CSSE280 Summer 2022

Exam 3 - Guess my number

You may use all of your prior coursework as well as the internet for searching. The only things prohibited:

- You must not talk with anyone else or exchange information with them during this exam.
- You must NOT use email, chat or the like during this exam. Close all such applications before you start the exam.

Your exam consist of Backend (server) code and Frontend (client side) code.

	Points
#1. Backend API methods	**___ / 50**
#2. Frontend HTML and JS	**___ / 50**
Total for this question	**___ / 100**

For both the frontend and backend code, you don't need to worry about weird edge cases (i.e. bad / evil users). You can assume all parameters passed in to the server are valid inputs (i.e. strings, numbers, etc within the correct range). Don't worry about defensive programming from would be attackers. The exam is only focused on the good users.

Setup

No given code for this problem. We'll just have your create folders and files using these instructions.

Create a folder called **Exam3**. Within that folder make two sub folders, one named **data**, the other named **webServer** (see the bulleted list view below of the folder structure). Data will have only 1 file **db.json**. Make the contents of that file be an empty **object** { }. So that file literally only has 2 characters in it. You should load the contents of that file into a local variable just like we did in prior Node web apps and you should save data to the file as well.

In the **webServer** folder copy in a blank frontend template (a **public** folder), and go into the index.html file and remove the firebase client scripts (to avoid Firebase errors). You will make the backend code from scratch (perhaps using prior code as a starting point) naming the main backend file **app.js** (I like that name better than integratedWebserver.js). Note: the app.js file should be in the **webServer** folder, but NOT in the **public** folder. app.js will contain your backend API methods and it will use express to serve your public folder (copy in a lot of your prior work from the Full Stack video or Hangman HW). app.js will use express, so you should do an `npm init` (all defaults are fine) and an `npm install express` command in your **webServer** folder.
Your folder structure should now look like this:

- **Exam3**
  - **data**
    - db.json
  - **webServer**
    - **node_modules**
    - **public**
    - app.js
    - package.json

Instead of serving the public folder to `/static` just serve the public folder to the naked domain. i.e.

```
app.use('/', express.static("public"));
```

Use port 3000 as was done in the follow alongs. Then start your webserver using the command: `nodemon app.js`. You should be able to visit the url: http://localhost:3000/ to see your frontend. We will use that url when we grade your work, so check it now. Example app.js

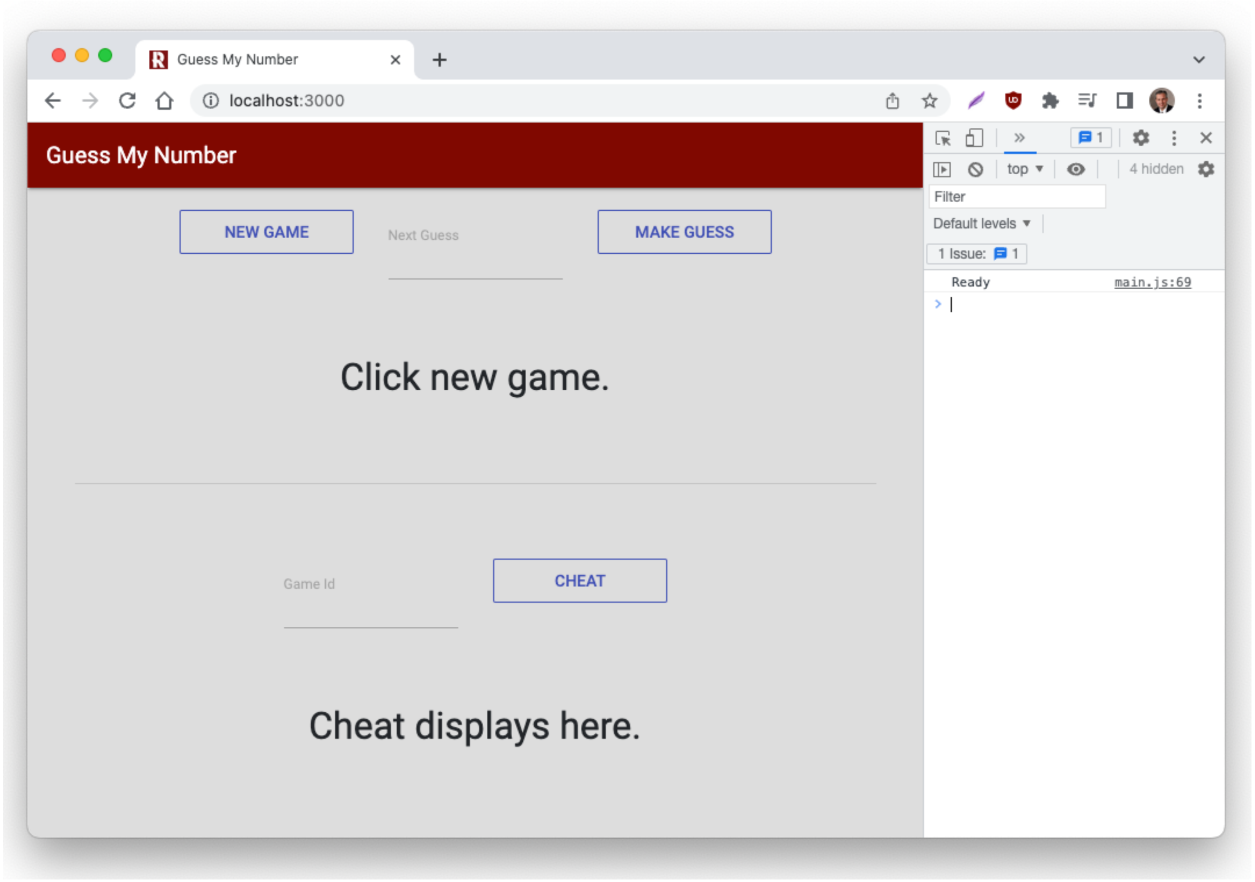
At the end of the exam you will create a .zip of the entire **Exam3** folder, submit it to Moodle, and complete a questionnaire in Gradescope. We don't actually want your node_modules folder, so feel free to delete it before making your .zip (optional).

Overview

This exam is called Guess My Number. There is no required format for the HTML "look", but there are some functionality requirements. Your game must be able to:

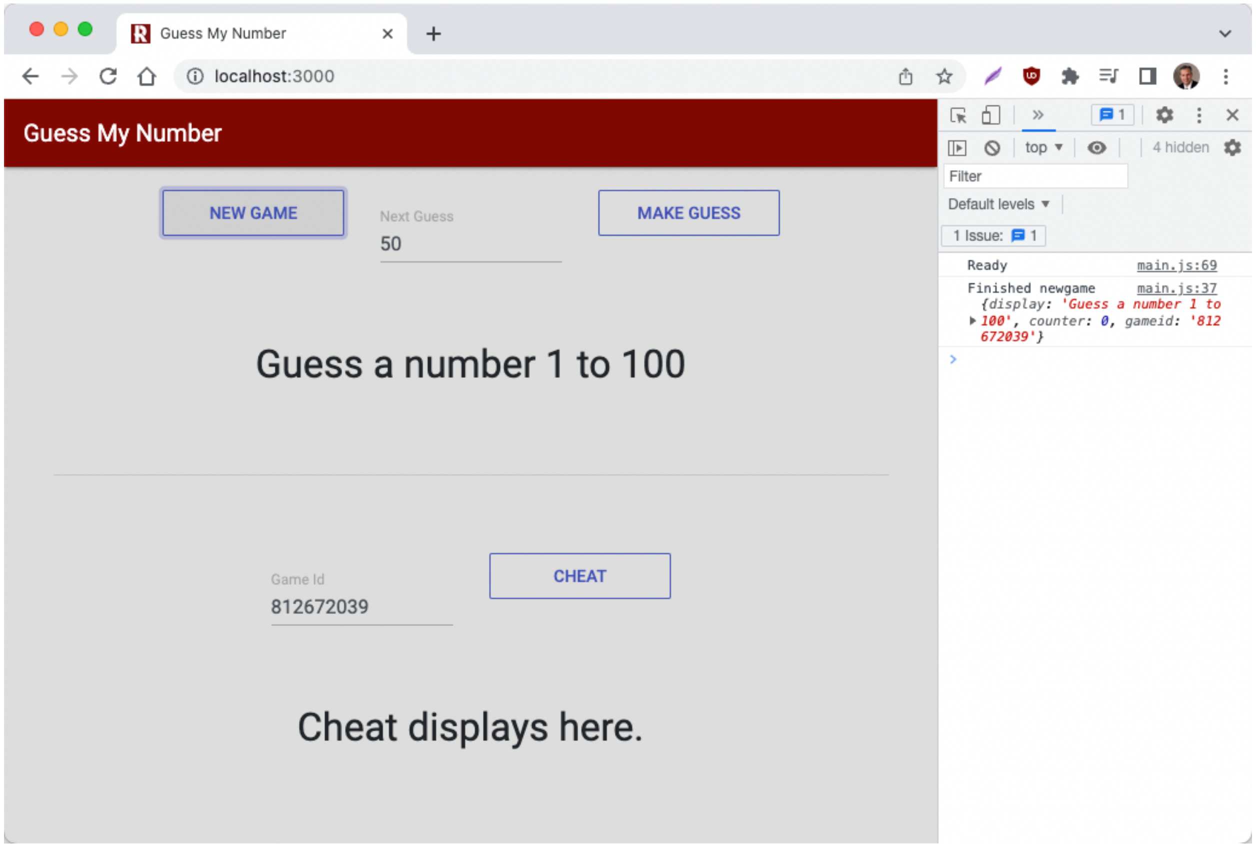
1. Make a new game
2. Guess a number
3. Cheat! (i.e. display the answer by asking the server to cheat for quick testing)

In order to make a guess you will also need an input. To help you quickly get the idea, your client web app might look like this initially:



```
Ready                         main.js:69
```

In Guess My Number, you make a new game via a POST message to the server. The server picks the random secret number within the range requested and sends a response to the client to display. For example, clicking new game might look like this:



Guess My Number

Next Guess
50

Guess a number 1 to 100

Game Id
812672039

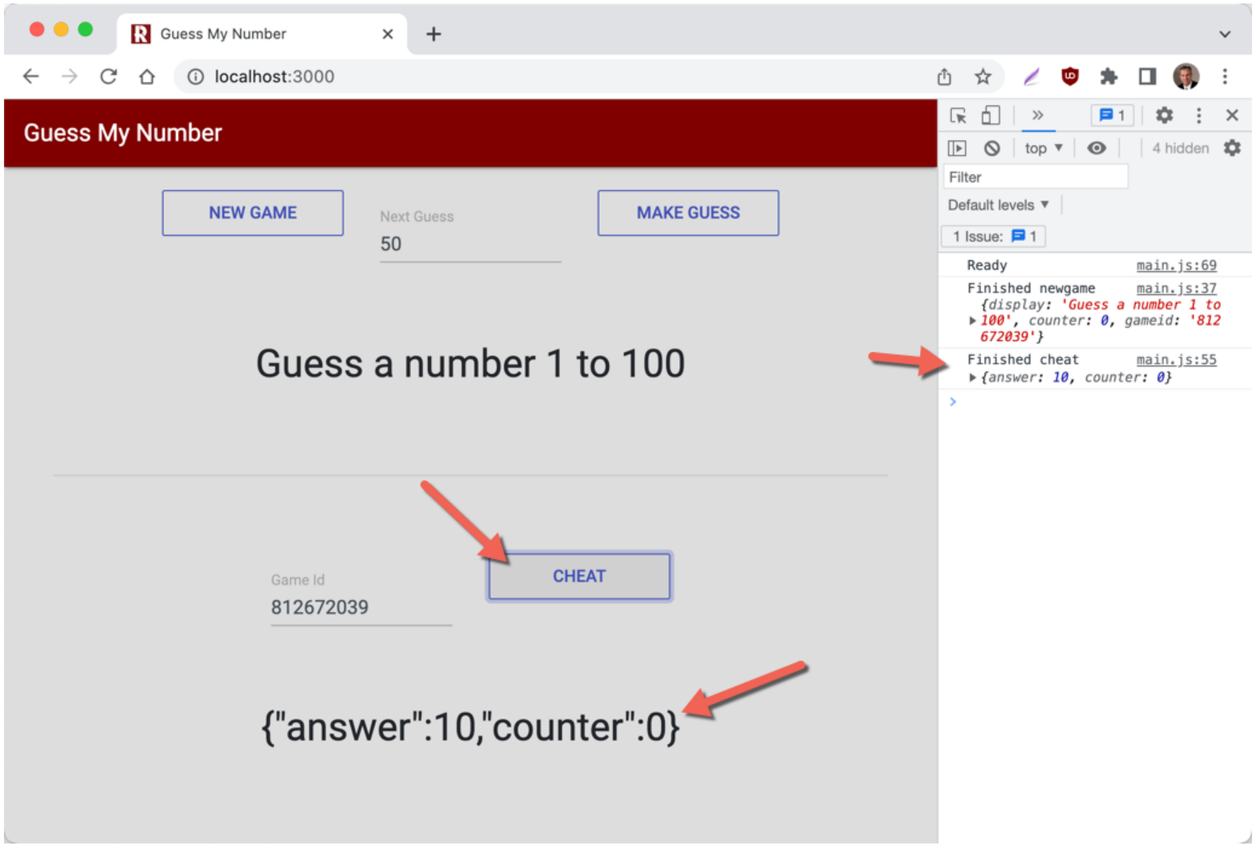
Cheat displays here.

```
Ready                      main.js:69
Finished newgame           main.js:37
  {display: 'Guess a number 1 to
▸ 100', counter: 0, gameid: '812
  672039'}
```

The request made to the server here was a POST with no POST body, which was sent to: http://localhost:3000/api/newgame/100 (the 100 is the max value for this game). Notice that whenever a server message is received it is always console logged (right side). Additionally, the `display` field of that response is displayed to the user on the page (`Guess a number 1 to 100`). Here the server made a new game, saved the new game to the **db.json** file and sent a response to the client. The server picked a random Game Id (`812672039` in this case), a random secret number, called **answer** (which is NOT sent to the client), and a counter set to 0. If this were the first ever game played, then the **db.json** file would look like this:

```
{
   "812672039": {
       "answer": 10,
       "counter": 0
   }
}
```

The server keeps a JSON object where the game Id is the key (`812672039` in this case). A game consists of a secret number called answer and a counter, which is initially 0. The client can see this information if they click the CHEAT button.



Guess My Number

Next Guess: 50

Guess a number 1 to 100

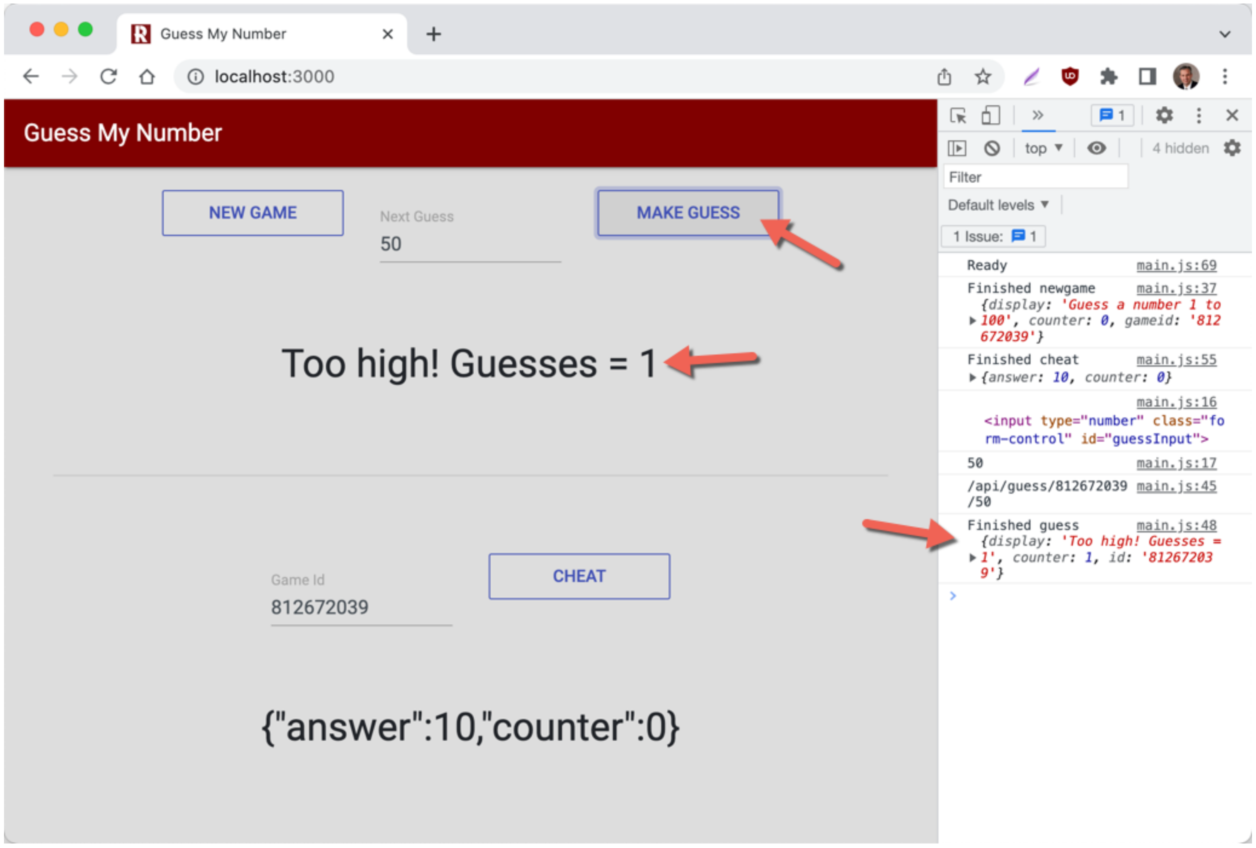
Game Id: 812672039

```
{"answer":10,"counter":0}
```

```
Ready                                        main.js:69
Finished newgame                             main.js:37
  {display: 'Guess a number 1 to 100', counter: 0, gameid: '812672039'}
Finished cheat                               main.js:55
  {answer: 10, counter: 0}
```

The cheat button sends a GET message to the server to get the game, in this case a GET request to http://localhost:3000/api/cheat/812672039   Again you can see the response from server is console logged and in this case the response is also shown to the user. Obviously, cheating is only for development, but it makes testing much easier.

To actually play the game correctly (the area above the line), a user needs to guess. Since this game was made for 1 to 100 (note, the server can do any max value since newgame receives a parameter for the max, but this web client always does 1 to 100), the user guesses 50….



Next Guess
50

Too high! Guesses = 1

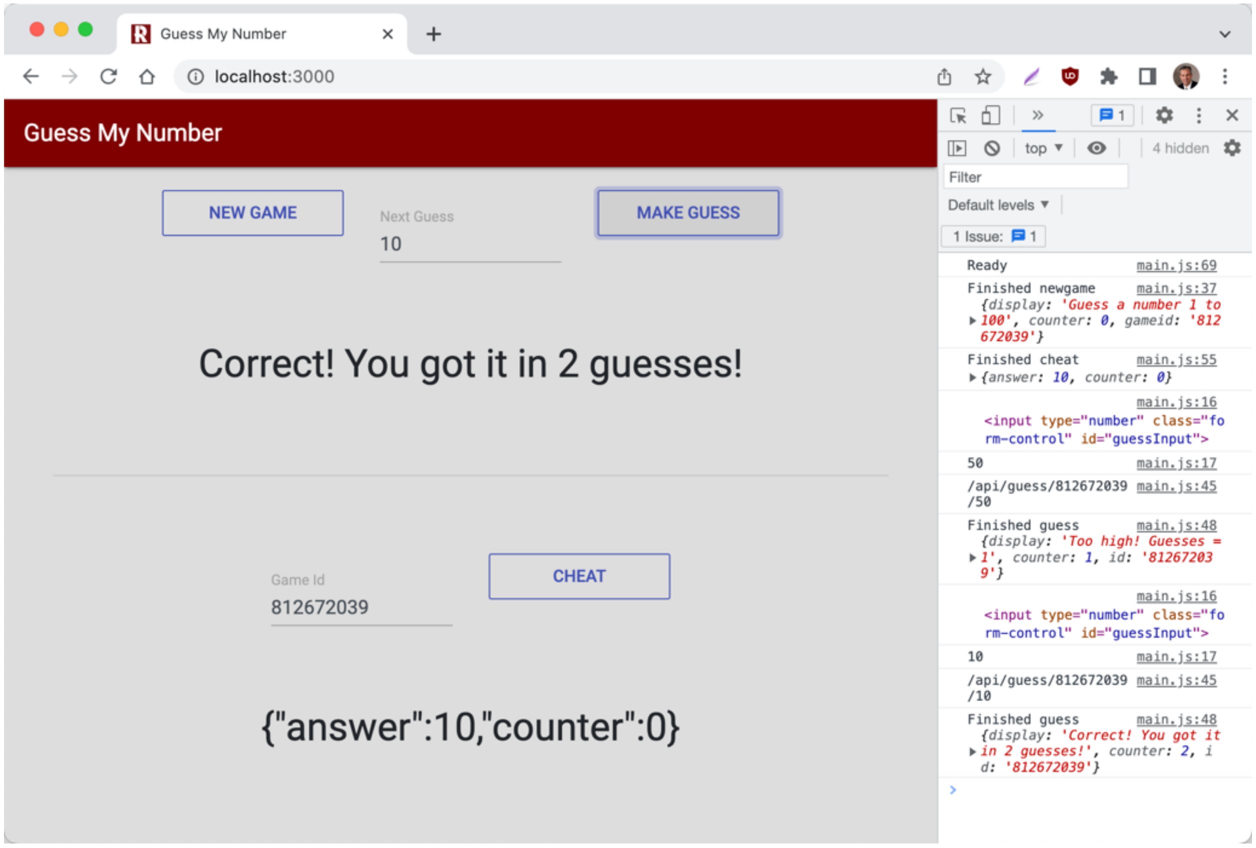
Game Id
812672039

{"answer":10,"counter":0}

```
Ready                     main.js:69
Finished newgame          main.js:37
  {display: 'Guess a number 1 to
▸ 100', counter: 0, gameid: '812
  672039'}
Finished cheat            main.js:55
▸ {answer: 10, counter: 0}
                          main.js:16
  <input type="number" class="fo
  rm-control" id="guessInput">
50                        main.js:17
/api/guess/812672039      main.js:45
/50
Finished guess            main.js:48
  {display: 'Too high! Guesses =
▸ 1', counter: 1, id: '81267203
  9'}
```

Again you can see that the response from the server is always console logged (plus a few other logs I need to remove). The resonse from the server includes the text to display which says both “Too low!” or “Too high!” plus the number of guesses. The web client has no idea the number of guesses, it just displays the display field of the server response. The only data the web client needs to save (to a simple variable) is the game id, which is still 812672039.

Obviously a user could continue to guess, getting too high or too low feedback until they are correct, but since we cheated earlier, I can just guess 10 now…

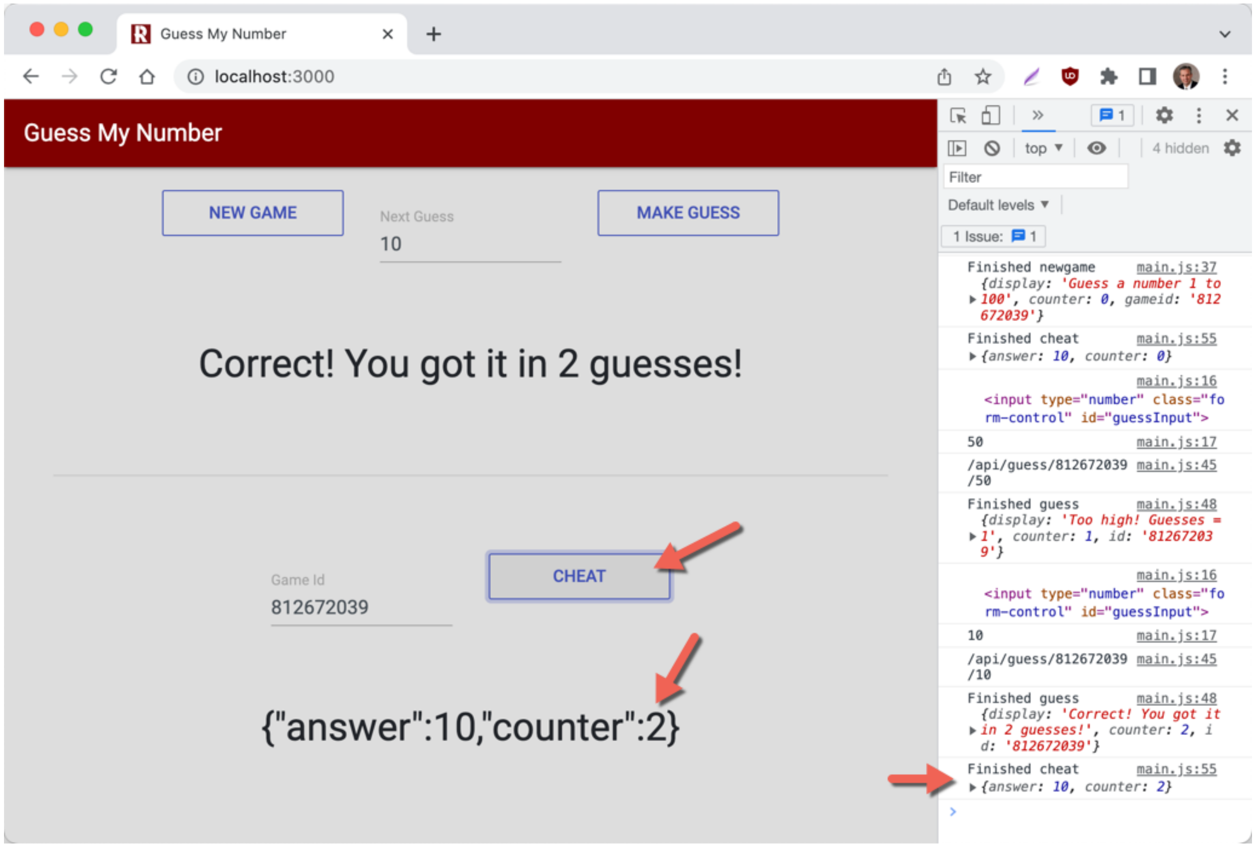


Correct! You got it in 2 guesses!

{"answer":10,"counter":0}

```
Ready                                  main.js:69
Finished newgame                       main.js:37
  {display: 'Guess a number 1 to
▸ 100', counter: 0, gameid: '812
  672039'}
Finished cheat                         main.js:55
▸ {answer: 10, counter: 0}
                                       main.js:16
  <input type="number" class="fo
  rm-control" id="guessInput">
50                                     main.js:17
/api/guess/812672039                   main.js:45
/50
Finished guess                         main.js:48
  {display: 'Too high! Guesses =
▸ 1', counter: 1, id: '81267203
  9'}
                                       main.js:16
  <input type="number" class="fo
  rm-control" id="guessInput">
10                                     main.js:17
/api/guess/812672039                   main.js:45
/10
Finished guess                         main.js:48
  {display: 'Correct! You got it
▸ in 2 guesses!', counter: 2, i
  d: '812672039'}
```

Notice the console log, and the display that is shown. Also observe that the cheat message doesn't magically update. If you wanted to update the cheat text, you'd need to click the CHEAT button again, which would now say this….



Correct! You got it in 2 guesses!

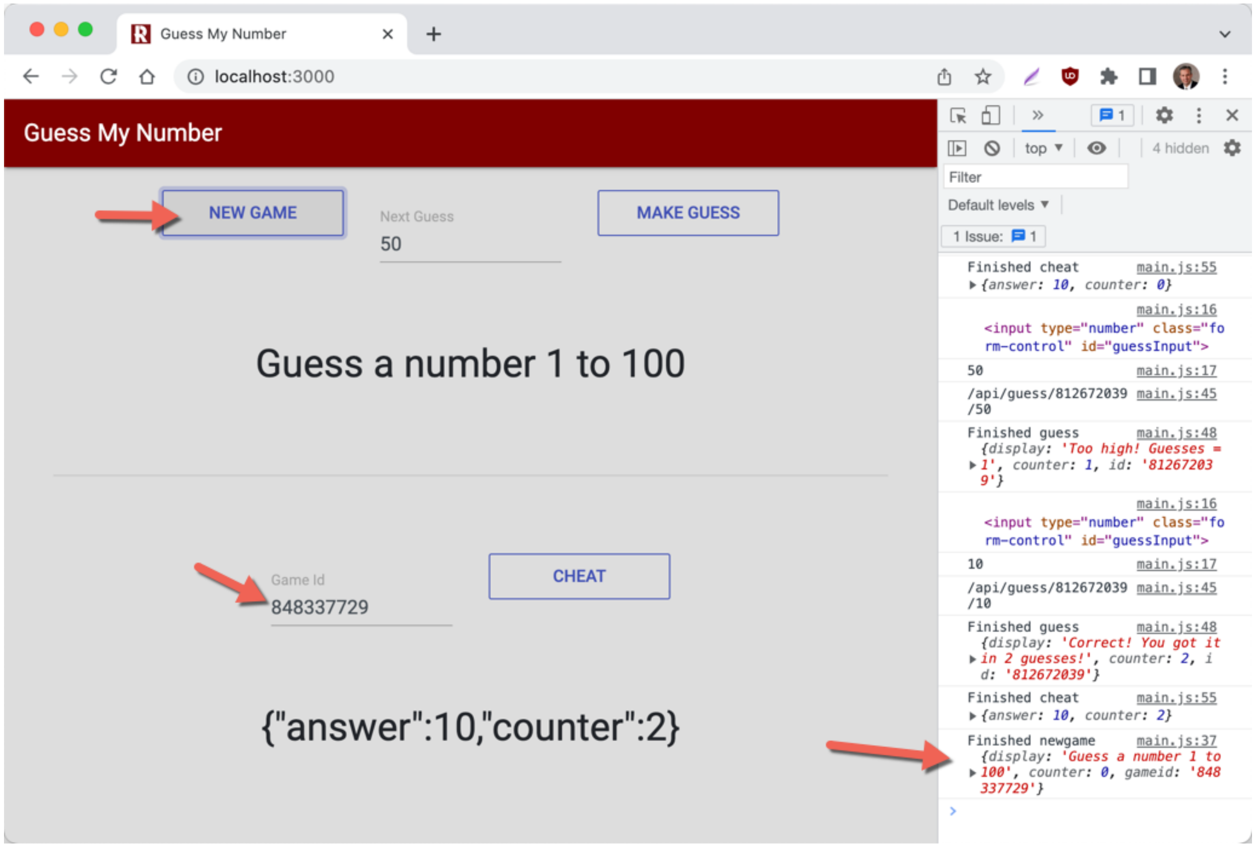
{"answer":10,"counter":2}

```
Finished newgame          main.js:37
  {display: 'Guess a number 1 to
▸ 100', counter: 0, gameid: '812
  672039'}
Finished cheat            main.js:55
▸ {answer: 10, counter: 0}
                          main.js:16
  <input type="number" class="fo
  rm-control" id="guessInput">
50                        main.js:17
/api/guess/812672039 main.js:45
/50
Finished guess            main.js:48
  {display: 'Too high! Guesses =
▸ 1', counter: 1, id: '81267203
  9'}
                          main.js:16
  <input type="number" class="fo
  rm-control" id="guessInput">
10                        main.js:17
/api/guess/812672039 main.js:45
/10
Finished guess            main.js:48
  {display: 'Correct! You got it
▸ in 2 guesses!', counter: 2, i
  d: '812672039'}
Finished cheat            main.js:55
▸ {answer: 10, counter: 2}
```

And no surprise if you were to go look at the server's db.json file, it would look like this:

```json
{
   "812672039": {
       "answer": 10,
       "counter": 2
   }
}
```

Interestingly nothing really happens on the server (backend) or in the client (frontend) when the user gets the number correct. It just displayed the correct message, but nothing is saved to indicate the user got it. A better client web app would disable the GUESS button, but that is not a requirement in this web app. At this point a good user would just click the NEW GAME button. If that happen it might look like this:



```
Finished cheat                   main.js:55
▸{answer: 10, counter: 0}
                                 main.js:16
  <input type="number" class="form-control" id="guessInput">
50                               main.js:17
/api/guess/812672039/50          main.js:45
Finished guess                   main.js:48
▸{display: 'Too high! Guesses = 1', counter: 1, id: '812672039'}
                                 main.js:16
  <input type="number" class="form-control" id="guessInput">
10                               main.js:17
/api/guess/812672039/10          main.js:45
Finished guess                   main.js:48
▸{display: 'Correct! You got it in 2 guesses!', counter: 2, id: '812672039'}
Finished cheat                   main.js:55
▸{answer: 10, counter: 2}
Finished newgame                 main.js:37
▸{display: 'Guess a number 1 to 100', counter: 0, gameid: '848337729'}
```

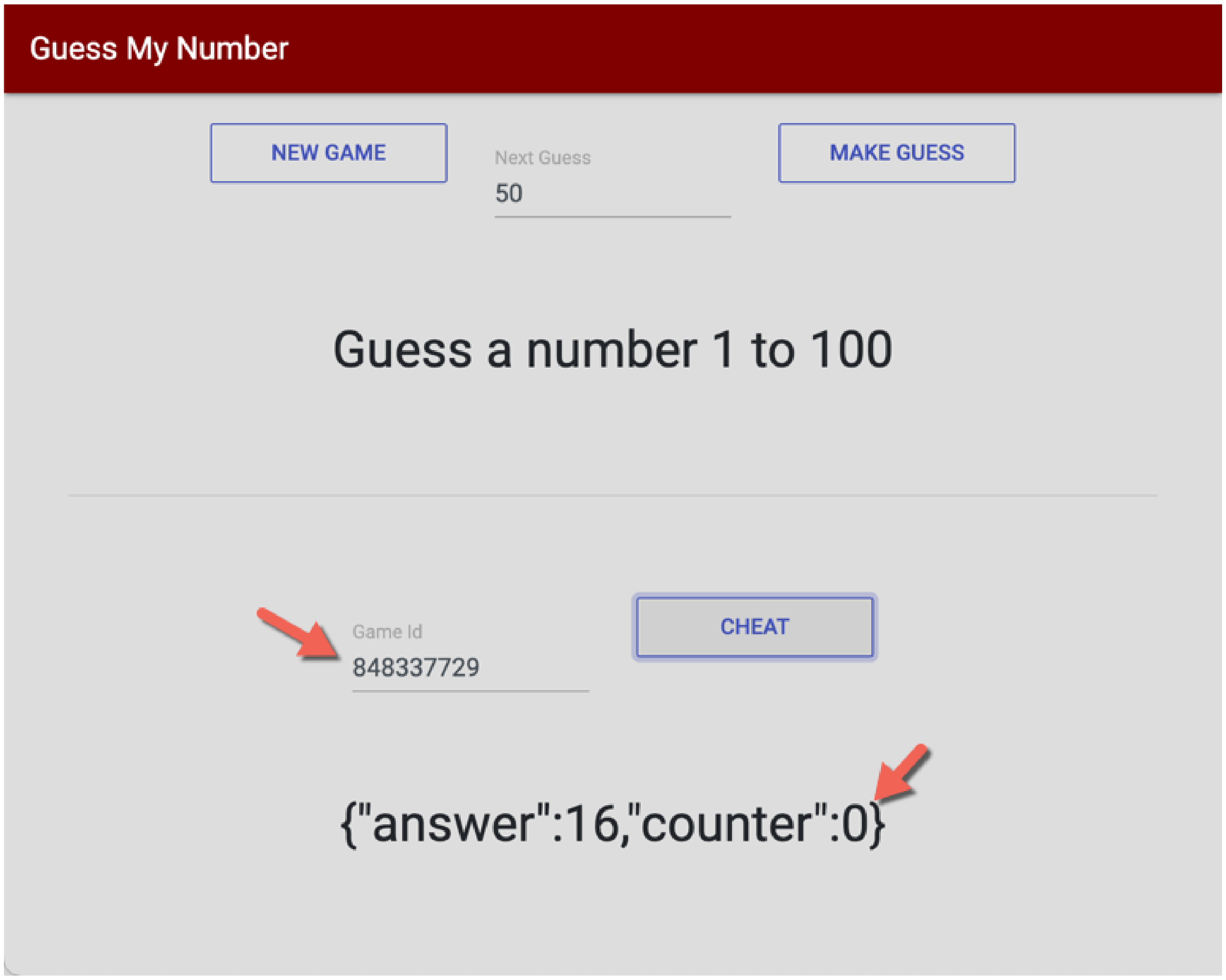
Again the server made a new game and sent a display string. The randomly choosen game id this time was 848337729 and the secret number isn't sent to the client. To get the secret number you could use the CHEAT button, or just look at the db.json file, which would now look like this:

```
{
   "812672039": {
       "answer": 10,
       "counter": 2
   },
   "848337729": {
       "answer": 16,
       "counter": 0
   }
}
```

Observe that the game Id is really a string, which works better in JSON files. I will give you a function for that. The secret number (called answer) was 16 this time, I will give you a function for that too. Here are those functions that you can put into your app.js file:

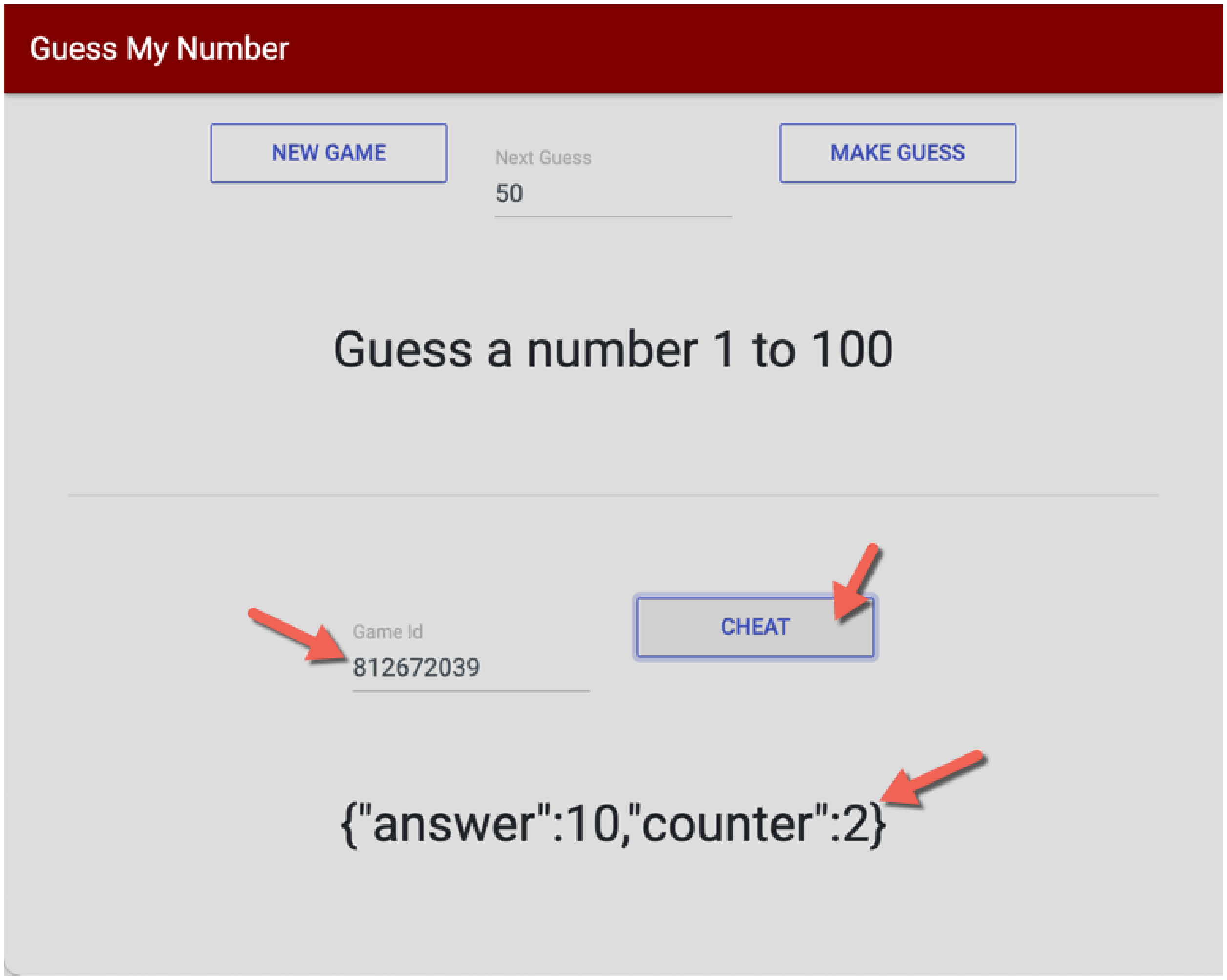
```
function getRandomNumber(maxValue) {
   return Math.ceil(Math.random() * maxValue);
}

function getRandomGameId() {
   return Math.floor(Math.random() * 1000000000).toString();
}
```

You don't need to worry about randomly getting a duplicate game id. It's unilily enough for us to ignore that possibiliity for this exam. In the image of the web client above you'll notice that in my web client the game id is displayed in the cheat area as an input. That is NOT a requirement, just something I did.   That way I can cheat for this game….



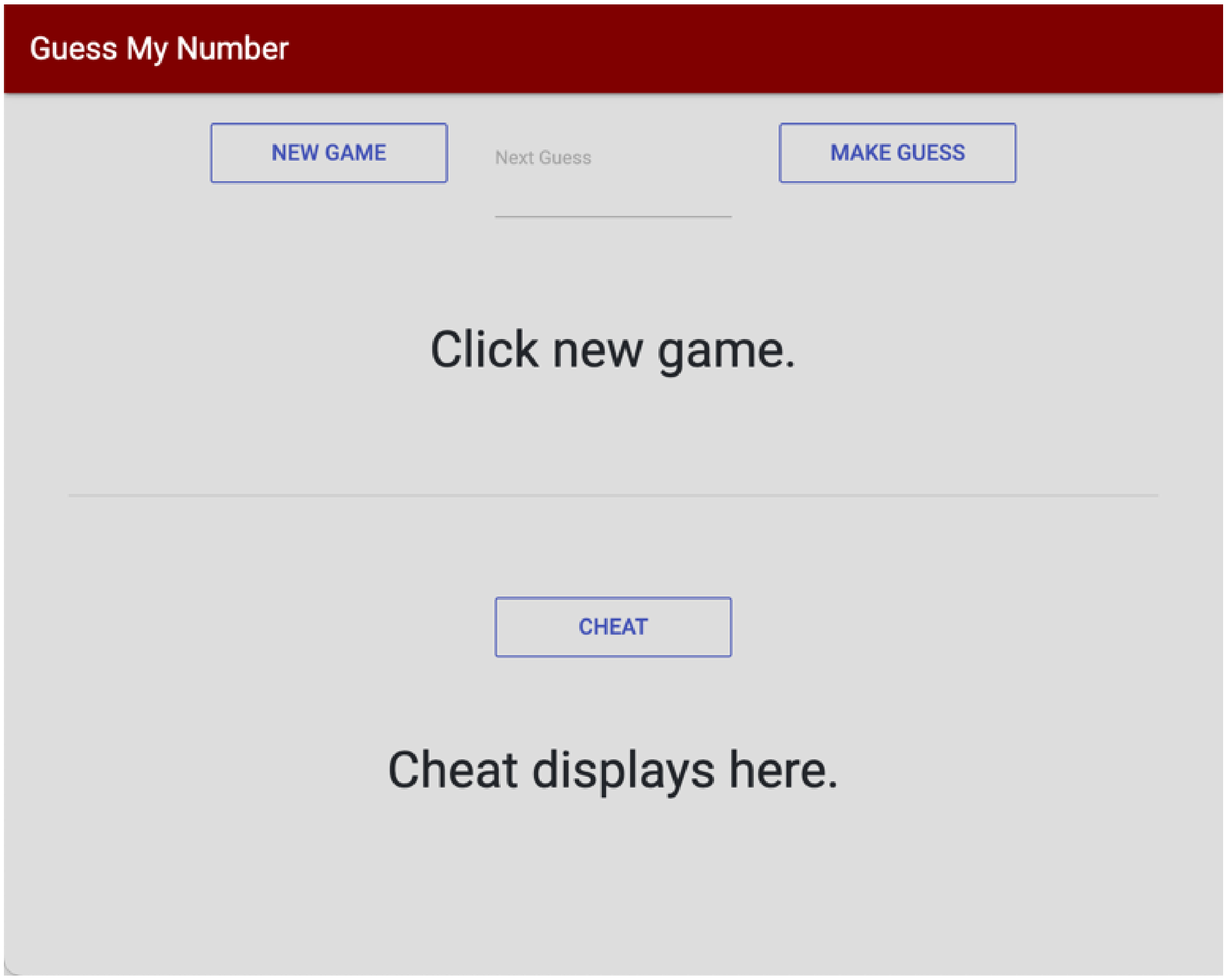
{"answer":16,"counter":0}

or I could change the input an cheat for a different game, or example the first game I played in this example…



```
{"answer":10,"counter":2}
```

Being able to CHEAT for the current game IS a requirement, but being able to type in a game id and cheat for any old game is NOT a requirement. So your web client could look like this…



Again, the "look" is not graded at all, just the functionality. For the functionality, you need in the top real game area, a way to make a NEW GAME, input a guess number, a button to GUESS, and a place to display the display text. Then in cheat area, you need a CHEAT button and a place to display the JSON of the server response (Hint: I used JSON.stringify() to print that JSON data).

Again, the **look** of this UI is not important at all (you don't even have to use Bootstrap) and worth no points, just the functionality.

Backend API Overview

You will make a total of 3 backend API methods in this exam.

1. cheat → Receives a game id, responds with the answer and counter (GET)
2. guess → Receives a game id and a guess, responds with a display message (PUT)
    a. Increments the counter for the game
3. newgame → Receives a max value, responds with a display message (POST)
    a. Creates a new game with an answer and a counter of 0 (see format from earlier)

API Summary

Name	method	path	Parameters	Response
Cheat	GET	/api/cheat/:gameid	gameid must be an existing game	
Guess	PUT	/api/guess/:gameid/:currentguess	gameid must be an existing game, currentguess is 1 to maxvalue	`{display: “Hi” counter: 42 gameid: “123456”}`
New Game	POST	/api/newgame/:maxvalue	maxvalue is an integer greater than 1, note there is no POST body	``{display: `Hi`, counter: 0, gameid: “123456”}``

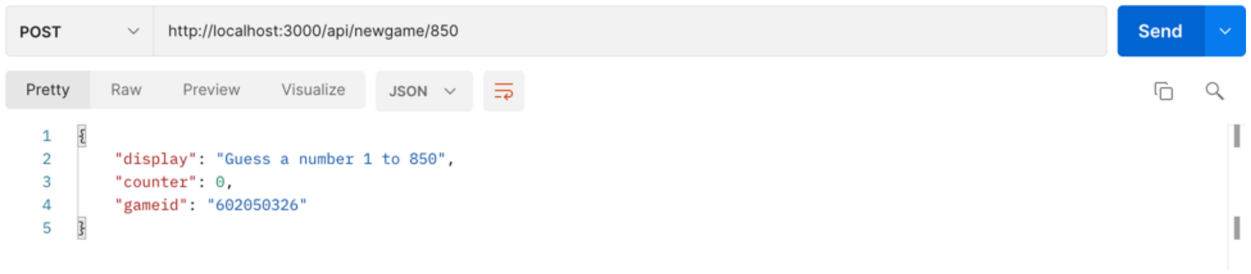
The data will be stored on the server using simple file storage into db.json. All commands should save such that you can freely stop and restart your backend without losing any data.

You should implement the backend API first. Test it with Postman, then create the frontend after the backend API is complete.

Question #1. Backend API

For each API call we've included a screenshot of what Postman might look like once your code works.

newgame (POST)

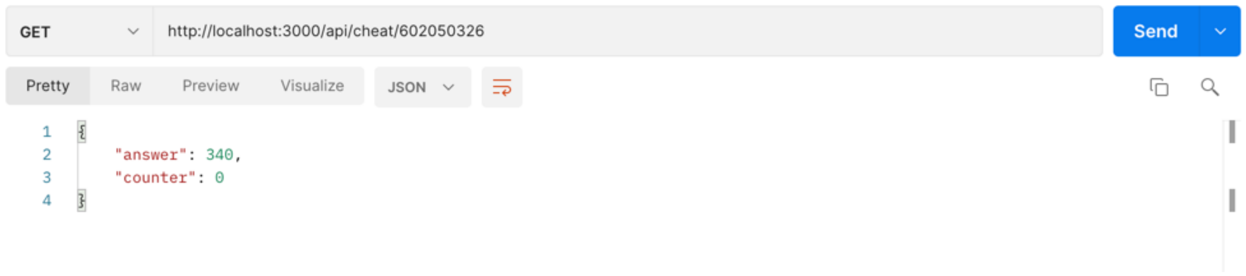


POST http://localhost:3000/api/newgame/850

```
{
    "display": "Guess a number 1 to 850",
    "counter": 0,
    "gameid": "602050326"
}
```

This game, just to prove that I could has a max value of 850. Notice the game id is 602050326.

cheat (GET)

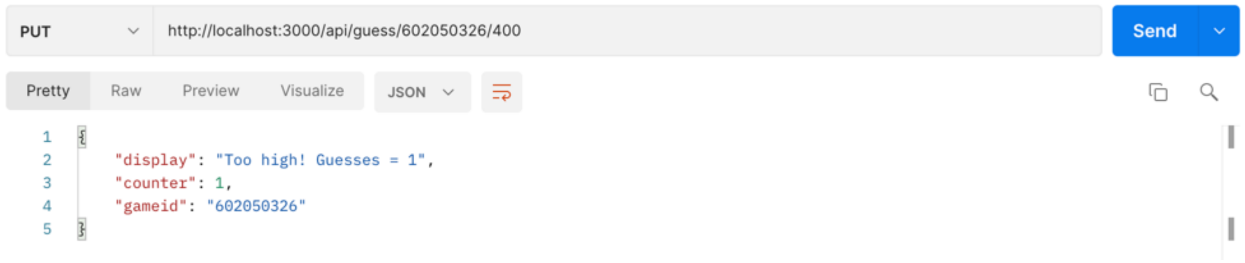


GET http://localhost:3000/api/cheat/602050326

```
{
    "answer": 340,
    "counter": 0
}
```

Looks like the answer is 340. Let's guess some.

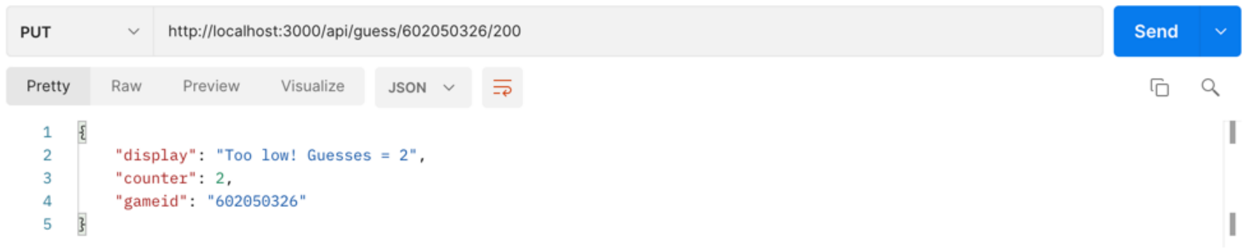
guess (PUT)
First guess of 400 since the game is 1 to 850.



PUT http://localhost:3000/api/guess/602050326/400

```
{
    "display": "Too high! Guesses = 1",
    "counter": 1,
    "gameid": "602050326"
}
```

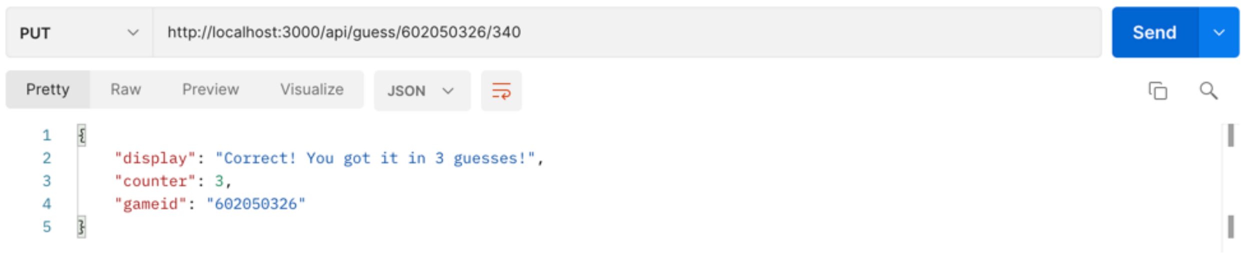
Next guess of 200



PUT http://localhost:3000/api/guess/602050326/200

```
{
    "display": "Too low! Guesses = 2",
    "counter": 2,
    "gameid": "602050326"
}
```

Then let's guess, oh say, 340



```
{
    "display": "Correct! You got it in 3 guesses!",
    "counter": 3,
    "gameid": "602050326"
}
```

Got it! What an amazing guesser!

No POSTMAN file for you as they are pretty easy to make.

- (as a POST) http://localhost:3000/api/newgame/850
- (as a GET) http://localhost:3000/api/cheat/602050326
- (as a PUT) http://localhost:3000/api/guess/602050326/340

Once you have all 3 backend API methods complete you are finished with question #1 and ready to move on to question #2. You will submit the whole exam together at the very end. We will test your backend API independently of your frontend, so make sure your API methods follow the above specifications exactly. You should try stopping and then restarting your server. Since all values are saved to db.json, it should pick up exactly where it left off. A stop and restart should have no effect on the data.

Question #2. Frontend

The frontend code was described before. Saying a few of the important things again. Your web client need only play with a maxvalue of 100 (i.e. guess 1 to 100). You can hardcode that in your main.js file. Your main.js code needs to save the game id when a new game is made and use that value when doing a guess or cheat. If a user makes a guess BEFORE hitting new game it can crash, that's fine, we are only writing code for good users. You need to have buttons for new game, guess (plus an input to guess), and cheat. Additionally you need to display the `display` field from the server for a newgame or guess request and the `JSON.stringify()` of the json data from a cheat request.

Good luck completing your web app!

Submitting your work

When you complete your work (or time runs out) you will submit 1 .zip file to Moodle. That .zip file should be your Exam3 folder, which contains all of your work. Note: we don't really need your **node_modules** folder, so you can delete that folder before you make your .zip. We will download your .zip, extract all, run `npm install` from the webServer folder (or have some other trick to get **express** on our grading computer), then run **`nodemon app.js`** to see your work. We will test your backend first, then explore your frontend by visiting: http://localhost:3000/

In addition to submitting your code, please complete the Exam 3 questionnaire on Gradescope.