



MINESTM

PivotalTM

RetroActive for Pivotal Tracker

Team:

Espen Roth
Jennifer Jacobs
Jesse DeMott
Taylor Rummel

Client:

Morgan Whitney

Table of Contents

1	Introduction.....	3
1.1	Client Description	3
1.2	Product Vision.....	3
2	Requirements	3
2.1	Functional Requirements	3
2.2	Non-Functional Requirements	4
3	System Architecture.....	4
4	Technical Design	5
5	Design Decisions	7
5.1	Framework	7
5.2	Database.....	7
5.3	Redux	8
6	Results.....	8
7	Appendix	9
7.1	Deployment	9
7.2	Additional Figures	9

1 Introduction

1.1 Client Description

Pivotal is an agile software development company that provides several software solutions. Pivotal has a focus on enabling clients to improve productivity through their software such as Pivotal Web Services and Pivotal Tracker. Tracker is a project management tool intended for software teams. It allows teams to organize and review the design process of a product, from research to implementation to debugging to production. The software is divided up into smaller tasks called stories that are organized into a logical order.

True to the agile process, each team at Pivotal performs a bi-weekly retrospective (or “retro”). A retro is an hour-long meeting which gives the team an opportunity to openly discuss how they are doing. Anyone can bring up a discussion item, categorized by how they feel: happy, confused, or sad. Important items are discussed, and the resulting action items are assigned to an individual to be resolved. After a retro is completed, the action items are recorded manually on a Google Sheet to be discussed at the beginning of the following retro.

1.2 Product Vision

The goal of RetroActive is to capture and streamline the retrospective (retro) process in a web application. Currently, retros are done verbally with a moderator recording discussion items on a google spreadsheet. Voting is done manually by raising hands and expressing interest. RetroActive will improve this process. Functionally, users must be able to create categorized discussion items, to vote for items they deem important, and to maintain the retro on a database for future reference. Any updates to a retro needs to also update the displays of all other clients in real-time. Moderators of the retro should be able to create an action item, which will correspondingly create a story in Pivotal Tracker, based on a discussion item and assign a team member to the task so that this item can be improved upon for the next sprint. This will require the application to utilize the Pivotal Tracker API to get and push information for the team.

2 Requirements

2.1 Functional Requirements

- The application should use Google Authentication to verify users.
- Users will be asked to provide a Tracker API token to be stored on the application’s database.
- Tracker API tokens should be used to get information about the user.
- Users should only see projects they are a member of.
- Users can only create retros for projects they are members of.
- Users can create, edit, vote, and un-vote for items in a retro.
- Users should be able to create and edit an action item from a retro item and a corresponding chore for it should appear in the Tracker project.
- Users can visit and participate on a mobile device with a reasonable format.

2.2 Non-Functional Requirements

- The application needs to use a database service provided by Pivotal Cloud Foundry.
- The application needs to use a web backend supported by Pivotal Cloud Foundry.

3 System Architecture

Our client left decisions about system architecture up to us. The only requirement was that we could deploy to Pivotal Web Services (PWS) for production. Therefore, since Ruby on Rails was supported by PWS and we had some backend experience with Rails we decided to use that for our server. We also made the decision to use a NoSQL, or non-relational, database to learn something new and because a document store fit our needs when storing retros. The Mongoid gem which connects ActiveRecord to Mongo was extremely useful, allowing us to use normal object syntax when interfacing with the database.

We also used a continuous delivery process. Whenever we finished a user story we would deploy that version to PWS (retroactive.cfapps.io), giving us constant confidence that we had at least one working version. This also made it possible for anyone to log in from any device to see the current state of the project. This allowed our client to see our progress and give us constant feedback, again true to the agile process.

Since we were also interfacing with Pivotal Tracker to store data related to action items, we had to design our system to sync two separate data sources. To deal with this we had the client do all of the syncing of data, and stored only a minimal amount of the action items' data on our side.

Clients are provided a single bundle.js file of the entire website (except for CSS and assets) built from the WebPack bundler tool. This allows us to ship the React.js library. React provides JavaScript objects called "components," which render into DOM objects when the page is loaded. All HTML and logic can then be translated to JavaScript and placed inside the bundle. React components can render JavaScript-manipulated data as HTML, create other components and pass data/functions to components nested within them. Additionally, components can have an individual state. Data flows from top to bottom, biggest components to smallest, meaning all of the state changes come from the root node.

Our design flow has a top-level component that creates sub-components and passes state variables and/or functions (called props) as needed. Props are what the child component receives as data from the parent, passed through like HTML attributes. Since functions are First Class Objects in JavaScript, functions they can be passed down just as any other datatype can. State variables, used only in the root node, are the authoritative source of information for the sub-components. They determine what the root passes down the children as props.

Functions passed down as props means that the root component can configure the children on how to React when one of the components' events are fired. Since the root node is the only one

which affects 'state,' it has all of the functions which handle the state of itself and everything below. When the state of the root node changes all affected components re-render themselves to reflect the change. This is done without the user ever having to refresh the page.

4 Technical Design

Our application can be split into several components: the API, the mobile and desktop clients, and the Pivotal Tracker integration.

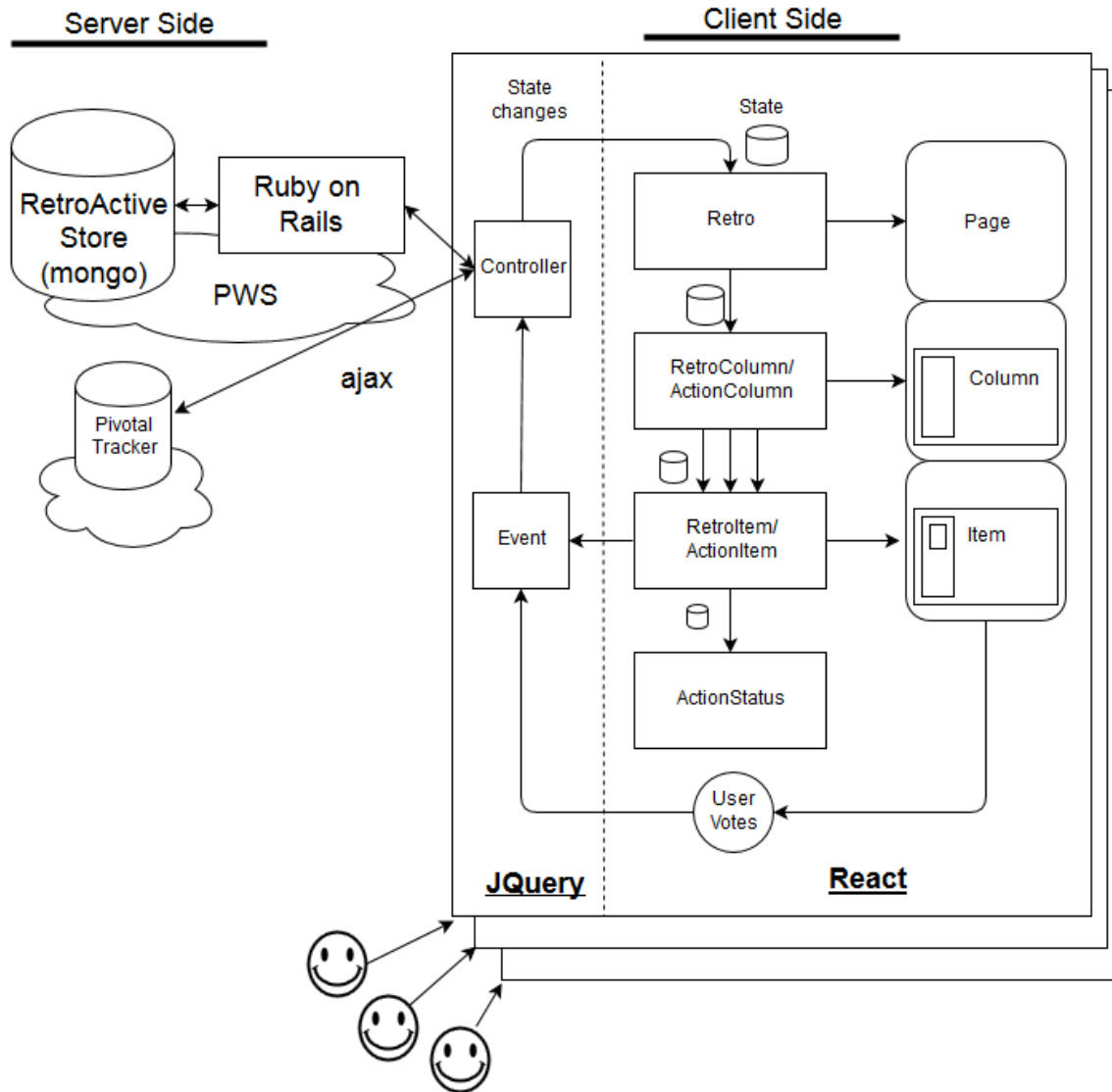


Figure 1, the overall design of RetroActive. The left side indicates that our application interacts with the Pivotal Tracker API and the Mongo database.

Our API (built with Ruby on Rails) interacts with the Mongo database, which stores the retros and all of the associated information. In addition, when someone creates an account on RetroActive, we store their name, email and Tracker token so that they can continue to access the application without supplying their token every time. Once a token is supplied, the users' Tracker projects are loaded onto the dashboard. A user can view, create and edit retros for any project of theirs on Pivotal Tracker.

Since we chose to use Ruby on Rails, the design of our back-end was mostly decided for us - a Model View Controller (MVC) application. We decided to use MongoDB for our database storage as a JSON file was a good model for our text-based data. In addition, this gave us more flexibility in the layout of the database because it was not as rigid as a SQL database, which would have required more setup time to design the schema.

Our original design used Redux to push the React state of an entire retro to cloud storage every time there was a change. Due to time constraints we dropped Redux in favor of using ajax and jQuery. Using this model we only pull data, such as number of votes, from a single retro item when the version number of the retro has changed. The version number is updated each time that the database is changed, and when the local version number does not match the database's version number a new version is pulled.

The page is statically generated by Rails, and updated dynamically by the React framework. Each page is built from several React components each with a specific purpose. This created a good organizational structure where code could be reused when needed.

Although Rails normally follows MVC with its own views, we only had one Rails view in our application. The Javascript framework we used (React) allowed us to render views dynamically on the client without having to use full page reloads. Then the Rails server is used just to respond to Ajax requests for data the client needs.

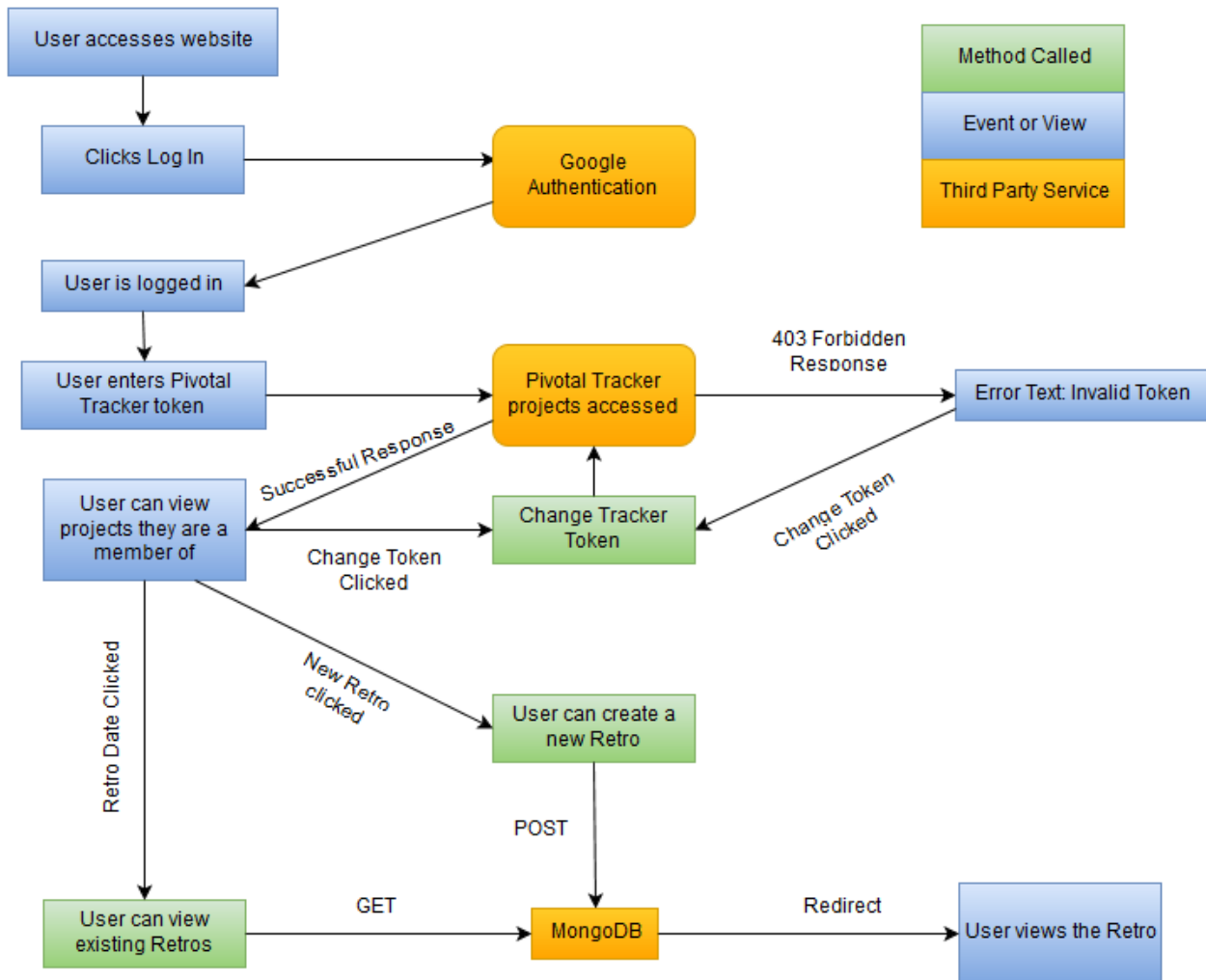


Figure 2, Workflow for login and dashboard.

5 Design Decisions

5.1 Framework

The framework chosen had to be supported by Cloud Foundry, which offers a wide selection of options. Ruby and Rails are easy to learn, set up, and plenty of good documentation exists. Rails is used on the Tracker back-end; this will enable our client to continue to develop the product after field session. In addition, several members of the group were already familiar with Ruby and/or Rails. Consideration went into using Node.js to more naturally fit our Javascript heavy product, but the additional learning curve for a small functionality pay-off was ultimately deemed not worthwhile.

5.2 Database

The choice of database was the major bootstrap decision. Like the framework, the database needed to be a provided service through Cloud Foundry, which offers several common databases at the click of a button. The decision was primarily from a functionality and usability standpoint.

Mongo was chosen because our product's data naturally fits well into a JSON document format. Although SQL would meet all functional requirements as well, maintaining it with a useable scheme would require a significant time investment that Mongo did not.

5.3 Redux

Redux is a Javascript addition which reduces the volume required for data transfers. Getting Redux set up was only a small problem. The larger issue was the amount of time it would take to learn how Redux changes the process of managing React state and the time to apply it to our product. Specifically, we would have to learn how to manage state changes for clients such that other clients get close to real-time updates to reflect the changes in an efficient manner. The idea of having the state change without having to refresh the page isn't unique to Redux, so it wasn't crucial to support it even if it does it better than the alternatives.

For those reasons our team decided not to make the time investment in learning the Redux framework; rather, we would focus on completing the project using the technologies we already knew (jQuery & ajax).

6 Results

To date the app runs on both Chrome and Firefox (Internet Explorer untested) and a wide variety of browsers for the phone, all successful. Older browser support is limited primarily by their CSS3 support, however all current versions of the most popular browsers do support CSS3 which is important.

The most glaring problem with the current implementation is how the website state changes as new data is pushed onto the server, and then to all clients. The state for the client's view does update automatically as information changes as per requirement. Currently the client's page sends GET requests to the server at short, regular intervals to see if the version of the retro the client has is up-to-date. A retro's version changes whenever an item is created, deleted, or voted for. If the version numbers do not agree, the client pulls the entire retro JSON document from the server and changes the client's page state to match the latest version. This is a sub-optimal solution to ensuring clients have the latest state of the retro. A better solution would be to have the server send a signal to the client whenever a change is made, and have the client pull the retro data. This would reduce network traffic by limited network traffic to when it a change has actually been made.

All requirements for version 1.0 have been met. Currently the app is deployed onto Pivotal Web Services and is publicly available for anyone to use. We have tested this application by performing retrospectives with the application once a week for our team as well as having other teams test and critique it by using it in their retros as well.

Additional features such as assigning moderator status to the retro creator, providing options to change the type of an action item from chore/story/bug, or allowing users to comment on retro items are all extra features that were not implemented due to time constraints.

7 Appendix

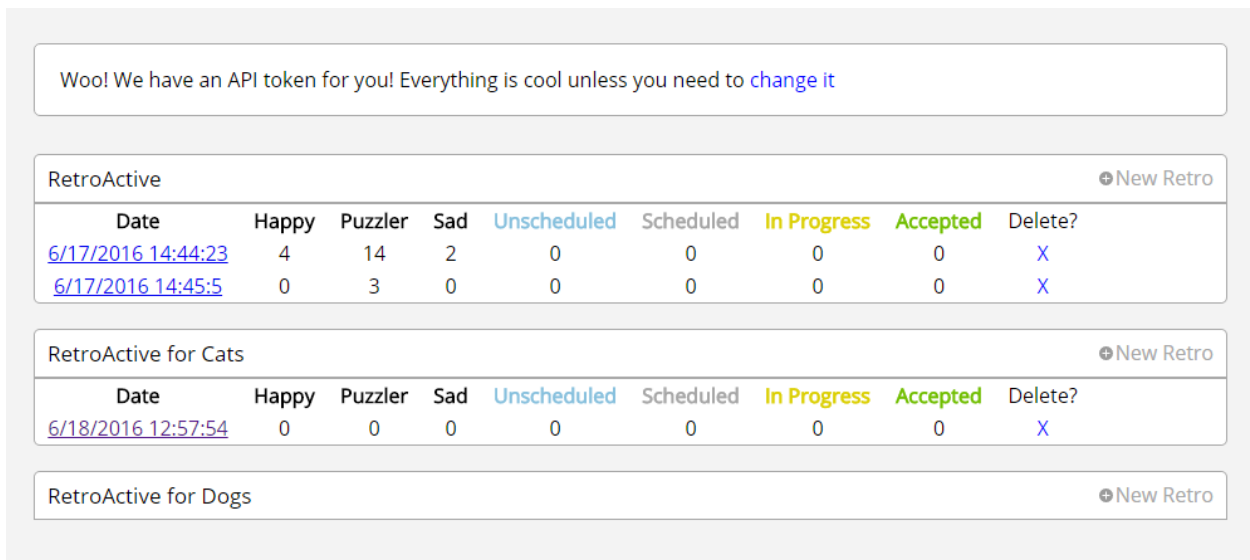
7.1 Deployment

We have created an installation script that will set up the development environment on a system running OSX which is what our client will be using to further develop RetroActive. This script is included in the project's git repository.

For development purposes we also have a build script named `build_client.sh`. This allows you to build the client without pushing to PWS for testing purposes. This script also copies any files within `node_modules` that we had to customize. Any of the files that are customized are placed into the build folder. This is because `npm install` is run during the build and will overwrite any changes made within `node_modules`.

We also have a deploy script in the the project folder that will use the above mentioned script to build the javascript client and deploy the app to PWS. This should only be used when the project is production ready.

7.2 Additional Figures



The screenshot shows a dashboard with a notification at the top: "Woo! We have an API token for you! Everything is cool unless you need to [change it](#)". Below this are three sections, each with a "New Retro" button:

- RetroActive**: A table with columns: Date, Happy, Puzzler, Sad, **Unscheduled**, Scheduled, **In Progress**, **Accepted**, and Delete?.

Date	Happy	Puzzler	Sad	Unscheduled	Scheduled	In Progress	Accepted	Delete?
6/17/2016 14:44:23	4	14	2	0	0	0	0	X
6/17/2016 14:45:5	0	3	0	0	0	0	0	X
- RetroActive for Cats**: A table with columns: Date, Happy, Puzzler, Sad, **Unscheduled**, Scheduled, **In Progress**, **Accepted**, and Delete?.

Date	Happy	Puzzler	Sad	Unscheduled	Scheduled	In Progress	Accepted	Delete?
6/18/2016 12:57:54	0	0	0	0	0	0	0	X
- RetroActive for Dogs**: A table with columns: Date, Happy, Puzzler, Sad, **Unscheduled**, Scheduled, **In Progress**, **Accepted**, and Delete?. (Table content is not visible in the image).

Figure 3, The Dashboard shows only projects you are a member of and any retros that have been created for it. The Dashboard also provides information regarding the state of all retros.

Your token is invalid. Please verify it and try again.
Woo! We have an API token for you! Everything is cool unless you need to [change it](#)

Figure 4, if an invalid Tracker token is provided. The dashboard will indicate that the user needs to re-check their token. Because it is invalid, there are no projects or retros the user can view.

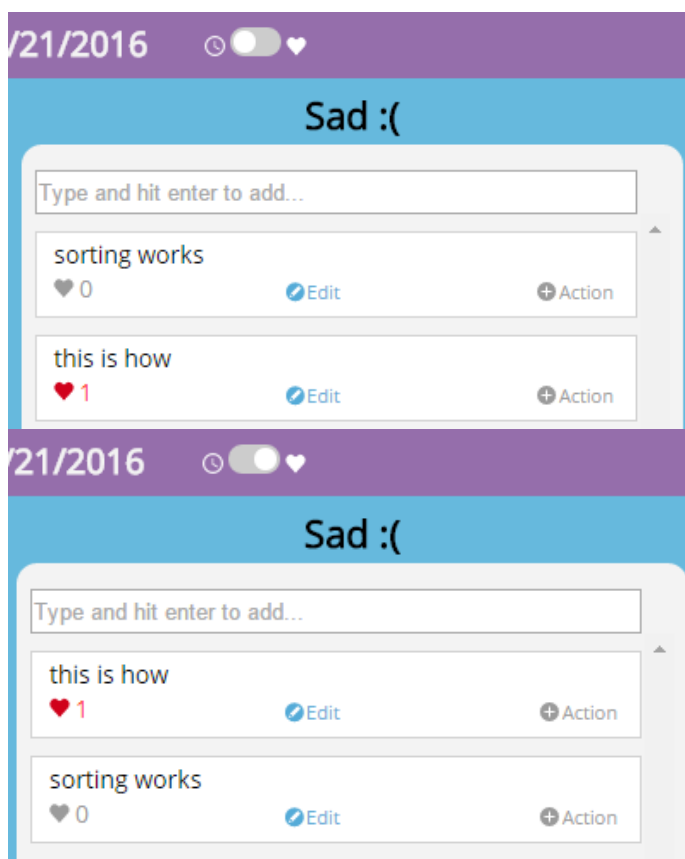


Figure 5, users can change between two modes of column item sorting by pressing the button at the top. The default option is sorting by time, where newer items appear further up. The second option sorts by votes, with items with more votes appearing at the top instead.

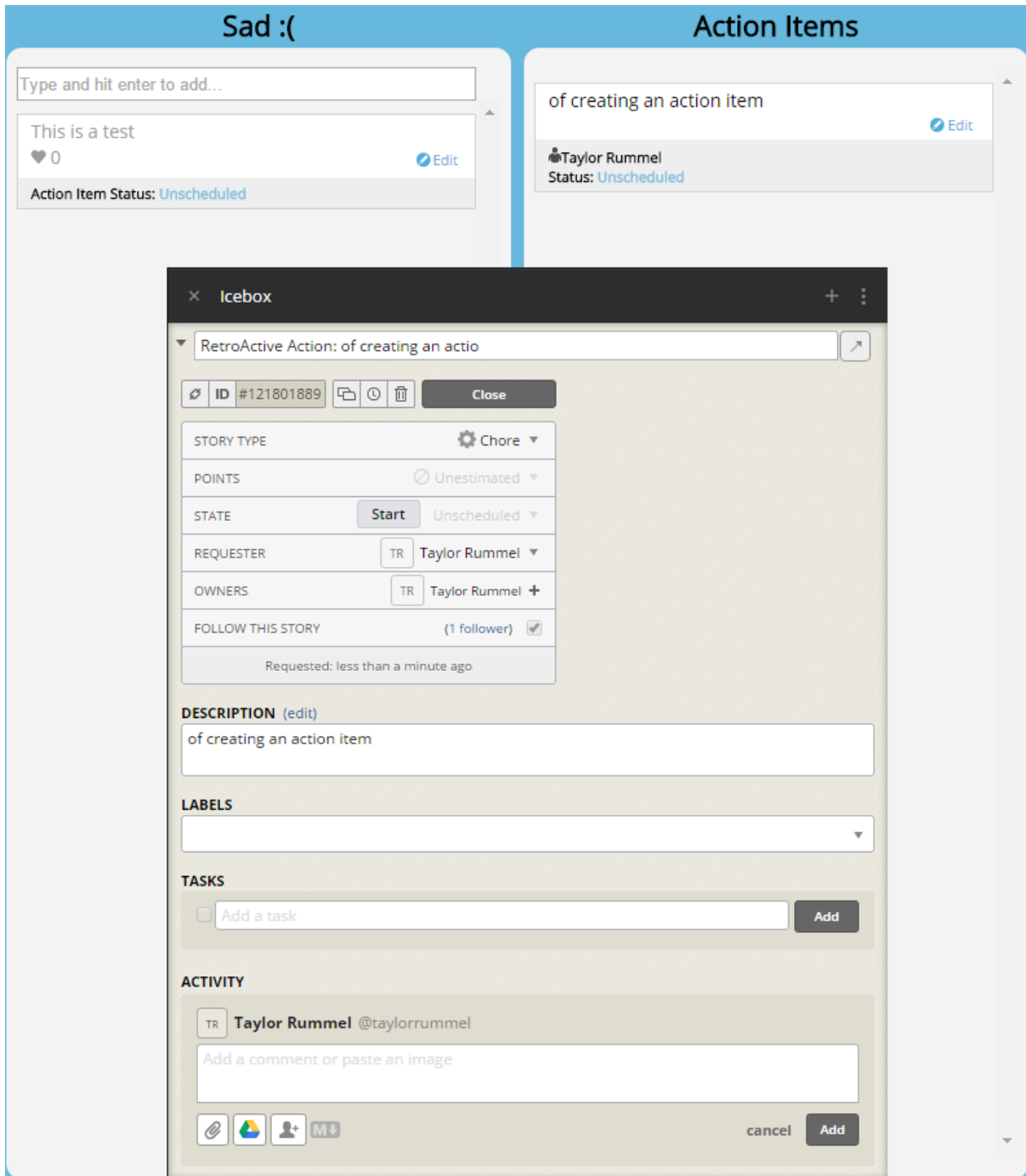


Figure 6, an action item created from a retro creates a corresponding chore in the project. The action item will reflect the current Owner and Status as it changes in Tracker. The title and description will reflect the text from the action item. The Requester is whoever creates the action item, typically the retro moderator.