INDIVIDUAL SYSTEMS PRACTICAL

HTTPictionary: A Real-Time Canvas Webapp

Leonidas Diamantis
VU University 2013

TABLE OF CONTENTS

Introduction

This report is describing the project for the *Individual Systems Practical* course, titled *HTTPictionary: A Real-Time Canvas Webapp*. For this project, I tried to create a web application where the users connected to it would be able to communicate and exchange various types of information with each other in real time. To demonstrate my work & research, I implemented a real-time, web-based canvas game.

The structure of this report goes as follows: first, I give a brief description about the concept of the application we created. I continue with some details about the different technology components I used to implement it. I provide some future work insight on the next section, and I finish the report with some conclusions about my work and what I gained from it.

The concept

The motivation for this assignment initially derived from the fact that nowadays, with the new features HTML5 provides such as WebSockets, it is possible for users to exchange information with each other in real-time, without having to care about platforms or plugins or browser types (under the honest assumption that browsers do their best to support the new technologies).

With the above thought in mind, I wondered whether it is possible for users to exchange visual information among other types. So I decided to create an simple web-based application to prove that this is actually possible.

For the scope of this assignment, I built a real time communication interface on top of a minimal, web-based implementation of the Pictionary game, where the players will be interacting with each other by means of a HTML5 canvas. The game includes two types of roles: the drawer, who is the one responsible to draw in their canvas the word or phrase that the team must find, and the guesser, who observes the canvas and tries to guess what it is the team is looking for. However, the game works in such way that the drawer and the guesser(s) can be in different places, and through this real-time interface the guesser is able to see what the drawer is painting at the very time when this is happening.

The game supports multiple guessers at a time, and only one drawer. Furthermore, as a scoping matter, the game works as a “single room” application for the time being, meaning that there can only be one team (one drawer and multiple guessers) playing at a time. However this is a game-specific issue which, by not being implemented, does not harm

the purpose of this assignment. Of course it is subject for further development, given that this application is extended to actually implement the full feature set of Pictionary.

Technology

This section explains the technologies which were used to build the stack of the application, their characteristics and what part each of them played in the development process.

Node.js

Node.js is a system which makes it possible to develop server-side software with JavaScript, along with all the features JavaScript provides. Such features include efficient non-blocking I/O, asynchronous communication (with the server system itself or through the network), prototyping and easy event-driven development. It is a packaged compilation of Google's V8 Javascript Engine.

Despite the fact that it is a very "young" project, Node has already drew much attention with its feature set and capabilities, and a large community is built around it with projects that contribute to low and high levels of the stack that has been introduced by Node.

Node.js was the backbone of the real-time canvas application. The complete server-side implementation is written in JavaScript and in Node's notation, and is run as a Node application. All HTTP request & response handling, routing, I/O & event handling was made by means of modules implemented on top of Node.js. More information about each individual tool & module will follow in this section.

Express.js

Express.js is a web application framework, tightly paired to Node.js. It offers a large number of features for building web applications, both in the server-side and the client-side. It embeds the middleware set of Connect.js, thus providing with infrastructure for building routes for the web application, logging, query-string parsing, defining paths for static files etc. An example of the usage of such middleware is demonstrated at Figure 1. Additionally, Express.js integrates with front-end template engines (such as Jade & EJS) for rendering the views of the web application and takes care of data exchange among the files which resemble the source code of the app.

```
// middleware
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));
```

Figure 1: Using Express.js middleware, powered by Connect.js

Express also runs as a Node package, and upon its first execution it creates the project structure and the dependencies file, called *package.json*. An example of a project structure created by Express and the *package.json* of the real-time canvas app are available in Figures 2 & 3.

```
create : .
create : ./package.json
create : ./app.js
create : ./public
create : ./public/javascripts
create : ./routes
create : ./routes/index.js
create : ./views
create : ./views/layout.jade
create : ./views/index.jade
create : ./public/images
create : ./public/stylesheets
create : ./public/stylesheets/style.css

install dependencies:
  $ cd . && npm install

run the app:
  $ node app
```

Figure 2: The default structure of a Express.js project

Figure 3: Express.js dependency file package.json

For the real-time canvas app, the same structure was used as the one demonstrated in Figure 2. In detail, the structure is as follows:

- *app.js*: the root server-side file, the one which is run by Node and puts the server in the "up and running" state
- *package.json*: the file which defines the dependencies of the project, so that Node knows what packages need to be present and compiled before running the app
- *node_modules/*: the directory which contains the compiled dependencies of the app, as defined in the *package.json* file
- *public/javascripts/*: the directory which contains the client-side scripts which are used by the drawer, the guesser and the index pages for event handling, DOM manipulation and drawing on the canvas component
- *public/stylesheets/*: the directory which stores the CSS files for styling the pages of the application
- *routes/*: the directory which contains the respective routes for the drawer, the guesser and the index pages. Each route is responsible for setting any page specific variables which will be used by the view before rendering the view itself. An example of the drawer route is demonstrated in Figure 4.
- *views/*: the directory which contains the respective view files for the drawer, the guesser and the index page, that are rendered in the browser

```
exports.index = function(req, res) {
    var word = words[Math.floor(Math.random()*words.length)];
    res.render('drawer', { title: 'Drawer', name: drawerObj.name, theWord: word });
};

exports.get_word = function(req, res) {
    var word = words[Math.floor(Math.random()*words.length)];
    res.send({'word': word});
};
```

Figure 4: How the *routes/drawer.js* file renders the drawer view, and how it returns a new word

It is clear that Express brings a set robust features to the developer and a clean & intuitive structure to the project, with minimal overhead and efficient modules.

Socket.io

Socket.io is a JavaScript library for realtime web applications. It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Both components have a nearly identical API. Like node.js, it is event-driven.

Socket.IO primarily uses the WebSocket protocol, but if needed can fallback on multiple other methods, such as Adobe Flash sockets, JSONP polling, and AJAX long polling, while continuing to provide the same interface. Although it can be used as simply a wrapper for WebSocket, it provides many more features, including broadcasting to multiple sockets, storing data associated with each client, and asynchronous I/O.

Socket.io is what makes communication possible in this real-time canvas application, and consumes the biggest part of logic and handling in the server-side code of the app. The general architecture of the event mechanism is the following:

- When a guesser wants to interact with the drawer, they fire up (emit) a socket event to the server. The server has handlers attached to the socket for that particular event, and propagates the communication action of the guesser to the drawer, as illustrated in Figure 5.
- The drawer, in contrast to the guessers, needs to let all the guessers know of their actions. Thus, when the drawer performs an action, a socket event is fired to the server and the respective server-side handler for that event is responsible for broadcasting all relevant information about that particular action the drawer wants to broadcast. Figure 6 shows what this scheme looks like.

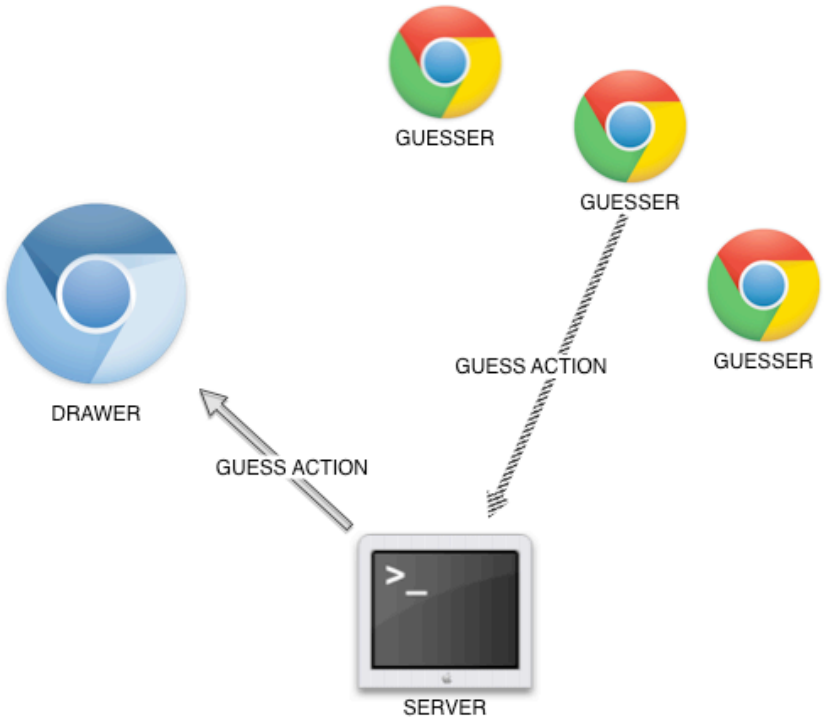

Figure 5: An action from the guesser to the drawer via the server

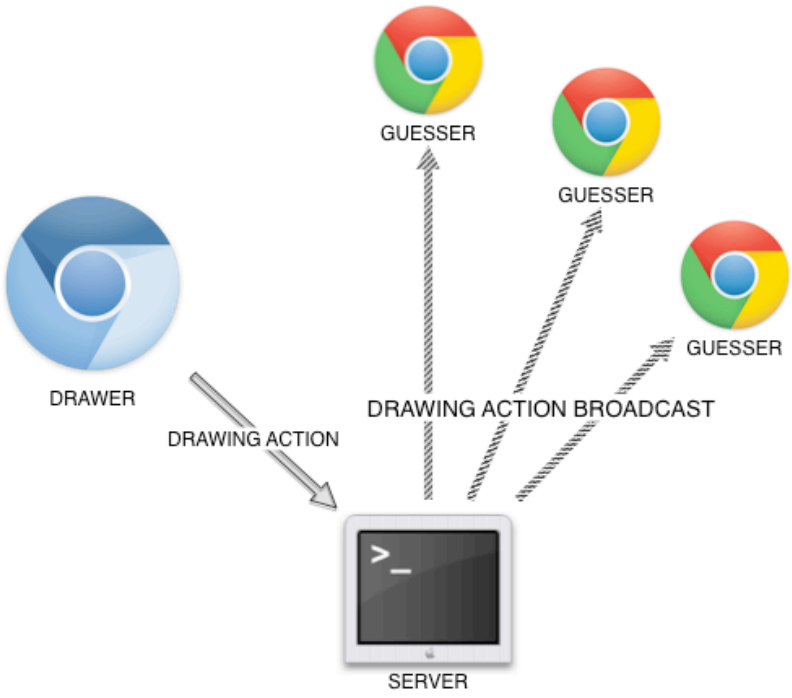

Figure 6: An action from the drawer being broadcasted by the server to all the guessers

Several events where defined for this application, and each one is handled in a different way in the server and the client side. These events are:

- *drawerconnected*: emitted when a user with the drawer role is connected to the system. In this case the server stores the socket id of this user for future end-to-end communication.
- *drawingaction*: emitted when the drawer is drawing on their canvas. The server handles this event by collecting the relevant information of the drawing action (action type and coordinates) and broadcasts that data to all the guessers.
- *passiveDraw*: used by the server to notify the guessers that there is a drawing action taking place, carrying specific data. On the client side, the guessers listen in their sockets for that event and they handle it by passively drawing on their canvas based on the data they receive by the server.
- *clearcanvasaction*: fired up when the drawer clicks their ”clear canvas” button. The server handles this by broadcasting a “*clearcanvasaction*” command to all the guessers and the guessers handle that event by passively clearing up their canvas.
- *guessaction*: emitted by the guesser when they perform a guess about what the drawer is drawing. The server handles this by sending the guess data only to the drawer.
- *guessinteraction*: emitted by the drawer when they interact with an incoming guess by rating it. The server handles this event by broadcasting the rating of the drawer to all the guessers.
- *gamefinished*: fired up when the team has found the word they are looking for, and is broadcasted to everyone by the server.
- *playagain*: emitted by any team member when they click the “play again” button. In this case, the server performs a check on whether someone else had already requested to play again, and if this is not the case then it broadcasts the “*playagain*” command to all connected users.
- *pokedrawer*: emitted by the guesser when they want to let the drawer know that they are waiting for some interaction to one of their guesses. The server handles this event by sending the “*poke*” command to the drawer only, and the drawer raises an alert upon arrival of such command.

- *disconnect*: fired up automatically when somebody disconnects from the system. The server checks then whether the client who disconnects is the drawer, and if this is the case it broadcasts a "*drawerdisconnected*" message to all the guessers.

Socket.io allowed the creation of a flexible communication platform between the connected users, making real-time communication and information exchange in the web a relatively easy process, since it took care of all the underlying infrastructure concerning networking and socket support. It is the kind of technology that made this project happen in the first place.

Jade

Jade is a template engine built for Node, which allows for cleaner development of the front-end markup of a website. It is the default template engine that Express.js uses, and it works in perfect harmony with the javascript infrastructure created by the framework. More specifically, Jade allows inline interpolation of strings which are stored in JavaScript variables, supports conditional & loop statements, mixins and partials among others. Figure 7 illustrates how Jade syntax is translated & rendered as HTML.

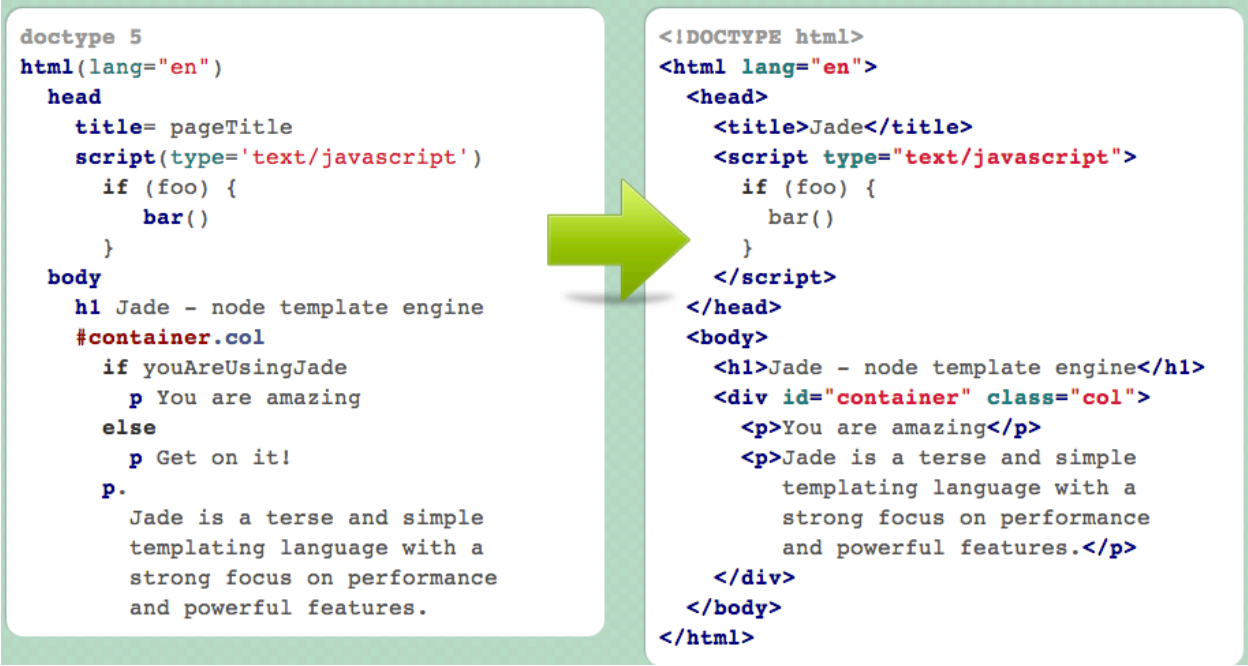

Figure 7: Jade syntax translated to HTML

For this application, four different Jade views where implemented:

- *views/layout.jade*: the root layout view, where the document type, the <head> part and the outer <body> of the document are defined
- *views/index.jade*: the view of the homepage
- *views/drawer.jade*: the view of the drawer's page
- *views/guesser.jade*: the view of the guesser's page

Each view contains included files, layout components and content which are specific to the nature of the page, for example the drawer's view requires scripts for drawing in the canvas, whereas the guesser's view requires scripts so that the canvas is being drawn passively. Figure 8 shows how part of the index view looks.

The views are rendered from their respective routes, with all relevant local information that might concern them. For example, Figure 4 shows how the drawer's route calls the *render* function to create the drawer's view, together with its local variables. This call triggers the compilation of the Jade template (namely *drawer.jade*) and it's rendering to the browser.

```
extends layout

block append head
    script(src='/javascripts/home.js')

block content
    - var target = '/drawer';
    - if (drawerPresent) target = '#sorryModal';
    div.row
        div.span12(style="float: none; margin: 0 auto; text-align:center;")
            h1(style="text-align:center;")= title
            div.row(style="margin-top:70px;")
                a#drawer-button.intro-text-link(href="#{target}", data-toggle="modal")
                    div.span5.intro-text-container
                        h2 DRAWER
                        p   You will have to draw the word or expression which your team has to guess,
                            | and you will be able to rate the incomming guesses based on their relevance to what you are drawing.
                a#guesser-button.intro-text-link(href="/guesser", data-toggle="modal")
                    div.span5.offset2.intro-text-container
                        h2 GUESSER
                        p   You will have to guess what your team-mate is drawing, and you will get feedback from the drawer
                            | on whether your guesses are relevant or not.
```

Figure 8: Part of *index.jade* view

jQuery

jQuery is a multi-browser JavaScript library designed to simplify the client-side scripting of HTML. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications. jQuery also provides capabilities for developers to create plug-ins on top of the JavaScript library. This enables developers to create abstractions for low-level interaction and animation, ad-

vanced effects and high-level, theme-able widgets. The modular approach to the jQuery library allows the creation of powerful dynamic web pages and web applications.

All the client-side scripting operations in the real-time canvas application were implemented with the use of various jQuery components, either the generic selectors it provides or particular plugins built on top of jQuery. Figure 9 shows the client-side initialization function of the guesser, where the HTML5 canvas and the guessing form are initialized and the guesser-specific socket events are defined.

```
$(function() {
    init_canvas();

    $('#guess-form').submit(function(e) {
        if ($('#guess-box').val() != '') {
            makeGuess($('#guess-box').val());
            $('#guess-box').val('');
        }
        e.preventDefault();
    });

    $('#guess-submit-button').click(function() {
        $('#guess-form').submit();
    });

    $('#poke-button').click(function(){
        socket.emit('pokedrawer');
    });
    $("#guesserModal").modal({show : true});
});
```

Figure 9: Client-side initialization function of the guesser

Bootstrap

Twitter Bootstrap is a free collection of tools for creating websites and web applications. It contains HTML and CSS-based design templates for typography, forms, buttons, charts, navigation and other interface components, as well as optional JavaScript extensions.

Bootstrap was used to define the general layout & look-and-feel, the typography and the color scheme of the application. Moreover, Bootstrap's modal plugin was used for the information pop-ups which appear when opening each page. The modal plugin is based on jQuery.

Heroku

Heroku is a cloud "platform as a service" (PaaS) supporting several programming languages. Heroku is one of the first cloud platforms, and when its development began it supported only the Ruby programming language, but has since added support for Java, Node.js, Scala, Clojure and Python and PHP. The base operating system is Debian or, in the newest stack, the Debian-based Ubuntu.

Heroku was used to deploy & host the application demo. The deployment process requires three steps.

The first step is the Heroku-specific configuration of the application. Heroku's stack does not support WebSockets yet, so the application's sockets need to be configured so that they use Ajax long polling for communicating and pushing the events among the connected users. In addition to this, the Socket.io instance of the application needs to be specifically configured so that it allows access to the sockets from all origins, to avoid getting *403 Forbidden* responses from the Heroku servers. Figure 10 shows how the real-time canvas application's sockets are configured for Heroku servers.

```
// heroku-specific configuration
io.configure(function () {
  io.set('origins','*:*');
  io.set("transports", ['xhr-polling']);
  io.set("polling duration", 10);
});
```

Figure 10: Heroku-specific configuration of sockets

The second step for deployment to Heroku concerns version control. Heroku servers accept applications which are deployed by means of GIT, so this tool needs to be installed. Additionally, Heroku servers have to know which command needs to be executed in order for the application to start running. For this purpose, a special file called *Procfile* needs to be created in the root path of the application. This file will contain the command which will make the application run once it has been deployed. The necessary command placement for this application is shown in Figure 11.

When all files are ready, a new repository needs to be created as shown in Figure 12. A note at this point: the *node_modules* directory does not have to be included in the repository, because it gets created every time the application is built and is usually unnecessarily large, which makes deployment slower.

```
touch Procfile
echo 'web: node app.js' >> Procfile
```

Figure 11: Creating the Procfile and inserting the necessary command to be run once deployed in Heroku

```
git add .
git commit -m 'initial commit'
```

Figure 12: Adding all necessary files to the GIT repository

The final step of deployment involves Heroku Toolbelt, a tool which carries all necessary command-line binaries to help connecting to Heroku's platform and deploying the project.

After installing Toolbelt, one needs to navigate to the root directory of the project, login to Heroku and create a remote app with the *create* binary, as shown in Figures 13 & 14. Once the remote app (which is actually an empty remote GIT repository) is created, the code of the project needs to be pushed to the remote, as shown in Figure 15. After this step, the command specified in *Procfile* is executed, and Heroku determines the type of application its dependencies, and proceeds by building the app and making it accessible via a URL, which can be defined by the user upon the creation of the remote app.

```
$ heroku login
Enter your Heroku credentials.
Email: adam@example.com
Password:
Could not find an existing public key.
Would you like to generate one? [Yn]
Generating new SSH public key.
Uploading ssh public key /Users/adam/.ssh/id_rsa.pub
```

Figure 13: Logging in at Heroku with Toolbelt

```
$ cd ~/myapp
$ heroku create
Creating stark-fog-398... done, stack is cedar
http://stark-fog-398.herokuapp.com/ | git@heroku.com:stark-fog-398.git
Git remote heroku added
```

Figure 14: Creating a Heroku remote GIt repository

```
$ git push heroku master
```

Figure 15: Deploying (pushing) the project to Heroku remote repository

Future work

During my research and implementation, I came across several ideas and concepts of web-based solutions that can take advantage of real-time communication and data exchange. It is clear that having such an advanced, flexible and interactive way for users to communicate with each other can result in equally advanced and interactive user experience. Below I mention a few of such ideas & concepts.

Real-time collaborative games

Following the same implementation philosophy of my project, all types of collaborative games (such as board games, turn-based or concurrently playable) whose rules can be implemented through message-passing between the players, can now be implemented in the browser, in a native way, without third-party plugins or software required from the user's side and with complete transparency of what the nature of the server is.

A good example of this notion is the recent experiment from the developers of Google Chrome called *Racer*, a collaborative game for the mobile version of Chrome which allows sharing of the racing track for up to five players.

WebRTC

WebRTC (Web Real-Time Communication) is an API definition being drafted by the World Wide Web Consortium (W3C) to enable browser to browser applications for voice calling, video chat and P2P file sharing without plugins.

Given this Node.js - Socket.io - Jade stack (or any other template engine for that matter), such scenarios become reality and the browser acquires a special position in a user's workflow, replacing P2P, messaging & communication software with native web applications which perform just as good as the standalone software we are all using nowadays.

Streaming APIs

Real-time platforms & interfaces enforce distribution of information in real time and to multiple users at once, which is the quintessence of a streaming API. Thus, in a web-based real time environment, we have the possibility to implement such API's with a very administrator friendly and lightweight server infrastructure, capable of handling distribution, concurrency and data volume issues with good performance.

Conclusion

My research has shown me that real-time communication in the web is a real situation. With all these new technologies and tools which solve so many problems that seemed unsolvable before, we can build systems and applications which would really shift the communication paradigm. Moreover. there is a large community out there full of people who try to take this one step further every day, and the results are really impressive. And we must not forget that these technologies are still "young" and under development, and the tools that provide with all these new possibilities are still under development, so when they become more mature I believe we shall see even more impressive achievements.