The following API can be accessed at `https://mysqlcs639.cs.wisc.edu`

Route	Auth Required	Token Required	Get	Post	Put	Delete
/login/	✔		✔			
/users/				✔		
/users/`<username>`		✔	✔	✔	✔	✔
/meals/		✔	✔	✔		
/meals/`<meal_id>`		✔	✔		✔	✔
/meals/`<meal_id>`/foods/		✔	✔	✔		
/meals/`<meal_id>`/foods/`<food_id>`		✔	✔		✔	✔
/activities/		✔	✔	✔		
/activities/`<activity_id>`		✔	✔		✔	✔
/foods/			✔			
/foods/`food_id`			✔			

Auth and Tokens

For this API, users need to provide credentials in order to access information specific to themselves. They get these credentials by requesting tokens, which are short-lived codes which tell the server that you are who you are saying you are, without having to provide a username and password each time. The steps to get these tokens are outlined below.

Signup

This can be done with a `POST` request to the `/users` route. You will need to tell the API a bit about the user. You should provide this data in the message body (stringified) in the following form:

```
{username:<str>,                // Required
 password:<str>,                // Required
 firstName:<str>,               // Optional
 lastName:<str>,                // Optional
 goalDailyCalories:<float>,     // Optional
 goalDailyProtein:<float>,      // Optional
 goalDailyCarbohydrates:<float>, // Optional
```

```
  goalDailyFat:<float>,          // Optional
  goalDailyActivity:<float>      // Optional
}
```

Only the `username` and `password` fields are required. Don't worry about the other ones for creating a user, as they can be updated later with `PUT` requests.

If the user is successfully created, you will receive a positive message back from the server. You will then need to log in with that user.

Login

You can do this via the `/login` route with a `GET` request. You will need to send the username and password in the authorization header using the header `Authorization` with value 'Basic {base64enc(username:password)'. e.g. 'Basic bXl1c2VyOnBhc3MxMjM0' for a hypothetical user 'myuser' with password 'pass1234'.

You will receive back a token that you can use to access information from the API. The token you receive can then be added in the `x-access-token` header.

User

Users cannot query `/users`, since that would involve exposing all the other users' data. Instead, they must get/modify the information for their user separately. They do this by using the `/users/<username>` route, where `<username>` is filled in with their actual username. Suppose your username is `Fred639`, then you could fetch (`GET`) `/users/Fred639` to get the information about yourself, but only if you provide the right token. Likewise, you can `PUT` to `/users/Fred639` to modify your goals, by providing the changes in the form of a `json`. You can modify the following fields:

- `pasword`
- `firstName`
- `lastName`
- `goalDailyCalories`
- `goalDailyProtein`
- `goalDailyCarbohydrates`
- `goalDailyFat`
- `goalDailyActivity`

Additionally, you can delete unused users using the `DELETE` method on the `/users/<user_id>` route. As articulated in the Final Notes, please be a good citizen and clean up after yourself.

React Native 2 α API Information

This is provided so you can get a sneak peek about the rest of the API!

User Meals

Each user can track their meals across days. These are interacted with using the `/meals` routes. Despite not including the user in the route, you will only be able to retrieve the information for your current user, since the token tells the API who you are. Unlike `/users`, you can request the full set of meals using a `GET` on `/meals`. This will return an array-like object like this:

```
{meals:[
 {id:<int>,
  name:<str>,
  date:<isostring>
 },
 {id:<int>,
  name:<str>,
  date:<isostring>
 }]
}
```

You can also access an individual meal by `meal_id` (`id`), using `GET` on `/meals/<meal_id>`. This will return the single meal data object:

```
{id:<int>,
 name:<str>,
 date:<isostring>
}
```

Posting to `/meals` creates a new meal. If not specified in ISO format, the `date` will default to the current date and time. You cannot specify the `id`, only the `name` and `date`. Using the `PUT` method is possible for the `/meals/<meal_id>` route, such that you can modify the `name` and `date`. Using `DELETE` on an individual meal removes the item (and all associated foods).

User Foods

You may notice that the meal doesn't have any food data. That is because they are accessed with a deeper route, `/meals/<meal_id>/foods` and `/meals/<meal_id>/foods/<food_id>`. These behave similarly to the meals, where using `/meals/<meal_id>/foods` returns the list of all foods associated with that meal:

```
{foods:[
 {id:<int>,
  name:<str>,
  calories:<float>,
```

```
  protein:<float>,
  carbohydrates:<float>,
  fat:<float>
 },
 {id:<int>,
  name:<str>,
  calories:<float>,
  protein:<float>,
  carbohydrates:<float>,
  fat:<float>
 }]
}
```

A single food can be accessed by `id`, using `/meals/<meal_id>/foods/<food_id>`, returning the single object:

```
{id:<int>,
 name:<str>,
 calories:<float>,
 protein:<float>,
 carbohydrates:<float>,
 fat:<float>
}
```

Like meals, you can modify all attributes with `PUT`, other than the `id`. Using `DELETE` on an individual food removes the item.

User Activities

By now you should see the pattern. That is the great thing about RESTful APIs! They are consistent. User activities follow the same pattern as meals, but don't have further routes like foods. You can use the routes `/activities` and `/activities/<activity_id>` in conjunction with relevant `GET`, `POST`, `PUT` and `DELETE` methods. The data structure of individual activities is as follows:

```
{id:<int>,
 name:<str>,
 duration:<float>, // Minutes
 date:<isostring>,
 calories:<float>
}
```

Using `GET` returns a list of the above objects, like in meals and foods.

Food Library

As a convenience, we have provided a set of basic foods that you can use to create and add to meals. These readonly data can be accessed with `GET` methods to `/foods` and `/foods/<food_id>`. They appear similar to foods in meal, with an additional attribute of `measure`, which tells you how much of each the

nutritional values are associated with, and what the grouping name is called. For example, bread comes in `slices`:

```
{id:2,
 name:"whole wheat bread",
 measure:"slice",
 calories:69.0,
 protein:3.6,
 carbohydrates:12.0,
 fat:0.9
}
```