B-ACTIVE

by the Active Bruins

KRISHNA BABU, EDDIE HUANG, ERIC KONG,
RYAN LO, RAMYA SATISH, & NICOLE WONG[1]

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

A demonstration of the project in operation may be found at

https://youtu.be/rZpm01iYCkY

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 C, the final draft, of the project.

CONTENTS

LIST OF FIGURES

[1] *Department of Computer Science, University of California, Los Angeles.*

1 MOTIVATION

A common problem faced by UCLA students is finding a time to meet with others for leisure activities, given everybody's busy schedules. For example, if a student wants to participate in an activity such as working out or playing basketball, but none of their friends are available, it is currently difficult to take advantage of the size of UCLA and find new people who are available at the specific time the student would like to meet up for the activity. Furthermore, given the stress associated with attending a university, a way to keep students active while promoting new friendships would be physically, mentally, and socially beneficial.

The application, as proposed in Part A, allows users to save their favorite activities along with their preferred times to meet up into a profile. A minor change from part A is that preferred location within UCLA is no longer part of the user profile, since we wanted to maximize potential matches and the UCLA campus is small enough such for the location to not be significant; instead, the application suggests a convenient on-campus location based on the activity. Another minor change is that the activity matching in the application is not for larger group activities, as we chose to focus on pair activity matching and other new functionalities that will be described in later sections.

The application matches each student with others who have similar athletic activity interests and time availabilities, also taking into account the users' skill levels for those activities. These suggestions will allow students to match with other students for an activity, taking advantage of the university atmosphere to promote new friendships between students with similar recreational interests. Ultimately, the vision for the application is to make it easier for UCLA students to meet up with other students for leisure activities, while encouraging new friendships and promoting an active lifestyle.

2 USER BENEFITS

B-Active aims to take advantage of the college campus atmosphere and encourage Bruins to make new friends while pursuing an active lifestyle. Specific user benefits are delineated below.

2.1 Removal of social barriers

B-Active *removes barriers while finding matches for users*. Many students struggle to find or reach out for people that are available for activities at the same time, but bActive entirely handles athletic activity scheduling for users, as proposed in Part A.

Upon registering, the application allows a user to fill out a profile with time availability. After completing the profile, the matching algorithm is run and the user will be shown a list of potential matches, ranked by the match score described in the Part B report. The event the application proposes for each match is the most ideal event given both users' activity interests and skill levels. The proposed event is fully-detailed event with location, activity, and a time guaranteed to match both users' availability listed in their profiles, and each user merely has to click accept on a generated event for the event to be added to the confirmed events page for both users. Unlike Facebook groups, which require users to do the work of scheduling, the application does all the work. Ultimately, this benefits users by removing barriers in reaching out and scheduling athletic events.

2.2 Encouragement of athletic lifestyle

B-Active *encourages athletic lifestyle in a beginner-friendly way*. One key barrier that inactive students face is finding people to participate in activities with at a beginner skill level, as club and team sports often have more experienced students. BActive incorporates skill level into the ranking algorithm and allows users to view other users' profiles, benefiting both experienced and new athletes.

When creating a user profile, the user enters skill levels for the desired activities. Upon completing the profile, the matching algorithm, described more thoroughly in part B of the report, prioritizes those with similar skill levels when displaying the potential matches. Additionally, the application provides the user with the option to view other users' profiles, so the user does not have to accept events with those at a higher skill level if they do not feel comfortable doing so. Therefore, the application encourages beginners to be active as well.

2.3 Self-determination of commitment

B-Active *allows users to choose their commitment frequency*. Many students are unable to join sports with a regular (i.e. weekly) time commitment, but BActive lets users accept activities at frequencies that most benefit them.

Upon completing the profile and matching users, the application generates new events regularly, but allows a user to accept events at any frequency he or she chooses. Commiting to an event on one Monday does not accept events for all Mondays, for instance. Thus, users have total control over when they choose to attend events. Furthermore, contact information is pro-

vided so that if the users do want to meet up regularly, they can do so. This view profile ability is a new feature that was not proposed in Part A or B.

2.4 Flakiness detection

B-Active *warns users about potentially flaky users*. BActive protects users from those who do not show up to events by allowing users to rate others. This benefit is a new functionality that was added in part C, and was merely a proposed stretch goal in part A.

After an event is over, each user is prompted to rate the other user. Prior to accepting an event, a user can view the other user's rating. This benefits users by encouraging people who accept events to actually show up to those events, and is a safeguard against people who regularly stand other people up after accepting events.

3 FEATURE DESCRIPTION

Our product comprises of several key features: making a profile on bActive, getting suggested event matches, and interacting with other users. We examine each of these components in more detail in the following sections, from the perspective of a user.

3.1 Making a Profile on bActive.

To register for B-Active, the user enters her name and e-mail address and selects a password. This leads her to her Profile page. The name is an additional piece of information that was added after Part B. Additionally, if the user is logged in, she can go to the standard Login page and input her email and password to Login and receive the registration token that way, and then will be redirected to her Profile. If the user accidentally went to the wrong page (i.e. she intended to login instead of register), there is a handy "BACK" button she can click to navigate back to the home page.

Upon entering the Profile page, the user can view their profile information. This page displays their basic contact information (their name and email) so that others viewing their profile can directly message them to contact them. However, for security the 'edit' functionality is only available on a user's own profile, not when viewing others' profiles. It also displays a rating out of five stars (and a small text to display the number of users who have rated the user) as a general indicator of how positive of an experience other members had with the user, all to discourage flaking from events. Below the basic user info are the items that directly relate to the app. First, there is a horizontal listing of all the user's preferred activities. A picture of each activity is displayed on screen and also contains a caption with the user's skill level and interest level, the last two of which will factor into our matching algorithm. Lastly, there is also a weekly calendar grid that shows the user's availability schedule, displaying which times (staggered every half-hour for all 24 hours of the day, 7 days a week) each day of the week they're free to do an activity together. This chart is taken from a grid in the user's fields that is also used for the matching algorithm later on.

If the current user is viewing their own profile, they will be able to click a button to edit their profile information. They are allowed to change their name, and add and select, or delete favorite activities and change their skill and interest levels with drop down menus. One can also edit one's availability schedule by clicking the checkboxes in the grid to indicate which half-hours of each day the user is free on.

3.2 Get Suggested Event Matches.

The matched event generation function will be run in two instances: updating a user profile and on a weekly job. If a user updates and saves the availability on their edit profile page, which includes new user registration, the matching algorithm is run to generate and store suggested events in the database. Because the events generated by the matching algorithm are limited to the scope of the upcoming week, in order to create new events every week, the server will be scheduled to run the matching algorithm once a week. In a possible production version of our application, one option would be to use AWS to host our website and use AWS Lambda to run the matching function when an AWS CloudWatch scheduled event is triggered.

The event list page shows all events, past and future, that the user has committed to, from the events that were generated via the matching page. After each event, users will be able to rate all other participating members (members to clicked "accept" for the events listed in the "match" page) on a five-star scale, promoting accountability and encouraging users to actually show up to events that they commit to.

3.3 Interact with Users.

Users can view who they are attending a specific event with, and explore their bActive profile to learn more about them. This equips users with info about their matches such as rating score, interests, and availabilities, as well as their email for contact. After attending a confirmed event, users have the ability to rate the user they met up with. This is the 'flakiness score' which is used by users to decide the flakiness of a certain matched user, which informs whether they accept a certain suggested match or not. These core functionalities for a friendly user experience were added on as well after part B.

4 FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

4.1 Functional Requirements

4.1.1 *Registration*

To register for B-Active, the user enters her name and e-mail address and selects a password. Default data is then generated for her, which she can then edit on her profile page.

4.1.2 *Login*

A user must be able to log into her profile. Upon doing so, she can use the application and perform all actions described below.

4.1.3 *Edit profile information*

The user selects, from a list of activities, the ones in which she seeks to participate (her interests), by rating them on a scale from 1 to 5. For each activity, she also indicates how proficient she fancies herself to be in it, on a friendly scale from 1 to 5. Pair activity matches are generated for the user. Finally, she checks off her availability so that she can be matched with other users.

4.1.4 *User matching*

The application matches users based on similar interests and availabilities, using a "similarity score" algorithm. The algorithm was described in great detail in part B. From the user's perspective, other users with similar interests and compatible availabilities will be suggested to her on the matches page. She can browse the profiles of these other users. She can then accept or decline matches as desired.

4.1.5 *Activity event generation*

When users agree with their matching, the system will generate an activity event for them, and invite each of them to the event displayed on the events page. This event will occur at a time which is compatible with all of its users involved, and at a place which is most convenient for the users. The users can adjust the time and place of the event. The system will tactfully broadcast each of the users' contact information to one another, so that they may communicate and coordinate. On her dashboard, a user can see a list of pending activities. We changed the UI of this page to no longer be a calendar, but instead organize the events by category.

4.1.6 *User points/ratings system*

After each activity is complete, the user can review her experience by giving a rating (out of 5) to participants who showed up, and penalising those who did not. This gives the participants an incentive to attend events that they accept. This functionality was listed as a stretch goal in Part A, but in Part C, we chose to prioritize it over the group activity feature.

4.2 Non-Functional Requirements

Some non-functional requirements that were critical to the success of our application are security, usability, testability, and modifiability.

4.2.1 *Security Considerations*

All the information currently requested by bActive is not sensitive enough to require login credentials, aside from the password. Thus, we determined that the only sensitive information that required care in terms of security was the password; everything else is publically available and requires no authentication to view.

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.

While viewing information about users (besides the hashed password) is unprotected, write operations to the database are protected by a RESTful API. As we decided to use Angular for our project, we needed some sort of API between the front-end and the back-end, and to achieve this, we defined several URLs with PUT, POST, and DELETE requests associated with them that ran the appropriate modifications. While preventing database writes from being conducted by anyone without a proper JSON web token hinders the ability of attackers from modifying the data of other people, we did not implement protections against injection attacks by attackers who may try to create their own user profile and then launch injection attacks through our write APIs. Thus, it is entirely possible for an attacker to cause severe defects to our database, and achieve other nefarious goals, by sending carefully structured PUT, POST, or DELETE requests to our API when they have the appropriate cookie. This was not handled due to the amount of time and energy required to conduct research on exactly how such attacks can be executed and how to check input to block them.

4.2.2 *Usability*

One caveat to usability, as mentioned in Part A, is that the number of matches likely depends on the number of people using the application. The more people using the application, the higher the chances that a person would be matched to similar times and activities; there is a network externality present.

4.2.3 *Testability*

Testability of the similarity scoring and matching process was primarily achieved by ensuring that the similarity score computation was deterministic. In other words, the score and the order of proposed users must thus be the same every single time the algorithm is run. Unit tests were written using the JavaScript test framework Mocha, in order to ensure correct scores for different parts of the matching score computation. Furthermore, Mocha was also used to create unit tests to ensure that ideal activities were being generated on mock user data. Ultimately, the deterministic nature of matching made the matching process extremely testable using mock data.

4.2.4 *Modifiability*

Modifiability is a key non-functional requirement, and we achieved this by taking the information-hiding principle into account when designing our application. For instance, our application involves using a database. We chose to use MongoDB, but we designed our code by including a database class to decouple which database we are using, so that if, for example, in the future we were to change the database. Our usage of the information-hiding principle with more examples is fully detailed in the Part B report.

5 DESIGN

Our application design stayed similar as planned in Part A, but the modifications and updated diagrams are provided below. We describe the modified class design in UML class diagram section. The libraries and packages are described in the section with the sequence diagram.

5.1 Use-Case Diagram

As demonstrated in our Use-Case diagram in Figure **??** on page ??, most modules and features we desired to implement are either fully functional or mostly functional, and designed to how we specified in Part A. However, some features have been removed and others have been added, based on what we prioritized for the application.

We have implemented the following user features in our webpages: update user information, accept or decline events, view accepted events, and register. A feature that was removed from part A is group events, as our web application currently only matches for pair events. Moreover, instead of implementing the logout feature we had mentioned in Part B, we currently wait for the cookie in the browser to expire since that achieves the same purpose. We did not implement these two features since we prioritized two other features that were not part of our original part A requirements, but were deemed more important. The two features that were added to part A were the viewing other users' profiles and rating other users. We prioritized the rate users feature since we considered it important that our application minimizes the chances of a user getting stood up by someone who accepted the invite. We prioritized the view profile feature since it allows the user to look at additional information (such as skill and rating) prior to accepting a match.

Furthermore, the matching is no longer directly user created, but rather, it is run automatically when the profile is edited. This is described in the feature description of this report and shown by the new include in the use case diagram.

All of our created modules 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.

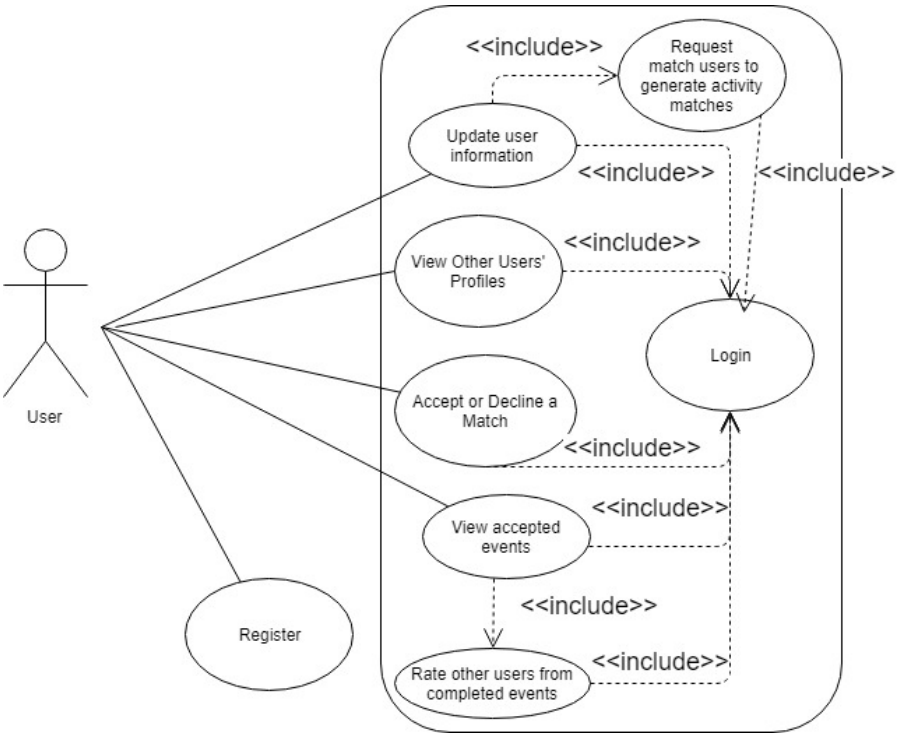


Figure 1: Our modfied use-case diagram.

5.2 Class Diagram

The UML class diagram in Figure **??** on page ?? is largely the same as in Part B. The application logic was separated into several classes to decouple the components. The Database class has the responsibility of inserting new users and events, updating information, and removing users. The Matches class has the responsibility of generating matched events for users. The User class holds the user's list activities and events, as well as their fields used for criteria for matching with other users (such as interests and availability schedule). The Verify class contains a simple checkLogin function. The Event class has the event information.

Some changes were made in the relationships and components of the class diagram. The Activity class was removed, since its main purpose was to store details useful for group activities, which was a removed feature. An event id field was created to keep track of the index of events in the database, and keep Events a modular entity. The Value component was added for the database to keep track of the maximum ids for users and events. A function was added to the Database class to insert an Event and many functions were added to the Matches class to assist with matching users and computing match scores. The newly added relationships are the relationship between the database and the events (since we are now storing events in the database), and the relationship between the database and the value data. The relationship between user and match was removed since they no longer directly interact; instead, the database interacts with the match and the user interacts with the database.

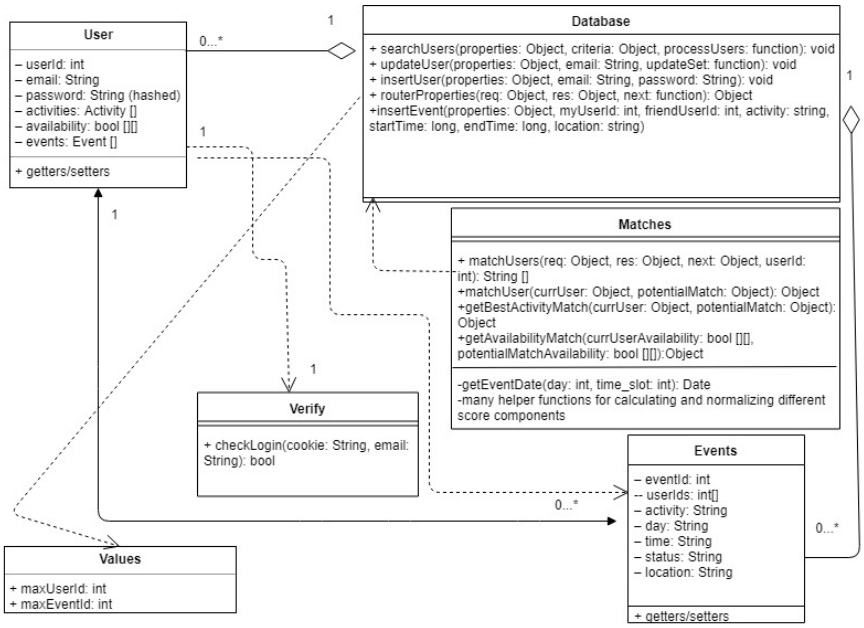


Figure 2: Our final class diagram.

6 API AND IMPLEMENTATION

6.1 Documentation for the Node project

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

6.1.1 *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:

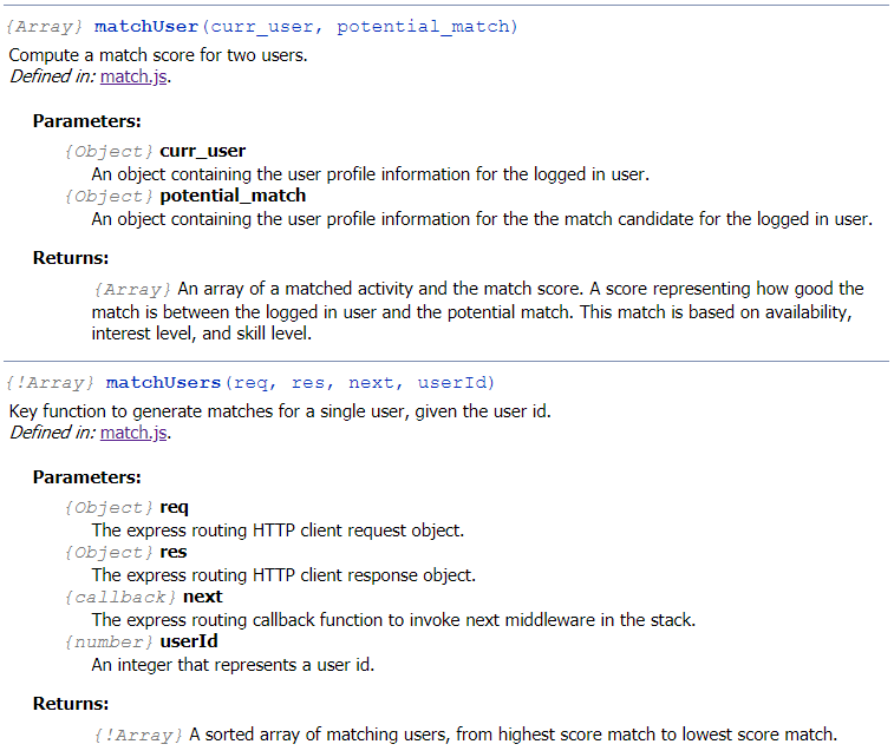
`{Array}` **curr_user_activities**
A list of activities for the current user with interest and skill level scores.
`{Array}` **potential_match_activities**
A list of activities for the potential match user with interest and skill level scores.

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`.

6.1.2 *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:

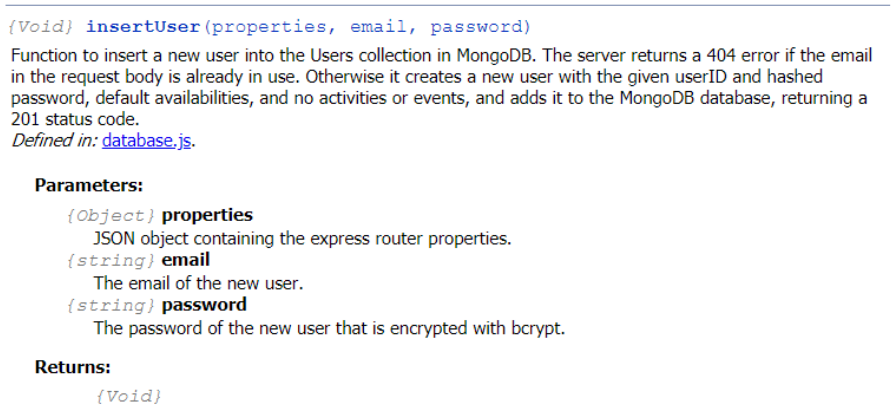
`{Object}` **req**
The express routing HTTP client request object.
`{Object}` **res**
The express routing HTTP client response object.
`{callback}` **next**
The express routing callback function to invoke next middleware in the stack.
`{number}` **userId**
An integer that represents a user id.

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`.

6.1.3 *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:

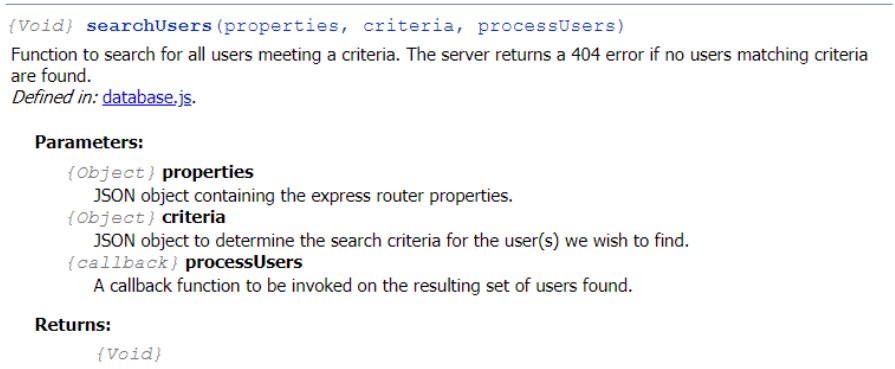
`{Object}` **properties**
JSON object containing the express router properties.
`{string}` **email**
The email of the new user.
`{string}` **password**
The password of the new user that is encrypted with bcrypt.

Returns:

`{Void}`

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

6.1.4 *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:

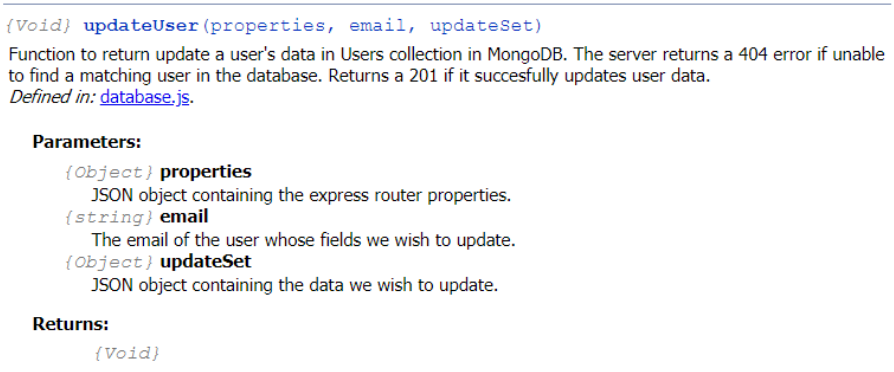
`{Object}` **properties**
JSON object containing the express router properties.
`{Object}` **criteria**
JSON object to determine the search criteria for the user(s) we wish to find.
`{callback}` **processUsers**
A callback function to be invoked on the resulting set of users found.

Returns:

`{Void}`

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

6.1.5 *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 succesfully 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`.

6.2 Documentation for the Angular project

We have also written documentation for our Angular project's API, and have generated a documentation website for it using the public utility `Compodoc`, which generates an Angular-specific suite of documentation using `jsdoc` comments.

6.2.1 *AppModule structure*

The Angular app is composed of the following modules and components.

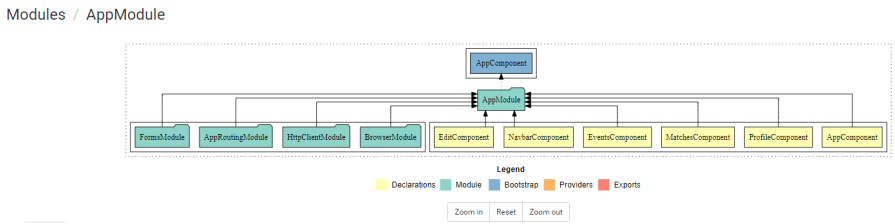


Figure 9: The documentation website's AppModule structure graph, generated by `Compodoc`.

6.2.2 *Sample APIs*

Some of the more important APIs in the Angular app are reproduced here.

otherParticipant

```
otherParticipant(invitedIds: number[])
```

Gets the other participant in an event.

Parameters :

- **userId** : `Array`
  The IDs of the users in the event.

Returns: `number`

notRated

```
notRated(eventId: number)
```

Determines if this event and user have been rated or not.

Parameters :

- **eventId**
  The ID of the event.

Returns: `boolean`

rateUser

```
rateUser(otherId: number, eventId: number, rating: number)
```

Allows the user to rate the other user for this event.

Parameters :

- **otherId**
  The ID of the user to be rated.
- **eventId**
  The ID of the event.
- **rating**
  The rating, from 1 to 5.

Returns: `void`

Figure 10: Documentation for EventsComponent functions, generated by `Compodoc`.

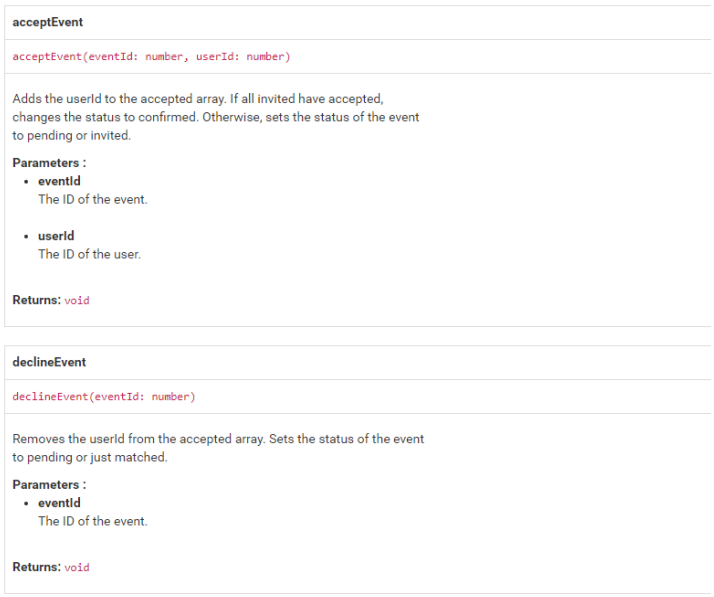


Figure 11: Documentation for MatchesComponent functions, generated by `Compodoc`.

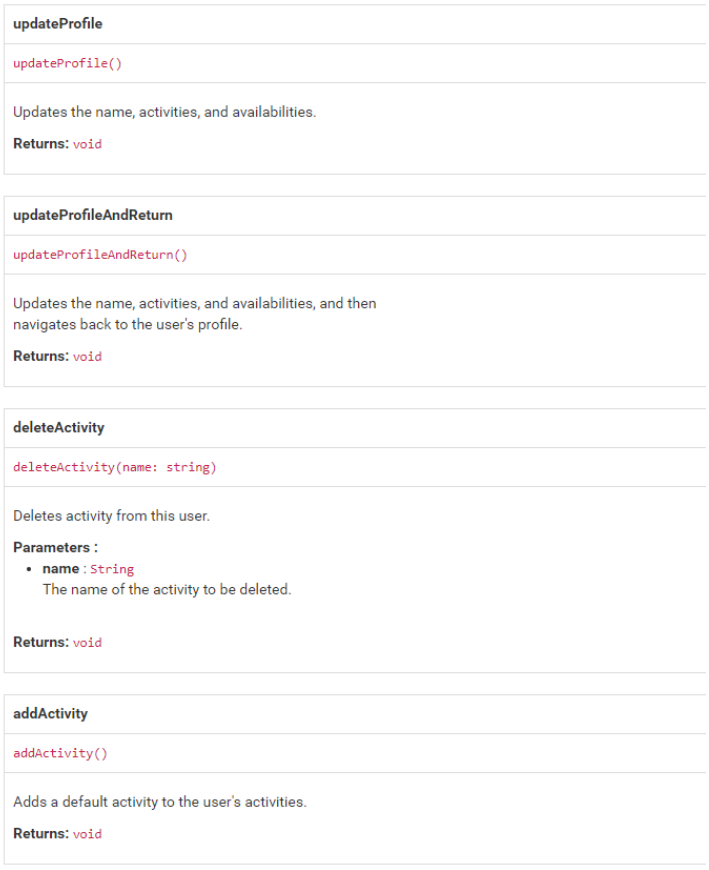


Figure 12: Documentation for EditComponent functions, generated by `Compodoc`.

We describe how these APIs follow the Information Hiding Principle in the Non-Functional Requirements section.

7 SEQUENCE DIAGRAMS

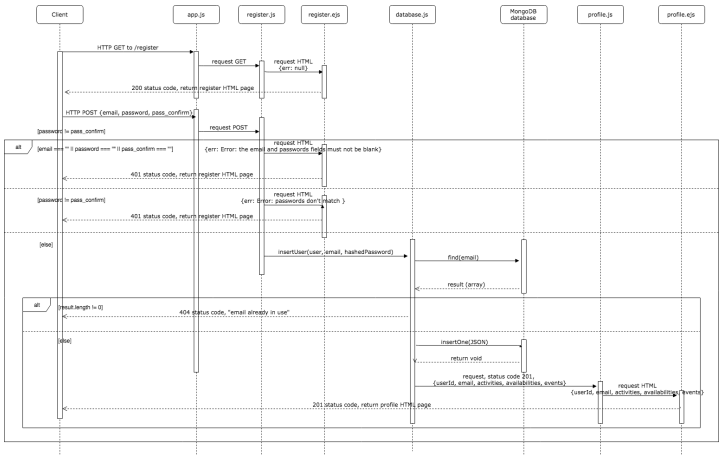


Figure 13: A sequence diagram of the Register module and its basic functionality with handling HTTP GET and POST requests.

Figure 13 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.

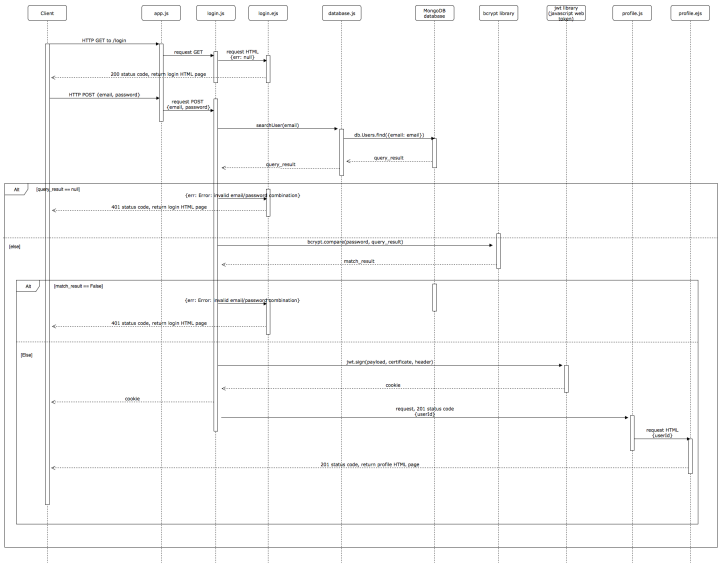


Figure 14: A sequence diagram of the Login module and its basic functionality with handling HTTP GET and POST requests.

Figure 14 is a sequence diagram of the Login 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), as well as showing any error messages if one exists. It simply asks the entry point page to get the HTML display from the Login file and renders it upon receiving it. The POST method is for when the user submits their email and password to request logging into their bActive account. When the user submits their information, it sends a POST request to the server then ask the Database class to first search the MongoDB database for a user with the input email. If a user with the given email doesn't exist, an error status code and message are returned and the login page is returned to be redirected to it. Otherwise, the Database gives the server the user object with the user's hashed password. It then takes the user-input plain-text password and the returned user's hashed password and gives it to the bcrypt Javascript library (which converts the plain-text password to a hashed key and compares it to the fetched hashed password) and returns a boolean if the password matches. If the password doesn't match, then an error status code and message are returned and the login page is returned to be redirected to it. Otherwise, we give the user's email as part of payload to the jwt Javascript library to request a new cookie with `jwt.sign()` function to sign the cookie. The cookie is returned and the server then stores the cookie on the user's browser and then sends a request to the profile page to retrieve the HTML document of the user's profile, which is returned to the client. 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.

8 TESTING

Our testing process consisted of incremental development and testing at each stage. We initially used Postman for testing HTTP GET and POST requests and wrote unit tests in JavaScript for testing the backend, as most of our part B work was backend work. Upon adding the frontend features during Part C, we added tests in Selenium. Furthermore, in Part C, the JavaScript unit tests from part B were enhanced to use the Mocha, a JavaScript test framework for Node.js.

8.1 Postman Tests

The Postman tests are the same as in Part B but are detailed below since this was a crucial step in the development process.

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.

End to end test cases involve interacting with the frontend, making changes, having them persist in the database, and seeing changes reflect back in the frontend. Tests 10-14 are end to end tests run using the Selenium IDE, which is available as both a Chrome and Firefox extension. Note that before running the tests, the database should be reset with the test users by running `mongo < db.sh` inside the `bactive/bactive` folder. The five tests are compiled into a unified test suite, which can be run all at once consecutively using the 'Run all tests' button. The test cases pass when the tests run to completion without errors in logging or stalls. These test cases were used to aide development of the angular server, as the test cases could easily be run to verify correct behavior when changes were made.

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

8.1.1 *Test Case 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.

8.1.2 *Test Case 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.

8.1.3 *Test Case 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.

8.1.4 *Test Case 4: HTTP GET Profile Page with Login Verification*

This was a unit 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.

8.1.5 *Test Case 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.

8.1.6 *Test Case 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.

8.2 Unit Testing using Mocha, JavaScript Unit Test Framework

Unit testing using Mocha, a JavaScript test framework, was critical in testing the functionality. The two tests in part B had initially been run manually with asserts, but in part C they were modified to use Mocha so that subsequent tests would be run even if one fails. Furthermore, a new test case was added in this part and will be described in more detail below.

The test case was run by running the command: npm test. Prior to running the test, npm mocha has to be installed and the package.json file must be identical to that in our Git repository: https://github.com/ryanlo7/bActive. Full details on how to run the test are in the readme. Running the command results in a message indicating which unit tests passed and failed.

The test case consisted of three test functions, which are described in detail below. The fixture involved a set up and tear down before and after running each test method, respectively. The set up consisted of populating two user objects with mock user data. The test methods were then run on this data. Finally, the tear down consisted of clearing the data in these two user objects. The set up and tear down were run so that fresh mock data would be populated each time, in the case that the data gets modified while running the test functions.

8.2.1 *Test Case 7: Unit Test for Time Availability Match*

This unit test case was described in Part B, but was modified in Part C to be run using Mocha. 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).

8.2.2 *Test Case 8: Unit Test for Activity Match*

This unit test case was described in Part B, but was modified in Part C to be run using Mocha. 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).

8.2.3 *Test Case 9: Unit Test for User Match*

This new test case was added in part C. This was a unit test case in JavaScript to determine whether the match object was being correctly populated with activity and event information. Since the matching algorithm is deterministic, given two mock users, the expected best activity match by applying the algorithm is known. Then, we can use the assert statements from the Mocha library to test whether the match information filled by matchUser is the same as the expected information.

8.3 End to End Tests using Selenium

8.3.1 *Test Case 10: Login End to End Test*

This test case uses the browser to access the login page, and proceeds to fill in the username and password before clicking the login button. This test should end on the logged in user's profile page.

8.3.2 *Test Case 11: Edit Profile End to End test*

This test case is contingent on the success of Test Case 10. Once logged in, the user selects the 'Edit Profile' button on the profile page to access the editing page, and proceeds to change the name, delete a favorite event, and edit the availability schedule.

8.3.3 *Test Case 12: Accept Match End to End test*

This test case is contingent on the success of Test Case 10. Once logged in, the user should have a matched event waiting for acceptance/decline from the default database script. The test then clicks the 'Accept' button, moving the event to the 'Pending' category.

8.3.4 *Test Case 13: Log in to other user, accept event, and rate user*

This test case is contingent on the success of Test Case 12. The test logs in to a different account, goes to the Matches page, and should see an event waiting for acceptance/decline in the 'Pending' category (since we accepted from the other user). Upon clicking the 'Accept' button, the event is removed from the Matches page, and is added to the Events page. Navigating to the Events page using the navigation bar, the user can view upcoming and past events. The default confirmed event is a past event, so we are able to give a rating to the other user and submit it, which removes the rating option from the Event page.

8.3.5 *Test Case 14: Registering a new user*

This test case goes to the BActive home page, clicks the 'register' button, fills in all the fields necessary to register a new user, and clicks the 'register' button. This takes the user to the newly created user's default profile page.

9 USER INTERFACE

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

The user is first greeted with the delectable welcome page, shown in Figure 15. She can register as a new user, or log in as an existing user.

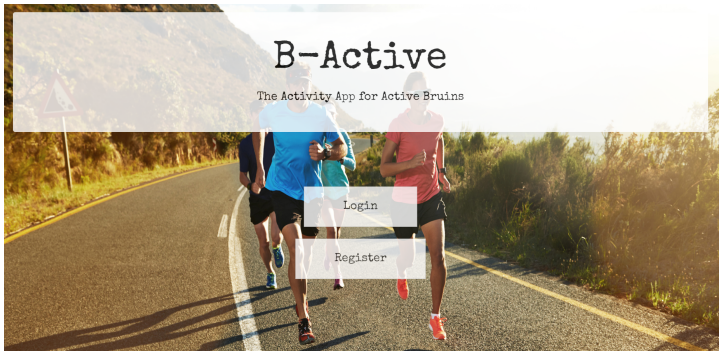


Figure 15: The welcome page of our application as of 3 December 2018.

In our registration page in Figure 16, we allow a user to register by submitting an e-mail address, a password, and a password confirmation in the appropriate fields.

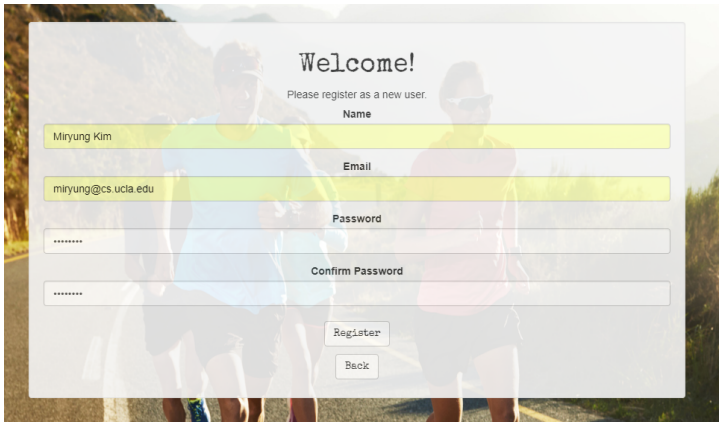


Figure 16: The registration page of our application as of 3 December 2018.

In our login page in Figure 17 on the following page, we allow a returning user to log in to her account, by entering her e-mail address and password.

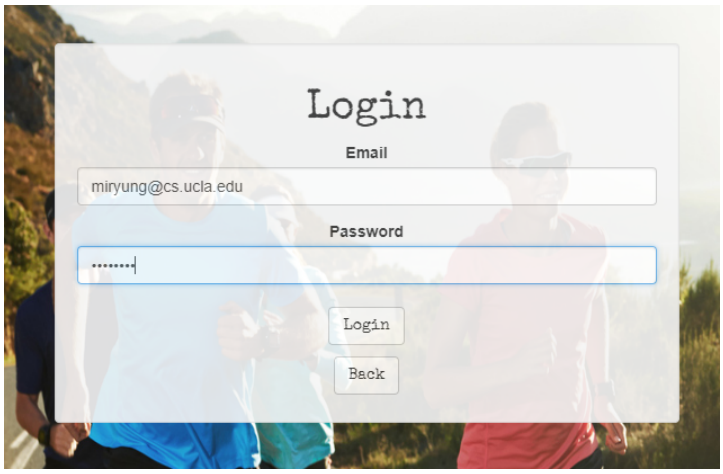


Figure 17: The login page of our application as of 3 December 2018.

Once the user has logged in, she can view her profile in the page shown in Figure 18 and Figure 19. This displays her name, e-mail, rating, activity preferences, and availability preferences.

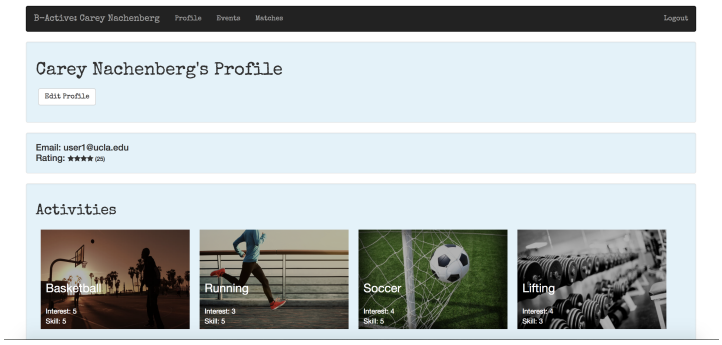


Figure 18: The profile page of our application as of 3 December 2018.

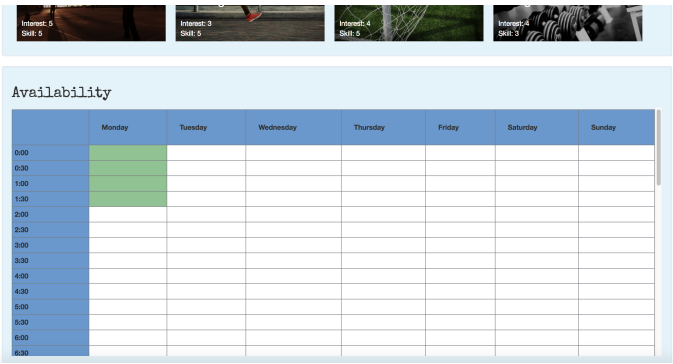


Figure 19: More of the profile page of our application as of 3 December 2018.

Once the user has logged in, she can edit her profile in the page shown in Figure 20 on the next page and Figure 21 on the following page. She can alter her name, as well as select preferences for her activities and availabilities.

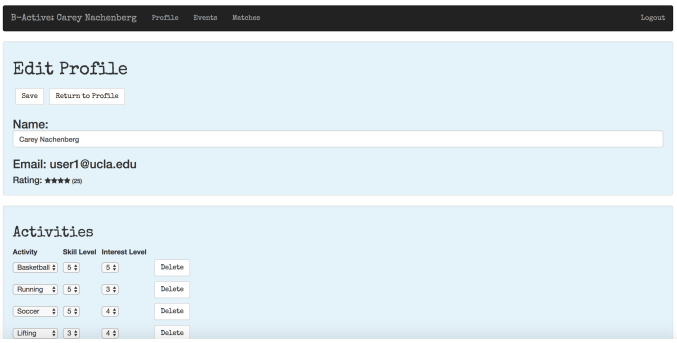


Figure 20: The profile editing page of our application as of 3 December 2018.

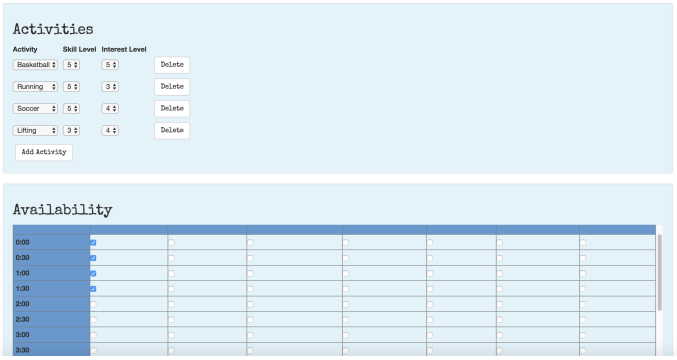


Figure 21: More of the profile editing page of our application as of 3 December 2018.

She can see a list of the users she has been matched to, in the events that the application has automatically generated for her, in Figure 22 and Figure 23 on the following page.

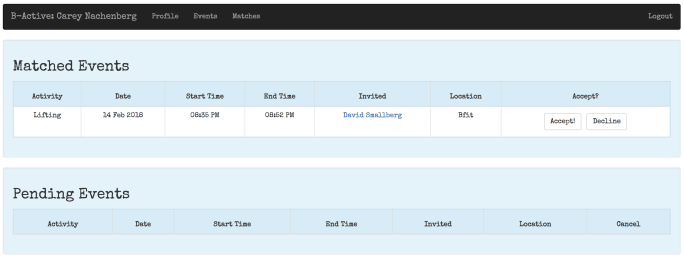


Figure 22: The matching page of our application as of 3 December 2018, with one matched event.

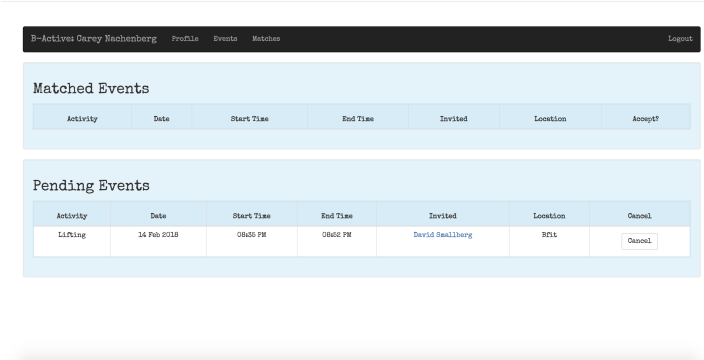


Figure 23: More of the matching page of our application as of 3 December 2018, with one pending event.

The user can also see her list of upcoming confirmed events, in the form of a friendly table as shown in Figure 24.

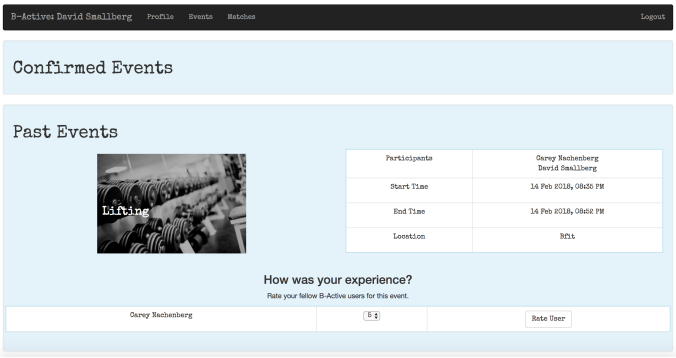


Figure 24: The events page of our application as of 15 November 2018, showing a confirmed event.

10 CONTRIBUTIONS

10.1 Krishna

Krishna worked on connecting the front-end to the back-end of the matching page, to render all new event information as well as create new events in the database. For this she wrote functions that contributed to the database class to easily access and modify the new collections, and added documentation in JSDoc format as required. Along the development process she contributed to redesigns of parts of the database as development needs changed. She also worked with Ramya on fixing the logic and routing of the top-level matching algorithm and consolidating new matches' information to pass on to the front-end in a way that abides by information hiding. Later she worked on fixing the register page such that a new user could successfully register and be redirected to their profile. For testing she researched using Istanbul and Jasmine to generate test info, and aided Ramya in testing matching functions through npm.

10.2 Eddie

Eddie wrote the majority of the API between the backend and the frontend; that is, the backend code that returns JSON objects and handles server-side interactions with MongoDB that the Angular front end used, requiring a continuous commitment due to the somewhat unpredictable and evolving nature of the frontend requirements of the API. In addition, Eddie set up many of the Angular components and wrote the routing component to handle the interactions between the different components of the profile page. In addition, he provided assistance to various group members, providing critical input and support in terms of nailing down ideas into code and debugging the various issues that cropped up.

10.3 Eric

Eric primarily worked on wondrous affairs of the front-end. In this capacity, he wrote a majority of the CSS stylings for the Node project, as well as the corresponding .ejs pages. In the Angular project, he worked on some sections of the API, the events component, and the matches component, and provided the majority of the CSS stylings in here. Some of his proudest accomplishments include the API calls necessary to power the front-end of the ratings system, the displaying of users' names in the events and matches page, and the utterly delicious welcome pages. He also wrote all of the documentation for the Node project, as well as a majority of the documentation for the Angular project, and generated the documentation websites with JSDoc and Compodoc. Finally, Eric worked on the several sections of the project report, proofread it, and typeset the whole report in LaTeX.

10.4 Ryan

Ryan worked on an amalgam of the different features. He first worked on the front and back-end parts of the profile page; he implemented the AngularJS side to request the ExpressJS server to pull the user from database and display the user's fields on the HTML display page, followed by the

CSS work to make the page look nicer, as well as the similar Edit Profile page to allow users to edit their profile information. He also implemented parts of backend of the match and events page to allow AngularJS to make requests to the ExpressJS server to pull other users, events, etc from MongoDB database (which would then be sent to display those objects or their info on the front end). He then modified the ExpressJS server to perform a redirect to the AngularJS code upon users registering or logging in. Lastly, he also rendered the user rating system so that the 5-star scales are displayed on each user's profile.

10.5 Ramya

Ramya worked on modifying the matching algorithm to also schedule ideal events, creating the JavaScript unit tests using Mocha, and writing most sections of the report. She first worked on adding functionalities to the matching algorithm. This involved adding the feature of finding the date and time for the event, using the user data and then using moment.js to find the precise dates. She then worked with Krishna on the matchUsers function to run the matching algorithm on all users in the database and rank the matches. She then added a unit test for the match algorithm and refactored the unit tests to use the Mocha JavaScript test framework and follow the JUnit test fixture style described in class. She wrote documentation in JSDoc format for the match module and the JavaScript unit tests. Finally, she worked on the motivation, user benefits, non-functional requirements, use case diagram, class diagram, and testing sections of the report.

10.6 Nicole

Nicole worked on the implementation of many of the Angular components, including profile, edit profile, matches, and events. After assisting Eddie with the creation of the API, she helped set up the user services so that the Angular components could obtain information from the database. She then set up the profile component to display all the user information, and added the edit functionality to the edit profile page. She also organized and displayed the matched and pending events on the matches page, as well as implement the backend for accepting and declining events. She also assisted Eric with the display of the events page by sorting the events into upcoming and past events, as well as making the rating feature available only to events that have not been rated already. The Selenium test cases were also created by her.