Lab 10: *REST API*   **Backend**

REST API - I

Number Guessing Game
Express JS

IMPORTANT NOTE: Install Node JS

Table of Content: *Implementing a Simple (HTTP) Web Server*

Related Labs:

- Number Guessing Game - II (Frontend)
- Blog Comments (Frontend)
- Web Server - II (Backend)

Lab Introduction

Prerequisites
None. This is the first lab in the REST API sequence. Assumes knowledge of Node, Express, and JSON.

Motivation
Use Express to implement a REST API

Goal
Design & Implement a backend REST API for a Number Guessing Game

Learning Objectives
- Express
- Route Parameters & variable URLs
- Request's Query String
- UUID & shortid
- REST API (Design & Develop)
- Backend app logic (Design & Develop)
- Export your own JavaScript modules

Server-side Architecture:
Start this project by making a project folder where all your assets & scripts will be organized. Create all necessary files and folders as illustrated below.
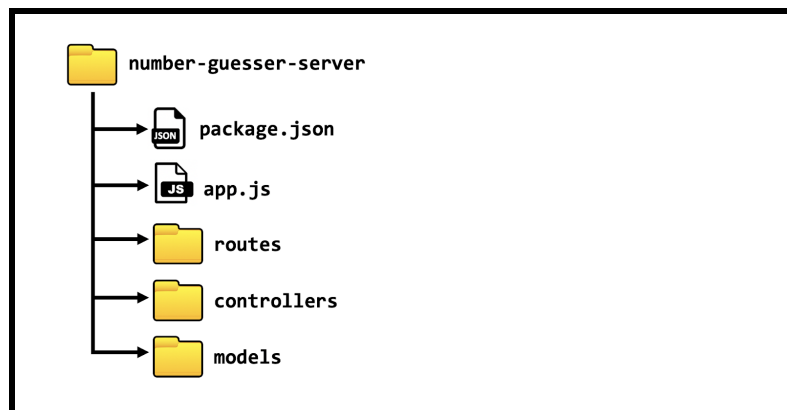
Concepts

REST Architecture

REST architecture is a common approach for backend services to interface with other backend or frontend apps.

- **API:** a collection of pre-implemented functions or methods a developer can import or invoke

- **REST:** REpresentational State Transfer. Refers to using the stateless HTTP protocol to provide access to an API

- **Endpoints:** When an API interacts with another system, the touchpoints of this communication are considered endpoints. For APIs, an endpoint can include a URL of a server or serviceI

Express JS

Express is a full web framework and server. http://expressjs.com/

- **Route Parameters:** Express allows variable URL routes to be defined with a callback

- **Request's Query:** The query string is located in the Request objet's query attribute

Unique IDs

Backend services usually must manage multiple unique session data for each of its clients. In order to track which data belongs to each client, a unique identifier is mapped to the data.

- **UUID:** the most common & popular Node module for generating unique IDs

- **shortid:** Node module that generates unique IDs that are human readable, intended for sharing

Iteration 0: Design REST API

'Approach' → Plan phase

Goal #1:   Specify the functions & routes the API will support

Approach: Pencil & paper

Before coding a REST API, you should first plan out the number of endpoints you intend to use and how those endpoints interface with controllers to interface with your backend models.

'Apply' → Do phase

Design Steps

Step 1: Write a table that specifies your API details

This table will serve as a guide on future iterations for implementing your backend service.

REST API: Number Guessing Game

Route	Method	Data	Description
/api/game/new	GET	start, end	**New Game:** Creates a new game and randomizes a number between the start & end values
/api/game/:id	GET		**Join Game:** Gets the data for an existing game using that game's id
/api/game/:id/guess	GET	guess	**Make Guess:** Guess a number in a game, returns whether the guess is 'correct', 'too low', 'too high', or 'gameover'
/api/game/:id/reset	GET		**Reset Game**: Resets a game with a new random number and toggles the game over variable to false

'Assess' → Test phase

Once you have a clear and full understanding of the API's specifications, it's time to start incrementally building the project!

Iteration 1: Setup Configuration File *(package.json)*

'Approach' → Plan phase

Goal #1:   Initialize configuration file: package.json

Approach: **NPM, package.json**

NPM uses a configuration file named `package.json` that installs all of the apps' dependencies and defines a launch command to start up the app.

'Apply' → Do phase

Setup Steps

Step 1 (JSON): package.json → Create configuration file

`package.json` contains the metadata for managing, building, & launching your Node app.

package.json

```json
{
    "name": "number-guesser-api",
    "version": "1.0.0",
    "description": "Backend API for Number Guessing Game",
    "author": "Your Name",
    "main": "app.js",
    "scripts": {
        "start": "npm install && node app"
    },
    "dependencies": {
        "express": "*"
    }
}
```

Step 2 (BASH): Use NPM to install dependencies

NPM uses the package.json file to fetch and download the app's dependencies with the install command.

```bash
npm install
```

'Assess' → Test phase

All the app's dependencies should be downloaded into a `node_modules` folder.

Iteration 2:   Setup Express App

'Approach' → Plan phase

Goal #2: Initialize an Express application to listen on a port

Approach: **Use Express to Listen on a Port**
Express has many 'middleware modules' that make building backend services much easier. Must import modules and use them in the app to listen for requests. This will make it easy to set up a web server.

'Apply' → Do phase

JavaScript Steps
- **Step 1**: app.js → import modules and setup data
- **Step 2**: app.js → use body parser (middleware)
- **Step 3**: app.js → listen for requests on port

Step 1 (JS): app.js → Import modules and setup data
Import the express module and instantiate it into a variable. Setup a port number.

app.js

```
const express = require('express');                //import module: express
const path = require('path');                      //import module: path
const bodyParser = require('body-parser');         //import module: body parser
const port = process.env.PORT || 3000;             //initialize data for port
```

Step 2 (JS): app.js → Use body parser
Express app uses body parser to read Request object's JSON data & URL encoded data from route

app.js

```
const app = express();                             //express fxn to create app
app.use( bodyParser.json() );                      //use body parser to read json
app.use( bodyParser.urlencoded({extended:false}) ) //use body parser to read url
```

Step 3 (JS): app.js → Listen for requests on port

The express app listens to port for requests. Print message to console to show server launched properly.

app.js

```
app.listen(port);                                    //app listen on port
console.log(`Server is running on port ${port}...`);  //print message to console
```

'Assess' → Test phase

Launch the web server application from the bash terminal with node:

```
node app
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

Iteration 3: Router & Controller → Test

'Approach' → Plan phase

Goal #3: Define a Route & Controller to trigger the API to send test data in Response

Approach: Router & Controller JavaScript files

Create a routes file that defines all API routes. Create a controller file that defines all the callback functions triggered by the routes.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: game.controllers.js → Make new file in folder: controllers
- **Step 2**: game.routes.js → Make new file in folder: routes
- **Step 3**: app.js → Refactor: import & use routes

Step 1 (JS): game.controllers.js → Make new file in folder: controllers

Create a Controller class that has a test method. The test method sends JSON data as a response. Export an instance of Controller as a module so that other JavaScript files may import it.

controllers/game.controllers.js

```
class Controllers{                               //class: Controllers
   test(request, response){                      //instance method: test
     response.json({'success': true});           //send JSON data
  }
}

module.exports = new Controllers();              //export Controller instance as module
```

Step 2 (JS): game.routes.js → Make new file in folder: routes

Create an express router and add a '/test' route for a GET method that triggers the controller's test callback function. Import your controllers methods into this script to use as the callbacks. Export the router as a module so that other Javascript files may import it.

routes/game.routes.js

```js
var express = require('express');                      //import module: express
var router = express.Router();                         //initialize Router
const {test} = require('../controllers/game.controllers');   //import test function

router.get('/test', test);                             //endpoint for test()

module.exports = router;                               //export router as module
```

Step 3 (JS): app.js → Refactor: import & use routes

Refactor app to import routes and use them on the path '/api/game' . This code should be implemented before the app listens on the port.

app.js

```js
const routes = require('./routes/game.routes');        //export router as module
app.use('/api/game', routes)                           //app use routes on path: /api/game
```

'Assess' → Test phase

Launch the web server application from the bash terminal with command:
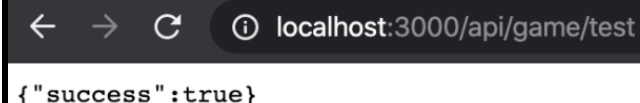
```
node app
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

Test: *(Success)*

Open browser to: `localhost:3000/api/game/test`

```
← → C  ⓘ localhost:3000/api/game/test

{"success":true}
```

Iteration 4: Endpoint → New Game

'Approach' → Plan phase

Goal #4: Create game model & implement REST API to make a new game

Approach: Game Model to manage Game Data

Create a model file that defines all number guessing game states and behaviors. Game model will create a dictionary of games, where key is a game id and value is a game object. Each game much track the min, max, random values, whether the game is over, and the game id.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: package.json → Refactor: Add to Dependencies
- **Step 2**: game.model.js → Make new file in folder: models
- **Step 3**: game.controllers.js → Refactor: import game model
- **Step 4**: game.controllers.js → Refactor Controller: add newGame()
- **Step 5**: game.routes.js → Refactor

Step 1 (JSON): package.json → Refactor: Add to Dependencies

`package.json` contains the metadata for managing, building, & launching your Node app.

package.json

```json
{
    "name": "number-guesser-api",
    "version": "1.0.0",
    "description": "Backend API for Number Guessing Game",
    "author": "Your Name",
    "main": "app.js",
    "scripts": {
        "start": "npm install && node app"
    },
    "dependencies": {
        "express": "*",
        "shortid": "*"
    }
}
```

Step 2 (JS): game.model.js → Make new file in folder: models

Create a Game class that manages all game data. Game model registers games with an id & adds a hashmap. Module shortid is used to generate game IDs. Create method adds a new game to hashmap, & requires two parameters(min, max).

models/game.model.js

```
const shortid = require('shortid');                            //import module: shortid

class Game{                                                    //class: Game
  constructor(){                                               //constructor: Game
    this.games = {};                                           //instance var: hashmap
  }                                                            //holds all game data

  create(start, end){                                          //method: create
    const number = ~~(Math.random() * end + start)            //generate random int
    const id = shortid.generate();                             //generate random id
    let game={'number':number,'start':start,'end':end,'gameover':false}; //add to hashmap with id
    this.games[id] = game;
    console.log(this.games);
    return id;
  }
}

module.exports = new Game();                                   //export Game instance as module
```

Step 3 (JS): game.controllers.js → Refactor: import game model

The controllers first line of code should import the game model into the script

controllers/game.controllers.js

```
const game = require('../models/game.model');                  //import module: game model
```

Step 4 (JS): game.controllers.js → Refactor Controller: add newGame()

Add a newGame method to the Controllers class. This method passes params to the game create method. It sends back the game id as JSON data.

controllers/game.controllers.js

```
newGame(request, response){
  const {start, end} = request.query;                          //destructure min, max from Query
  const id = game.create(start, end);                          //invoke game's create method
  response.json({'success': true , 'gameID': id});             //send JSON data
}
```

Step 5 (JS): game.routes.js → Refactor

Destructure the newGame method from the controllers module. Then setup router to trigger newGame method on a GET request to path '/new'

routes/game.routes.js

```javascript
var express = require('express');
var router = express.Router();
const {test, newGame} = require('../controllers/game.controllers');

router.get('/test', test);
router.get('/new', newGame);

module.exports = router;
```

'Assess' → Test phase

Launch the web server application from the bash terminal with command:

```
npm start
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

Test: *(Success)*

Open browser to: **localhost:3000/api/game/new?start=0&end=1000**

```
← → C    ⓘ localhost:3000/api/game/new?start=0&end=1000

{"success":true,"gameID":"PgShV00aE"}
```
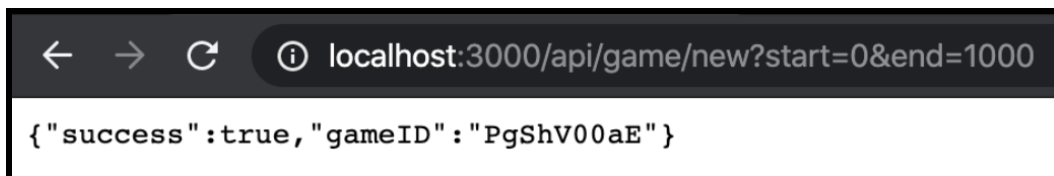
In the server console should display:

```
Server is running on port 3000...
{
  PgShV00aE: { number: 258, start: '0', end: '1000', gameover: false }
}
```

Note: id and number will be different as they are both generated random

Iteration 5: Endpoint → Get Game

'Approach' → Plan phase

Goal #5: Implement REST Endpoint to get data for a game

Approach: Use Model, Controller, Routes to implement REST Endpoint

In this iteration, must implement a REST endpoint to get game data given a game ID. The routes use parameters to allow variable url paths based on the unique game IDs

'Apply' → Do phase

JavaScript Steps
- **Step 1**: Add get method to Model
- **Step 2:** Add getGame method to Controller
- **Step 3**: Add endpoint to trigger getGame

Step 1 (JS): game.model.js → Refactor Game: add get()

In Game class, implement a 'get' method that takes an id parameter. Check id has data & destructure it to remove the secret number then return data. If id isn't in the hashmap then return null.

models/game.model.js

```
get(id){
    if (this.games[id]){                        //check ID has game data
        const {number, ...data} = this.games[id];  //Destructure game data except number
        return data;                            //return game data
    }
    else{                                       //if ID not in hashmap
        return null;                            //return null
    }
}
```

Step 2 (JS): game.controllers.js → Refactor Controller: add getGame()

Add a getGame method to the Controllers class. This method passes params to the game's get method. It sends back the game data as JSON data. If the game returns null, then send success: false response.

controllers/game.controllers.js

```js
getGame(request, response){
   const id = request.params.id;               //get id from request parameters
   const data = game.get(id)                   //invoke game's get method
   if (data){                                  //if data exists
      data["success"] = true;                  //add 'success' to data
      response.json(data);                     //send as JSON data
   }
   else{                                       //if data not exists
      response.json({"success": false})        //send JSON data with no success
   }
}
```

Step 3 (JS): game.routes.js → Refactor

Destructure the getGame method from the controllers module. Then setup router to trigger the getGame method on a GET request to path '/:id'. Note: the colon (:) specifies a route parameter.

routes/game.routes.js

```js
var express = require('express');
var router = express.Router();
const {getGame, newGame, test} = require('../controllers/game.controllers');

router.get('/test', test);
router.get('/new', newGame);
router.get('/:id', getGame);                   //define variable route with colon(:)
                                               //Note: variable routes go after defined routes
module.exports = router;
```

'Assess' → Test phase

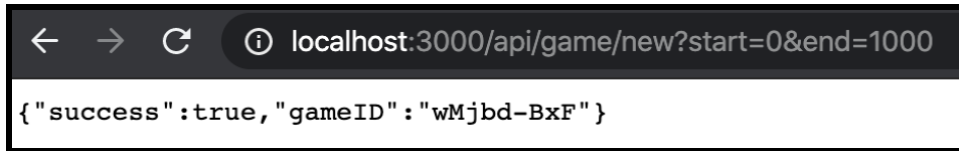Launch the web server application from the bash terminal with command:

```
npm start
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

Make a Game & Game ID

Open browser to: `localhost:3000/api/game/new?`**`start=0&end=1000`**

```
←  →  C    ⓘ localhost:3000/api/game/new?start=0&end=1000

{"success":true,"gameID":"wMjbd-BxF"}
```

Use this game id to test endpoints

Test: Success

Open browser to: `localhost:3000/api/game/`**`wMjbd-BxF`**   *Note: Game ID will vary*

```
←  →  C    ⓘ localhost:3000/api/game/wMjbd-BxF/

{"start":"0","end":"1000","gameover":false,"success":true}
```

Test: Failure

Open browser to: `localhost:3000/api/game/`**`not-valid-id`**

```
←  →  C    ⓘ localhost:3000/api/game/not-valid-id

{"success":false}
```
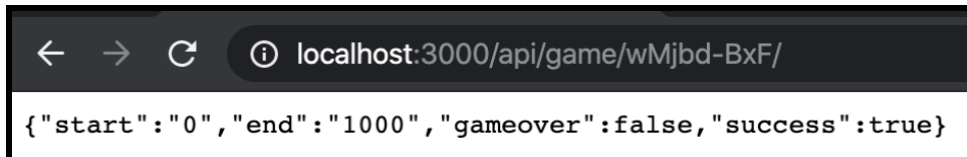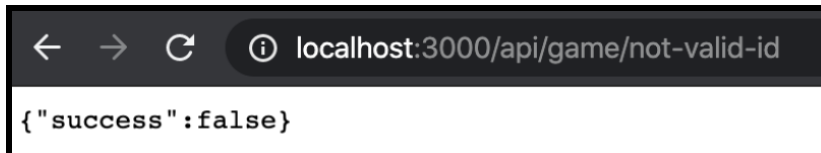
Iteration 6: Endpoint → Guess

'Approach' → Plan phase

Goal #6: Implement REST Endpoint to Guess the number

Approach: Use Model, Controller, Routes to implement REST Endpoint

In this iteration, implement a REST endpoint to guess the number. The routes use parameters to allow variable url paths based on the unique game IDs

'Apply' → Do phase

JavaScript Steps
- **Step 1**: Add guess method to Model
- **Step 2:** Add guess method to Controller
- **Step 3**: Add endpoint to trigger guess

Step 1 (JS): game.model.js → Refactor Game: add guess()

Add a 'guess' method to Game class. Check whether the game is over, or whether the guess is correct, too low, or too high Otherwise its an error.

models/game.model.js

```javascript
guess(id, guess){
    if (!this.games[id]){                               //if id has no data
        return null;                                    //   then return nothing
    }
    const game = this.games[id];                        //get game data from hashmap
    if (game.gameover == true){                         //if game is over
        return {'guess': 'gameover'}                    //   then it's game over
    }
    else if (game.number == guess){                     //if guess is number
        game.gameover = true;                           //   then set game as over
        return {'guess': 'correct'}                     //   then it's correct
    }
    else if (game.number > +guess){                     //if number is greater than guess
        return {'guess': 'too low'}                     //   then it's too low
    }
    else if (game.number < +guess){                     //if number is less than guess
        return {'guess': 'too high'}                    //   then it's too high
    }
    else{                                               //otherwise
        return {'guess': 'error'}                       //   then it's an error
    }
}
```

16

Step 2 (JS): game.controllers.js → Refactor Controller: add guess()

Add a guess method to the Controllers class. This method passes params to the game's guess method. It sends back the result as JSON data. If the game returns null, then send a success: false response.

controllers/game.controllers.js

```
guess(request, response){
   const id = request.params.id;                    //get id from request parameters
   const { guess } = request.query;                 //destructure guess from Query String
   const result = game.guess(id, guess);            //invoke game's guess method
   if (result){                                     //if result exists
      result['success'] = true;                     //add success to it
      response.json(result);                        //send as JSON data
   }
   else{                                            //if null
      response.json({'success': false})             //send 'no success' as JSON data
   }
}
```

Step 3 (JS): game.routes.js → Refactor:

Destructure the guess method from the controllers module. Then setup router to trigger the guess method on a GET request to path '/:id/guess'. Note: the colon (:) specifies a route parameter.

routes/game.routes.js

```
var express = require('express');
var router = express.Router();
const {guess, getGame, newGame, test} = require('../controllers/game.controllers');

router.get('/test', test);
router.get('/new', newGame);
router.get('/:id', getGame);
router.get('/:id/guess', guess);              //define variable route with colon(:)
                                              //Note: variable routes go after defined routes
module.exports = router;
```

'Assess' → Test phase

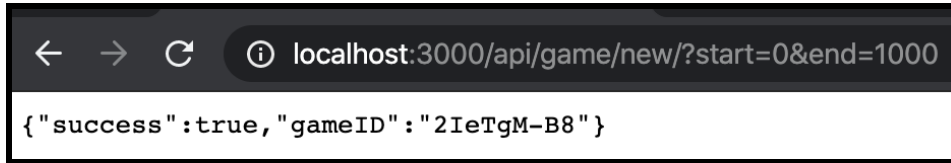Launch the web server application from the bash terminal with command:

```
npm start
```

You should get a display message from the terminal:

```
Server running on port 3000...
```
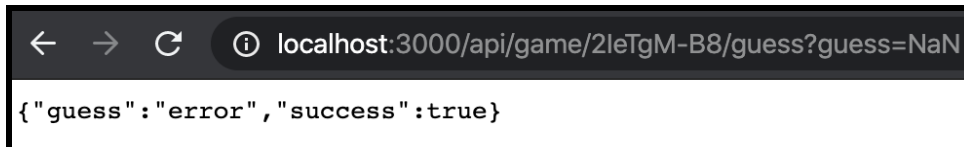
Make a Game & Game ID
Open browser to: **localhost:3000/api/game/new?start=0&end=1000**

```
←  →  C      ⓘ localhost:3000/api/game/new/?start=0&end=1000

{"success":true,"gameID":"2IeTgM-B8"}
```

Test: Error *(Success)*
Open browser to: **http://localhost:3000/api/game/2IeTgM-B8/guess?guess=NaN**

```
←  →  C      ⓘ localhost:3000/api/game/2IeTgM-B8/guess?guess=NaN

{"guess":"error","success":true}
```

Test: Too Low *(Success)*
Open browser to: **http://localhost:3000/api/game/2IeTgM-B8/guess?guess=0**

```
←  →  C      ⓘ localhost:3000/api/game/2IeTgM-B8/guess?guess=0

{"guess":"too low","success":true}
```

Test: Too High *(Success)*
Open browser to: **http://localhost:3000/api/game/2IeTgM-B8/guess?guess=1000**

```
←  →  C      ⓘ localhost:3000/api/game/2IeTgM-B8/guess?guess=1000

{"guess":"too high","success":true}
```

Test: Correct *(Success)*

Open browser to: `http://localhost:3000/api/game/`**`2IeTgM-B8`**`/guess?`**`guess=649`**

← → C ⓘ localhost:3000/api/game/2IeTgM-B8/guess?guess=649

```
{"guess":"correct","success":true}
```
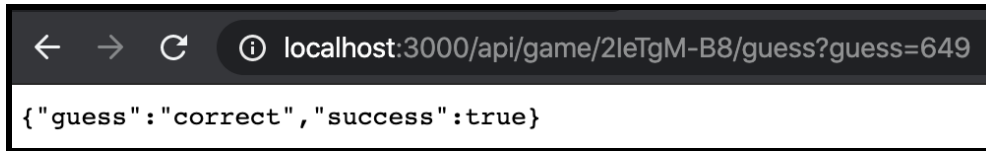
Note: Use the server console logs to get the game's correct number.

Test: Game Over *(Success)*

Open browser to: `http://localhost:3000/api/game/`**`2IeTgM-B8`**`/guess?`**`guess=500`**

← → C ⓘ localhost:3000/api/game/2IeTgM-B8/guess?guess=500

```
{"guess":"gameover","success":true}
```

Note: You can only test this condition after the correct answer was submitted

---

Test: Failure

Open browser to: `localhost:3000/api/game/`**`not-valid-id`**`/guess?`**`guess=500`**

← → C ⓘ localhost:3000/api/game/not-valid-id/guess?guess=500

```
{"success":false}
```
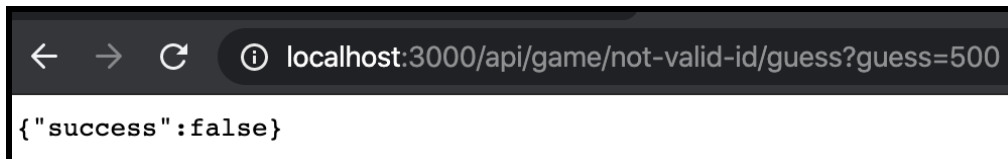
Iteration 7: Endpoint → Reset Game

'Approach' → Plan phase

Goal #7: Implement REST Endpoint to Reset a Game after it's over

Approach: Use Model, Controller, Routes to implement REST Endpoint

In this iteration, implement a REST endpoint to reset the game. The routes use parameters to allow variable url paths based on the unique game IDs. The game must be over to reset it. Resetting a game toggles game over to false and randomizes a new number.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: Add reset method to Model
- **Step 2**: Add resetGame method to Controller
- **Step 3**: Add endpoint to trigger resetGame

Step 1 (JS): game.model.js → Refactor Game: add reset()

Add a 'reset' method to the Game class. If the game exists & it's over then randomize the number and toggle the game over variable Return data without number, or null if game is not over or does not exist..

models/game.model.js

```
reset(id){
   if (this.games[id] && this.games[id].gameover){      //check game exists & its over
      const game = this.games[id];                       //get game from hashmap
      game.number = ~~(Math.random() * game.end + game.start)   //set new random int
      game.gameover = false;                             //set gameover to false
      const {number, ...data} = game;                    //destructure to remove number
      return data;                                       //return data, without number
   }
   return null;                                          //game not exists or not over
}                                                        //return nothing
```

Step 2 (JS): game.controllers.js → Refactor Controller: add resetGame()

Add a reset method to the Controllers class. This method passes params to the game's reset method. It sends back the result as JSON data. If the game returns null, then send a success: false response.

controllers/game.controllers.js

```js
resetGame(request, response){
    const id = request.params.id;              //get game id from response parameters
    const result = game.reset(id);             //invoke game's reset method
    if (result){                               //if result exists
        result['success'] = true;              //set result with a success:true
        response.json(result)                  //send data as JSON
    }
    else{                                      //if result does not exists
        response.json( {'success':false} );    //send no success as JSON
    }
}
```

Step 3 (JS): game.routes.js → Refactor

Destructure the resetGame method from the controllers module. Then setup router to trigger the resetGame method on a GET request to path '/:id/reset'. Note: the colon (:) specifies a route parameter.

routes/game.routes.js

```js
var express = require('express');
var router = express.Router();
const {resetGame, guess, getGame, newGame, test} = require('../controllers/game.controllers');

router.get('/test', newGame);
router.get('/new', newGame);
router.get('/:id', getGame);
router.get('/:id/guess', guess);
router.get('/:id/reset', resetGame);           //define variable route with colon(:)
                                               //Note: variable routes go after defined routes

module.exports = router;
```

'Assess' → Test phase

Launch the web server application from the bash terminal with command:
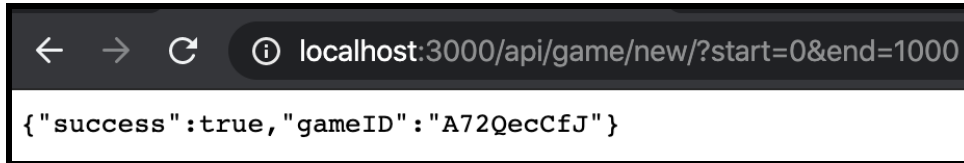
```
npm start
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

Make a Game & Guess Correct   *Note: Use server console for number*

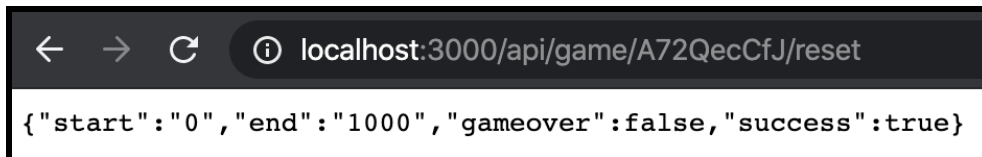Open browser to: `localhost:3000/api/game/new?start=0&end=1000`

```
← → C    ⓘ localhost:3000/api/game/new/?start=0&end=1000
{"success":true,"gameID":"A72QecCfJ"}
```

Open browser to: `http://localhost:3000/api/game/A72QecCfJ/guess?guess=185`

```
← → C    ⓘ localhost:3000/api/game/A72QecCfJ/guess?guess=185
{"guess":"correct","success":true}
```

Test: *Success*

Open browser to: `http://localhost:3000/api/game/A72QecCfJ/reset`

```
← → C    ⓘ localhost:3000/api/game/A72QecCfJ/reset
{"start":"0","end":"1000","gameover":false,"success":true}
```
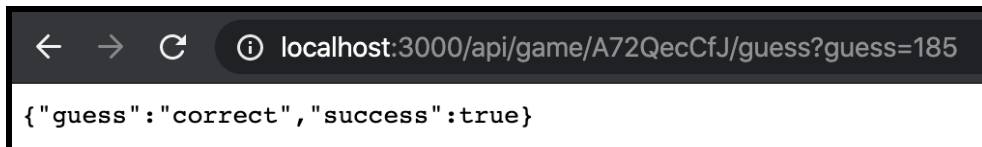
Test: Game Not Over (Failure)

Open browser to: `ttp://localhost:3000/api/game/A72QecCfJ/reset`

```
← → C    ⓘ localhost:3000/api/game/A72QecCfJ/reset
{"success":false}
```
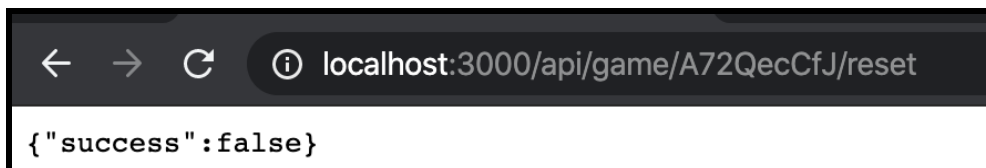
Test: Wrong ID (Failure)

Open browser to: `localhost:3000/api/game/not-valid-id/reset`

```
← → C    ⓘ localhost:3000/api/game/not-valid-id/reset
{"success":false}
```
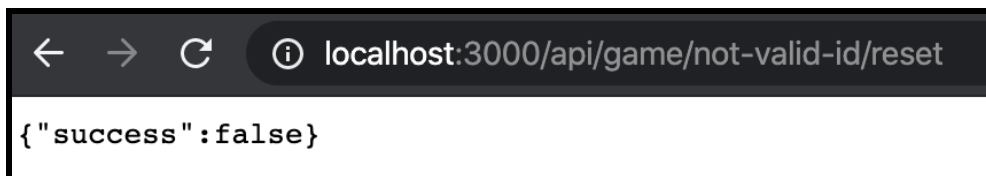
Iteration 8: Middleware: Localhost Handler

'Approach' → Plan phase

Goal #7: Configure this app to ignore CORs on localhost for local testing (next lab)

Approach: Access-Control-Allow-Origin

Next lab, we'll build a frontend client that uses this REST API to implement a multiplayer number guessing game. In this iteration, configure this app to bypass CORs restrictions on localhost

'Apply' → Do phase

JavaScript Steps

- **Step 1**: Refactor app.js to use our own middleware

Step 1 (JS): app.js → Refactor:

Refactor app.js to add a localhostHandler function and to have the app use it. This code should be implemented before the app listens on the port.

app.js

```javascript
app.use(localhostHandler);

function localhostHandler(request, response, next){
    response.header('Access-Control-Allow-Origin', '*');
    next();
}
```

'Assess' → Test phase

Launch the web server application from the bash terminal with command:

```
npm start
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

23

Conclusions

Final Comments

In this lab you implemented a backend REST API for a Multiplayer Number Guessing Game. This lab covered: Route Parameters, Query Strings, unique ids, REST API (Design & Develop), Export JavaScript modules, & REST Endpoints.

Future Improvements

- Design a frontend app that uses this Backend App
- Add Usernames for a game and show name & win record as a scoreboard
- Synchronize users together and display who is playing
- Store the games in a database
- Deploy into production on heroku

Lab Submission

Compress your project folder into a zip file and submit on Moodle.