Group 20

Sean Hinchee

Sam Westerlund

Languages used: Node.JS, HTML(5), Javascript

External Libraries: None

Tested With: Firefox

To Run:

cd to the base project directory.

Run *node server.js*

Checklist

- Templating system used within *.html files
- Statefulness without cookies or PHP
- State-machine-esque handler system
- Unified, monolithic design within primary, listening, function
- Canvas/image integration from browser to the server to a new browser
- No page redirection or messy url's, all internal management done through POST/GET (again, without PHP)
- Fully asynchronous system
- Durable design with little room for errors
  - State-machine-like design ensures transitions between states within the client are controlled and systematic (See: *ordo*)
- Purposefully limited backend with GET only having two cases (base page and picture requests)
- Self-hosting web server, does not require Apache, NGINX, nor any other existent web server

Group 20

Sean Hinchee

Sam Westerlund

Node.Paint – An exploration of Node.JS programming within a stateful web application.

Table of Contents

Overview

Node.Paint was created with the intent of making an interactive web application that one user could connect, paint using their cursor within a canvas along with an answer. The answer would be accepted by the server as an image and answer pair. A secondary user could enter a session identical to a submitted image and answer pair and attempt to guess the answer to the image and be informed whether they were correct, or not.

After brief conference, we concluded to use Node.JS as our language of choice due to both group members not having used Node.JS before and looking to expand our experiences. Furthermore, the decision to create a web application was decided to the still infantile experience both members possess with web development. Portfolio one would have been both of our first formal web applications so we looked to design and build a different model of web application with a different take on infrastructure while maintaining a well-designed system. The two portfolios, should there be any doubt, are wholly separate.

Portfolio one was designed in a more systems-oriented mindset, Node.Paint was designed to be a wholly web-based application with a standalone, non-dependent infrastructure. As such, many particulars and procedures within Node.JS had to be learned to work around crutches such as cookies and PHP which may be traditionally used for certain elements of web applications. Statefulness was a requirement since a given “Session” had to possess a guessable/drawable state regardless of the system accessing the server.

Multiple over-arching designs were cycled through, with the result being a logically-monolithic, state-based design, taking cues from state machine systems. We went through, as a group, a series of critical analyses to judge which design was most optimal from a benefits of simplicity perspective and then furthermore a potential scalability, or future improvement/support perspective.

A self-contained design is optimal for a small experiment such as Node.Paint. Portions of the assembly were created to be both easily tested, easily examined, and easily extended. Basic grassroots design principles helped us expand functionality past our basic concepts without undoing any work or losing any time, while all the while making the concepts and designs easy to explain to each other.

New and Complex

Node.JS is a language that is based upon the br;owser implementation of JavaScript. The system runs in a virtual machine-like environment and possesses a plethora of features that enable JavaScript-like code to be run on a server-side environment, separate from a client-side browser. In practice, this results in a system much like traditional programming, but simply adopting the JavaScript syntax and adding extensions to make the language more useful as a systems-like environment.

In class, we have worked with some JavaScript, outside of class, neither of the group members have worked in JavaScript formally, as such, this was a form of new frontier as it forced us to learn a greater depth and intricacy with the JavaScript environment to make full use and take full advantage of the system.

We wanted to design a self-serving web application. In this sense, we wanted the experience from a deployment point of view to be that of a drag and drop experience. The folder containing our project can simply be moved to an arbitrary directory, run, and experienced from the configured port of choice. In this way, it is similar to a statically linked binary-based application where the binary is portable and simply needs to co-exist with its dependent files in the project's directory.

As a group, we experienced asynchronous Node.JS programming for the first time, naturally, due to a lack of exposure to Node.JS, but also because both of us lacked thorough, if any, real experience in asynchronous JavaScript prior to this project. AJAX was used as well as Node.JS, meaning two individual platforms for asynchronous JavaScript had to be made to play nicely together.

A templating system was implemented, using an easily recognizable syntax and a simple find-replace structure within the *.html* files contained within the project. Templating was leveraged to move variables from the server side to the client side with ease. Typically, this could be implemented with an internal API for AJAX to request data from the server, but this method felt more fluid and was in line with our sub-goal to pursue simplicity and a logical structure. More internal API's adds unnecessary complexity that achieves little beyond the initial struggle.

A failed, proposed, solution to the templating issue initially, had been to inject multi-line concatenated string into the web page by way of having an HTML and a JavaScript portion of each file split in half with the server injecting data at the beginning of the HTML portion and between the HTML and JavaScript portions of the files. The strategy was not long lived and was surpassed by true templating on the grounds of being less useful and unnecessarily complex.

Statefulness, as a feature and a problem both, was solved by way of providing a series of reference points for a session to utilize to recognize whether it was an active (able to be guessed) or inactive (requiring an image) state. While images may persist between sessions due to file systems being separate from the in-memory system, state of answers if contained on an instance-by-instance basis as all information for sessions is stored in memory, thus being readily accessible and easily served to clients. Due to the nature of the system's design, a given user could stay in a POST-based state of providing a drawing until the given connection/session was terminated (typically a tab). Although this does not ensure a client-by-client-based state/session tracking system, it ensures that a given instance of connection can retain its state regardless of happenings in other sessions or tabs.

The general design was that of avoiding PHP and cookies. A state-machine-like system where a user begins in a certain, entry, state, then progresses through various other states, or *ordos*, as they are referenced in the system source. These *ordos* have limited transitions to other *ordos* and as such, passively control session management by not accidentally mis-stating a given session.

Bloom's Taxonomy

A defining portion of our new ground covered within the project was the choice of language. Although both group members have performed JavaScript based programming within the scope of the class, neither of us have utilized the Node.JS platform in the past. A great deal of research went into the project, as Node.JS takes the traditional JavaScript language and converts it into a more systems-like language with pseudo-packages and some additional type-like methods. Among other things that were learned was the structure of a Node.JS instance. Specifically during the call to *http.createServer()* there is a function passed into the method which takes in a *request* and a *response*. This provides a standardized input/output format by which information would be acted on as though it were a read-from input and a write-to output, anonymous to the fact that it is a network connection. The stream-like abstraction was utilized heavily to write compact, easy to read, code which could be easily explained.

Asynchronous systems were explored in a new environment. Each time a session communicated with the server, it was handled by the *http.createServer()* asynchronously, with a new, fresh, call to the method being called for every session. Logically, this meant that a way to represent state via page calls over POST, with the client tracking state outside of a given handler-loop model, would be an efficient progression. When we built portfolio one, state was tracked exclusively on the server-side, with no real way of tracking, or relying on, state being present within the system. With portfolio two, we required statefulness to be a feature due to the fact that there must be a role as a "Drawer" and a role as a "Guessor."

With the note on the prior portfolio, there is the matter of the further exploration of the development of web applications. In regards to portfolio one and two's separate relationships, this is the matter of a traditional/systems-style approach with portfolio one, and a more progressive/web-style approach with portfolio two. Statefulness was achieved and enforced over HTML and JavaScript between the client and server using a state machine model, with a lightweight, compact self-hosting server. The similarities between the two portfolios are those of self-hosting, using a state-machine-like model for controlling the experience of a client, and a simple, efficient, scalable, concurrent design.

Front and back-end unification was experimented with due to the nature of Node.JS. While traditional systems would have two separate languages to develop in, JavaScript, in some form, was utilized on both the frontend and backend meaning that there was little mental context switching and the development cycle could be very productive. AJAX's POST system was utilized to communicate state and other details/information to the server, with Node.JS's POST system being respectively used by AJAX to gather information from the server alongside the homebrew templating system used to pass document-level information/variables.

Various design complications and implementation difficulties were encountered in both testing and initial development. During testing, a bulletproof design was necessary to ensure that everything was both thread-safe (which was taken care of primarily by Node.JS) and reliable, as per the system would not go down after a given period of time or a heavy load. Several designs were trialed, such as templating being done by inserting string literals into the middle of a file split into halves as strings (HTML and JS halves of web pages). Templating was changed to rely on a system where placeholders were replaced with variable values prior to serving the page to the client. The core system design was designed form the beginning to be a state-machine-like system dictated by *ordos* which encompass a given service being provided by the system at a given time. A page would dispense an order and then whatever further information would be needed by the system at the time of request. By using this model, all states are restricted by the states which have access to transfer to a given state and then furthermore which states can lead to other states. Variety lessens and any amount or number of data elements can be transferred without worry over POST so long as they conform to the internal standards for a given *ordo*. As a result, debugging is very limited and a large number of variables do not need to be co-existent in a given *ordo*'s code.