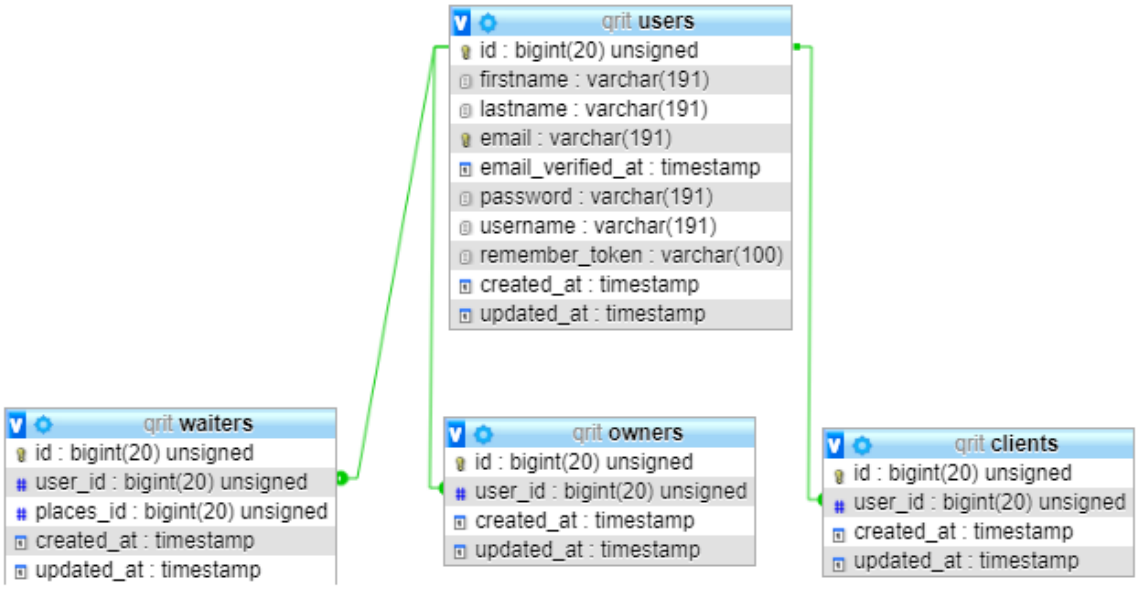
Phase2: Database details

First details for the second phase of the application will start by describing the structure we have built for the tables in database and the way are connected. For creating the tables, we used migrations in which we specified the type and the constraints (foreign key to other tables). We have created separate mysql users for our usecases: the user which performs the migrations has all rights granted for the tables that belong to our database, while the user which provides the connection for interacting with the app only has CRUD rights for the records inside our tables.

In the application we have separated the roles for a user in **client**, **owner** and **waiter**.

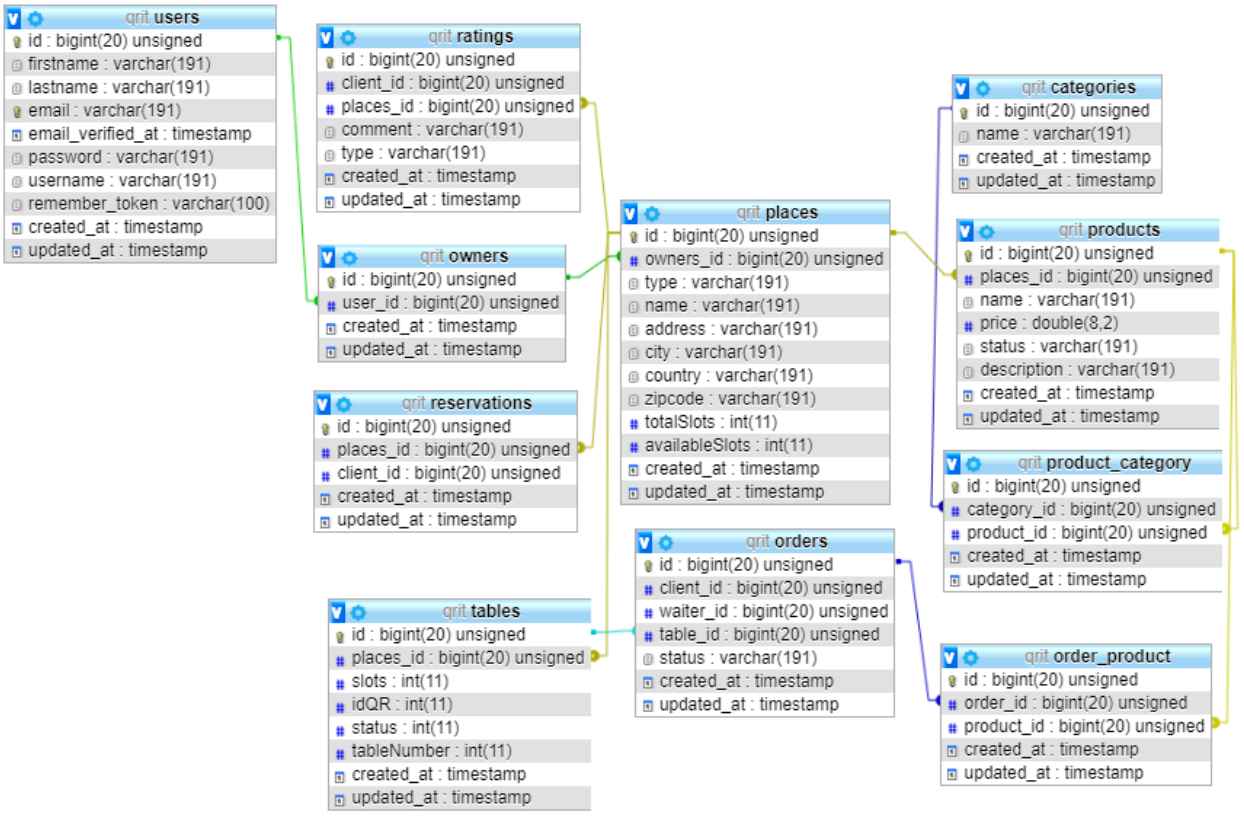
The attached image contains a representation of the structure for users in our application. The table **users** is considered a super table that holds the common information from the other type of users which is firstname, lastname, email, username, password. Other types of users are owners, waiters and clients.



In the attached image are represented the relations between businesses and the way are managed in our application by keeping track of the orders, reservations and the clients that take advantage of the provided services.

As you can see in the attached image, we have defined pivot tables that help us managing the many-to-many relations. This kind of pivot tables are product_category and order_product.

After creating the database migrations we built the API routes for the application. Each route for a resource has defined five actions: ‘index’, ‘show’, ‘store’, ‘update’ and ‘destroy’, which are mapped to http action verbs GET, POST, PUT, DELETE. The resources in our application are the places, orders,



products, reservation, clients, users, waiters, owners. Additionally, we will be using resources for implementing the authentication and authorization requirements, with regards to our discussion in class (URLs will not denote actions, only resources).

The third aspect for database service is the way we built the API, by creating controllers for each defined entity. Each Controller contains the methods **index, store, show, update, destroy,** that were specified in the API routes part that we previous stated.

The **index** method, mapped over GET for the whole resource, will list all records that match the specified resource type.

The **store** method, mapped over POST for the whole resource, has as parameter a request from user side and will save it if it is valid. In order to validate it, we have used validators to check if the required field was provided with the right type of accepted input.

In the **show** method, mapped over GET for a certain id, we search the the item in database and if we get a result, we return the item with 200 status code. Otherwise, if no results are found, the 404 code will be returned.

In the **update** method, mapped over PUT for a certain id, we search the enclosed entity's id in our database. If we don't find it, and the user has the right to create the specified resource, we shall create it. Otherwise the request will be denied. Should the entity's id be found in our database, the method will provide a full update of the specified resource.

In the **destroy** method, mapped over DELETE, we search the id in database and if we get a result, we delete the entry.

The ids are mostly server side auto generated ones, which is why it makes sense for POST to be used for creating resources. We took into consideration composite keys, but in our case, they weren't favorable, since any pairs would eventually cause duplicate entries in our tables.

All the methods return standard status codes, according to REST principles. Also, any method that throws an exception will have its contents logged before returning the according status code, as discussed during one of the courses. The log files would help us take action against similar security threats in the future. Example log files from our tests are located in the **storage/log** folder.