# Database Setup

To connect my database to a web application, I first needed to host the database so it could be connected to from my web application. I used a Linux virtual machine (VM) and *XAMPP*, local web hosting software, to get my database on my local network.

Graphical user interface, text

Description automatically generated

Once my database was on the local network, the private IPv4 of the system hosting the databased needs to be determined, so other systems/software can properly connect to the database. By running the CLI command *ifconfig*, a list of current configured networks is displayed containing information for each network, including the private IPv4 of the hosting system.

A screenshot of a computer

Description automatically generated with medium confidence

By typing the private IPv4 into a browser (192.168.1.XXX), and appending */phpMyAdmin* to it, I can access the database’s contents via *PhpMyAdmin*. Through here, I added an additional user, besides the default *root* user, allowing me to use the new user’s database credentials in my web application.

Graphical user interface, application

Description automatically generated

# Web Application Setup – Node.js

Now that my database is hosted on my local network and I have database user credentials, I can connect to the database using Node.js and the dependencies *fs*, *mysql*, and *knex.js*. By using fs (file system), I can read a local JSON file into my JavaScript code, allowing me to store the database authentication credentials safely on my local system, therefore establishing a layer of abstraction for database authentication.

Text

Description automatically generated

The JSON database credentials are then passed through a created *knex* object, establishing a connection to the specified database from the JSON, using *mysql* as the client. In this case, *knex.js* is a third-party library for abstracting the construction of queries. So, *knex.js* will be the main way I handle my SQL queries.

In order to execute dynamically constructed SQL queries using *knex.js*, I utilized *express*, a backend Node.js framework, to development a REST API, using HTTP requests to query data. The developed API endpoints are strictly HTTP GET requests due to the nature of my TangoTwo project (querying data).

To establish a REST API connection, an *express* app is created and utilizes *cors*, an *express* middleware for configuring access to resources of HTTP requests, which then listens on the chosen port, 3000, or a previously determined environmental variable port.

Text

Description automatically generated

# REST API Implementation

From here, the *express* app is used in conjunction with implemented endpoint routes, defined in a separate file, appRoutes.js, under a model-view-controller (MVC) architecture.

Text

Description automatically generated

Each API endpoint corresponds to a specific route, associated with entities from the database. When these routes are accessed as HTTP GET requests, they execute their given unique function, received from the controller file, appController.js.

Text

Description automatically generated

The controller handles the data/errors received from the model entity functions and propagates the data and/or errors up as the handling is finished. Both data and errors are sent as the body of the HTTP result, finalizing the GET request. The model entity functions are defined in the model file, appModel.js.

Text

Description automatically generated

The model requests data from the database by constructing dynamic SQL queries using *knex.js*, as mentioned earlier. As of now, only “GET ALL” and “GET ONE” requests are implemented for each database entity. However, “GET ALL” requests do allow for users to send URL queries with their HTTP GET requests, making SQL queries contain additional parameters when constructed, such as filtering, paginating, sorting, and ranges.

When accessing the running REST API, by entering in the loopback address and its designated port (127.0.0.1:3000) in the address bar of a browser, routes can be appended to the address to execute the defined GET requests, such as *127.0.0.1:3000/players*.

Text

Description automatically generated

Query results are returned as JSON and displayed as the body of the HTTP result. I am currently using a JSON prettier plugin, called *JSON Viewer*, to reformat the JSON result to human readable JSON.