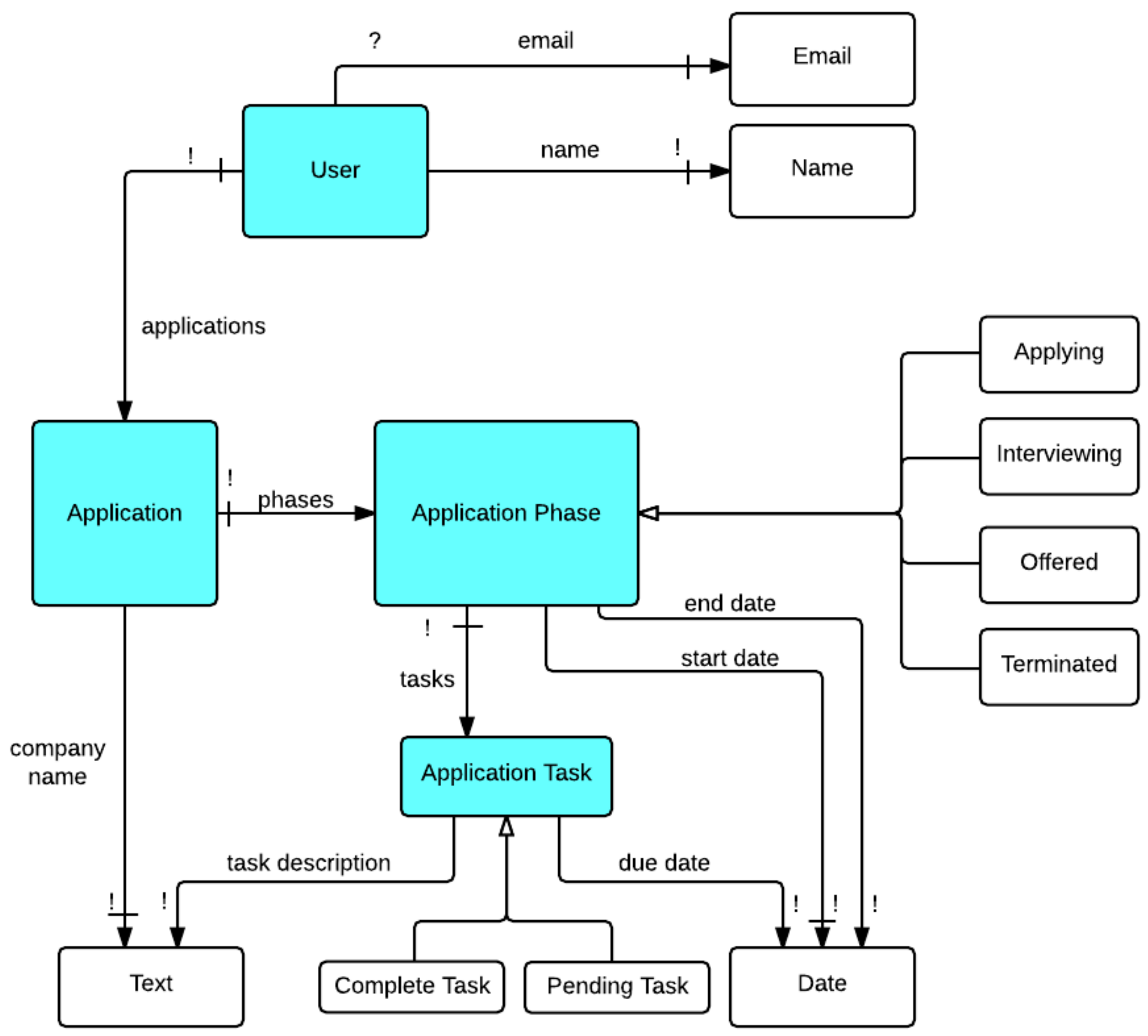


- Sign In/Authentication with Google
    - User approves app login through Google, login completes
    - Potential problem: user doesn't have Google account

Concept:

- Application: Motivated by first purpose. An application to a job - must have a company name associated with it. Each application is in one of four phases (Applying, Interviewing, Offered, Terminated)
- Application Phase: Motivated by first purpose. A stage in the application process. Must have a start date, can have an end date, and tasks associated with it. It doesn't necessarily need an end date if it is the current phase. Only needs an end date when you finish a phase.
    - Applying – When you first begin an application to a company, either via an online form or resume drop or any other method.
    - Interviewing – When a company offers you an interview, your phase of Applying ends and Interviewing begins
    - Offered – When the company gives you a job offer
    - Terminated – Can occur after any stage, either via voluntary withdrawal or via company rejection. Has start date = end date.
- Task: Motivated by second and third purposes. Has description and optional due date and is associated with an application phase. There are two types: Pending Task and Complete Task.
    - For example a task can be: "go to Google info session" or "interview with Facebook"

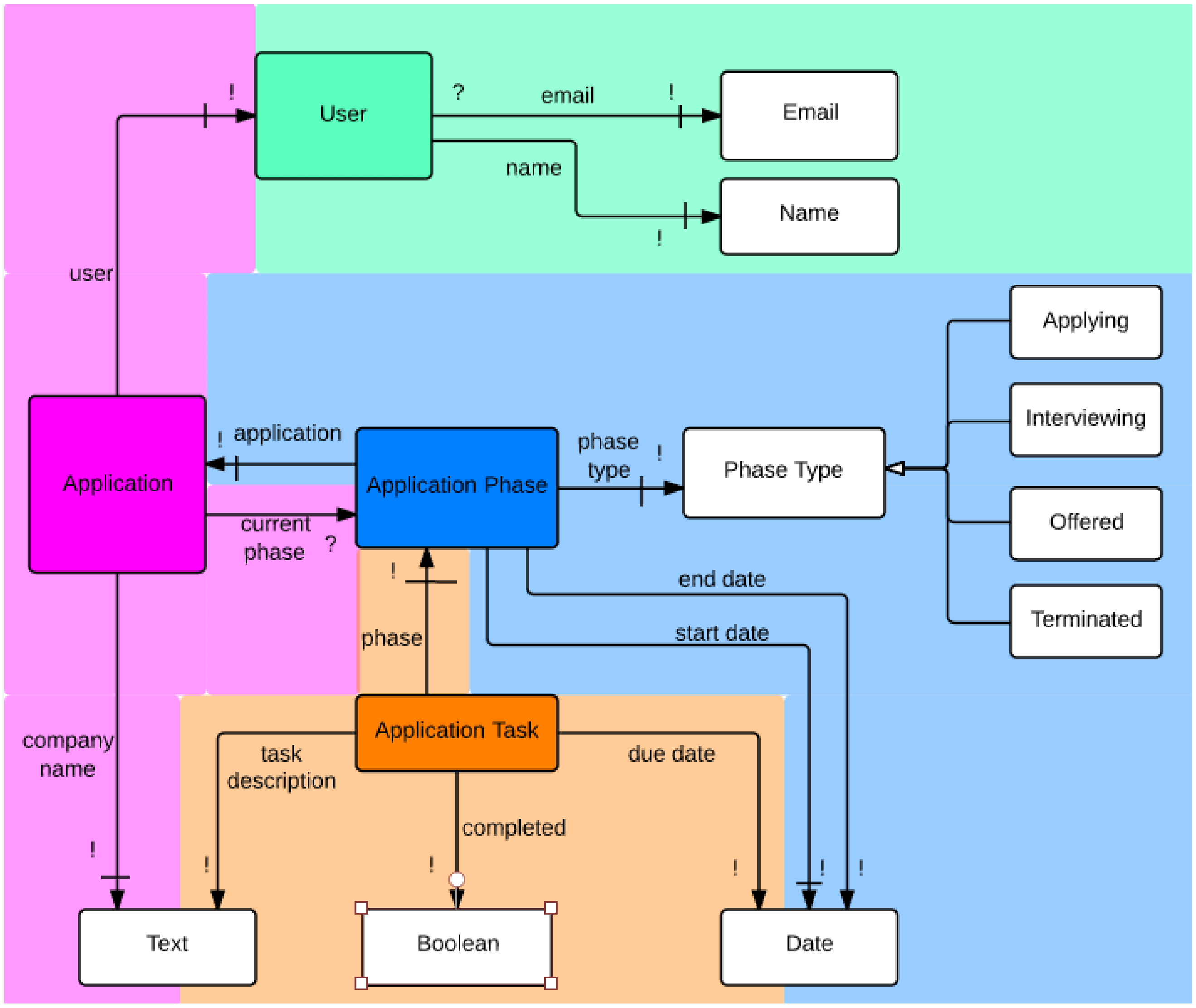
Data model:



- **User**: represents a user of AmazeJobs
  - name: a non-empty string
  - email: email used to authenticate users
  - applications: a list of current and old Applications
- **Application**: represents a single job application made by a user
  - company name: a non-empty string
  - phases: a list of current and past (not future) phases of the job application
- **Application Phase**: represents a phase in the application (currently there are four: Applying, Interviewing, Offered, Terminated), only one can be active at a time, and the completion of one will give rise to another (unless it is a terminal phase such as Offered or Terminated. Terminated can be started after any of the other phases.

    - start date: date user started the phase, immutable
    - end date: date user ended the phase, used for history purposes
    - tasks: list of Application Tasks associated with the phase
- **Application Task**: represents a task that must be completed (there are two kinds: pending task and complete task)
    - description: non-empty text description of the task
    - due date: date the task should be completed by (optional)
    - complete: whether or not the task is complete (false by default)

Data Design:



- **Reversal of relations** between user/application, application/phase, phase/task. We changed the larger item having a list of smaller items to the smaller item having a reference to its larger item because this removed redundancy and confusion when smaller items would get deleted (previously we would have had to find its owner and update that every time).

- **Phase type** as an enum. To make our system more easily extendable, we decided to create a field in phase that would determine its type. Also, all phases have the same information and behavior, so making entirely new schemas for each one seemed silly.
- **Task's completed field** is a boolean, because tasks should be easily converted from pending to completed and both types have the same information and behavior. Also, no enum even had to be created because Mongoose supports (and validates) booleans.
- **Application's current phase field** holds a reference to the current phase, if there is one. This was added to make keeping track of and progressing through phases.

Design Challenges:

From Part 2:

- How should progressing through phases work
  - Options:
    - Have a PUT modify the old phase (if any), then do a POST to create a new one
    - Have one PUT that will do both of these in a transaction
  - Decision:
    - We went for the latter because having a POST to create a new phase could potentially be abused and break the rep invariant and creating a mechanism to validate the POST would be really complicated for very little gain. This would essentially be re-creating a transaction, so we just went for the latter.
- Individual vs batch GETs:
  - Options:
    - Do we want to have individual GETs of applications, phases, and tasks?
    - Or leave it as seeing a list of applications, phases, or tasks?
  - Decision:
    - Get everything in bulk, as we have no use for individual objects. The UI will require all of the applications, all of the phases, and all of the tasks (associated with one user) when displaying the data. We do batch GETs, but individual POST/PUT/DELETE.
- How to keep track of the current phase
  - Options:
    - Each phase has an isCurrent marker
    - Each application holds a currentPhase reference
  - Decision:
    - We opted for the latter because only the application should care about what the current phase is, not the phases. Also, it would be easier to look up the current phase like this.
- Whether or not to keep our reorganizable task list idea
  - Options:
    - Add more things to our model (eg. task rank) to create this task list
    - Make code better elsewhere, work on this later

- Decision:
  - We probably won't have enough time to augment the data model for task to include something to help with ordering. Learning how to use Google's API is more crucial and we would rather finish this part than not have enough time, so we will not include the reordering of the task list for this part.

- Whether or not to keep Google Calendar
  - Options:
    - Keep trying to get Google Calendar to work
    - Make code better elsewhere, work on this later
  - Decision:
    - One of our team members spent an enormous amount of time trying to get Google Calendar integration to work, but finally we decided that we need help elsewhere and should not spend all our time trying to get this to work.

From part 1:

- We were unsure of how to represent progress in each job application.
  - Options:
    - We could have some string indicator of the current phase
    - We could have defined phase objects
  - Decision:
    - We opted for the latter because each job application went through roughly the same phases and this was more elegant than a string encoding. We also made it easy to add another phase, creating a generic phase set that contained specific phases. Also, having predefined phases makes it easier to display progress consistently across different job applications.
- We were unsure of how to visually represent tasks and events.
  - Options:
    - We could have a list of dates
    - We could have a calendar
  - Decision:
    - We decided on both: we will have a task list as well as a calendar, to aid in planning (user can see their other activities) and seeing things in perspective (how close together or far apart their deadlines are).
- We were unsure of how to keep track of users.
  - Options:
    - We could implement some username/password design
    - We can have users login with an outside service (Google, Facebook, LinkedIn, etc).
    - We can have a certificate-based authentication system
  - Decision:
    - We opted for the using an outside service (Google) primarily because we planned to integrate with Google Calendar, and for that we would need the

user to allow us access to their Google account anyhow. Also, this kind of login is easier for the user and doesn't require remembering usernames and passwords. This also allows us the possibility to integrate with other Google Apps if we deem it necessary. Additionally, we did not want a certificate-based authentication system because certificates work differently across operating systems.

- We were unsure where tasks belonged in our design.
    - Options:
        - We could have associated them with the job application, have them be their own entity tied to each user.
        - We can have them be associated with a particular phase.
    - Decision:
        - We chose the last option because tasks should be tied to a specific phase, and would not stretch across phases, and certainly not across applications, so for reduction of scope, we placed tasks under their respective phase.
- How to integrate a calendar with an application
    - Options:
        - Use Google accounts and Google Calendar
        - Create our own authorization and calendar system
    - Decision:
        - If we used Google accounts and somebody didn't have a Google account, they would be unable to use our service. But creating our own authorization and calendar system just to provide service to the smaller fraction of potential users without Google accounts seemed excessive, so we will be using Google's APIs and services to implement calendars
- Whether or not to allow users to modify the calendar outside of the application
    - Options:
        - Allow the user to modify outside and detect event changes when the app is loaded, and modify db as needed.
        - Create events with a service account that are locked and then shared with the user to prevent modification outside of the application
    - Decision:
        - We decided it would be much easier to figure out how to use service accounts and create locked events than try to detect changes to events and update the database to change it.
- Whether to associate applications with a company or companies with applications
    - Options:
        - Associate applications with companies
        - Associate companies with applications
    - Decision:
        - Associate companies for applications because we do not think that it will be so common to apply to a company for multiple positions, and if a user

would want to look up old applications for a company, this data model will not cause too much of a backend mess to look up.

<u>**API**</u>

POST /login
Request Body: *empty*
Response:

- error: error if there was one

Logs in the user via Google OAuth2.0 by redirecting to Google. Once Google has received login/permssions, redirects to /oauthcallback.

POST /logout
Request Body: *empty*
Response:

- error: error if there was one

Logs out the user, destroys cookies

GET /oauthcallback
Request Body:

- authorization code as a query from Google

Response:

- error: error if there was one

Gets tokens from using the authorization code and saves them as the client tokens. Adds user to database if they don't exist, sets cookies.

GET /user/{id}/applications
Request Body: *empty*
Response:

- applications: list of Applications
- error: error if there was one

Gets all applications associated with a particular user

POST /user/{id}/applications
Request Body:

- company name

Response:

- applicationId: new application id
- error: error if there was one

Creates a new application for a specified user

POST /application/{id}/phases
Request Body:

- whether the application has been terminated or not (a boolean)

Response:

- phaseId: new phase id if a new one has been created
- error: error if there was one

Ends the current phase, creates a new phase if the ended phase was not a terminal phase and the application was not terminated by the user

DELETE /application/{id}
Request Body: *empty*
Response:
- error: error if there was one

Deletes the specified application and its associated phases and tasks

GET /application/{id}/phases
Request Body: *empty*
Response:
- phases: list of Phases
- error: error if there was one

Gets all phases for a specific application

DELETE /phase/{id}
Request Body:
- phase id

Response:
- error: error if there was one

Deletes the specified phase and its tasks

GET /phase/{id}/tasks
Request Body: *empty*
Response:
- tasks: list of Tasks
- error: error if there was one

Gets all the tasks associated with a specific phase

POST /phase/{id}/tasks
Request Body:
- description
- due date (optional)

Response:
- taskId: new task id
- error: error if there was one

Creates a new task for the specified phase. The event starts out not completed.

PUT /task/{id}
Request Body:

- description (optional)
- due date (optional)
- completed (optional)

Response:

- error: error if there was one

Modifies the task fields

DELETE /task/{id}
Request Body: *empty*
Response:

- error: error if there was one

Deletes the specified task