Benjamin Carlson
Node.js SQL Server for Data Analytics and Multiplayer Games
Excerpt from upcoming MQP-1404 Project Report
Advisor: Brian Moriarty

Abstract

Knecht is a Node.js client/server architecture that allows networked http-based applications to store arbitrary data for later retrieval and pass data between multiple clients. It is designed to support game analytics and multi-player game functionality for game engines written in HTML5/Javascript and other browser technologies.

Project History

From the outset of the project, we wanted to work on a project that was portable; a project that would work independent of the devices we would test our project on. After going through preliminary ideas and some rudimentary testing, we eventually developed server code and tests designed to calibrate and ensure that our server code and client code would be able to save data and facilitate communication with multiple users, indifferent toward the calling application. Below is the account of the project's inception, planning, development, and testing.

At first the project began as the idea of a portable game engine that could have code be transferred from device to device and still be able to play because it would run independently of the machine. Our thoughts turned to Professor Brian Moriarty's game engine: Perlenspiel. Perlenspiel is minimalistic in terms of graphics and the core engine is written completely in JavaScript and is ordinarily interpreted using web browsers. Due to its minimalistic graphics, testing would not be as demanding and could easily show our proof of concept.

For efficiency, we were originally going to write the engine in C++ using the Simple Direct Media Layer (SDL) Library to handle graphics due to it being supported on Windows, MacOS, Linux, and AndroidOS, and Mozilla Spidermonkey to handle interpreting the JavaScript code. With this we were planning on essentially making Perlenspiel act as a type of interpreter where the game code could be transferred to various machines without having to be recompiled or rewritten based on architecture.

We quickly realized that the initial project idea seemed a bit redundant. Because of how many of these devices already have web browsers that are able to play Perlenspiel games accessed from the web, we decided to look at other project ideas. We eventually decided to make server code to help add multiplayer functionality to games made in Perlenspiel. Because of how Perlenspiel is an engine instead of its own standalone game, we decided that this server would be game agnostic, meaning that the server code was not being written around an existing game. Any game could use this server provided they sent the data packaged into a JSON object. Additionally, to add more to the technical aspect of the project, we decided to add the feature of being able to save data to a database on the server to introduce permanence to Perlenspiel games.

We decided to use Node.js to write the server code. A Node.js server is a C++ program that uses JavaScript to work. This decision comes from Perlenspiel being written in JavaScript, the ubiquity of JavaScript as the main scripting language of the web and the growing interest in Node.js being used for server code. Additionally Node.js is modular, meaning if there was any extra functionality needed, we could search for the module that would facilitate our need. For an IDE, we considered both Aptana and WebStorm before ultimately choosing WebStorm. This

decision comes from WebStorm's customizability, the ease of which to set up test servers, and the fact that Aptana at the time was not being as actively developed as WebStorm was at the time. We also decided to use MySQL for the database. This decision comes from MySQL being one of the most common databases (meaning there would be plenty of resources to refer to) as well as the fact that Node.js has a module for accessing MySQL databases.

For data transfer, we decided to implement a RESTful API. This decision comes from REST being adopted as a standard for server requests and data transfer. All request types in REST (POST, PUT, GET, DELETE) could be easily applied to both the multiplayer functionality and the data storage and permanence. Additionally, REST is asynchronous, which would let handle many users sending requests at once and being able to handle them when necessary. Because we were using JavaScript, we decided to use JSON objects to send requests and transfer data. The JSON objects could easily be converted into strings and have those strings stored on the server.

Because of the scope of the project and the fact that there were three members and two of them had to work on the project for four terms, we decided on using Github to store and distribute all of the necessary files for the project. In the main folder of the project is the code and documentation for both the server and the client. Within the subfolders are all of the files required for testing both data permanence and multiplayer communication, as well as code for the most up to date version of Perlenspiel.

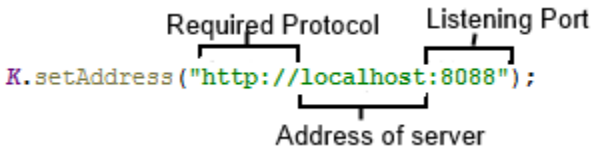
We began with simple server testing. A basic server was made and run locally on our computers to determine how to send data and store it in a database. The first successful test was that of storing and retrieving data from the database. For this, we used the felixge/mysql

module for Node.js. This revealed how to set up the necessary tables for storing data, as well as how to make database queries in Node.js. All queries are made by calling the query() function which takes the query as a string and a callback function used to handle any errors and results from the query. All queries return an array of the results in a JSON format. We also implemented query escaping, which protects the server against SQL injection attacks.

Even harder was testing client requests to the server. All requests are standard AJAX requests, however there were two major obstacles to overcome to get the requests to be successfully sent: the format of the request header and the format of the address URI. Our web browsers were preventing us from making these requests because they were cross-domain requests or requests that were being sent to an outside location. While this is prevented to avoid security risks, it interfered with our project. It is possible to get around the cross domain restriction. W3C gives us CORS which, when implemented, allows data to be accepted from machines outside the server.[i] CORS just requires a couple extra fields in the server's response to the request: one detailing how to view an outside request, one detailing who to accept it from, and one detailing what types of requests and what types of data are acceptable to send.[ii] As long as the request abides by all of these specifications, the request will be accepted.

Another issue was having Ajax requests recognize the address to send the data to. Eventually we recognized that the address has to be formatted in three parts. The first is that the address must begin with the "http://" string to indicate that the request is being made with the Hypertext Transfer Protocol. The second part is the address name or the IP address of the server to send requests to. The last part of the address must be a colon followed by the port

number that the server is listening to. If the address is formatted in this way, then the address will be recognized and the request will be sent.



An example of code used for testing.

After the initial server testing, both the server code and the client code was developed and tested further and given the name “Knecht” (named after the character in Herman Hesse’s novel *Das Glasperlenspiel*, which originally inspired the Perlenspiel engine). This time we needed to test both saving the data and using multiplayer. Both of these tests were written in Perlenspiel and provided both proof of concept and found errors that were quickly fixed.

The first of these tests was that of server data storage. This was done by modifying the Paint demonstration program for Perlenspiel. An extra row/menu was added to the program that gave four options: register the account (under a valid email address), log into an existing account, saving data, and retrieving existing data. For this test the data was to be the paint image, which was saved to the database as an array of RGB values. After successfully testing user registration and application registration, we were able to successfully save data, log back into our account, and retrieve the data. The major obstacle debugging revealed was a discrepancy in the documentation which was later corrected.

The second test was a simple multiplayer game that involved two players on their respective sides moving independently of each other. Multiplayer is handled by a series of inputs and updates. A host starts the application on their machine where all of the calculation takes place and invites clients to their game session. The client(s) submit updates to the server

which the host polls and updates the game session. These updates are sent to the game server and, if the clients have permission to access that specific data, update their game instance with the data presented in the update. In the test itself, the client would send input in the form of its XY coordinates. The host would read this data and draw the client's character on the screen. The host would send its coordinates in an update that the client had permission to view, and it too would draw the host's character on the screen. Apart from a few more discrepancies in documentation, the major bugs encountered in this test was that the data was coming back as a string without being parsed into a JSON object, garbage collection on the database, and there were some errors in assigning permissions to the client. This test was run both locally as two game instances in different browsers (Firefox and Chrome) on the same machine as well as over a distance and both tests passed.

Instructions for installing Node.js on a developer's machine for testing purposes

The following instructions assume that the user already has WebStorm installed on their computer.

To install Node.js

1. go to http://nodejs.org/download/ and click on the appropriate installer to download
2. Run the installer.
    a. IMPORTANT: Pay attention to the location "node.exe" was installed to.

To enable Node.js in WebStorm

1. Open WebStorm
2. Go to File>Settings
3. On the left side of the screen, expand "Javascript" and click on "Node.js"
4. On the line indicated by "Node.js interpreter" Put in the location where "node.exe" is located

To install Modules from WebStorm[1]

1. Open WebStorm
2. Go to File>Settings
3. On the left side of the screen, expand "Javascript" and click on "Node.js"
4. Under the "Packages" area, click "install"

[1] These instructions are known to work for WebStorm version 7.x and later. Earlier versions might not contain the required "Packages" area.

5. Scroll down to “mysql” and select it and install it

To install Modules from the command prompt

1. Open the command prompt window

2. type “npm install mysql” in the command prompt to install the mysql module

---

[i] “Cross-Origin Resource Sharing: W3C Candidate Recommendation 29 January 2013.” W3C. Electronic. < http://www.w3.org/TR/2013/CR-cors-20130129/>. 2013. March 27, 2014

[ii] Blinov, Alexey. “Node.js CORS”. Github.com. Electronic. < https://gist.github.com/nilcolor/816580>. 2013. March 27, 2014