**Building RESTful APIs Using Express**

**1) Introduction**:

A REST API defines a set of functions which developers can perform requests and receive responses via HTTP protocol such as GET and POST.

**2) RESTful Services**:

RESTful Web Services are basically REST Architecture based Web Services. In REST Architecture everything is a resource. RESTful web services are light weight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.

REST stands for Representational State Transfer. REST is web standards-based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

**HTTP methods**:

Following four HTTP methods are commonly used in REST based architecture.

1. GET − Provides a read only access to a resource.
2. POST − Used to create a new resource.
3. DELETE − Used to remove a resource.
4. PUT − Used to update an existing resource or create a new resource.

**3) Introducing Express**:

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is an open source framework developed and maintained by the Node.js foundation.

Express was developed by TJ Holowaychuk and is maintained by the Node.js foundation and numerous open source contributors.

**Install Express**:

For install express run the following command in console

npm i express

**4) Building Your First Web Server (Using Express)**:

Create a file name index.js in our project root directory. In this file write the following code.

const express = require("express");

const app = express();

app.get("/", (req, res) => {

res.send("Hello World");

});

app.listen(3000, () => console.log("Listening on port 3000..."));

Here, first we have to lode the module "express"

const express = require("express");

Here, require("express") return a function. express() function return an object and by convention we store it in app variable.

const app = express();

Now we have to define a route.

app.get("/", (req, res) => {

res.send("Hello World");

});

* Here, get() method have two parameter "/" and a callback function.
* "/" represent the root of the website
* callback function has two parameters (req, res). req -> Request object have a branch of useful properties that gives us information about incoming request. For more info we have to see Express documentation <http://expressjs.com/> . res -> Response is use to response something to the client.

Finally, we need to listen on a given port.

app.listen(3000, () => console.log("Listening on port 3000..."));

Now go to the console and run "node index.js". Program will run and print " Listening on port 3000..." in console. Go to browser and in the address bar write <http://localhost:3000/> and press enter. In the browser we will see "Hello World"

Now another routes in our application.

const express = require("express");

const app = express();

app.get("/", (req, res) => {

res.send("Hello World");

});

app.get("/api/courses", (req, res) => {

res.send([1, 2, 3, 4, 5]);

});

app.listen(3000, () => console.log("Listening on port 3000..."));

**5) Nodemon**:

Every time we make change in our code, we have to go to the terminal stop the process and start it again. This is not a good practice. For resolve this problem we can use Nodemon.

nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.

nodemon does not require any additional changes to your code or method of development. nodemon is a replacement wrapper for node, to use nodemon replace the word node on the command line when executing your script.

**Install**:

We can install nodemon globally to your system path or install nodemon as a development dependency.

**Globally**:

npm i -g nodemon

**Development dependency**:

npm install --save -dev nodemon

Now if we change anything in our code nodemon restart our server automatically.

**6) Environment Variables**:

Working with environment variables is a great way to configure different aspects of our Node.js application. Many cloud hosts (Heroku, Azure, AWS, now.sh, etc.) and Node.js modules use environment variables. Hosts, for example, will set a PORT variable that specifies on which port the server should listen to properly work. Modules might have different behaviors (like logging) depending on the value of NODE\_ENV variable.

When we deploy our application in a hosting environment, the port is dynamically assigning by the hosting environment. In our application we set the value 3000 which is not working in hosting environment. The way to fixed this is by using an environment variable.

An environment variable is basically a variable that is part of this environment in which a process run. It value is set outside the application.

We can define the environment variable by using process object. The process is a global object. Here we use dynamic port or 3000.

const port = process.env.PORT || 3000;

Now we can replace 3000 with our dynamic port.

**app.js**:

const express = require("express");

const app = express();

app.get("/", (req, res) => {

res.send("Hello World!!!");

});

app.get("/api/courses", (req, res) => {

res.send([1, 2, 3, 4, 5]);

});

const port = process.env.PORT || 3000;

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

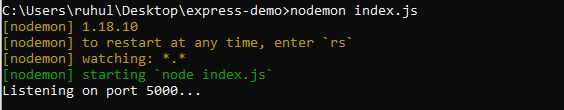
*//Listening on port 3000...*

Now when we run the program, (app.js) we will see the output

*//Listening on port 3000...*

Because we don’t set any environment variable call PORT. Now we set the environment variable PORT as 5000.

Go to console and run the command "set PORT=5000" for set 5000 as port and then run the command "nodemon index.js"



**7) Route Parameters**:

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.

**Example**:

The following code is an example of a very basic route.

var express = require("express");

var app = express();

*// respond with "hello world" when a GET request is made to the homepage*

app.get("/", function(req, res) {

res.send("hello world");

});

Now we implement a route for get a single course.

const express = require("express");

const app = express();

app.get("/", (req, res) => {

res.send("Hello World!!!");

});

app.get("/api/courses", (req, res) => {

res.send([1, 2, 3, 4, 5]);

});

*//implement a route for get a single course*

app.get("/api/courses/:id", (req, res) => {

res.send(req.params.id);

});

const port = process.env.PORT || 3000;

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

*/\**

*in browser : http://localhost:3000/api/courses/1*

*output: 1*

*\*/*

Also, it is possible to have multiple parameters in a route. Suppose we are building a service for a blog. We have a route like this

**Example**:

const express = require("express");

const app = express();

app.get("/", (req, res) => {

res.send("Hello World!!!");

});

app.get("/api/courses", (req, res) => {

res.send([1, 2, 3, 4, 5]);

});

*//implement a route for get a single course*

app.get("/api/posts/:year/:month", (req, res) => {

res.send(req.params);

});

const port = process.env.PORT || 3000;

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

*/\**

*in browser : http://localhost:3000/api/posts/2018/1*

*{"year":"2018","month":"1"}*

*\*/*

With express we can also get query string parameter. This are parameters that we can add in the URL after question mark (?). Now we have to read query parameter.

const express = require("express");

const app = express();

app.get("/", (req, res) => {

res.send("Hello World!!!");

});

app.get("/api/courses", (req, res) => {

res.send([1, 2, 3, 4, 5]);

});

*//implement a route for get a single course*

app.get("/api/posts/:year/:month", (req, res) => {

res.send(req.query);

});

const port = process.env.PORT || 3000;

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

*/\**

*in browser : http://localhost:3000/api/posts/2018/1?sortBy=name*

*{"sortBy":"name"}*

*\*/*

**8) Handling HTTP GET Requests**:

Let’s implement a new endpoint to get a single course from the server.

const express = require("express");

const app = express();

*//define courses array*

const courses = [

{ id: 1, name: "courses\_1" },

{ id: 2, name: "courses\_2" },

{ id: 3, name: "courses\_3" }

];

app.get("/api/courses", (req, res) => {

res.send(courses);

});

*//implement a new endpoint to get a single course from the server*

app.get("/api/courses/:id", (req, res) => {

const course = courses.find(c => c.id === parseFloat(req.params.id));

if (!course) {

res.status(404).send("The course with the given ID was not found.");

}

res.send(course);

});

const port = process.env.PORT || 3000;

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

*/\**

*in browser: http://localhost:3000/api/courses*

*[{"id":1,"name":"courses\_1"},{"id":2,"name":"courses\_2"},{"id":3,"name":"courses\_3"}]*

*in browser: http://localhost:3000/api/courses/1*

*{"id":1,"name":"courses\_1"}*

*in browser: http://localhost:3000/api/courses/10*

*The course with the given ID was not found.*

*\*/*

**9) Handling HTTP POST Requests**:

Now we are working with http post request. Using an http post request, we create a new course.

app.post("/api/courses/", (req, res) => {

});

In this route handler we need to read the course object that should be in the body of the request. Use these properties to create a new course object and then add the course object to our courses array. Now create a new course object.

*//post request*

app.post("/api/courses", (req, res) => {

const course = {

id: courses.length + 1,

name: req.body.name

};

});

Now in order for this line "name: req.body.name" for work, we need to enable parsing of JSON objects in the body of the request. Because by default, this feature is not enabling in express. So on the top after we get the app object, we need to call "express.json()"

const express = require("express");

const app = express();

app.use(express.json());

Here in the line "app.use(express.json())" we are adding a pic of middleware. When we call "express.json()" method, this method return a pic of middleware and then we call "app.use" to use that middleware in the request processing pipeline.

Now back to our new route handler and push the course object to our array. And finally by convention, when we post an object to the server, when the server creates a new object or a new resource, it should be turn that object in the body of the response.

courses.push(course);

res.send(course);

*//post request*

app.post("/api/courses", (req, res) => {

const course = {

id: courses.length + 1,

name: req.body.name

};

courses.push(course);

res.send(course);

});

**10) Calling Endpoints Using Postman**:

To call http services we use a chrome extension call "postman". For add a course follow the following steps.

1. Open postman
2. Select Post
3. Put the URL"<http://localhost:3000/api/courses>"
4. Select body and from list select "row"
5. Select JSON from list
6. Now put a JSON object in body of the request

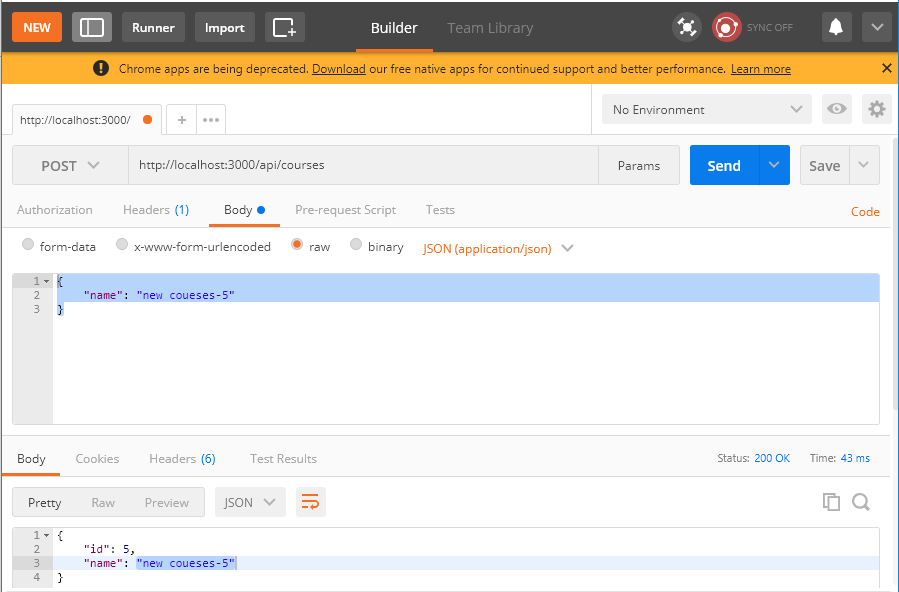
**JSON object**:

{

"name": "new coueses-5"

}

1. Now press "Sent" button
2. Now we will see the "new coueses-5" course added.



**11) Input Validation**:

Here we add input validation in our API. As a security base practice, we should never ever trust what the client sends us. We should always validate the input.

Here we are working with a simple object with only one object that is name. Here we add some validation logic.

If the "req.body.name" is not exist or the length of the name is less than 3 character ("req.body.name.length") then we will return an error to the client.

if (!req.body.name || req.body.name.length < 3) {

}

The RESTful convention is to return and response with the HTTP status code of 400. Here 400 means bad request. And then finally return here because we don’t want to rest of the function is executed.

if (!req.body.name || req.body.name.length < 3) {

*//400 bad request*

res.status(400).send("Name is required and should be minimum 3 characters");

return;

}

**Example**:

const express = require("express");

const app = express();

app.use(express.json());

*//define courses array*

const courses = [

{ id: 1, name: "courses\_1" },

{ id: 2, name: "courses\_2" },

{ id: 3, name: "courses\_3" }

];

app.get("/api/courses", (req, res) => {

res.send(courses);

});

*//implement a new endpoint to get a single course from the server*

app.get("/api/courses/:id", (req, res) => {

const course = courses.find(c => c.id === parseFloat(req.params.id));

if (!course) {

res.status(404).send("The course with the given ID was not found.");

}

res.send(course);

});

*//post request*

app.post("/api/courses", (req, res) => {

*//validation logic*

if (!req.body.name || req.body.name.length < 3) {

*//400 bad request*

res.status(400).send("Name is required and should be minimum 3 characters");

return;

}

const course = {

id: courses.length + 1,

name: req.body.name

};

courses.push(course);

res.send(course);

});

const port = process.env.PORT || 3000;

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

Now it’s a simple request. In the real world when we are working with a complex object, we have to write complex validation logic. For make the validation simple there is a module in Node. The name of the module is "joi". Install the exact version

npm i joi@13.1.0

Now load the module. The return of the module is a class. As a best practice put all of us require call at the top of the file. Now we have Joi class.

const Joi = require("joi")

Now back on our route handler. Now with JOI first we have to define a schema. A schema defines the shape of our objects? What property we have in the objects? What is the type of this property? Do we have an email or name? what is the minimum and maximum character. This is the job of a schema.

First, we have to define a schema.

*//define a schema*

const schema = {

name: Joi.string()

.min(3)

.require()

};

Now we validate it and store the result in a variable. Log the result in the console.

const result = Joi.validate(req.body, schema);

*//post request*

app.post("/api/courses", (req, res) => {

*//define a schema*

const schema = {

name: Joi.string().min(3).require()

};

*//validate joi*

const result = Joi.validate(req.body, schema);

console.log(result);

*//validation logic*

if (!req.body.name || req.body.name.length < 3) {

*//400 bad request*

res.status(400).send("Name is required and should be minimum 3 characters");

return;

}

const course = {

id: courses.length + 1,

name: req.body.name

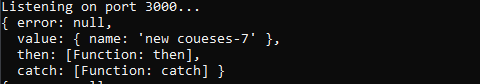
};

courses.push(course);

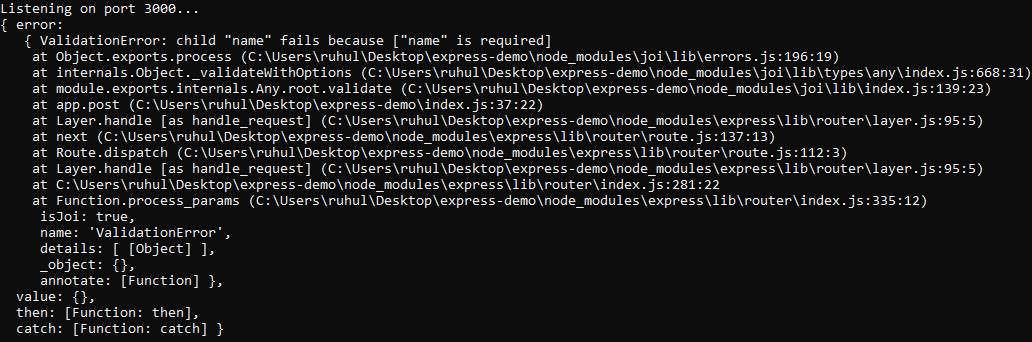
res.send(course);

});

Now go back to the postman and add a course. In the terminal we see the following result.



Here we have two properties, error and value. Only one of this can have a value. In this case since we pass a valid course object, hence here have the value here as the value property. And the error is null. If we sent an invalid object the value is null and the error will be set.



Here, in the result object the error property is set to and object that has ValidationError

"ValidationError: child "name" fails because ["name" is required] "

And the object is empty {}

Now instead of this manual validation logic we can cheek the value of result that error property.

*//validation logic*

if (result.error) {

*//400 bad request*

res.status(400).send(result.error);

return;

}

Now if we send an empty object from postman, we will get the following error.

{

"isJoi": true,

"name": "ValidationError",

"details": [

{

"message": "\"name\" is required",

"path": [

"name"

],

"type": "any.required",

"context": {

"key": "name",

"label": "name"

}

}

],

"\_object": {}

}

Now this object is too complex to send to the client, We can simplify this.

*//validation logic*

if (result.error) {

*//400 bad request*

res.status(400).send(result.error.details[0].message);

return;

}

Now we get the result

"name" is required

Now if we send an object from postman which name is less than three charecter, we will get the following error.

"name" length must be at least 3 characters long

**12) Handling HTTP PUT Requests**:

For update a course we have to do the following things.

1. Look up the course
2. If not exist, return 404 (404 => resource not found)
3. Validate the course
4. If not valid return 400 (400 => bad request)
5. Update course
6. Return the update course

**First part (1 and 2)**:

Now we look up a course and if the course not exist return a 404 (resource not found) error.

const course = courses.find(c => c.id === parseFloat(req.params.id));

if (!course) {

res.status(404).send("The course with the given ID was not found" );

return;

}

});

**Second part (3 and 4)**:

The second part is validation. For working with validation, we need the schema. But there is a problem here. In this case we have a very simple schema, but when we are working a complex object with many properties? Then our validation logic would be duplicated in two different handlers.

const schema = {

name: Joi.string()

.min(3)

.required()

};

const result = Joi.validate(req.body, schema);

if (result.error) {

res.status(400).send(result.error.details[0].message);

return;

}

**Third part (5, 6 and 7)**:

Now we have a course object and we want to update its property.

course.name = req.body.name;

res.send(course);

Now this is the process how we handle a http put request.

Now we here the validation is duplicated. For make the validation is more simplify separate this.

**Separate validation logic**:

function validateCourse(course) {

const schema = {

name: Joi.string()

.min(3)

.required()

};

return Joi.validate(course, schema);

}

**Object Destructur**:

The two most used data structures in JavaScript are Object and Array.

Objects allow us to pack many pieces of information into a single entity and arrays allow us to store ordered collections. So, we can make an object or an array and handle it as a single entity, or maybe pass it to a function call.

Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes they are more convenient. Destructuring also works great with complex functions that have a lot of parameters, default values, and soon we’ll see how these are handled too.

Now by using object destructor.

We can make this code a little bit cleaner and shorter by using Object Destructuring feature in modern JavaScript.

const { error } = validateCourse(req.body);

Now move all the validate logic into a separate method.

*//validate course*

function validateCourse(course) {

const schema = {

name: Joi.string()

.min(3)

.required()

};

return Joi.validate(course, schema);

}

**Update route**:

*//update course*

app.put("/api/courses/:id", (req, res) => {

const course = courses.find(c => c.id === parseInt(req.params.id));

if (!course) {

res.status(404).send("The course with the given ID was not found");

}

const { error } = validateCourse(req.body);

if (error) {

res.status(400).send(result.error.details[0].message);

return;

}

course.name = req.body.name;

res.send(course);

});

Now we can update an object. For update an object do the following thing

1. Go to postman
2. Select put method
3. In the url box select an id <http://localhost:3000/api/courses/1>
4. Now put a valid name on the body and press Send button
5. For see the change, open new tab and hit the link <http://localhost:3000/api/courses>
6. We will see all the object list include our changes.

**13) Handling HTTP Delete Requests**:

Now we handle http delete request. The HTTP DELETE request method deletes the specified resource. For delete a course we have to do the following thing.

1. Look up the course
2. Not existing, return 404
3. Delete course
4. Return the same course (Response to the client)

**Step-1: Look up the course and if not exist return 404**:

*//Look up the course*

*//Not existing, return 404*

const course = courses.find(c => c.id === parseInt(req.params.id));

if (!course) {

res.status(404).send("The course with the given ID was not found");

}

**Step-2: Delete course**:

Now for delete a course, first we have to find the index of the course in our courses array.

*//Delete course*

const index = courses.lastIndexOf(course);

courses.splice(index, 1);

**Step-3: Return the same course (Response to the client)**:

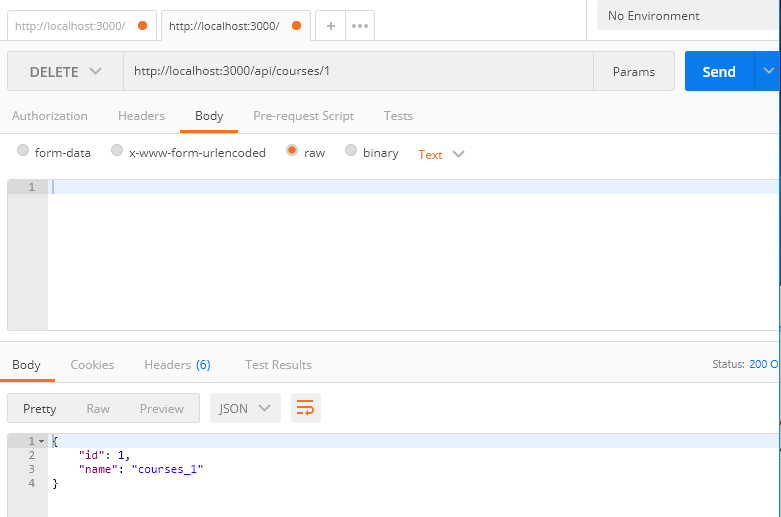
Now we the course is deleted and we have to display the course to the client that we are deleted.

*//Return the same course*

res.send(course);

Now go to postman and do the following thing to delete a course.

1. Open postman
2. Select "DELETE"
3. In URL write "http://localhost:3000/api/courses/1" "DELETE" and press send
4. Course 1 is deleted and it display in body section.



**Delete Request**:

*//DELETE request*

app.delete("/api/courses/:id", (req, res) => {

*//Look up the course*

*//Not existing, return 404*

const course = courses.find(c => c.id === parseInt(req.params.id));

if (!course) {

res.status(404).send("The course with the given ID not found");

}

*//Delete course*

const index = courses.lastIndexOf(course);

courses.splice(index, 1);

*//Return the same course*

res.send(course);

});

**Fix Bug**:

Here we have three bugs in our code. If we don’t have a course in the given id then we return a 404 message to the client. But this point we should exist this rout handler otherwise the rest of the code will be executed. So, lets implement the route handler

if (!course) {

res.status(404).send("The course with the given ID not found");

return;

}

Or in sort way

if (!course)

return res.status(404).send("The course with the given ID not found");

To make this code cleaner let’s use the same technique in case we have an invalid request for put request.

*//PUT request*

app.put("/api/courses/:id", (req, res) => {

const course = courses.find(c => c.id === parseInt(req.params.id));

if (!course) {

res.status(404).send("The course with the given ID was not found");

}

const { error } = validateCourse(req.body);

if (error) return res.status(400).send(error.details[0].message);

course.name = req.body.name;

res.send(course);

});

Do the same issue in all cases

**14) Project- Build the Genres API**:

04 Building RESTful APIs Using Express