Touchbase

Prepared for: Keshav Murthy, Gerald Sangudi

Prepared by: Pranav Mayuram

August 11, 2015

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Objective

The overall goal of this project is to produce an open source social network template that displays usage of the Couchbase NoSQL database, including its query language, N1QL. The project should make the code easily accessible to the public, such that their own customization of this product could be achieved quickly and painlessly. This will allow developers to spend the least time changing the basic necessities in the backend, and more time modifying the UI and the document models needed to do what they choose.

The final product should have an available version which can be customized and used by others who would like to build upon the template, implementing their own UI and data. There will also be a second product which will be a social network to be used internally by Couchbase employees. This will be an example of a product built upon the social network template.

Data

The data will contain a detailed user object stored in one bucket of the Couchbase Server, and will contain basic attributes like an email, name, hashed password, etc. There will also be a second data bucket which will contain base64 binary blobs for user profile pictures that the user has uploaded, and this object will have the same document ID as the user document with '_pic' appended after it so that the two can be linked when one wants to view a complete user profile in the social network. Anyone who uses the template in the future will be free to alter the user model and other aspects of the data as they please.

Functionality

The final functionality that this app will have will include the ability to:

Create a User Profile - with Profile Picture
Search User Attributes
Track User Statistics
Sign Up With Email Verification
Login to See User Information and Access Account
Edit/Delete Personal Information
Create Posts
Edit/Delete Posts

The social network template will aim to show different examples of potential user-input fields, and user data that could be tracked. It will also aim to show different forms of N1QL queries that could be performed to present such data in different fashions.

User Experience

The UI will be designed with a side-navigation bar to maneuver through the different sections of the website. The website is built to be mobile optimized and is made so that the side-navigation bar is extended on large devices, but must be popped out using a button on medium and small devices. This app was built first for web, and so it uses HTML5, CSS3 and JavaScript/Angular.js for the front-end design. There were many front-end UI frameworks used, but the primary was Angular Material Design. In the final implementation, the UI will also be customized when the user tries to customize their back-end, however, this is not currently a feature.

This UI will be implemented for the internal Couchbase social network and will be used for a demo of the social network template, however, through the beauty of the design of the REST APIs, anyone who decides to create their own social network using this template will be free to change the UX/UI as they please. This is possible because the RESTful nature of communication between back-end and front-end using Angular and Node allows the backend to remain independent. This means that the front-end can be completely different while maintaining the same data on the back-end.

Architecture

This social network platform was built using a complete JavaScript stack, incorporating Node, Express, as well as Angular. Node was used to query the database using N1QL queries, and functions in Couchbase's Node SDK (2.0.11). Express was implemented for its wide range of simple and helpful server-side, JavaScript functions that were especially helpful for handling API requests and responses. Angular was implemented to relay data, primarily JSON object, from the front-end, through a series of REST APIs, to the back-end. In the case of image upload, an HTML post request was used to pass information to the back-end using a node module called Multer.

Angular was also used heavily for form verification to make sure that the app was always receiving proper and relevant data for its queries. However, this is not considered extremely safe, and back-end validation was used as well. This is key to the application, as it is largely API based and therefore cannot rely on proper information consistently being passed to the front-end. Error handling is a large part of this project, so as to ensure that the server it is running on is not at risk of crashing.

Assumptions/Limitations

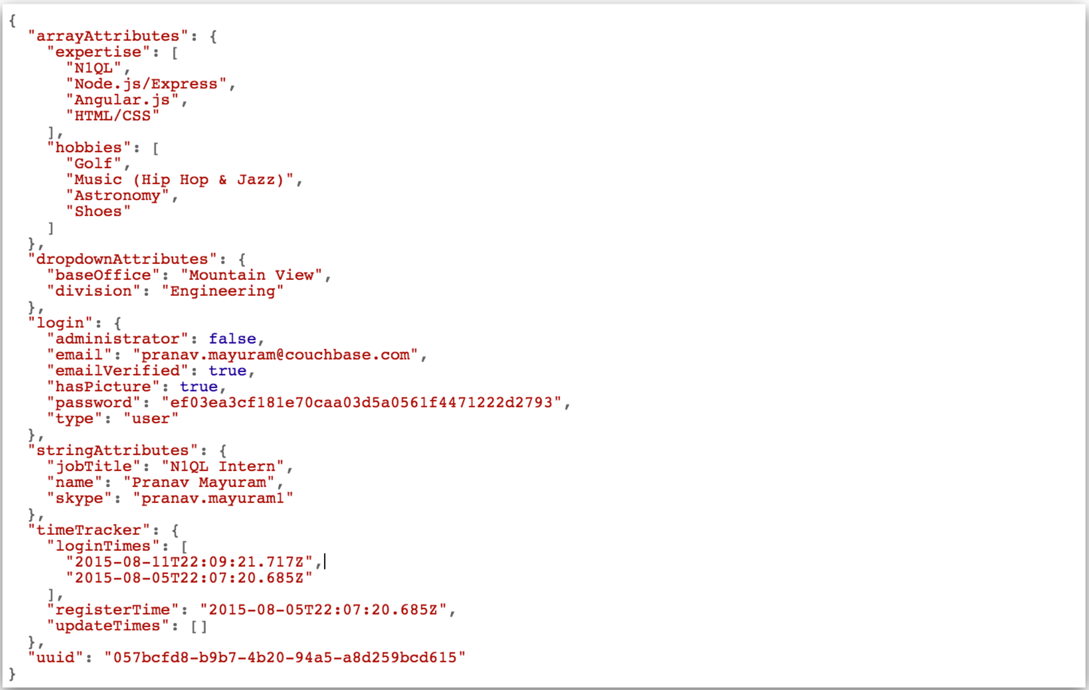
This social network template was built with certain limitations and is not capable of handling highly secure information, as parts of Couchbase's statements (INSERT, UPDATE, DELETE) are still experimental and could potentially alter sensitive information.

The method by which user authentication currently works is via session models, which require a sessionID from the front-end for authentication. In this case, because session models rely on expiry, they could be considered not to be entirely RESTful, which could be seen as a limitation of the application.

DATA

User Document

In this project, the data is designed to be focused around a “User” object. This object will be generated by a Registration form where the users will input data about themselves to generate a usable account. The data that they input about themselves will be saved as JSON objects. There will be optional fields, but certain fields (Name, Email, Password) will be required to generate an account. The user documents will be stored in a data bucket called “users”, and their document IDs will be “Universally Unique Identifiers” (UUIDs). Many different kinds of information could be stored in the document, including arrays (interests of the user), emails, booleans (check for administrators), numbers, etc. The user documents will be stored using a N1QL insert statement, which will insert the JSON object containing all the user data that was input in the online form. There will also be a second bucket called “users_pictures” which will store the profile pictures for each user. Their connection will be discussed later.



```
{
  "arrayAttributes": {
    "expertise": [
      "N1QL",
      "Node.js/Express",
      "Angular.js",
      "HTML/CSS"
    ],
    "hobbies": [
      "Golf",
      "Music (Hip Hop & Jazz)",
      "Astronomy",
      "Shoes"
    ]
  },
  "dropdownAttributes": {
    "baseOffice": "Mountain View",
    "division": "Engineering"
  },
  "login": {
    "administrator": false,
    "email": "pranav.mayuram@couchbase.com",
    "emailVerified": true,
    "hasPicture": true,
    "password": "ef03ea3cf181e70caa03d5a0561f4471222d2793",
    "type": "user"
  },
  "stringAttributes": {
    "jobTitle": "N1QL Intern",
    "name": "Pranav Mayuram",
    "skype": "pranav.mayuram1"
  },
  "timeTracker": {
    "loginTimes": [
      "2015-08-11T22:09:21.717Z",
      "2015-08-05T22:07:20.685Z"
    ],
    "registerTime": "2015-08-05T22:07:20.685Z",
    "updateTimes": []
  },
  "uuid": "057bcfd8-b9b7-4b20-94a5-a8d259bcd615"
}
```

An example of a User document. Its UUID is 057bcfd8-b9b7-4b20-94a5-a8d259bcd615.

The password is secured using hashing through a certain Node.js middleware called Forge. This prevents any security risk for the user information through the obtainment of passwords. Though not developed with the intention of storing highly sensitive information, this will prevent passwords from being easily obtainable through basic hacker attacks, like rainbow table hacks.

Profile Pictures

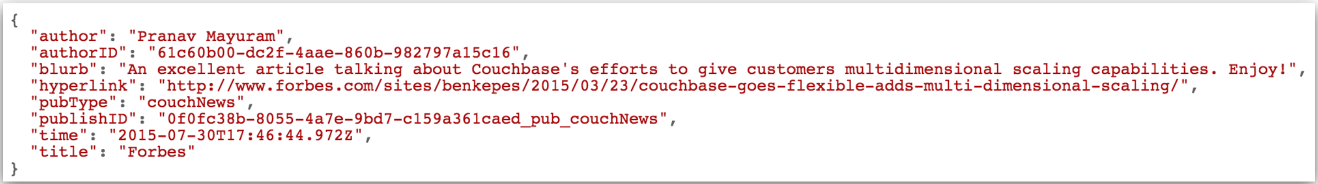
Using the document ID here, the picture stored by the user could be tracked as well. The profile picture would first be uploaded through the front-end using an HTML POST request and temporarily stored in an 'uploads' folder, where it would then be read by the Node.js 'fs' library. Using a node module called 'graphicsMagick', this image would then be altered, and then converted to a base64 string to be stored in Couchbase as binary. Then the UUID generated in the user document would be used as the key for the binary data by simply appending '_pic' to the user document ID and setting that as the document ID for the binary data.

Though this is not considered the quickest method of image storage (many would vouch that storing images in a folder, and storing the metadata and file-path in the user document would be better), it does display how Couchbase could be used to store and output binary data. This provides an advantage from a usability standpoint for developers using the social network template, because if a user chose to delete their profile, or to update a profile image, they would simply be able to do it all from one location, rather than having to then access the file-path and attempt to delete the specific file. It also means that a user would only have to allocate RAM/storage based on what is in their Couchbase buckets, and not a folder or file with file uploads. This ease of use is important in displaying the potential feasibility of storing multiple kinds of information in Couchbase.

To output the data, it could be accessed using a simple 'GET' from the Couchbase Node.js SDK, and the base64 string could be put in the HTML 'img' tag as the 'src'. This is the simplest implementation of this feature and allows the image processing to be rather painless.

Posts

The concept with posts is to allow users to share information with one another beyond just personal information. In the current implementation, this is done by creating a document in a third bucket called users_publishments. Each document contains the userID of the user who created the document under the attribute 'authorID', as well as a unique document ID, and a 'pubType' which outlines the certain type of post that is being used, such that if there were multiple types of posts, they could be searched and displayed differently. In the implementation used for Couchbase's internal social network, the two types are 'Github Projects' and 'Couchbase in the News'. This is then displayed separately in two different tabs of the website, but is shows all together under 'My Posts' within the My Profile page. This kind of flexibility comes with the use of the 'pubType' attribute. The information for who created the post comes from the user authentication that occurs on all protected routes.



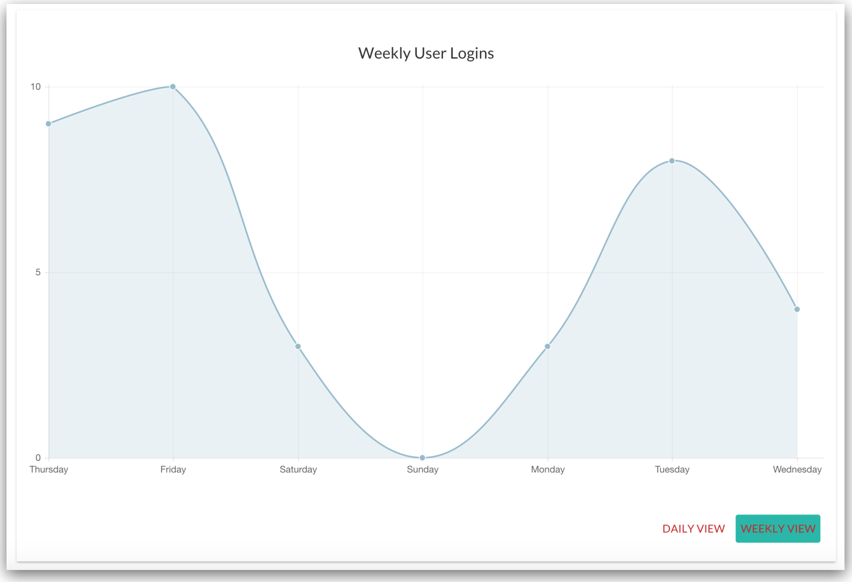
```
{
  "author": "Pranav Mayuram",
  "authorID": "61c60b00-dc2f-4aae-860b-982797a15c16",
  "blurb": "An excellent article talking about Couchbase's efforts to give customers multidimensional scaling capabilities. Enjoy!",
  "hyperlink": "http://www.forbes.com/sites/benkepes/2015/03/23/couchbase-goes-flexible-adds-multi-dimensional-scaling/",
  "pubType": "couchNews",
  "publishID": "0f0fc38b-8055-4a7e-9bd7-c159a361caed_pub_couchNews",
  "time": "2015-07-30T17:46:44.972Z",
  "title": "Forbes"
}
```

An example of a 'Couchbase in the News' post. It follows a simple data model with no nested objects.

Statistics

Finally, there could also be many statistics collected about each user, so that querying Couchbase can be shown in many different ways. The time at which each login takes place could be stored in an array so as to track each user's usage of the application. This is a tool that many websites, such as Gmail and Facebook, use to show users when their account may be in jeopardy. If a login is shown at a time that the user knows that they were not online, they could be alerted, and make sure that they change their password. This same statistic could be used for a network administrator or developer to track the site traffic.

Another statistic that could be tracked is the geographic location of each user. This would be interesting to show where users of the app are currently distributed across the globe, and the browser always asks for permission to collect this data, so it will not be a problem for users to keep their location private if they choose. It will be up to the developers who use this social network template to decide what information they would like to track, or not track, based on how they later customize the product. All of this information should also be kept in the user document, so as to minimize time spent searching for the data when it is queries later on. This is not currently in use in the current implementation, but simply another option for developers who choose to customize Touchbase.



This is an example of how statistics are currently displayed in this version of Touchbase. This takes the data in the 'timeTracker.loginTimes' array and finds counts of logins for each day/hour (weekly/daily view) using N1QL date functions. This graph uses an Angular implementation of the 'Chart.js' library to display the graph.

FUNCTIONALITY

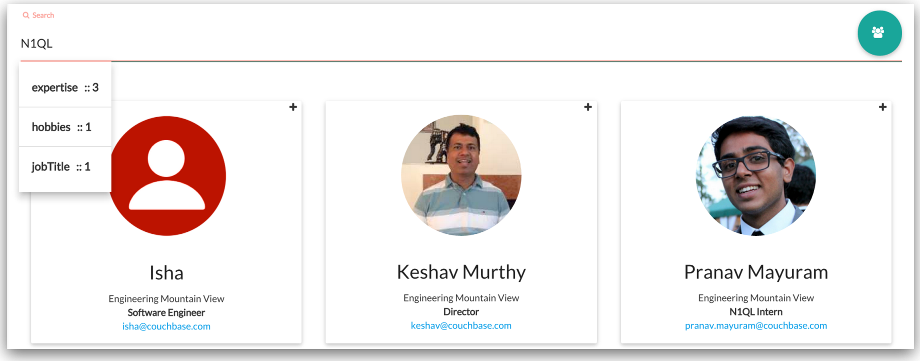
The main functionality of the social network template is to be able to have a cleanly designed backend which can easily create and access user profiles, via a user document and a user profile picture. Beyond this, it will have the ability to search different attributes of each user using a custom search feature. The idea is that this will make things simple for anyone who develops upon it to be able to develop more complex features and well-designed UIs instead of focusing on the basic functionality they are hoping to get from the social network already.

As mentioned earlier, it will also be possible to track many statistics of each user, and this template will give examples of different N1QL queries that can be used to manipulate the data as the developer sees fit. The queries will help to give an idea of what kind of data could be obtained, and its significance to the developer will shape how they create their version of the product.

The basic functionality of developing an account, registering, and also logging into one's account will all be taken care of by the template. Due to the speed and reliability of Couchbase, it will be simple to maintain all of this information. Using the UPSERT function in Couchbase, it will also be simple to edit one's user profile. This way, if there is some information that a user has not yet input, and wants to include, it will be added. Also, if there is any information they want to update, the UPSERT function will also take care of that.

One of the more difficult parts of building an account is to store and access a profile picture. For one to store this in a database, it must be stored as a base64 string or as binary, and this will be taken care of by the template. This often provides a significant challenge for people who want to build a website that stores images, so this will add simplicity, and again allow for the advancement of one's social network by spending less time on such tasks.

Another aspect of the functionality will be the fact that the database of the users will be searchable by attribute. Usres will have the ability to enter a search field in a search box, and get results related to what they are looking for. For example, if they are searching for 'N1QL', they would most probably find the highest number of results in expertise, and they could click to see the corresponding results.



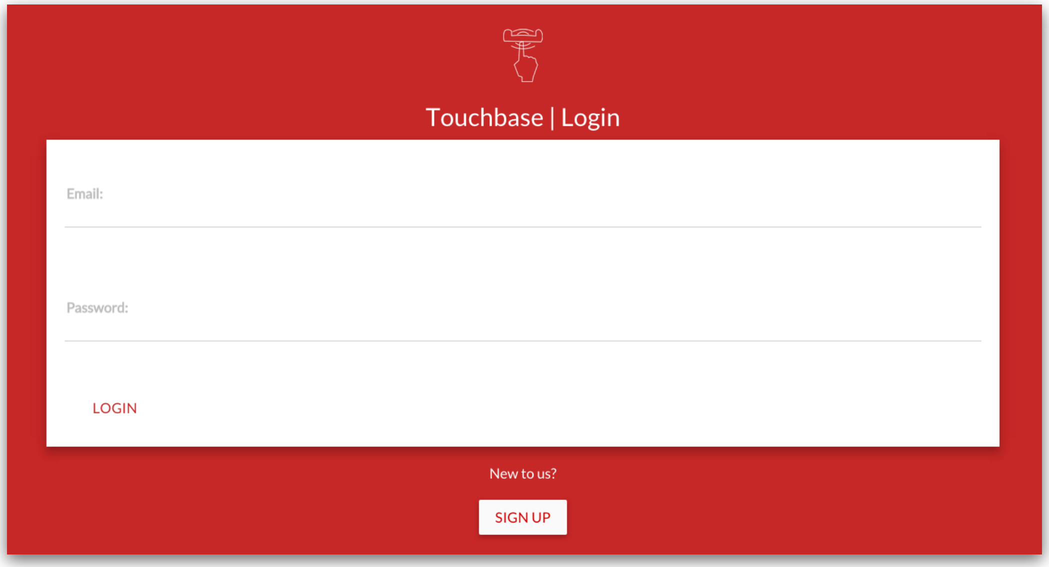
This is an example of how an 'Intelligent Search' might look and be executed in this social network platform.

USER EXPERIENCE

The User Experience applies only to the example application built with this social network platform, and it will be provided for users to implement as they create their social network, however, it is not a core part of the functionality of the application. The User Experience is focused around a clean layout that will help to display all of the core functionality that the social network template provides, as well as some examples of additional features that could be added.

Login and Register

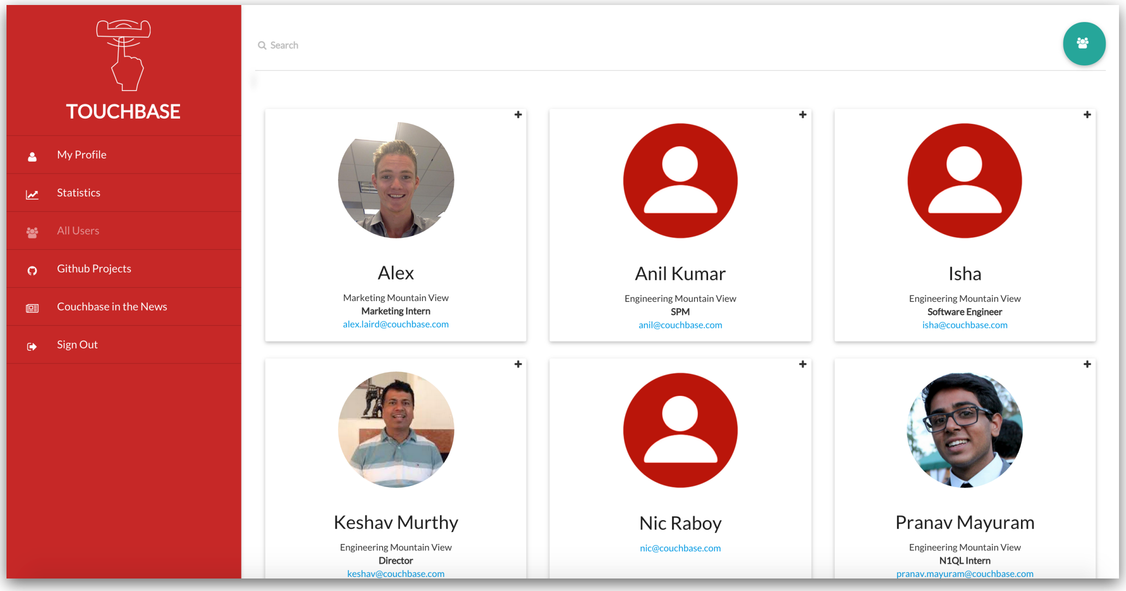
The main, critical screens will be the Login and Register areas where users can create and access their accounts. These will be built so as to make the process simple for any developers who want to alter these form fields. This part of the UI is important because anyone building their own UI or social network should see how the form data for Registration or Login is expected to be handled. The login page is typically part of the home screen, and the same will apply here. Users will click on the home screen to be registered if they still need to create an account, and then they will be redirected to a registration page.



This is an example of what the user login page may look like. It maintains a color scheme outlined by Angular Material Design, and its $mdThemingProvider function. The idea is to keep this consistent throughout the application, and provide a coherent look.

User Pages

The main idea will be to showcase how the directory would work, having a region of the website in which all the users can be seen, and they can be searched for their many attributes, effectively creating a filter on this directory. There will also be different regions for different kinds of posts. In this example, we see 'Github' and 'Couchbase in the News' posts. These also follow the card-based design which can be seen throughout the UI.



Alex
Marketing Mountain View
Marketing Intern
alex.laird@couchbase.com

Anil Kumar
Engineering Mountain View
SPM
anil@couchbase.com

Isha
Engineering Mountain View
Software Engineer
isha@couchbase.com

Keshav Murthy
Engineering Mountain View
Director
keshav@couchbase.com

Nic Raboy
nic@couchbase.com

Pranav Mayuram
Engineering Mountain View
N1QL Intern
pranav.mayuram@couchbase.com

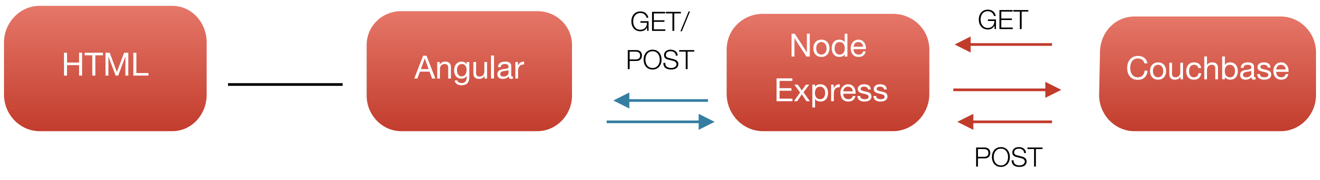
This graphic shows a basic example of what the UI for Touchbase looks like for the page where one can see all the people in the directory. There will also be the side Navigation bar where users can access other parts of the website, as shown here.

ARCHITECTURE

This app is built using REST API calls between Angular and Node/Express. REST documentation specifies that the system must be “stateless” meaning that the state of the front-end cannot change any action of the back-end. The back-end (Node/Express with Couchbase) is acting on requests made from the front-end. This may require certain data/parameters from the front-end, or it may not.

Request Types

The main types of requests made by the front-end in this case are POST and GET requests. Though these are only syntactic sugar, they are a convention that should be followed, especially if others are to use these APIs. The POST is used to insert or change information in the database, and could also retrieve certain information to give back to the front-end for display, or other use. A GET request means that the front-end is only operating to receive information, not to add or change anything in the database. If the request to the back-end requires certain information from the front-end his can be done using parameters in the GET request to give certain information to the back-end so that it retrieves pertinent information. For example, in the case of a login attempt, the front-end may pass the email address and password as parameters so that the back-end can check if that is the correct combination of user information.

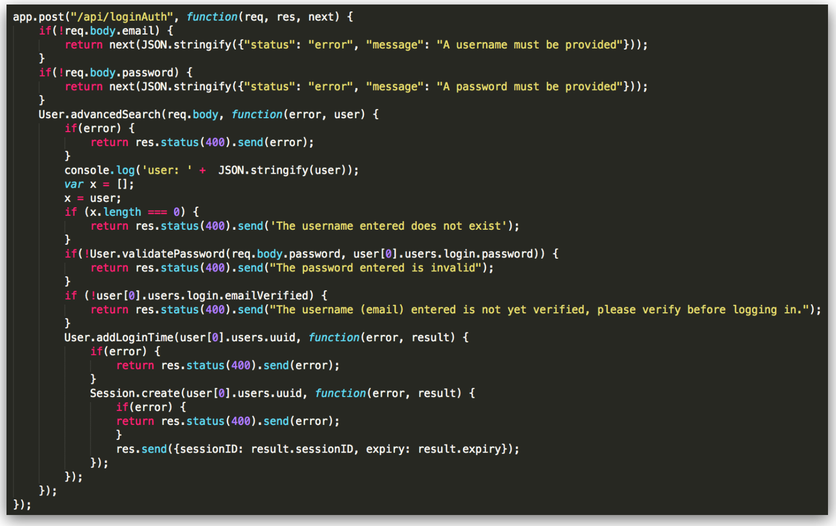


The primary difference between GET & POST can be seen at the red arrows where GET simply retrieves information, whereas POST has a channel to both add/alter information and retrieve information.

One important portion of the architecture are the queries that it is written with. In this current implementation, Angular sends ‘$http’ requests to the API endpoints, and Express handles the data for these requests in a simple object, within the req.data (for POST), or req.query (for GET). Node/Express are then used together to access the data and then alter the data as necessary, and then send N1QL queries with INSERT, SELECT, UPSERT, UPDATE, or DELETE to Couchbase server using the Node SDK ‘bucket.query’ command. The information is then sent back using the Express ‘res.send’ or ‘res.json’ command for objects.

Examples

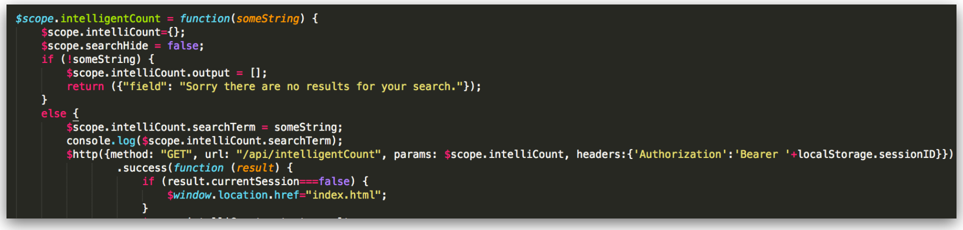
Some examples of this might be that the ‘/api/loginAuth’ POST route takes the login and password info from the front-end, checks the information, appends the necessary data to the user’s timeTracker.loginTimes array, and then sends back the sessionID that is generated for the user’s current login session. On the other hand, the ‘/api/intelligentCount’ GET route will take the search string, check the results in each field, and send back the number of occurrences the search string has in each category. No data is changed, so this can be classified as a GET request.



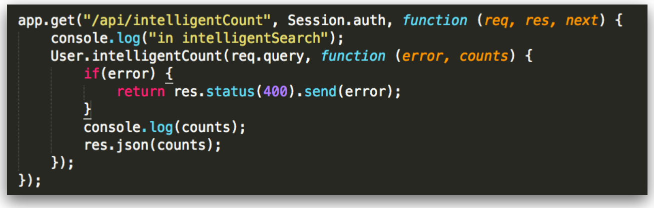
```
app.post("/api/loginAuth", function(req, res, next) {
    if(!req.body.email) {
        return next(JSON.stringify({"status": "error", "message": "A username must be provided"}));
    }
    if(!req.body.password) {
        return next(JSON.stringify({"status": "error", "message": "A password must be provided"}));
    }
    User.advancedSearch(req.body, function(error, user) {
        if(error) {
            return res.status(400).send(error);
        }
        console.log('user: ' +  JSON.stringify(user));
        var x = [];
        x = user;
        if (x.length === 0) {
            return res.status(400).send('The username entered does not exist');
        }
        if(!User.validatePassword(req.body.password, user[0].users.login.password)) {
            return res.status(400).send("The password entered is invalid");
        }
        if (!user[0].users.login.emailVerified) {
            return res.status(400).send("The username (email) entered is not yet verified, please verify before logging in.");
        }
        User.addLoginTime(user[0].users.uuid, function(error, result) {
            if(error) {
                return res.status(400).send(error);
            }
            Session.create(user[0].users.uuid, function(error, result) {
                if(error) {
                return res.status(400).send(error);
                }
                res.send({sessionID: result.sessionID, expiry: result.expiry});
            });
        });
    });
});
```

This is the '/api/loginAuth' API endpoint in the routes.js file which manages all endpoints. This is an example of a function in node, using a N1QL query and a function from the Express library, to check if a login's credentials are correct, and then add the current time to a 'loginTimes' array, then create a session and return its ID.

These kinds of requests must match up with the same requests being made from the front-end. For example, if a request in Angular was a GET request, the same kind of request must be happening at the same URI in the back-end (Node). If this condition is not fulfilled, the request will not go through.



```
$scope.intelligentCount = function(someString) {
    $scope.intelliCount={};
    $scope.searchHide = false;
    if (!someString) {
        $scope.intelliCount.output = [];
        return ({"field": "Sorry there are no results for your search."});
    }
    else {
        $scope.intelliCount.searchTerm = someString;
        console.log($scope.intelliCount.searchTerm);
        $http({method: "GET", url: "/api/intelligentCount", params: $scope.intelliCount, headers:{'Authorization':'Bearer '+localStorage.sessionID}})
                .success(function (result) {
                    if (result.currentSession===false) {
                        $window.location.href="index.html";
                    }
```

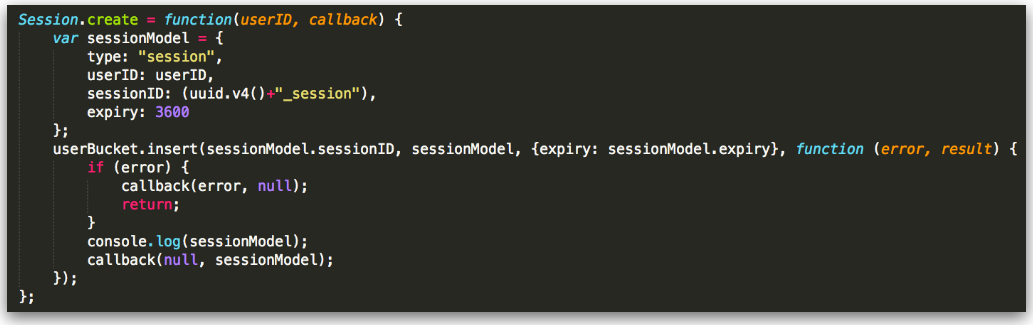


```
app.get("/api/intelligentCount", Session.auth, function (req, res, next) {
    console.log("in intelligentSearch");
    User.intelligentCount(req.query, function (error, counts) {
        if(error) {
            return res.status(400).send(error);
        }
        console.log(counts);
        res.json(counts);
    });
});
```

An example of a function in an Angular.js (TOP) file which sends a GET request to Node using the http GET method. This would send information through a REST API to the code snippet (BOTTOM) of the page, which is a GET request to the server in Node/Express.

SECURITY

The current method for keeping Touchbase secure is using session models, which are generated every time a user logs in. If one was to look at the '/api/loginAuth' API endpoint (this endpoint can be seen in the Architecture session), at the very end of the route, a function called 'Session.create' is called to generate a user session. This contains the user ID for the person who generates the login session. This session expires after one hour, and is created to make sure that a user continually authenticates themselves, and does not leave their account exposed for an extended amount of time to people who are not users of the network.



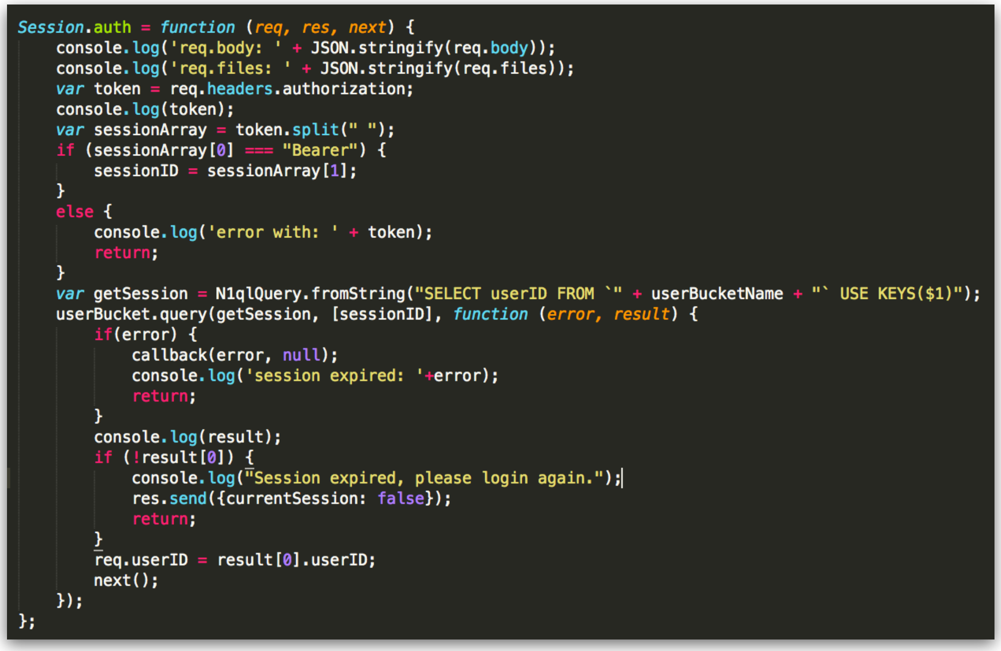
```
Session.create = function(userID, callback) {
    var sessionModel = {
        type: "session",
        userID: userID,
        sessionID: (uuid.v4()+"_session"),
        expiry: 3600
    };
    userBucket.insert(sessionModel.sessionID, sessionModel, {expiry: sessionModel.expiry}, function (error, result) {
        if (error) {
            callback(error, null);
            return;
        }
        console.log(sessionModel);
        callback(null, sessionModel);
    });
};
```

This is the Session.create function, which generates a session model. This session model is passed back to the front-end where it is then stored in some form of a cookie or localStorage where it can be sent with requests.

The function will be used to send back the session model in the 'res.json' object using Express. This session model's session ID should be stored on the front-end so that it can be used on the http request headers, or somehow be sent to all future protected routes. In the current front-end, Angular.js implementation, the request header contains the session ID and is passed to all protected routes. This implementation can be seen on the following page.

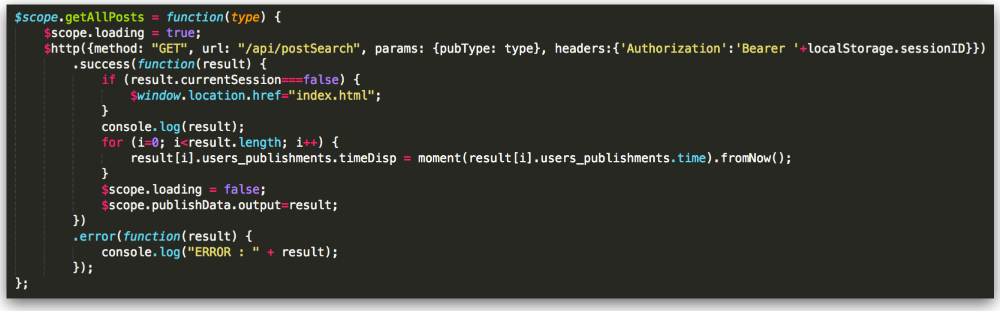
A protected route is for actions that should only be performed by users of the website, and the function parameters of the protected API endpoints all posses a Session.auth function evaluation. When this is evaluated, it ensures that the session ID of the session stored on the front-end for that user still exists.

If it is not stored, some front-end action will be performed to potentially re-route the application to the login page (current implementation), or some other action. Regardless of the front-end, the route will not return any critical data to the unauthorized user. An example of the Session.auth function being used can be seen in the architecture '/api/intelligentCount' endpoint. The function itself is pasted below, so that one can better understand its function in the scheme of the application.



```javascript
Session.auth = function (req, res, next) {
    console.log('req.body: ' + JSON.stringify(req.body));
    console.log('req.files: ' + JSON.stringify(req.files));
    var token = req.headers.authorization;
    console.log(token);
    var sessionArray = token.split(" ");
    if (sessionArray[0] === "Bearer") {
        sessionID = sessionArray[1];
    }
    else {
        console.log('error with: ' + token);
        return;
    }
    var getSession = N1qlQuery.fromString("SELECT userID FROM `" + userBucketName + "` USE KEYS($1)");
    userBucket.query(getSession, [sessionID], function (error, result) {
        if(error) {
            callback(error, null);
            console.log('session expired: '+error);
            return;
        }
        console.log(result);
        if (!result[0]) {
            console.log("Session expired, please login again.");
            res.send({currentSession: false});
            return;
        }
        req.userID = result[0].userID;
        next();
    });
};
```

This is the Session.auth function which is evaluated on all protected routes, and ensures that the session currently exists, and throws an error if not. If it succeeds, it will add the userID of the current session to the 'req' object so that it can be accessed to view personal information in 'My Profile', or in other areas of the app like posts.



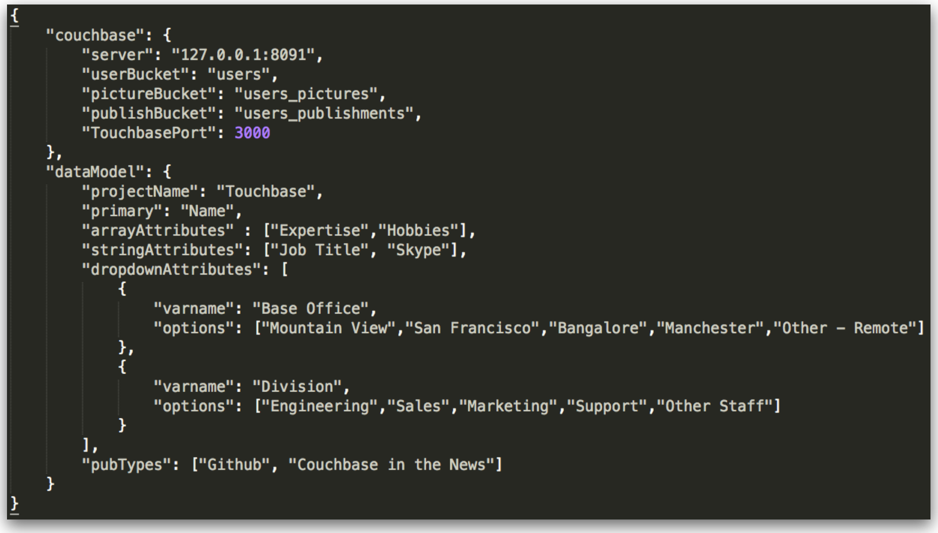
```javascript
$scope.getAllPosts = function(type) {
    $scope.loading = true;
    $http({method: "GET", url: "/api/postSearch", params: {pubType: type}, headers:{'Authorization':'Bearer '+localStorage.sessionID}})
        .success(function(result) {
            if (result.currentSession===false) {
                $window.location.href="index.html";
            }
            console.log(result);
            for (i=0; i<result.length; i++) {
                result[i].users_publishments.timeDisp = moment(result[i].users_publishments.time).fromNow();
            }
            $scope.loading = false;
            $scope.publishData.output=result;
        })
        .error(function(result) {
            console.log("ERROR : " + result);
        });
};
```

This is an example of how the Angular.js '$http' request is sent with a session ID header and the action it performs if 'currentSession===false'

Currently, on the front-end, if the Session.auth function does not evaluate to an authorized user, it will return a JSON object of '{currentSession: false}', and the front-end '$http' request will perform some action if currentSession is false, as is described above.

CUSTOMIZATION

Touchbase focuses on having a data model that could be easily repurposed across the front-end and back-end of the application. On the back-end, the application has a 'config.json' file from where a JSON object with basic attributes of the application is stored. The JSON object is used in the back-end for search, intelligent search, user model creation, and more.



```
{
    "couchbase": {
        "server": "127.0.0.1:8091",
        "userBucket": "users",
        "pictureBucket": "users_pictures",
        "publishBucket": "users_publishments",
        "TouchbasePort": 3000
    },
    "dataModel": {
        "projectName": "Touchbase",
        "primary": "Name",
        "arrayAttributes" : ["Expertise","Hobbies"],
        "stringAttributes": ["Job Title", "Skype"],
        "dropdownAttributes": [
            {
                "varname": "Base Office",
                "options": ["Mountain View","San Francisco","Bangalore","Manchester","Other - Remote"]
            },
            {
                "varname": "Division",
                "options": ["Engineering","Sales","Marketing","Support","Other Staff"]
            }
        ],
        "pubTypes": ["Github", "Couchbase in the News"]
    }
}
```

This is an example of the config.json file being used for the Couchbase internal social network that is based upon Touchbase. The portion of it that changes data will come from the 'dataModel' sub-object.

On the front-end, the JSON object is pulled using Angular's '$http' request and pulls the file and adds it to a '$rootScope' object where the variable can be accessed by any controller in the application. This essentially means that every page will have access to the object, and will show its results accordingly. For example, the 'All Users' page will display the user information based on the customized JSON object, and then creating attributes of the card for each user accordingly. On the same page, this JSON object is used to generate the intelligent search which tells users how often there search term appears in different user attributes. The terms it search for depends entirely on this 'config.json' document. To see an example, look at the 'Functionality' page.

This document also heavily influences the Registration page and changes what user's are asked to include in their user profiles. The data will populate the dropdown menus, the login attributes and more. Through the use of this document, the social network will become more and more flexible, and this will continue to evolve in later iterations of Touchbase.

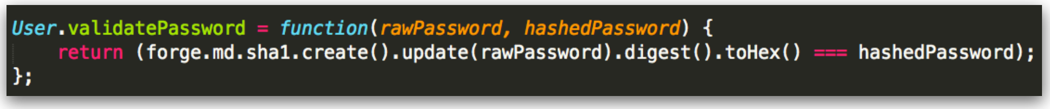
ASSUMPTIONS & LIMITATIONS

In this social network platform, there are certain assumptions and limitations that exist for further development upon this Touchbase platform. The core idea of this social network template is that this is simply created to manage user data that one would be comfortable sharing with others within the network. There are currently no privacy settings in which a user could hide their information from certain other users, or make it such that certain aspects of their profile were not visible to everyone. In principle, this social network platform is intended for use only amongst one's peers who they are comfortable sharing information with. The underlying assumption here is that if there is any information a particular user is not comfortable sharing, then they would not like to share it with anyone in the particular social network they are part of. There is no subset of people within the group that will have more or less access to any one user's information.

Another important limitation that is currently in place is that the information is not highly secure. The middleware "Node-Forge" is currently implemented to secure passwords by hashing them and storing them in an encoded fashion. The way a user login works is by checking the hashed password that is stored in the user document against a hashed version of the password entered for login. This way the two passwords are never compared as raw strings, only in their encoded version.

There is an underlying assumption that "Node-Forge" has a method of hashing that is not widely known, nor easily accessible by any hacker who wanted to get passwords. Though "Node-Forge" has a reputation for being secure, since the security technology is not directly handled by Touchbase, it is an assumption that the passwords will remain secure. Another issue with the current implementation of this, is that the password can occasionally be passed to the front-end using 'SELECT * FROM' in a query, and though it is hashed, some hackers could access it. This is an improvement for the near future.

Another assumption that this implies is that no developer will change the storage of the user document to store the actual password strings, and will instead continue to use the hashed version that uses "Node-Forge".



```
User.validatePassword = function(rawPassword, hashedPassword) {
    return (forge.md.sha1.create().update(rawPassword).digest().toHex() === hashedPassword);
};
```

This is the function that uses the 'Node-Forge' middleware to hash the password entered upon login, and check it against the password used in the original registration.

Finally, certain aspects of Couchbase Server 4.0 that are currently 'experimental' are included in this project, such as the INSERT, UPDATE, UPSERT and DELETE statements. These are being improved upon, and made more reliable by Couchbase currently, and this is not expected to be a limitation in the future. However, currently, since it is not endorsed as a fully functional feature of the Couchbase server, it is not considered 100% safe to use, and thus serves as a limitation to the reliability of the social network platform, Touchbase.

INDEX OF REST APIS

The set of REST APIs developed for this application are all run using Node JS and can be found in the “routes.js” file within the “routes” folder. Each one is comprised of one or more functions defined in different “models” files. All of these can be found within the “models” folder, and they are used to develop a robust library of functions that can be used within the API routes. Within this section of the document, each API will be explained in the order that a user may most probably come in contact with them, and each URI will be included so that one can find the particular code snippet quickly.

Registration APIs

- URL: **“/api/emailSearch”** type: **“GET”**

The emailSearch API is different from the Advanced Search (search that is used for all other user searched), primarily in one unique way. The emailSearch API is unprotected, this is explained more in the explanation of the ‘/api/loginAuth’ API, and with the unprotected route, it finds all user emails and makes sure that they are not taken when a user is signing up. This is so that an error can be thrown before the user tries to login.

- URL: **“/api/registerUser”** type: **“POST”**

This API route is used to register a user. There will be information entered into a “formData” type JSON object on the front-end and this object will be passed to the back-end. The current implementation used for testing uses Angular.js and sends the object using a “$http” request. The API contains strong form validation which will ensure that the necessary information is included, and that the passwords are relatively secure. The validation also ensures that there is not already an account made with the email that is submitted. In the future, the goal is to create some form of email validation such that one will receive an email that they will have to click to confirm that they have created an account on Touchbase. The back-end processes this information into a user document-model which is defined in the “usermodel.js” file within the “models” folder. The user document is then inserted into the “users” bucket, but has the attribute ‘login.emailVerified’ set to a default of false. The route then sends an email verification using the Sendgrid web API and inserts a verification document into the ‘users’ bucket . This email has a button that has a link to the ‘/api/verify/:verificationID’ endpoint, with the documentID of the verification document embedded in the email’s button, which allows users to verify their account.

- URL: **“/api/verify/:verificationID”** type: **“GET”**

This route is used to verify the email addresses and make sure that people are truly using the email addresses they claim. This prevents people from creating an array of fake accounts, or trying to impersonate someone else. This API uses the verificationID parameter, which is embedded in the link from the email, and then searches for that document, finds the userID in the verification document, and then change’s that user’s ‘login.emailVerified’ attribute to true. This allows the user to access their account, update their profile, view other users’ profile data and upload a profile picture.

URL: **“/api/uploadAttempt”** type: **“POST”**

IMPORTANT: Graphics Magick must be installed prior to the use of this image upload API. One must install Graphics Magic using “brew install graphicsmagick” (Mac) or some other form on other operating systems.

After the user registers successfully, they are directed to a different page where they can upload a profile picture. This API route assumes that the image is uploaded using an HTML post request, which is then put into an “uploads” folder within the root project directory using node middleware called Multer. Multer will automatically upload the image to the “uploads” folder with the proper setup, so the process afterwards is to validate the file and make sure that it is uploaded properly to Couchbase. In the current implementation of the file upload, Multer sends cookie data with the HTML post request, so that the sessionID can be passed to a function. This function then finds the according userID that goes along with the sessionID that was passed. This userID is then added to the request body and is passed to the function that will truly upload the image.

To make sure that the process does not begin until the file written to the “uploads” folder, the “Picture.attempt” function waits until a boolean variable called “done” becomes true. The route first verifies that the image is not larger than 7.5 MB and that the file is of a valid image format. If it fails any of these conditions, an error is thrown, and the file is deleted from ‘uploads’ using node’s ‘fs’ library. One important feature is that this uses UPSERT so that if a user wants to edit their profile picture later, no extra routes need to be created. The current implementation also passes cropping data to the image in the form of a JSON object through Multer; this cropping data is created using ‘ng-cropper’ on the front-end. Then another node middleware called Graphics Magick (gm) is used to crop the image using the crop data and the file-path (from Multer) on the back-end. Graphics Magick is also used to downsize the image, and scale it such that it will be of a smaller size, and could be mroe easily rendered on the front-end. The app then uses the ‘fs’ library from node to read the image using the same file-path and convert the image binary to base64 and then uploads it to Couchbase with its document ID being “[passedUserID]_pic”. The image is finally deleted using the ‘fs’ library, and shows that the “uploads” folder is effectively used as a temporary cache. Finally, a query is also sent to the users bucket to update the user’s document to update the boolean attribute ‘hasPicture’ to true, from its default false.

Login APIs

- URL: **“/api/loginAuth”** type: **“GET”**

The loginAuth API is used for login verification, and to create session models. It accepts a JSON object with a user email and password. First, it checks that the necessary fields are passed. It then checks to find the user document that aligns with the user email that was passed. This is done using the ‘advancedSearch’ function which is defined in the ‘usermodel.js’ file and is built to return the user documents that occur on the search of any attribute of a user. It then verifies if the hashed passwords are equal, and if so, it will allow the user to access the website’s contents by creating a session model, which can be viewed in the ‘sessionmodel.js’ file. This model includes the time made, an expiry, a sessionID (random document ID for the session), and the userID. The sessionID is then returned and passed back to the front-end. In the current implementation, this data is stored in the browser's localStorage, so that it can be used for later authentication using authorization headers for protected API requests. The session model is given an expiry, so that it will be deleted after one hour in the database, and then a user will have to login again to create a fresh sessionID. This is common practice with session models, and is secure since it does not rely on front-end cookies to expire, as those could be easily manipulated by any user. For all protected routes, a 'Session.auth' function will be used as a parameter that will check to make sure that the route is only being used by a user that has an active session. If not, the user will be redirected to the login page to login again.

Publication APIs

- URL: **"/api/publishPost"** type: **"POST"**

The publishPost API is used to take basic data for a post from a user and insert the document into the database. The main elements of the post are simply a hyperlink if there is a specific link which the user would like to post about, and then a blurb to discuss the information they are posting. The information used by the back-end comes partially from the Javascript cookie as well, since a user must be logged in to create a post. This cookie will be used to track who the author of the post is, so that it can be included in the post's information. Another attribute that is stored is the type of post that is made. For example, one section of the social network could be for users to post their Github projects, and another could be for news articles. The type of publication, 'pubType', could be defined on the front-end and then passed to the back-end so that the information can be separated from one another.

- URL: **"/api/postSearch"** type: **"GET"**

The postSearch API is used to search through different posts, by user, etc. This is not implemented on the search that occurs within the search page (that is a front-end Angular filter), however it is used to get all the posts, for the different post pages. For example, the current implementation, it is used for three different things: Github posts, Couchbase in the News, and My Profile. For the first two, Github and Couchbase News, it searches with a 'pubType' filter, where it will only display Github posts, or Couchbase in the News posts depending on the tab one is on. This 'pubType' is specified on the front-end $http request from Angular. In the final example, for My Profile, the search specifies a user ID which comes from the Session.auth function, so it will find all the posts that the current user has, regardless of type. The posts all return with their critical display information, and the 'moment.js' library is used to show how long ago a post was made in a simple, readable format.

- URL: **"/api/deletePost"** type: **"DELETE"**

The deletePost API is only accessible to users from their 'My Profile' page, where they can delete their own posts if they choose to. The API is very simple, and allows users to delete their posts if they please, by simply passing the post's document ID to the back-end, and deleting it. In the front-end, it may be ideal to ask the user with an alert, or some other dialog box to see if they would want to delete their post, since it is not information that could be retrieved at a later point.

Search APIs

- URL: **"/api/intelligentCount"** type: **"GET"**

The intelligentCount API focuses on taking a search string, and searching different user attributes for it. It checks to find how many counts of each type of attribute show up for the search string. For example, if one was to search N1QL, it might display 'Expertise :: 3 | JobTitle :: 1", etc. It uses a series of N1QL queries which are customized to each type of user attribute (stringAttributes, dropdownAttributes, arrayAttributes), and then returns the count accordingly. To see an example of how this may look on a UI, refer to the 'Functionality' section. This is based on the 'LIKE' predicate primarily, except for dropdownAttributes, which will be used as filters in a later iteration of the product.

- URL: **"/api/advancedSearch"** type: **"GET"**

This API is used for almost all searches done in the application. It takes certain parameters, and searches for their results exactly. For example, if one was to search for 'N1QL' and then select 'Expertise' as the correct field in '/api/intelligentCount', then the '/api/advancedSearch' endpoint would be used to find the according user documents. The API also gets the base64 data for each image of each user, or applies the stock image, all by way of the 'Picture.receive' function in 'picturemodel.js'. This API is probably the most heavily used in the application, and is used to receive personal information in the 'My Profile' page, as well as to get the data of the users/search results in 'All Users'.

User Update APIs

- URL: **"/api/updateUser"** type: **"POST"**

The updateUser API is used to change any user text information. The idea is that the user information will all be stored on the front-end in some form, and then the data model could be updated on the front-end, and the entire document could be sent to the back-end to be update the model. This method is rather weak, and needs improvement, as it relies on the front-end to have all the necessary information. It also should not use UPSERT, as that will insert a document regardless of whether or not the document currently exists. This API is currently considered **EXPERIMENTAL**. If one wanted to update their image, they can simply be redirected to an image upload page where they will access the '/api/uploadAttempt' route since that uses UPSERT.

Statistics APIs

- URL: **"/api/graphData"** type: **"GET"**

The graphData API uses N1QL date functions, UNNEST, and GROUP BY to organize the count of logins by hour or day using the 'timeTracker.loginTimes' array that each user has. This allows the content to be grouped by the number of hours since a certain login has occurred, and then using the aggregate COUNT function, the data is generated. The route requires some sort of time span to look at (either day or hour), so if 'hour' is specified, it will return a 'daily view' for the past 24 hours, including the x-axis of hours for the past 24 hours (generated with an array and the 'moment.js' library), and a y-axis of the actual login counts. Example display in the 'Data' section.

REFERENCES

Software Design Document Guidelines. (2007, May 1). Retrieved June 30, 2015, from http://www.atilim.edu.tr/~dmishra/se112/sdd_template.pdf

How to Write an Effective Design Document. (2007, May 1). Retrieved June 30, 2015, from http://blog.slickedit.com/2007/05/how-to-write-an-effective-design-document/

Forge: JavaScript Security and Cryptography. (2010, July 1). Retrieved June 30, 2015, from http://digitalbazaar.com/forge/

Couch411 Social Network Template (2015 June 25). Retrieved June 30, 2015, from https://github.com/pranavmayuram/Couch411

Mayuram, P. (2015, August 14). Couchbaselabs/touchbase. Retrieved August 14, 2015, from https://github.com/couchbaselabs/touchbase

What Is REST? (2015, May 29). Retrieved August 14, 2015, from http://www.restapitutorial.com/lessons/whatisrest.html

Couchbase N1QL Docs. (2015, June 1). Retrieved August 14, 2015, from http://docs.couchbase.com/4.0/n1ql/

Express - Node.js web application framework. (2015, August 12). Retrieved August 14, 2015, from http://expressjs.com/

AngularJS Docs (2015, August 12). Retrieved August 14, 2015, from https://angularjs.org/

Craig, C. (2015, July 7). Tc-angular-chartjs. Retrieved August 14, 2015, from http://carlcraig.github.io/tc-angular-chartjs/

Heckmann, A. (2015, July 18). GraphicsMagick for node.js. Retrieved August 14, 2015, from http://aheckmann.github.io/gm/

Angular Material. (2015, August 11). Retrieved August 14, 2015, from https://material.angularjs.org/latest/#/