Node Workshop

Node.js is a JavaScript runtime. Project will work to write a few RESTful endpoints running a server saving, retrieving, updating and deleting data.

	Project Creation

- npm is the Node Package Manager (like nuget for .net or apt-get for Linux)
- hapiJS is a node package (library) used to create a web server, including RESTful endpoints

Step Description	Command
Open command line	Open Git Bash On windows or terminal on Linux/Mac
Change to root directory	➢ **cd /c** on windows **cd /** on Linux/Mac
Make a new directory for project	➢ mkdir nodeproj
Change directory to new project directory	➢ cd nodeproj
Initialize a git repository	➢ git init
Initial npm	➢ npm init
answer npm questions	➢ defaults are fine
Add hapi.js that will be used in this project www.hapijs.com	➢ npm install --save hapi

	Project Setup

Step Description	Command

Create a new file **.gitignore** (filename begins with a dot)
 Add files / directories git will ignore
   node_modules - packages installed by npm
   *.map - files that are for debugging
   *.bak - some editors keep original files with bak extension
   ~* - some temp files begin with ~
	➢ ```add the following:```
	 node_modules
	 *.map
	 *.bak
	 dist
	 ~*

	HelloWord

Simple test to make sure node is installed properly

Step Description	Command
Create a directory called src	➢ mkdir src
Create a new file called main.js in src directory	➢ add the following: console.log('Hello World!');

Setup npm to execute main.js	➢ edit package.json
	 ➢ under scripts (line 6/7)add the following
	 ➢ "start": "node src/main.js",
	 ➢ The file should be:
	   `"main": "index.js",`
	   `"scripts": {`
	     `"start": "node src/main.js",`
	     `"test": "echo \"Error: no test specified\" && ex`
	   `},`

Execute the application	➢ npm start
	   returns:
	   Hello World!

	Git Commit

Check in changes to git

Step Description	Command
Add files to git from command line in the root project directory	➢ git add .gitignore ➢ git add package.json ➢ git add src
Verify what is ready to be committed	➢ git status
Commit	➢ git commit -m "initial project commit"

	Hapi HelloWorld

Modify **main.js** to use hapi and return hello world with the following code:

```
'use strict';
var Hapi = require('hapi');
```

```
var server = new Hapi.Server();

server.connection({port: 3000});

server.route({
    method: 'GET',
    path: '/',
    handler: function (request, reply) {
        reply('Hello World! from Hapi');
    }
});

server.start(function (err) {
    if (err) {
        throw err;
    }
    console.log('Server running at ', server.info.port);
});
```

Step Description	Command
From command line start application	npm start
Start Chrome or another browser	Type **http://localhost:3000**
Server will response with Hello, World! from Hapi	
To stop the server use **ctrl-c**	

Commit changes to git

Let's commit code changes to git

Step Description	Command
shows modified files	git status
modified files	git add -u add
commit changes	git commit -m "hapi helloworld"

Create Some Data

Create a new file **games.json** & save it in the **src** directory with the following:

```
[
  {
    "id": 1,
    "name": "Tic-Tac-Toe"
  },
  {
    "id": 2,
    "name": "Checkers"
  },
  {
    "id": 3,
    "name": "Chess"
  }
]
```

Add this file to our main.js after the var server line (around line 3)
```
var games = require('./games.json');
console.log(games);
```
start the application **npm start** and the result should be the following:

```
[ { id: 1, name: 'Tic-Tac-Toe' },
  { id: 2, name: 'Checkers' },
  { id: 3, name: 'Chess' } ]
Server running at  3000
```

Check-in changes

Step Description	Command
Show status	git status
a new untracked file games.json is listed, add it to git change list	git add src/games.json
also listed are modified files not on commit list. Add these as well	git add -u
commit change list	git commit -m "add games list"

<div align="center">

First RESTful Route

</div>

Add another route to the main.js file. Add this just before server.start (about line 17). This route will return a full list of all games.

```
server.route( {
    method: 'GET',
    path: '/games',
    handler: function (request, reply) {
        reply(games);
    }
});
```

Step Description	Command
Start the server	npm start
Open a browser window and enter	**http://localhost:3000/games**
Result in browser	`[{"id":1,"name":"Tic-Tac-Toe"},{"id":2,"name":"Checkers"},{"id":3,"name":"Chess"}]`

<div align="center">

Check in changes

</div>

Step Description	Command
stage changes in modified files	git add -u
Commit changes	git commit -m "added games endpoint"

<div align="center">

Second Endpoint

</div>

The last route returned a complete list of games. Let's return just a game by its id. To do this use a library called lodash. Lodash needs to be install first.

Step Description	Command
Install lodash and save the dependency in package.json	npm install --save lodash

At the top of the main.js file using the _ (underscore is common for defining the library lodash.)

```
var _ = require('lodash');
server.route( {
    method: 'GET',
    path: '/games/{id}',
    handler: function (request, reply) {
        var game = _.find(games, {'id': parseInt(request.params.id, 10)});
        reply(game);
    }
});
```

In the function **.find** the first parameter, games, is the data being searched. The second parameter an object of what to search for in games. In this case search the property 'id' for the value **request.params.id** which is what is sent in the path {id}. parseInt is converting it to a number.

<div align="center">

Validation

</div>

The /games/{id} endpoint works, but we can validate that id is a number using a library called joi. Let's install this library

Install joi npm install --save joi

Add require statement to top of main.js **var Joi = require('joi');**

Modify the /games/{id} add the **config** object and remove the parseInt function:

```
server.route({
    method: 'GET',
    path: '/games/{id}',
    handler: function (request, reply) {
        // var game = _.find(games, {'id': parseInt(request.params.id, 10)});
        var game = _.find(games, {'id': request.params.id});
        reply(game);
    },
    config: {
        validate: {
            params: {
```

```
            id: Joi.number().integer().min(1).required()
        }
    }
}
});
```

By adding the config object the param id is being converted to a number that must be an integer (no decimal) and a minimum value of 1. It is also required.

Start the server **npm start**

From the browser try the following: http://localhost:3000/games/1   http://localhost:3000/games/2   http://localhost:3000/games/x
http://localhost:3000/games/0   http://localhost:3000/games/-99

Check-in changes (Do you remember the steps?)

Boom – return html errors

Hapi has a library to return html error codes easily. For example 404 error if an a game is not found. For example, /games/4 doesn't exist.

Install boom npm install –save boom

Require boom var Boom = require('boom');

Change the endpoint **/games/{id}** add the **if block**

```
var game = _.find(games, {'id': request.params.id});
if (!game){
    return reply(Boom.notFound('game id not found'));
}
reply(game);
```

The if (!game) will be true if game is not found. The ! is a not operator. i.e. !true is false

Note: It is a good idea to always return reply(). This avoids an issue of replying twice.

Start the server: **npm start**

Try **http://localhost:3000/games/4**

Create endpoint to add a new game

Create a new endpoint that will add a new game to the list. This will use the method of **POST.**

Data for the new game (POST method) is sent as the request payload.

Use a Lodash function to get game with the max index, add 1 when adding the new game.

Add the following to **main.js**

```
server.route({
    method: "POST",
    path: '/games',
    handler: function (request, reply){
        var index = _.maxBy(games, 'id').id + 1;    // Get the max index and add 1
        var name = request.payload.name;
        var game = {id: index, name: name};         // Create a new game object
        games.push(game);                  // push game on the games array
        return reply (game);                          // reply with the added game object
    },
    config: {
        validate: {
            payload: {
                name: Joi.string().required()
            }
        }
    }
});
```

Use Postman to test endpoint

The browser URL bar does a GET method request. To perform a **POST** and include the payload we will use the **Postman** tool.

- Start the server to test the route: **npm start**
- Start **postman** application
- Using postman create two endpoints. One to display the list of games and the other to add a new game.
- Create a GET Games list request
  - From Collection Menu Select New Collection
  - Name the collection: **node-workshop**
  - In the Enter URL here: http://localhost:3000/games
  - Click the **Save** button.
  - In the **Save Request** dialog entry for **Save to existing collection** Select **node-workshop** from the drop-down.
- Create a POST Games request
  - Click the + tab in the top middle of the app to create a new request
  - In the Enter URL here: http://localhost:3000/games
  - Change the method from **GET** to **POST**
  - Select the **Body** Tab
  - Below the **Body** tab there are formatting options. Select **Raw** and **JSON (application/json)** from the drop-down.
  - Enter:
    ```
    {
        "name": "backgammon"
    }
    ```
  - Click the **Save** button.
  - In the **Save Request** dialog entry for **Save to existing collection** Select **node-workshop** from the drop-down.
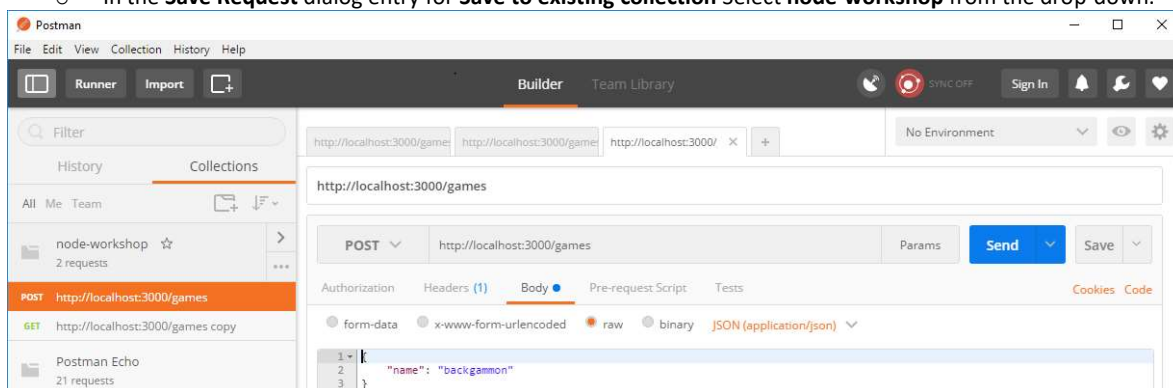
- Click the send button to execute the endpoint. If successful, the response will appear at toward the bottom of the app.
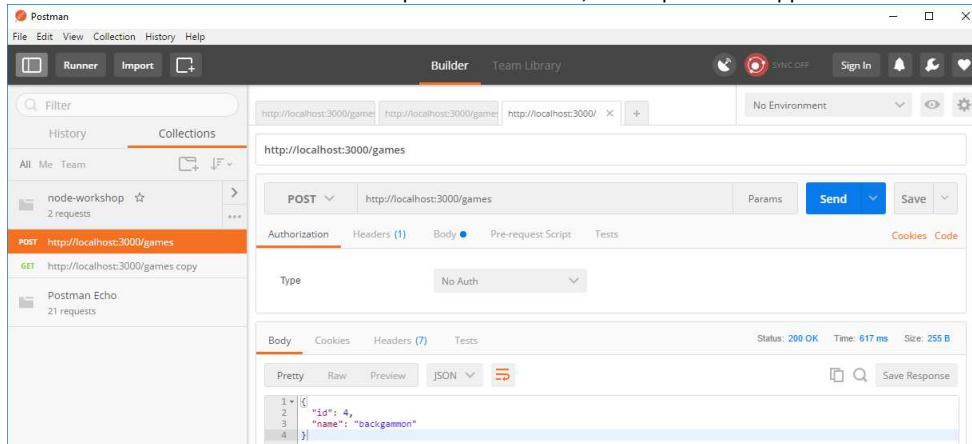
Any added games will be lost when the server is stopped. The additions are only in memory. A future workshop will persist the data.

Create endpoint to modify a new game

Create a new endpoint that allows updating a game already in the list. This will use the method **PUT.**

- Open **main.js** and add the new endpoint.

```
server.route({
    method: "PUT",
    path: '/games/{id}',
    handler: function (request, reply){
        var id = request.params.id
        var index = _.indexOf(games, _.find(games, {'id': id})); // find id, array index
        var name = request.payload.name;
        var game = {id: id, name: name};                    // create new game object
        games[index] = game;                                // Update (replace) existing game object
        reply (game);                                       // reply with the updated game
    },
    config: {
        validate: {
            params:{
                id: Joi.number().integer().min(1).required()
            },
            payload: {
                name: Joi.string().required()
            }
        }
    }
});
```

- The response returns the game object with the new id. In this example, the game name is not changed, but the server could add additional data in the response. For example, entering city, state could return city, state and zip.

Use Postman to test endpoint

Like a **POST** method, **PUT** sends data as payload. **Postman** will be used to send the PUT request.

- Start the server to test the route: **node start**
- Start **postman** application
- In the Enter URL here: http://localhost:3000/games/1
  - The 1 is the id of the first game.
- Change the method from **GET** to **PUT**
- Select the **Body** Tab
- Below the **Body** tab there are formatting options. Select **Raw** and **JSON (application/json)** from the drop-down.
- Enter:

```
{
"name": "3d Tic-Tac-Toe "
}
```

- Click the **Save** button.
- In the **Save Request** dialog entry for **Save to existing collection** Select **node-workshop** from the drop-down.
- Click the send button to execute the endpoint. If successful, the response will appear at toward the bottom of the app.

HapiJS Plugins

As application grow having all routes in one file becomes difficult to manage. The file becomes larger and difficult to follow. Hapi.js solves this with plugins. A plugin is a set of endpoints and related code. In addition to organizing your own code, plugins are available as libraries that can be used in multiple projects. Review the list of some plugins: http://hapijs.com/plugins

In the workshop example Games is one resource. Adding Categories, Players, Scores, etc... could be other pluging. Netflix might have Movies, actors, directors, and user ratings.

Refactor Game endpoints to plugin

More Info: http://hapijs.com/tutorials/plugins?lang=en_US

Step Description	Command
Create a new file, games.js, for the /game endpoints	Create a new file src/games.js

Edit src/games.js	`exports.register = function (server, options, next) {`
	`    // paste server.routes here`
	`    next();`
	`};`
	
	`exports.register.attributes = {`
	`    name: 'games',`
	`    version: '1.0.0'`
	`};`

Move the server.route functions to games.js	Cut the server.route calls and paste into src/games.js The routes go inside the exports.register function before the next(); function call Leave the route '/' that returns Hello World in main.js

Edit src/main.js
 	
Add the code to load the games plugin.
 	
The parameter being provided is an array of objects. Even though only one plugin is added time.
	Add the following between server.connection & server.start:
	`server.register([{`
	`    register: require('./games'),`
	`    options: {}`
	`    }`
	`], function(err){`
	`    if(err){`
	`        throw err;`
	`    }`
	`})`

Since we moved routes to /src/games dependencies are needed in the file.
 	
Remove the dependencies no longer needed in main.js as well as console.log(games);
	Add to src/games.js
	`var _ = require('lodash');`
	`var Joi = require('joi');`
	`var Boom = require('boom');`
	`var games = require('./games.json');`
	Remove unused dependencies from src/main.js
	`var _ = require('lodash');`
	`var Boom = require('boom');`
	`var games = require('./games.json');`
	`console.log(games);`

The application only has 1 plugin, but as the application grows it could have 20, 30 or more. Using plugins allows the code to be organized. If modification to games plug is require, developer doesn't have to wade through all the code for all the other plugins.

Add Endpoint Documentation (LOUT)

LOUT is a hapi.js plugin that documents endpoints. LOUT uses vision and inert plugins which we will also install.

Step Description	Command
Install LOUT dependency and save to package.json	npm install –save vision inert lout

Modify the code in **main.js**	server.register([{
	    register: require('./games'),
	    options: {}
	    },
	    require('vision'), require('inert'), { register: require('lout') }
	], function(err){
	    if(err){
	        throw err;

	`}` `});`
Start Server	npm start
In browser	localhost:3000/docs

Add Watch & Reload

Step Description	Command
Add a developer dependency called nodemon	npm install --save-dev nodemon
edit package.json	Change the start script to: "start": "nodemon src/main.js"
Start the server using nodemon	npm start
Open the brower	localhost:3000

Setup Test Framework

A test framework will verify the application works as expected. Tests provide feedback that changes to code haven't introduced errors. Test Driven Development is a process for writing code. The process is:

- Red: Write a test that fails
  - Verifies that the test is being called
  - If a test should fail and it actually succeeds maybe the code doesn't work as expected.
- Green: Make just enough changes to make it succeed
  - Focus on making the code pass as quickly as possible.
- Refactor: Improve code quality and readability, remove duplicate code.
  - During refactoring focus on the code the exists and increasing readability, stability.
  - Don't introduce new features.

Step Description	Command
Install jasmine & jasmine-spec-report. Jasmine is a testing framework to test app code. Jasmine-spec-reporter, outputs a report of the tests status. There are other reporters as well.	`npm install --save-dev jasmine jasmine-spec-reporter`
Initialize jasmine. This will create a spec folder	`From the root node-workshop project folder run jasmine init`

Create a new file called jasmine-runner.js
 	
This file is placed in the root project directory, the parent folder of src.
	`Add the following to jasmine-runner.js`
	
	`var Jasmine = require('jasmine');`
	`var SpecReporter = require('jasmine-spec-reporter');`
	
	`var jrunner = new Jasmine();`
	`jrunner.env.clearReporters();`
	`jrunner.addReporter(new SpecReporter());`
	`jrunner.loadConfigFile();`
	`jrunner.execute();`

Add test script to package.json to run tests	`Open package.json and replace`
	`"test": `**`"echo \"Error: no test specified\" && exit 1"`**
	`With`
	`"test": `**`"node jasmine-runner.js"`**

| Verify setup by running tests | ```
npm test

result:
C:\dev\codenorman\node-workshop>npm test

> node-workshop@1.0.0 test C:\dev\codenorman\node-workshop
> node jasmine-runner.js

Spec started

Executed 0 of 0 specs SUCCESS in 0.006 sec.

C:\dev\codenorman\node-workshop>
``` |

Write Our First Test

Tests are written to verify that the code works as expected. Each js file is tested in isolation.

You can review the docs: https://jasmine.github.io/2.5/introduction

To get the basics of writing tests a new file will be created.

Step Description	Command
Start by writing a test. Create a new file in /spec/math.spec.js By convention all test end in spec.js. spec is short for specification. This file will contain all the tests for the	`New file /spec/math.spec.js`
| Write a test<br>**describe used to describe the thing being tests.**<br>A test suite is started by using **describe** function that takes two arguments. A string description and function.<br>**it is a test that verifies some operation**<br>A test is started by using the **it** function with two arguments, first a string explaining the test and a function comparing expected to actual result. | ```
var math = require('../src/math');
describe ('math', function(){
    describe('add', function() {
        it('1+2 should equal 3', function () {
            expect(math.add(1, 2)).toEqual(3);
        });
    });
});
``` |
| Run test<br>It should fail. This proves the test is being called. The math module has not been written yet which will be done next. | ```
npm test
module.js:471
    throw err;
    ^

Error: Cannot find module '../src/math'
    at Function.Module._resolveFilename (module.js:469:15)
    at Function.Module._load (module.js:417:25)
    at Module.require (module.js:497:17)
    at require (internal/module.js:20:19)
    at Object.<anonymous> (C:\dev\codenorman\node-workshop\spec\math.spec.js:1:74)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
npm ERR! Test failed.  See above for more details.
``` |
| Write the math.js file | ```
Create a new file /src/math.js and add the following:

module.exports = {
    add: function(n1, n2){
        return 3;
    }
}
``` |
| Run test<br>It passes. | ```
npm test
C:\dev\codenorman\node-workshop>npm test

> node-workshop@1.0.0 test C:\dev\codenorman\node-workshop
> node jasmine-runner.js

Spec started

  math

    add
      √ 1+2 should equal 3

Executed 1 of 1 spec SUCCESS in 0.014 sec.
``` |
| Add a second test | ```
Add the following to /spec/math.js after the first
test ~line 7
    it('2+3 should equal 5', function(){
        expect(math.add(2,3).toEqual(5));
    })
``` |
| Make sure it fails | `npm start` |

Make it pass	Edit src/math.js
	return 3 to return n1 + n2;
	
	C:\dev\codenorman\node-workshop>npm test
	
	> node-workshop@1.0.0 test C:\dev\codenorman\node-workshop
	> node jasmine-runner.js
	
	Spec started
	
	  math
	
	    add
	      √ 1+2 should equal 3
	      √ 2+3 should equal 5
	
	Executed 2 of 2 specs SUCCESS in 0.015 sec.

Refactor	At this time there is nothing to refactor

homework

- /games is an example plugin of the games resource. Create a new resource of anything you'd like (persons, videos, etc…) Include GET list, GET item, POST, PUT has a new plugin. Write using TDD.
- Add the GET /games/{id} endpoint to postman
- Add missing test for the games plugin

Books

You don't know Javascript: https://github.com/getify/You-Dont-Know-JS

Learning Javascript Design Patterns: https://addyosmani.com/resources/essentialjsdesignpatterns/book/

Tutorials, Projects, etc.

Think Like a Git: http://think-like-a-git.net/

Git-It: https://github.com/jlord/git-it

Nodeschool Javascripting: https://github.com/workshopper/javascripting

Learn You Node: https://github.com/workshopper/learnyounode

How to npm: https://github.com/workshopper/how-to-npm

Make Me Hapi: https://github.com/hapijs/makemehapi

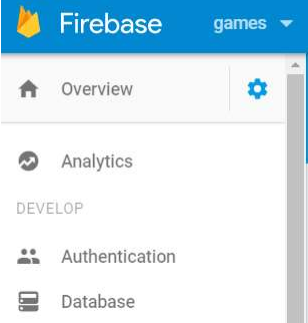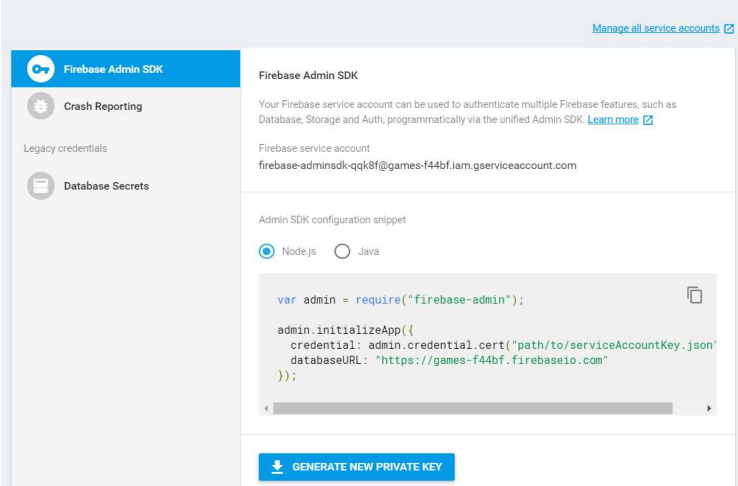**Elevator Saga** http://play.elevatorsaga.com/

Practice

Project Euler – Project Euler has quite a few simple to hard problems that require developing an algorithm to solve.

Katas - https://en.wikipedia.org/wiki/Kata_(programming)

Firebase

Firebase will allow us to persist game data.

Step Description	Command
Create a new source file for that will be the source for the firebase.	Create a new file src/games-db.js
Sign up for a firebase account.	https://firebase.google.com/
Create a new firebase project	Click the **Create New Project** button Welcome to Firebase Tools from Google for developing great apps, engaging with your users, and earning more through mobile ads. Learn more CREATE NEW PROJECT or import a Google project
Name the project	Enter **games** for the name and click **Create Project** Create a project × Project name games Country/region ⑦ United States ▼ By default, your Firebase Analytics data will enhance other Firebase features and Google products. You can control how your Firebase Analytics data is shared in your settings at anytime. Learn more **By proceeding and clicking the button below**, you agree that you are using Firebase services in your app and agree to the applicable terms. CANCEL **CREATE PROJECT**

Setup a service account for the node server application	Click the gear icon in the left panel and select Project Settings Click the Service Accounts Tab. Click to **Generate new private key.** This will download the private key to be used for the project. This should be secured and not be added to github or other public location. This is only for the server to have access to the firebase. If the private key is made public anyone can use it to access the firebase project. Copy the code on the page for node and paste it after the require statement
Copy the private key to a config directory. Add the config directory to .gitignore to avoid checking it into get.	Create src/config directory and move the private key file into this directory Open .gitignore and add **/src/config** In games-db.js modify the path **path/to/serviceAccountKey.json** to **./config/** and the name of you private key file.
Firebase data is stoed as a JSON structure. Structure Your Database Best Practices for Firebase DB data structure	https://firebase.google.com/docs/database/web/structure-data
The games-db is initialized with the credentials to read & write all data. Let's add a database reference to store the data and a reference for games to be stored	var games = defaultDatabase.ref("games");

| Add code to create some test data

Note: Instead of typing all this you can get it from the github repository | ```js
var admin = require("firebase-admin");
var https = require('https');

admin.initializeApp({
  credential: admin.credential.cert("./config/games-f44bf-firebase-adminsdk-qqk8f-f94c799213.json"),
  databaseURL: "https://games-f44bf.firebaseio.com"
});

var defaultAuth = admin.auth();
var defaultDatabase = admin.database();

var games = defaultDatabase.ref("games");
var players = defaultDatabase.ref("players");

games.once('value').then(function (snapshot) {
  if (!snapshot.exists()) {
    console.log('INITIALIZE DATABASE ');
    games.set({
      'tic-tac-toe': {
        minPlayers: 2,
        maxPlayers: 2
      },
      'checkers': {
        minPlayers: 2,
        maxPlayers: 2
      },
      'Chess': {
        minPlayers: 2,
        maxPlayers: 2
      },
      'Klondike': {
        minPlayers: 1,
        maxPlayers: 1
      }

    });
    addGame('Poker', 2, 5);
    addGame('Go Fish', 2, 4);
    addGame('War', 2, 2);
    addGame('Go', 2, 2);
    addGame('Uno', 2, 10);
  }
});

var addPlayer = function(player, name){
  var newPlayer = players.child(player);
  newPlayer.push(name);

};

games.on('child_added', function (data) {
``` |

	```for (var i = data.val().minPlayers; i <= data.val().maxPlayers; i++) { addPlayer(i, data.key) } });```
Write the function addGame, updateGame, deleteGame and gameList	```var addGame = function (name, minPlayers, maxPlayers) { console.log('addgame', name) var newGame = games.child(name); newGame.set({minPlayers: minPlayers, maxPlayers: maxPlayers}); }; var updateGame = function (name, minPlayers, maxPlayers) { var minKey = name + '/minPlayers'; var maxKey = name + '/maxPlayers'; games.update({ minKey: minPlayers, maxKey: maxPlayers }) }; var deleteGame = function (name) { games.remove(name); }; var gamesList = function() { var result; games.on("value", function (snapshot) { result = snapshot.val(); }); return result; };```
Export functions for use when module is required in another javascript file.	```module.exports = { addGame: addGame, updateGame: updateGame, deleteGame: deleteGame, gamesList: gamesList };```