San Francisco State University
CSC 667 Spring 2017

Term Project
Tank City

GitHub Repository:
https://github.com/SFSU-CSC-667/term-project-spring-2017-ivm_

Heroku:
http://tankcity.herokuapp.com

Team Members:
Alex Aichinger
Maroun Abi Ramia
Vivian Lee
Ivan Yu

Table of Contents

Architecture

This project uses the MVC Design Pattern in order to separate our logic from the user interface. Our View consists of all pug files for the user, which also utilizes javaScript for handling requests, and jQuery for various functionalities. Our Model contains functions necessary for accessing the database to retrieve, update, or insert information for each table. Our Controller is our routes folder, whose files contain API calls - HTTP GET and POST requests.

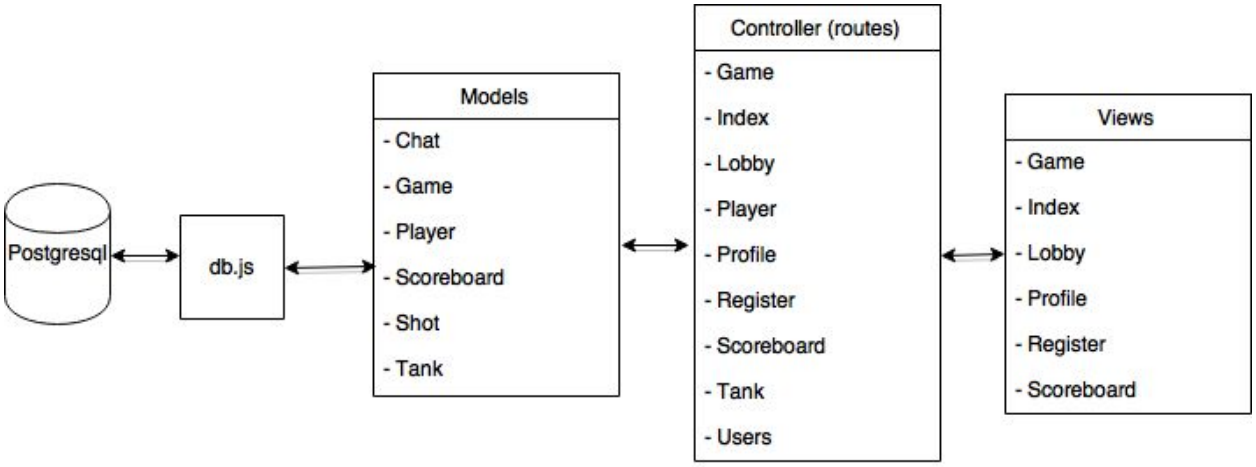


Diagram 1: Software Architecture

Diagram 1 follows the MVC software architecture with additional refactoring to keep the organization and structure of our program easily refactorable. Our models have a helper class called db.js, which creates a connection to our postgresql database and is able to handle multiple queries. It uses the “pg” library in order to make these functionalities work. Our database also uses a thread pool to processing a certain amount of queries without hanging. After a query is done, it is released from the pool, creating room for another query.

Our models all use db.js to be able to access our database. The controllers, otherwise known as the routes files in this technology stack, is used to update the view and models accordingly. This gave our controller the ability to focus more on update the views without having to deal with complex queries.

The views are all implemented as pug files. Some of the views have more than one controller depending on how much information is needed to use. Views are also connected to the public/js and public/css files to provide styling and javascript functionality.

Problems Encountered During Implementation

Organization

We ran into a lot of issues trying to organize the structure of our project to keep tasks separated in its own class. For example, we did not want to have too many calls to the database in our routes classes, since the purpose of routes is to handle the redirection of our views. Our solution was to follow the MVC pattern where the routes would represent the Controller. We created all our Models that would connect to the database and handle any queries we needed, which would also be called from the Controller. The controller would then update the view appropriately based on modifications to the data.

Database Issues

One of the major problems that we encountered in our project was frequently experiencing hanging from the website after interacting with it for a short period of time. Once we created a connection to the database and were able to process query requests, our thread pool would fill up and cause our website to hang.

The first solution we came up with was storing data in a global variable so we wouldn't have to make so many query calls but that didn't solve anything since it was still hanging, so we undid this solution. After spending time debugging the project, hunting for the issue, we found out that the reason our thread pool was not able to query a number of consecutive requests was because we were not releasing the previously executed queries from it. Therefore, it would reach its maximum number of requests and not be able to handle any additional requests. The solution was to release the query from the pool once the request was done, and our website was no longer hanging.

Authentication

Integrating passport.js was a difficult task because the documentation on it was lacking and not very clear. It was also a challenge trying to figure out where to add the authentication and how it would work. We started out trying to integrate passport-http-bearer, but that created problems for us beyond what the documentation or stackoverflow would provide. We decide to switch to passport-local, which was more simple and made things easier to integrate to our project.

Difficulties

Working in a group can quickly lead to various difficulties. A difficulty that we seemed to face constantly throughout the project was merging code. This made us realize the importance of a common coding style among the team members.

The way our group's github repository was structured was to have everyone in the group have their own separate branch to work on. However, separate features, which different group members are assigned to work on simultaneously, tend to rely on the development of the same file. When such features require different members from the group to create large bodies of codes, on the same file, pulling from someone else's branch resulted in many merge conflicts. In such circumstances, the changes made by one group member on the file may unknowingly negate or override the changes made by another group member. Such circumstances required the group to communicate with one another to achieve a solution, and prevent the accumulation of unused or overwritten code.

There were also times when large chunks of codes had to be moved around in order to implement a specific feature of the application. For instance, on the initial implementation of the game, the group had implemented it solely to display the tanks, backgrounds and objects in the game. However, we noticed that the backgrounds and tanks were different for the screens of each player in the game. We intended to make these parts of the game consistent between both screens of the user, but once this initial implementation of the tanks and backgrounds had already been accomplished, large groups of codes were already written in order to account for such implementation. We then discovered that in order to make the backgrounds, tanks, and game objects consistent between both of the players' screens, socket connections had to be implemented. In order to accomplish this, large groups of codes had to be moved around. Once the implementation has been finished, other members of the group struggled understanding the code, since the file containing these codes has been changed significantly.

Working with many files also proved to be difficult. The different files for the group's project were made to localize groups of codes that collectively accomplish specific tasks - such as routings and socket connection handling for example - so that the project structure is organized in such a way that it is easier for group members to locate the group of codes intended for certain tasks. However, one difficulty the group experienced is tracing the flow of execution, from one file to another. At times, when debugging, the group needed to know the order in which the code executes from one file to another, in order to find the specific sections causing the errors. Although there were times when the group found such tasks easy, at other times, debugging in such a way has proven to be tedious.

Learning about the different frameworks and libraries used to develop our application has initially proven to be difficult also. For example, learning about the Express JS framework and how the application's routing can be implemented through this framework was difficult at first.

During the early developments of our application, the group struggled in implementing the different routes of our application and the ways to transmit variables from one page to another. Such implementations required knowing how the methods provided by Express JS can be used, and since our group lacked prior experience in using Express JS, it was more difficult to familiarize ourselves with such framework.

Description of Test Plan

Development Testing

Local testing played a crucial role for our test plan when merging code on Github. Due to the simplicity of ExpressJS, running a local server was straightforward and useful. During the early stages of this project, visual confirmation and a quick check of the console log for the various HTTP Requests was sufficient. Starting the project by implementing wireframes of the UI first played a role at our approach to this testing plan. When backend testing came into play, our testing slightly evolved.

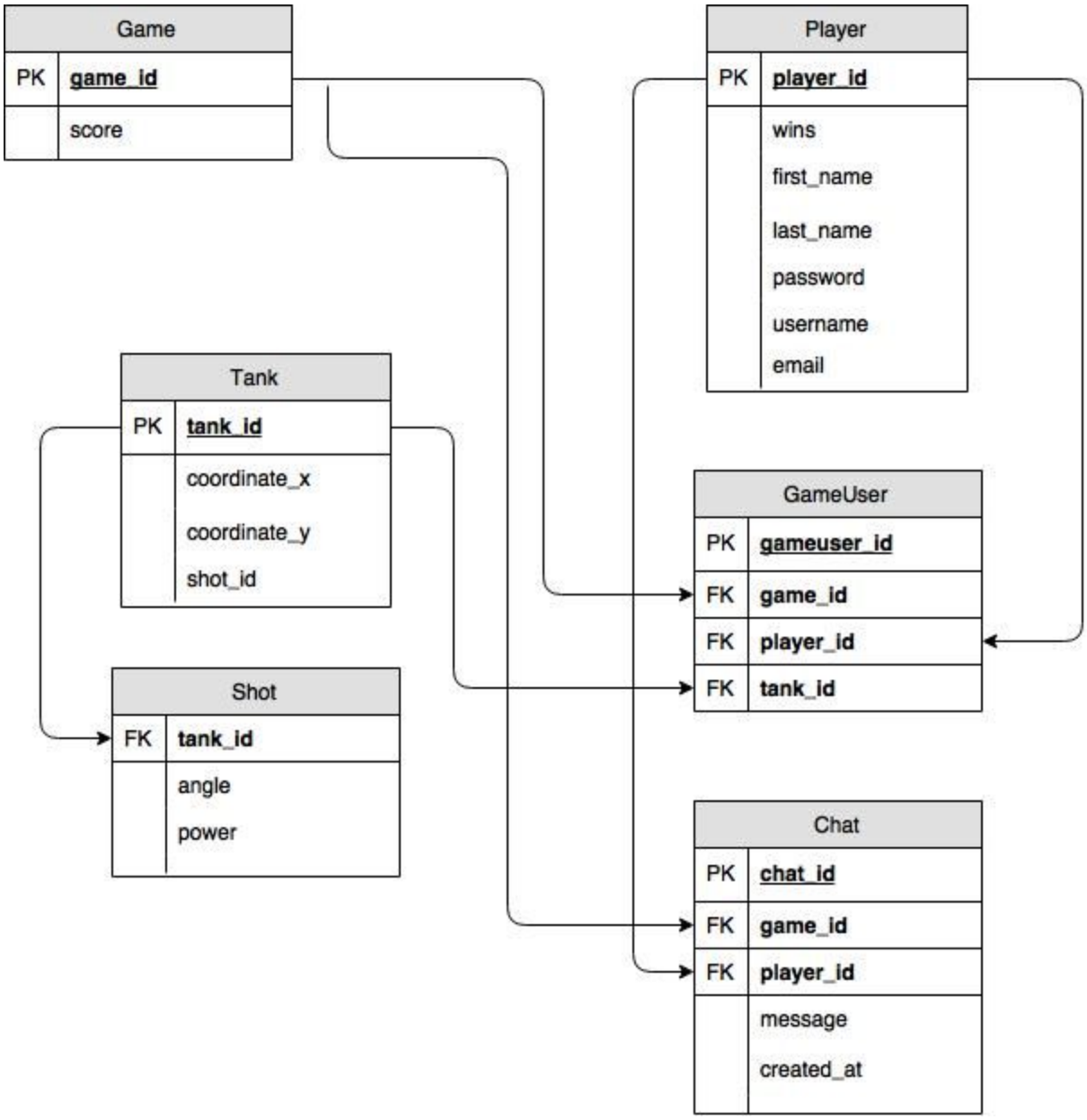
Once backend code became more prominent, we tested using various log statements to the console, along with log statements to the terminal that was running the server. Along with visual confirmation that the game was still running as expected, we utilized a basic smoke test to ensure the product was stable enough to continue building on.

To test the game logic locally, using an incognito window, or using different web browsers, was the solution. This allowed being able to log in as two different players on the same machine, thereby being able to play the game on the localhost alone. This way, we were able to ensure tanks moving, firing, and being hit is functioning properly.

Production Testing

When it came to the final tests to ensure the game was stable, reliable, and functions as expected, we utilized a more extensive testing suite. This testing proved to be a lot of fun! The best way to test the game was to ensure every situation and edge case we could think of would behave how we expected. To do this, we played the game extensively, putting ourselves in different situations, including: rotating who wins, who joins the game first, who wins the game first, the amount of movement by each player's tank, etc. Something that came out of this testing was the need to erect walls on each side to prevent tanks from falling off the screen. This was simple enough to fix, but was something that we didn't realize was important until tanks were "falling" off the screen.

Entity Diagram



API - State Machine Diagram