COEFFICIENT | SUMMER OF CODE 2020 PRODUCTION | CLIENT: PETER HALPIN

TEAM: CHRISTINA ALPERI, ROHAID BAKHEET, JUSTIN DO, NATHAN MULLANEY

INTRODUCTION: THE PROBLEM

Coefficient is a web application that instructors can use to host web activities for the classroom, as well as make these activities more collaborative. In a traditional classroom setting, students can gather around one computer in a group and discuss an activity. However, the rise of distance learning has presented challenges to this model. With students unable to physically share a computer with their group members, a number of questions are raised.

- How can we make sure each student can access the activity on their computer?
- How can we divide students into groups without physically doing so?
- How can we emulate the experience of working together when everyone is geographically distributed?

These are the questions that Coefficient was made to address. The word root “co” indicates a sense of togetherness, and we hope that this app makes it “efficient” to facilitate this feeling in the online classroom.

INTRODUCTION: THE PRODUCT

The idea behind Coefficient is simple. Instructors will prepare a static web project that they intend to use for group work. The basic elements of such a project include:

- HTML file named “index.html” (REQUIRED)
- CSS file for styling (optional)
- JavaScript file to facilitate interaction (optional)
- CSV or JSON file to include external data (optional)

Once logged into Coefficient, instructors are able to upload their project files and assign a name for their game's activity. The HTML file will be modified to embed the functionality of TogetherJS - a library that allows users to jointly interact with the same instance of a web page, view each other's cursors, and chat with one another. Think of the collaborative aspect of Google Docs, and imagine how it might work on a web page. That's what TogetherJS does. After TogetherJS is added, the TogetherJS-enabled project will be hosted and a URL will be provided to access the activity.

Instructors are able to view a list of every project they have published, and add or delete projects as necessary. When it is time for them to use an activity in the classroom, they can specify the number of groups that they want to create and the application will generate that number of unique URLs, each of which corresponds to a unique session of that activity. Students can then be assigned a URL, and when they visit the URL they will be able to chat and interact with their group members, who were assigned the same URL. All information about student interactions and chat messages can be optionally logged, providing opportunities to research how students interact with each other in the virtual classroom.

So, to summarize, Coefficient provides the following value to instructors:

- Ability to host static web applications with minimal knowledge about servers and hosting.
- Easily manage and access all class activities.
- Allow students to collaborate in groups on a web activity without the need for an external communication device.
- Generate unique URLs for an activity which correspond to different groups of students.
- View interactions between students on published web activities.

INTRODUCTION: THE TECHNOLOGIES

A number of technologies were used to make Coefficient happen. In this section, we discuss the components of the project, and the technologies behind them.

The Front-End

The instructor-facing Coefficient application is where activities may be uploaded and activity session URLs may be generated. The front-end is rendered with standard HTML, CSS and JavaScript, and it primarily serves as the gateway to interact with the content on the backend. To accomplish this, the following SDKs were used: Firebase and Amazon Web Services.

TogetherJS

TogetherJS is an open-source Mozilla library that enables the functionality detailed in the introduction. TogetherJS operates on a client-server model, in which clients for a particular session relay information about page state and user actions to the server. The server then relays this information to all clients in the same session and synchronizes their browser state. This functionality is limited to static web pages, so projects that utilize data or dynamic elements must do so through files local to the project.

The client code can be added to any HTML web page by simply including a script tag pointing to the client code (currently hosted by Mozilla here) in the HTML's header. The server code must be constantly running in order to respond to requests, which is achieved by publishing it as a Node application. Mozilla is not currently hosting a TogetherJS server, so you must publish your own through a service such as Heroku or Glitch.

Selenium

TogetherJS session URLs are generated by visiting a TogetherJS-enabled web page, interacting with the TogetherJS client, and copying the URL corresponding to your session. This functionality works for the common use case of visiting a website and sharing the session URL with a friend so that you can browse together, but it means that session URLs cannot be pre-generated without manually visiting the website. Selenium is a web-scraping tool that allows you to programmatically interact with web pages, and we use it to automate the process of visiting a TogetherJS-enabled web page and retrieving the session URL. With Selenium, we can generate however many sessions URLs we want and distribute the URLs, as if they were pre-generated to begin with.

Firebase

Firebase is a platform that offers a number of web services to be used in applications. In particular, we made use of Firebase Authentication through Google accounts to manage the information saved for each user, and we used Firebase Cloud Firestore to manage a table of each user's uploaded activities and the URLs associated with them.

Amazon S3

Amazon's Simple Storage Service (S3) is a cloud-based object storage service that lets users store files into unique "buckets". Each bucket may optionally be accessed as a static web page through which the objects in the bucket may be viewed in the browser. In essence, each bucket acts as a miniature hosted website with a distinct URL and access to the resources it contains. For each activity an instructor uploads, an S3 bucket is created, the project's files are uploaded to the bucket, and the bucket is configured to host the project files. Once this is done, the S3 bucket's URL is stored in the Firebase database so that the user may access it later.

Amazon Lambda

Lambda is a function-as-a-service provider that allows you to upload code and the proper environment, then execute the code on demand. With Coefficient, a user should be able to generate TogetherJS session URLs on demand. To accomplish this, our Selenium script (written in Python) was published as a Lambda function and configured to execute when called through a RESTful API created in Amazon API Gateway.

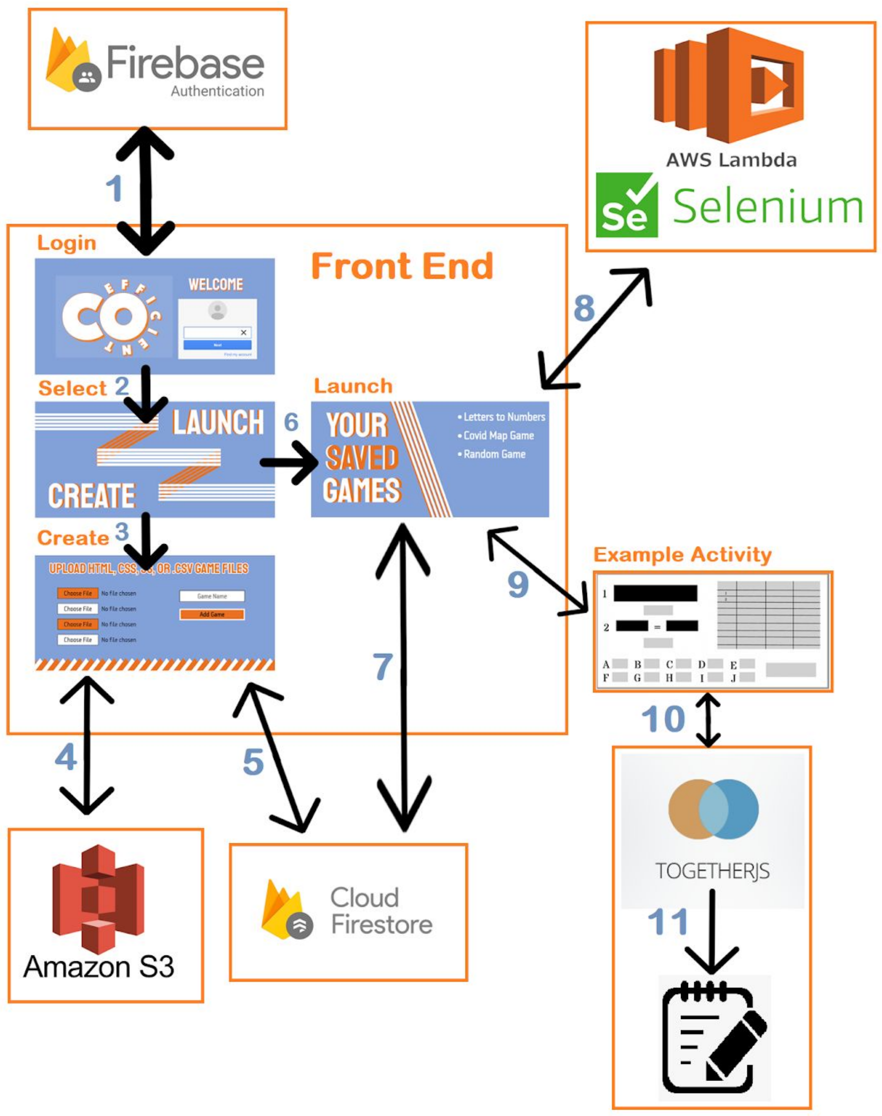
Letters-to-Numbers

To demonstrate the use of this application, we developed a web activity called Letters-to-Numbers, in which students perform addition and subtraction operations on a set of letters A-J, each of which corresponds with a number from 0-9. Gradually, they work together to figure out which letter is mapped to which number, and must guess correctly within 10 turns.

This application serves as an example of an activity that an instructor may create on their own and then upload through Coefficient in order to add TogetherJS functionality.

INTRODUCTION: THE DIAGRAM

There are many pieces to this project that interact with each other, so we provide this diagram to illustrate the dataflow of the application and when/where each technology is used. Each arrow represents the flow of data, and the number next to each arrow maps to a textual description of what is happening at that arrow point.



1) The Login page communicates with Firebase Authentication in order to allow a user to either log in or create an account.
2) After logging in, the user is taken to the selection page. They can either go to (3) and upload a new activity or go to (6) and launch a previously uploaded activity.
3) By clicking the “Create” button, you are taken to the activity upload page.
4) After selecting all the files to upload, a bucket is created on Amazon S3 and the files are uploaded there. The URL for the bucket is returned to the page.
5) Once the URL has been returned to the page, the URL and the current user’s ID are posted into Firebase Cloud Firestore.
6) By clicking the “Launch” button, the user is taken to the activity launch page.
7) Firebase Cloud Firestore is queried, and all of the activities uploaded by the current user are rendered in a list.
8) After choosing a game and the desired number of URLs, a call is made to the Selenium script on Amazon Lambda. The URLs are then rendered on the page.
9) By visiting one of the returned URLs, the user is taken to a TogetherJS-enabled web page.
10) TogetherJS client information is sent to the TogetherJS server; information about users in other sessions is relayed back to the client for synchronized state.
11) Each TogetherJS event is logged to a file alongside the server code.

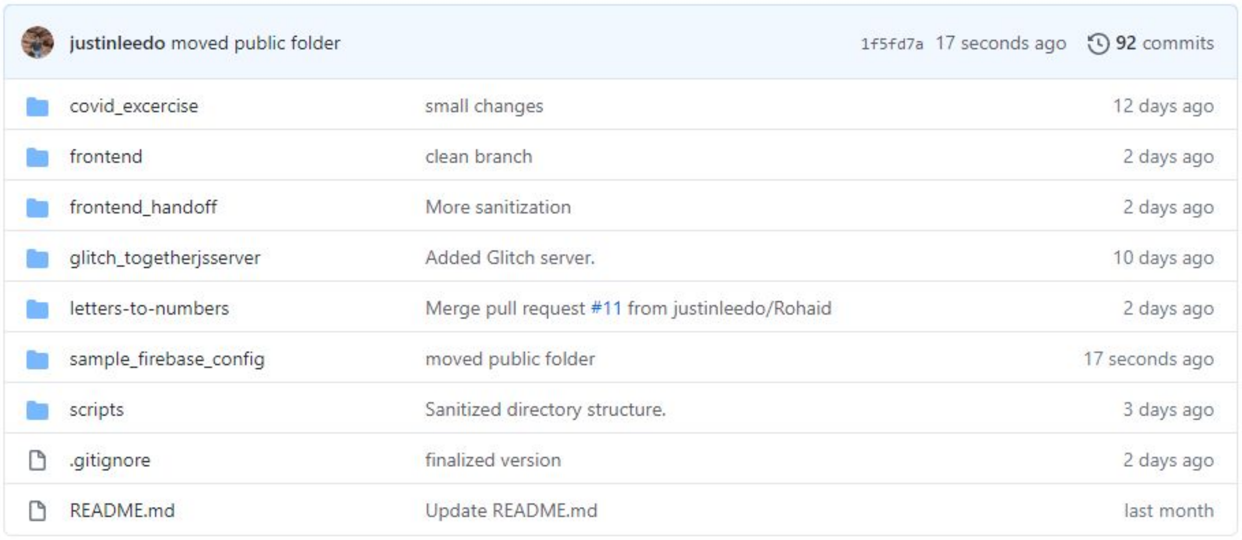
COEFFICIENT: THE EXPERIENCE

To use Coefficient, you must use a special version of Google Chrome that has CORS-functionality disabled due to some of the cross-origin API calls that are made. For instructions on how to do this, view this page.

When you navigate to the home page of Coefficient, you are greeted with a Firebase Authentication login widget asking for an email and password to make an account. After proceeding with sign in, you will be given two options: to “Create” or “Launch” a game. Creating a game allows you to upload an HTML, CSS, JS, and a CSV or JSON file to render the web app and then choose a name for it to be saved in the database. Adding a game will automatically redirect you to the “Launch” page from the previous step. This page includes a list of all the games saved in the database associated with the current user’s account. You can select a game to play and type in a number of groups for the session manager to generate that number of unique URLs. After a short waiting period, the page automatically reloads with the URLs hyperlinked to the game for each group. The URLs are special in that they have added Together JS functionality that allows the game to be multiplayer, so anyone on a specific link will have interactivity with the other players. On the same “Launch” page you can also delete a game from the list or navigate back to the upload page with the “Upload New Game” or “Delete Old Game” buttons.

GITHUB: NAVIGATING THE REPOSITORY

The following is a picture of the structure of this project's GitHub repository.



justinleedo moved public folder 1f5fd7a 17 seconds ago 92 commits

covid_excercise	small changes	12 days ago
frontend	clean branch	2 days ago
frontend_handoff	More sanitization	2 days ago
glitch_togetherjsserver	Added Glitch server.	10 days ago
letters-to-numbers	Merge pull request #11 from justinleedo/Rohaid	2 days ago
sample_firebase_config	moved public folder	17 seconds ago
scripts	Sanitized directory structure.	3 days ago
.gitignore	finalized version	2 days ago
README.md	Update README.md	last month

The following is a brief description of which project components can be found in each folder:

- **covid-exercise:** Framework of an exercise that makes use of Google's Geolocation API. Not complete due to timing constraints, but an explanation of the code is present at the end of the documentation.
- **frontend:** Version of the front-end that we deployed and used for testing purposes. Contains references to keys and access codes that were used in testing, but should not be used in the final version.
- **frontend_handoff:** A cleaned-up version of the front-end code that removes all references to specific access keys and configuration settings. This is what you should use when deploying the project on your own, and you will need to input your own configuration settings.
- **glitch_togetherjsserver**: The TogetherJS test server that was uploaded to Glitch for testing. The file ***server.js*** is the one that acts as a server for all TogetherJS requests and will need to be hosted somewhere.
- **letters-to-numbers:** All of the files needed for the cryptographic Letters To Numbers activity. Used as a test activity to upload through the Coefficient front-end.
- **sample_firebase_config:** Configuration files used for local deployment. You should not need to use these, as you will generate your own configuration files when you deploy, but they may be useful in showing the file structure you can expect when deploying.
- **scripts:** The Selenium script in Python that is used to web-scrape TogetherJS pages for session URLs. Includes: .zip file containing the entire environment necessary to upload to AWS Lambda; a version of the script to run on AWS Lambda; and a version of the script that can run in a local terminal.

TOGETHERJS: INTRODUCTION

TogetherJS is the lynchpin of this product. It is what allows students to communicate and interact with one another while visiting a web page. The Coefficient front-end does not use TogetherJS itself; it simply adds the functionality to other HTML pages. The easiest way to understand what TogetherJS does is to read the official documentation or watch this video.

TOGETHERJS: THE CLIENT

Adding TogetherJS to an HTML page is very simple. All you need to do is host an instance of the TogetherJS server somewhere and add the following HTML tags to the page you would like to use TogetherJS with:

```
<script>TogetherJSConfig_hubBase =
"https://sustaining-classic-beam.glitch.me/"; </script>
```

This script tag adds a pointer to the location of the TogetherJS server. The above URL is a sample server that we used during development, but a new server should be hosted for deployment.

```
<script> TogetherJSConfig_suppressJoinConfirmation = true </script>
```

This tag prevents a pop-up from loading on the TogetherJS-enabled page when you first visit it. The purpose of this is to give less work for the web scraper, which must interact with the page in order to generate session URLs.

```
<script> TogetherJSConfig_autoStart = true </script>
```

This tag makes the TogetherJS client load on the web page immediately rather than appearing through the click of a button. Again, the purpose of this was to give less work to the web scraper, and to have fewer steps in the way of students using the TogetherJS features.

```
<script src="https://togetherjs.com/togetherjs-min.js"></script>
```

This script tag adds a pointer to the TogetherJS client, and is what actually loads the TogetherJS functionality in the web page. The URL above points to a Mozilla-hosted version of the client, but you may want to host this file somewhere else in case it gets taken down.

The way that the Coefficient front-end adds these HTML tags to an HTML page is very simple. Before the HTML file is uploaded and hosted, the Coefficient client parses through the contents of the HTML file and searches for the </head> tag. It then replaces the tag with all of the above script tags, followed by </head>. The net result is that the script tags get appended right before the ending of <head>.

NOTE THAT YOU MUST HAVE A HEAD TAG IN YOUR HTML FILE FOR TOGETHERJS TO BE SUCCESSFULLY ADDED. THE CONTENTS CAN BE BLANK, BUT YOU MUST HAVE <HEAD> HEADCONTENTS </HEAD> **SOMEWHERE IN YOUR FILE.**

TOGETHERJS: THE SERVER

TogetherJS synchronizes what is displayed on everybody's page by having the client send information about the session to the server and then relaying the information back to the client of everybody who is on the same TogetherJS session. That being said, the server must be up and running somewhere in order for this to happen. All that is necessary is to get the server code hosted as a Node application. Heroku and Glitch are examples of services where this can be done. We used Glitch for development, but it is a free service and the server will go to sleep when it isn't receiving requests. The details of setting up one of these servers is beyond the scope of this documentation, but it is pretty simple. This video demonstrates how the server was set up on Glitch, and the content of our development Glitch project is available in the repository.

The server code provides the ability to log all of the requests it receives from clients, meaning that information such as mouse location and chat messages from each student can be output to an external file. This does not occur automatically; it is a feature you must enable. This is done by modifying the constructor of the Logger object in the server code. By default, it is:

```
var logger = new Logger(0, null, true);
```

The first parameter corresponds to the level of verbosity that will be logged on a scale from 0 to 5 (definitions are present in the server code) and the second parameter corresponds to the name of the local file that logs should be output to. So we can log information to the file by changing this line to something like:

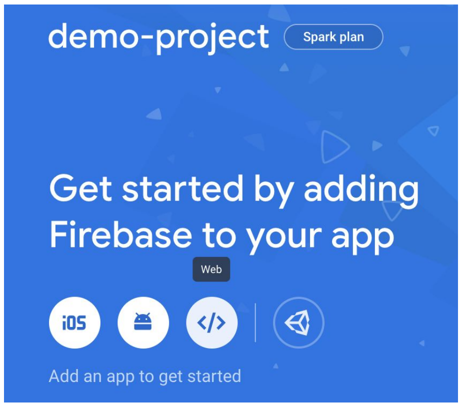
```
var logger = new Logger(0, "output.txt", true);
```

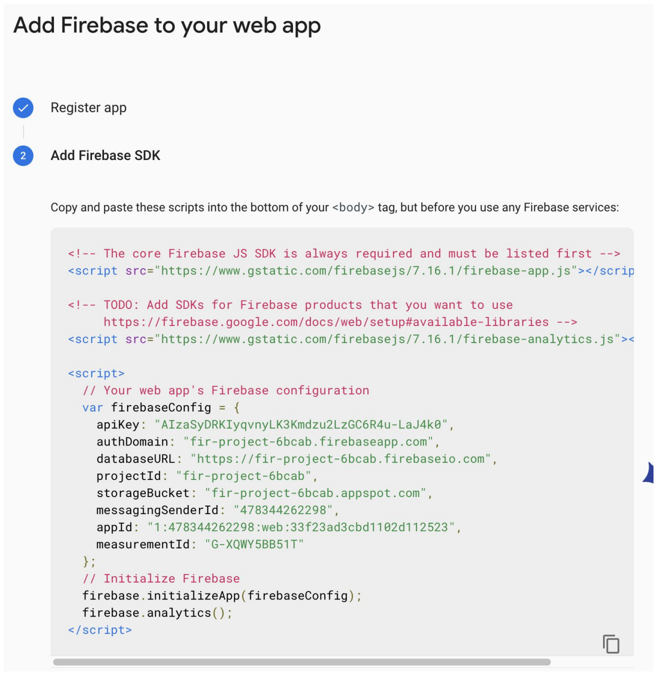
The logs can then be parsed through in this file to see what occurred in each session. This functionality fell outside the scope of this project, but it should be relatively simple to convert the data into a more human-readable format.

FIREBASE AUTHENTICATION AND DATABASE: SETUP

Firebase offers this project the ability to store and manage multiple accounts for instructors to upload and store their activities without having to manage a complicated authentication infrastructure.

You will start by having a Gmail account to login to Firebase. Navigate to the console and follow the official documentation to get started creating a project. Go ahead and enable Firebase Analytics and choose the default Firebase account to send updates from.


demo-project
Spark plan
Get started by adding Firebase to your app
Web
iOS
Add an app to get started


Add Firebase to your web app

Register app

2 Add Firebase SDK

Copy and paste these scripts into the bottom of your <body> tag, but before you use any Firebase services:

```
<!-- The core Firebase JS SDK is always required and must be listed first -->
<script src="https://www.gstatic.com/firebasejs/7.16.1/firebase-app.js"></scrip

<!-- TODO: Add SDKs for Firebase products that you want to use
     https://firebase.google.com/docs/web/setup#available-libraries -->
<script src="https://www.gstatic.com/firebasejs/7.16.1/firebase-analytics.js"><

<script>
  // Your web app's Firebase configuration
  var firebaseConfig = {
    apiKey: "AIzaSyDRKIyqvnyLK3Kmdzu2LzGC6R4u-LaJ4k0",
    authDomain: "fir-project-6bcab.firebaseapp.com",
    databaseURL: "https://fir-project-6bcab.firebaseio.com",
    projectId: "fir-project-6bcab",
    storageBucket: "fir-project-6bcab.appspot.com",
    messagingSenderId: "478344262298",
    appId: "1:478344262298:web:33f23ad3cbd1102d112523",
    measurementId: "G-XQWY5BB51T"
  };
  // Initialize Firebase
  firebase.initializeApp(firebaseConfig);
  firebase.analytics();
</script>
```

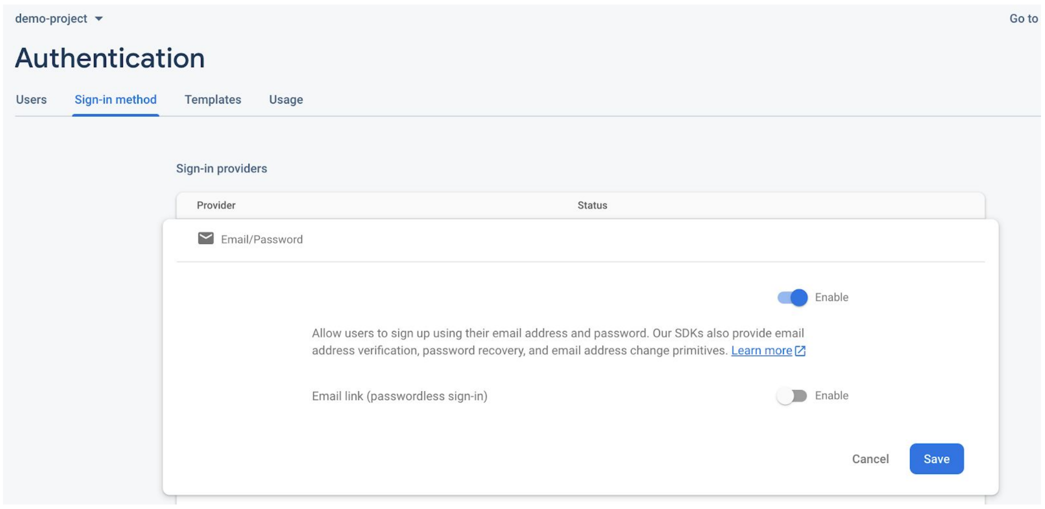
Click on the </> “Web” tag and then choose a project nickname to get to the Firebase SDK step. Copy and paste the given scripts into the bottom of your <body> tag of any of your web app’s HTML page(s), but before you use any Firebase services. Above the firebase configuration object is the place to put scripts to the Firebase SDKs. We will be using Firebase Authentication and Firebase Firestore and the Firebase Authentication UI.

Add the below CDNs to the <head> tag of every HTML page in your web app’s folder.

```
<!--Import Firebase libraries-->
       <script
src="https://www.gstatic.com/firebasejs/5.0.3/firebase-app.js"></script>
       <script
src="https://www.gstatic.com/firebasejs/5.0.3/firebase-auth.js"></script>
       <script
src="https://www.gstatic.com/firebasejs/5.0.3/firebase-firestore.js"></scri
pt>
       <!--Import Firebase UI-->
       <script
src="https://cdn.firebase.com/libs/firebaseui/3.5.2/firebaseui.js"></script
```

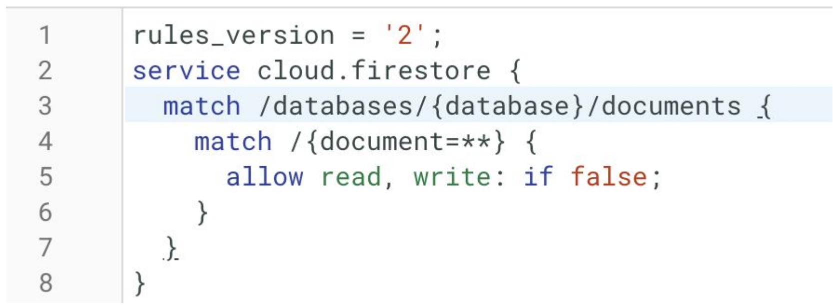
```
>
       <link type="text/css" rel="stylesheet"
href="https://cdn.firebase.com/libs/firebaseui/3.5.2/firebaseui.css" />
       <!--Lastly, reference personal style sheet-->
       <link rel="stylesheet" href="styles.css">
```

Go to the console for your new project and under Authentication, enable the email/password sign in method (as shown in the image). The purpose of adding Firebase Authentication is that it allows us to have multiple teacher accounts for the session manager that uses a database to store all of the web apps teachers want to host with added Together JS functionality. It is simple, trusted, and well-documented.


demo-project
Go to
Authentication
Users
Sign-in method
Templates
Usage
Sign-in providers
Provider
Status
Email/Password
Enable
Allow users to sign up using their email address and password. Our SDKs also provide email address verification, password recovery, and email address change primitives. Learn more
Email link (passwordless sign-in)
Enable
Cancel
Save

Now we will set up the database that stores the games. In the console, navigate to the Database tab and create a cloud database, following these instructions. Start in test mode for open read/write access during setup and then choose your proper time zone to create the database. Ignore the “Setup your development environment” step in the documentation, as we did that by including the appropriate CDNs in the above step. You don’t need to manually create any entries in the database, as we will set up write functions to do that.

Note: After you deploy the whole Firebase project to its own URL, the default security rules on the database will change to the following:



```
1    rules_version = '2';
2    service cloud.firestore {
3      match /databases/{database}/documents {
4        match /{document=**} {
5          allow read, write: if false;
6        }
7      }
8    }
```

To fix this, in line 5 delete the “if false” rule : `allow read, write;`

The functionality of the Firebase database is exposed in the front-end, meaning write access is enabled for all users but the organization of reading games associated to one’s account is determined by the unique userID field stored with each game in the database.

FIREBASE HOSTING: SETUP

Firebase hosting will allow us to deploy the final application to a server in order to save teacher accounts and store the database containing the game URL, title, and user ID for each game. To enable this, follow along with the official quickstart guide. Choose the option to add the Firebase SDK to enable Cloud Firestore and Firebase authentication. After creating an app nickname (which will be the hosted URL) and initializing the project you will be able run a series of commands from the Firebase CLI in order to serve the app locally and later deploy globally.

Now you have a Firebase project that can authenticate multiple users, read and write data to the database, and be hosted to a unique URL for use.

AMAZON WEB SERVICES: SETUP

In order to use the requisite AWS features (S3, Lambda, API Gateway), there will be some setup involved. Namely, you must have an AWS account, a profile on the account with administrative rights, and the access keys associated with your accounts. All of this is necessary in order to create / modify buckets on S3 and deploy projects on Lambda and API Gateway.

First, you will need to create and activate an AWS account. In the process of doing so, you will be asked to enter in payment information for billing. AWS users have access to a free service tier for the first year of account ownership, and afterwards are billed for the storage / bandwidth / computational power that is used. The Coefficient activities that will result in charges include: creating a bucket, uploading files to buckets, keeping files in storage, accessing a bucket’s hosted content, and making API calls to the web scraper. Since Coefficient will have a small user base, the cost should be negligible; however, pricing details are available here.

To make use of AWS, you will need to create an IAM (Identity and Access Management) user with administrative rights. The instructions are available in the official AWS documentation under the heading called “Creating an Administrator IAM User and Group (Console)”. When you reach Step 4b, you should check the box that says “Programmatic access” in addition to the one that says “AWS Management Console access”, contrary to what the instructions say. When you finish these steps, you will be able to download a .CSV file containing your account information, Access key ID, and Secret access key. Additionally, you have the ability to log in to the AWS console as an IAM Administrator user. ***Make note of the Access key ID and Secret access key. They must be entered in the Coefficient front-end code before the front-end can programmatically interact with AWS with JavaScript’s AWS SDK.***

The place where the keys should be entered in the front-end code will look like:

```
AWS.config = new AWS.Config();
AWS.config.accessKeyId = "";
AWS.config.secretAccessKey = "";
AWS.config.region = 'us-east-1';
```

Note that it is best practice to NOT write your keys directly in these fields in a deployed version of the application, for security purposes. For more details on how to safely set your keys, please read this documentation.

Once you have completed the above steps, you will be ready to configure some actual Amazon Web Services.

AMAZON WEB SERVICES: S3

S3 is the mechanism used to upload activities from the front-end and then have them then be hosted. All file uploads and bucket configurations are done programmatically through the front-end with the AWS SDK, so there are few things that must be configured on the AWS website. Here, we present a brief guide of some of the configurations that are done through the front-end so that it is more clear what the front-end code is doing. For additional information, it is recommended that you read the official documentation (notable sections include: Introduction, Making requests, Buckets, and Hosting a static website).

The main idea of a bucket is that you can upload files (objects) to it, and the files can then be accessed from a URL corresponding to the bucket. In the case of static HTML pages, the page is displayed in the browser. However, this functionality does not come right out of the box; there are further steps that must be taken after creating a bucket. Here, we explain what the steps are and identify the corresponding lines of front-end code in ***app.js*** that make it happen.

Bucket Creation

The basic JavaScript code for creating a bucket looks like this:

```
var bucketParams = {
            Bucket : 'coefficient' + gameName,
            ACL : 'public-read'
     };

await s3.createBucket(bucketParams, function(err, data) {
            if (err) {
            console.log("Error", err);
            } else {
            console.log("Success", data.Location);
            }
        });
```

Here, the name of the bucket and its public access settings are set in the JavaScript object *bucketParams* and the bucket is created with the *createBucket* function call. Note that the value corresponding to *Bucket* in *bucketParams* will be the name of the bucket. Bucket names are unique across the entire world, so you cannot name your bucket a name that has already been taken. The concatenation of “coefficient” and an activity name should be unique enough to not exist already, but if you have issues creating a bucket, it is likely because there is already a bucket with that name. Bucket names should be in all lowercase. In the front-end application, it is important to make sure that all references to a bucket name are the same. The default name of “coefficient” concatenated with the activity name should suffice, but if you want the bucket naming scheme to be different, you will have to modify each line of front-end code that mentions this scheme.

Make Bucket a Static Website

After creating a bucket, you need to configure it to act as a static website that hosts its contents. The JavaScript code for this is as follows:

```
var websiteParams = {
            Bucket: "coefficient" + gameName,
            ContentMD5: "",
            WebsiteConfiguration: {
             ErrorDocument: {
                  Key: "error.html"
                    },
                    IndexDocument: {
                        Suffix: "index.html"
                          }
                  }
           };
```

```
await s3.putBucketWebsite(websiteParams, function(err, data) {
                    if (err) console.log(err, err.stack);
                    else    console.log("success" + data);
           });
```

Configuration settings for the website are set in *websiteParams*, and the *putBucketWebsite* function call applies the settings to your bucket. Of note, in *websiteParams*, the value of the *Bucket* should be the same as the name of the one that was just created. The values of *ErrorDocument* and *IndexDocument* specify the landing page and error page of the website once it is hosted. By default, the HTML page that you upload will be renamed to *index.html*, so this should always work.

Creating Bucket Policy

The last piece of configuration is setting a viewing policy for your bucket so that anybody is able to access the contents from the URL. This mostly boils down to boilerplate code, which is as follows:

```
var policy = {
                Version: "2012-10-17",
                Statement: [
                    {
                        Sid: "PublicReadGetObject",
                        Effect: "Allow",
                        Principal: "*",
                        Action: "s3:GetObject",
                        Resource: "arn:aws:s3:::coefficient"
+ gameName +"/*"
                    }
                ]
        };
        var policyParams = {
            Bucket: "coefficient" + gameName,
            Policy: JSON.stringify(policy)
        };

await s3.putBucketPolicy(policyParams, function(err, data) {
                    if (err) console.log(err, err.stack);
                    else    console.log("success" + data);
      });
```

As we saw previously, the *Resource* and the *Bucket* fields must have values corresponding to the actual name of your bucket. Once you do this, you are ready to upload some files to your bucket.

Uploading Files to Bucket

The process for uploading files is relatively formulaic and the same for every file, so we only show what is done to upload the HTML file:

```
var uploadParams = {
            Bucket: "coefficient" + gameName,
            Key: 'index.html',
            Body: togetherJS_String,
            ContentType: 'text/html'
       };

       await s3.upload (uploadParamsJS, function (err, data) {
if (err) { console.log("Error", err); }
if (data) { console.log("Upload Success", data.Location); }
       });
```

The parameters of *uploadParams* are pretty self-explanatory. *Bucket* should receive the value of the bucket you are uploading to, *Key* should be the name of the uploaded file, and *Body* should be the contents of the uploaded file. *ContentType* is not strictly necessary, but is shown here to emphasize that an HTML file is what gets uploaded.

In the front-end code, the *Key* is always the same for a particular type of file. This is to help with consistency. All HTML files will be named ***'index.html'***, all CSS files will be named ***'styles.css'***, all JavaScript files will be named ***'app.js'***, and all CSV files will be named ***'data.csv'***. This means that if the uploaded files do not originally have these names, the uploaded file will have a different name and references to the old name will not work. The best practice is to assign your activity files these names while developing them, so that nothing changes when they are uploaded.

Bucket URLs

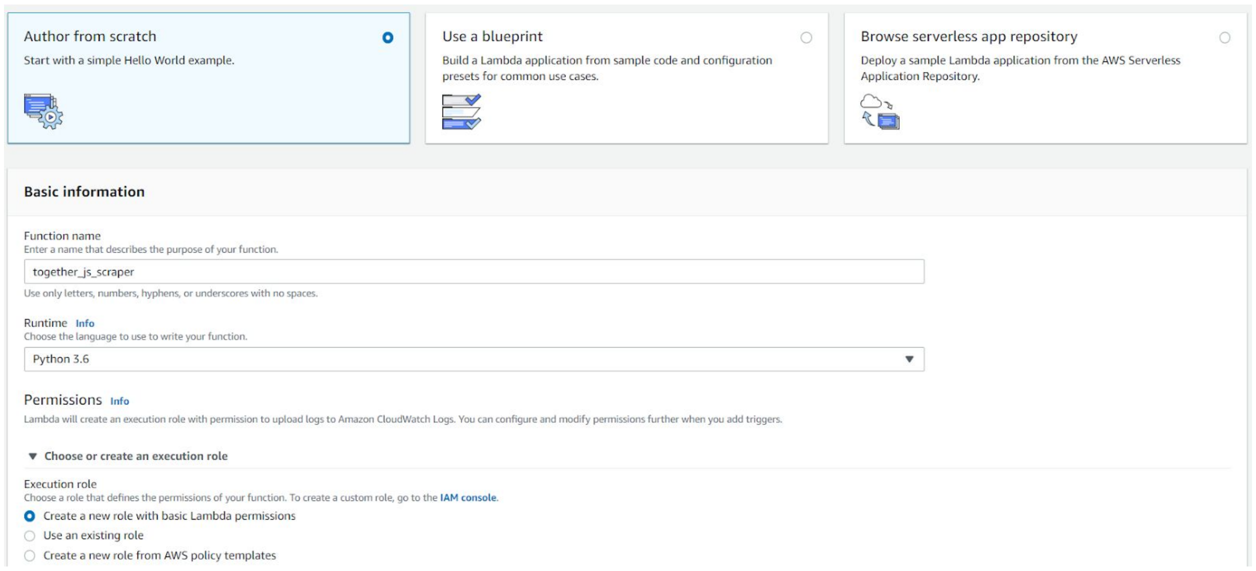
When you upload an activity to a bucket, the URL where the bucket can be accessed will be posted to the Firebase database. The format of the bucket URL should always be in the format:

http://<BUCKET NAME>.s3-website-us-east-1.amazonaws.com

AMAZON WEB SERVICES: LAMBDA

Lambda is used to host the Selenium web scraper, which generates session URLs for a TogetherJS-enabled web page. Lambda provides users with the ability to simply upload their code and all of its dependencies / libraries in a .zip file and it will automatically create the environment needed for the code to run on demand. We provide the .zip file with the Selenium script and all of its dependencies in a file called ***“lambdaDeploymentEnvironment.zip”***. To get things up and running, follow these instructions.

First, you will need to select Lambda from the list of Services. It will take you to the AWS Lambda landing page, where you can see all of your Lambda functions. The only one necessary is the one we are about to create, so click “Create function”. On the following page, enter the following settings:



[✓ Author from scratch]
Start with a simple Hello World example.

Use a blueprint
Build a Lambda application from sample code and configuration presets for common use cases.

Browse serverless app repository
Deploy a sample Lambda application from the AWS Serverless Application Repository.

Basic information

Function name
Enter a name that describes the purpose of your function.
together_js_scraper
Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime Info
Choose the language to use to write your function.
Python 3.6

Permissions Info
Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. You can configure and modify permissions further when you add triggers.

▼ Choose or create an execution role

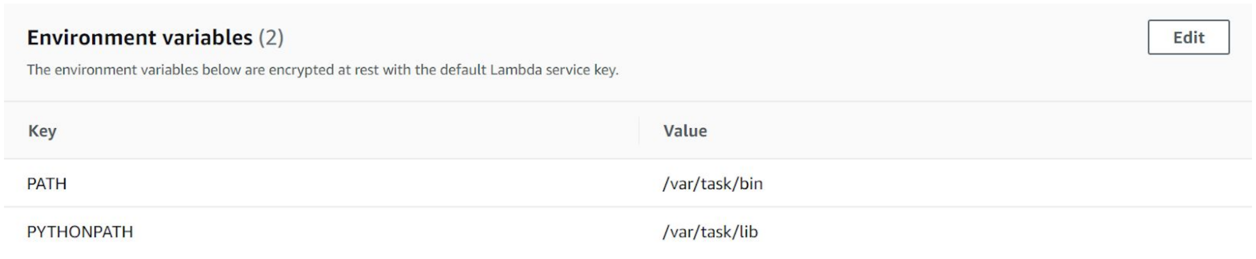
Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.
[✓ Create a new role with basic Lambda permissions]
Use an existing role
Create a new role from AWS policy templates

Advance to the next page, and you reach the Configuration page for your Lambda function. Do not bother with the text editor, as we can simply upload the .zip file. Scroll down to the section called “Function code” and select the box that says “Actions -> Upload a .zip file”. Upload the .zip file.

Once the .zip file has been successfully uploaded, scroll down to the section called “Environment variables”. Enter the following <KEY:VALUE> pairs (omit the quotation marks):

- PATH: “/var/task/bin”
- PYTHONPATH: “/var/task/lib”

The section should look like this when you are done:

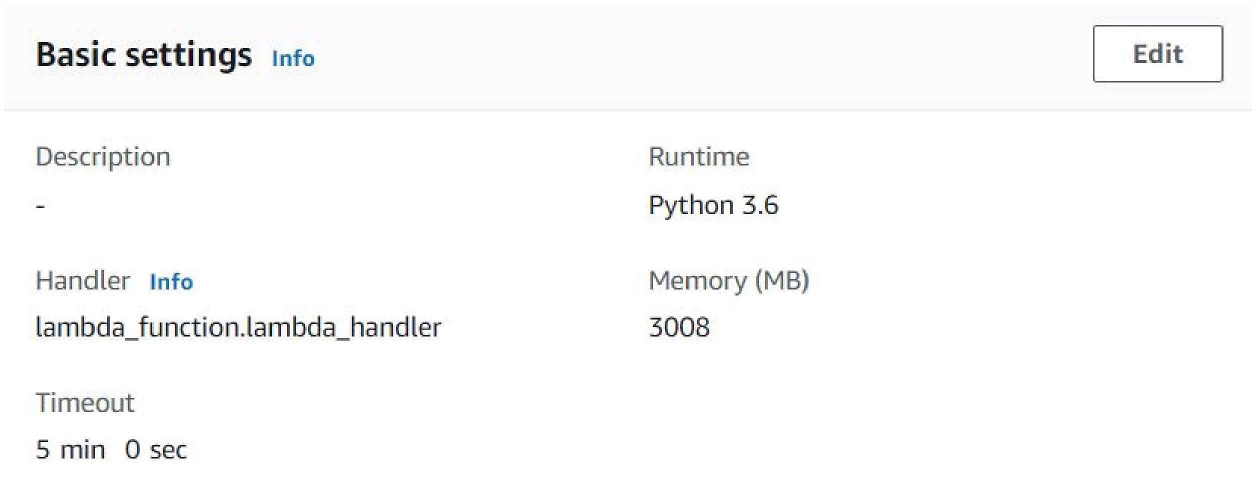


Environment variables (2)

The environment variables below are encrypted at rest with the default Lambda service key.

Key	Value
PATH	/var/task/bin
PYTHONPATH	/var/task/lib

You will also have to change some of the fields in “Basic Settings”. Click on the “Edit” button and set the “Timeout” to some long period of time, such as 5 minutes. Set “Memory” to be as large a value as possible. The Selenium script takes a couple of seconds to generate each URL, so the default “Timeout” value of 3 seconds will not be enough time in most cases.



Basic settings

Description: -

Runtime: Python 3.6

Handler: lambda_function.lambda_handler

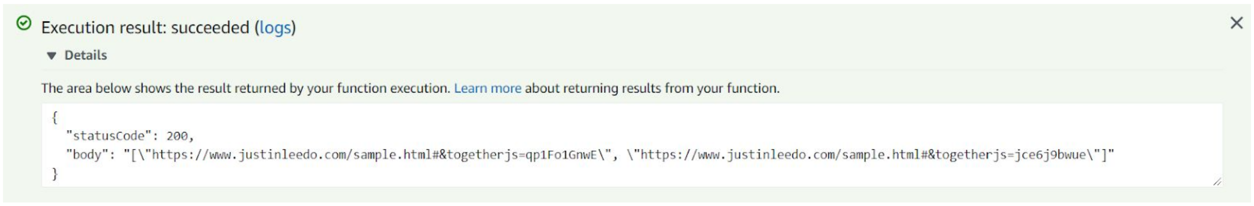
Memory (MB): 3008

Timeout: 5 min 0 sec

Finally, you can test that the Lambda function works. Scroll to the top of the page, and in the box called “Test Events”, click on it and add a new test event. The Lambda function takes two parameters: the URL to make sessions for (url), and the number of sessions to make (groups). Format the test event JSON to look something like the following: (can change the URL and number of groups to something different):

```
{ "queryStringParameters":
    {
      "url": "https://www.justinleedo.com/sample.html",
      "groups": 2
    }
}
```

If all is successful, you should see a response that includes the specified number of TogetherJS URLs, such as below:



Execution result: succeeded (logs)

Details

The area below shows the result returned by your function execution. Learn more about returning results from your function.

```
{
  "statusCode": 200,
  "body": "[\"https://www.justinleedo.com/sample.html#&togetherjs=qp1Fo1GnwE\", \"https://www.justinleedo.com/sample.html#&togetherjs=jce6j9bwue\"]"
}
```

Now that we know our Lambda function is working, the next step is to set up an endpoint so that the function can be called from the Coefficient front-end.

AMAZON WEB SERVICES: API GATEWAY

Amazon API Gateway is a service that lets users create endpoints for served code, through HTTP or REST requests. For this project, we will make a REST endpoint such that when a request is made to a particular URL, our Lambda function executes and returns the generated URLs in the response body.

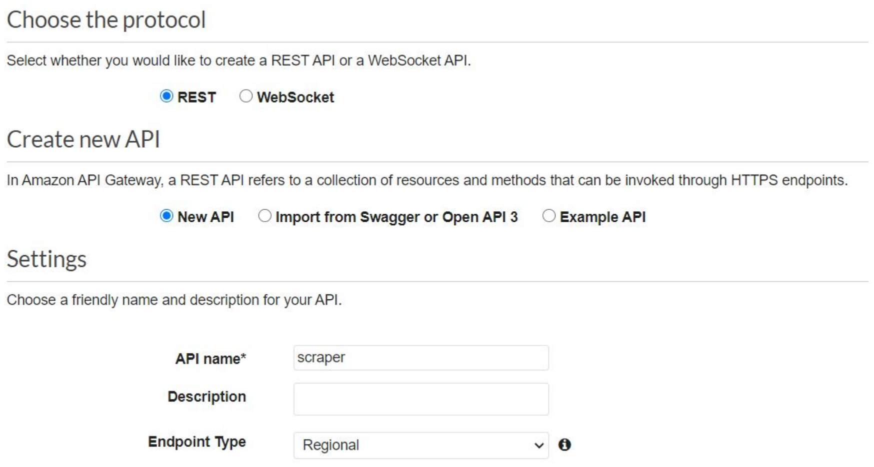
The URL will take the form:

<Deployed API URL>?url=<TogetherJS URL to Generate>?groups=<Number of Groups>

As an example, the following URL will make a request to an old version of the API we deployed and generate 5 TogetherJS URLs for the page at https://www.justinleedo.com/sample.html.

https://oxreg1dkaa.execute-api.us-east-1.amazonaws.com/prod/scraper?url=https://www.justinleedo.com/sample.html&groups=5

To begin the process of creating our endpoint, select the “API Gateway” service from the list of Amazon Web Services. Select the button that says “APIs”. The following page will ask you to select an API type. Find the box that says “REST API” and select “Build” within that box. Make the following selections in the page that follows and click “Create API”:



Choose the protocol

Select whether you would like to create a REST API or a WebSocket API.

[✓ REST] WebSocket

Create new API

In Amazon API Gateway, a REST API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

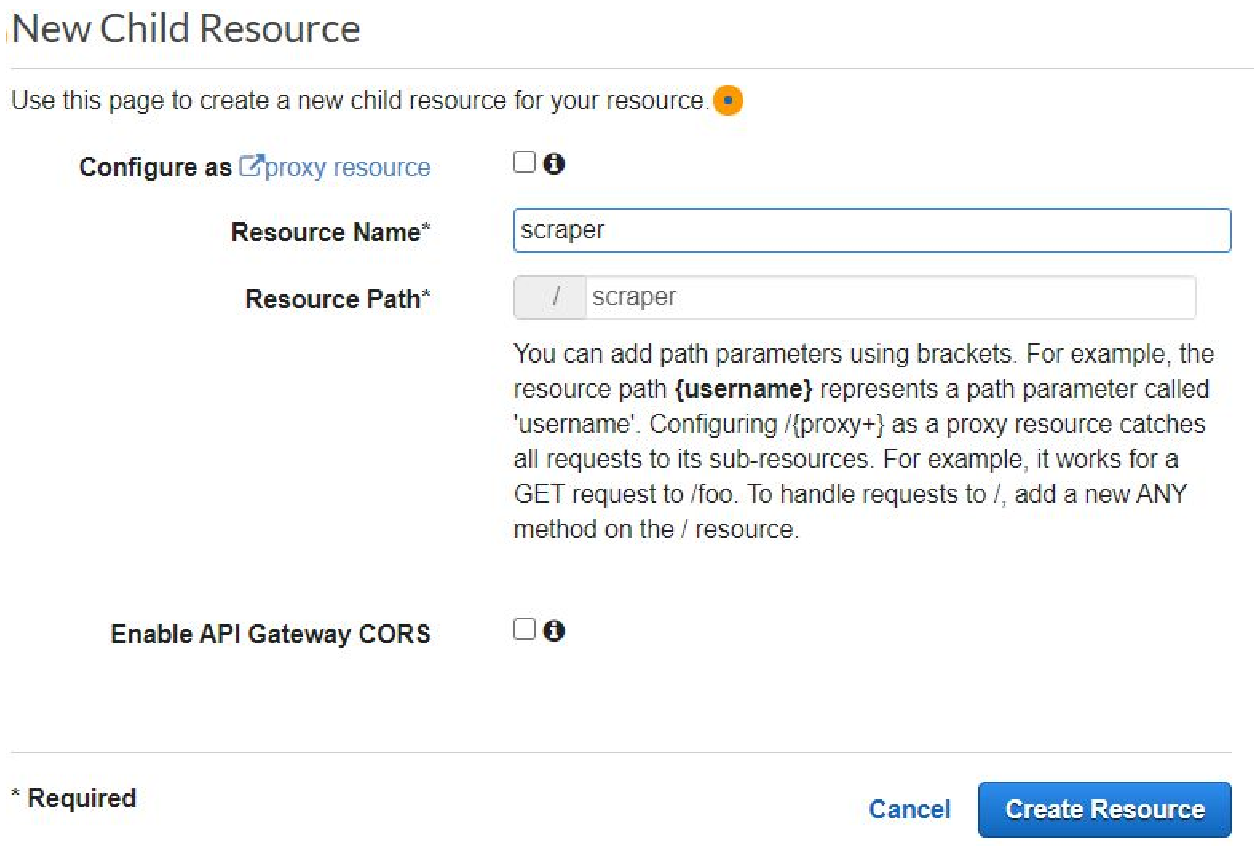
[✓ New API] Import from Swagger or Open API 3 Example API

Settings

Choose a friendly name and description for your API.

API name*	scraper
Description	
Endpoint Type	Regional

You will now be taken to a page where we can define Resources and Methods. We only need one Resource and one Method. Under the “Actions” button, select “Create Resource” and fill in the blanks as follows. You should only need to modify the field called “Resource Name”.



New Child Resource

Use this page to create a new child resource for your resource.

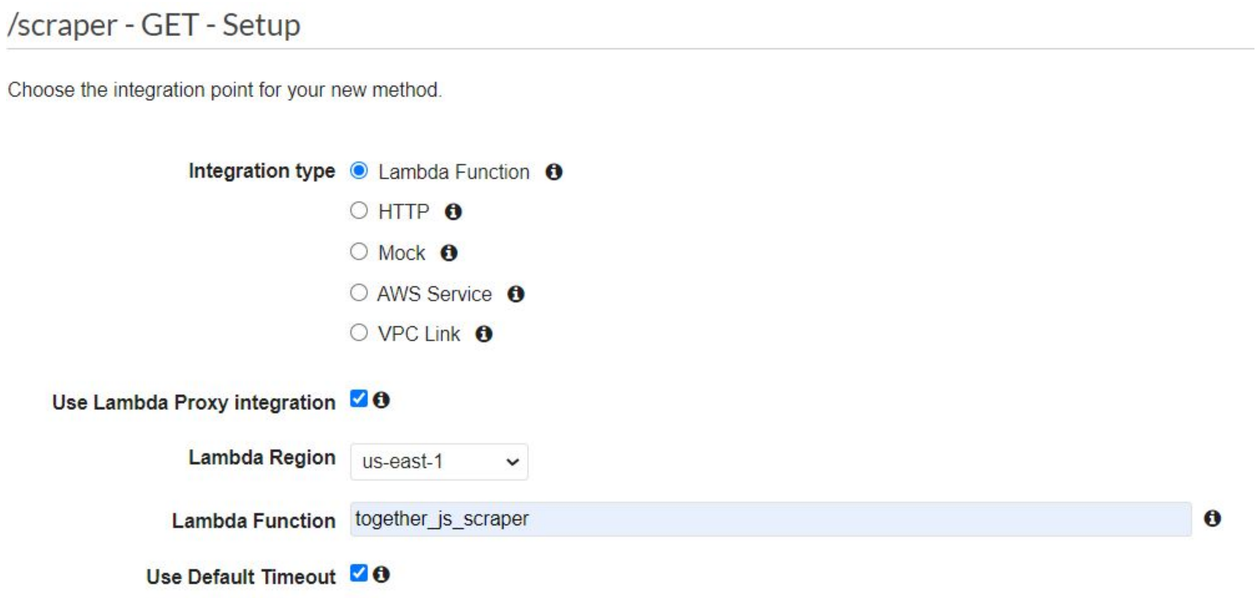
Configure as proxy resource	☐
Resource Name*	scraper
Resource Path*	/ scraper

You can add path parameters using brackets. For example, the resource path **{username}** represents a path parameter called 'username'. Configuring /{proxy+} as a proxy resource catches all requests to its sub-resources. For example, it works for a GET request to /foo. To handle requests to /, add a new ANY method on the / resource.

Enable API Gateway CORS	☐

* Required

Once the resource is created, select it and choose “Create Method” from the “Actions” button. For the method type, select “GET” and click the check box. You will be prompted to enter some settings, including the Lambda function that you want to connect to! Complete the fields as follows. The “Lambda Function” field should auto-populate based on the one you have made. Fill out the form as follows (note that the “Use Proxy integration” box should be checked) and then click “Save”.

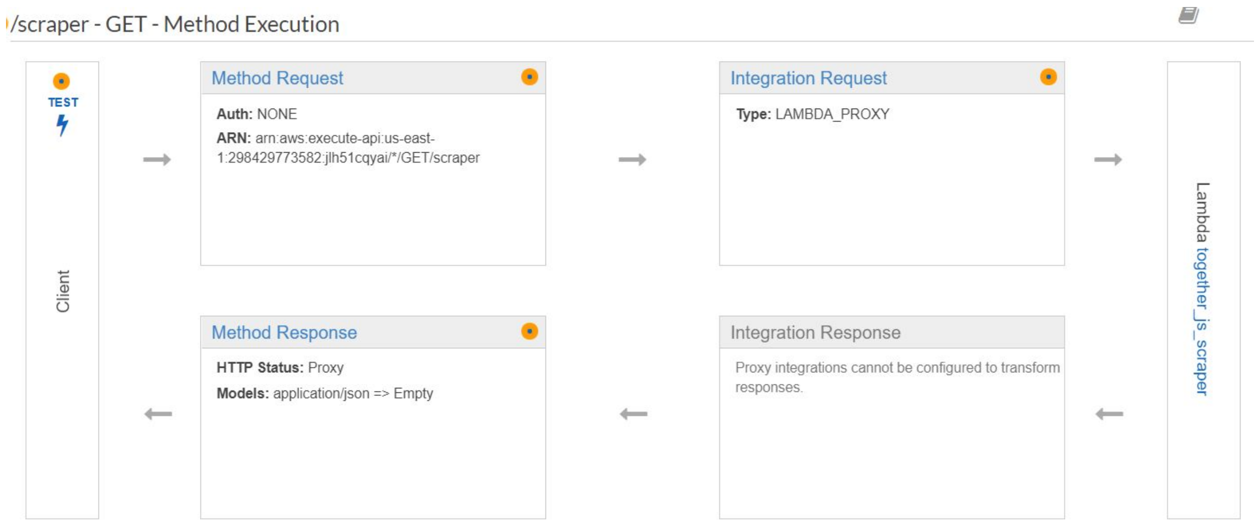


/scraper - GET - Setup

Choose the integration point for your new method.

Integration type	[✓ Lambda Function] HTTP Mock AWS Service VPC Link
Use Lambda Proxy integration	✓
Lambda Region	us-east-1
Lambda Function	together_js_scraper
Use Default Timeout	✓

You should now see a wireframe showing the data flow of the API. We need to configure the API to know what parameters to expect and how it should behave. Click on “Method Request” in the menu below.



/scraper - GET - Method Execution

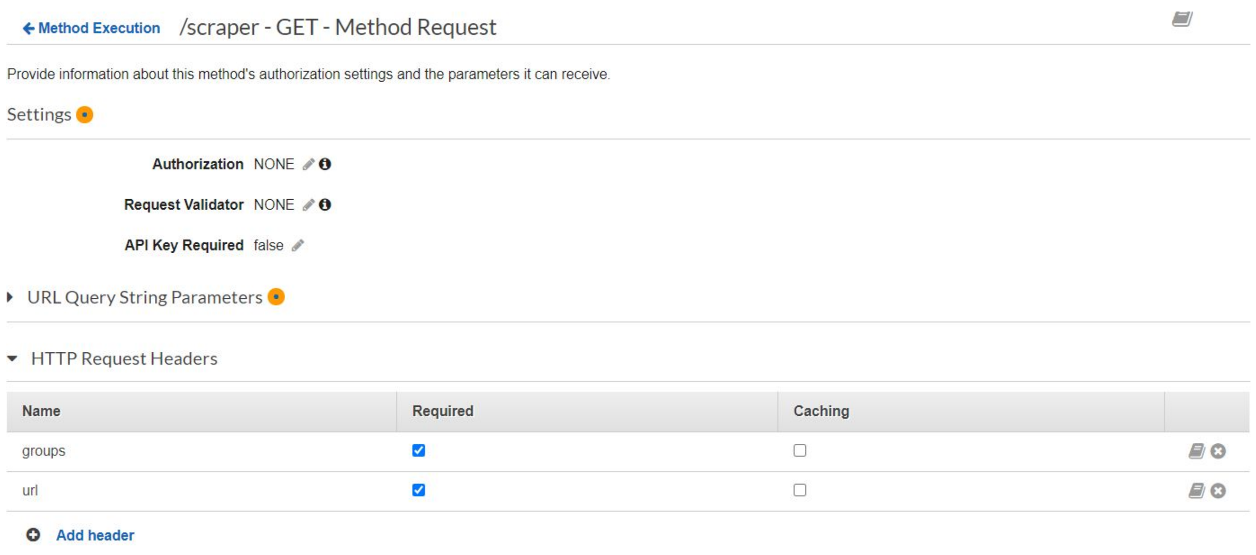
Auth: NONE
ARN: arn:aws:execute-api:us-east-1:298429773582:jlh51cqyai/*/GET/scraper

Type: LAMBDA_PROXY

HTTP Status: Proxy
Models: application/json => Empty

Proxy integrations cannot be configured to transform responses.

In the menu that follows, add a field called “url” and a field called “groups” in the section called “HTTP Request Headers”. Make both fields required.


Method Execution /scraper - GET - Method Request

Provide information about this method's authorization settings and the parameters it can receive.

Settings

Authorization NONE

Request Validator NONE

API Key Required false

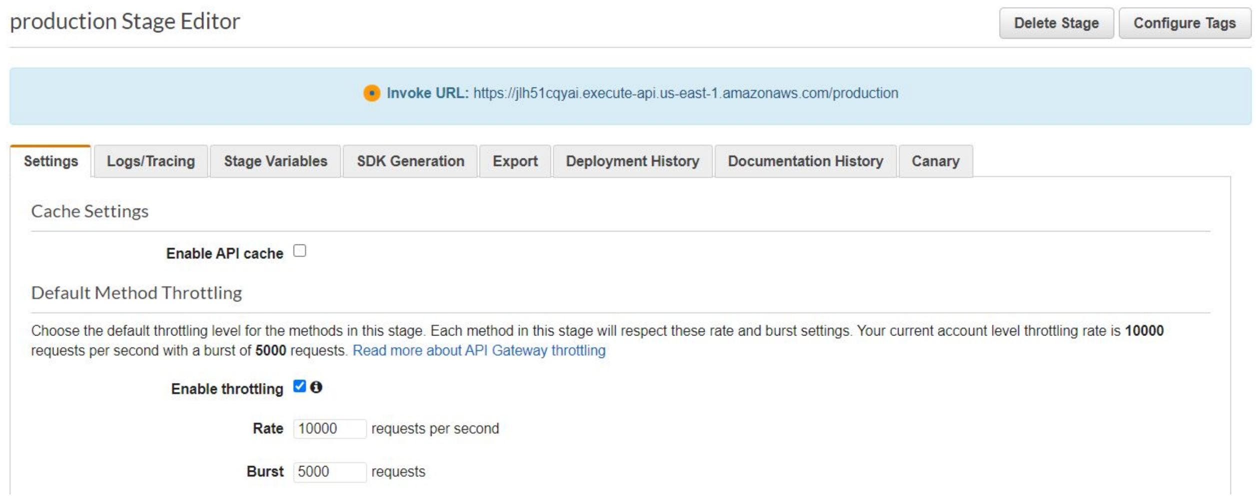
URL Query String Parameters

HTTP Request Headers

Name	Required	Caching
groups	✓	☐
url	✓	☐

Add header

The API is now ready to be deployed. In the “Actions” box, select “Deploy API”. You will be prompted to enter a “Stage” to deploy it to. By having different stages, you can publish different versions of the same API. We only really need one stage though, and it can be named anything. Something like “prod” or “production” would be appropriate. After entering a name, click on the “Deploy” button. You will be taken to a page that shows the URL where you can call your API.


production Stage Editor

Invoke URL: https://jlh51cqyai.execute-api.us-east-1.amazonaws.com/production

Cache Settings

Enable API cache ☐

Default Method Throttling

Choose the default throttling level for the methods in this stage. Each method in this stage will respect these rate and burst settings. Your current account level throttling rate is **10000** requests per second with a burst of **5000** requests. Read more about API Gateway throttling

Enable throttling ✓

Rate 10000 requests per second

Burst 5000 requests

Note that to call the API, you need to specify the resource you are trying to access. So for this case, an example function call might be:

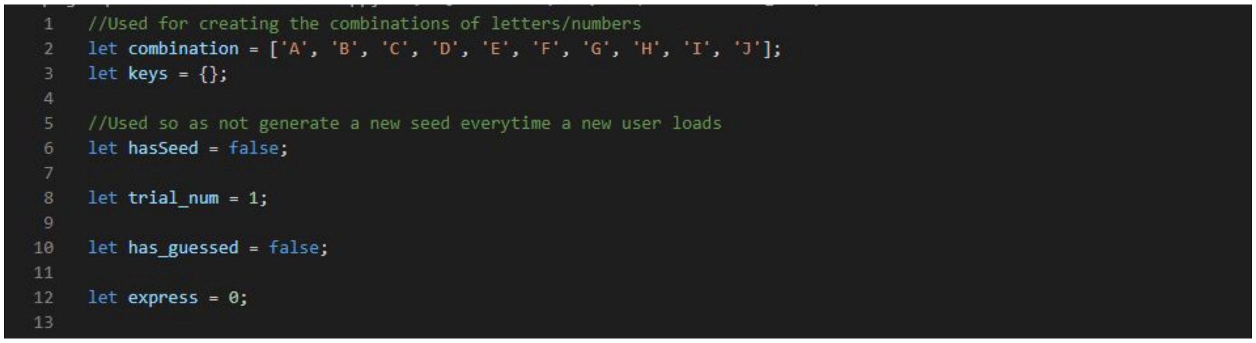
https://jlh51cqyai.execute-api.us-east-1.amazonaws.com/production/scraper?url=https://www.justinleedo.com/sample.html&groups=5

The API is now deployed and the Lambda function will execute whenever a valid HTTP request is made to the URL. You will need to locate the line of code in the front-end (in the ***launch.js*** file) where a fetch call is made to a URL. Substitute the URL you just created for the placeholder that is currently there. Now the front-end can call your Lambda function and generate URLs!

<u>LETTERS TO NUMBERS: FUNCTIONALITY</u>

Within the Github repository, a folder titled Letters-to-Numbers should be found. Within the folder, the 3 main files app.js , index.html, and styles.css are there. App.js handles the entirety of the functionality, and uses JQuery to manipulate the HTML elements as well as handle sending and receiving Together.JS code.

The Main Variables

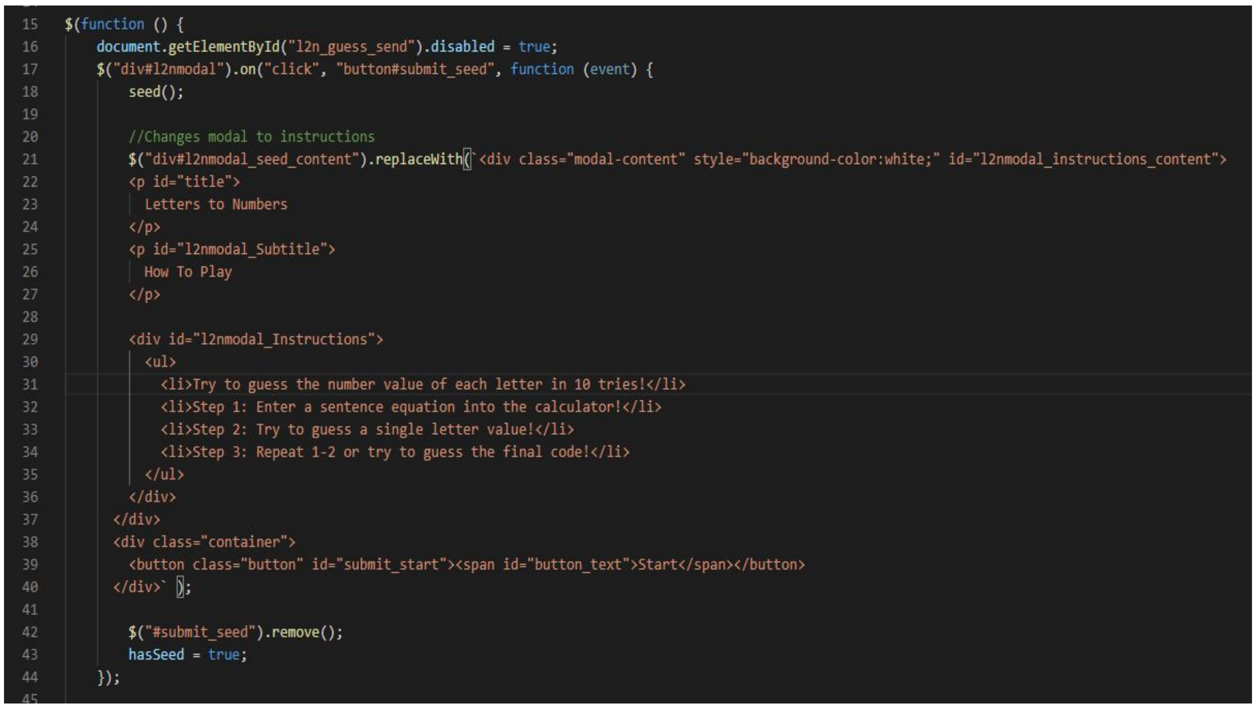


```
//Used for creating the combinations of letters/numbers
let combination = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'];
let keys = {};

//Used so as not generate a new seed everytime a new user loads
let hasSeed = false;

let trial_num = 1;

let has_guessed = false;

let express = 0;

```

- **<u>Combination:</u> An array that contains the letters A-J. The index of each letter is the corresponding digit assigned to it. Ex. If A=9, then A is in the last part of the array.**
- **<u>Keys:</u> A dictionary that will create a key of a letter from A-J and then assign it a value that corresponds to its index in Combination.**
- **<u>hasSeed:</u> Boolean variable that is used to determine whether a page has just loaded and needs the Combination/Keys from other TogetherJS pages.**
- **<u>Trial_num:</u> An integer variable that counts the current trial a user is on.**
- **<u>Has_guessed:</u> Boolean variable that keeps track of whether a player is on Step 1 or Step 2.**
- **<u>Express:</u> An integer variable that keeps track of the number of operation symbols a player has input.**

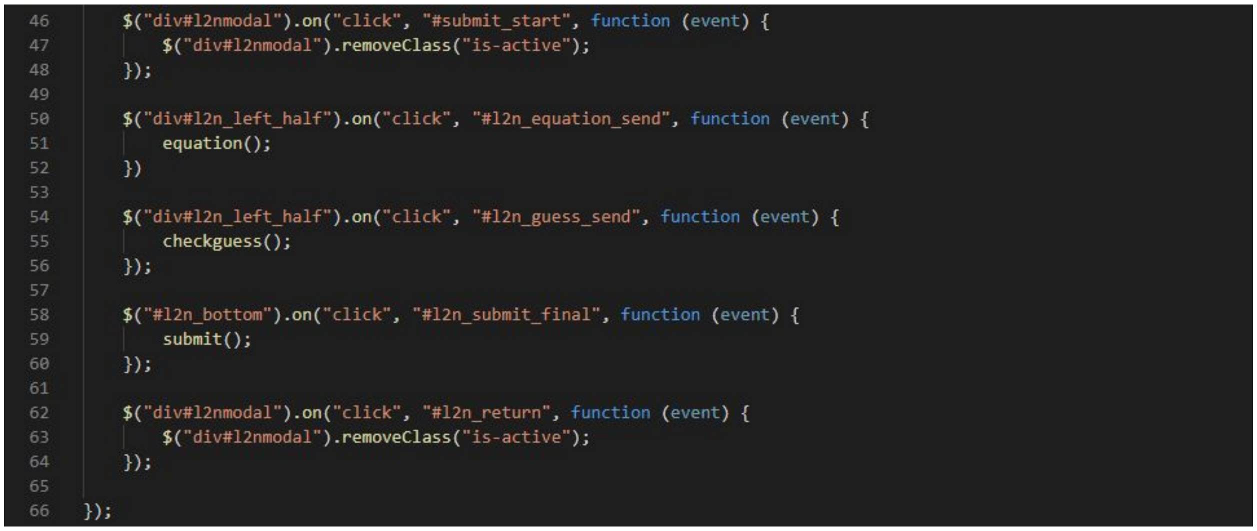
Functions and the Order they should be running

1, Main JQuery function



```
15  $(function () {
16      document.getElementById("l2n_guess_send").disabled = true;
17      $("div#l2nmodal").on("click", "button#submit_seed", function (event) {
18          seed();
19
20          //Changes modal to instructions
21          $("div#l2nmodal_seed_content").replaceWith(`<div class="modal-content" style="background-color:white;" id="l2nmodal_instructions_content">
22          <p id="title">
23            Letters to Numbers
24          </p>
25          <p id="l2nmodal_Subtitle">
26            How To Play
27          </p>
28
29          <div id="l2nmodal_Instructions">
30            <ul>
31              <li>Try to guess the number value of each letter in 10 tries!</li>
32              <li>Step 1: Enter a sentence equation into the calculator!</li>
33              <li>Step 2: Try to guess a single letter value!</li>
34              <li>Step 3: Repeat 1-2 or try to guess the final code!</li>
35            </ul>
36          </div>
37        </div>
38        <div class="container">
39          <button class="button" id="submit_start"><span id="button_text">Start</span></button>
40        </div>` );
41
42          $("#submit_seed").remove();
43          hasSeed = true;
44      });
```

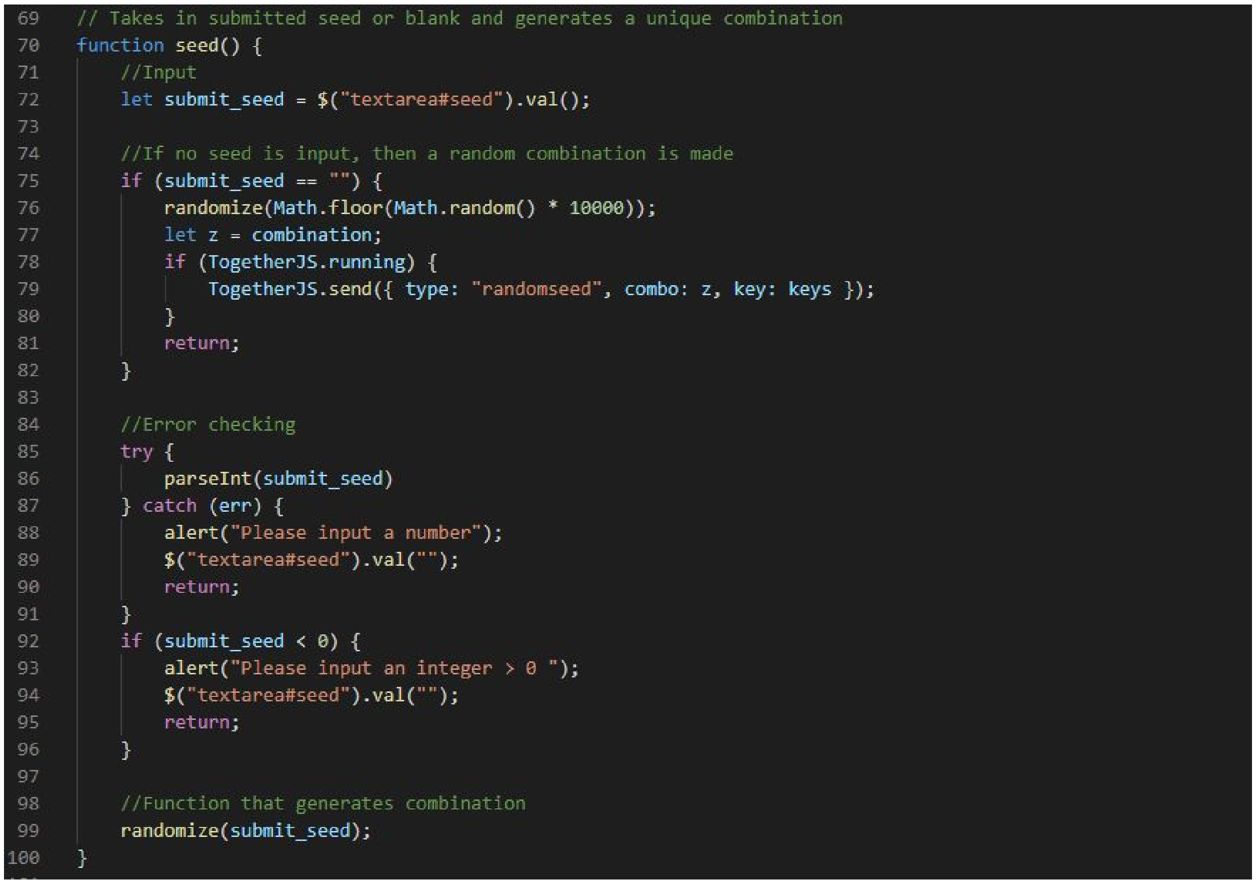
- **<u>What the screen should look like by Line 16</u>: A modal with a text area to input a seed, a short description of what to do, and a button to submit.**
- **<u>Line 16</u>: Disables the guess button so that users can not skip to step 2 automatically.**
- **<u>Line 17-40</u>: Once the submit button is clicked, the seed generation function is run. The modal's content will then be replaced with instructions and a button to start. Line 21-40 are *NOT* generating a new modal, they are simply swapping the content.**



```
46      $("div#l2nmodal").on("click", "#submit_start", function (event) {
47          $("div#l2nmodal").removeClass("is-active");
48      });
49
50      $("div#l2n_left_half").on("click", "#l2n_equation_send", function (event) {
51          equation();
52      })
53
54      $("div#l2n_left_half").on("click", "#l2n_guess_send", function (event) {
55          checkguess();
56      });
57
58      $("#l2n_bottom").on("click", "#l2n_submit_final", function (event) {
59          submit();
60      });
61
62      $("div#l2nmodal").on("click", "#l2n_return", function (event) {
63          $("div#l2nmodal").removeClass("is-active");
64      });
65
66  });
```

- **Line 46-48: Deactivates the modal, showing the “main” letters-to-numbers page.**
- **Line 50-52: Attaches an event listener to the send button, calling the equation function.**
- **Lines 54-57: Attaches an event listener to the guess button, calling the checkguess function that checks if the guess is correct or not.**
- **Lines 58-60: Attaches an event listener to the submit button, calling the submit function that checks if the final answer is correct or not.**
- **Lines 62-64: Attaches an event listener to the return button in the modal that is activated when a user submits the wrong final answer. Pressing the return button should close the modal again.**

3,Seed

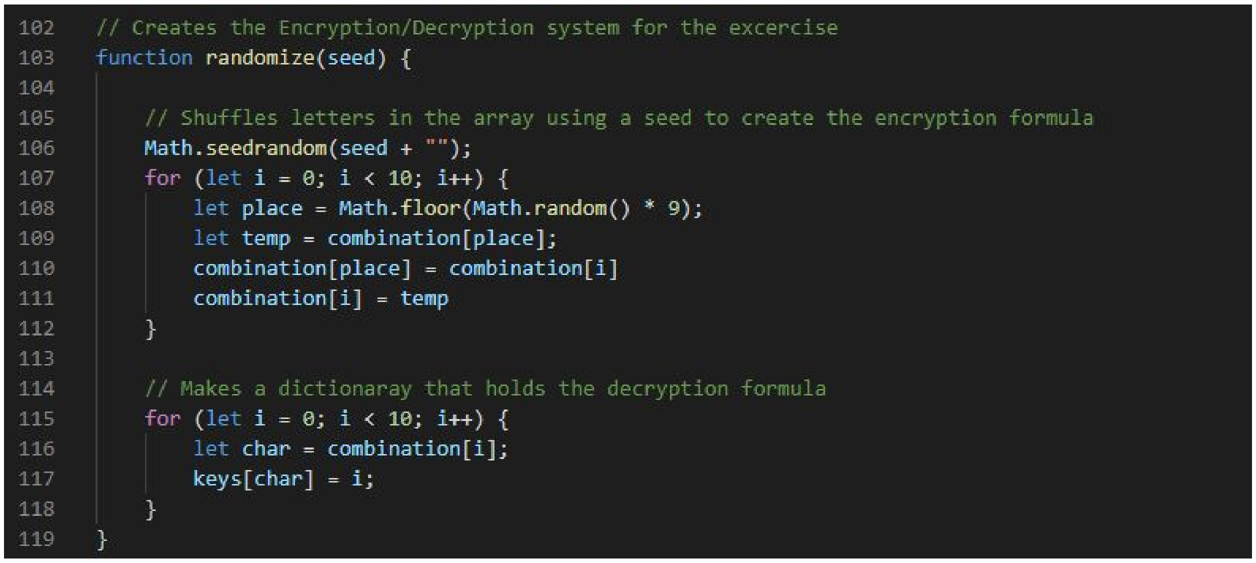


```
 69  // Takes in submitted seed or blank and generates a unique combination
 70  function seed() {
 71      //Input
 72      let submit_seed = $("textarea#seed").val();
 73
 74      //If no seed is input, then a random combination is made
 75      if (submit_seed == "") {
 76          randomize(Math.floor(Math.random() * 10000));
 77          let z = combination;
 78          if (TogetherJS.running) {
 79              TogetherJS.send({ type: "randomseed", combo: z, key: keys });
 80          }
 81          return;
 82      }
 83
 84      //Error checking
 85      try {
 86          parseInt(submit_seed)
 87      } catch (err) {
 88          alert("Please input a number");
 89          $("textarea#seed").val("");
 90          return;
 91      }
 92      if (submit_seed < 0) {
 93          alert("Please input an integer > 0 ");
 94          $("textarea#seed").val("");
 95          return;
 96      }
 97
 98      //Function that generates combination
 99      randomize(submit_seed);
100  }
```

- **Input: None**
- **Output: None**
- **Functionality:**
    - **The function when activated will take the value from the seed text area and assign it to submit_seed**
    - **Lines 75-81 are for handling if no seed has been entered.**
    - **If that's the case, a random number will be generated and passed to the randomize function.**
    - **Since there are multiple browsers running the seed function at the same time, different combinations/keys are made. In order to ensure all browsers have the same combination lines 78-80 use the TogetherJs.send function to send to every browser the current keys/combination that are generated. The other browsers will then receive the information and update to the appropriate combinations and keys. More documentation on TogetherJs.send and TogetherJs.hub.on functions below.**
    - **Lines 85-91 are to insure the values that were inputted are actually numbers. If not, the text area will be cleared and an alert given. This section may not be necessary since the imported Math.seedrandom() function can also accept strings.**

- Lines 92-96 are there to ensure no negative numbers have been inputted. Both lines 85-91 and 92-96 are left over code from when a different combination system was in place. As they don't impair the program and only limit the users seed input, they have been left in.
- Line 99 calls the actual function that handles creating the combination/keys sequence.

4.Randomize



```
102  // Creates the Encryption/Decryption system for the excercise
103  function randomize(seed) {
104
105      // Shuffles letters in the array using a seed to create the encryption formula
106      Math.seedrandom(seed + "");
107      for (let i = 0; i < 10; i++) {
108          let place = Math.floor(Math.random() * 9);
109          let temp = combination[place];
110          combination[place] = combination[i]
111          combination[i] = temp
112      }
113
114      // Makes a dictionaray that holds the decryption formula
115      for (let i = 0; i < 10; i++) {
116          let char = combination[i];
117          keys[char] = i;
118      }
119  }
```

- **Input:** The number that has been put in the seed text area.
- **Output:** None
- **Functionality:**
    - Line 106 uses Math.seedrandom(), a function that has been imported from an outside module. The line of code that imports this module is in index.html in line 16.
    - Math.seedrandom() will set Math.random() to generate a predetermined order of outputs depending on what seed was given. For instance, if the following code is run,

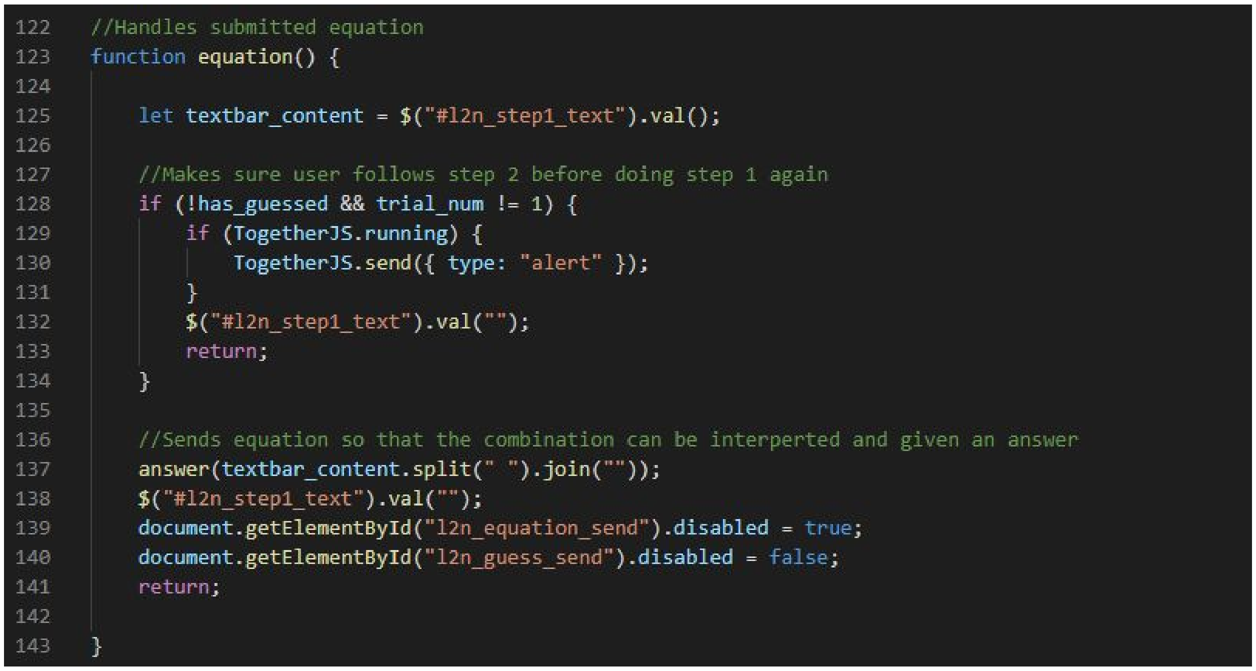
```
Math.seedrandom('hello.');
console.log(Math.random());
console.log(Math.random());
```

The output will always be 0.9282578795792454 and 0.3752569768646784 no matter how many times you run the code.

- Additional documentation is provided at this link as well as at this github. The input has to be a string and so the seed is concatenated with "".
- Lines 107-112 randomize the order of the letters in combination. A random index is generated at line 108 and then floored so as not to create an out of bounds index. Lines 109-111 then switch the letters at the current index and the index generated randomly. The for loop repeats till all 10 letters have been rearranged.

    - **Lines 115-118 set the values in keys by assigning each letter a respective number. Both Keys and Combination will be accessed in the translate and retranslate functions.**

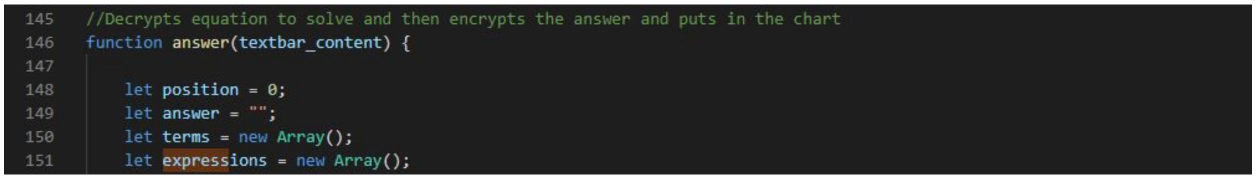
<u>5.Equation</u>



```
122  //Handles submitted equation
123  function equation() {
124
125      let textbar_content = $("#l2n_step1_text").val();
126
127      //Makes sure user follows step 2 before doing step 1 again
128      if (!has_guessed && trial_num != 1) {
129          if (TogetherJS.running) {
130              TogetherJS.send({ type: "alert" });
131          }
132          $("#l2n_step1_text").val("");
133          return;
134      }
135
136      //Sends equation so that the combination can be interperted and given an answer
137      answer(textbar_content.split(" ").join(""));
138      $("#l2n_step1_text").val("");
139      document.getElementById("l2n_equation_send").disabled = true;
140      document.getElementById("l2n_guess_send").disabled = false;
141      return;
142
143  }
```

- **<u>Input:</u> None**
- **<u>Output:</u> None**
- **<u>Functionality:</u>**
    - **Line 125 assigns textbar_content the equation that was typed into the text area under step 1.**
    - **Line 128-Line 135 is leftover code from when buttons were not disabled, and alerted users when they tried to guess before completing step 1.**
    - **Line 137 passes textbar_content to the answer function to be parsed and interperted.**
    - **Line 139-140 disable the send button and enable the guess button, allowing the user to only guess and not submit another equation.**

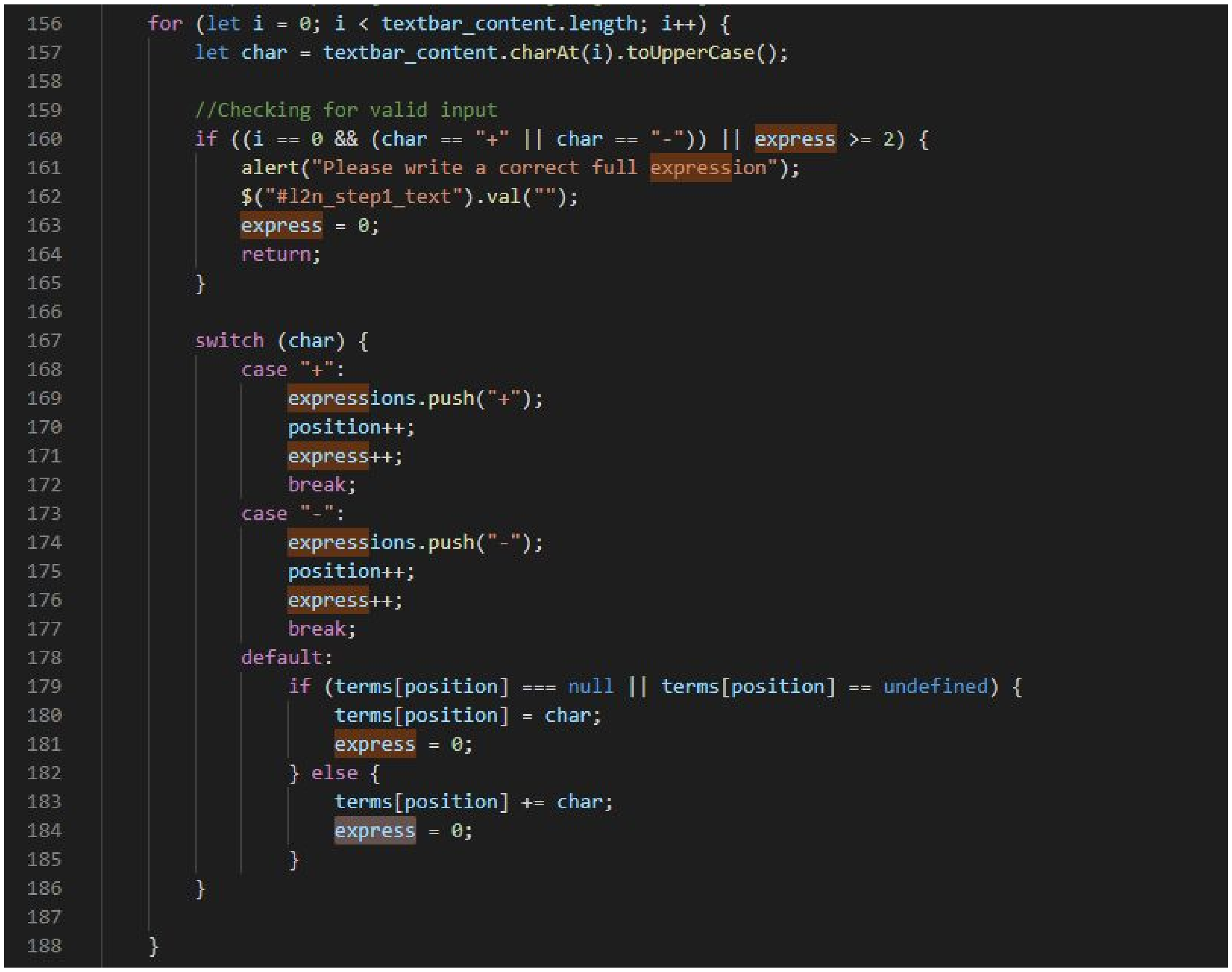
<u>6.Answer</u>



```
//Decrypts equation to solve and then encrypts the answer and puts in the chart
function answer(textbar_content) {

    let position = 0;
    let answer = "";
    let terms = new Array();
    let expressions = new Array();
```

- **<u>Position:</u>An integer variable that keeps track of the amount of numbers being added or subtracted.**
- **<u>Answer:</u> A string variable that holds the final result of the equation**
- **<u>Terms:</u> A string array that holds each individual variable that is being added or subtracted.**

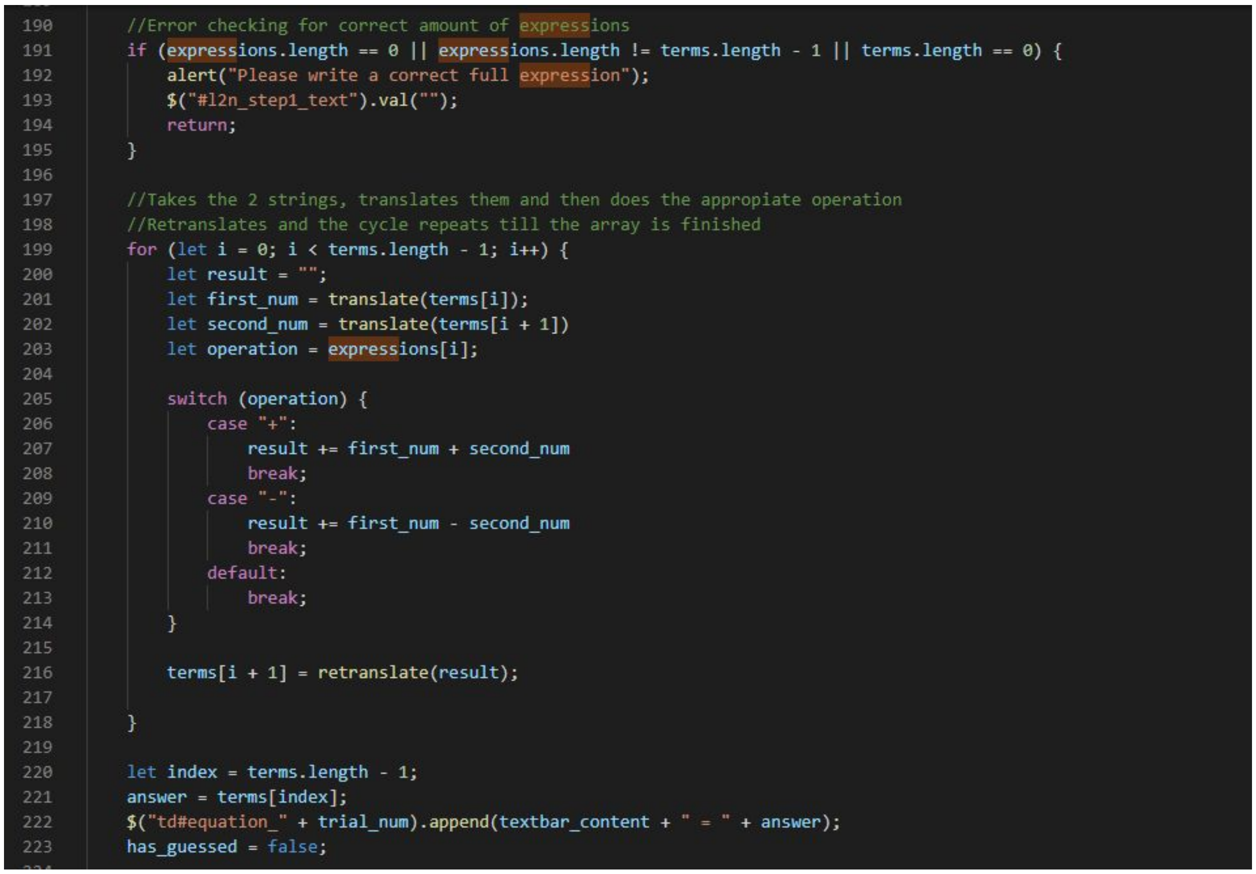
- **Expressions:** A string array that holds each operation character



```
156     for (let i = 0; i < textbar_content.length; i++) {
157         let char = textbar_content.charAt(i).toUpperCase();
158 
159         //Checking for valid input
160         if ((i == 0 && (char == "+" || char == "-")) || express >= 2) {
161             alert("Please write a correct full expression");
162             $("#l2n_step1_text").val("");
163             express = 0;
164             return;
165         }
166 
167         switch (char) {
168             case "+":
169                 expressions.push("+");
170                 position++;
171                 express++;
172                 break;
173             case "-":
174                 expressions.push("-");
175                 position++;
176                 express++;
177                 break;
178             default:
179                 if (terms[position] === null || terms[position] == undefined) {
180                     terms[position] = char;
181                     express = 0;
182                 } else {
183                     terms[position] += char;
184                     express = 0;
185                 }
186         }
187 
188     }
```

- **Functionality:**
  - **Line 157: Takes the character at the i-th index of the submitted string and stores it in char.**
  - **Line 160-165: This is an error check to detect if the first character is an operation.**
  - **This is done because at Lines 191-195, the if statement checks if there are more operator symbols than things being added/subtracted but not if the position of said operator symbols does not make sense (i.e at the very beginning or end of a string).**
  - **This If statement will also check to make sure that there aren't equations that have 2 operator symbols in a row. For example, if a user inputted (a++b) or (a+b—c), these functions will get flagged down and the answer function will end.**
  - **Line 167-186: Through every iteration of the for loop, the current value of char will be evaluated and then either assigned to express or terms.**

    - If the character is an operation, it will be pushed to the expression array and express and position variables will be incremented.
    - If the character is a letter, then if the position in the terms array is empty, it will be set to the current letter value. The position value is not incremented because since the for loop is going through the inputted string one letter at a time, there could be a situation where the sentence is (a+ab+b) and since a letter represents a number, it is necessary that terms[1] have "ab" and not just "a". As such, Lines 182-185 handle that scenario.

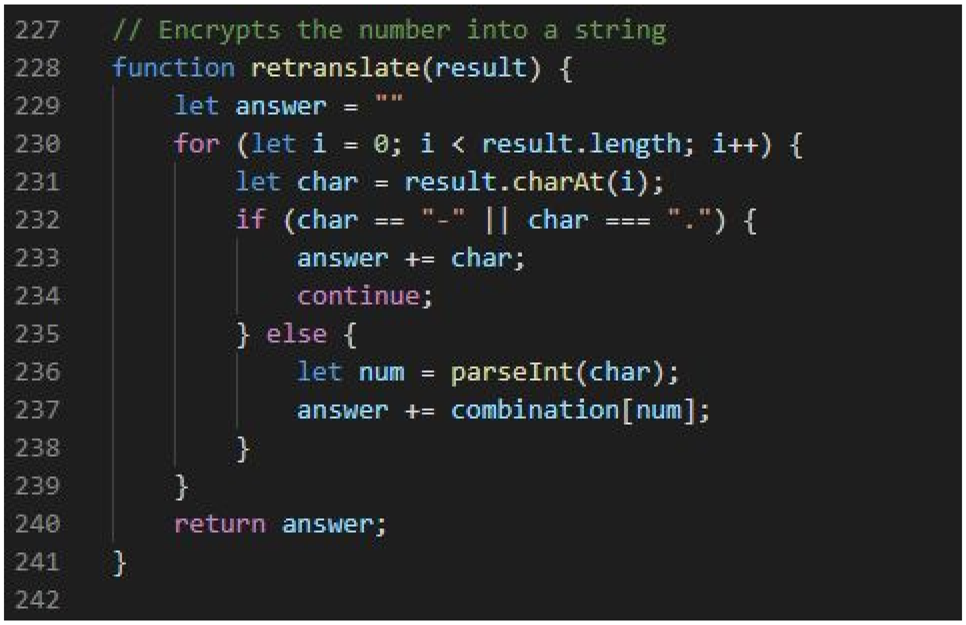


```
190     //Error checking for correct amount of expressions
191     if (expressions.length == 0 || expressions.length != terms.length - 1 || terms.length == 0) {
192         alert("Please write a correct full expression");
193         $("#l2n_step1_text").val("");
194         return;
195     }
196 
197     //Takes the 2 strings, translates them and then does the appropiate operation
198     //Retranslates and the cycle repeats till the array is finished
199     for (let i = 0; i < terms.length - 1; i++) {
200         let result = "";
201         let first_num = translate(terms[i]);
202         let second_num = translate(terms[i + 1])
203         let operation = expressions[i];
204 
205         switch (operation) {
206             case "+":
207                 result += first_num + second_num
208                 break;
209             case "-":
210                 result += first_num - second_num
211                 break;
212             default:
213                 break;
214         }
215 
216         terms[i + 1] = retranslate(result);
217 
218     }
219 
220     let index = terms.length - 1;
221     answer = terms[index];
222     $("td#equation_" + trial_num).append(textbar_content + " = " + answer);
223     has_guessed = false;
```

- Lines 191-195: Error checking to make sure the inputted expressions are not strings like (aab) or (aa+b+c++) or (++---).
- Lines 199-218: This is where the actual terms are translated from letters to numbers. For instance if a=1 and a term is "aa" it will be translated to 11.
- The for loop will take 2 terms, translate them and then commit the proper operation. The result will then be taken, retranslated from its number value to its letter equivalent and put back into the array.
- This process will be repeated until all terms have been added or subtracted.

- **Lines 220-223: The answer will be stored in the last value of terms, and the result will be added into the table under the respective trial.**
- **Has_guessed will be set to false in order to stop users from submitting another equation without having completed step 2.**

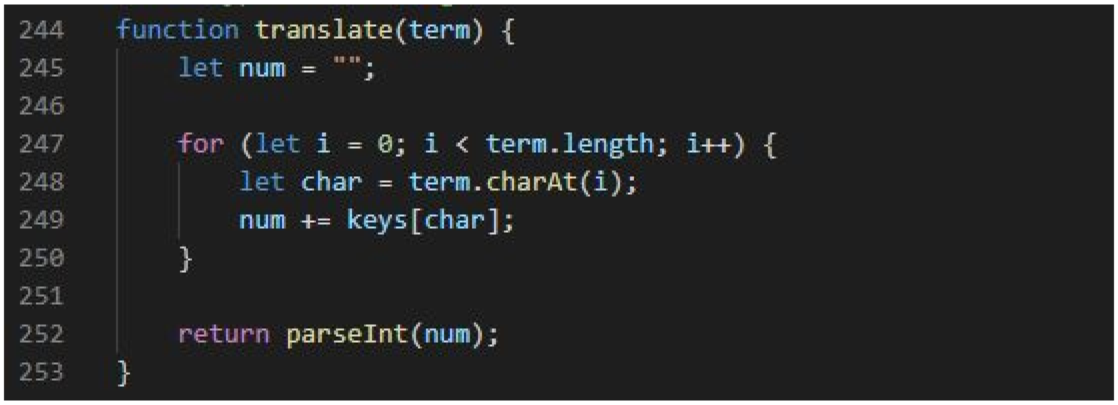
7.Retranslate



```
227  // Encrypts the number into a string
228  function retranslate(result) {
229      let answer = ""
230      for (let i = 0; i < result.length; i++) {
231          let char = result.charAt(i);
232          if (char == "-" || char === ".") {
233              answer += char;
234              continue;
235          } else {
236              let num = parseInt(char);
237              answer += combination[num];
238          }
239      }
240      return answer;
241  }
242
```

- **Input: A number string**
- **Output: A letter string**
- **Functionality:**
    - **Lines 232-235 account for a term being possibly negative or a decimal. The decimal section was left over from when the exercise had multiplication and division functionality.**
    - **Lines 236-237 translate the character into a number and then “translates” by accessing the letter stored in the respective index of combination.**
    - **This cycle is repeated until all numbers have been translated and added to the answer string.**

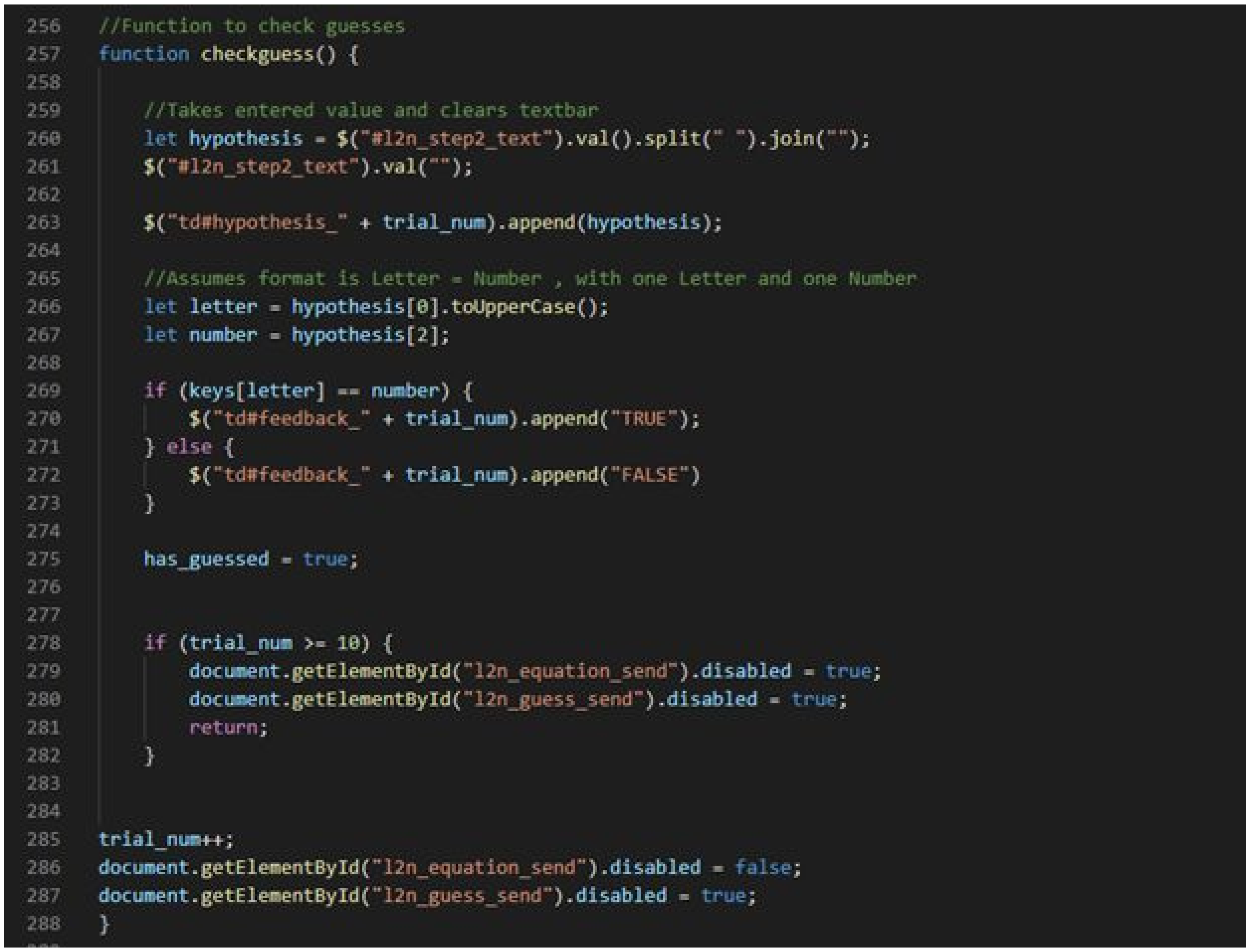
8.Translate



```
function translate(term) {
    let num = "";

    for (let i = 0; i < term.length; i++) {
        let char = term.charAt(i);
        num += keys[char];
    }

    return parseInt(num);
}
```

- **<u>Input:</u> A letter string**
- **<u>Output:</u> A number**
- **<u>Functionality:</u>**
    - **Similarly to retranslate, a for loop goes through each term of the string, but instead of accessing combination, it pulls the respective value associated with key at char.**
    - **All of the number values will be stored into a string Num and then parsed into a number and returned.**

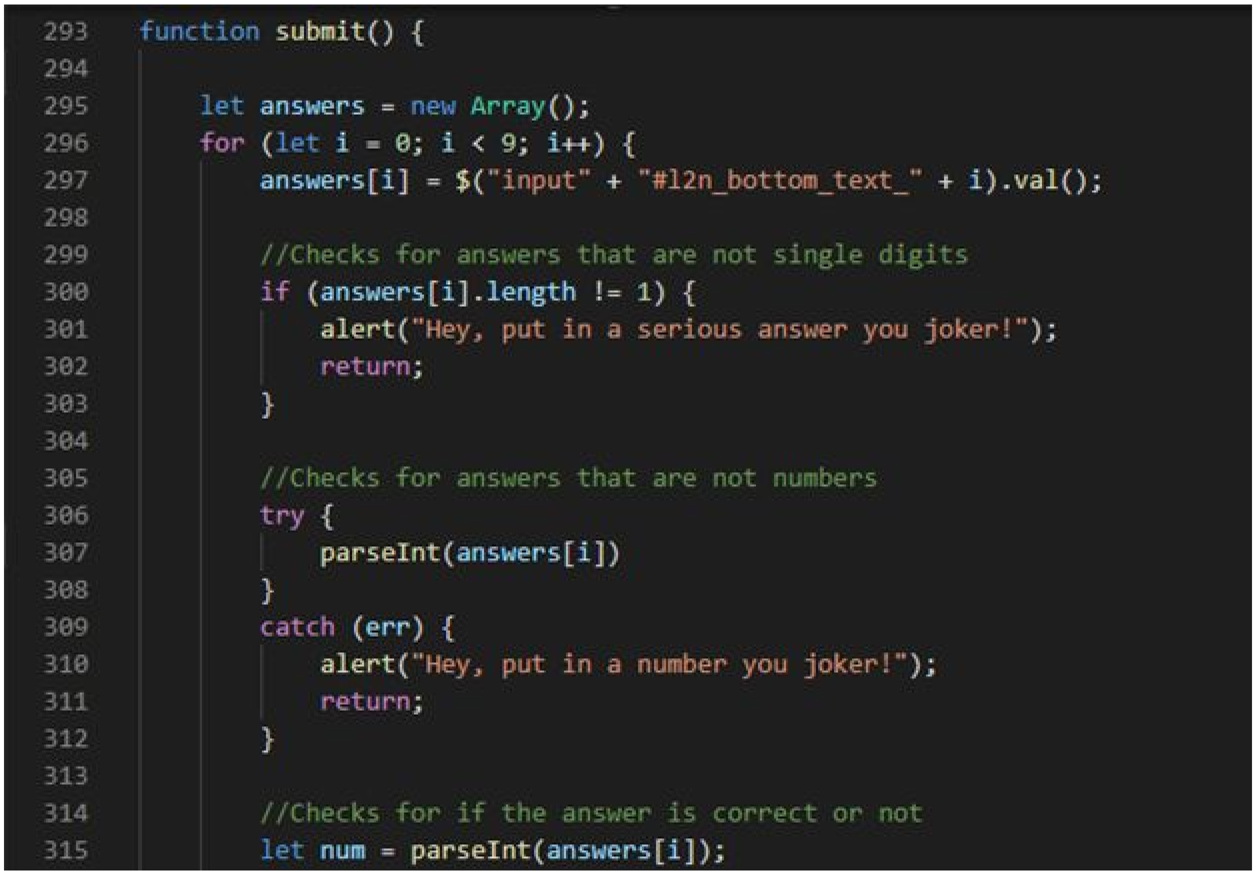
<u>9.Checkguess</u>



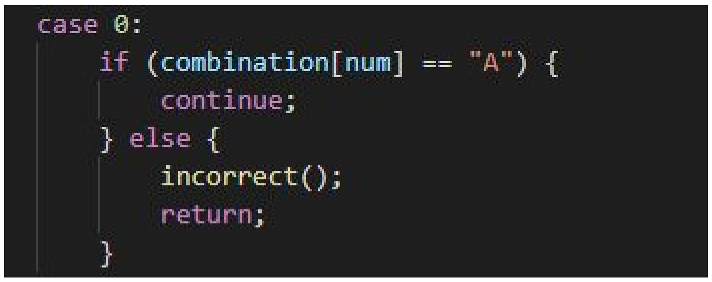
```
256 //Function to check guesses
257 function checkguess() {
258 
259     //Takes entered value and clears textbar
260     let hypothesis = $("#l2n_step2_text").val().split(" ").join("");
261     $("#l2n_step2_text").val("");
262 
263     $("td#hypothesis_" + trial_num).append(hypothesis);
264 
265     //Assumes format is Letter = Number , with one Letter and one Number
266     let letter = hypothesis[0].toUpperCase();
267     let number = hypothesis[2];
268 
269     if (keys[letter] == number) {
270         $("td#feedback_" + trial_num).append("TRUE");
271     } else {
272         $("td#feedback_" + trial_num).append("FALSE")
273     }
274 
275     has_guessed = true;
276 
277 
278     if (trial_num >= 10) {
279         document.getElementById("l2n_equation_send").disabled = true;
280         document.getElementById("l2n_guess_send").disabled = true;
281         return;
282     }
283 
284 
285 trial_num++;
286 document.getElementById("l2n_equation_send").disabled = false;
287 document.getElementById("l2n_guess_send").disabled = true;
288 }
```

- **Input:** None
- **Output:** None
- **Functionality:**
    - **The hypothesis variable stores the guess that was entered in step 2. Line 260 accounts for any spaces that may have been entered and removes them from the string.**
    - **Line 263 displays the hypothesis the student has entered.**
    - **Line 266 has the letter be capitalized because all of the values in keys are capitalized and as such that is the only way for Line 269 to work.**
    - **Line 269-273 checks if the inputted answer is correct by accessing the value associated with the letter in keys.**
    - **Lines 278-282 disable the send and guess buttons after the 10th guess has been made, forcing the user to submit the final answer.**
    - **Lines 285-288 increment trial num and since trial_num < 10 , the send button is enabled and the guess button disabled.**

10.Submit



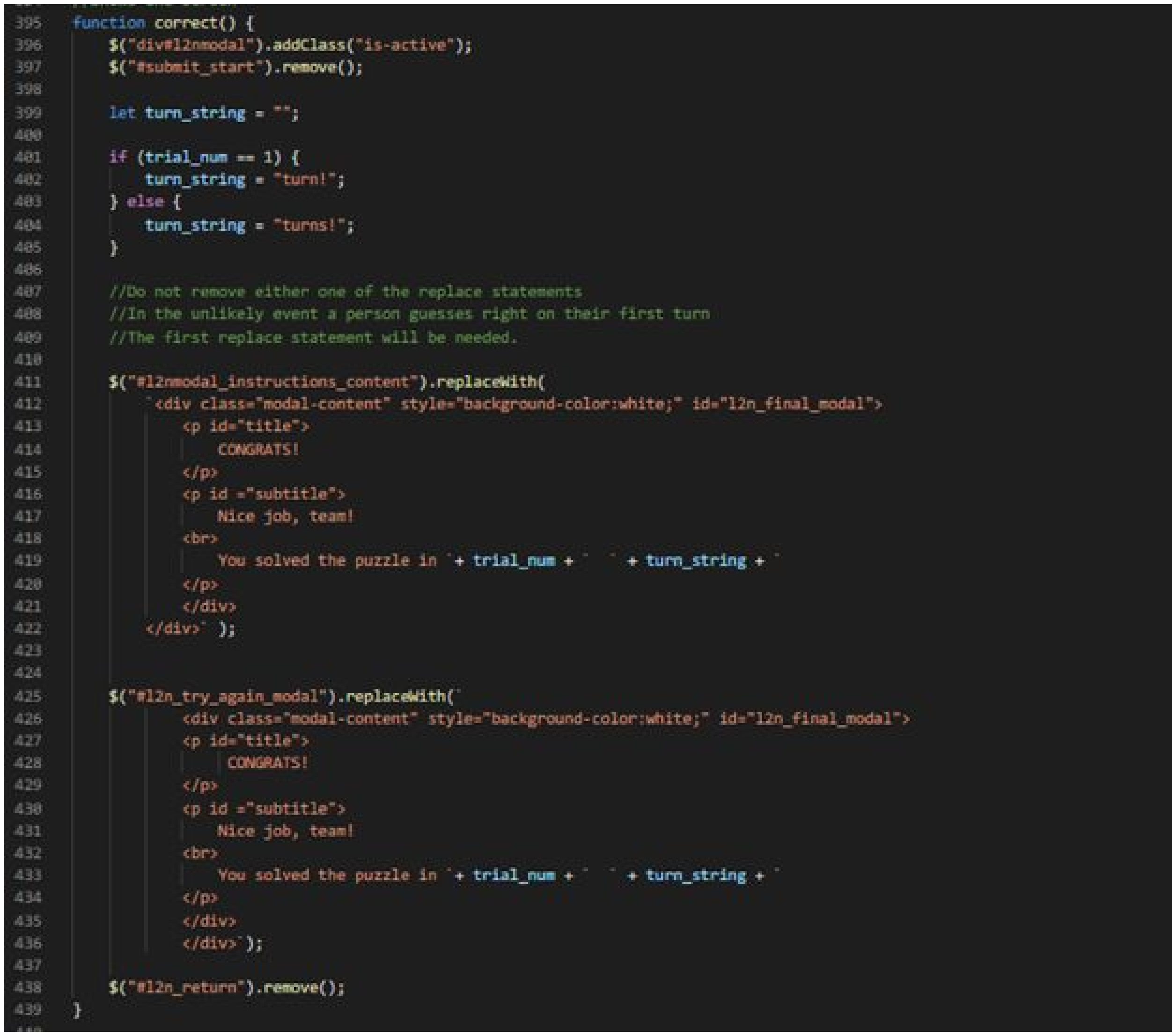
```
293  function submit() {
294
295      let answers = new Array();
296      for (let i = 0; i < 9; i++) {
297          answers[i] = $("input" + "#l2n_bottom_text_" + i).val();
298
299          //Checks for answers that are not single digits
300          if (answers[i].length != 1) {
301              alert("Hey, put in a serious answer you joker!");
302              return;
303          }
304
305          //Checks for answers that are not numbers
306          try {
307              parseInt(answers[i])
308          }
309          catch (err) {
310              alert("Hey, put in a number you joker!");
311              return;
312          }
313
314          //Checks for if the answer is correct or not
315          let num = parseInt(answers[i]);
```

- **<u>Input:</u> None**
- **<u>Output:</u> None**
- **<u>Functionality:</u>**
    - **Lines 296-390 are a for loop, and every time the loop runs, 3 checks are done.**
    - **First is to check if an appropriate answer has been put in place, and that's it not empty or absurdly long.**
    - **Second is to check if the actual input is a number and not a letter.**
    - **Lastly, a check is done to see if the input is the correct number for the respective letter.**
    - **Since these last checks are of a similar format, Lines 315-Lines 289 are not completely screenshotted here.**
    - 

```
case 0:
    if (combination[num] == "A") {
        continue;
    } else {
        incorrect();
        return;
    }
```

- ○ **Case [number] is not checking if the input is equal to that number, it is checking to know what textarea box the for loop is currently on. The textareas are numbered 0-9 , corresponding to the alphabetical order of A-J.**
- ○ **The if statement then checks if the number entered actually does correspond to the letter at said text area by accessing a letter in the combination array whose index equals to the input. If correct, the for loop will continue. If incorrect, the incorrect function is called.**
- ○ **If the for loop has completed it's run, then all the input answers were correct and the correct() function will be called.**

11.Correct

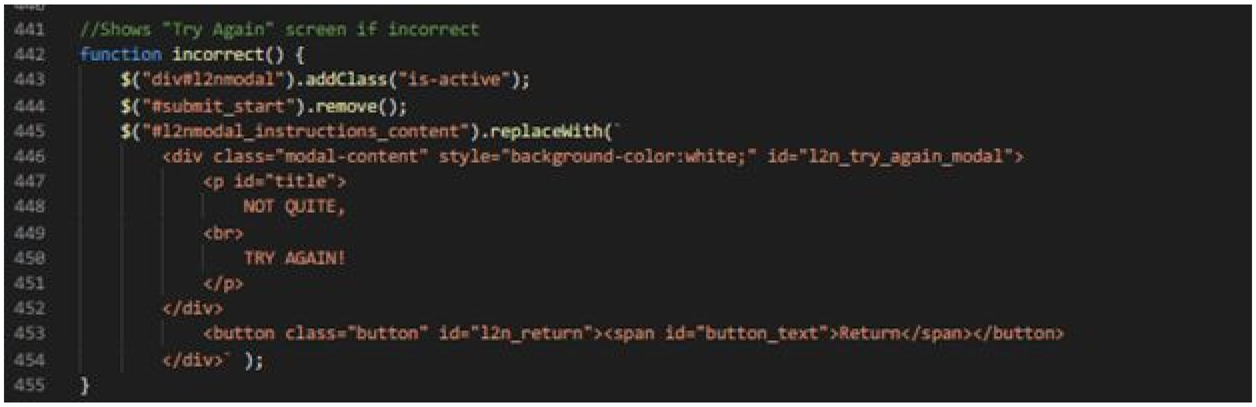


```javascript
function correct() {
    $("div#l2nmodal").addClass("is-active");
    $("#submit_start").remove();

    let turn_string = "";

    if (trial_num == 1) {
        turn_string = "turn!";
    } else {
        turn_string = "turns!";
    }

    //Do not remove either one of the replace statements
    //In the unlikely event a person guesses right on their first turn
    //The first replace statement will be needed.

    $("#l2nmodal_instructions_content").replaceWith(
        `<div class="modal-content" style="background-color:white;" id="l2n_final_modal">
            <p id="title">
                CONGRATS!
            </p>
            <p id ="subtitle">
                Nice job, team!
            <br>
                You solved the puzzle in `+ trial_num + `  ` + turn_string + `
            </p>
            </div>
        </div>` );


    $("#l2n_try_again_modal").replaceWith(`
            <div class="modal-content" style="background-color:white;" id="l2n_final_modal">
            <p id="title">
                 CONGRATS!
            </p>
            <p id ="subtitle">
                Nice job, team!
            <br>
                You solved the puzzle in `+ trial_num + `  ` + turn_string + `
            </p>
            </div>
            </div>`);

    $("#l2n_return").remove();
}
```

- **<u>Input:</u> None**

- **Output:** None
- **Functionality:**
    - It will activate a modal to display how many turns a team has taken and that they've completed the exercise.
    - It does not create a new modal, all it does is simply replaces the text from the instruction modal with new text and activates.
    - This was done because while new modals can be activated and deactivated, synchronizing these states throughout Together.JS proved to be unsuccessful.

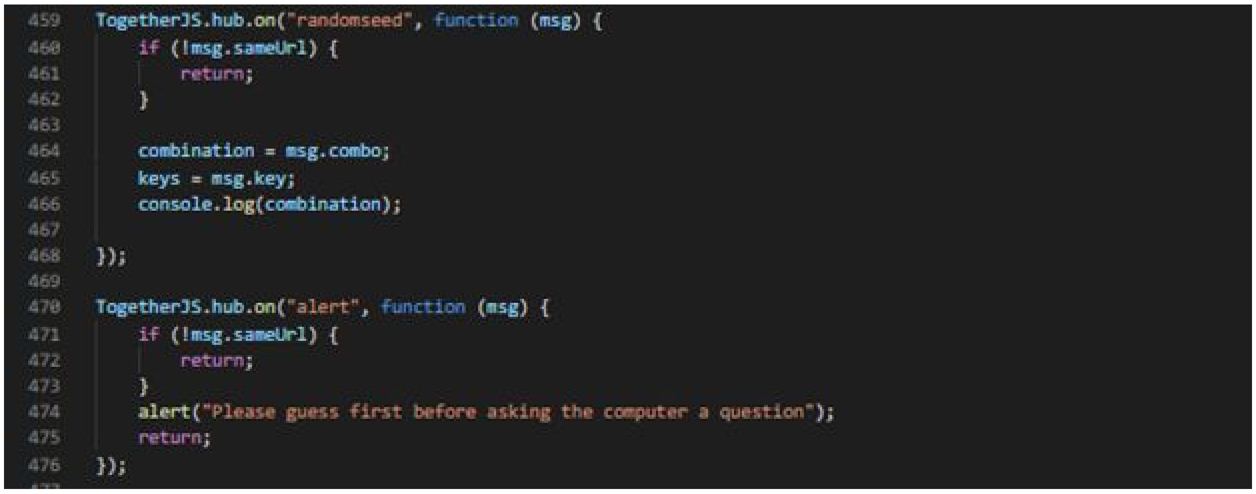
12.Incorrect



```
//Shows "Try Again" screen if incorrect
function incorrect() {
    $("div#l2nmodal").addClass("is-active");
    $("#submit_start").remove();
    $("#l2nmodal_instructions_content").replaceWith(`
        <div class="modal-content" style="background-color:white;" id="l2n_try_again_modal">
            <p id="title">
                NOT QUITE,
            <br>
                TRY AGAIN!
            </p>
        </div>
            <button class="button" id="l2n_return"><span id="button_text">Return</span></button>
        </div>` );
}
```

- **Input**: None
- **Output**: None
- **Functionality**:
    - Activates a modal that lets the user know that their final answer is incorrect.
    - It does not create a new modal, all it does is simply replaces the text from the instruction modal with new text and activates.

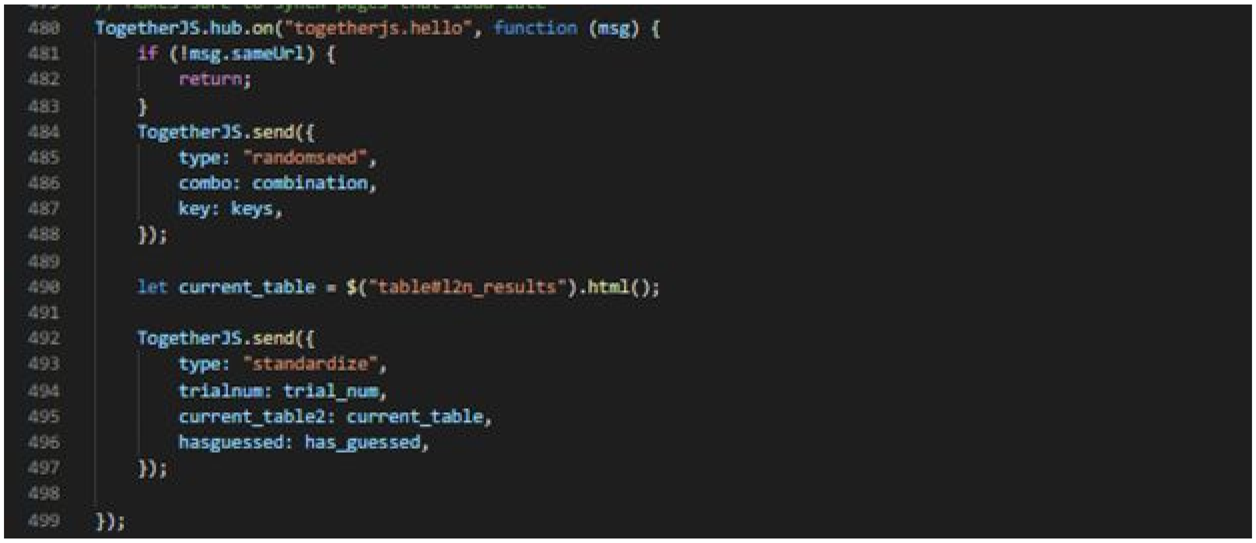
13.Randomseed and Alert



```
459  TogetherJS.hub.on("randomseed", function (msg) {
460      if (!msg.sameUrl) {
461          return;
462      }
463
464      combination = msg.combo;
465      keys = msg.key;
466      console.log(combination);
467
468  });
469
470  TogetherJS.hub.on("alert", function (msg) {
471      if (!msg.sameUrl) {
472          return;
473      }
474      alert("Please guess first before asking the computer a question");
475      return;
476  });
```

- TogetherJS.hub.on is an imported function from the Together.JS module, and is specifically designed to activate when a message from another browser is sent with a string that matches the first parameter.
- In this case, when a message is sent with the "randomseed" string, Lines 464-466 will activate and access the combo and key variables that are stored in the msg object that was sent.
- Lines 460-462 are checking to make sure that the message being sent isn't from a different Together.JS session. For instance, if group 1 has a user that sends a message, group 2 will not be affected from that message.
- Lines 470-475 were created at a different stage of development and are no longer necessary.

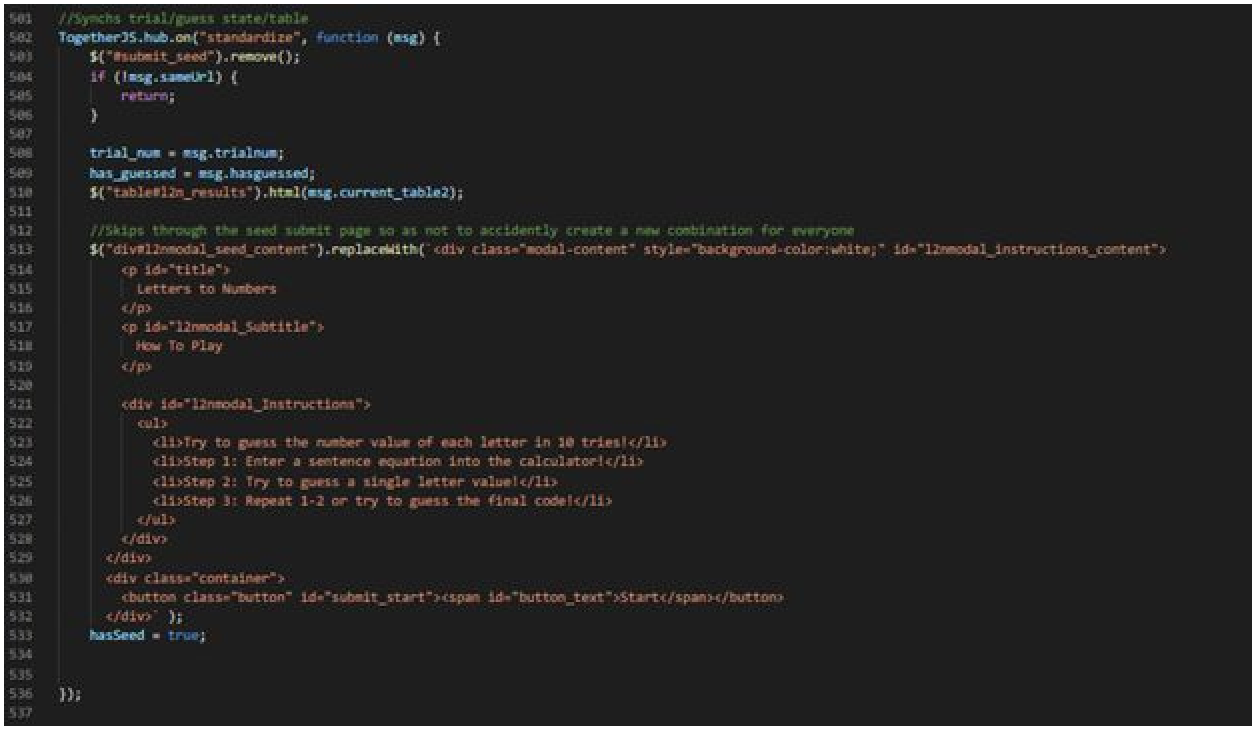
<u>14.Togetherjs.hello</u>



```
TogetherJS.hub.on("togetherjs.hello", function (msg) {
    if (!msg.sameUrl) {
        return;
    }
    TogetherJS.send({
        type: "randomseed",
        combo: combination,
        key: keys,
    });

    let current_table = $("table#l2n_results").html();

    TogetherJS.send({
        type: "standardize",
        trialnum: trial_num,
        current_table2: current_table,
        hasguessed: has_guessed,
    });

});
```

- **Lines 480-499 are important for synchronization across all web browsers in the current session.**
- **The type “togetherjs.hello” is unique and is only sent when a new member has joined a session.**
- **If a user joins a session late, what current combination, keys, table, trial numbers, and guess state should all sync.**
- **Lines 484-488 and Lines 492-497 use TogetherJs.send to send the respective messages required, and the type key is necessary so that a browser can recognize what function to run.**

<u>15.Standardize</u>

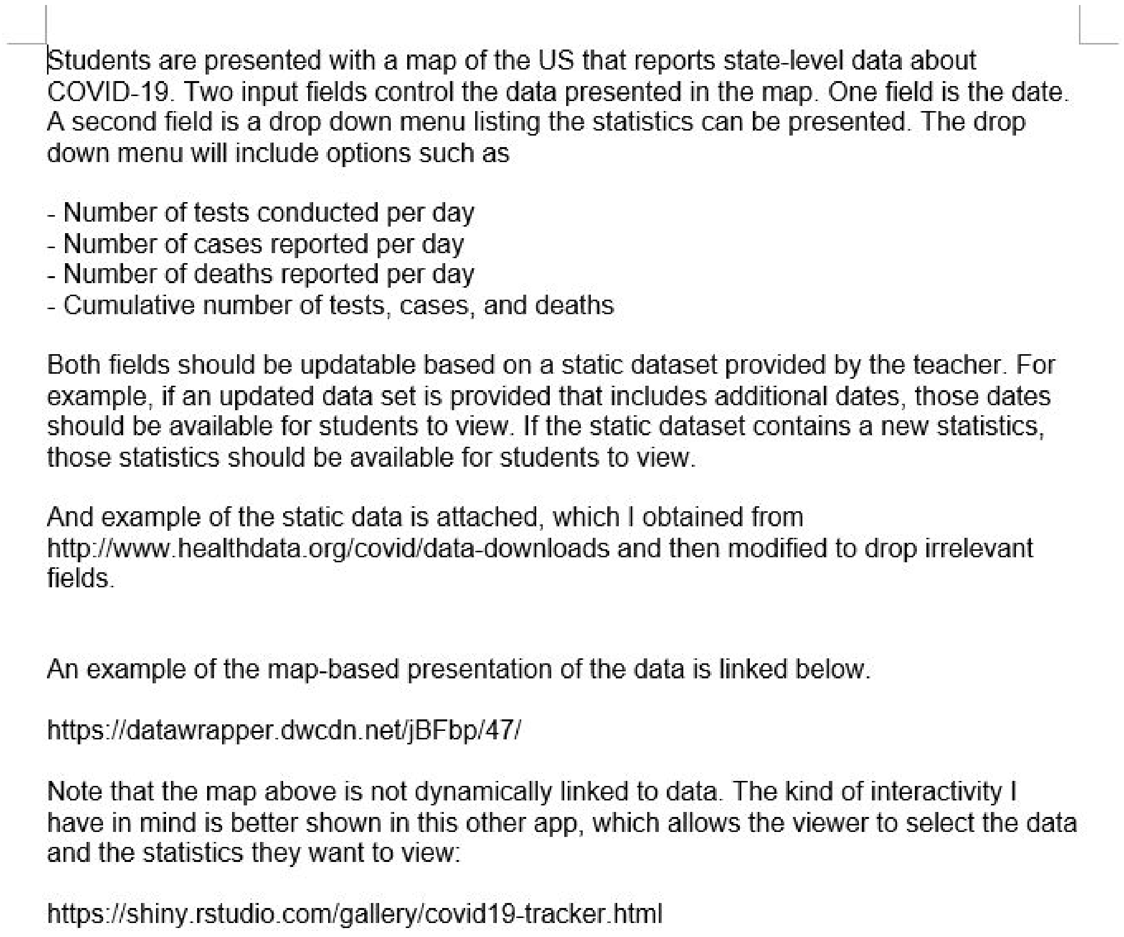


```
501  //Synchs trial/guess state/table
502  TogetherJS.hub.on("standardize", function (msg) {
503      $("#submit_seed").remove();
504      if (!msg.sameUrl) {
505          return;
506      }
507
508      trial_num = msg.trialnum;
509      has_guessed = msg.hasguessed;
510      $("table#l2n_results").html(msg.current_table2);
511
512      //Skips through the seed submit page so as not to accidently create a new combination for everyone
513      $("div#l2nmodal_seed_content").replaceWith(`<div class="modal-content" style="background-color:white;" id="l2nmodal_instructions_content">
514          <p id="title">
515            Letters to Numbers
516          </p>
517          <p id="l2nmodal_Subtitle">
518            How To Play
519          </p>
520
521          <div id="l2nmodal_Instructions">
522            <ul>
523              <li>Try to guess the number value of each letter in 10 tries!</li>
524              <li>Step 1: Enter a sentence equation into the calculator!</li>
525              <li>Step 2: Try to guess a single letter value!</li>
526              <li>Step 3: Repeat 1-2 or try to guess the final code!</li>
527            </ul>
528          </div>
529        </div>
530        <div class="container">
531          <button class="button" id="submit_start"><span id="button_text">Start</span></button>
532        </div>` );
533      hasSeed = true;
534
535
536  });
537
```

- **This function will only run when a new member has joined a session and will change the table, trial_num, has_guessed to all equal what the other members have on screen.**
- **Lines 513-533 are necessary because the browser will automatically show the seed generation modal first before the instructions, and in order to prevent browsers in the same session from having different combination/keys, the modal is manually changed at run time to have the instructions text.**

<u>COVID-19 GEO-MAP: WRITE UP AND DOCUMENTATION</u>

- **Exercise Write-up:**



Students are presented with a map of the US that reports state-level data about COVID-19. Two input fields control the data presented in the map. One field is the date. A second field is a drop down menu listing the statistics can be presented. The drop down menu will include options such as

- Number of tests conducted per day
- Number of cases reported per day
- Number of deaths reported per day
- Cumulative number of tests, cases, and deaths

Both fields should be updatable based on a static dataset provided by the teacher. For example, if an updated data set is provided that includes additional dates, those dates should be available for students to view. If the static dataset contains a new statistics, those statistics should be available for students to view.

And example of the static data is attached, which I obtained from http://www.healthdata.org/covid/data-downloads and then modified to drop irrelevant fields.

An example of the map-based presentation of the data is linked below.

https://datawrapper.dwcdn.net/jBFbp/47/

Note that the map above is not dynamically linked to data. The kind of interactivity I have in mind is better shown in this other app, which allows the viewer to select the data and the statistics they want to view:

https://shiny.rstudio.com/gallery/covid19-tracker.html

- **Due to this exercise being largely unfinished, all the necessary documentation is in the code.**