Real-Time Marketing and Sales Data

6/21/2016

Chase West
Eric Sheeder
Marissa Renfro

Table of Contents

Introduction

About JumpCloud:

Typically, to manage a business's employees and software/site access, an LDAP (Lightweight Directory Access Protocol) system is implemented in-house (i.e. on a local server). Managing a system in-house can take time and resources away from a business that could be better spent using a vendor. JumpCloud's mission is to fulfill this need and allow directories to be managed on the cloud. They offer the ability to centralize control of employee identities. This means that the identity of an employee can be automatically authenticated when signing into various applications like Dropbox, Office365, Salesforce, etc. The authentication system can also be linked to the employee's Google account such that remembering credentials can be made easier. When it comes to management, the JumpCloud administrator can take advantage of useful features such as remotely installing software and managing user accounts (adding, updating, and deleting). Overall, JumpCloud is providing a security solution and a "modern alternative to Active Directory and LDAP".

Product Vision:

Founded in 2012, JumpCloud has grown significantly in the past few years and is now looking at better ways to analyze their sales data and client patterns. Currently, while JumpCloud has kept track of large amounts of customer data, they rely on manual queries for viewing data about Monthly Recurring Revenue (MRR), the number of systems/users, and the number of administrators. This 'primitive' query approach is not only cumbersome, but also does not accommodate non-technical users. If a person with limited or no database experience wanted to query the database, he or she would be completely lost (or would have to befriend a coder). Also, it can be hard to grasp what the data is representing without graphical assistance. Therefore, visualizing the data via interactive graphs and charts would further the insights the data provides. The primary goal of this internal website is to improve analytics of data and empower non-technical users with an easy-to-use web interface; specifically, the web application should act as business intelligence software for the sales team and a user look-up interface for the support staff.

Requirements

Functional Requirements

- Homepage
    - MRR (Monthly Recurring Revenue) graph
    - System count graph
    - User count graph
    - Admin count graph
    - Admins logged in graph
    - LDAPs (Lightweight Directory Access Protocol) active graph
    - Time adjuster that allows observation of data in different timeframes
    - Automatic refresh of the page every day
- Search by company page
    - Lists companies with names like the search criteria
    - Link to each company's sales information page
    - Link to each company's admin and user information page
- Search by ranges page
    - List information of companies that fall within user defined search parameters such as:
        - MRR
        - System count
        - Admin count
        - User count
        - LDAP active
    - Has link to each company's sales page
- Company sales page
    - Display graphs similar to the ones on the homepage for given company
    - Display information about company for most recent day
- Company admin and user information page
    - Display admin and user information such as
        - First name
        - Last name
        - Id in the database
        - Email

Non-Functional Requirements

- Ease-of-use for non-technical users (e.g. salespeople, support team)
- Queries must be done using MongoDB
- Recommended to use Go or Node.js for web page
- Code must be put on GitHub for the client to get

System Architecture

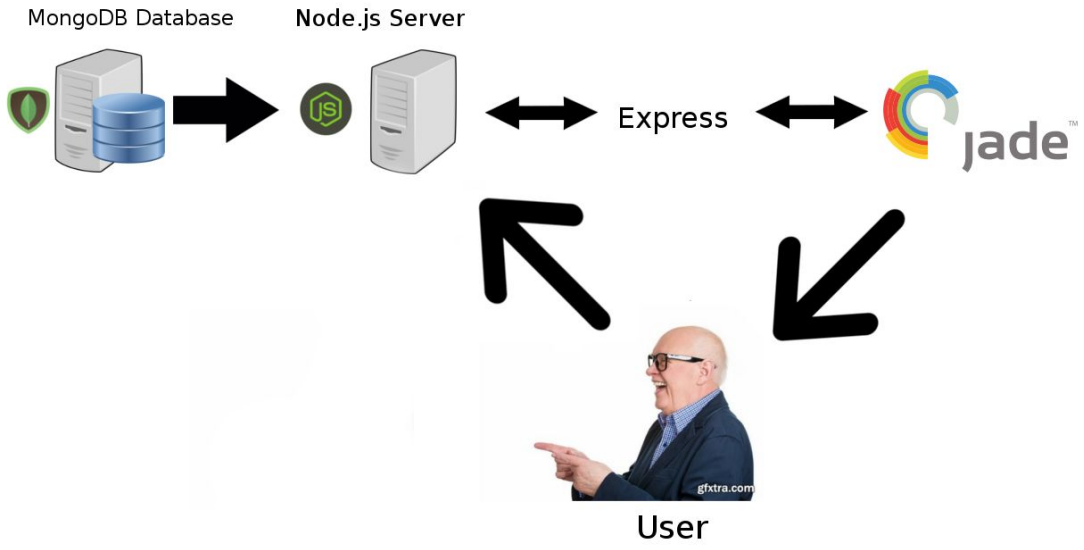


Figure 1: System Architecture

Figure 1 shows the overall architecture of how the main web server interacts with the data server and the user. The Node.js server handles incoming requests from the user and receives information from the Mongo DB based on the user's query. The Node.js server uses Express to handle both page navigation as well as HTML generation based on the data from the query. In the end, Jade generates the final HTML file that is generated in the user's browser, which also serves as a way for the user to interact with Node.js (and indirectly query Mongo).

Due to the fact that Mongo is schemaless, there is no database UML to display.

Technical Design

Our website has six main pages to it: the homepage, search by company page, search by ranges page, search by percent-change page, specific company sales page and the company userlist page. In the different search pages, the user is allowed to enter inputs and queries are generated based off of their inputs. The inputs from the forms in the pages is then sent back to the server-side of the system and the queries are generated there. Then, those queries are used to search through the database for what the user is looking for. The data gets sent from the database to the server-side of the system. From the server-side of the system, the data gets sent to the client-side and a page is generated using javascript and jade templating.

Homepage and Specific Company Page

The homepage is the page that displays JumpCloud's aggregated sales data for a given time period. We set the default to be the previous 30 days. All of the information displayed on this page is showed on graphs. The information consists of the bullets listed above in homepage portion of the functional requirements.

The specific company page is very similar due to the information being displayed. Most of the information displayed (monthly recurring revenue, user count, etc.) is the same except that it is for a specific client of JumpCloud. On top of the graphs, the page has a table that displays all of that company's statistics for the end date chosen. By default, this end date is the most recent day with data.

Due to these pages being very similar, we used a bit of inheritance in the design. To generate the HTML for these two pages we used the same jade template called 'homepage.jade' and check for if a company name is present and being passed to the template. From here, we can easily let the template know which page is being displayed. This helps us decide whether to create the table of most recent sales information, which headers and titles to display, and which graphs to generate.

In the specific company page, we omit the "LDAP active" graph and the "number of paying companies" graph from the page for obvious reasons. The user will know if the company has LDAP active based off of what the table that is generated displays.

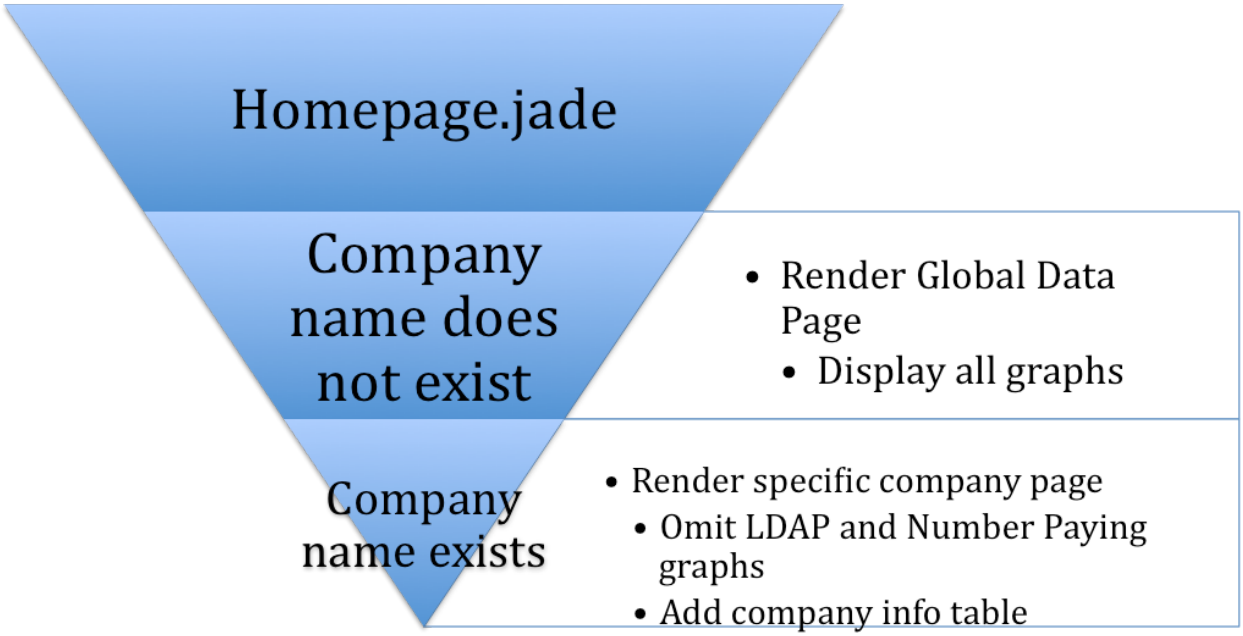
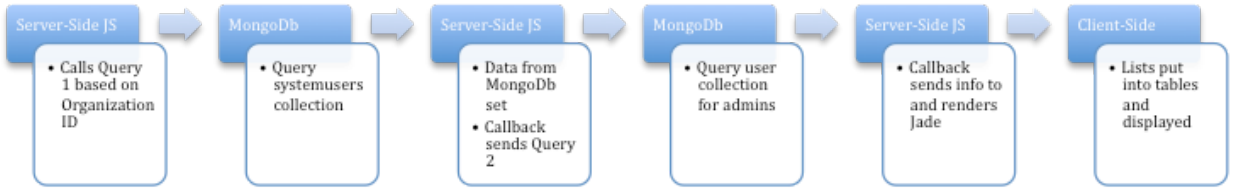


Figure 2: homepage.jade differences

MongoDb and Multiple Queries

When generating the user and admin lists for the specific company, we had to get data from multiple collections within in the database. For an SQL database, this is a simple and quick task because of the JOIN operation. With a JOIN clause in SQL, getting the necessary data from multiple tables (collections in a NoSql database) can be done in one query. However, Mongo does not have this capability. Because of this, we were required to do multiple queries to link foreign keys for the “organization id” of the company and get the data we needed. A schema of the three collections in the db is shown below. After you get the data, you can then aggregate it all and put it into a data-structure of your choosing. With Javascript and Node this can be a very tricky task. Due to the asynchronous nature of javascript, queries and functions are called in an unpredictable order. Therefore, we must use callbacks to get everything to run in the order that we want. In our implementation, we query the database to get all the basic users in an organization. On each query usually, we use a callback to generate render the jade template to generate the HTML. For the first query in this function, in the callback that is called after we get the data, we put the data into a Javascript object and then call the next query that gets a list of admins and their information for that same company. In the callback for that function, we then grab the data that we set from the original query and pass it all to the jade template that is rendered in the second callback. Figure 3 provides a visual representation of what is happening.



Users
{ "_id" : ObjectId,
"__v": int,
"allowed_system_count": # of allowed systems,
"bad_login_attempts": # of bad login attempts,
"connectKey": connect key,
"created": date of creation,
"disableIntroduction": boolean,
"disabled": boolean,
"email": admin email,
"enableMultiFactor": boolean,
"firstname": firstname of user,
"lastAccess": date of last access,
"needsPasswordReset": boolean,
"organzation": ObjectId,
"pendingSystemCerts": certificates pending array,
"registerToken": token register,
"registered": boolean,
"role": ObjectId,
"systems": array of systems under admin,
"usersTimeZone": timezone of users
}

organization_snapshots
{ "_id" : ObjectId,
"active_systems_count" : count of active systems for a company,
"admin_logged_in" : if admin logged in that day,
"admins_count" : number of admins for a company,
"billing_explanation" : description of billing information,
"date" : Date of the record e.g. --ISODate("2015-08-14T06:45:03.871Z")--,
"internal_org" : if they are an internal organization,
"ldap_active" : if the feature "ldap" is active,
"monthly_max_users" : number of monthly max users,
"mrr_in_cents" : Monthly Recurring Revenue,
"name" : Who submitted the data,
"organization" : Organization_id e.g.--ObjectId("54354c474444b514110067c0")--,
"paying" : if the company is paying for the service,
"sfdc_name" : name of the company,
"users_count" : how many users the company has }

Systemusers
{ "_id" : ObjectId,
"__v" : integer,
"activated" : boolean,
"addresses" : addresses of the user,
"allow_public" _key : boolean,
"email" : email of user,
"enable_managed" _uid : boolean,
"expired_warned" : boolean,
"externally_managed" : boolean,
"firstname" : first name of user,
"lastname" : last name of user,
"ldap_binding_user" : boolean,
"organization": ObjectId,
"password_date" : date of password,
"password_expired" : boolean,
"passwordless_sudo" : boolean,
"pendingProvisioning" : boolean,
"phoneNumbers" : array of phone numbers ([work, home, mobile, ect]),
"relationships" : array of relationships,
"ssh_keys" : array of keys,
"sudo": boolean,
"totp_key": key for totp,
"unix_guid": guid,
"unix_uid": uid,
"username": username of user
}

Figure 3: Schema and Process

Design and Implementation Decisions

Node.js

Since our project consisted of building a functional website, we had to consider languages that were used for web development. Our client suggested that we use either Node or GO. We decided to use Node instead of GO because it seemed as though there was much better documentation, more tutorials that helped get a website up and running quickly, and a lot more existing packages and tooling. We learned that GO is much better for concurrency, scalability, and performance. However, due to the nature of our project, we decided that we did not absolutely need to use a platform that was better in those regards.

Chart.js

Our project heavily relies on graphs to display the data that we want to display. There are many packages that allow for one to easily create a graph in Node so we decided to use one of those instead of creating one ourselves. This led to the search for the right one. We looked at many other graphs with the functionality that we needed in mind. While many were very “pretty” and had a lot of functionality, we found that the most robust was ChartJS. While being pretty and easy to implement, right out of the box it allowed the “hover over a node to display the specific values” functionality which was a major necessity for us.

Bootstrap 4

We know how to use CSS to make a website pretty but we had all heard of the famous “Twitter BootStrap” package that allowed one to easily “beautify” a website. Because of this, we decided to give it a try so we can learn it for later ventures as well as make this current project look professional and well-developed.

TableSorter.js

We needed a way to easily display data in a responsive table. While we know how to do this in HTML and CSS, we decided to speed up the process by using TableSorter. This package allowed us to easily import data and populate a table that was responsive. TableSorter also allows for dynamic sorting on clicks and loads.

Results

Performance testing results

All pages and queries load in a reasonable amount of time (<3 seconds). The one exception to this is the global data page, which takes around 1 minute to load. However, this is not an issue as this page is likely to be loaded only once per day in order to display the data on a monitor in the office. In addition, we found that indexing the mongo database by date greatly decreased the time on some of our queries, so we will recommend to our client that they do the same.

Summary of testing

This project has been tested in the current versions of Internet Explorer, Firefox, Google Chrome and Safari. The project works well and as expected with no flaws in Internet Explorer, Firefox and Google Chrome. However, in Safari, the time is off for our graphs. The only difference is that instead of displaying the date of entries on the x-axis, Safari causes our site to display the time instead. We will advise the client to use any of the three browsers that work properly. All functional and non-functional requirements have been met and we have no problems with our data or web-pages.

Results of usability tests

Usability tests were conducted at the client's office in Boulder. The sales and support teams all found the site easy to use. All could navigate the site easily and were able to easily see the data they wanted to see.

Future work

Possible future work could include doing more to prevent against XSS attacks and adding more features that allow the support and sales teams to do more. Throughout the course of using the website, the sales team and support team will come up with more queries that could be useful. Other future work could include providing projections for sales and other data.

Lessons learned

- Timezones and daylight savings time provide a surprising amount of complexity. This is because javascript does time in local time and MongoDb does time in UTC time. We figured out a few workarounds by adding the time difference to the local time to get to UTC.
- XSS attacks are common attacks against web-applications. They are done by entering data into a form or database that looks like javascript or html code (e.g. Name: “<scRipT> alert(1) <ScripT>). Javascript provides ways to prevent against these attacks by providing functions that parse strings and convert special characters in them to characters that are not harmful. These functions are “escape(string)” and “unescape(escaped string).” The way we worked around XSS, however, was by having our queries omit data that contained possibly malicious strings.