Group 20

Sean Hinchee

Sam Westerlund

Language(s) used: Node.JS, HTML(5), Javascript

External Libraries: None

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

1. Overview
   1. Initial considerations
   2. Initial design
2. New and Complex
   1. Introduction to Node.JS
   2. Self-serving web applications
   3. Asynchronous Node.JS programming
   4. Templating and injection techniques (initial and final)
   5. Statefulness without cookies or PHP
3. Bloom’s Taxonomy
   1. Language choice
   2. Asynchronous systems
   3. Web applications (new and old)
   4. Front and Back-end unification
   5. Design complications and implementation difficulties

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 browser 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