NodeJS

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Authorized & published by Summitworks Technologies Inc

Agenda

- Frameworks for Node.js
- Express Framework
- Advantages of Express.js
- Install Express.js
- Express.js Web Application
- Create Web Server
- Configure the routes
- Nodemon Express
- Setting Port with an Environment Variable
- Express Route Parameters
- Creating Routes
- Creating RESTful APIs
- HTTP GET Requests
- HTTP POST Requests
- HTTP PUT Requests
- HTTP DELETE Requests

Frameworks for Node.js

- You learned that we need to write lots of low level code ourselves to create a web application using Node.js in Node.js web server section.
- There are various third party open-source frameworks available in Node Package Manager which makes Node.js application development faster and easy.
- You can choose an appropriate framework as per your application requirements.
- Few frameworks for Node.js.
  - Express
  - Geddy
  - Hapi.js
  - Derbyjs
  - Koa
  - Sails.js
  - Chocolate.js

Express.js

- Express.js is a web application framework for Node.js.
- "Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.“ - official web site: Expressjs.com
- It provides various features that make web application development fast and easy which otherwise takes more time using only Node.js.

Install Express.js

- You can install express.js using npm.
- The following command will install latest version of express.js globally on your machine so that every Node.js application on your machine can use it.

```
npm install -g express
```

Express.js Web Application

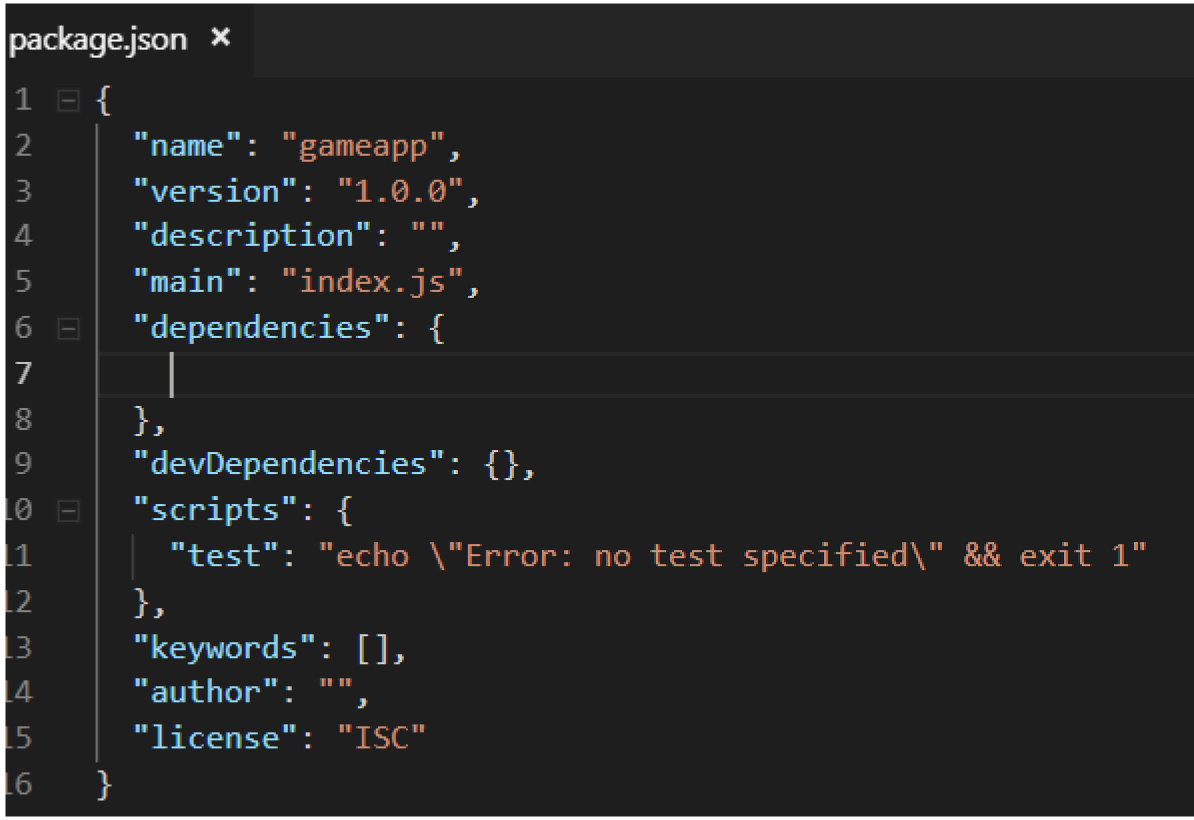
- In this section, let’s learn how to create a web application using Express.js.
- Express.js provides an easy way to create web server and render HTML pages for different HTTP requests by configuring routes for your application.
- Let’s see how to build a simple REST API using Express to allow for all of the CRUD operations.
- ***REST*** is referring to ***Representational State Transfer***. It is a convention for building HTTP services via a client and server architecture.
- In REST, the HTTP protocol is used to facilitate Create, Read, Update, and Delete of resources on the server.

Express.js Web Application cont.

- These operations can be referred to collectively as CRUD operations.
- Each operation uses a different HTTP verb.
- Create uses a POST request.
- Read uses a GET request.
- Update uses a PUT request.
- Finally, Delete uses a DELETE request.
- In this tutorial, we will create all of these services using Node.js and Express.

Create Web Server

- Create a directory named **Game App**.
- Navigate to the directory using command: **cd Game App**.
- Create package.json using the command: **npm init –yes**
- Install express using the command **npm i express**.
- We also need an index.js file in the root of the project, so we can create that too.

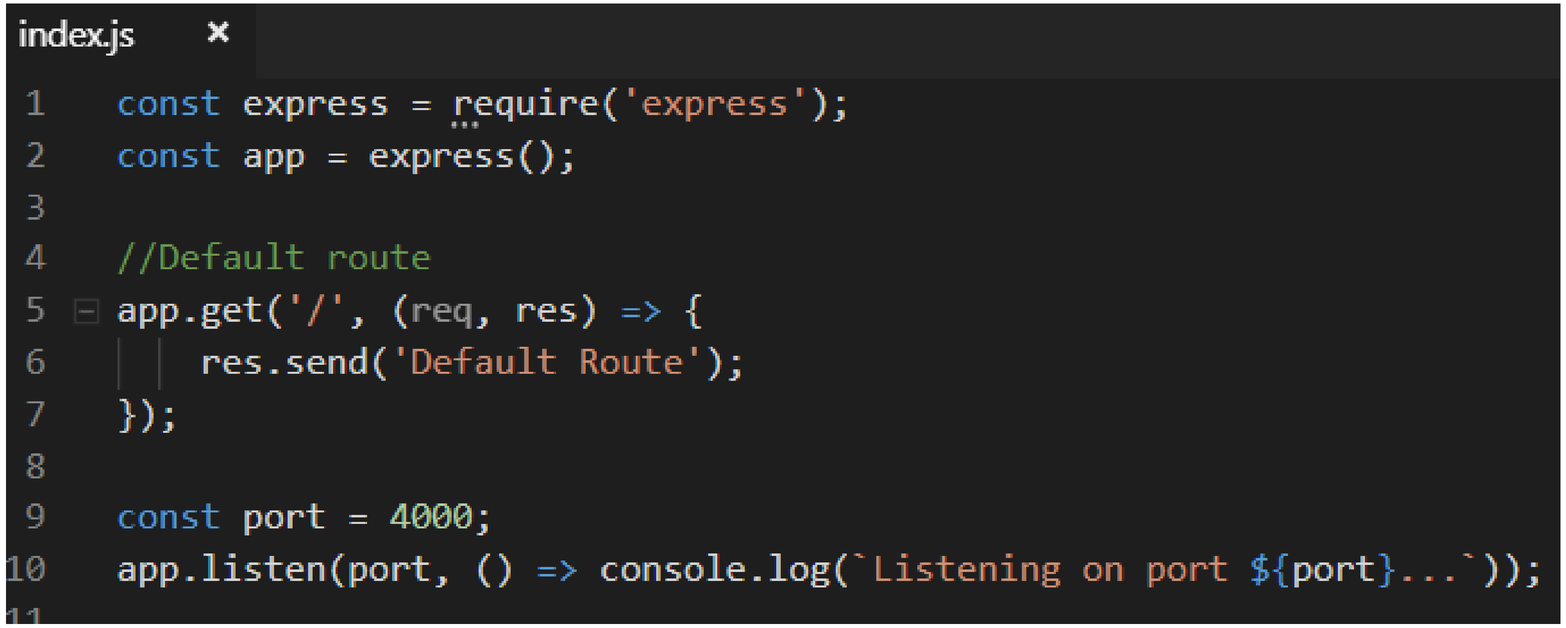


package.json ×

```
{
  "name": "gameapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "dependencies": {

  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Create Web Server cont.

- Our index.js file looks like as shown here.
- We are requiring the express module, and then calling the express() function and assigning the result to the app constant.
- The result is an Object, and by convention it is typically named app.

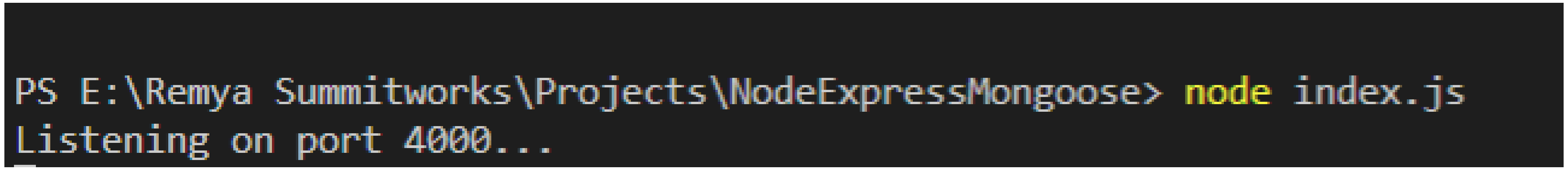


index.js ×

```
const express = require('express');
const app = express();

//Default route
app.get('/', (req, res) => {
    res.send('Default Route');
});

const port = 4000;
app.listen(port, () => console.log(`Listening on port ${port}...`));
```

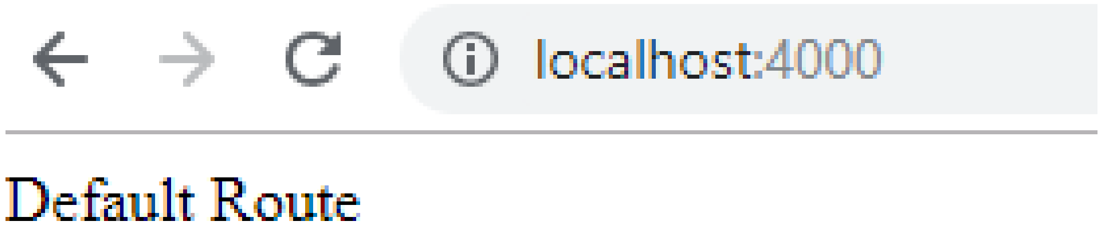
Create Web Server cont.

- We can launch the web server by typing node index.js at the command prompt.



```
PS E:\Remya Summitworks\Projects\NodeExpressMongoose> node index.js
Listening on port 4000...
```

- Now that the server is running, we can load up http://localhost:4000 in the browser to see that all is good.



localhost:4000

Default Route

Basic Routing

Basic Routing:

- Routing determines how the application responds to a client request to a particular endpoint which is a URL or path and a specific HTTP request method (GET,POST,PUT,DELETE)
- Each route can have one or more handler functions, which is executed when the route is matched.
- The structure of route is router.method(Path, Handler)

  router – instance of **express.Router**

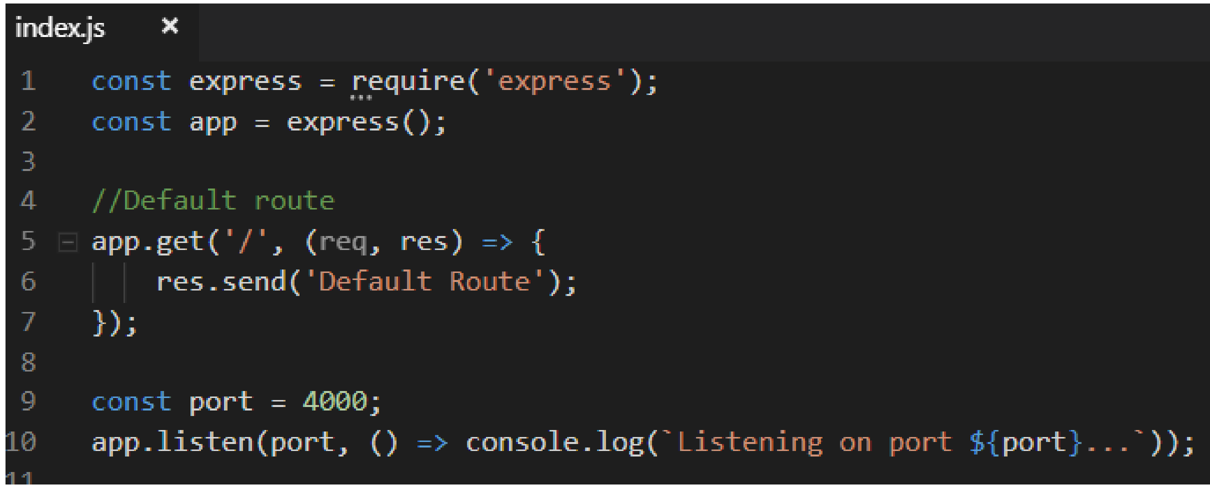
  Method – Can be any HTTP request method

  Path – Is the path on the server

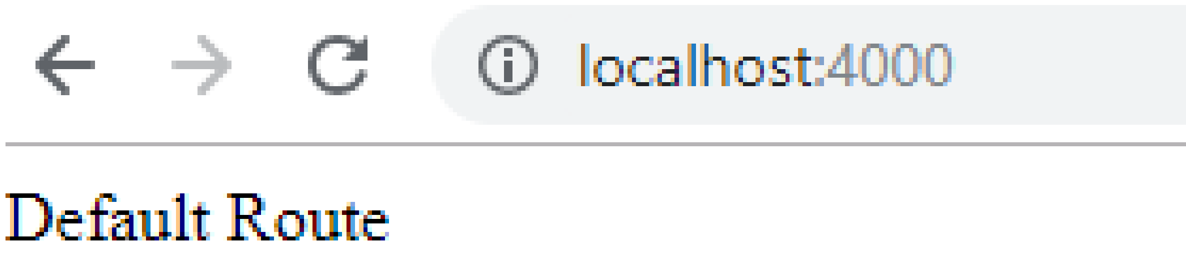
  Handler – Is the function executed when the route is matched

Configure the routes

- Now we will add a default route.
- We want to be able to visit the localhost:4000/ and see “Default Route” on the browser.
- In order to test this, you’ll need to restart the web server at the command line sing the command node index.js.
- Now, go ahead and visit http://localhost:4000/ in the browser.


index.js

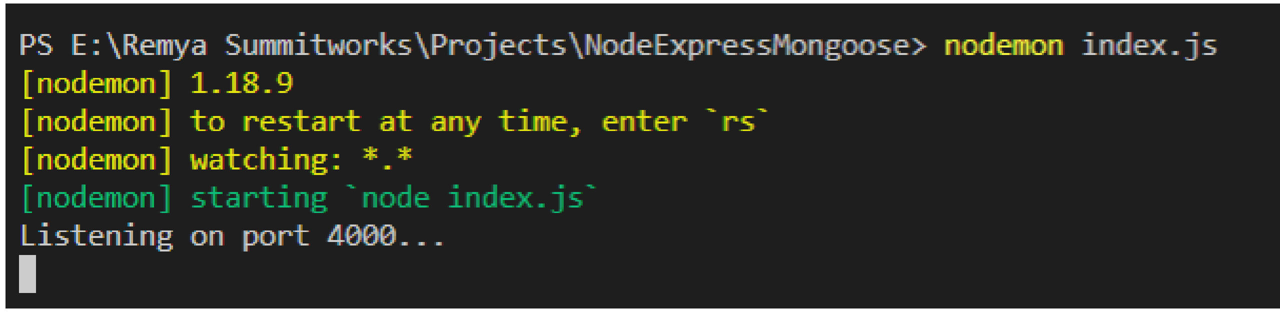
```
const express = require('express');
const app = express();

//Default route
app.get('/', (req, res) => {
    res.send('Default Route');
});

const port = 4000;
app.listen(port, () => console.log(`Listening on port ${port}...`));
```


localhost:4000

Default Route

Nodemon Express

- As we move along, it is going to be tiresome to manually restart the server on every update we make.
- We can easily deal with this problem by making use of nodemon when launching the server.
- Nodemon can be installed using the command: npm i -g nodemon at the command line.
- Once ready, halt the server and re launch it using nodemon index.js.
- Now when you make changes to your files, You will no longer need to stop and then restart the server.
- Go ahead and make your updates, and you'll see the command line reflect this automatically and the result in the browser is updated as well.

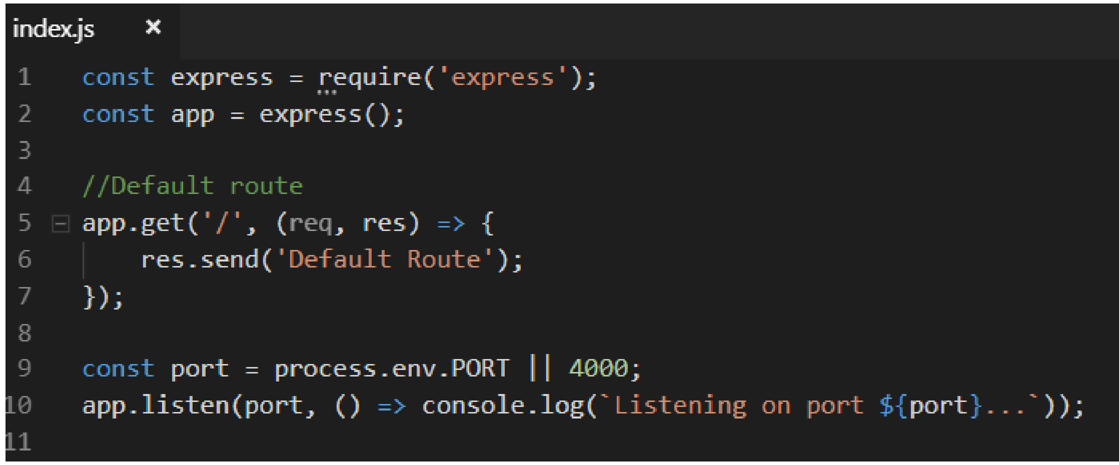


```
PS E:\Remya Summitworks\Projects\NodeExpressMongoose> nodemon index.js
[nodemon] 1.18.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Listening on port 4000...
```

Setting Port with an Environment Variable

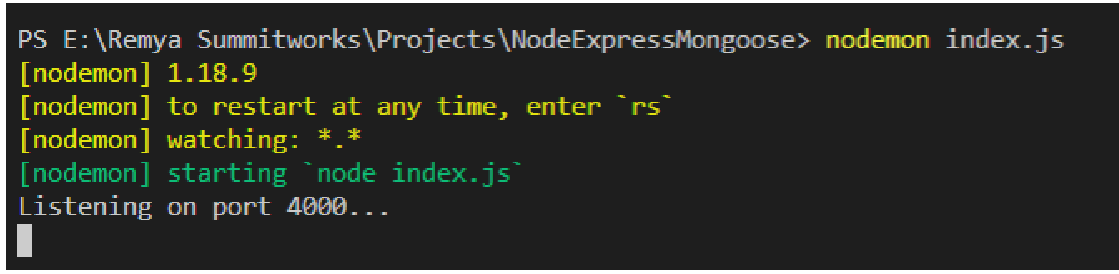
- So far we have been hard coding the values for different application variables such as the listening port.
- The hardcoded values work in the development environment, but its not going to work in production environment. When you deploy the app to the hosting Env. The port is dynamically assigned by the hosting environment. So we can't rely on this port to be available.
- We can fix this is by using the environment variable. Typically in hosting Env. for the node application, we have an Env. variable called PORT which is a variable that is part of Env in which the process runs. We can use this process object and property called Env and the variable name PORT.

  process.env.PORT || 4000.
- Here we check for an environment variable with the name of PORT, and if it is set, use that value. If it is not set, you can fall back to a sensible default.



index.js

```
const express = require('express');
const app = express();

//Default route
app.get('/', (req, res) => {
    res.send('Default Route');
});

const port = process.env.PORT || 4000;
app.listen(port, () => console.log(`Listening on port ${port}...`));
```



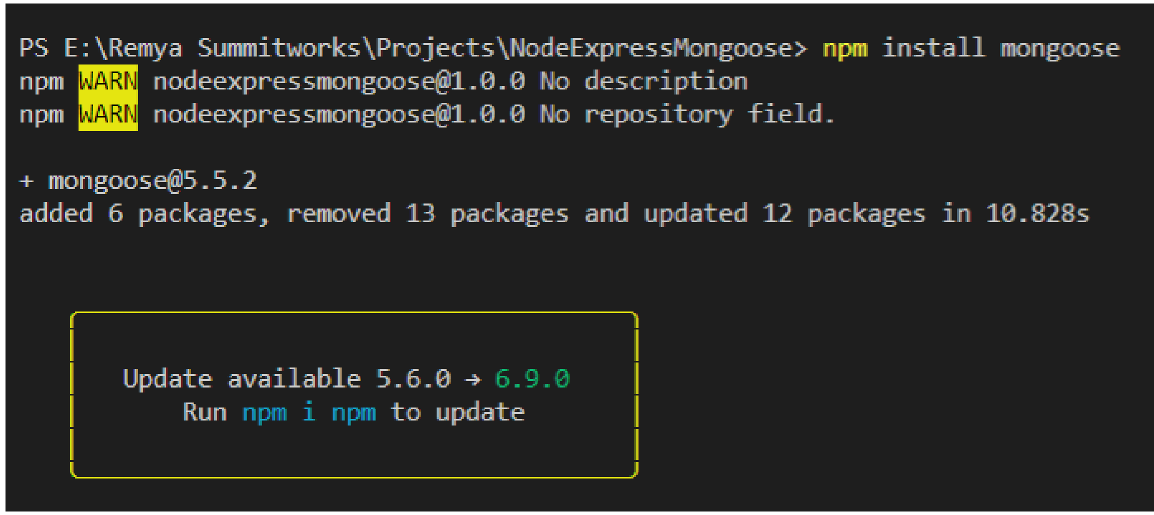
```
PS E:\Remya Summitworks\Projects\NodeExpressMongoose> nodemon index.js
[nodemon] 1.18.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Listening on port 4000...
```

Mongoose Introduction

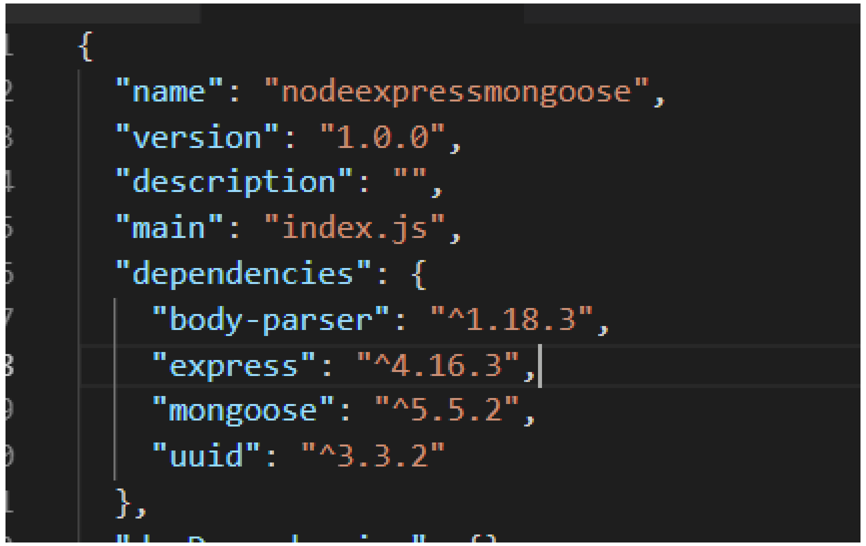
- Mongoose is one of the Node.js libraries that provides MongoDB object mapping, in a simple manner.
- Mongoose is a library of Node.js, it provides interaction with MongoDB using Object-Relation- Mapping.
- Mongoose provides a straight-forward, schema-based solution to model our application data. It includes built-in type casting, validation, query building, business logic hooks and more.
- For example consider that we are operating a store. Store has items. Each item could have properties : name, id, price, discount price, etc. With Mongoose we can model our items and do insertions or reads from the MongoDB Collection in terms of model objects, not bothering about the details of an object. Mongoose provides abstraction at Model level.
- Let’s learn about Mongoose package, how to install it, and different operations that are available with MongoDB.

Mongoose Installation

- Let's now install **Mongoose** using npm package manager
- Open your terminal and type the following command: **npm i mongoose**.
- You will see mongoose added in your package.json.



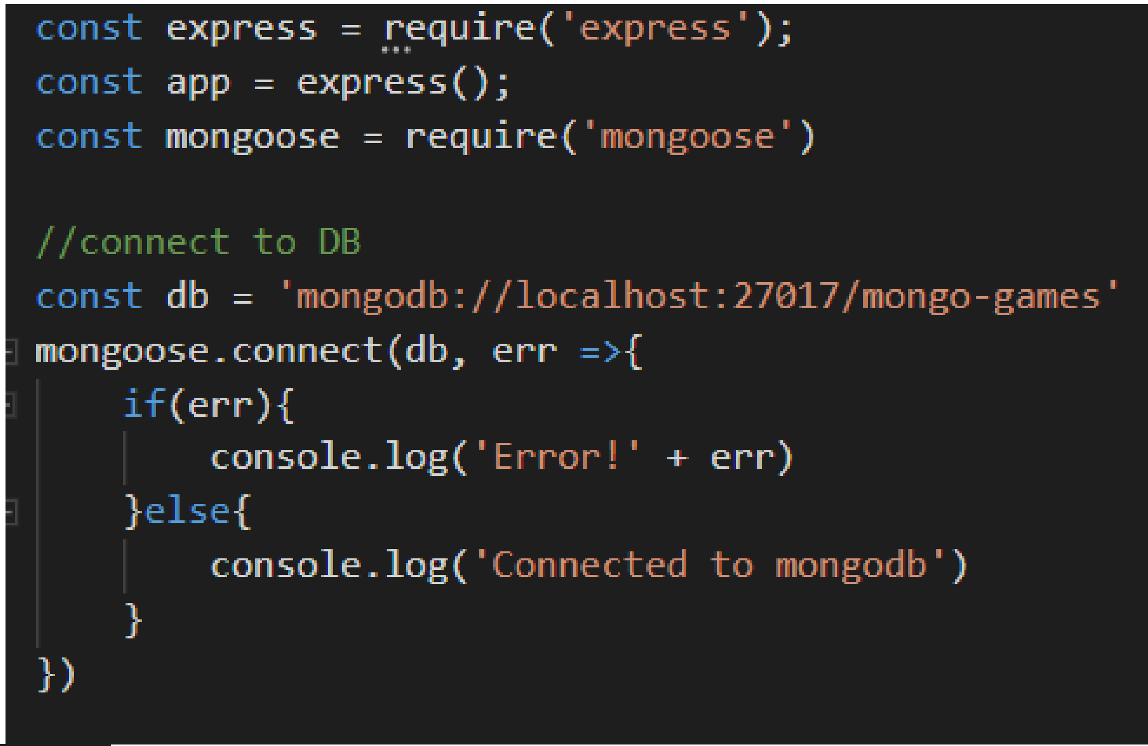
```
PS E:\Remya Summitworks\Projects\NodeExpressMongoose> npm install mongoose
npm WARN nodeexpressmongoose@1.0.0 No description
npm WARN nodeexpressmongoose@1.0.0 No repository field.

+ mongoose@5.5.2
added 6 packages, removed 13 packages and updated 12 packages in 10.828s


        Update available 5.6.0 → 6.9.0
          Run npm i npm to update
```



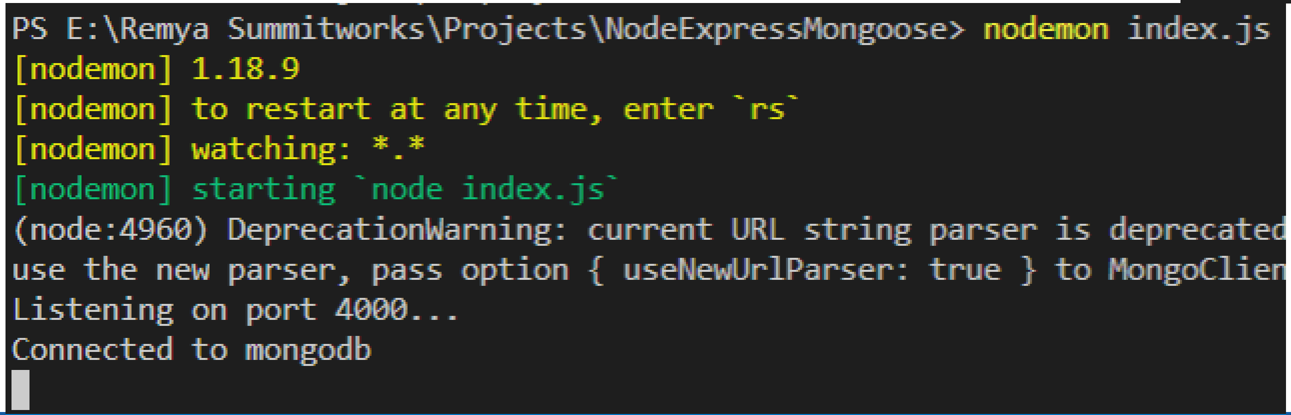
```
{
  "name": "nodeexpressmongoose",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "dependencies": {
    "body-parser": "^1.18.3",
    "express": "^4.16.3",
    "mongoose": "^5.5.2",
    "uuid": "^3.3.2"
  },
```

Connecting to the database using Mongoose

- Refer mongodb slides for installation instructions.
- Once mongodb is installed, we can connect to the database using mongoose.
- In your index.js file write the code shown here.
- Here we are importing mongoose using the require().
- Then connecting to database (mongo-games) using mongoose connect().
- If the connection is successful, you will see the following message in the terminal.



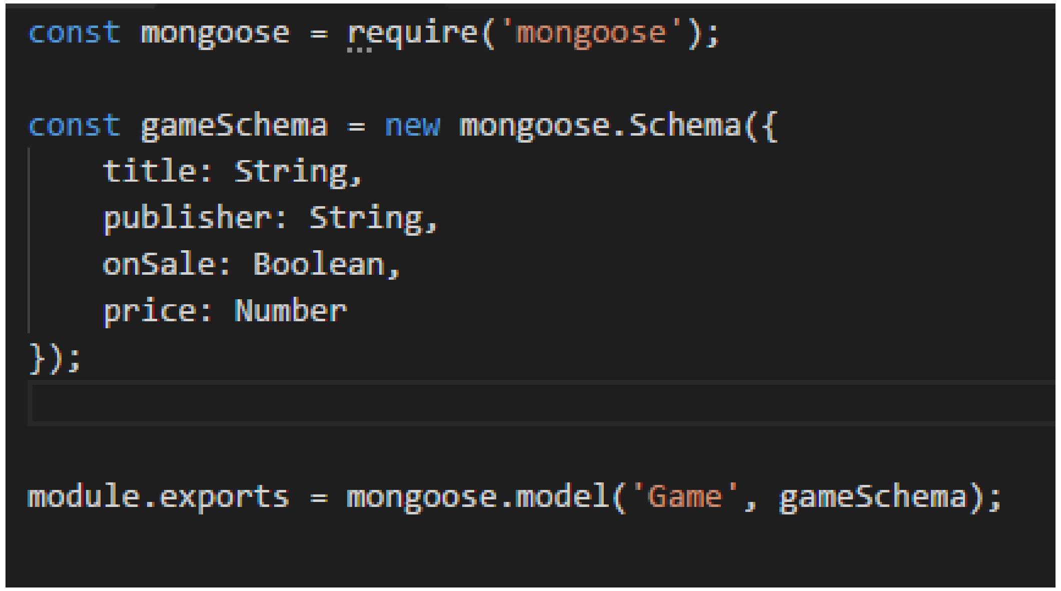
```
const express = require('express');
const app = express();
const mongoose = require('mongoose')

//connect to DB
const db = 'mongodb://localhost:27017/mongo-games'
mongoose.connect(db, err =>{
    if(err){
        console.log('Error!' + err)
    }else{
        console.log('Connected to mongodb')
    }
})
```



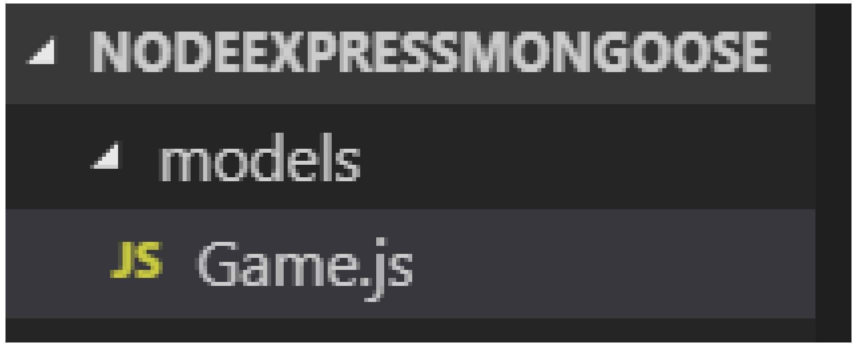
```
PS E:\Remya Summitworks\Projects\NodeExpressMongoose> nodemon index.js
[nodemon] 1.18.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
(node:4960) DeprecationWarning: current URL string parser is deprecated
use the new parser, pass option { useNewUrlParser: true } to MongoClien
Listening on port 4000...
Connected to mongodb
```

Mongoose Schema

- ***Mongoose*** will use the terms ***Schema*** and ***Model***.
- While doing any operation, in this entire session we will use the terms 'Schema' and 'Model' frequently so we need to know. Lets know those.
- Mongoose Schema will create a mongodb collection and defines the shape of the documents within that collection.
- If we want to create a mongodb collection in a structured manner similar to SQL tables then we can do that using mongoose Schema.
- In the schema creation we will specify the details of fields like field name, its type, default value if need, index concept if need, constraint concept if need.
- Let's see how to create a Schema in Mongoose.
- Create a directory 'model' and file 'Game.js' inside it.
- Our mongoose schema looks like as shown here:



```
const mongoose = require('mongoose');

const gameSchema = new mongoose.Schema({
    title: String,
    publisher: String,
    onSale: Boolean,
    price: Number
});


module.exports = mongoose.model('Game', gameSchema);
```

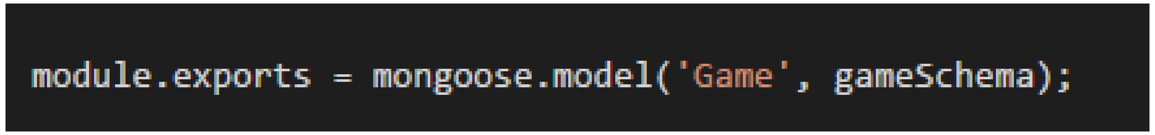


Mongoose Schema cont.

- We want to insert a new game into our Mongo database, so we need to define the Schema of the data in Mongoose first.
- That is exactly what we did in the code above. Here is what the code does.
- gameSchema: Defines the shape of game documents in MongoDB
- new mongoose.Schema({}): Creates a new instance of the Mongoose Schema class where an object is passed in.
- title: String Each game needs a title, which is a string value.
- publisher: String Each game also has a publisher, also a string value.
- onSale: Boolean Is the game currently on sale? Yes or no (true or false) represented by a Boolean.
- price: Number Finally, each game has a price – represented by a number.
- When creating a Schema, the types which can be used are String, Number, Date, Buffer, Boolean, ObjectID, and Array.

Mongoose Model

- Models are higher-order constructors that take a schema and create an instance of a document equivalent to records in a relational database. It's the same idea as creating a new Object from a Class in Object Oriented Programming.
- Now we have a Schema that defines the shape of game documents to be added to a collection in MongoDB.
- This Schema must now be compiled down to a **Model**. Here is how we create a Model.

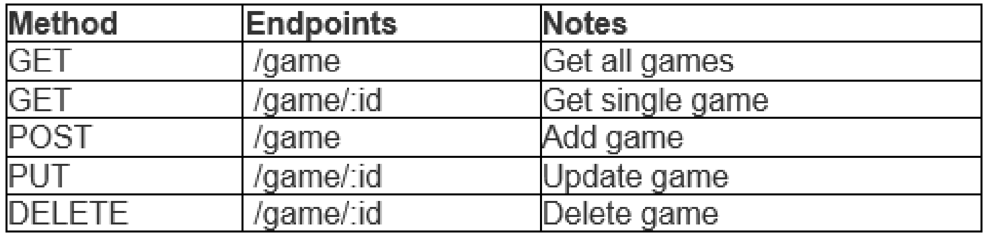


```
module.exports = mongoose.model('Game', gameSchema);
```

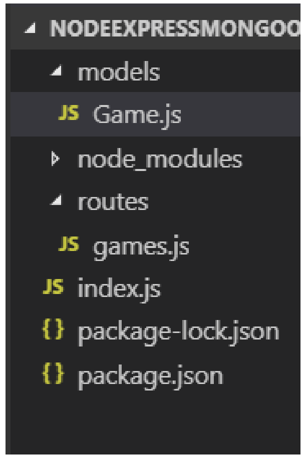
- Mongoose provides the model() method which accepts two arguments.
- The first argument is the singular name of the collection your model is for. Mongoose automatically looks for the plural version of your model name by convention. Therefore, in the code above, the model 'Game' is for the 'games' collection in the database.
- The second argument to the model() method is the schema that defines the shape of documents in this Collection. This gives us a new Game class in our application. This is why in the line of code above, we are assigning the result of the call to mongoose.model() to that Game constant.
- It is capitalized because what is stored here is a Class, not an object.

Create Routes for REST API endpoint

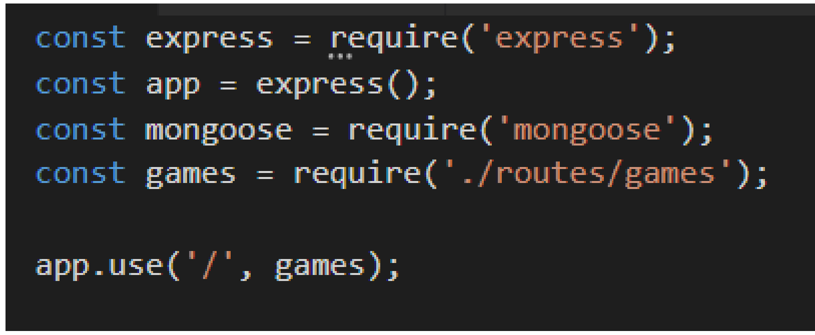
- Our REST API that we build will have following methods (GET, POST, PUT DELETE).
- To create routes, add the JavaScript file games.js to routes folder.
- Import express, router modules, games.js file and mongoose model.
- We use express Router() to create our routes.
- We will call an instance of the express.Router(), apply routes to it, and then tell our application to use those routes, app.use ('/api/games', games).
- Then our routes would be http://localhost:4000/api/games.



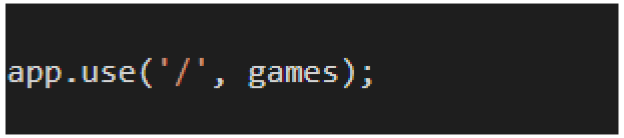
Method	Endpoints	Notes
GET	/game	Get all games
GET	/game/:id	Get single game
POST	/game	Add game
PUT	/game/:id	Update game
DELETE	/game/:id	Delete game



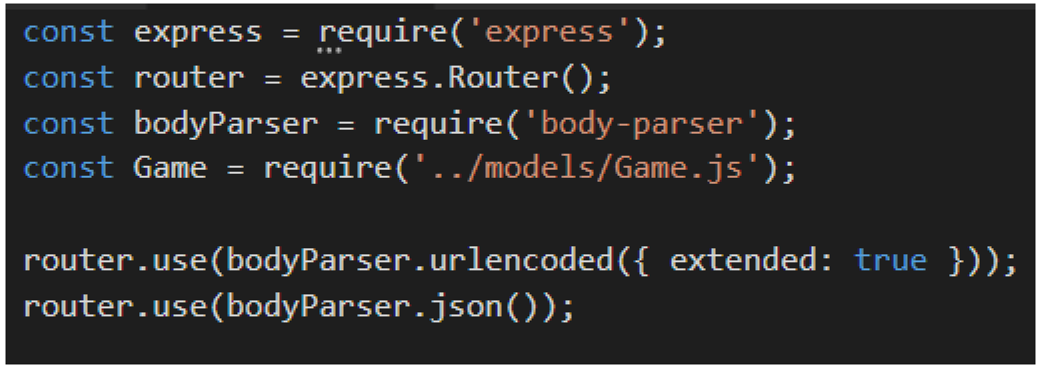
```
◢ NODEEXPRESSMONGOO
  ◢ models
     JS Game.js
  ▷ node_modules
  ◢ routes
     JS games.js
  JS index.js
  {} package-lock.json
  {} package.json
```



```
const express = require('express');
const app = express();
const mongoose = require('mongoose');
const games = require('./routes/games');

app.use('/', games);
```



```
app.use('/', games);
```



```
const express = require('express');
const router = express.Router();
const bodyParser = require('body-parser');
const Game = require('../models/Game.js');

router.use(bodyParser.urlencoded({ extended: true }));
router.use(bodyParser.json());
```

Key Methods of Mongoose

- **Creating documents**

  Model.create()

- **Finding documents**
- Documents can be retrieved through find, findOne and findById. These methods are executed on your Models.

  Model.find(), Model.findOne(), Model.findById(),

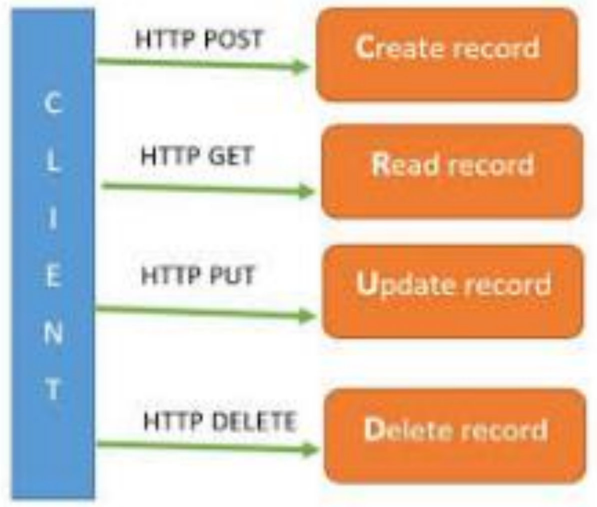
- **Updating documents**
- Documents can be updated using Model.update, Model.findByIdAndUpdate. These methods are executed on your Models.

  Model.update(), Model.findByIdAndUpdate()

- **Deleting documents**
- Documents can be deleted using Model.remove, Model.findByIdAndUpdate. These methods are executed on your Models.
- **Model.remove(),** Model.findByIdAndRemove()

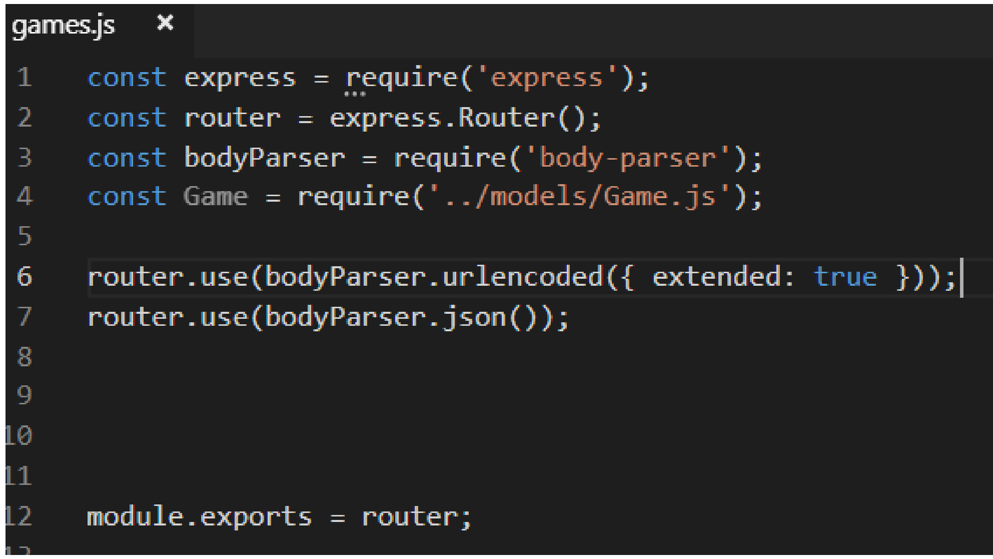
RESTful APIs

- Let's just make a quick intro to REST, to better understand the 4 actions we have on our disposal to interact with our database.
- They are called CRUD stands for **C**reate, **R**ead, **U**pdate and **D**elete.
- Using HTTP requests, we can use the respective action to trigger every of these four CRUD operations.
- **POST** is used to send data to a server — **Create**
- **GET** is used to fetch data from a server — **Read**
- **PUT** is used to send and update data — **Update**
- **DELETE** is used to delete data — **Delete**



Creating Routes

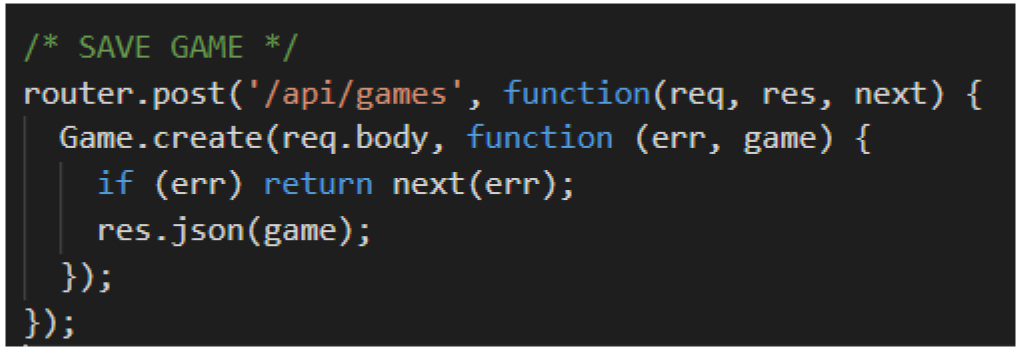
- Now comes the fun part, creating routes for your app and binding them to respective actions.
- You'll be using the express router to create a subset of routes which can be modular and independent from the whole app.
- If you ever need to re-structure your program, you can easily do so because this approach gives you the ability to just *plug it out* from one place and *plug it in* somewhere else.
- body-parser is a module available in express which helps to parses the HTTP request body. This is usually necessary when you need to know more than just the URL being hit, more specifically in the context of a POST, GET, PUT or DELETE HTTP request where the information you want is contained in the body.
- Using body parser allows you to access req.body from within your routes, and use that data for example to create a user in a database.
- Express needs to be able to parse json objects in the body of the request so this middleware turns that on.
- At the bottom of the file you export the router you can use elsewhere if needed. Finally, in our index.js file, we need to require our games.js file.


games.js ×

```
const express = require('express');
const router = express.Router();
const bodyParser = require('body-parser');
const Game = require('../models/Game.js');

router.use(bodyParser.urlencoded({ extended: true }));
router.use(bodyParser.json());




module.exports = router;
```

HTTP POST Requests

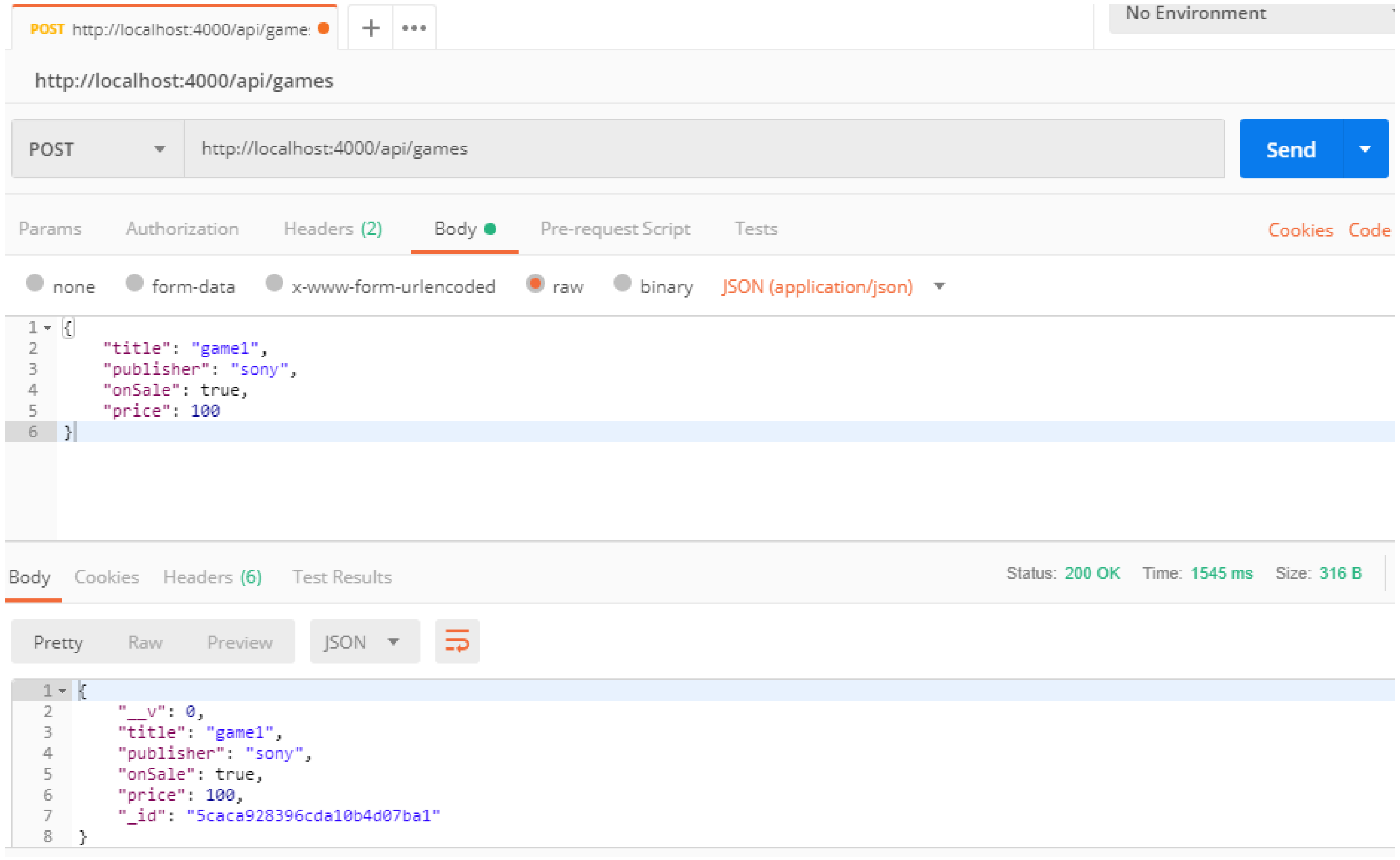
- Now we need to set up the code that will allow our web server to respond to http post requests. This corresponds to the ***Create*** of crud in a rest api.
- We can use a post request to add a new game to the system.
- In this example, we are posting to the collection, or in other words to /api/games using mongoose **create** function.
- Finally, we simply push the new game onto our database using mongoose create() and then send back the game as a response by convention.



```
/* SAVE GAME */
router.post('/api/games', function(req, res, next) {
  Game.create(req.body, function (err, game) {
    if (err) return next(err);
    res.json(game);
  });
});
```

Testing Endpoints With Postman

- The way we can test out sending a post request with a json object in the body of the request is by using **postman**.



http://localhost:4000/api/games

POST http://localhost:4000/api/games

Body: raw JSON (application/json)

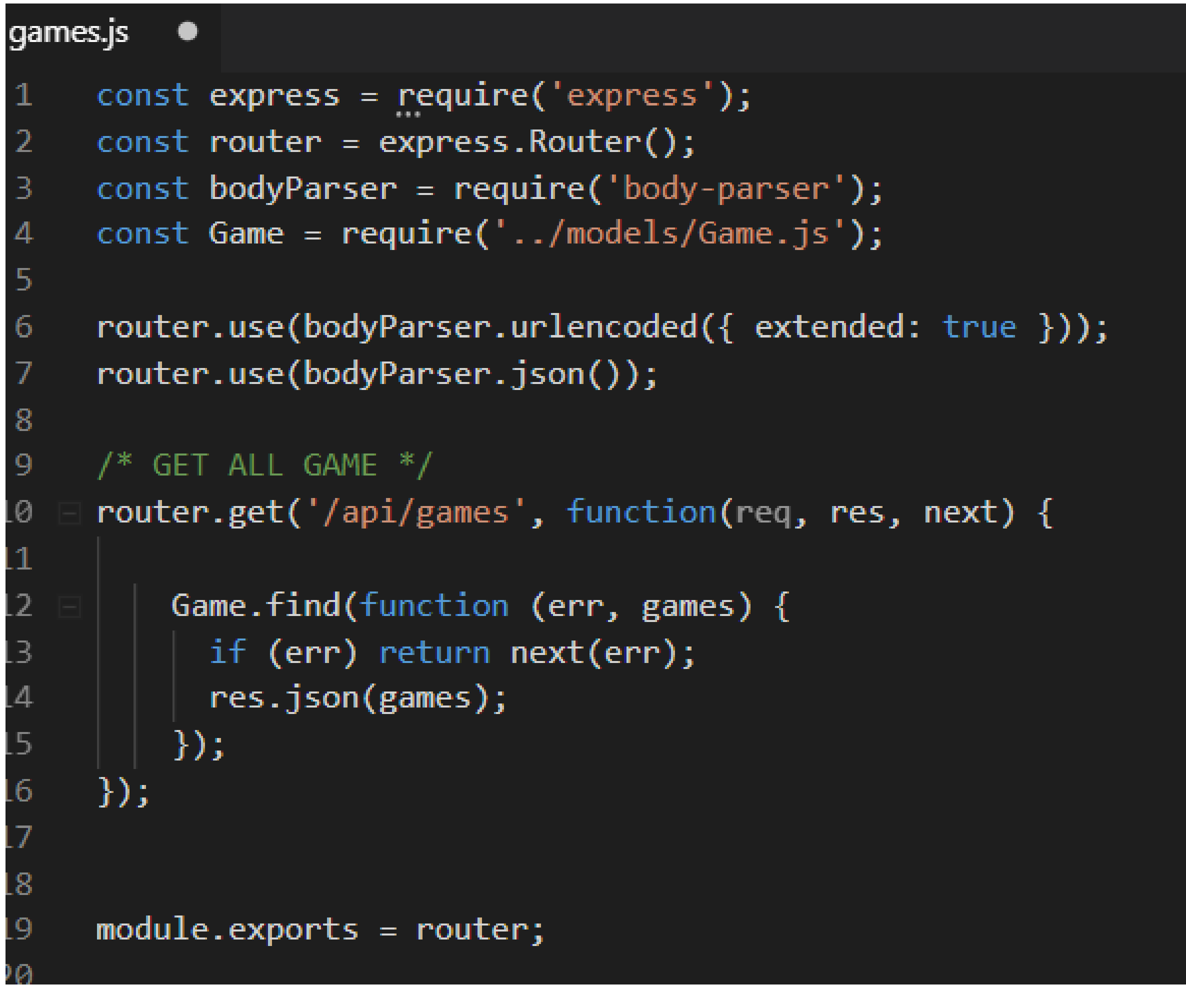
```
{
    "title": "game1",
    "publisher": "sony",
    "onSale": true,
    "price": 100
}
```

Body — Status: 200 OK Time: 1545 ms Size: 316 B

```
{
    "__v": 0,
    "title": "game1",
    "publisher": "sony",
    "onSale": true,
    "price": 100,
    "_id": "5caca928396cda10b4d07ba1"
}
```

HTTP GET Request

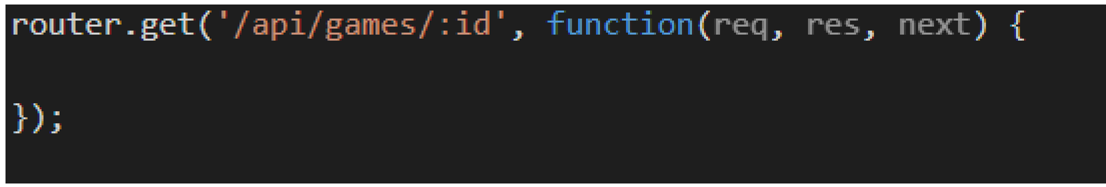
- Now we can set up get requests for fetching games from the server.
- This corresponds to the ***Read*** of crud in a rest api.
- Using the get() available in express router module, we create a get route.
- Now with mongoose find(), we get all the games from the database.
- Typically in the RESTful convention, if you make a get request to the api with no route parameters specified, then you should get back all resources. So if we visit http://localhost:4000/api/games, then we should see all games. This code here should do the trick.
- Test it in postman


games.js

```javascript
const express = require('express');
const router = express.Router();
const bodyParser = require('body-parser');
const Game = require('../models/Game.js');

router.use(bodyParser.urlencoded({ extended: true }));
router.use(bodyParser.json());

/* GET ALL GAME */
router.get('/api/games', function(req, res, next) {

    Game.find(function (err, games) {
      if (err) return next(err);
      res.json(games);
    });
});


module.exports = router;
```

Express Route Parameters

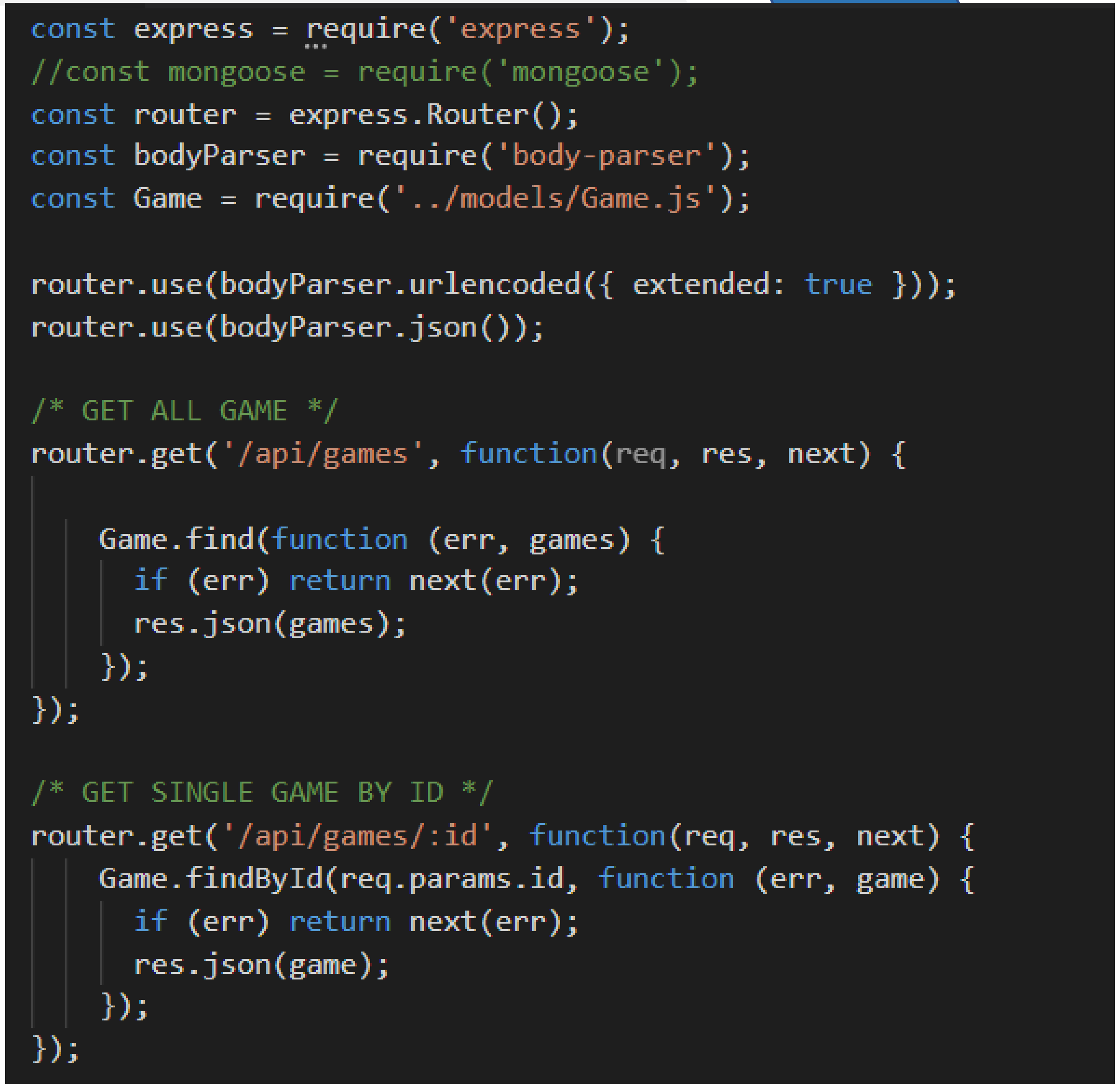
- Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the req.params object, with the name of the route parameter specified in the path as their respective keys.
- To define routes with route parameters, simply specify the route parameters in the path of the route as shown here.
- What this says is that anytime a get request is made to /api/games, anything that comes after that is a dynamic route parameter.
- So for example if we visit http://localhost:4000/api/games/25, then 25 is the route parameter.
- To fetch this in the code, you can use **req.params.id**.
- You can have multiple route parameter in the url.



```
router.get('/api/games/:id', function(req, res, next) {

});
```

HTTP GET Requests

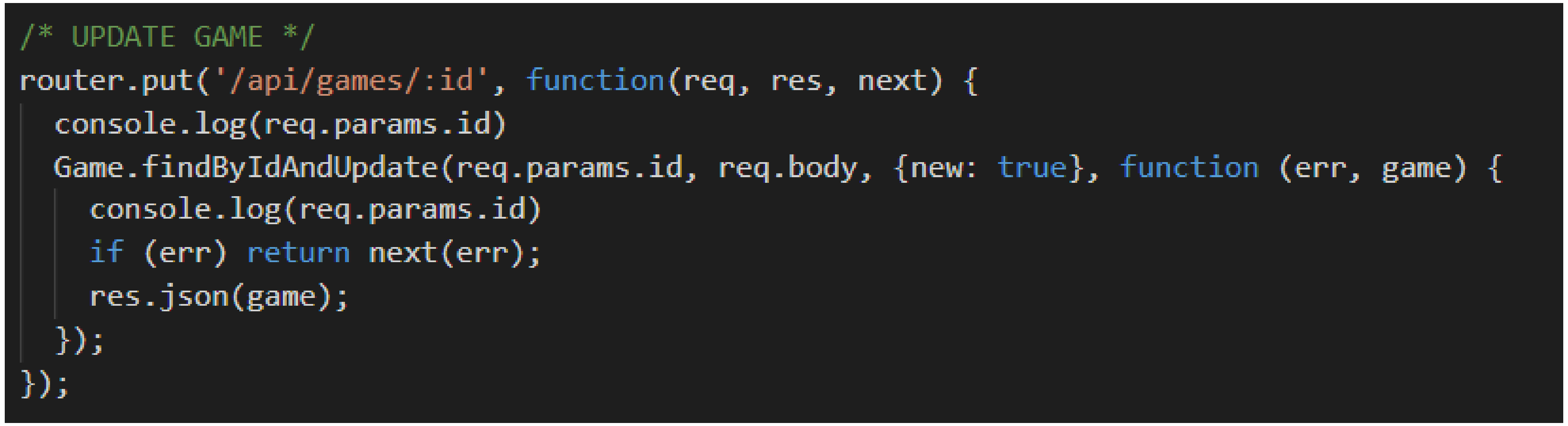
- Now we want to be able to find a specific game only using a route parameter. For this, we can use the mongoose **findById** function.
- Create a route to get a single item from database.
- Test it in postman.



```
const express = require('express');
//const mongoose = require('mongoose');
const router = express.Router();
const bodyParser = require('body-parser');
const Game = require('../models/Game.js');

router.use(bodyParser.urlencoded({ extended: true }));
router.use(bodyParser.json());

/* GET ALL GAME */
router.get('/api/games', function(req, res, next) {

    Game.find(function (err, games) {
      if (err) return next(err);
      res.json(games);
    });
});

/* GET SINGLE GAME BY ID */
router.get('/api/games/:id', function(req, res, next) {
    Game.findById(req.params.id, function (err, game) {
      if (err) return next(err);
      res.json(game);
    });
});
```

HTTP PUT Requests

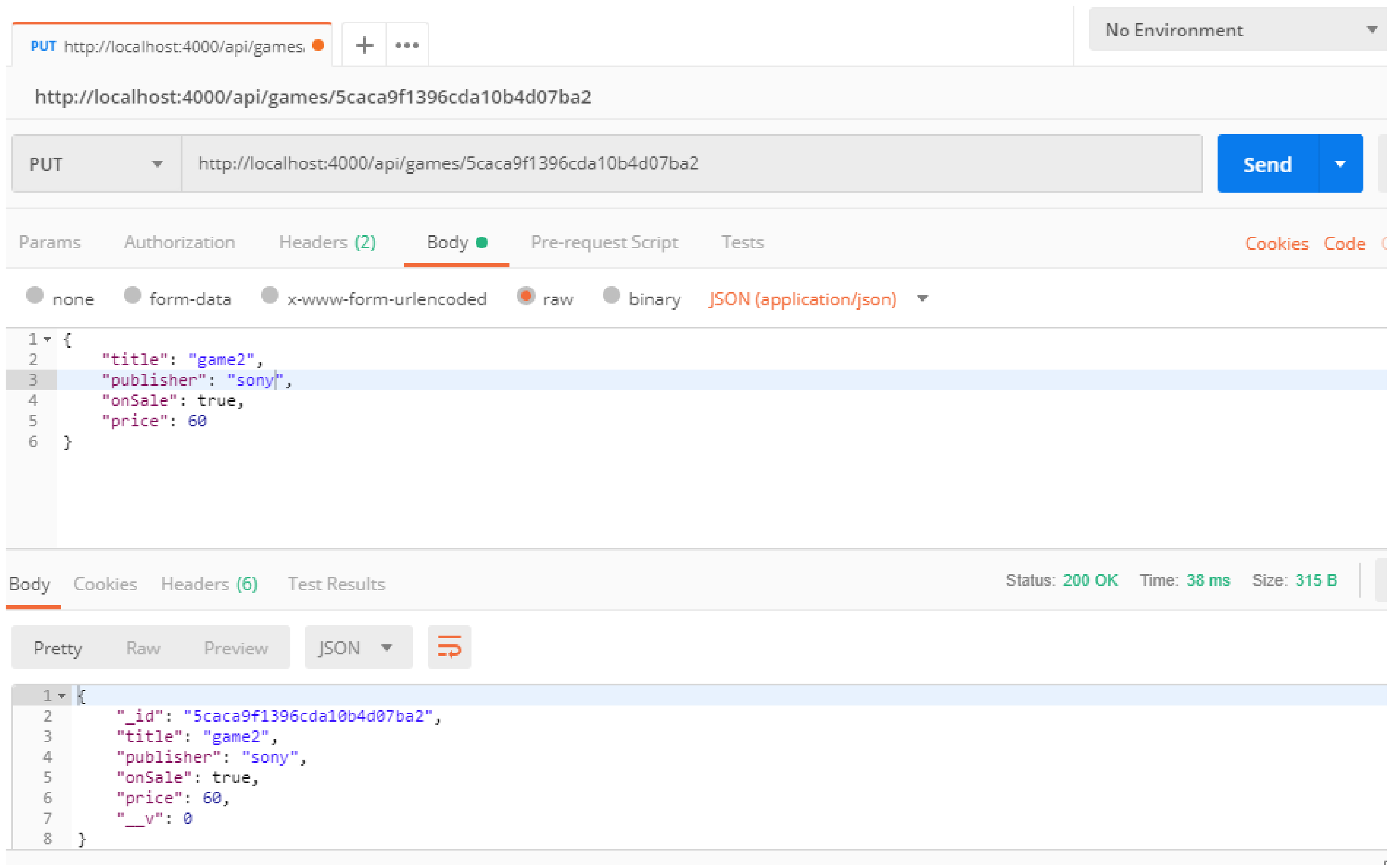
- To update an existing resource on the server you can use a PUT request.
- This corresponds to the ***Update*** of crud in a rest api.
- Let's see how to set up the code to handle a PUT request so we can update a game in the application.



```
/* UPDATE GAME */
router.put('/api/games/:id', function(req, res, next) {
  console.log(req.params.id)
  Game.findByIdAndUpdate(req.params.id, req.body, {new: true}, function (err, game) {
    console.log(req.params.id)
    if (err) return next(err);
    res.json(game);
  });
});
```

Testing Endpoints With Postman

- Let's test it out!
- We will send a PUT request to the server
- We get a response back like we expect.



http://localhost:4000/api/games/5caca9f1396cda10b4d07ba2

PUT http://localhost:4000/api/games/5caca9f1396cda10b4d07ba2

Body: [✓ raw] JSON (application/json)

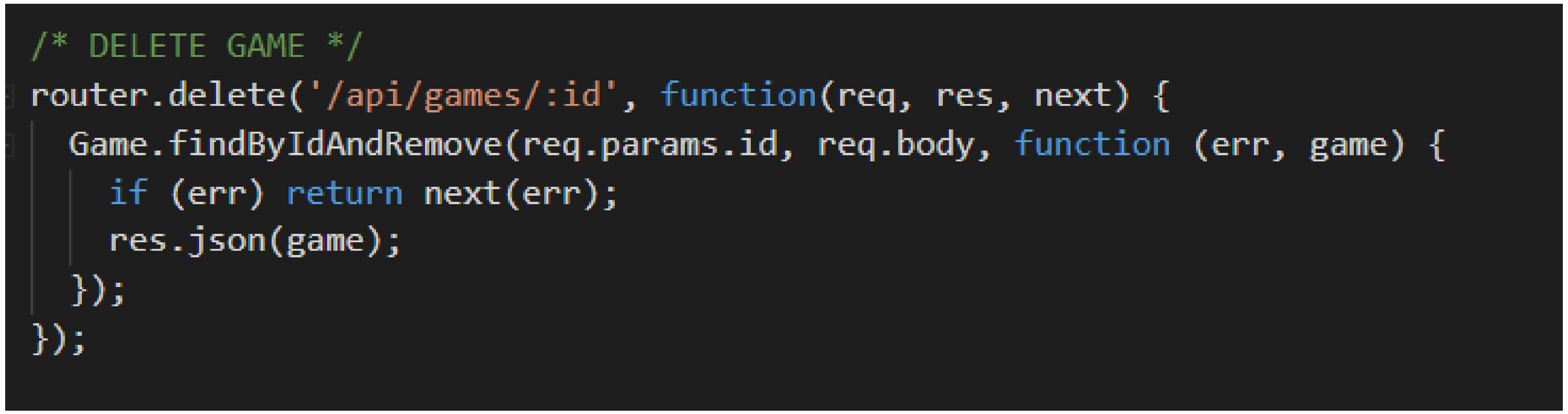
```
{
    "title": "game2",
    "publisher": "sony",
    "onSale": true,
    "price": 60
}
```

Status: 200 OK Time: 38 ms Size: 315 B

```
{
    "_id": "5caca9f1396cda10b4d07ba2",
    "title": "game2",
    "publisher": "sony",
    "onSale": true,
    "price": 60,
    "__v": 0
}
```

HTTP DELETE Requests

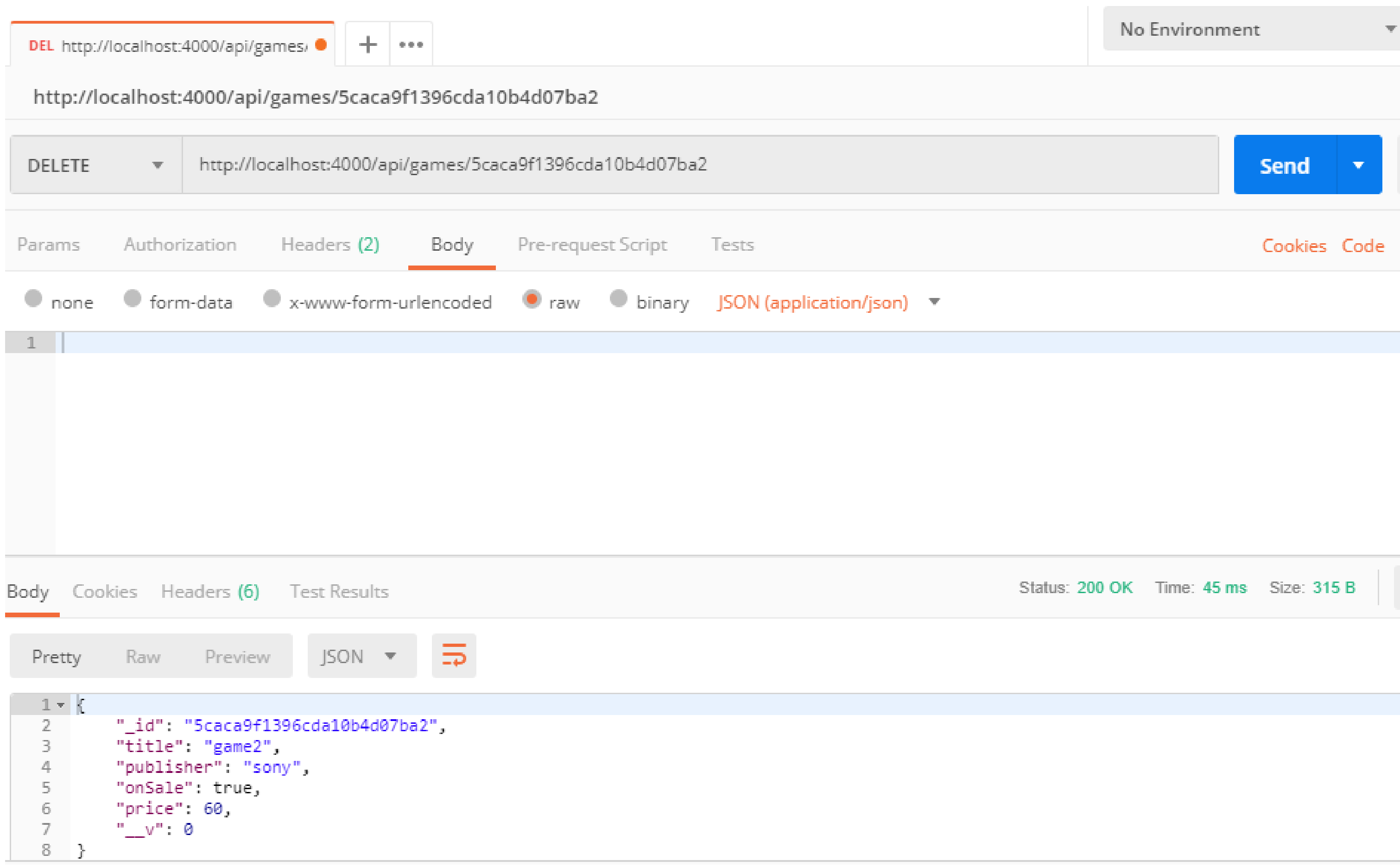
- Finally, we will learn how to implement the ***Delete*** of crud operations in our rest api.
- We can follow a similar logic as updating a course.
- Here is the code we can add.



```
/* DELETE GAME */
router.delete('/api/games/:id', function(req, res, next) {
  Game.findByIdAndRemove(req.params.id, req.body, function (err, game) {
    if (err) return next(err);
    res.json(game);
  });
});
```

Testing Endpoints With Postman

- Now, let's send a DELETE request in postman
- We get back the game which was deleted as a response, which is what we expect.



http://localhost:4000/api/games/5caca9f1396cda10b4d07ba2

DELETE http://localhost:4000/api/games/5caca9f1396cda10b4d07ba2

Body: raw, JSON (application/json)

Status: 200 OK Time: 45 ms Size: 315 B

```
{
    "_id": "5caca9f1396cda10b4d07ba2",
    "title": "game2",
    "publisher": "sony",
    "onSale": true,
    "price": 60,
    "__v": 0
}
```