EvaluRate

The Hack Jacks

Dylan Grayson
Conner Swann
Brandon Paree
Brian Saganey

Design Document

March 25, 2016
V1.1

Table of Contents

1. Introduction

In today's workplace, it's almost impossible to avoid working with a group of people on a project. In nearly every organization from the university to corporate America, teams of people are organized to get large tasks done. For the most part, the inner workings of how teams operate have been optimized by a multitude of modern software applications (ex. Skype for communication, Trello for task management, etc.). However, there is one area that's remained virtually untouched, and that is the area of performance evaluation.

For the most part, teams are a black box, a project goes in and the result comes out. The only way of determining how well a team is performing, if you are not part of the team, is to compare the results to the original requirements and see how well they did. This is not optimal, as a weak team member can make or break a project, potentially wasting the time of the entire team if the end-result doesn't live up. In order to prevent this from happening, managers require more granular control over how performance is reported and team interactions are monitored. However, if they aren't physically next to their subordinates all day they aren't capable of seeing what goes on between team members on a regular basis. Additionally, when directly asked how one feels about a project, the common inclination is to not speak up about issues in order not to anger management or "rock the boat."

Over the years, industry and academia have attempted to solve this problem through the use of regularly issued anonymous peer evaluations. Peer evaluations are a tool often used as an early indicator of a project's success and allow a manager to intervene if it appears that there is an underperforming team member. Additionally, the anonymous nature of the evaluations encourage members to respond honestly to questions about the project. More often than not, these evaluations are completed on paper or via email, requiring the issuer to spend an inordinate amount of time consolidating all the data and analyzing it so they can make informed decisions. There are several applications that currently exist in an attempt to solve this problem, however there are some issues with the current offerings. All existing solutions rigidly focus on academia and are designed explicitly for the classroom, ignoring the private sector. Additionally, they are not the easiest applications to use and put form *far* behind function.

Ideally, there would be a software application that would provide structure to the evaluation process. This application would:

- take the time out of creating, issuing, and tabulating the results from peer evaluations.
- make it easy to take the data from evaluations and make meaningful inferences about the team from that data.
- be a central location to visualize the data from these evaluations over time.
- be not only functional, but visually pleasing and nice to use.

EvaluRate is a scalable and customizable web application created for the purpose of issuing and managing peer evaluations -- eliminating the need for combing through and manually computing results from surveys. Within EvaluRate, managers are able to model the structure of their organization by separating users into a hierarchy of divisions. From there, projects are created to model real world projects in the system, and teams are assigned to the project. Managers are then able to pick from a series of predefined evaluation templates, and schedule them to be released at a time when it is meaningful to collect feedback from team members.

Members of these teams are informed that they have an outstanding evaluation to complete by a variety of methods such as email, text message, or in-app notification. The link that is sent via one of these methods can be followed to take the user directly to the evaluation, quickly fill it out, and get back to what they were doing.

At this time the managers are able to see the completion progress of a given evaluation, and once completed, generate meaningful data that can be used in a variety of ways. In addition to providing in-app analytical ability, EvaluRate also provides the ability to export raw evaluation data in a variety of formats for use in other applications or reports.

2. Architecture Overview

Since EvaluRate is web application, the underlying architecture for the application is handled mostly with Meteor JavaScript web framework. This framework provides all the code necessary to create a responsive and modern web application. Meteor accomplishes this by running a Node JavaScript server that interfaces with a MongoDB database, and serves a robust client-side JavaScript application. This client-side application communicates with the server over HTTP and a custom protocol known as DDP.

2.1. Distributed Data Protocol (DDP)

The Distributed Data Protocol uses websockets to maintain a persistent connection between client and server. This means that new data being added to the database will be automatically updated on the client without the client needing to poll for updates. DDP facilitates constant updates so the clients will always have up-to-date data at any given moment.

2.2. Case for MongoDB

EvaluRate leverages MongoDB to enable the application to store a variety of custom user-generated data, primarily the evaluations themselves. In addition, the internal implementation and database objects have been carefully considered in order to provide for a large number of client configurations. For this, MongoDB is incredibly useful due to its general flexibility in how data can be stored within it.

2.3. Architecture Diagram

The diagram in *Figure 2.1* shows the modified client-server architecture of the application as it exists with Meteor. The ‘App Microservices’ section refers to all the code being written for EvaluRate on the server, while all other components on the server are defined by Meteor. Similarly, the ‘App Components and Logic’ section on the client side refers to all of the business logic for EvaluRate, and the ‘Handlebars, Blaze, or React’ section refers to the graphical user interface for EvaluRate. The other components on the client are also defined by Meteor.

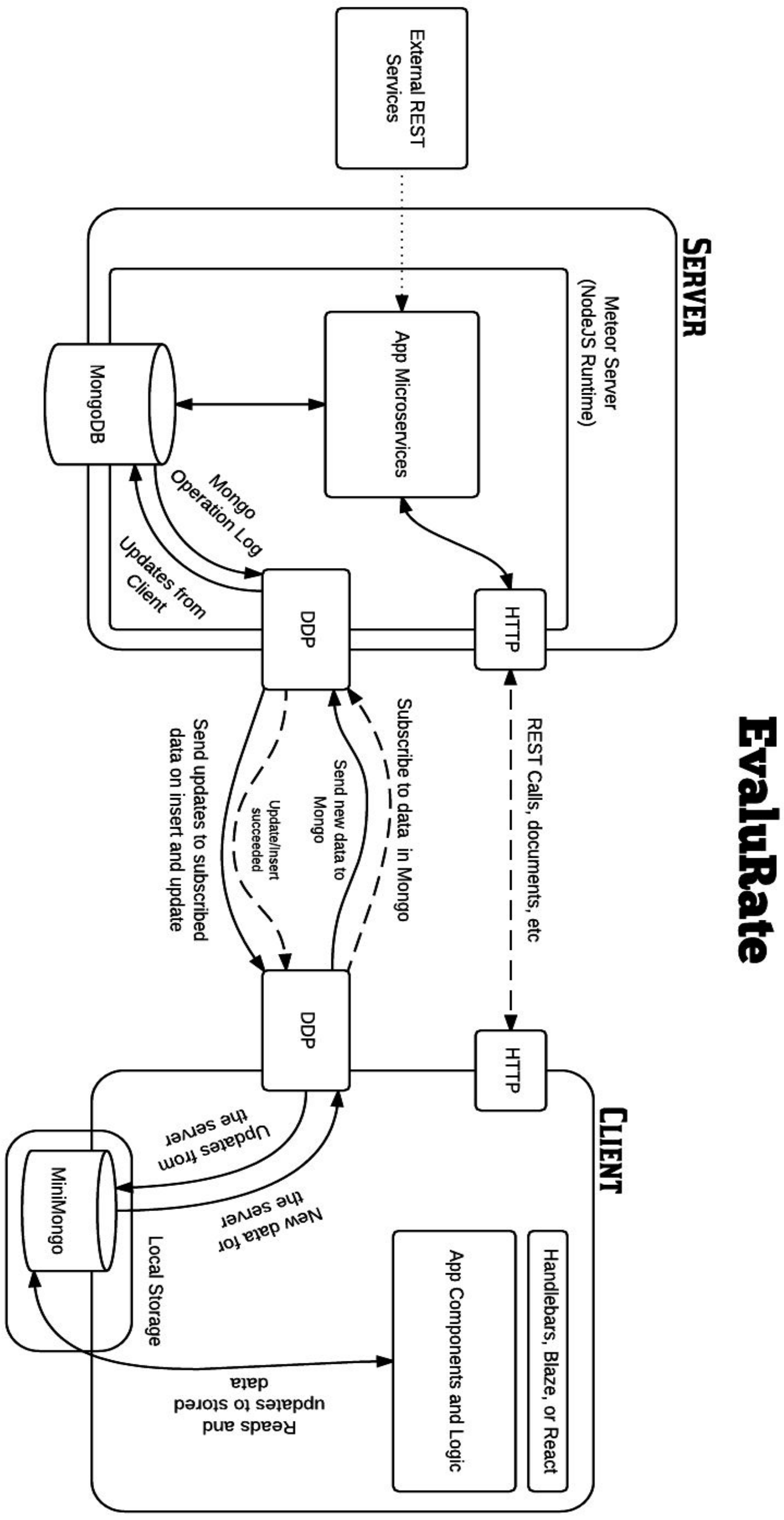

Figure 2.1 - Architecture Diagram

3. Module and Interface Descriptions

EvaluRate is split up into several logical modules to allow for ease of understanding and code organization. These modules are merely logical separations in the code based on their functionality, but serve to better explain how EvaluRate works. The modules are as follows:

- Evaluation Engine
  - The core logic of the application, handles dynamic creation of evaluations from templates, interface to complete an evaluation, and storage of results.
- Unit Management Module
  - All logic dedicated to creating, disbanding, or modifying units and their permissions.
- Data Digestion
  - All logic pertaining to displaying/manipulating the data resulting from completed evaluations.
- User Management Module
  - All logic dedicated to adding, deleting, or modifying users
- Notification Module
  - All logic dedicated to alerting a user of internal system events
- Database
  - All Schemas and models for reliable storage of system data

3.1. Evaluation Engine

The evaluation engine is a major module within EvaluRate. It is the system that facilitates the steps of the evaluation process from form creation to data collection, it allows evaluations to be created, maintained, delivered and taken. Internally, the evaluation engine consists of a few main components, each with their own unique challenges:

- Dynamic Form Builder
- Form Generation from a Schema
- Custom Input Types

3.2.1. Dynamic Form Builder

One of the biggest challenges in creating an application like EvaluRate is forms, specifically custom forms. While it is possible to have a limited number of evaluations that group admins can choose from to give to their members, we believe that giving users the option to build completely custom forms is a much better route. However, we do plan to have a system in place where users are able to use and edit predefined evaluations.

To create dynamic forms, we will be using a Meteor package called Autoform which is used to generate forms using predefined schemas. The behind the scenes of the form builder works as follows:

- With the use of Subdocument arrays users will be able to create forms of any size or type.
- To actually use these forms later, the data submitted into our collection must be formatted as another a schema itself, hence the use of Subdocuments.

Users are able to add and remove fields, choose types for each field, add names for each field and do other customization.

3.1.2. Form Generation from a Dynamic Schema

Creating custom forms is one issue to tackle, but displaying them and saving them to a Mongo collection is another issue. Just like with the Dynamic Form Builder, the Meteor package Autoform will be heavily used. As mentioned before, Autoform generates forms from a predefined schema. Here is how it all works:

- Because of the way we structure data using the Form Builder we are easily able to generate our custom evaluations by just plugging the schema into Autoform.
- The correct schema is pulled from the Evaluations collection simply by using an ID.
- When forms are submitted they are placed into an semi-unstructured collection for easy retrieval later.

The user will simply be presented with an evaluation form exactly as it was as it was designed by the Unit Admin.

3.1.3. Custom Input Types

Generally speaking there are a limited number of input types that can be used in a form, these are things like:

- Strings
- Numbers
- Booleans
- Selects

Even though any sort of evaluations *could* be done using these input types there custom inputs that can be built to make taking the evaluations much easier. Custom inputs are another functionality of Autoform that can be used to our advantage. A lot of evaluations are done by handing out a specific number of points to a specific number of members in a group. A custom input can be made that automatically generates sliders depending on the members in a group and locks the group to a specific amount of points to give out.

3.2. Unit Management

The unit management system encompases every way that the user interacts with the *unit*, the basic nesting data structure that may contain either users or units. This system can be broken into 4 main parts:

- Unit creation
- Member management
- Permission settings
- Unit division system

3.2.2. Unit Creation

Users can manually create top level root units at any time, or subunits inside units they have permissions for, by providing a name for the new unit.

3.2.3. Member Management

Admin users can invite users into units with 3 different methods:

- Email invitation
- Distributed key
- In-app invitation
    - Uses the notification system to send invites

Once users are added to a unit, the admin of that unit can manually move users into subunits of that unit.

3.2.4. Permission Settings

The permissions are specific to each unit, where each unit contains two permissions objects, one for admins and one for parent admins. These permissions objects consist of three boolean values:

- canView
    - Without view permissions, no other permissions matter, only the name of the unit can be seen.
- canEvaluate
    - Users with this permission can administer evaluations to members in a unit.
- canDivide
    - This allows users to create units within this unit.
- canAlterPerms
    - Users with this permission can change the permissions for this unit.
- canAlterMembers
    - Users with this permission can invite and remove users from the members list of this unit.

3.2.5. Unit Division System

This is a method for users to create subunits inside of units of which they are admin. It exists for the purpose of dividing members of one unit into subunits. This system combines the functionality of unit creation, member management, and permission settings into a system to make unit division easier.

3.3. Data Digestion

The administrator of an evaluation and/or the person who has permission of that evaluation can get statistical information and graphical depiction of peer evaluation results. This module will use responses that have been submitted by users to display the data in as many relevant ways as possible.

- Download Information
- Data Visualization

3.3.2. Download Information

The administrator or a person with permission to the evaluation will be able to download statistical information and results from surveys to their desire file form:

- XML
- CSV
- Others

3.3.3. Data Visualization

The administrators and person with permission to the evaluation will be able to view the resulting data in a variety of formats:

- Bar Graph
- Pie Chart
- Line Graph

3.4. User Management

The User Management Module deals with a lot of the functionality that users come to expect from any modern web app when it comes to their own personal accounts. Luckily, Meteor has core functionality that deals with user accounts. There are also a large number of packages that expand this functionality.

The following is a list of items that will be implemented on the back of of the core accounts packages.

- Account Creation
    - Email/Password
    - CAS
    - GMail
- Login
- Logout
- Password Recovery
- Add and Edit Account Details
- Delete/Disable Account

The first four pages are available to users that are not logged in, and the last two options are only available to those that are already authenticated.

3.5. Notifications

The administrator and persons with permission will have the ability to create events. Also the notification will be broken down into three types of notification: text, email, and in app notification.

- Create Events
- Notification
  - Text
  - Email
  - In App Notification

3.5.2. Create Events

Administrator and persons with permission to the evaluations will be able to create events that will inform on new events, updates on events, and alerts subunits on due dates.

3.5.3. Notification

Notification to the units will be control by the administrators and people with permissions to the units. There will be three type of notification that the unite managers can push information and alerts to their units:

- Text
  - Signing up for text notification is an option. But if users decide to receive text notification, notification will be push to users mobile phone at user own rates.
- In App Notification
  - Administrators and people with permission access will be able to push notification to all people in a unit.

- Email
    - Email notification will be automatically push to users’ email services; users’ email will be the main source of notification because users will have to signup with a email before using Evaluate.

3.6. Database

Since Meteor uses MongoDB for it’s database, and MongoDB doesn’t employ strict schemas, we are using the SimpleSchema2 package to create strict schemas.

3.6.2. Unit Schema

Field Name	**Field Type**	**Field Description**
_id	String	Unique ID generated by Meteor
owner	UserId	The creator user’s ID by default
name	String	Name for unit set by creator
children	[UnitId]	List of unit IDs that are immediate children
admins	[UserId]	List of user IDs that act as admin to unit
members	[UserId]	List if user IDs for all members of unit
evaluations	[EvaluationId]	List of active or scheduled evaluation IDs
adminPerm	{ canEvaluate: boolean, canCreateSubunits: boolean }	Permissions for admins of this unit
parentAdminPerm	{ canView: boolean, canEvaluate: boolean, canCreateSubunits: boolean }	Permissions for admins of all parent units

3.6.3. Unit Schema

Field Name	Field Type	Field Description
_id	String	Unique ID generaged by Meteor
owner	UserId	The creator user's ID by default
unitId	[UnitId]	List of Unit IDs the project is attached to
openDate	Date	When the project starts
closeDate	Date	When the project closes

3.6.4. Evaluation Schema

Field Name	Field Type	Field Description
_id	String	Unique ID generated by Meteor
templateId	EvalTemplateId	ID of corresponding evaluation template
owner	UserId	The creator user's ID by default
projectId	ProjectId	project ID that this evaluation acts on
dueDate	Date	due date of the evaluation

3.6.5. Evaluation Template Schema

Field Name	Field Type	Field Description
_id	String	Unique ID generated by Meteor
title	String	User defined title
owner	UserId	The creator user’s ID by default
fields	[Object]	A list of question objects which construct the form later

3.6.6. Response Schema

Field Name	Field Type	Field Description
_id	String	Unique ID generated by Meteor
userId	UserId	The ID of the user that completed the evaluation
templateId	EvalTemplateId	The ID of the evaluation template that identifies the questions for this response
evalId	EvaluationId	The ID of the evaluation instance that the user responded to
data	Object	The data collected from the evaluation

4. Implementation Plan

For the most part the implementation plan has been largely centered around the User Interface design. Once the user interface had a good design, the rest of the functional work started to fall into place. An actual plan would be very hard to predict, as well as actually stick to. Implementation is done in a Kanban style fashion, where we prioritize which issues need to be done at a certain period of time and reevaluate that constantly. So far we have realized more than a few times that we want to implement one part of a module, but realize that we actually need to implement another piece of a module first to make workflow more efficient.

week	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Database	🟦	🟦	🟦											
UI Design	🟩	🟩	🟩	🟩										
Evaluation Eng			🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦			
Unit Manage			🟦	🟦					🟦	🟦	🟦	🟦	🟦	🟦
User Interface					🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Usability Test								🟪	🟪	🟪	🟪	🟪	🟪	🟪

🟦 Backend	🟩 Frontend	🟪 Testing

4.1. Summary

The implementation plan starts with getting a solid schema for the database, as well as a good picture of what the user interface will look like. These two things will drive development of the rest of the application by giving a solid foundation to both the server side and the client side of the application. Usability testing is outlined to continue through the majority of development, which will hopefully allow us to catch problems early.

4.2. Work

With Meteor it is hard to describe actual roles in the development of our application because the client side contains much more logic than a standard web framework. A large portion of code in Meteor runs on both client and server.

- Client - Templates, Interface logic
    - Brian Saganey
    - Brandon Paree
- Server - Units management, Evaluation Engine
    - Conner Swann
    - Dylan Grayson
- Both - Mongo Collections, Routing
    - Dylan Grayson
    - Brandon Paree
- Testing
    - Everyone