Immortal Blade API

API Calls

API calls are generally formatted as follows:

hostURL/<router>/<function>/<tableName>/<ID>

Router
- The ExpressJS router that the API call is being sent to. At the time of writing this document, the only route that has been implemented is */test*. More routes will be added as the front end continues to be developed.

Function
- The action being performed. This determines what SQL will be generated for the query. Examples include *view, edit, delete,* ext…

Table Name
- The database table that the action is being performed on.

ID
- The ID of the entry that the action is being performed on. Only included if the action is being performed on a particular entry in the table.

Payloads

Data from the database is retrieved as JSON in the following format:

```
{
        0: {
                Field: entry,
                Field: entry
        }
        1: {
                Field: entry,
                Field: entry
        }
        ....
}
```

Each row is indexed in the response JSON by an integer, starting at 0. This is true even if the request only asks for a single entry.

To access a particular entry in the front end code, the following format must be used:

```
const <xhttp request variable> = new XMLHttpRequest();

<xhttp request variable>.open('<method>','<API route>');
<xhttp request variable>.onload = function () {

        // The code inside the onload function will not run until the database has sent the response to the front end script
        let <data variable> = JSON.parse(<xhttp request variable>.responseText);
        let <entry variable> = <data variable>[<row index>][<field name>];

        // Do things with the response data here
}
<xhttp request variable>.send();
```

There may be times where data needs to be read from the URL on the front end script *(usually the ID of a row that is going to be edited in some way).* In such cases, the data from the last section of the URL can be retrieved by importing **parseData** from **'./parseURL.js'** and storing the return value of **parseData(location.href);** as a variable.

ParseData() is designed to interpret the last section of the URL *(formatted as '<field name>':<value>-'<field name>':<value>)* as JSON by replacing the single quotes with double quotes, delineating entries with '-', and adding opening and closing brackets. The function then returns a JavaScript object with all of the data in it.

(Note: In the URL, field names and string values will need to be surrounded by single quotes, non string values should not be surrounded by anything)

Routing

There are two npm packages that need to be imported at the top of the file
- **express**
- **path**

The **'../database-data/connection'** also needs to be imported, and a variable needs to be declared with the value **express.Router()**

This is a template for what a route might look like

```
router.get('/function/:variable, function (req, res) {
        Let input = [ res.param.variable ]
        let sql = 'SQL QUERY WHERE variable = ?';
```

```
        pool.query(sql, input, function (err, result) {
                if (err) throw err;
                res.json(result);
        });
});
```

This route would be accessed with website.com/**function**/<*some variable*>

- **/function** can be anything. The name itself is arbitrary, and it is how the front end XML requests (seen above) will access the API. Good function names for the Immortal Blade project might include **getNames** and **getAll**

- **/:variable** is much like a parameter passed to a function. This could be the name of the table being accessed, or the ID of a particular table entry. They can be accessed using **req.params.<variable name>**

For example, **/get-unit/:tableName/:id** might be used to get data from a particular unit (using it's id) from the table specified by the user (either characters or enemys in this project)

Or, **/get-actions/:unitTable/:actionTable/:id** could be used to get all actions (from either attacks or supports) associated with a particular unit (either characters or enemys)

Queries

The front end API should only be using SELECT queries, since the user will not be allowed to edit any database entries.

Also, to prevent any SQL injection from the user, an array if inputs can be declared for the **pool.query** function

The SELECT statement for getting data from tables joined by a junction table is important to understand, and goes as follows.

let **junctionTable** = **unitTable** + "_" + **actionTable**;
let **input** = [**actionTable, actionTable, actionTable, junctionTable, unitTable, id**];
let sql = "SELECT * FROM rpg_project_db.**?** WHERE id_**?** IN (SELECT id_**?** FROM rpg_project_db.**?** WHERE id_**?** = **?**');"

pool.query(**sql**, **input**, **function** (err, result) …

The junction table contains an action id and a unit id for every action that unit has. So to get the actions for that unit, SELECT the action id from the junction table WHERE the unit id matches

the input **id**, then, SELECT from the action table WHERE the action id matches the action id retrieved from the previous query.

This would set '**result**' to contain every action (either support or attack) from the given unit (id) in the unit table (characters or enemys), which could then be sent to the front end using res.json(**result**);