# JTEC Technical Documentation

## Calibration

### HTML

In the index.html file all calibration information is contained in the calibration . Inside of the calibration  are various hyperlinks (<a>) and another  to constitute its elements.

### CSS

The calibration overlay and overlay content are styled with the rule-sets '.overlay' and '.overlay__content' respectively. The '.overlay a' rule-set continues to style the text in the calibration page.

The close and help buttons are styled by '.overlay .closebtn' and '.overlay .helpbtn' while their hover colors are styled by '.overlay .closebtn:hover' and '.overlay .helpbtn:hover' for when a mouse hovers over either button.

Finally the '.overlay .calDot' styles the red dot that is clicked by the users during calibration.

## Javascript

### Functions:

moveCalibrationDot()

Finds the current location of the calibration dot and moves it to the next designated location on the screen. If all calibration dots have been clicked endCalibration(). Called when the calibration dot is clicked.

helpCalibration()

Displays an alert to the user with the calibration instructions. Called when the help button is clicked on the calibration screen.

closeCalibration()

Prompts the user to make sure they want to close the calibration. If yes the calibration screen is closed and WebGazer is stopped. Called when the close button is clicked on the calibration screen.

endCalibration()

Closes the calibration screen and turns off the mouse listener in WebGazer so that mouse clicks do not add more points to the regression model.

openCalibration()

Initializes a user in the parse database and opens the calibration overlay. Called when the webpage has loaded and WebGazer has been initialized.

## Validation

### HTML & CSS

The setup for validation is already done if calibration was set up correctly. The only difference is Validation has its own css overlay that has a red circle that moves to each of the quadrants. The red circles in validation also blink after they are clicked by using a css animation @keyframes blink which sets the color to change of the circle every second for the 3 seconds the user is looking at it.

## Javascript

### Functions:

openValidation()

Opens the validation overlay.

closeValidation()

Prompts the user to make sure they want to close the validation. If yes the validation screen is closed and WebGazer is stopped. Called when the close button is clicked on the validation screen. Sets inValid to false to stop saving prediction points to the array that will be sent to database.

endValidation()

Closes validation overlay and sets inValid to false. It then calculates the valid percent of the data collected during validation by dividing the number of points that were within the correct location by the total number of points collected during validation. Then it makes a call to pushData( …) that sends the data collected between validation periods to the database. Then it clears the global variables that store the data between validations.

moveValidationDot()

Finds the current location of the validation dot and moves it to the next designated location on the screen. If all validation dots have been clicked endValidation(). Called when the

validation dot is clicked. The click actually calls a method startTimer() that sets collectingValidation to true so it knows to check if the prediction points are within a certain distance. Then startTimer() calls moveValidationDot() after a 3 second delay (`setTimeout(moveValidationDot,3000);`).

## Pushing Data to the Server

All communication with the server is handled in the file "JTECServer.js" in the FrontEnd folder. The purpose of this file is to send the data retrieved from WebGazer to the server.
It is necessary to edit the first two lines:

```
Parse.initialize("a3Vzn4zfnMObdOfhNJJEdhiFaAdG62rz1Z8ictHmG");
//Parse.serverURL = 'https://jtec-dev.us-east-1.elasticbeanstalk.com/parse'
Parse.serverURL = 'https://server.jtecdesign.com/parse'
var EyeData = Parse.Object.extend("EyeData")
```

The Parse.initialize line must contain your appId from the dashboard config file. The Parse.serverURL line must contain the URL for the server you are using. The third line must contain the name of the of the class that you are pushing to.

**Functions:**

initializeUser()

creates a new user to push data to. If the user is not new, it will continue to push data to the existing user.

makeID()

generates an ID to be used for the username of a user.

pushData()

takes a x array, y array, timestamp, screen size, valid value, valid percent, and url. It takes all of this data and pushes it to the server. If you want to create new fields edit both the parameters and the lines which set their values:

```
newSet.set("user",Parse.User.current());
newSet.set("x_array",x_array);
newSet.set("y_array",y_array);
newSet.set("timestamp_array",timestamp_array);
newSet.set("screen_size", screen_size);
newSet.set("valid", valid);
newSet.set("valid_percent", valid_percent);
newSet.set("url", url);
```

logOut()

logs out a user.

randomValue()
generates a random value. This is a helper function for makeid()

## Dashboard

The Parse Dashboard documentation is located here. We strongly recommend reading it and using it for reference.

**Dashboard Config file contents:**


serverURL specifies the URL of where the server is stored. If running locally, change line to "serverURL": "http://localhost:1337/parse"
The appId and masterKey were randomly generated and inserted into the config file.
Lastly, the appName is whatever you want your app to be called.

Code to put in terminal to run dashboard:
parse-dashboard --config dashboard-config-AWS.json

## Plotly

The Plotly JavaScript Graphing Library can be found here. We strongly recommend reading it and using it for reference.

Plotly handles all of the data visualization, specifically generating the heat maps. All of the functionality of Plotly is written in the file Plotly.js. This file contains three functions, create_blank_array(a,b), populate_array_binned(), and make_heatmap().

**Functions:**

create_blank_array()

generates an array of zeroes with width a and height b.

populate_array_binned()

takes in the x and y array and populates the array with how many times the user looked in the binned area (as determined by bin_size)

make_heatmap()

uses the previous functions to generate an array that represents the heat map, and plot it with plotly.

The object named data can be edited to further customize the heat map:

```
    var data = [
  {
    z: populate_array_binned(x_array, y_array,screen_size,100),
    type: 'heatmap',
        //opacity: 0.5  //uncomment to add opacity
  }
];
```

The example code above has a bin_size of zero. Secondly, further customizations such as adding opacity to the heat map can be done. This can all be found in Plotly's JavaScript documentation.

## Querying the Database

The Parse JavaScript Developer's Guide can be found here. We strongly recommend reading it and using it for reference.

Data is retrieved from the database by using Parse's query methods. All of the functionality for querying is contained in the file Query.html. This file contains four functions, generate_query(), get_most_recent_user(), generate_recent_user_heatmap() as well as generate_heatmap(). Generate_query() begins by creating a variable called EyeData is a Parse object called "EyeData". This is because the query is looking for "EyeData" objects from the database. Next, a new parse query object is created and stored in the query variable. The function then generates a query greater or equal to the valid percent specified, equal to the objectID, and queried in the order that they were created. Upon success, it passes the results into completion function that was passed in.

**Functions:**

generate_heatmap()

parses the data_array that is passed in into an x_array and y_array that can later be used for generating the heat map. The function then calls make_heatmap passing in the computed arrays.

get_most_recent_user()

generates a query that returns the object of the most recent user on the database.

recent_user_heatmap()

a helper function that makes it simple to generate a heat map on the most recent user. It Generates a query on the user_var that is passed into it, and generates a heat map. This allows for a get_most_recent_user() to be called with this function as a parameter.

The following two lines generate the two different types of queries:

A)

```
generate_query("M3NKfBm9kR", generate_heatmap,30); // use for specific user
```

This generates a query on a specific user (i.e. userID “M3NKfBm9kR”)

B)

```
get_most_recent_user(recent_user_heatmap);  //use for most recent user
```

This generates a query on the most recent user

## Scrum

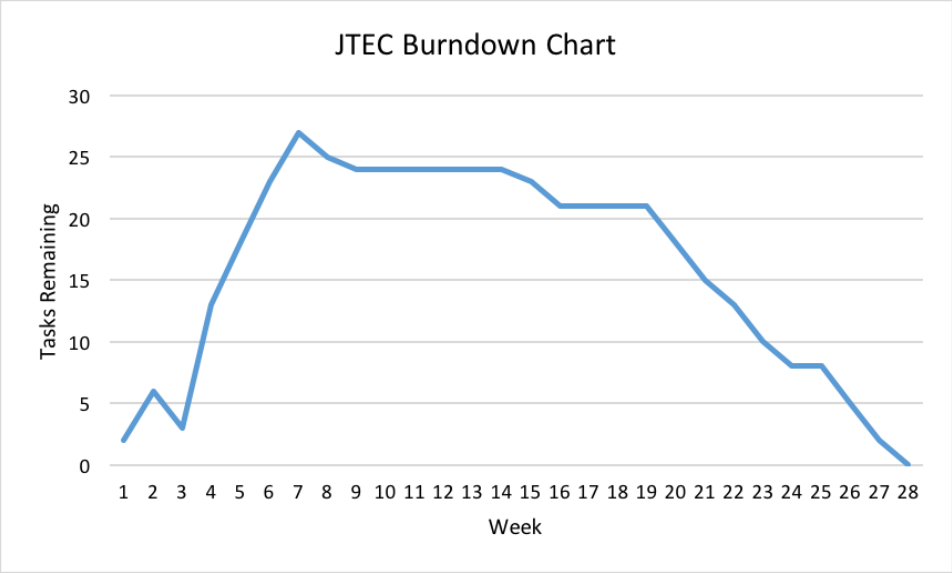


Figure 1: Burndown Chart

It can be seen by our burndown chart that our first few weeks consisted of adding many tasks. This was due to the steep learning curve of our project. As we went on learning the new technologies, we created new tasks that had to be done in order to complete the project, that we had not known before. It can be seen that there was a plateau period around the beginning of December through January. This was due to the holiday season and winter break. Also there is another plateau just around spring break. The overall velocity for our project was not very consistent, though the last few months were very productive.

Project tasks were divided evenly among the members of our group. John and Terence worked with one another on integrating WebGazer, calibration, validation, and building our application into a website. Elias and Chris worked on setting up the server, setting up the database, pushing to the database, querying the database, and generating analytics.

We accomplished most of the goals and user stories that we set out to. These include but are not limited to: creating a backend, calibration, validation, and some brief analysis. We would have maybe liked to have improved our analysis capabilities by adding more extensive plots, possibly with a time aspect.

Overall, this was a challenging project to work on. The initial learning curve was very steep, but after we learned the technologies we had a much clearer vision of how to execute the project. Throughout the implementation phase we hit a few hiccups, such as having to execute everything through HTTPS, issues successfully pushing data to the server from a real trial, among others, though we worked as a team to figure them out. In the end we are happy with the final result.