My project is The Workout Tracker.

The value that this software provides to users is that The Workout Tracker helps users stay on top of their workouts by allowing them to log workouts and exercises done on any given day.

The main features this software provides include:

One, allowing weights done for an exercise to be recorded in pounds, but also show the weight in kilograms for users who may be more comfortable with another units of measuring. This is done by Microservice A, which was created by one of my teammates.

Two, generating a random exercise based on a muscle group if users are not sure with what they want to do for a workout. This is done by Microservice B that I implemented.

Three, allows users to update an exercise’s attributes’ values, such as name of the exercise, number of sets done for an exercise, and so on. This is done by Microservice C that I implemented.

Fourth, ensures users enter valid email addresses when sending a message to the software’s help team requesting for assistance with the software, so that the help team can get in touch with the users, as appropriately. This is done by Microservice D that I implemented.

As you can see from what I am showing, the main application started in one terminal. Similarly, microservices A, B, C, and D were all individually started in different terminals.

What I am showing now is how the main program communicates to the four microservices.

1. Between the main program and Microservice A, when React loads the exercise page of a workout, then the React page makes an axios request to call microservice A’s API in an iterative loop based on how many exercises have been logged for the workout (show code and URL in fetchExercises block: `http://localhost:8080/api/convert/weight?value=${eachExercise.weight}&fromUnit=pounds&toUnit=kilograms`). In the HTTP request, the eachExercise.weight parameter is passed the data of the weight done for an exercise in the workout, which is in pounds by default as designed by the database structure. Microservice A then parses the HTTP request of the exercise, including the exercise weight value, and converts the value into kilogram based on some calculations, as shown in microservice-a -> api-conversion -> src -> main -> java -> ConversionController.java file (show code sections for switch case pounds to kilograms). Then microservice A sends the data to the React page as a JSON object, and the value gets added to a list. This process is repeated until all exercises’ weight values have been processed. The list of converted values eventually get processed and mapped out to dynamically show the table of exercises, including the weight of each exercise in pounds and in kilograms, as shown in the following code block: <tbody>… <th>{item.weight} lbs ({item.exerciseInKg} kg)</th></tbody>
2. Now, between the main program and Microservice B, on the same exercise page for a workout, the React page makes an axios request to call microservice B’s API (show the API code section: `http://localhost:3002/exercise/${muscleGroup}`). When a user chooses a muscle group, then React passes the chosen muscle group to the API’s parameter ${muscleGroup}, and this gets processed by Microservice B. Microservice B then gets a random exercise from an external API and sends to React information on the random exercise as a JSON object, and by calling .name and .instructions on this object, the name of the exercise and instructions on how to do it gets rendered as shown in <div></div> block with random-exercise-generator-description.
3. Additionally, between the main program and Microservice C, when the user wants to update an exercise and clicks on this icon to update the exercise, then the user is taken to the update page (show UpdateExercise..js in src -> components -> crud). When the user enters in the values to be updated, and the submit button is clicked, then React makes an axios call to microservice C’s API (show code section in handleSubmissionOfUpdate and link: `http://localhost:3003/update-exercise/${specificWorkout}/${sessionexercise}`, exerciseAttributes). Microservice C then processes the values and sends a SQL statement to the database to update the specific exercise’s attributes’ values. If the exercise was successfully updated, then a status code of 200 and result of the SQL code update are sent back to React. The update page will check to see if the update request’s status code is 200. If it is 200, then the user will be directed to the exercise page again with the values updated. If the status code is not 200, then the user will be taken to an under construction or error page to inform the user that something went wrong.
4. Lastly, between the main program and Microservice D, when the user is on the Contact Us page, and enters information and click on submit, then the React page will make an axios request to call Microservice D’s API (("http://localhost:3004/contact-us/response", contactUs). The microservice then processes the request, and sends back to the React page an integer of 1, which means the email the user entered exists, or an integer of 0, which means the email the user entered does not exist. React will then determine if the email is valid by the integer that Microservice D provides. If the email exists, then React renders information that shows the submission for contact us page was successful. If the email does not exist, then React renders information that shows the email address was not valid to inform the user to provide a valid email address.

As you can see, the main program does not import any microservice code, but instead, calls the API for each microservice.