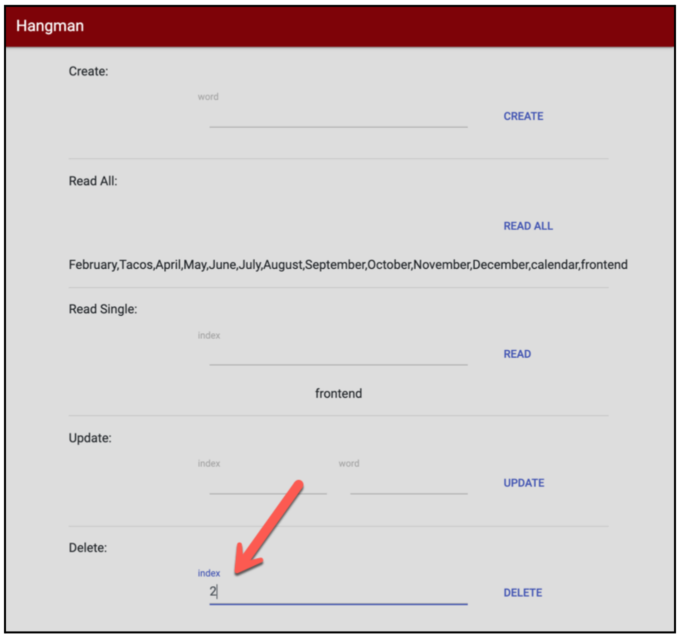
Hangman HW

Goal

In this lab you will be making backend API methods for a Hangman game, testing them with Postman, and then making a frontend website. This homework was once a final exam question, so hopefully it is a very good example of the kinds of things you should be able to do on your next exam.

Admin page for admin CRUD on the word list:

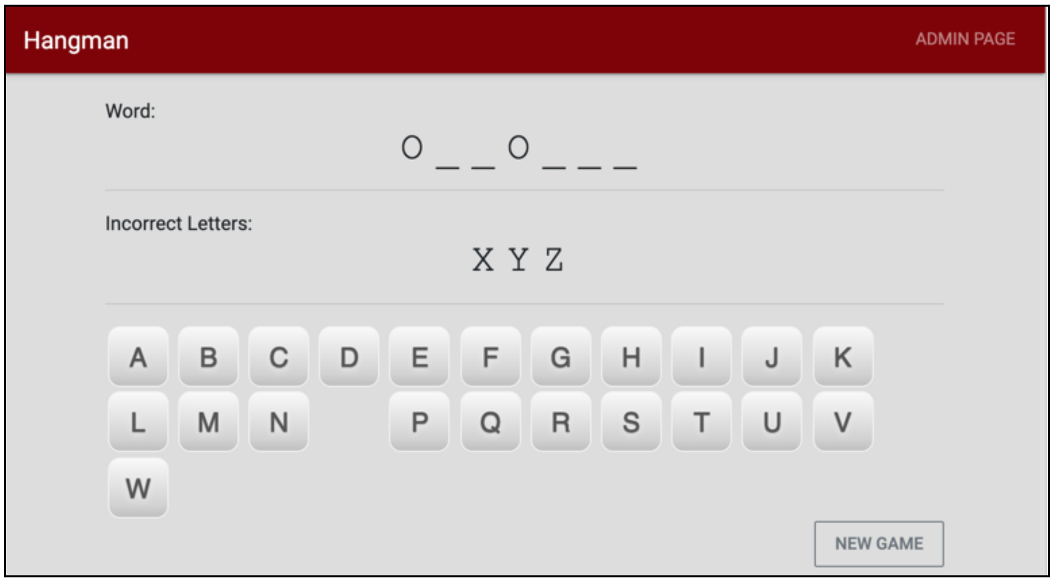


February,Tacos,April,May,June,July,August,September,October,November,December,calendar,frontend

frontend

2

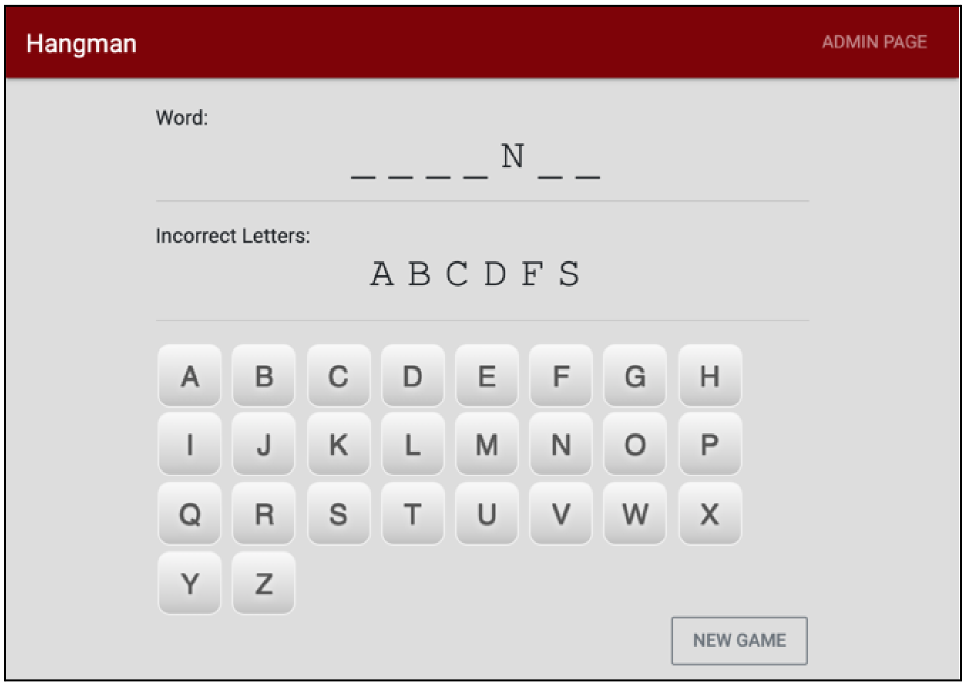
Player page to play a *mostly* feature complete game of Hangman (no gallows or person):



Setup

Before you can begin this homework, you need to get the starter code setup on your computer.

- Download this .zip file: Hangman starting code .zip
- Expand it (i.e. Extract All on Windows) to some location on your computer. Note: if you put this homework into version control, make sure it is in a **private** repo.
- Open the Hangman folder in VS Code.
- Open a terminal (usually within VS Code) and navigate to the integratedWebserver folder. `cd integratedWebserver`
- This homework uses express, so do an `npm init` (all the defaults are fine)
- Then `npm install express` (from within the integratedWebserver folder)
- Use nodemon to start serving the integratedWebserver.js file via the command, `nodemon integratedWebserver.js` (you hopefully already have nodemon installed globally on your computer. If not run `npm install nodemon -g`)
- Visit http://localhost:3000/ to make sure it's working. You should see this page:



You can click on the various buttons and see logs, but no buttons truly do anything yet (other than generating console logs). Also this homework uses 2 pages. You are currently looking at the Player page (which is the index.html file). You can visit the admin page by clicking the ADMIN PAGE button in the upper right (or just navigate to /admin.html). You can click the buttons there too to see logs, but again nothing truly happens for them yet. Then, come back to the Player page by clicking the word Hangman (upper left of that page). You can look at the starting code to your heart's content until you are comfortable with it. When you are ready, start the homework (next page).

Overview

This homework is broken into 2 parts, an Admin part (questions #1 and #2) and a Player part (questions #3 and #4). An Admin is responsible for controlling the list of possible words used in the Hangman game. Initially, the word list is set to the 12 months of the year via the db.json file:

```
["January", "February", "March", "April", "May", "June", "July",
 "August", "September", "October", "November", "December"]
```

However, the Admin has the rights to do Create, Update, and Delete on the word list. To allow CRUD on the word list, the Admin API needs to have the following methods.

Admin API

Description	method	path	Request	Response
Create	POST	/api/admin/add	word via JSON	word, index
Read All	GET	/api/admin/words	--	words array, array length
Read One	GET	/api/admin/word/:id	id via params	word, index
Update	PUT	/api/admin/word/:id	id via params word via JSON	word, index
Delete	DELETE	/api/admin/word/:id	id via params	index

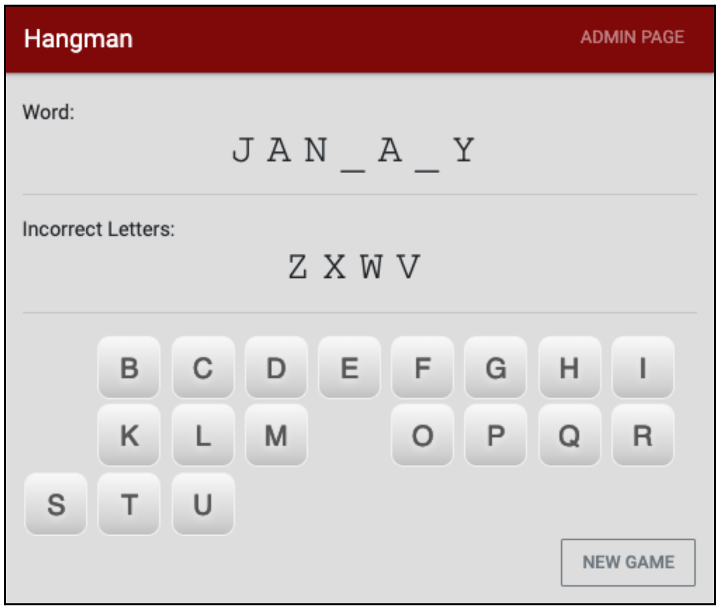
The data will be stored on the server using simple file storage into db.json as an array of Strings. You will implement the backend API for an Admin in question #1, then the frontend admin code in question #2.

The other part of this web app is the Player part. The Player part, questions #3 and #4, can be worked independently from the Admin part even if you have trouble with questions #1 and #2. A Player also has backend API calls, but they are different. A Player can never see the word they are guessing, so no responses from the backend actually return the word. The backend just tells the player the number of words, the length of an individual word, and the locations of the guessed letter (see API below).

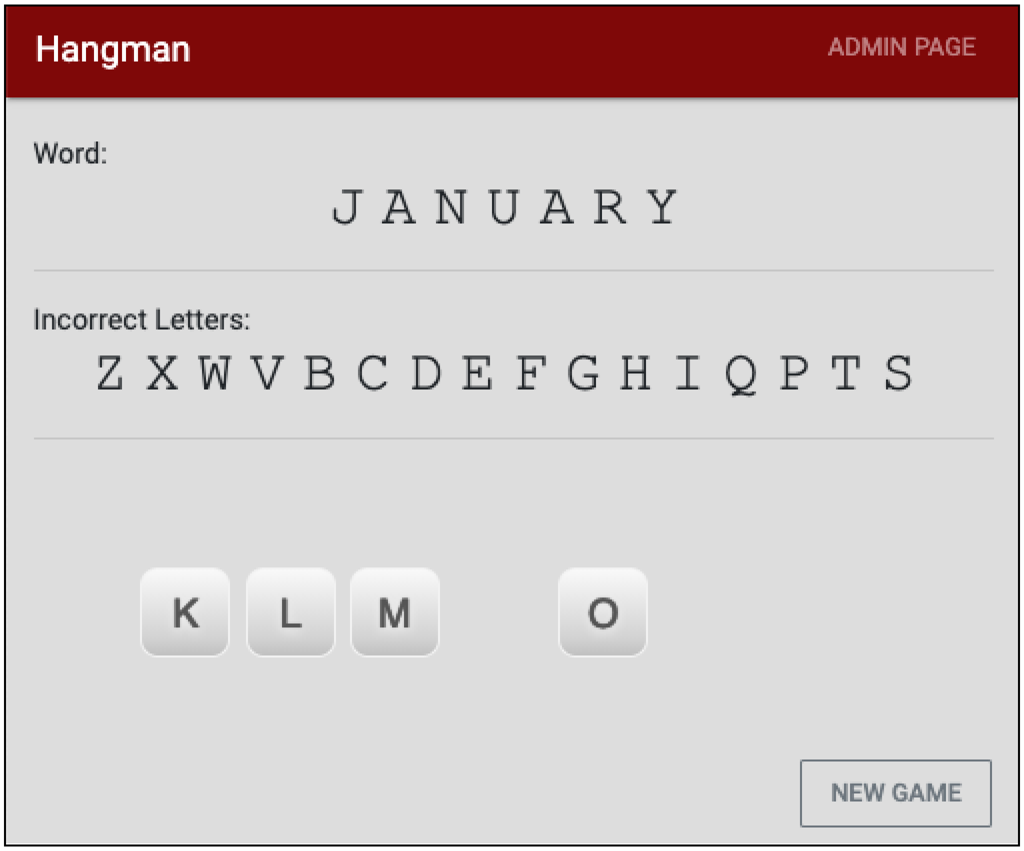
Player API

Description	method	path	Request	Response
Num words	GET	/api/player/numwords	--	length (i.e. the number of words)
Word length	GET	/api/player/wordlength/:id	id via params	index, length (i.e. the length of this word)
Guess	GET	/api/player/guess/:id/:letter	id via params letter via params	locations (an array of hits), letter, index, length

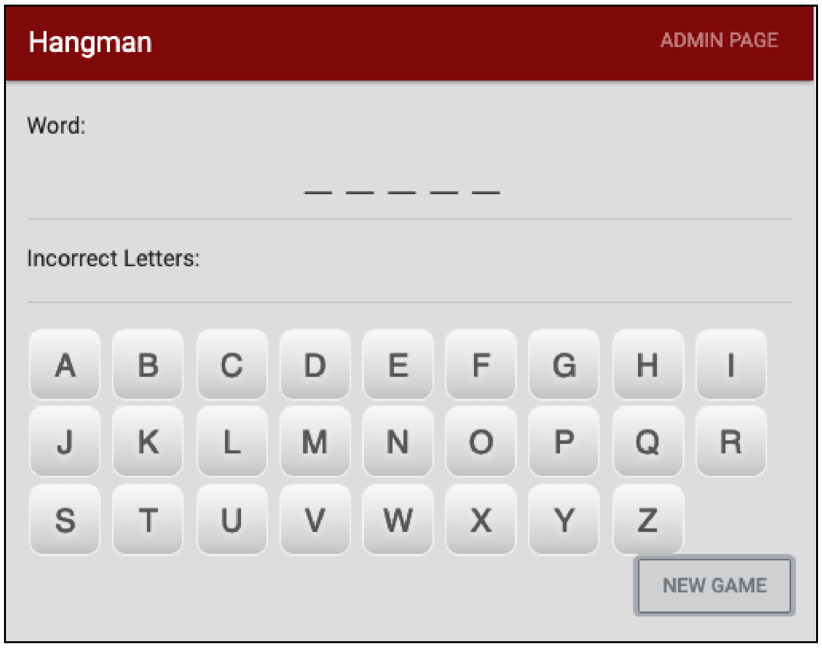
The Player API is used to allow a player to play the game of Hangman. Note, that in our version there is no winning or losing, you just display the letters that are hits (in the correct locations) and keep a list of the letters that are misses. For example, in the default screen that you have now on localhost:3000, it shows a pretend word in progress, the incorrect letters, and the keyboard (note, what that default screen doesn't show is that keyboard keys should be hidden after they have been used). So a real game in progress might look like this.



Notice that keys are hidden when they are used. A guessed letter either goes into the word or into the Incorrect Letters area. Again, there is no winning or losing in our game. So even after the word is complete:



Nothing changes. Users can keep guessing. There is no win or lose message. If a player clicks New Game, though, a new game would begin.



You will be implementing the Backend API for a Player in question #3, and then implementing the Frontend JavaScript for the Player page in question #4.

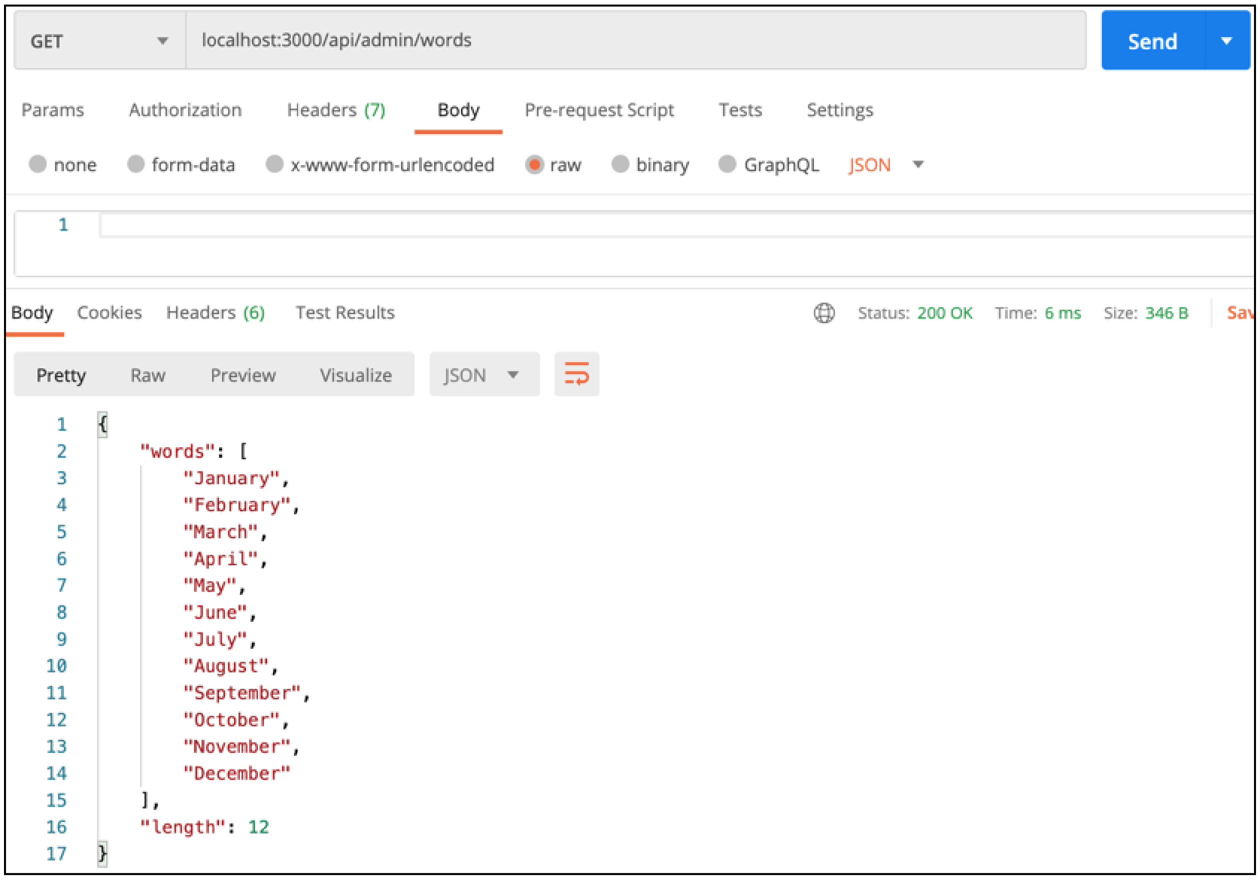
Question #1. Admin Backend API

You will only modify the file integratedWebserver.js for this question and test your API using Postman. You do not need to use Chrome at all for this question. There are 5 API methods you need to build. For each API call we've included a description and a screenshot of what Postman might look like once your code works.

Read All

Despite the memory mnemonics being the word **CRUD**, where C is first, I would suggest you implement **read all** first. This will return the entire word list. You should be able to find the location in integratedWebserver.js where this method goes and use your knowledge from the Full Stack app follow along with Jason to implement it. Here is a screenshot:

```
/**
 * Read all    - Get all words
 *   method:                    GET
 *   path:                      /api/admin/words
 *   expected request body:     none
 *   side effects:              none (as is the case for all GETs)
 *   response:                  {"words": ["...", "..."], "length": #}
 */
```



localhost:3000/api/admin/words

Body: [● raw] JSON

Status: 200 OK Time: 6 ms Size: 346 B

```
{
    "words": [
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    ],
    "length": 12
}
```

Notice the method, the path, the lack of a Body, and the result. Your solution should be able to perform the exact same request in Postman and your result should look like the one shown above (note: the order of **length** being first or **words** being first doesn't matter).

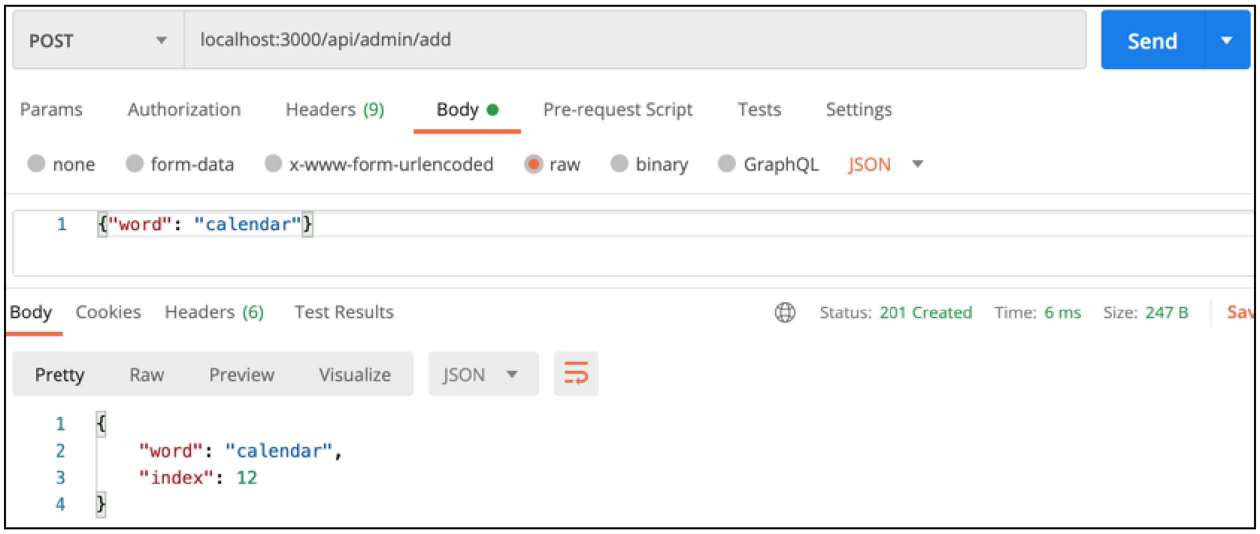
Postman Collection File

Optional way to save you time… There are a total of 8 different Postman queries in this homework. You can make them 1-by-1 as you test OR you can import a Postman Collection file. If you want to try to save time and import a file with the 8 example queries read about it here.

Create

Next, I would implement the method to add a word onto the end of the list.

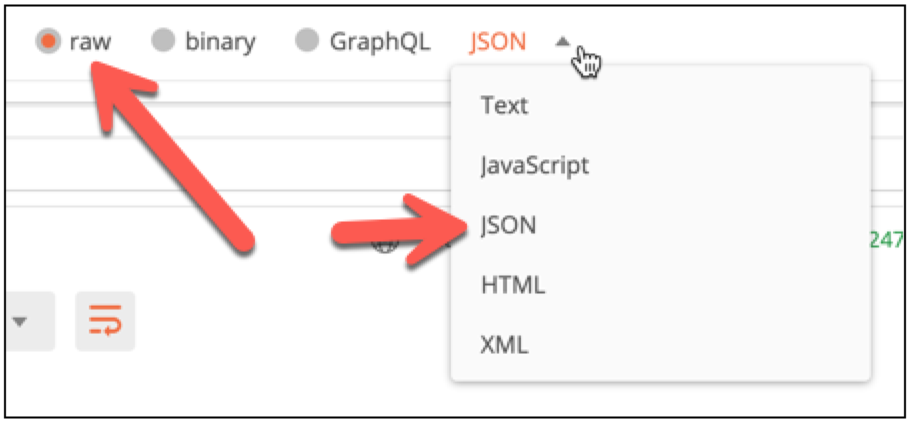
```
/**
 * Create - Add a word to the word list (i.e. the data array)
 *   method:                       POST
 *   path:                         /api/admin/add
 *   expected request body:        {"word": "..."}
 *   side effects:                 saves the word onto the data array
 *   response:                     {"word": "...", "index": #}
*/
```



```
POST  localhost:3000/api/admin/add
Body: raw  JSON
{"word": "calendar"}
```

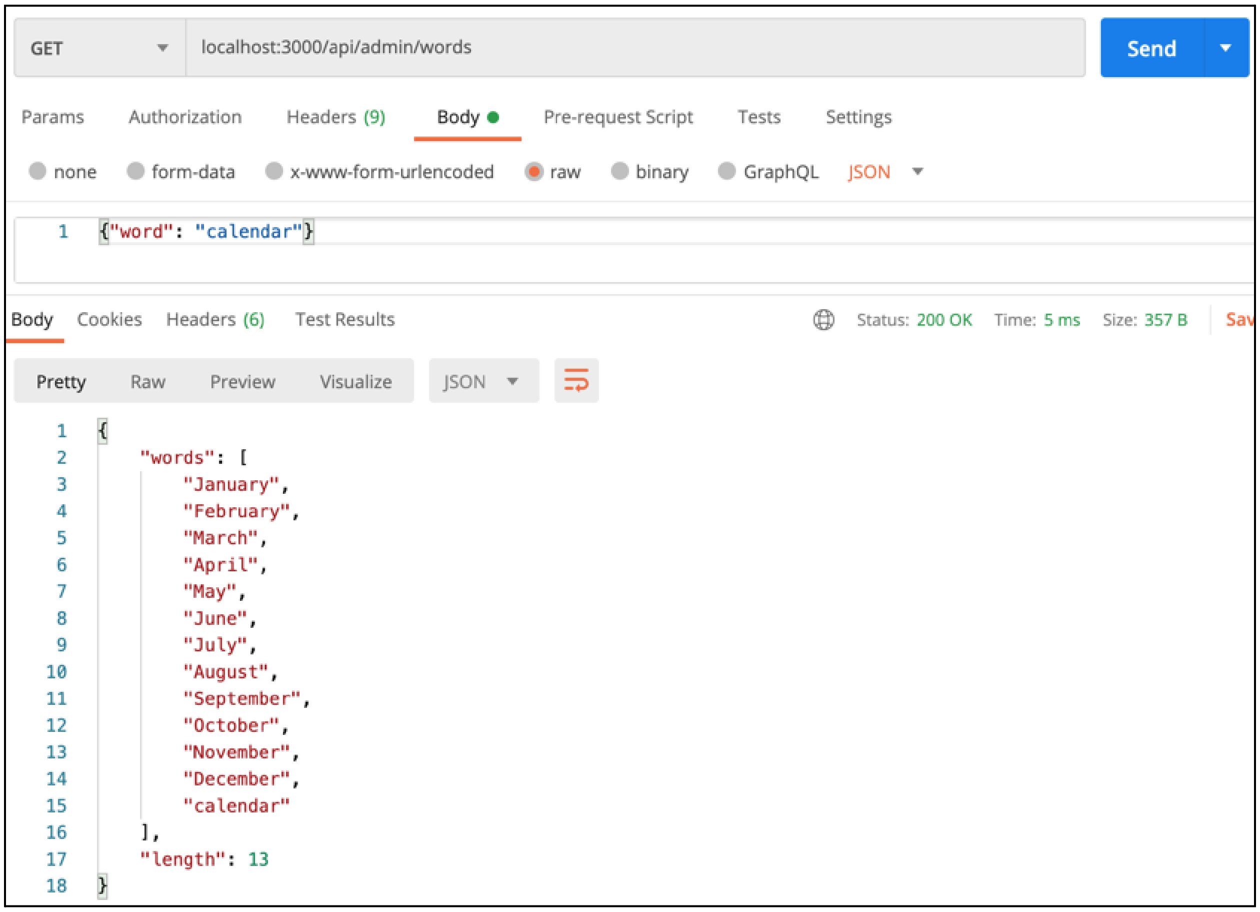
```
Status: 201 Created   Time: 6 ms   Size: 247 B
{
    "word": "calendar",
    "index": 12
}
```

Notice the path is different this time and the method is a POST. Additionally, a Body has been added using raw → JSON (make sure to click the raw tab and select JSON from the dropdown).



The body used in this example is `{“word”: “calendar”}`. The response repeats the word back and lets you know the index for that word (i.e. it’s index in the array). The word is saved to the db.json file as well.

If you did a get all words at this point, the response would now include that word:



```
GET localhost:3000/api/admin/words
```

```
{"word": "calendar"}
```

Status: 200 OK Time: 5 ms Size: 357 B

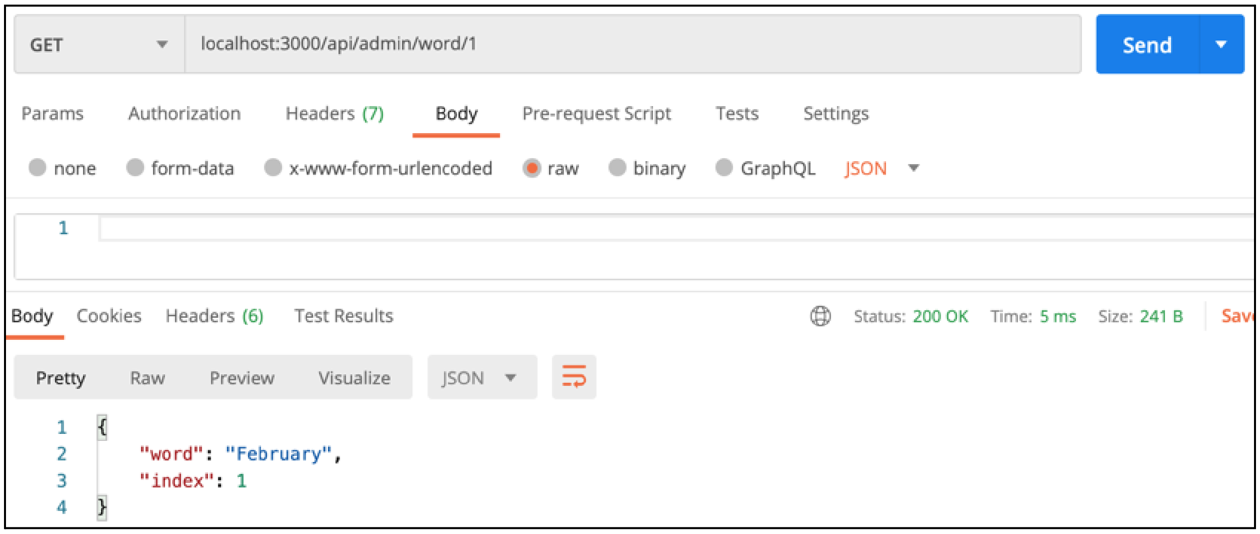
```
{
    "words": [
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December",
        "calendar"
    ],
    "length": 13
}
```

Notice that the method is GET again for this check and the path is different. Also I was very lazy on purpose in this screenshot. There is no body on a GET. However, I was lazy and just left that in Postman, but the body isn’t used at all. I knew that, so lazy was ok and I didn’t delete it from the Postman GET (go lazy!). Anyway you can see that calendar has been added to the list if your code is working correctly. Feel free to play around and add a few words. We’ll test your code with different data anyway, so don’t worry about messing up your db.json file with extra junk (do remove any **null** values if you accidentally make a goof during development though).

Read One

Next we'll add a method that you don't really **need** in this app (since the data in this app is so simple). We just add it for completeness. A Read for a single word. Example.

```
/**
 *  Read one - Get a single word at index
 *    method:                   GET
 *    path:                     /api/admin/word/:id
 *    expected request body:    none (notice the path parameter instead)
 *    side effects:             none (as is the case for all GETs)
 *    response:                 {"word": "...", "index": #}
 */
```



```
{
    "word": "February",
    "index": 1
}
```

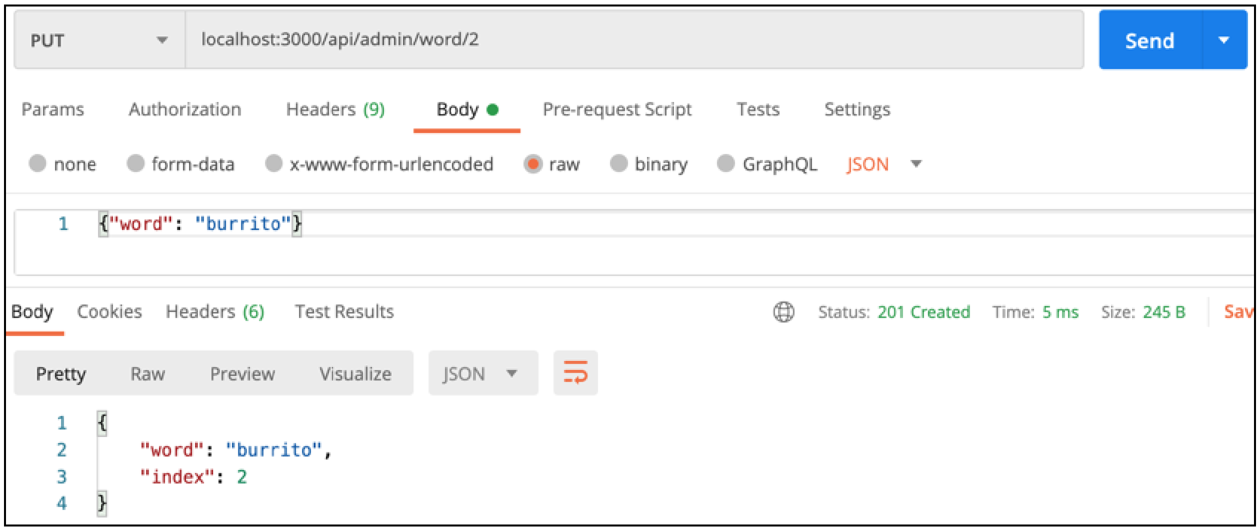
Notice the method is a GET. The id is passed in via a path parameter (index = 1 in this case), and there is no body. The response repeats the index back and tells you the word at that location. Notice that 0 would've gotten you January. Your solution should look exactly like the image above (again the order of word and index don't matter).

Another minor note: Throughout this homework you can assume users are good users. For example, you don't ever need to check that an id is valid. When we grade your work we'll, only test the positive test case. We won't try invalid requests.

Update

An Admin needs to be able to update words as well.

```
/**
 *  Update - Update a word at index
 *    method:                   PUT
 *    path:                     /api/admin/word/:id
 *    expected request body:    {"word": "..."}  (has a path parameter also)
 *    side effects:             saves the word into that index in the array
 *    response:                 {"word": "...", "index": #}
 */
```



localhost:3000/api/admin/word/2

Body: raw, JSON

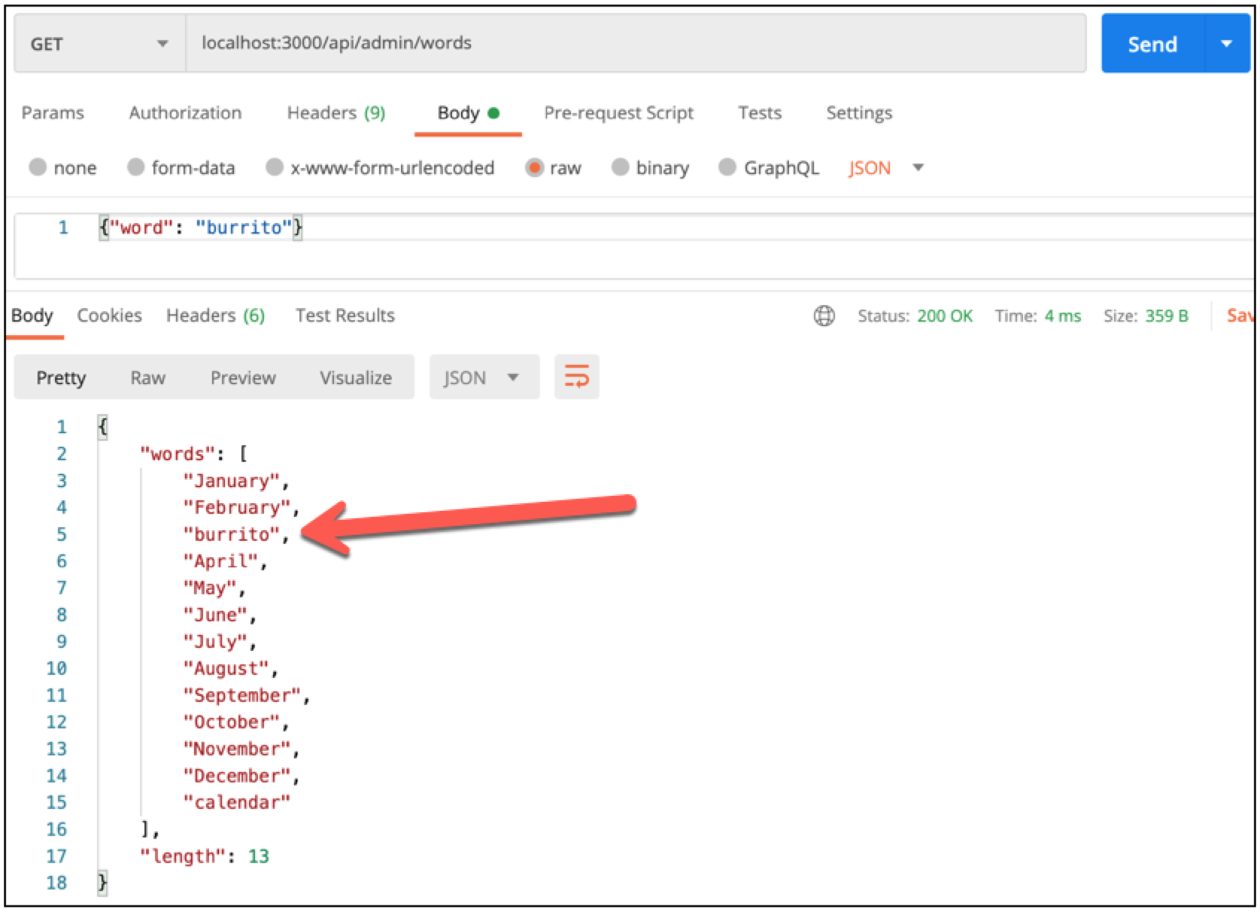
```
{"word": "burrito"}
```

Status: 201 Created   Time: 5 ms   Size: 245 B

```
{
    "word": "burrito",
    "index": 2
}
```

Notice the method is a PUT, the path is the same path as a Read One. In this example the index value is a 2 as set in the path. There is also a JSON body with the PUT. It contains the new word. The old word in that index will be blown away by the new word.

So a read all at this point would look like this.



localhost:3000/api/admin/words

Body: raw, JSON

```
{"word": "burrito"}
```

Status: 200 OK   Time: 4 ms   Size: 359 B

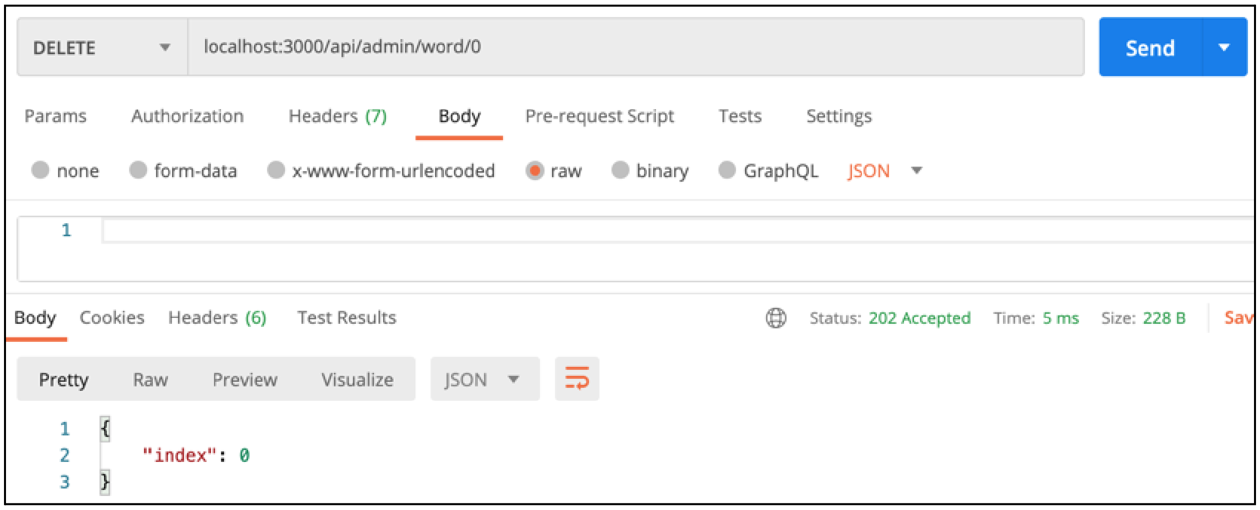
```
{
    "words": [
        "January",
        "February",
        "burrito",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December",
        "calendar"
    ],
    "length": 13
}
```

Notice in this check, the method is the GET and the path is changed. We've used this method many times before. The word at index 2 (which was March before), has now been updated to the word **burrito** (taken from the body of the PUT). The change is saved into the db.json file as well. Oh, also notice in this screenshot, I was lazy and left the body in there again for this check, because I knew the GET would ignore it. Just being lazy again. The body was only needed for the PUT.

Delete

You can't spell CRUD without a D. An Admin needs to be able to delete words.

```
/**
 *  Delete
 *    method:                       DELETE
 *    path:                         /api/admin/word/:id
 *    expected request body:        none (notice the path parameter instead)
 *    side effects:                 deletes the new word at that index
 *    response:                     {"index": #}  (index that got deleted)
 */
```



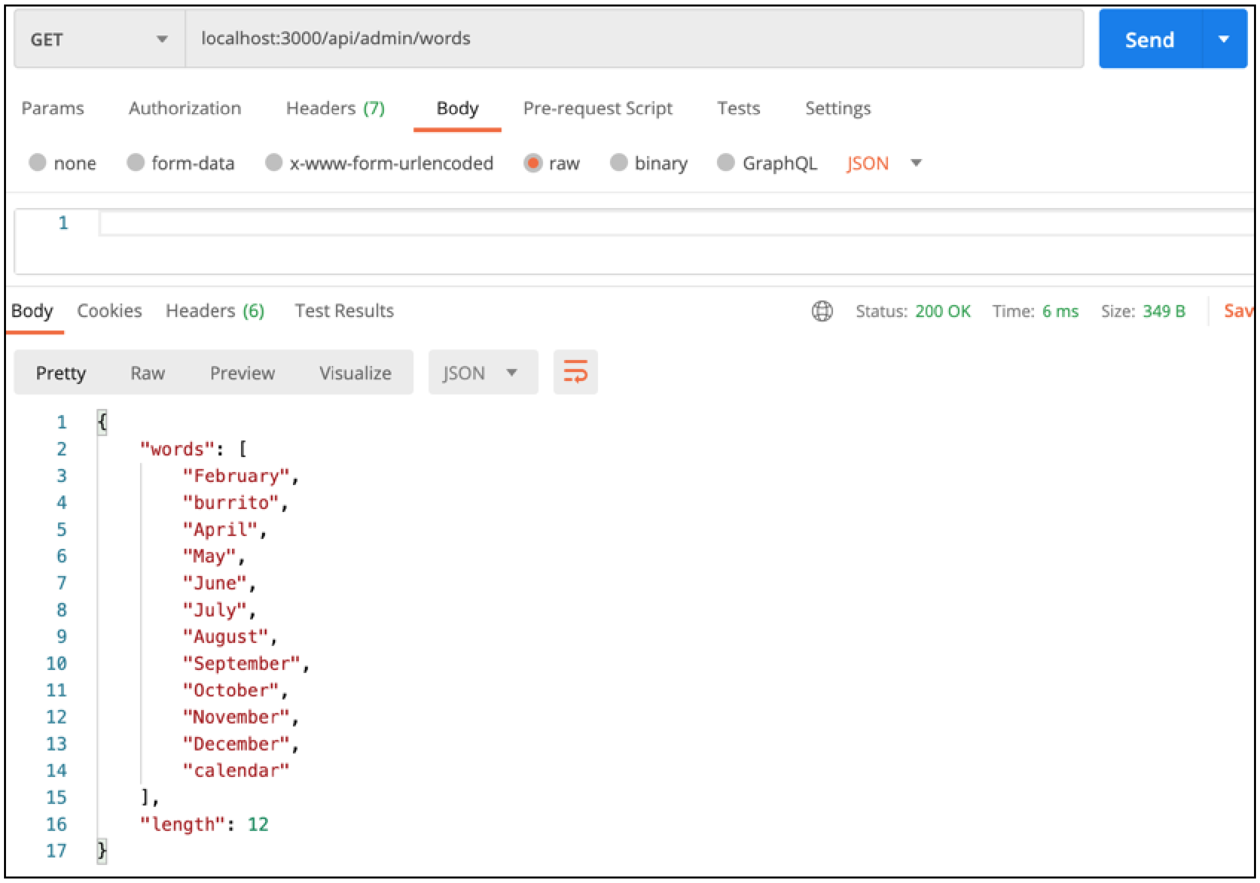
```
DELETE  localhost:3000/api/admin/word/0
Body: raw JSON (empty)

Status: 202 Accepted  Time: 5 ms  Size: 228 B

{
    "index": 0
}
```

Notice the method is a delete, the path this time is set for index 0. There is no post body (I was no longer lazy and just removed it). The response just repeats back the index that was removed. If you look very closely at this image (and all images), you should also set the response code. An accepted delete with a non-null response is a 202. The PUT and POST were 201 and the others were just 200. That is not graded, but good practice. I recommend you always add response codes.

Doing a read all at this point would give you the list without January.



localhost:3000/api/admin/words

Status: 200 OK Time: 6 ms Size: 349 B

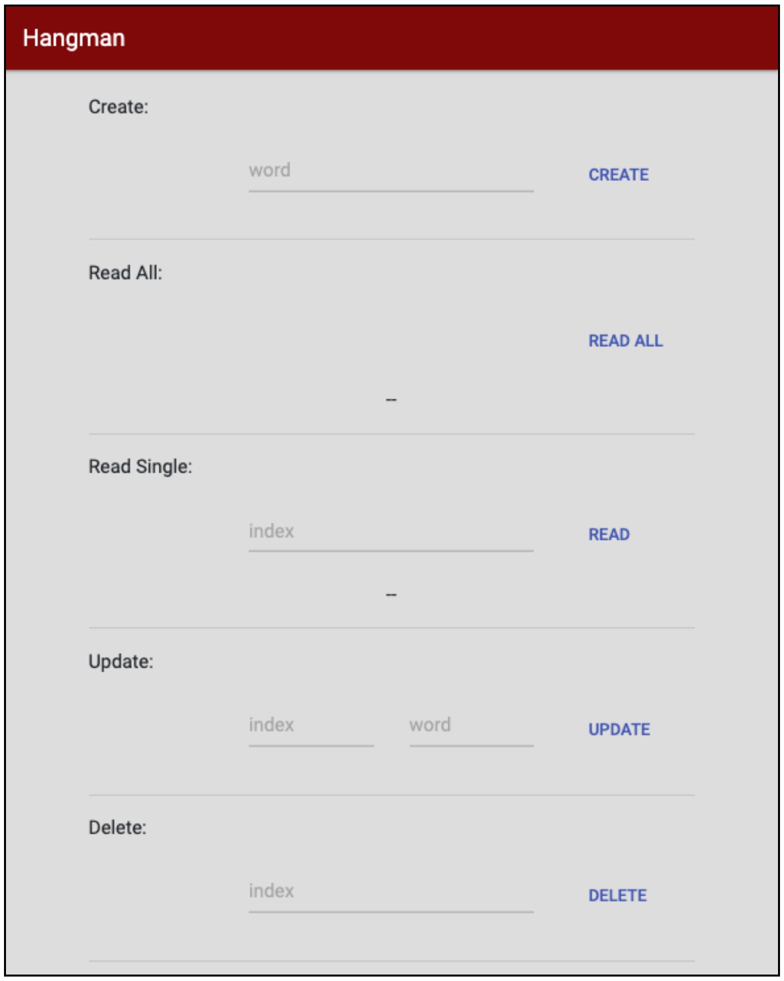
```
{
    "words": [
        "February",
        "burrito",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December",
        "calendar"
    ],
    "length": 12
}
```

Again that change is also saved to the db.json file. You are done with the Admin API.

Once you have all 5 backend admin API methods complete you are finished with question #1 and ready to move on to question #2. You will submit the whole homework together at the very end.

Question #2. Admin Frontend JS

Assuming you have the Admin backend API working with Postman, you can now move on to the frontend code. Your changes this time will happen within **main.js**. You will use Chrome to see your work and you don't need Postman at all for this question. Within Chrome visit the Admin Page (click the ADMIN PAGE button in the top right or just visit the url http://localhost:3000/admin.html). It should look like this:

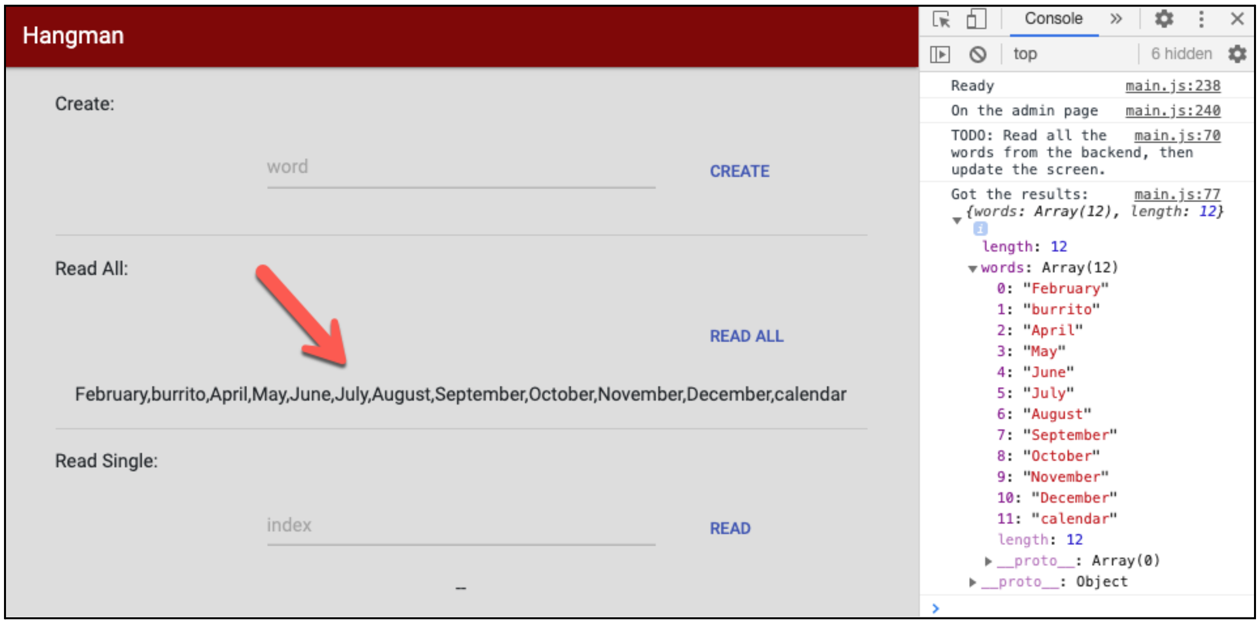


You will write frontend code to test your 5 backend admin API methods using fetch requests. Open main.js and find the AdminController class. You will notice that we are NOT doing an MVC separation, only a separation between pages. So your Model and Controller code will all go into the AdminController class. You should notice that the constructor is done, which connects the HTML buttons etc. to the code, but the methods need to be implemented by you. Add a few items into the input boxes and click on a few of the buttons until you understand how everything is connected before you start.

Read All

GET /api/admin/words - Get all words

Again despite the word being CRUD, which starts with a C, I would recommend you start with **Read All** on the client side as well. So you will implement the readAll() method in the AdminController using a fetch request to your backend. So once it is implemented, a click to the READ ALL button should look like this:



February,burrito,April,May,June,July,August,September,October,November,December,calendar

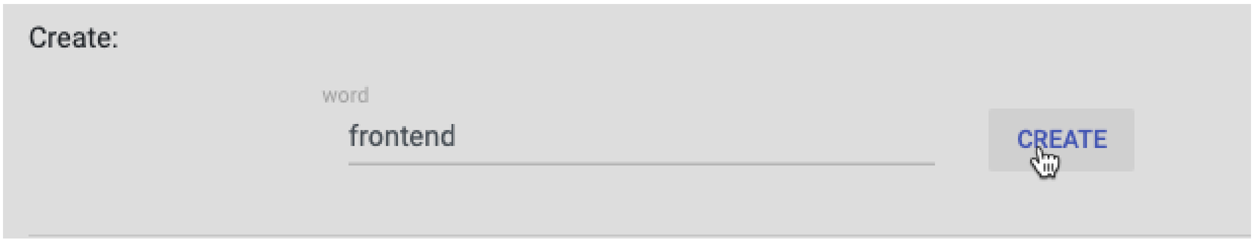
```
Ready                              main.js:238
On the admin page                  main.js:240
TODO: Read all the                  main.js:70
words from the backend, then
update the screen.
Got the results:                    main.js:77
▾{words: Array(12), length: 12}
    length: 12
  ▾words: Array(12)
      0: "February"
      1: "burrito"
      2: "April"
      3: "May"
      4: "June"
      5: "July"
      6: "August"
      7: "September"
      8: "October"
      9: "November"
      10: "December"
      11: "calendar"
      length: 12
    ▸__proto__: Array(0)
  ▸__proto__: Object
```

Notice that it's the same data that we saw in Postman, but this time using frontend code. The graded requirement is that you take the words array from the response and use it to set the innerHTML of the readAllOutput div. Again your changes should only be within main.js. We are not worried about it looking pretty, the image above is fine. You can also see that I optionally choose to print out the response to the console. The console logs are not graded requirements, but good for debugging as you are doing your development. I always print backend responses.

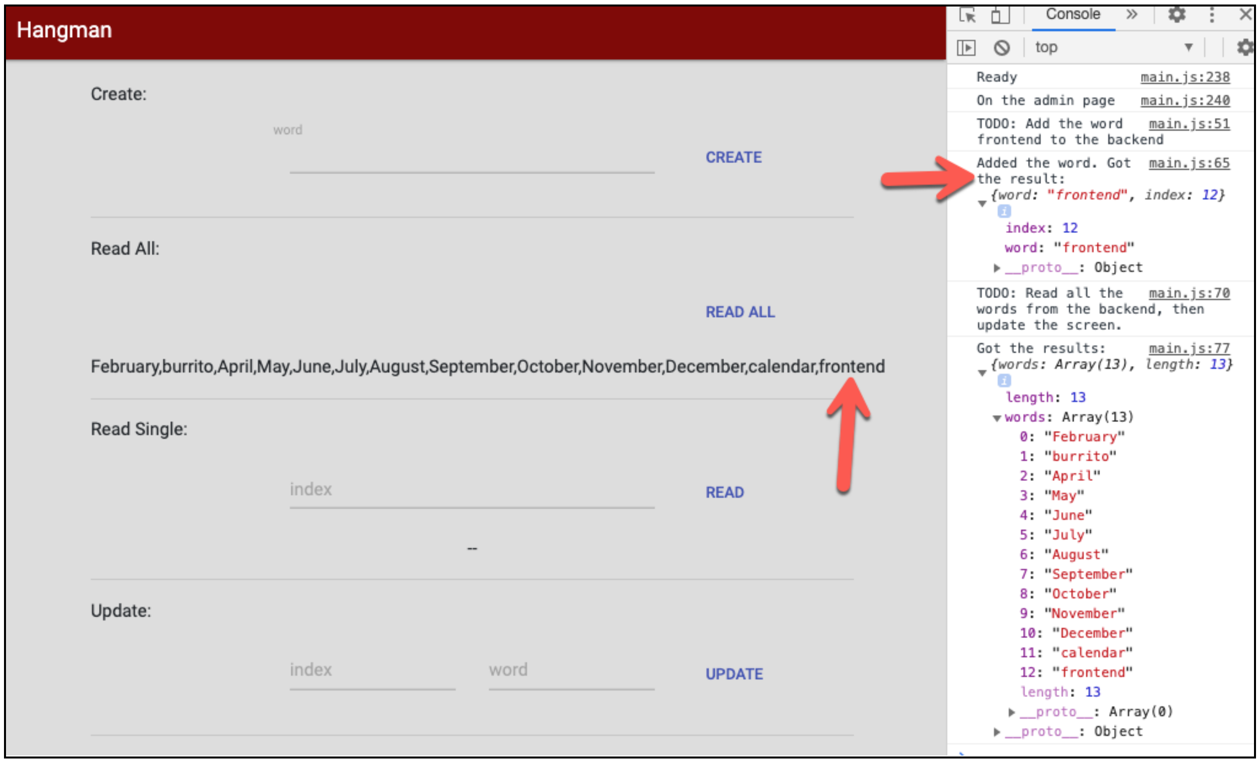
Create

POST /api/admin/add with body {"word": "..."} - Add a word to the word list

If the user types in a new word and clicks Create.



Then the word, **frontend** in this case, is added to the word list and saved to the db.json file. So if a Read All is done again the word **frontend** should be present at the end of the list.



February,burrito,April,May,June,July,August,September,October,November,December,calendar,frontend

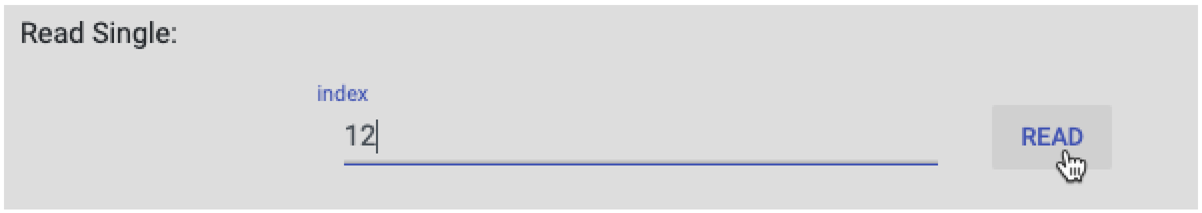
```
Ready                          main.js:238
On the admin page              main.js:240
TODO: Add the word             main.js:51
frontend to the backend
Added the word. Got            main.js:65
the result:
 ▼{word: "frontend", index: 12}
    index: 12
    word: "frontend"
  ▶__proto__: Object
TODO: Read all the             main.js:70
words from the backend, then
update the screen.
Got the results:               main.js:77
 ▼{words: Array(13), length: 13}
    length: 13
   ▼words: Array(13)
      0: "February"
      1: "burrito"
      2: "April"
      3: "May"
      4: "June"
      5: "July"
      6: "August"
      7: "September"
      8: "October"
      9: "November"
      10: "December"
      11: "calendar"
      12: "frontend"
      length: 13
    ▶__proto__: Array(0)
  ▶__proto__: Object
```

You can see in the logs that I printed the response to the create, which just echoes back the word and tells you the index for that word. Then I did another READ ALL and the word **frontend** is present at the end of the list.

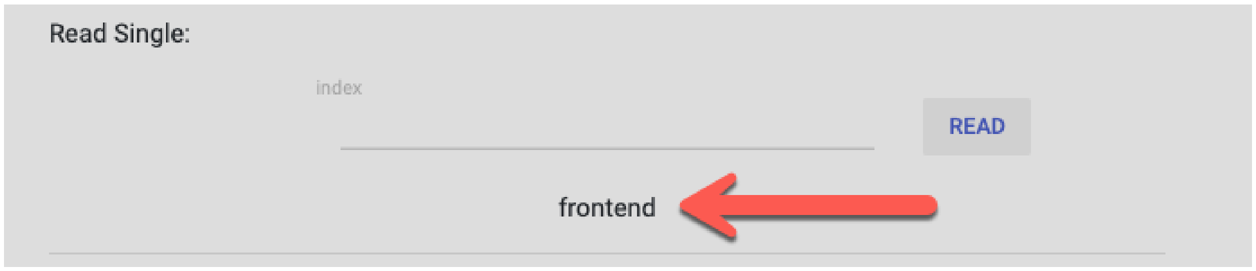
Read One

GET /api/admin/word/:id - Get a single word at index

If the admin wants to get a single word for a given index.



They can put in the index value, then click READ to display the word.

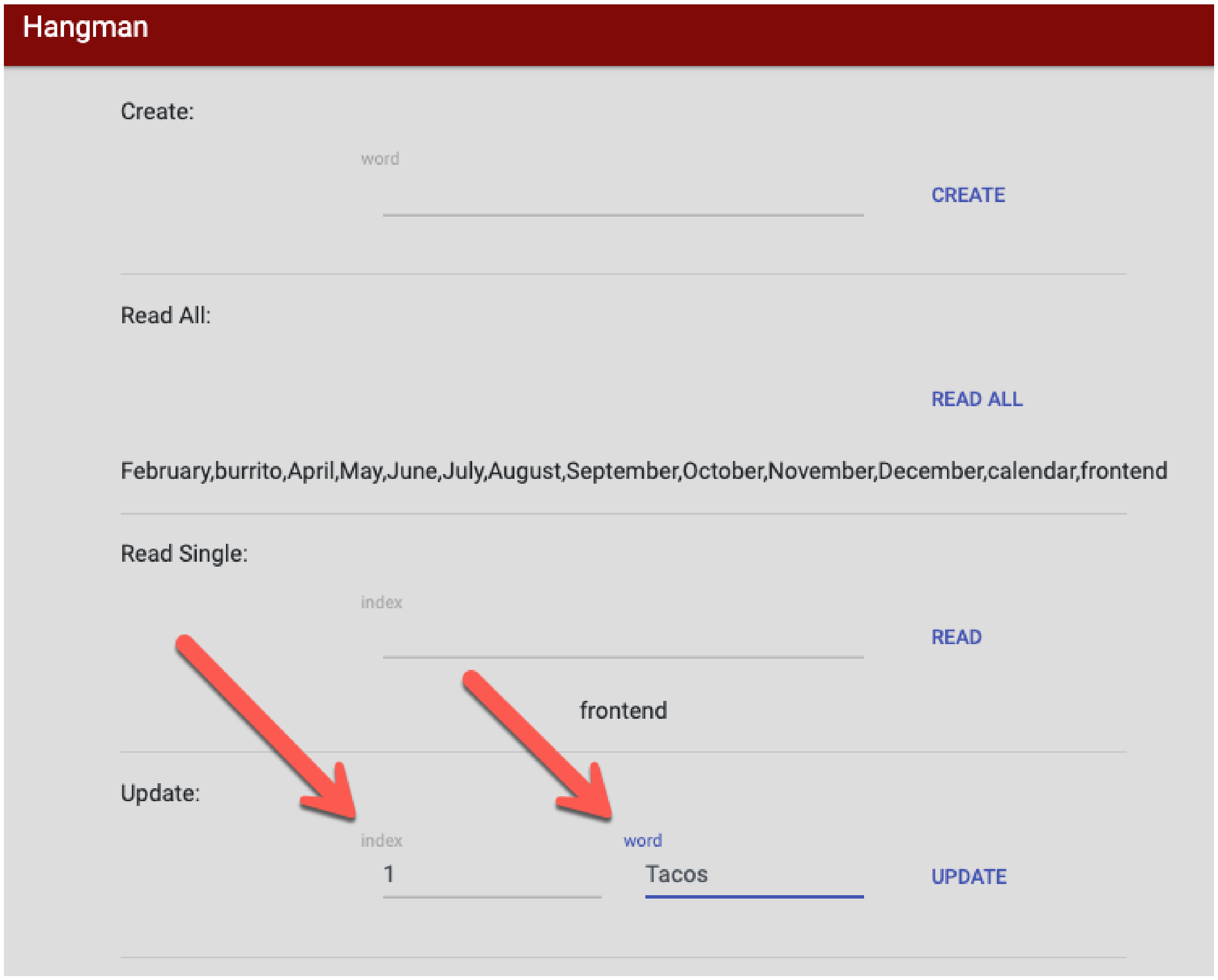


Again there isn't much value in Read One in this app, but it's present for completeness since more complex data **would** need a Read One API.

Update

PUT /api/admin/word/:id with body {"word": "..."} - Update a word at index

The admin also needs to be able to change words using the frontend. The admin provides an index and a new value.



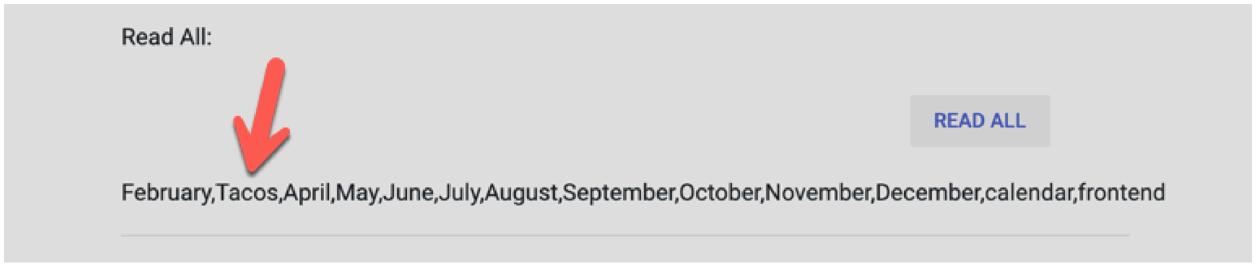
February,burrito,April,May,June,July,August,September,October,November,December,calendar,frontend

frontend

1

Tacos

When the admin clicks UPDATE, that word is set and the old word in that location is gone. You can see it worked via another click to READ ALL.



February,Tacos,April,May,June,July,August,September,October,November,December,calendar,frontend

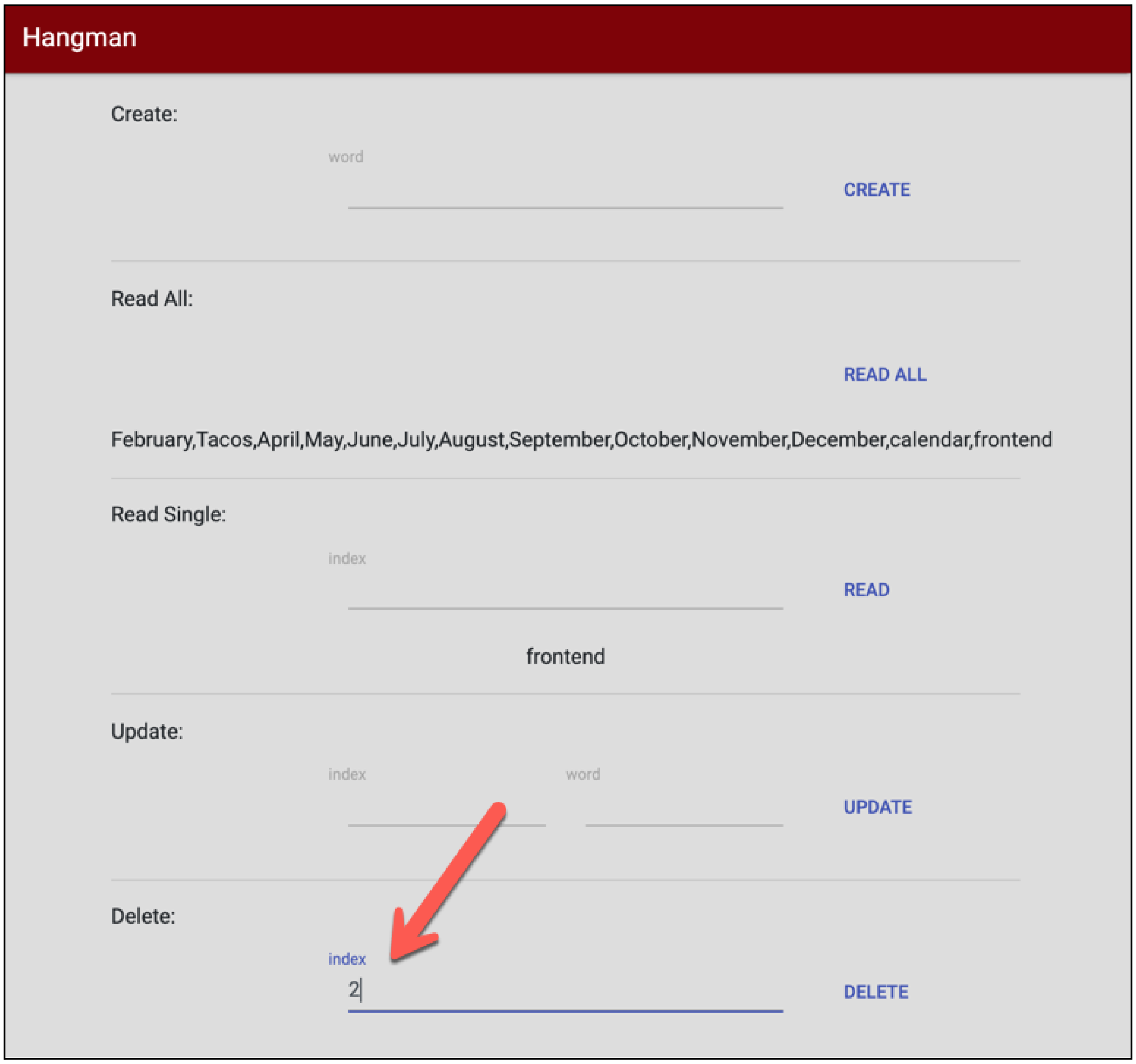
Note, as is best practice, your app should be able to handle bad users (i.e. indexes that are too big or negative, or not numbers etc.). However, when grading your work **we will only ever be**

good users. We will always provide valid indices. We will always put in properly formatted values in the inputs boxes etc. So for a 100% grade, you just need to handle the normal cases. However, if you have time, some error checking is a good real world skill too! Anyway Update on the frontend should work once you complete this part.

Delete

DELETE /api/admin/word/:id - Delete a word at index

Last, but not least, we have delete. If the admin provides an index and clicks DELETE.

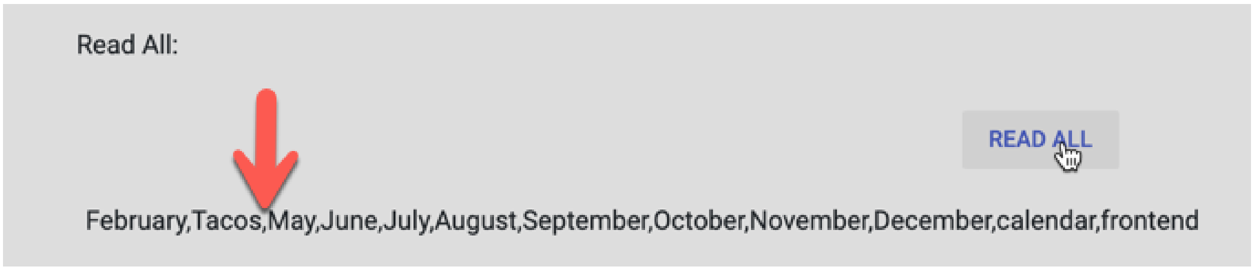


February,Tacos,April,May,June,July,August,September,October,November,December,calendar,frontend

frontend

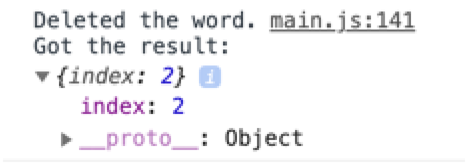
2

That element is removed and the list is saved to the db.json file. You can see the delete worked via another click to READ ALL.



February,Tacos,May,June,July,August,September,October,November,December,calendar,frontend

The word **April** was removed as expected. As with all my methods, I also printed the server's response to the console (optional).



```
Deleted the word. main.js:141
Got the result:
▼{index: 2}
   index: 2
 ▶__proto__: Object
```

Once you get that part finished, you are done with the Admin Frontend JavaScript. You are ready to move on to question #3, the Player Backend API.

Question #3. Player Backend API

For this question you will go back to the backend code to write the Player API. For this question, you will not need Chrome. You will use Postman, and all of your changes will be made within the **integratedWebserver.js** file. For the player API you have only 3 methods to write. They are all GET methods, so no changes are made to the db.json file (a GET never makes changes, it just reads data). You need to be able to get 3 things:

- The number of words in the word list
- The length of a particular word
- The locations of where a guessed letter appears in that word

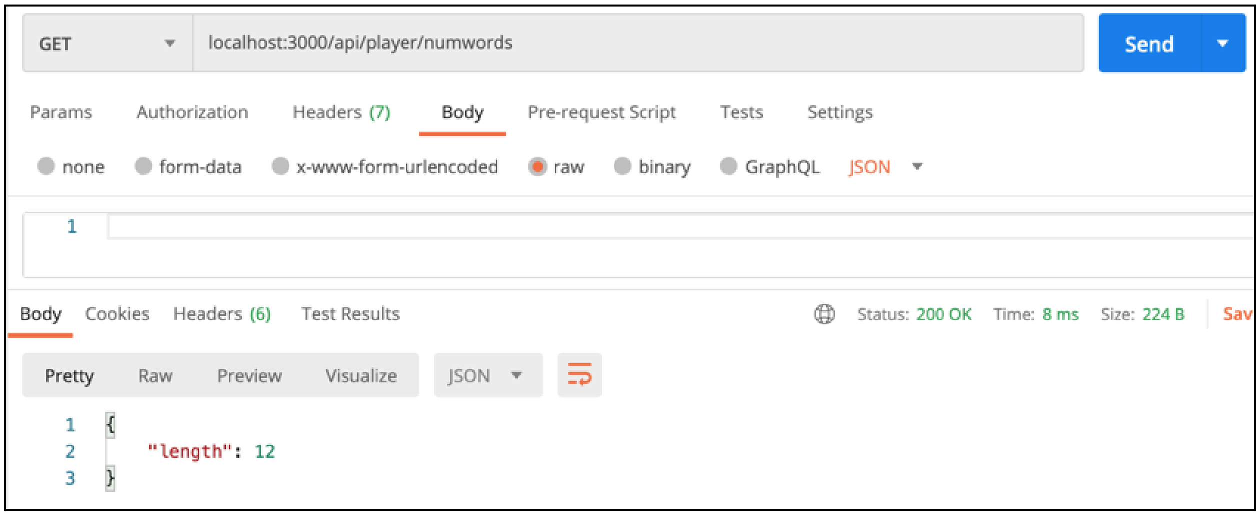
Note, that for security reasons the player backend API methods will never send the actual word. Pretend like the Admin API is secure with authentication, but the Player API messages could be intercepted by people that want to cheat at Hangman, so we never send the word in the Player API.

Num Words

The player page will randomly choose an index to use as the word. In this app, it is the responsibility of the frontend code to select an index (i.e. the index of the word to use for the current game). Initially that index could be 0 to 11. However, the Admin page might have added words or removed words. Perhaps the admin has added MANY words, or maybe they only have a few left. So the player page needs to be able to ask how many words are available.

```
/**
 *  Num Words - Get the number of words (i.e. the size of the data array).
 *    method:                     GET
 *    path:                       /api/player/numwords
 *    expected request body:      none
 *    side effects:               none (as is the case for all GETs)
 *    response:                   {"length": #}
 */
```

Here is an example of Postman using this API to get the number of words.



```
localhost:3000/api/player/numwords
```

```
{
    "length": 12
}
```

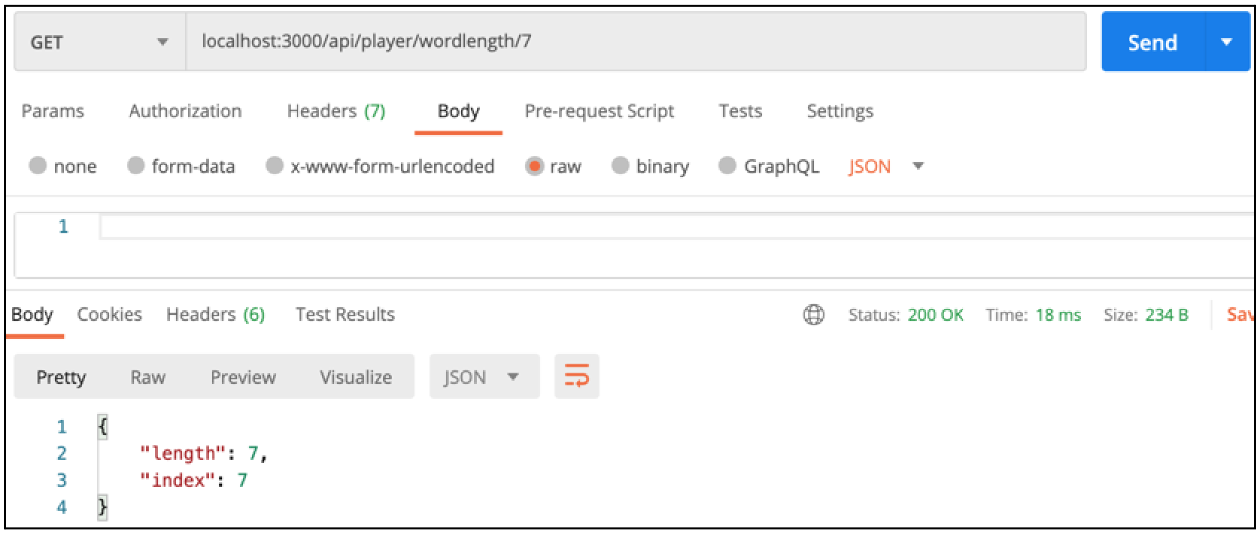
Notice the path used, the method type of GET, and the lack of a body. This method returned the length of 12 because there are currently 12 words (I started with 12, added 2 and deleted 2 for the frontend and backend tests, so back to 12).

Word Length

After the player page knows the number of words, the frontend code (to be written later) will randomly select an index in that range. For example, 7 (could've been any number 0 to 11 in **this** case). Then it needs to know how long the word is at location 7, so that the frontend can display the correct number of blanks to the player.

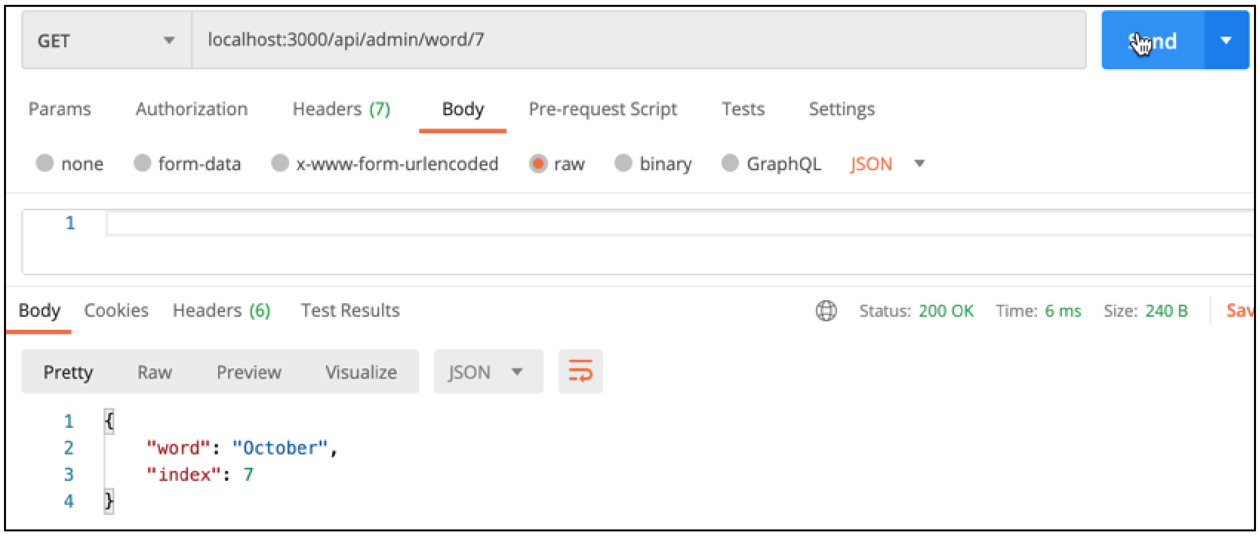
```
/**
 *  Word Length - Get the length of a single word at the given index.
 *    method:                     GET
 *    path:                       /api/player/wordlength/:id
 *    expected request body:      none (notice the path parameter instead)
 *    side effects:               none (as is the case for all GETs)
 *    response:                   {"length": #, "index": #}
 */
```

Here is an example from Postman.



```
{
    "length": 7,
    "index": 7
}
```

Notice the method, still GET, the path, the index in that path, and the result. Is that right? length = 7? Hum? Index 7 would be the 8th month, but I deleted 2 months in prior testing, so not the 8th month, but the 10th month. That would be October. I can totally cheat and test as an admin to see for sure what the word is at location 7.



```
{
    "word": "October",
    "index": 7
}
```

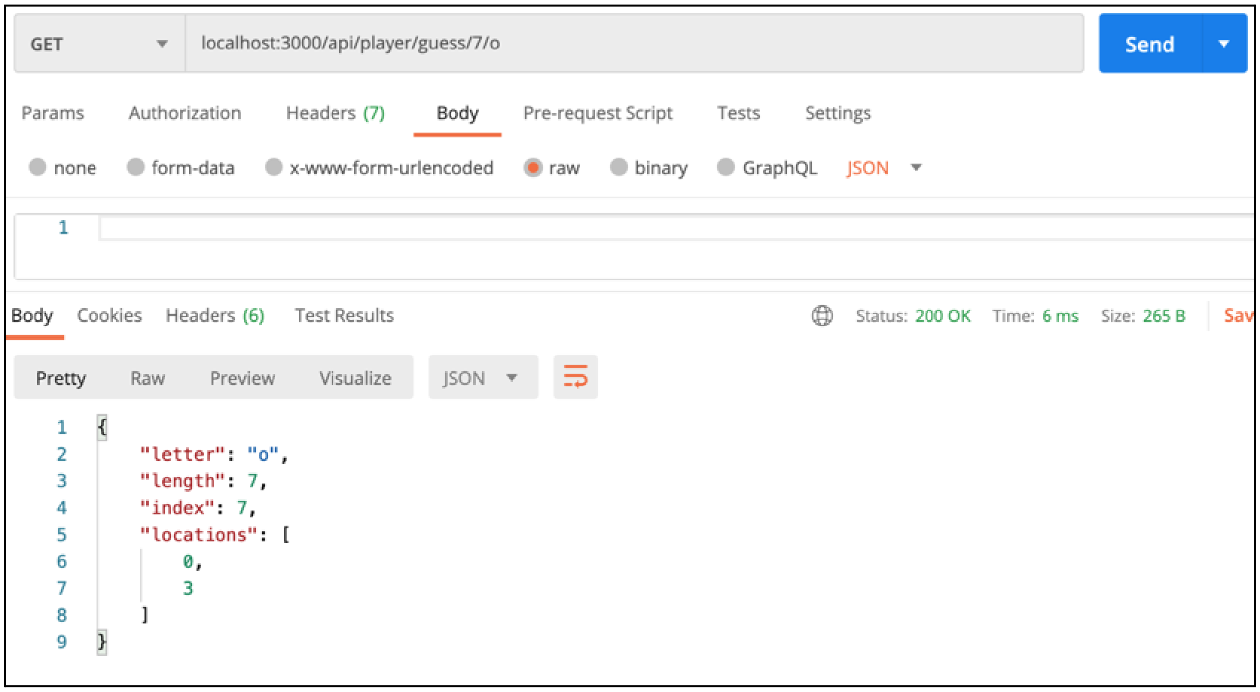
Notice, how I went back into the admin API for this cheater check. Yep, the word at index 7 is October, which does indeed have a length of 7. Notice, how we separate out the Admin from the Player api.

Guess

Finally, the most important method for a Hangman game, the ability to guess a letter.

```
/**
 *  Guess - Allow the player to make a guess. Returns where that letter is found.
 *    method:                     GET
 *    path:                       /api/player/guess/:id/:letter
 *    expected request body:      none (notice multiple path parameters instead)
 *    side effects:               none (as is the case for all GETs)
 *    response:                   {"letter":".","length": #,"index": #, "locations": [#, #, #]}
 *     example real response:     {"letter": "A", "length": 6, "index": 7, "locations": [0]}
 *     Notice the length and index are repeat info like Word Length would give.
 *     The interesting field is locations which is always an array, but could be an
 *             empty array [] if the guessed letter is not present in the word.
 */
```

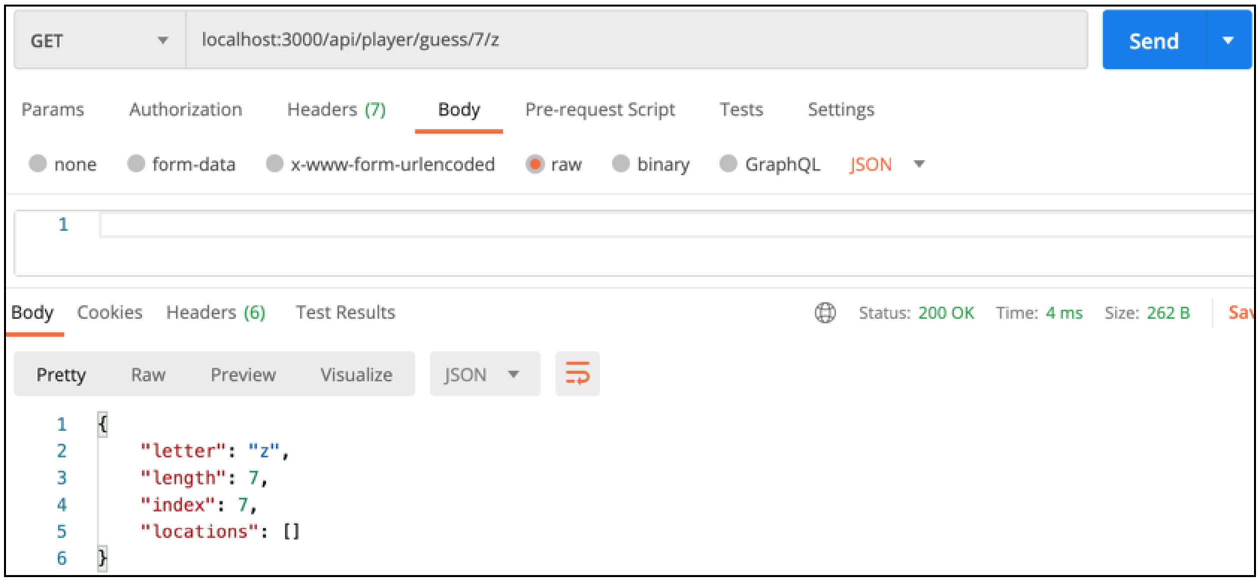
Again the word is not exposed from the backend. A response only includes information about that letter for the given word index. You will need to loop over the word on the backend and share which character indices in the word match the guessed letter. For example, if we guess the letter O in word 7, which we know is October we get:



```
{
    "letter": "o",
    "length": 7,
    "index": 7,
    "locations": [
        0,
        3
    ]
}
```

Notice the method, path, the 2 path parameters (word index = 7, letter = o), then the response. The response echoes back the index and letter that was guessed. It also repeats the length of the word, which should already be known. Then the most important field is the locations array. In this case, the array is [0, 3], since the letter **o** appears in the word October at those locations. Note, that your backend should be **case insensitive**. It should be able to receive upper or lower case letters and it should be able to compare them to the word using a case insensitive approach. So the case of the words saved in db.json don't matter at all to the player API. We will test this.

If a letter is not found, here I will guess z:



```
localhost:3000/api/player/guess/7/z
```

Status: 200 OK Time: 4 ms Size: 262 B

```
{
    "letter": "z",
    "length": 7,
    "index": 7,
    "locations": []
}
```

Then the locations array comes back as an empty array. No errors are thrown. No special fields are set, just an empty **locations** array.

Once you have that method working, you are finished with the Player API backend and ready to move on to the frontend for a player.

Question #4. Player Frontend JS

Finally, the most interesting part in a Hangman app, the frontend player code. You will not need Postman for this question. You will use Chrome with the player page http://localhost:3000/ and all of your changes will be made within main.js. Note: no HTML or CSS changes are needed and those files won't be submitted, so do all of your work in only main.js. If you open main.js and look at the PlayerController class you will notice that there are 3 methods that you need to implement.

- handleNewGame()
- handleKeyPress(keyValue)
- updateView()

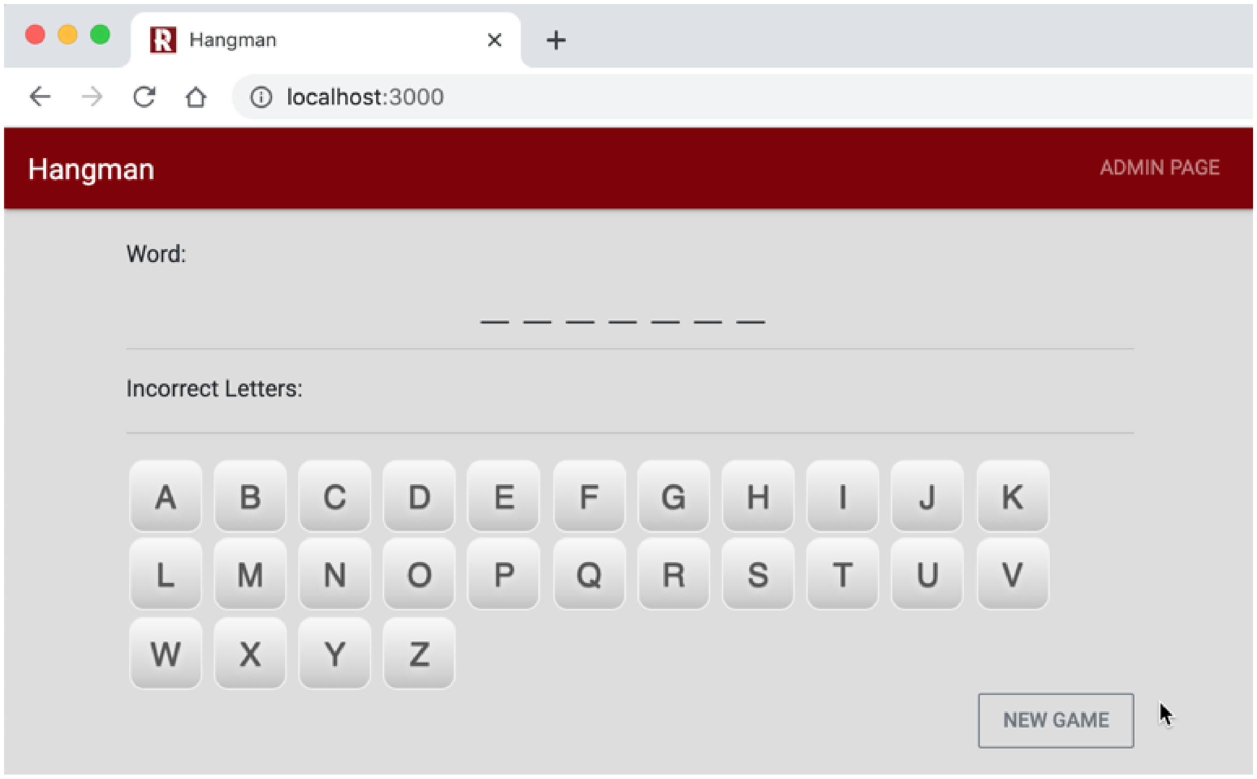
Unlike the Admin frontend, these methods do not map directly to the backend calls. You will need to figure out how to use the backend calls as you implement those methods. You will also need to figure out what instance variables you need to keep in the PlayerController to track the game state. Note: we're not messing with MVC, just put your state variables directly into the PlayerController class. You can add code to the PlayerController constructor if you would like (optional). For best practice, I usually list all the instance variables in the constructor with a default value, but, in JavaScript, you don't have to declare what instance variables you are using during the constructor. Instance variables can just be added later. Anyway, the backend does not keep the state of the current game at all, so it is up to you to decide what state variables you need to track the game in progress. You will, of course, need to use the 3 backend player API methods (note you may NOT use admin API methods). Those method include:

Num Words
GET   /api/player/numwords - Get the number of words (useful so you know what max value to use when a random index is selected)

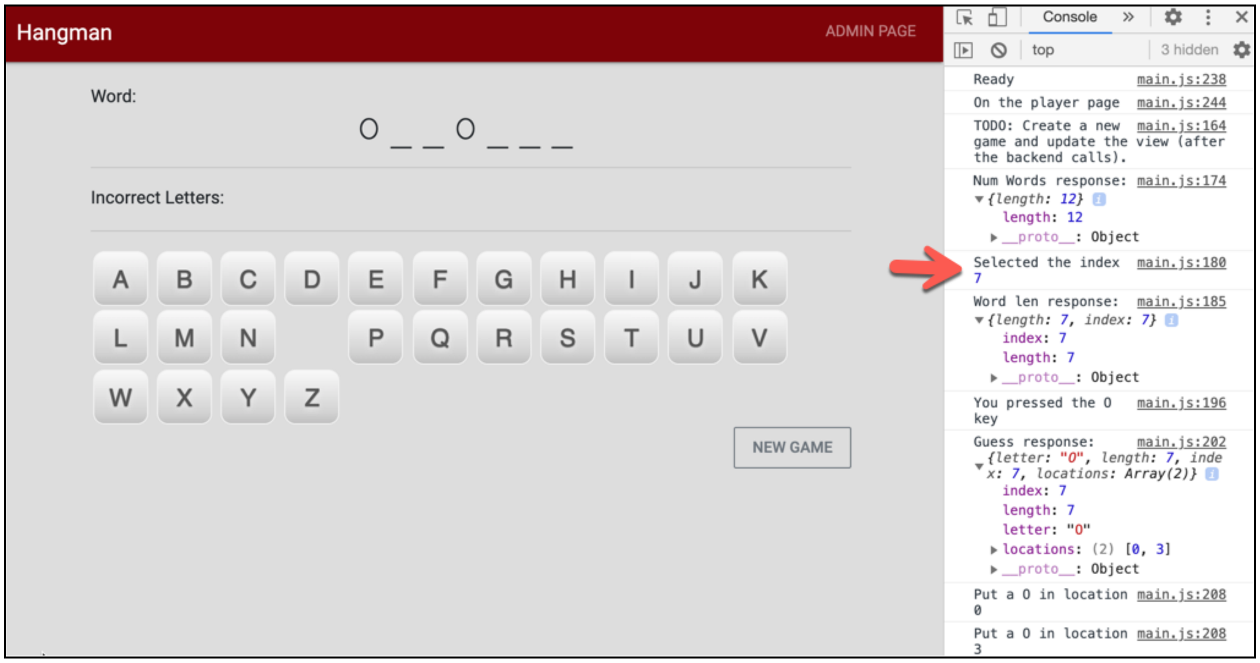
Word Length
GET   /api/player/wordlength/:id - Get the length of a single word at index (useful when you start a new game and want to show the correct number of blanks to the player)

Guess
GET   /api/player/guess/:id/:letter - Guess a letter in a word (useful when a key is pressed)

So think about what instance variables you want to use to save the game state. Here is an image that might help. For consistency with the earlier parts of this homework, I refreshed the page until I randomly got the word for index 7, which we know is October, which is length 7 (notice 7 blanks).

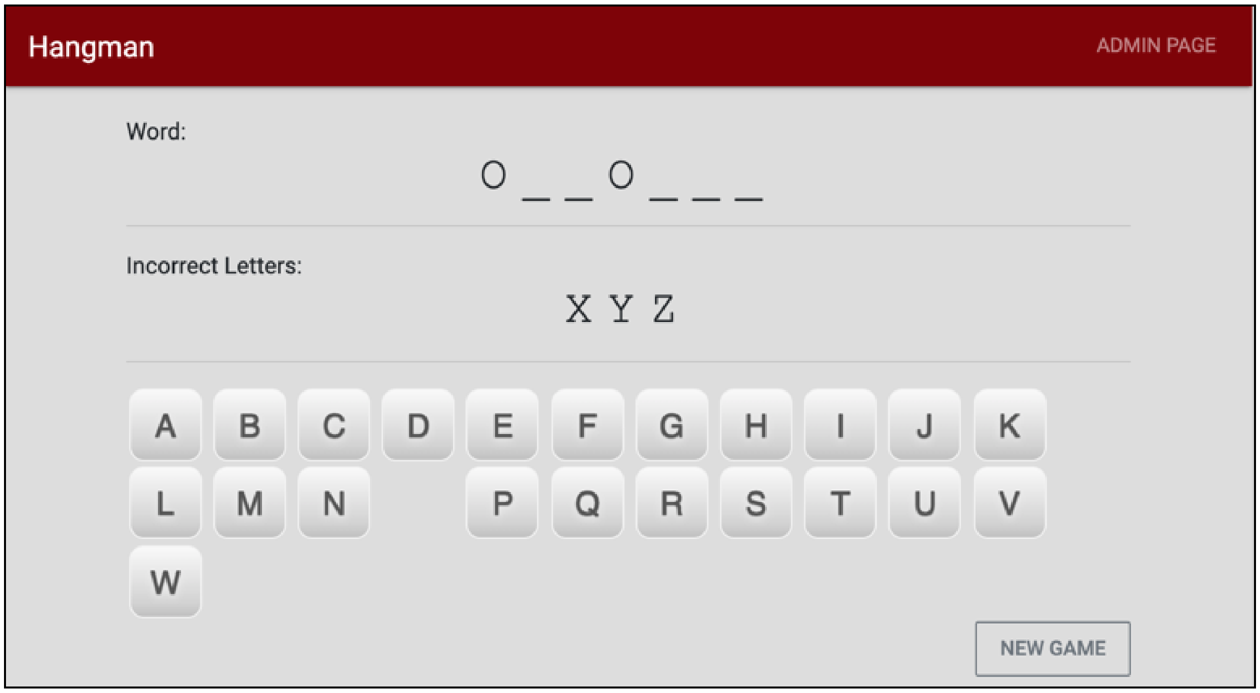


Then I guessed O.



```
Ready                    main.js:238
On the player page       main.js:244
TODO: Create a new       main.js:164
game and update the view (after
the backend calls).
Num Words response:      main.js:174
▼{length: 12}
    length: 12
  ▶__proto__: Object
Selected the index       main.js:180
7
Word len response:       main.js:185
▼{length: 7, index: 7}
    index: 7
    length: 7
  ▶__proto__: Object
You pressed the O        main.js:196
key
Guess response:          main.js:202
▼{letter: "O", length: 7, inde
 x: 7, locations: Array(2)}
    index: 7
    length: 7
    letter: "O"
  ▶locations: (2) [0, 3]
  ▶__proto__: Object
Put a O in location      main.js:208
0
Put a O in location      main.js:208
3
```

You might notice that in my logs, when the page first loaded, I got the number of words available and printed the response. Then my code randomly **selected an index within an appropriate range based on the number of words**. Then, I printed out the index that was randomly chosen to help me cheat when doing my testing. Then, I got the number of letters and displayed that many blanks. Notice the spacing is done via CSS, so "_______" (no spaces) will display fine, spaced out via CSS. Then when the player clicked on O, I did a backend call and printed the response. Then, I saved data to my instance variables and finally called update view. You can see this question is more work than prior questions in this homework. So what instance variables do you need? Well, I need the display word, and I need to know which letters I've guessed, so that keys can be removed. Also, if I guess X, Y, and Z the screen would look like this.



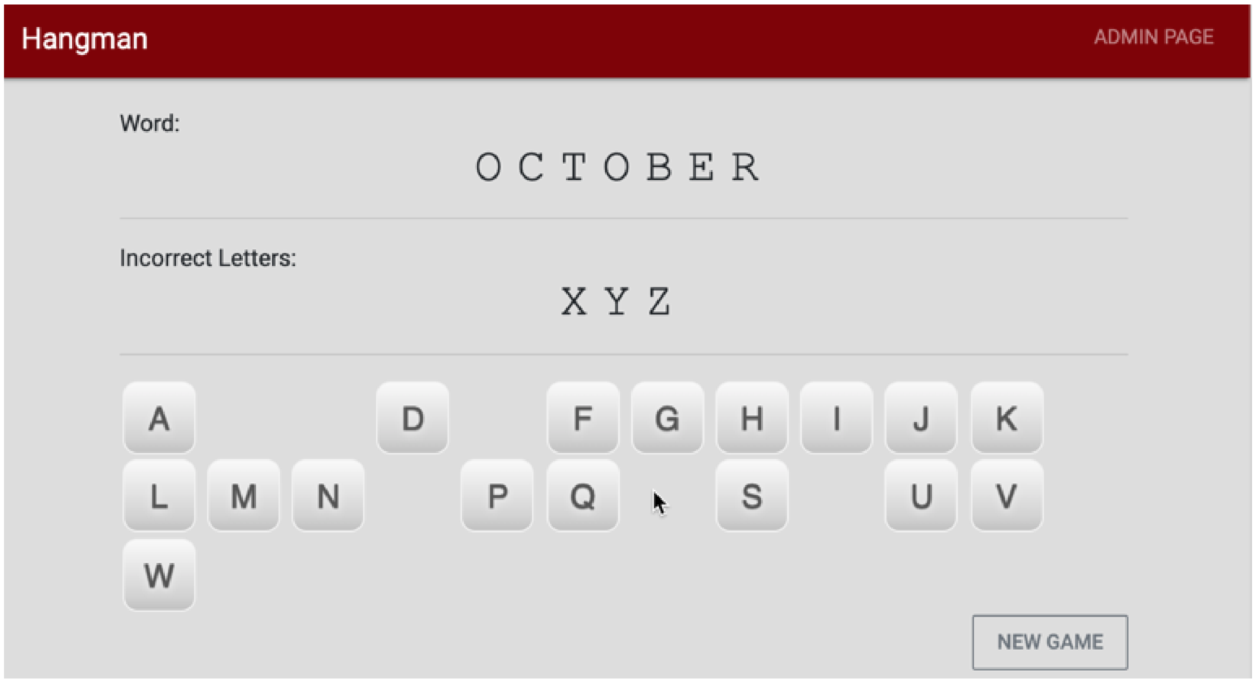
So I need to track the incorrect letters as well. That should be all the information you need to complete this question, but I'll provide a few optional hints as well.

A few hints (completely optional):

- You can save something as an array and then convert it to a string for display pretty easily. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join

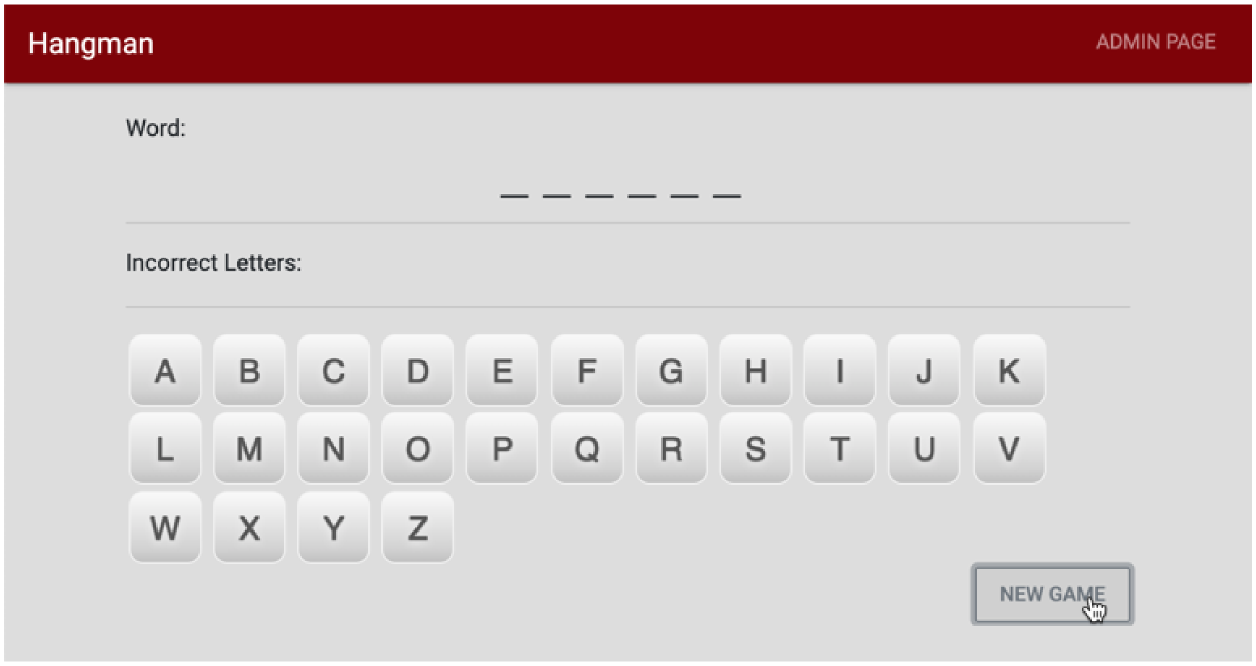
- You can get a random number from 0 to some max value using Math library functions. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

When you finish your app you should be able to play a game of Hangman. Again, nothing happens at the end of a round. For example:



Players could just keep guessing even after the word is complete. Nothing really happens when the word is done.

However, a click to New Game should start a new round.



Notice that the incorrect letters are removed, the keyboard keys are restored, and a new word is chosen. This word appears to be 6 letters long. I wonder what it could be?

Good luck completing your player frontend for Hangman!

Submitting your work

When you complete your work (or time runs out) you will submit 2 files to Gradescope:

1. integratedWebserver.js
2. main.js

You should only make changes to those 2 files. No CSS or HTML changes are needed for this homework.