Nizar Maan

July 13, 2018

**Workday Talent Marketing Rationale and Analysis**

**Questions**

Analysis

The core concept of the Talent Marketing system is the Opportunity Market. This is where users login to look for potential opportunities (Projects) to work on and/or post opportunities asking for help. For some of our pilots, user engagement is starting to drop - users log in, browse some opportunities and don’t come back.

**How would you address this problem?**

I would try to implement functionality like an “abandoned online shopping cart”, where if a user browsed an opportunity without applying or without any sign of site-interactivity, the system would send out an email encouraging/reminding the user about the opportunity they navigated away from. This would require certain measures to avoid spamming users with emails.

Also, perhaps content is not personalized enough, maybe users are seeing things which they are not actually interested in or that don’t reflect their activity on the platform. Another possibility is to have an online forum/chat where users can actively chat in real time.

Additionally, after users completed projects, they could be asked to make a small blog post about their experience and why it was worthwhile. This can allow the platform to have a user-driven review system for projects of certain skill requirements, instead of just a one-way project owner reviewing users case.

**What are some solutions you would propose to encourage users to post and join opportunities?**

Providing an incentive for encouraging users to post and join opportunities can come in many forms. There can be a reputation or point system like in StackOverflow, or achievement levels or points earned per post or application to an opportunity. Something for people to obtain bragging rights on a friendly level. One could also start to provide relevant links to courses or tutorials for skills that a user has in their “interested” list.

**The Project Assembly**

The classes were written on Visual Studio 2017 as a C# .Net Core MVC Web-app Project. As such, much of the design is predicated on a “web-based” assumption.

The project can be cloned from its repository at: <https://github.com/NizarMaan/TalentMarketing>

I used a Model-View-Controller design along with a “Services” namespace to hold business logic. I think all required data-models/classes are there to be exposed by whatever front-end logic and routing/controller set-up. The data/structure is available, just a matter of however it is decided it is to be retrieved – So I did not implement any sort of working routing or views.

**The relevant code/classes that I wrote are contained in the Models, Controllers, and Services folders.**

There is a lack of comments in the code and perhaps a lack of rationale in this document, due to time constraints. Also, no classes/implementation for User project recommendations was written due time constraints, although it would have worked similarly to the skills-match analyzer.

Assumptions

* No database connection so did not write any database connection or table logic with any ORM library - purely object oriented approach. Had I had designed with a database in mind I would have used the ORM known as Entity Framework for a code-first approach to creating SQL-server tables
* Duplicate skills are unwanted, also it is NOT case-sensitive, e.g. a skill of “C#” is considered the same as “c#”
* On skill addition for a user there is an API Post route for use as an AJAX call every time a skill is added via UI
* Assuming validation in front-end (as a .net web app, we can use javascript validation library) but i also perform checks in back end to mitigate any possible front-end bypasses or multi-thread/async issues
* Project Owner is counted toward in team member count
* No duplicate emails allowed, however, duplicate names etc. allowed upon user creation/editing
* Duplicate project names allowed
* During Project creation process a project owner submits two separate web-forms that list “required skills” and “skills to gain” that, in ASP.NET, get submitted as List<string> objects to the back-end POST route.
* Desired and learned skills can be removed/added at any point
* If a project team is full (e.g. not recruiting) one cannot apply to it as the project’s status will be set to “filled”, this relies on the assumption that the front-end will reflect the projects status and not display any application functionality
* Projects can change state between recruiting, filled, completed, at any point as people can always shift around or a project could re-open
* Team capacity can be changed at any point, this leads to having to check whether there are now too few team members (have to set project status to “recruiting”). If the new capacity is lower than current team members, capacity stays the same, in this case team members would have to be removed before any capacity editing
* There can be a limit on how many required skills/skills to learn a project can have. The limit of 30 that I use is completely arbitrary.
* There is no “dictionary”/database of skills. Skills are entered in “Free-form” i.e. not constrained to some predetermined list of values
* The Analyzer class is meant for use when a user clicks a “My Owned Projects” view and the user happens to be that project’s project owner. When that view is rendered, the user will see a unique view for each project for which he is the owner, where he can see all the applicants and the skills which match up to the project along with other applicant metadata
  + However, as I have it now, it would be required to perform analysis/matching every time a user opens their “Owned Projects” window or whenever user browses projects. This can probably be improved by setting the analysis/matching engine to compute and store matches for users in the background every however much time.
* Also, since we’re not working with a database here, I set object id’s manually. Normally, with an ORM tool id’s are auto-incremented.
* For reviews, I decided for a max rating of 10, i.e. a rating out of 10.

With more time, I probably would have attempted the following:

* ProjectManager, ApplicationManager, and UserManager service classes could have inherited from a “Manager” interface, as a lot of logic is similar
* More constructors with different signatures could have been added for our available models
* Since project, user, and application lists in DataContext have sequential id’s, we could probably have used a binary search whenever a look up by id was required
* Write out the front-end controllers
* Session/Cookie manager to keep track of currently logged-in user. With this we can display user’s current projects, applications, as well as owned projects, etc.
* User password encryption
* Password recovery/change mechanism such as email with URL unique token
* API key so it is not exposed
* Rather than having enums of statuses for Application and Project objects, create a status class
* Implement mechanism to force Project Owners to give endorsements and reviews upon project completion. Perhaps a “Task” class queue data structure for users which, if it has any “high priority” tasks in it, will display a “non-escapeable” interface until the high priority tasks have been complete. The order of the endorsement/review tasks that an owner has to perform for each team member will be based on whatever first-to-last order the list of team members is in
* No other implementation logic was applied for Reviews, only a Review model was provided
* For recommendations I would have assumed that for a recommendation to occur, we want at least a 50% match in skills to required project skills
* Would have tried to devise a more robust API so that data is accessible on different platforms, e.g. different websites, desktop applications, mobile app, etc.
* Analyzer matches skills in O(n \* m \* w \* u) time. This can probably be improved
* Would have also made use of database relational tables to reduce duplication of data across objects/classes