

# B-ACTIVE

by the Active Bruins

KRISHNA BABU, EDDIE HUANG, ERIC KONG,  
RYAN LO, RAMYA SATISH, & NICOLE WONG<sup>1</sup>

## PREAMBLE

This project is being developed under Dr. Miryung Kim's COM SCI 130, in Brett Chalabian's Discussion 1C. Its Github repository may be found at

<https://github.com/ryanlo7/bActive>

The UCLA IDs of each team member are as follows.

Krishna Babu	604460009
Eddie Huang	204607720
Eric Kong	304601223
Ryan Lo	704579791
Ramya Satish	104601436
Nicole Wong	104651105

This report covers Part B, the mid-point checkup, of the project.

## CONTENTS

1	Design	3
1.1	Overview	3
1.2	Design Changes	3
2	Description of API	7
2.1	API Declarations	7
2.2	Information Hiding	10
2.3	API Integration Sequence Diagram	13
3	Testing	14
3.1	Login Page HTTP GET	14
3.2	Login Page HTTP POST (incorrect password)	14
3.3	Login Page HTTP POST (correct password)	15
3.4	HTTP GET Profile Page with Login Verification	15
3.5	HTTP GET Event Page with invalid userId	15
3.6	HTTP GET Event Page with valid userId, without cookie	15
3.7	Unit Test for Availability Match	15
3.8	Unit Test for Activity Match	16
4	User interface	17

5	Contributions	20
5.1	Krishna	20
5.2	Eddie	20
5.3	Eric	20
5.4	Ryan	20
5.5	Ramya	21
5.6	Nicole	21

## LIST OF FIGURES

Figure 1	Use-case diagram	4
Figure 2	Updated class diagram	5
Figure 3	jsdoc Method Table	7
Figure 4	generateActivityMatches description	8
Figure 5	matchUsers description	8
Figure 6	insertUser description	9
Figure 7	searchUsers description	9
Figure 8	updateUser description	9
Figure 9	Back-end designs of core pages implemented without information hiding.	10
Figure 10	Back-end designs of core pages implemented with information hiding.	11
Figure 11	Changeability matrix.	12
Figure 12	Sequence diagram of Register module functionality	13
Figure 13	Registration page	17
Figure 14	Login page	17
Figure 15	Profile page	18
Figure 16	Profile editing page	18
Figure 17	Events page	19

---

<sup>1</sup> Department of Computer Science, University of California, Los Angeles.

# 1 DESIGN

## 1.1 Overview

For Part B, our team has made significant progress on our web application that helps find groups of people to do physical activities with, bActive. We have implemented many parts of our core features on the user-interface, server, and database aspects of the project.

For this checkpoint, we decided to focus on the backend work first. Our backend is a MEAN-stack web server. This means that the database uses MongoDB to store any information about users or activities as JSON objects, and uses the web frameworks Express, Angular, and Node JavaScript libraries to render the websites dynamically. For instance, whenever the user requests to update their information (such as their email, password, favorite activities, etc), the client sends a POST request to the server, which then performs the update on that user's entry in MongoDB. At the current state, we have the backend implementation of registration and login, the profile page, and the matching algorithm for pair activities. Most of the backend was written in JavaScript using the Node and Express web programming libraries to handle the HTTP GET and POST requests that the server sends and receives, as well as calling the MongoDB JavaScript libraries to connect to and asynchronously perform reads and writes to the database. For security measures, we used the JavaScript bcrypt library so that we can encrypt the users' passwords before storing them in the database; this will ensure that users' passwords are securely stored in the database and attackers do not easily steal them if they hack the database. Additionally, to check for user login with the stored cookie before accessing any sites outside of login and register, we used the JWT (JsonWebToken) library to generate the token and store it in the user's browser.

We also made some progress on the frontend, which currently consists of HTML code. For instance, our login page has the two input boxes for email and password, followed by text to label said boxes and the "submit" button. These bare-bones HTML pages for profile, login, and registration are able to perform their intended functionalities; they are able to correctly perform the HTTP GET requests upon connecting with the server and properly display to the client the rendered webpage, as well as perform POST requests to the server to send to or retrieve information from the database. The remaining work for Part C would be to extend the matching algorithm to account for group activities, create the selected activities page, continue working on the edit profile page, and enhance the front-end of our web application.

## 1.2 Design Changes

### 1.2.1 Overview of Design Changes

For the most part, our application largely stayed the same as planned in Part A. As demonstrated in our Use-Case diagram in [Figure 1 on the following page](#), the modules and features we desired to implement are either fully functional or mostly functional, and designed to how we specified in Part A. For Part B, we have implemented the following user features in our webpages: Sign Up, Login, Match Users, View Active Events, and Update Favorite Activities. The Logout and Events pages plan to be implemented in Part C. All of our created modules all are User-available actions, and they

are also all dependent on whether or not the user has logged in through our Login module, as this is checked with a JavaScript web token cookie that is saved to the user's browser. Logout will likely be a method that the user can activate upon clicking a logout button on screen, that will only be available if the user has the login cookie. Additionally, the Events page will include the ability to modify, enter, or exit pair or group events made, but will also only be allowed if the user has logged in.

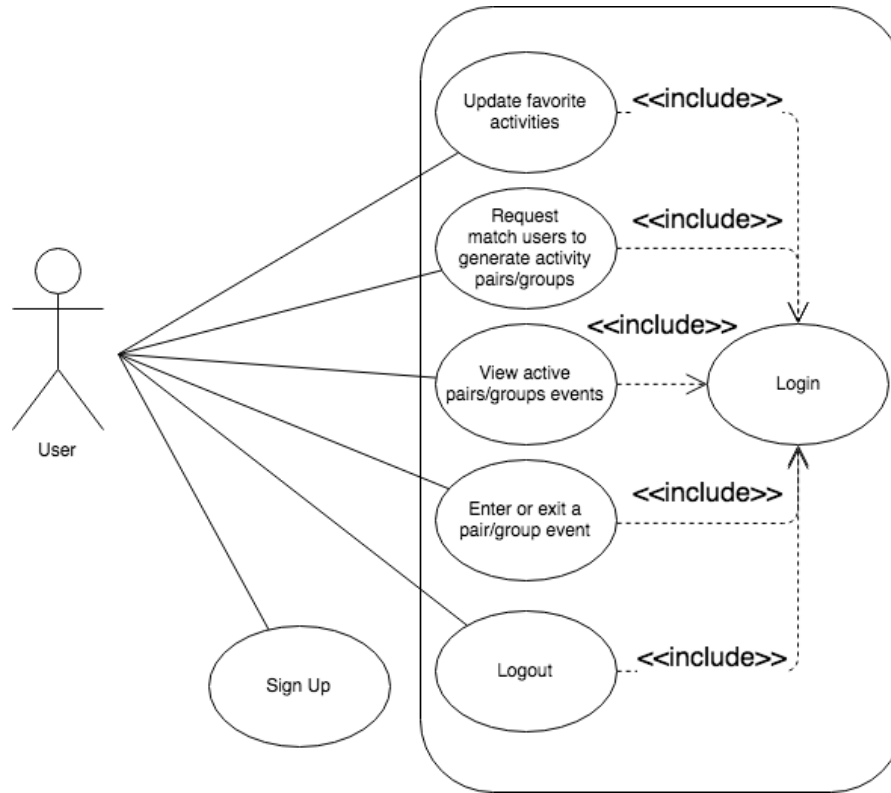


Figure 1: Use-case diagram of our project, as seen in Part A.

According to our UML diagrams from part A, we were to create five classes: User, UserPreferences, UserAuthentication, Activity, and ApplicationLogic. The UML class diagram is largely the same, with a few small modifications, as shown in Figure 2 on the next page. ApplicationLogic was separated into several classes for better decoupling of components. A Database class was created with the responsibility of inserting new users, updating information, and removing users. A Match class was created with the responsibility of generating matched events for users. Also, in the UserPreferences class the three member variables are stored in the database. The User class is implemented in databases.js, in which we define the structure and attributes of a user and implement getters and setters as envisioned. UserAuthentication was simplified into a Verify class with a simple checkLogin function. Instead of the Activity class storing the activity and event, the information is spread across the Activity class, which stores information about a certain activity name and a user's skill and interest level, and the Event class, which stores information about a matched event between users, including its status, location, and time.

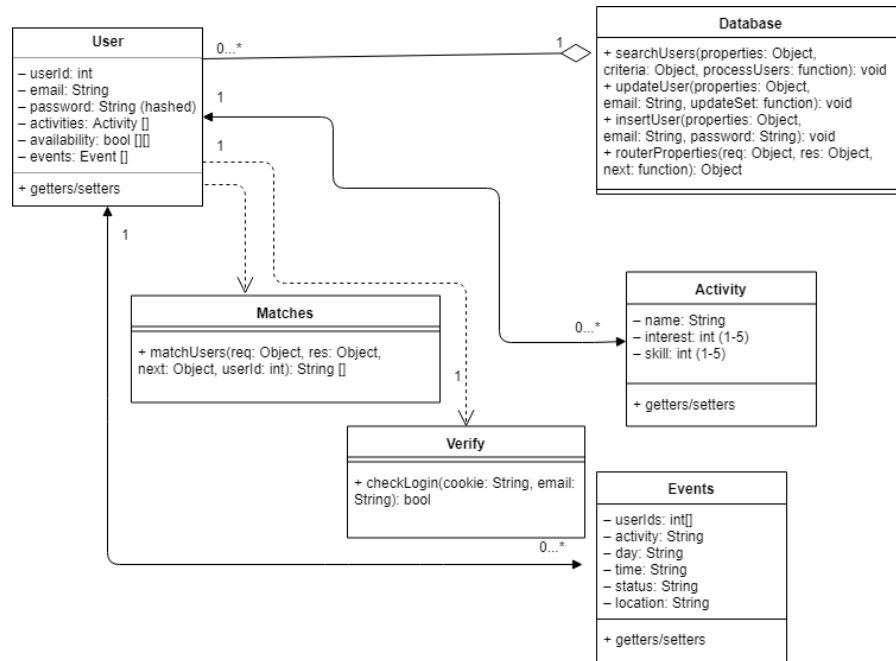


Figure 2: Updated class diagram for Part B.

### 1.2.2 Matching Algorithm

The matching currently consists of two core parts: time availability matching and the activity matching. These two sections are delineated below.

The time availability matching involves taking two arrays and computing an integer for the number of continuous time slots that match up between two users. A time slot is defined as a 30-minute interval on one day of the week, such as Sunday from 1:00 PM to 1:30 PM. The maximum number of continuous time slots is currently capped at a 3-hour maximum, and 30-minutes or less of a time match results in a match score of 0 for this category. Therefore, the score for this category currently ranges from 0 to 6.

Next, the activity matching involves using the interest scores and the skill scores. When creating a profile, interest and skill preferences are ranked by all users, and these scores range from 1 to 5. For the interest match score for an activity, the user's score was added to a fraction of the score of the potential match for that activity. This was done so that the interest score for a user depends more highly on his or her own score, and secondarily based on the other user's interest. For the skill score, the score was inversely related to the absolute value of the difference between the user's score and the potential match's score, in order to provide the highest skill score for two users with similar skill levels. Note that the interest score is higher when two users have higher scores and lower when two users have low scores. On the contrary, the skill score is highest when two users have similar skill scores (even if they are both low skill scores). Note that if the user's interest in an activity is 1 (the lowest possible interest score), then the interest score is 0.

These three scores are then scaled such that they are all out of the same denominator. For example, the time availability score ranges from 0 to 6 while the the difference in skill level has a different range. All scores are normalized by taking into account the maximum and minimum score for each category, and setting all the scores to be out of the same value. Then,

the interest and skill scores are added post-normalization. The activity yielding in the highest interest and skill score sum is selected as the best activity for two users, ensuring that the interest score is not equal to 0.

Finally, we add the three scores to determine the total score, such that interest and time availability scores being 0 would result in the overall score being 0. This process is then applied to compute a score for each potential match user.

### 1.2.3 *Security Considerations*

Due to the sensitive nature of the information that bActive requests from registered users, ensuring that such confidential information remains private is of utmost importance. As such, we have encapsulated all client-side interactions with such confidential information behind a RESTful API; that is, we ensure that only users who have the proper credentials are able to successfully access such information.

We implement a MongoDB server that contains login information, linking the user's email to a one-way cryptographic hash of their password generated by the bcrypt module in node; the server then compares the cryptographic hash of the user-entered password with the password located on the server. If the passwords don't match, the user is denied access to login-protected information; however, if they do match, the server will issue a JSON web token containing the username and an expiration date two hours after creation of the cookie.

The above security protocol prevents hackers who obtain access from obtaining passwords in plaintext, as all passwords are stored only after being processed through a one-way cryptographic hash. However, other security concerns still exist, and due to the difficult nature of addressing them, we will not be preventing them for this project. One such concern is that even though a hypothetical hacker may not be able to obtain passwords in plaintext, all other information on the server is stored in plaintext, which means that confidential information, aside from the password, would be available to hackers with access to our server. Additionally, as the passwords are hashed on the server-side, rather than the user-side, and our application does not use the HTTPS protocol, a hacker can use a "monkey-in-the-middle" attack to retrieve passwords sent to the server in plaintext. A similar attack can be used to steal cookies and create illegitimate copies; however, this particular issue is partially remedied by the two hour expiration on cookies.

## 2 DESCRIPTION OF API

### 2.1 API Declarations

We have written documentation for our API using the JSDoc format, and generated a documentation website using the jsdoc-toolkit utility, jsdoc.

Our API is described in the following method table.

Method Summary	
	<code>computeInterestMatch(curr_user_interest, potential_match_interest)</code> Returns a score for interest match.
	<code>computeNormalizedScore(curr_score, curr_max)</code> Normalizes the score for the three categories, so all three types of scores have the same maximum and minimum values.
	<code>computeSkillMatch(curr_user_skill, potential_match_skill)</code> Returns a score for skill match.
	<code>generateActivityMatches(curr_user_activities, potential_match_activities)</code> Generates a list of activity matches with an interest match and a skill match for each activity.
	<code>getAvailabilityMatch(curr_user_availability, potential_match_availability)</code> Returns maximum number of consecutive overlapping half-hours for two users.
	<code>getAvailabilityMatchScore(curr_user_availability, potential_match_availability)</code> Returns a score for the availability match between two users.
	<code>getBestActivityMatch(curr_user_activities, potential_match_activities)</code> Returns an object containing the best activity match between two users and the interest and skill level scores for that activity.
	<code>getBestActivityMatchFromList(activity_matches)</code> Returns an object containing the best activity match between two users and the interest and skill level scores for that activity, given a list of activities and scores.
	<code>insertUser(properties, email, password)</code> Function to insert a new user into the Users collection in MongoDB.
	<code>matchUser(curr_user, potential_match)</code> Compute a match score for two users.
	<code>matchUsers(req, res, next, userId)</code> Key function to generate matches for a single user, given the user id.
	<code>routerProperties(req, res, next)</code> Function to return a JSON object containing the express router properties.
	<code>searchUsers(properties, criteria, processUsers)</code> Function to search for all users meeting a criteria.
	<code>updateUser(properties, email, updateSet)</code> Function to return update a user's data in Users collection in MongoDB.

Figure 3: Our API is described in this method table, which was generated by jsdoc.

In the following parts, we describe a few important functions, and then argue how our API strictly adheres to the Information Hiding Principle.

### 2.1.1 *generateActivityMatches*

---

```
{Array} generateActivityMatches(curr_user_activities,
potential_match_activities)
```

Generates a list of activity matches with an interest match and a skill match for each activity.  
Defined in: [match.js](#).

**Parameters:**

**{Array} curr\_user\_activities**  
A list of activities for the current user with interest and skill level scores.

**{Array} potential\_match\_activities**  
A list of activities for the potential match user with interest and skill level scores.

**Returns:**

**{Array}** List of objects with the potential match activities. Each object has the name of the potential match activity, a skill score, and an interest score.

Figure 4: The entry for the key function `generateActivityMatches`, which was generated by jsdoc.

### 2.1.2 *matchUser and matchUsers*

---

```
{Array} matchUser(curr_user, potential_match)
```

Compute a match score for two users.  
Defined in: [match.js](#).

**Parameters:**

**{Object} curr\_user**  
An object containing the user profile information for the logged in user.

**{Object} potential\_match**  
An object containing the user profile information for the the match candidate for the logged in user.

**Returns:**

**{Array}** An array of a matched activity and the match score. A score representing how good the match is between the logged in user and the potential match. This match is based on availability, interest level, and skill level.

---

```
{!Array} matchUsers(req, res, next, userId)
```

Key function to generate matches for a single user, given the user id.  
Defined in: [match.js](#).

**Parameters:**

**{Object} req**  
The express routing HTTP client request object.

**{Object} res**  
The express routing HTTP client response object.

**{callback} next**  
The express routing callback function to invoke next middleware in the stack.

**{number} userId**  
An integer that represents a user id.

**Returns:**

**{!Array}** A sorted array of matching users, from highest score match to lowest score match.

Figure 5: The entries for the key functions `matchUser` and `matchUsers`, which were generated by jsdoc.



### 2.1.3 *insertUser*

---

```
{Void} insertUser(properties, email, password)
```

Function to insert a new user into the Users collection in MongoDB. The server returns a 404 error if the email in the request body is already in use. Otherwise it creates a new user with the given userID and hashed password, default availabilities, and no activities or events, and adds it to the MongoDB database, returning a 201 status code.

Defined in: [database.js](#).

**Parameters:**

**{Object} properties**  
JSON object containing the express router properties.

**{string} email**  
The email of the new user.

**{string} password**  
The password of the new user that is encrypted with bcrypt.

**Returns:**

**{Void}**

Figure 6: The entries for the key function `insertUser`, which were generated by jsdoc.

### 2.1.4 *searchUsers*

---

```
{Void} searchUsers(properties, criteria, processUsers)
```

Function to search for all users meeting a criteria. The server returns a 404 error if no users matching criteria are found.

Defined in: [database.js](#).

**Parameters:**

**{Object} properties**  
JSON object containing the express router properties.

**{Object} criteria**  
JSON object to determine the search criteria for the user(s) we wish to find.

**{callback} processUsers**  
A callback function to be invoked on the resulting set of users found.

**Returns:**

**{Void}**

Figure 7: The entries for the key function `searchUser`, which were generated by jsdoc.

### 2.1.5 *updateUser*

---

```
{Void} updateUser(properties, email, updateSet)
```

Function to return update a user's data in Users collection in MongoDB. The server returns a 404 error if unable to find a matching user in the database. Returns a 201 if it successfully updates user data.

Defined in: [database.js](#).

**Parameters:**

**{Object} properties**  
JSON object containing the express router properties.

**{string} email**  
The email of the user whose fields we wish to update.

**{Object} updateSet**  
JSON object containing the data we wish to update.

**Returns:**

**{Void}**

Figure 8: The entries for the key function `updateUser`, which were generated by jsdoc.

In the following section, we argue how our API strictly adheres to the Information Hiding Principle.

## 2.2 Information Hiding

Our project is implemented and designed with the information hiding principle in mind; thus, for most, if not all, anticipated design change scenarios, the API declaration is unlikely to change. For example, we noticed that all the main pages interact with the MongoDB database in some way. If we were to theoretically decide to change the database from MongoDB to MySQL format, we might need to change all the pages so that it could support the format we read the user information out from the database, as well as call the right database functions to search or update the database. Therefore, we delegated a JavaScript class “database.js” that does all the work of handling the interactions with the MongoDB database. This allows us to decouple from all the other web pages any need for knowledge of the database and we can instead just simply call helper methods from “database.js” to update, insert, or select users from the database if any of our other pages desire. We will therefore no longer need to do any additional work of changing the other pages for our website if we need to change our database format from MongoDB. The following figures can help visualize how this scenario can play out with and without information hiding.

In Figure 9, without information hiding, all of the pages must change their internal function designs in response to changing the MongoDB Database as they all call MongoDB specific functions to read and write to the database. In contrast, in Figure 10 on the next page, with information hiding, only a few internal details inside Database class have to change to perform reads and writes.

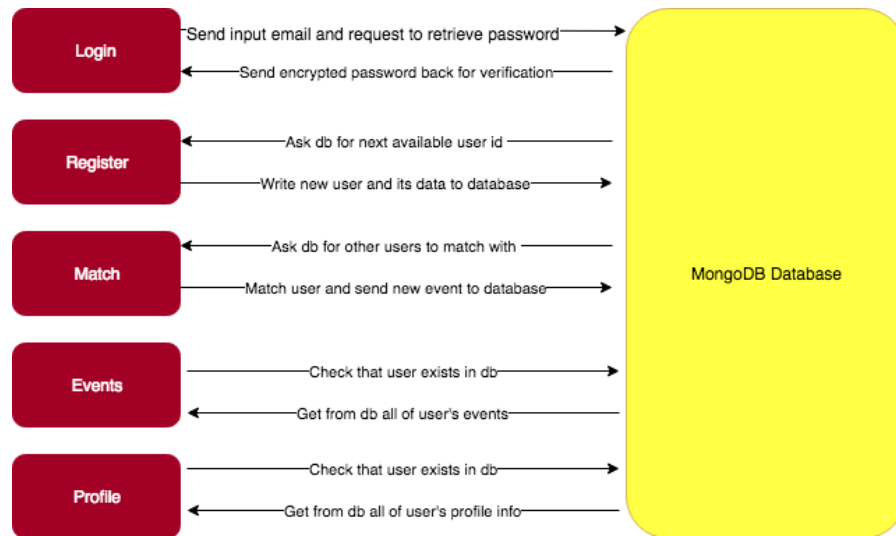
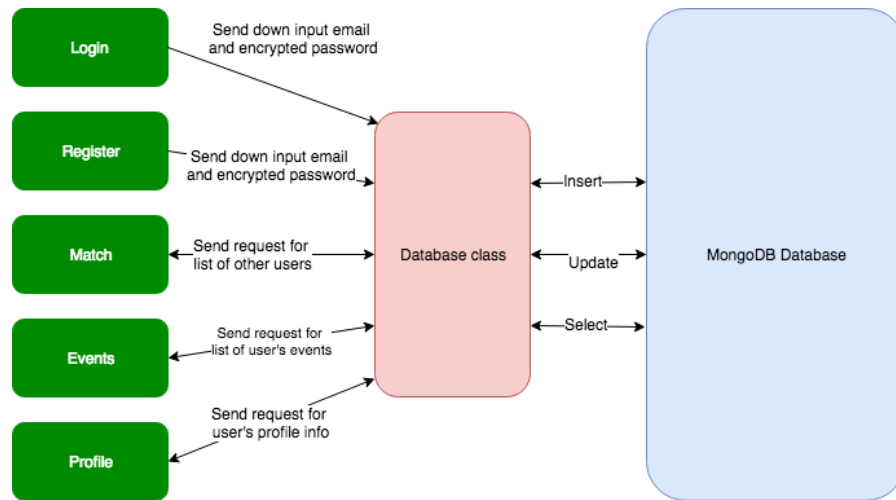


Figure 9: Backend designs of core pages implemented without information hiding. All of the pages must change their internal function designs in response to changing the MongoDB Database as they all call MongoDB specific functions to read and write to the database.



**Figure 10:** Backend designs of core pages implemented with information hiding. If we change our MongoDB database format, then only a few internal details inside Database class have to change to perform reads and writes. Meanwhile, our core pages don't have to change since they pass information to the Database class to handle database interactions for them. Then the Database class responds with formatted lists which we do not foresee changing in the future.

Additionally, our main pages are only responding to HTTP GET and POST requests, we do not need to worry about having to change the parameters of the GET and POST functions. This is because the get and post methods are Express JavaScript library methods that are only allowed to take the HTTP request and response JavaScript objects, and next callback function, with the request object's body and parameters containing the fields that we need. As for accessing parameters, because our main website's pages revolves around only a Users collection (or the table equivalent in MySQL) in the database, this greatly simplifies what information we need to display to the client upon reading from database. The user has the following fields: email, user ID, password, list of activities, list of events, and list of available times. Email, password, and user ID are essential fields from the request that we will not be changed anytime soon because they are the foremost ways of identifying the unique users and accessing their information from MongoDB. Also, the email and sometimes the user ID can be directly passed as a singular input to a lot of the functions, such as the `updateUser()` function in Database class, as this function processes the user's information by looking at its ID first before updating it with the input JSON object of the set of fields to update. Meanwhile, the activities, events, and available times must be in list format because a user can have an aggregation of all of them. While the internal format of these data structures might change in the future, we implemented all functions that use them to only use basic comparator operators or just send the fields to be displayed on the front end. So although the front end displays might need to know a little, the backends of all the modules do not need to know the internal format of their data to do either of these tasks.

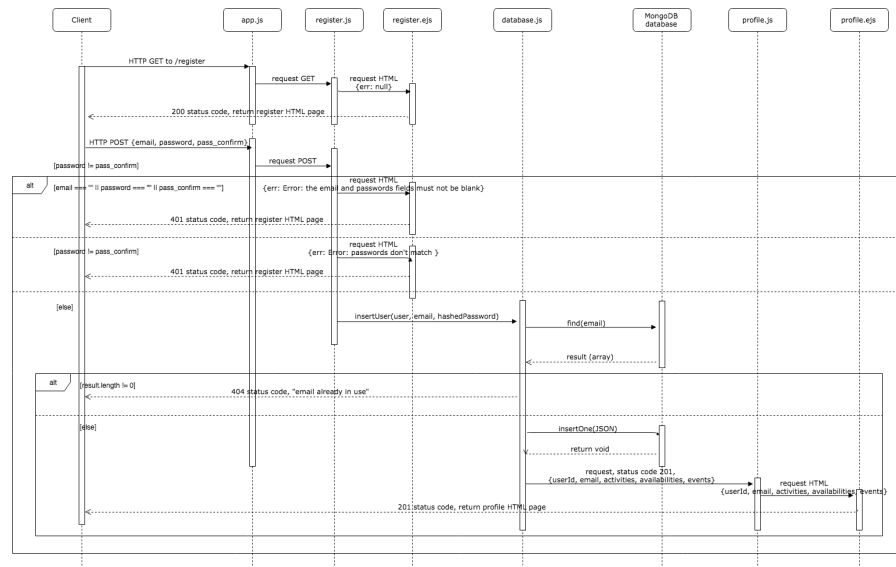
Below, in [11 on the following page](#) is a changeability matrix that can be used to assess the ways in which five possible anticipated design changes we might consider that might affect the core modules of our application. The

key colors are green, yellow, and red, and they represent unaffected, implementation dependent, and affected modules due to the change, respectively.

	Login	Register	Match	Events	Profile
Change database format from MongoDB					
Change format of user's list of available times from 2D Array of booleans					Only the front end of profile module needs to change to display
Adding a new activity for users to participate in.					
Change of format of user's activities list					
Adding user interactions in groups made in the events page					

Figure 11: A changeability matrix describing how the core modules of our application may change under five possible anticipated design changes.

## 2.3 API Integration Sequence Diagram



**Figure 12:** Sequence diagram of Register module functionality, illustrating interactions between key parts of our API.

The above figure, Figure 12, is a sequence diagram of the Register module and its basic functionality with handling HTTP GET and POST requests. The GET method renders on the client the registration page (input boxes for the user's email, password, and confirmation password), as well as showing any error messages if one exists. It simply asks the entry point page to get the HTML display from the Register file and renders it upon receiving it. The POST method is for when the user submits their email and passwords to request creating a bActive account. When the user submits their information, it sends a POST request to the server to first sanity-check the inputs; if any of the fields are blank or the passwords don't match, an error status code and message are returned and the registration page is returned to be redirected to it. Otherwise, the server then asks Database class to look up any users with the given email from MongoDB; if a user with the email already exists, it returns an error status code and message, along with a redirect to the Register page. Otherwise, it requests Database insert a new user into MongoDB and performs the insert and then returns a request to redirect the user to the user's profile page. Overall, this sequence diagram and the functionality described illustrates how the Profile and Register APIs will interact with each other and the server-side database at runtime whenever a new user wants to create a new bActive account.

### 3 TESTING

As our project is a web application that users interact with via HTTP GET and POST requests, we found it most appropriate to design our test cases around these sort of interactions, and to do so, we generated GET and POST requests via Postman.

Our basic use-case tests involved simply verifying that web pages returned content appropriate without errors. As we expect the specific content of our web pages to change significantly over the course of this project, we abstained from specifying exact contents in the returned HTML from the server; rather, we based the success of the test cases on whether the returned HTML contained some basic information and the appropriate status code, as defined by HTTP protocol standards.

For integration test cases, we decided to test interactions such as the interaction between the login API, which generates cookies for successfully logged-in users to authenticate themselves with, and login-restricted portions of our website that require the aforementioned cookie to access. Passing such test cases requires that the server successfully returns a cookie upon successful login, and that the cookie in question can then be used to properly authenticate the user in question when accessing restricted portions of the website. As with the basic use-case tests, as the returned HTML is subject to significant change over the course of the project, the success of the test cases was again based on whether the returned HTML contained some basic information and the appropriate status code.

The below test cases can be found in our github project page under the tests folder: <https://github.com/ryanlo7/bActive/tree/master/tests>

#### 3.1 Login Page HTTP GET

This was a very basic use-case test that we used simply to verify that our server was running as intended. Upon receiving an HTTP GET request to the login page, we checked whether the server returned the appropriate status code of 200 along with an HTML page containing an email and password field. Any other status code, or an HTML page not containing the aforementioned fields, would indicate some sort of server-side error that would need to be addressed.

#### 3.2 Login Page HTTP POST (incorrect password)

This was another basic use-case test that we used to verify that the server successfully prevented login attempts with an incorrect password to ensure security. Upon receiving an HTTP POST request containing a valid username and incorrect password in its body, the server should make a call to MongoDB, compare the cryptographic hash of the incorrect password with the cryptographic hash of the password stored on the server, and recognize that an incorrect password was entered, thus redirecting the user back to the login page with a status code of 401 (unauthorized). Any other status code would imply some sort of error; notably, a status code of 200 (OK) would indicate a massive security vulnerability in the password verification system that would require urgent attention, especially if the server were to also return a valid cookie that the user could then use to authenticate themselves to login-protected pages.

### 3.3 Login Page HTTP POST (correct password)

This was a unit test-case designed to test whether the server could recognize a correct password, redirect to the profile page, and issue a cookie that would authorize the user to access login-protected content. As mentioned previously, password checking is implemented by computing a cryptographic hash of the password provided by the user, and comparing with the cryptographic hash stored on the server. As a redirect is involved, the server should send a status code of 301; any other status code would imply some sort of error; in particular, a status code of 401 (unauthorized) would imply that the server was not properly accepting the credentials of the user.

### 3.4 HTTP GET Profile Page with Login Verification

This was an integration test-case designed to test whether the profile page displays the profile for a user given a verified cookie indicating that the user has logged in within the last two hours. In order to execute this, the POST request described in Test Case 3 must first be run to obtain the JWT authorization cookie, and appended to the GET request header for the profile page. The status of the response should be 200, and the body of the response should be an HTML page that includes the user email, list of activities, and colored table of availability.

### 3.5 HTTP GET Event Page with invalid userId

This was a basic use-case test designed to test whether the events page displays the events page for an invalid userId. To test this, the GET request must contain an invalid userId (not in the database at that point in time). We expect a status code of 404, indicating userId not found. Rendering other status codes or pages indicate a server-side error needing further investigation.

### 3.6 HTTP GET Event Page with valid userId, without cookie

This was a basic use-case test designed to test whether the events page displays the events page for an valid userId without a cookie. To test this, the GET request must contain a valid userId (in the database at that point in time). We expect a 401 status redirect to the login page since the user does not have a cookie and is thus not logged in. Rendering other status codes or pages indicate a server-side error needing further investigation.

### 3.7 Unit Test for Availability Match

This was a unit test case in JavaScript to test whether the time availability match between two users was computed correctly. To test this, we fill two time availability matrices with values for availability of two users. Then, we compare the time availabilities of the two users and call the function to compute the maximum overlap in availability between the two users. Assert statements are used to check that the computed overlap is correct for each of three cases that will be described below; the true value of maximum overlap is known since the availability arrays were created with a known number of overlap.

Three different availability matrix combinations are tested with the assert statements in this function to ensure that the maximum overlap is as expected: one combination ensures that the function returns 0 when there is no match, one combination ensures that if there are two matching time periods, the largest availability is returned, and the third combination ensures that contiguous time periods are detected across multiple days (i.e. if two users are available from late night on one day to the morning on the other day, we must detect this).

### 3.8 Unit Test for Activity Match

This was a unit test case in JavaScript to determine whether the correct activity was selected given two users' activity preferences. The true best activity is known prior to the test case since the activity arrays are manually filled with synthetic activity data. The function uses assert statements to check that the expected activity is selected from the lists when determining the best match. Furthermore, the function also uses assert statements to verify that the interest and skill level scores are within bounds of the expected scores (between 0, and the constant used for the maximum score post normalization).



## 4 USER INTERFACE

In this section, we describe the existing user interface and user experience.

In our registration page in Figure 13, we allow a user to register by submitting an e-mail address, a password, and a password confirmation in the appropriate fields.

### Welcome! Please register as a new user!



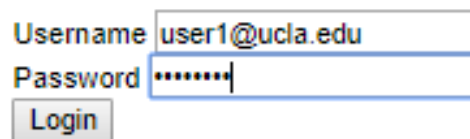
Registration form fields:

- Email:
- Password:
- Confirm Password:
- Register:

Figure 13: The registration page of our application as of 15 November 2018.

In our login page in Figure 14, we allow a returning user to log in to her account, by entering her e-mail address and password.

### Login



Login form fields:

- Username:
- Password:
- Login:

Figure 14: The login page of our application as of 15 November 2018.

Once the user has logged in, she can view her profile in the page shown in Figure 15 on the following page. This displays her e-mail, activity preferences, and availability preferences.

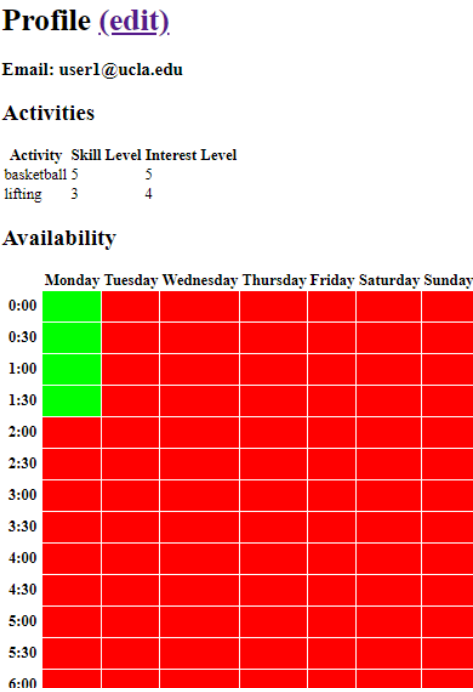


Figure 15: The profile page of our application as of 15 November 2018.

Once the user has logged in, she can edit her profile in the page shown in Figure 16. She can alter her e-mail and password, as well as select preferences for her activities and availabilities.

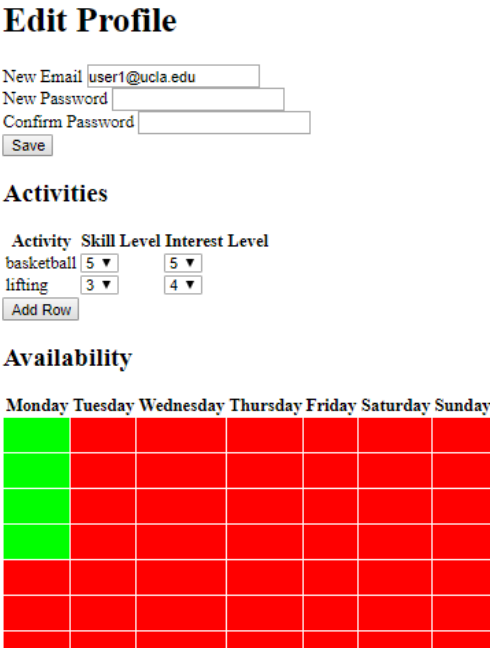


Figure 16: The profile editing page of our application as of 15 November 2018.

The user can also see her list of upcoming events, in the form of a friendly table as shown in Figure 17 on the following page.

## Upcoming Events

Activity	Date	Time	Meetup Location
lifting	monday	1AM-2AM	bfit
-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-

Figure 17: The events page of our application as of 15 November 2018.

## 5 CONTRIBUTIONS

### 5.1 Krishna

**Krishna** worked on the back-end and front-end for the events page, and wrote several test cases for them by leveraging Postman. The events page pulls user's accepted matched events from the database and renders them in the EJS file in a tabular display. It also includes a menu to access other application pages. She incorporated the user authentication logic from the profile page into the events page to ensure only users who are logged in can access their respective events page.

### 5.2 Eddie

**Eddie** worked on the user authentication for the server-side API, including implementing the login page and encapsulating all login-only pages behind a RESTful API; that is, ensuring that only authenticated users can view them; otherwise, they are redirected to the login page. He designed the MongoDB user server such that passwords were stored as cryptographic hashes, rather than in plaintext, so that passwords could not be stolen by hackers with access to the server. In addition, he implemented cookies to track login, formatted as JSON web tokens, that the browser can then use to enable access to login-only pages. In addition, Eddie did significant work in writing the unit and integration test cases, formatted as HTTP GET and POST requests via Postman into JSON.

### 5.3 Eric

For Part B, **Eric** worked on adding JSDoc documentation to each of the important functions in the JavaScript files, and generating the documentation website therefrom, which can be found at <https://github.com/ryanlo77/bActive/tree/master/docs>. He also worked on the several sections of the project report, proofread it, and typeset it in  $\text{\LaTeX}$ .

### 5.4 Ryan

**Ryan** worked, for Part B, on the user registration page, both server-side and front end. This page allows new users to create their new bActive account and the user is stored to the database. This part was implemented in the MEAN stack so that we save all user information to a MongoDB user server database and they are stored in JSON objects as the format. Additionally, after implementing the registration page, he started work on making several of the functions, especially searches and updates to the MongoDB database, follow the Information Hiding principle; this involved either refactoring the function entirely, or moving it to a separate JavaScript file, so that we can call them from so that the core pages did not need to know the internal information of the users from the database and the format they are in. This was successfully done for the registration page and he is in the process of converting the profile and login pages.

## 5.5 Ramya

For Part B, **Ramya** worked on writing the matching algorithm to determine the best matches for a user for a pair activity. She decided to split the matching algorithm into two parts: determining time matches, and determining activity matches. Developing this algorithm involved creating several functions to compute scores for different categories, setting minimum and maximum caps on scores for each category, and normalizing the score for each category. She developed a unit test for the availability match section, for different sets of inputs. She implemented these sections in JavaScript and wrote JSDoc for these sections. Ramya also worked on finalizing the formats of the data that would be stored in MongoDB with Nicole.

## 5.6 Nicole

For Part B, **Nicole** worked on the server-side and front-end for the profile and edit profile pages. These pages get user data from the MongoDB database and render them in the EJS files so that personal information, activities, and availability are displayed. This also included finalizing the formats that all data would be stored in the MongoDB documents. She also integrated the user authentication to the profile page so that only logged in users can view their profiles, and users can only access their own profiles.