Simple worth through of how the api will handle a request.

First the request will come in and get handed off to the relevant method so /newLobby will call the newLobby() function in the code.

The request will then be decoded. This is done using one of Golang’s standard libraries “encoding/json”. Simply pass in the data structure and it will parse it into a golang map type which can then be used.

Once this is done a check will be done with the cookie field to check that it is valid. If so process will continue otherwise it will terminate, send a failed message.

Before continuing a check is made to ensure that the parsed game and creator values exist in the database in there respective table. An error will be thrown if not.

A check is then done to see if there is already an existing lobby with the creator from the request. If does exist that lobby is considered to be unsued and will be removed.

Next the 6 digit unique number needs to be created so we use the golang rand library to generate between 1 and 999999 so up to 6 digits and if number doesn’t meet this requirement 0’s are appended to the front.

The rest of the params in the request are then checked that they exist so for lobby it is creator game and size. A method called required params then appends to the querystring when required using a pointer and creates a args array which is what is passed to the db function as the params.

All database func are then executed using two libraries one is the driver for setting up the connection and handerling any errors which is a github contrubtion. The other is a golang standard library called ‘database/sql’ which makes the request to the already connected database using the driver.

The results of this is then handled and the response is sent accordingly. A fail sends a fail message constructed with a simple map method and parsing back to JSON string. A success appends the relevant data and constructs a different map parsed in the same way using the “encoding/json” library.

This is then sent in the response from the according request.

When creating the backend there were a few challenges. The first one was the previous implementation which was in node. The problem with this was when the backend would then want to communicate with the Engine it would have to establish a connection through something like TCP or another mode. This would slow everything down and added an extra unnecessary level so the decision was made to move it into golang. This however mean rewriting everything that had been done and that we now had to learn and move everything into golang. The time constraints for learning and making this was a challenge.

Challenges within the scope of the backend were ensuring that any potential error that could occur would always send an error/ fail response and not crash and return nothing. The security measure of adding a salt added an interesting challenge and step in retrieval and adding users. Another complexity was an issue with the access control headers which caused the browser to block the requests. This was solved by an override of the mux(“gorilla/mux”) contributed libraries so that the Backend would work with Openshift. Once the application was at a useable state and was starting to be used by rest of the platform it was croo-shal that any additions didn’t break existing functionality and was kept consistent. If something did break it would have to be fixed immediately to ensure that it did not slow development elsewhere.

Anon user