TECHNOLOGY 13 MINUTE READ

# Creating a Secure REST API in Node.js

MARCOS HENRIQUE DA SILVA
Marcos has 15+ years in IT and development. His passions include REST architecture, Agile development methodology, and JS.

Consider all the software that you use in your life. It doesn't matter if you're a developer or a regular user that just casually browses the internet and checks up on social networks. Almost all software that you can identify uses some form of API.

APIs (application programming interfaces) enable software applications to communicate with other pieces of software consistently. Either internally to connect to a component of the application or externally to connect to a service.

Using API-based components and services in development is also a great way of maintaining scalability and productivity as it enables you to develop multiple applications based off of modular and reusable components, allowing scalability and facilitating maintenance.

We should also take into account that multiple online services have front-facing APIs and that you can use them to easily integrate things like social media logins, credit card payments, behavior tracking, and many other functionalities.

Implementing such a variety of services via their APIs can be greatly facilitated by using a common platform for communication, a standardized protocol shared and supported by all of them. We'll be using REST for that.

REST stands for REpresentational State Transfer and is used to access and manipulate data using several stateless operations. These operations are integral to the HTTP protocol and represent an essential CRUD functionality (Create, Read, Update, Delete).

The HTTP operations available are:

- POST (create a resource or generally provide data)
- GET (retrieve an index of resources or an individual resource)
- PUT (create or replace a resource)
- PATCH (update/modify a resource)
- DELETE (remove a resource)

TRENDING ARTICLES

ENGINEERING > WEB FRONT-END

How to Approach Modern WordPress Development (Part 1)

ENGINEERING > WEB FRONT-END

A Look at JavaScript's Future

ENGINEERING > DATA SCIENCE AND D..

An Intro to SQL Window Functions

ENGINEERING > TECHNOLOGY

Using the above-listed operations and a resource name as an address, we can build a REST API by basically creating an endpoint for each operation. And by implementing the pattern, we will have a stable and easily understandable foundation enabling us to evolve the code rapidly and maintain it afterward. As mentioned before, the same foundation will be used to integrate third-party features, most of which likewise use REST APIs, making such integration faster.

While a multitude of platforms and programming languages can be used to build a REST API, in this article, we will be focusing on Node.js.

Node.js is a JavaScript runtime environment that runs server-side. Within that environment, we can use JavaScript to build our software, our REST APIs, and invoke external services through their APIs. This fact is especially convenient for developers who are crossing over from front-end development as they should already be familiar with JavaScript, making the transition more natural. It also has the bonus of unifying all of the codebase under a single programming language.

As an infrastructure, Node.js is designed for building scalable network applications. It is also relatively simple to set up on a local machine, and you can have your server running with a few lines of code. Even some cloud services such as AWS (Amazon Web Services) run Node.js, enabling you to run a serverless application.

Now, of course, nothing is quite so clear-cut in the real world, and development communities are always ripe with discussions about which programming language is the best and which environment is the most suitable for a specific purpose, but I'll leave that up to you. If you're curious, though, you can also find articles on REST APIs in ASP.NET Core, Laravel (PHP), Bottle (Python), and many others.

For now, let's start creating our secure REST API using Node.js!

In this tutorial, we are going to create a pretty common but practical REST API for a resource called `users`.

When Does it Make Sense to Use Google BigQuery?

SEE OUR RELATED TALENTS

Nod

Our resource will have the following basic structure:

- id (an auto-generated UUID)
- firstName
- lastName
- email
- password
- permissionLevel (used to control user's permissions)

And we will create the following operations for that resource:

- [POST] endpoint/users
- [GET] endpoint/users (list users)
- [GET] endpoint/users/:userId (get specific user)
- [PATCH] endpoint/users/:userId (update the data for the specified user)
- [DELETE] endpoint/users/:userId (remove the specified user)

We will also be using JWT (JSON Web Token) for access tokens, and to that end, we will create another resource called `auth` that will expect user email and password and in return will generate the token used for authentication on certain operations. If you are interested in knowing more about this, Dejan Milosevic wrote a great article on JWT for secure REST applications in Java a while back; the principles are the same.

**Related:** Is It Time to Use Node 8?

## Setup

First of all, make sure that you have the latest Node.js version installed. For this article, I'll be using version 8.11.2 from nodejs.org. Version 8 came with some neat improvements (you can read the more here if you're interested).

Next, make sure that you have MongoDB installed, or install it from www.mongodb.com.

Create a folder that we'll be using for our project and name it `simple-rest-api`.

Open up a terminal (or a git CLI console) in that folder and run `npm init` to create the package.json file for the project.

We'll also be using Express on this project. It is a decent Node.js framework with some built-in add-ons that'll speed up our development.

To keep the tutorial focused and on-topic, here's the package.json with the appropriate dependencies.

```
{
 "name": "rest-api-tutorial",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
 },
 "repository": {
   "type": "git",
   "url": "git+https://github.com/makinhs/rest-api-tutorial.git"
 },
 "author": "",
 "license": "ISC",
 "bugs": {
   "url": "https://github.com/makinhs/rest-api-tutorial/issues"
 },
 "homepage": "https://github.com/makinhs/rest-api-tutorial#readme",
 "dependencies": {
   "body-parser": "1.7.0",
   "express": "^4.8.7",
   "jsonwebtoken": "^7.3.0",
   "moment": "^2.17.1",
   "moment-timezone": "^0.5.13",
   "mongoose": "^5.1.1",
   "node-uuid": "^1.4.8",
   "swagger-ui-express": "^2.0.13",
   "sync-request": "^4.0.2"
 }
}
```

Paste the code into the package.json file and after you save it, run `npm install`. Congratulations, you now have all of the dependencies and setup required to run our simple REST API.

Our project will contain three module folders:

- “common” (handling all shared services and information between user modules)

- "users" (everything regarding users)
- "auth" (handle the flow to generate JWT and login flow)

## Creating the User Module

We will be using Mongoose, an ODM (object data modeling) library for MongoDB, to create the user model within the user schema.

First, we need to create the schema in `/users/models/users.model.js`:

```
const userSchema = new Schema({
   firstName: String,
   lastName: String,
   email: String,
   password: String,
   permissionLevel: Number
});
```

Once we define the schema, we can easily attach the schema to the user model.

```
const userModel = mongoose.model('Users', userSchema);
```

After that, we can use this model to implement all the CRUD operations that we want within our endpoints.

Let's start with the "create user" operation by defining the route in `users/routes.config.js`:

```
app.post('/users', [
   UsersController.insert
]);
```

At this point, we can test our Mongoose model by running the server and sending a `POST` request to `/users` with some JSON data:

```
{
   "firstName" : "Marcos",
   "lastName" : "Silva",
   "email" : "marcos.henrique@toptal.com",
   "password" : "s3cr3tp4sswo4rd"
}
```

We can use the controller to hash the password appropriately in `/users/controllers/users.controller.js`:

```
exports.insert = (req, res) => {
   let salt = crypto.randomBytes(16).toString('base64');
   let hash = crypto.createHmac('sha512',salt)
                                    .update(req.body.password)
                                    .digest("base64");
   req.body.password = salt + "$" + hash;
   req.body.permissionLevel = 1;
   UserModel.createUser(req.body)
       .then((result) => {
           res.status(201).send({id: result._id});
       });
};
```

At this point the result of a valid post will be just the id from the created user: `{ "id": "5b02c5c84817bf28049e58a3" }`

And we need to add the `createUser` method to the model in `users/models/users.model.js`:

```
exports.createUser = (userData) => {
    const user = new User(userData);
    return user.save();
};
```

All set, we need to see if the user exists. For that, we are going to implement the get user by id feature for the following endpoint: `users/:userId`.

First, we create a route in `/users/routes/config.js`:

```
app.get('/users/:userId', [
    UsersController.getById
]);
```

Then we create the controller in `/users/controllers/users.controller.js`:

```

exports.getById = (req, res) => {
   UserModel.findById(req.params.userId).then((result) => {
       res.status(200).send(result);
   });
};

```

And finally add the `findById` method to the model in `/users/models/users.model.js`:

```
exports.findById = (id) => {
    return User.findById(id).then((result) => {
        result = result.toJSON();
        delete result._id;
        delete result.__v;
        return result;
    });
};

```

The response will be like this:

```
{
   "firstName": "Marcos",
   "lastName": "Silva",
   "email": "marcos.henrique@toptal.com",
   "password":
"Y+XZEaR7J8xAQCc37nf1rw==$p8b5ykUx6xpC6k8MryDaRmXDxncLumU9mEVabyLdpotO66Qjh0igVOVerdqAh+CUQ4n/E0z48mp8SDTpX2i

   "permissionLevel": 1,
   "id": "5b02c5c84817bf28049e58a3"
}
```

Note that we can see the hashed password. For this tutorial, we are showing the password, but the obvious best practice is never to reveal the password, even if it has been hashed. Another thing we can see is the `permissionLevel`, which we will use to handle the user permissions later on.

Repeating the pattern laid out above we can now add the functionality to update the user. We will use the `PATCH` operation since it will enable us to send only the fields we want to change. The route will, therefore, be `PATCH to /users/:userid` and we'll be sending any fields we want to change. We will also need to implement some extra validation since changes should be restricted to the user in question or an admin, and only an admin should be able to change the `permissionLevel`. We'll skip that for now and get back to it once we implement the auth module. For now, our controller will look like this:

```
exports.patchById = (req, res) => {
   if (req.body.password){
       let salt = crypto.randomBytes(16).toString('base64');
       let hash = crypto.createHmac('sha512', salt).update(req.body.password).digest("base64");
       req.body.password = salt + "$" + hash;
   }
   UserModel.patchUser(req.params.userId, req.body).then((result) => {
           res.status(204).send({});
   });
};
```

By default, we will send an HTTP code 204 with no response body to indicate that the request was successful.

And we'll need to add the `patchUser` method to the model:

```
exports.patchUser = (id, userData) => {
    return new Promise((resolve, reject) => {
        User.findById(id, function (err, user) {
            if (err) reject(err);
            for (let i in userData) {
                user[i] = userData[i];
            }
            user.save(function (err, updatedUser) {
                if (err) return reject(err);
                resolve(updatedUser);
            });
        });
    })
};
```

The user list will be implemented as a `GET` at `/users/` by the following controller:

```
exports.list = (req, res) => {
   let limit = req.query.limit && req.query.limit <= 100 ? parseInt(req.query.limit) : 10;
   let page = 0;
   if (req.query) {
       if (req.query.page) {
           req.query.page = parseInt(req.query.page);
           page = Number.isInteger(req.query.page) ? req.query.page : 0;
       }
   }
   UserModel.list(limit, page).then((result) => {
       res.status(200).send(result);
   })
};
```

The corresponding model method will be:

```
exports.list = (perPage, page) => {
    return new Promise((resolve, reject) => {
        User.find()
            .limit(perPage)
            .skip(perPage * page)
            .exec(function (err, users) {
                if (err) {
                    reject(err);
                } else {
                    resolve(users);
                }
            })
    });
};
```

The resulting list response will have the following structure:

```
[
   {
       "firstName": "Marco",
       "lastName": "Silva",
       "email": "marcos.henrique@toptal.com",
       "password":
"z4tS/DtiH+0Gb4J6QN1K3w==$al6sGxKBKqxRQkDmhnhQpEB6+DQgDRH2qr47BZcqLm4/fphZ7+a9U+HhxsNaSnGB2l05Oem/BLIOkbtOuw1

       "permissionLevel": 1,
       "id": "5b02c5c84817bf28049e58a3"
   },
   {
       "firstName": "Paulo",
       "lastName": "Silva",
       "email": "marcos.henrique2@toptal.com",
       "password":
"wTsqO1kHuVisfDIcgl5YmQ==$cw7RntNrNBNw3MO2qLbx959xDvvrDu4xjpYfYgYMxRVDcxUUEgulTlNSBJjiDtJ1C85YimkMlYruU59rx2z

       "permissionLevel": 1,
       "id": "5b02d038b653603d1ca69729"
   }
]
```

And the last part to be implemented is the `DELETE` at `/users/:userId`.

Our controller for deletion will be:

```
exports.removeById = (req, res) => {
   UserModel.removeById(req.params.userId)
       .then((result)=>{
           res.status(204).send({});
       });
};
```

Same as before, the controller will return HTTP code 204 and no content body as confirmation.

The corresponding model method should look like this:

```
exports.removeById = (userId) => {
    return new Promise((resolve, reject) => {
        User.remove({_id: userId}, (err) => {
            if (err) {
                reject(err);
            } else {
                resolve(err);
            }
        });
    });
};
```

We now have all the necessary operations for manipulating the user resource, and we're done with the user controller. The main idea of this code is to give you the core concepts of using the REST pattern. We'll need to return to this code to implement some validations and permissions to it, but first, we'll need to start building our security. Let's create the auth module.

## Creating the Auth Module

Before we can secure the `users` module by implementing the permission and validation middleware, we'll need to be able to generate a valid token for the current user. We will generate a JWT in response to the user providing a valid email and password. JWT is a remarkable JSON web token that you can use to have the user securely make several requests without validating repeatedly. It usually has an expiration time, and a new token is recreated every few minutes to keep the communication secure. For this tutorial, though, we will forgo refreshing the token and keep it simple with a single token per login.

First, we will create an endpoint for `POST` requests to `/auth` resource. The request body will contain the user email and password:

```
{
   "email" : "marcos.henrique2@toptal.com",
   "password" : "s3cr3tp4sswo4rd2"
}
```

Before we engage the controller we should validate the user in `/authorization/middlewares/verify.user.middleware.js`:

```
exports.isPasswordAndUserMatch = (req, res, next) => {
   UserModel.findByEmail(req.body.email)
       .then((user)=>{
           if(!user[0]){
               res.status(404).send({});
           }else{
               let passwordFields = user[0].password.split('$');
               let salt = passwordFields[0];
               let hash = crypto.createHmac('sha512', salt).update(req.body.password).digest("base64");
               if (hash === passwordFields[1]) {
                   req.body = {
                       userId: user[0]._id,
                       email: user[0].email,
                       permissionLevel: user[0].permissionLevel,
                       provider: 'email',
                       name: user[0].firstName + ' ' + user[0].lastName,
                   };
                   return next();
               } else {
                   return res.status(400).send({errors: ['Invalid email or password']});
               }
           }
       });
};
```

Having done that we can move on to the controller and generate the JWT:

```
exports.login = (req, res) => {
   try {
       let refreshId = req.body.userId + jwtSecret;
       let salt = crypto.randomBytes(16).toString('base64');
       let hash = crypto.createHmac('sha512', salt).update(refreshId).digest("base64");
       req.body.refreshKey = salt;
       let token = jwt.sign(req.body, jwtSecret);
       let b = new Buffer(hash);
       let refresh_token = b.toString('base64');
       res.status(201).send({accessToken: token, refreshToken: refresh_token});
   } catch (err) {
       res.status(500).send({errors: err});
   }
};
```

Even though we won't be refreshing the token in this tutorial, the controller has been set up to enable such generation to make it easier to implement it in subsequent development.

All we need now is to create the route and invoke the appropriate middleware in `/authorization/routes.config.js`:

```
    app.post('/auth', [
        VerifyUserMiddleware.hasAuthValidFields,
        VerifyUserMiddleware.isPasswordAndUserMatch,
        AuthorizationController.login
    ]);
```

The response will contain the generated JWT in the accessToken field:

```
{
   "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI1YjAyYzVjODQ4MTdiZjI4MDQ5ZTU4YTMiLCJlbWFpbCI6Im1hcmNvcy5
i44VQlUEWP3YIAYXVO-74803v1mu-y9QPUQ5VY",
   "refreshToken":
"U3BDQXBWS3kyaHNDaGJNanlJTlFkSXhLMmFHMzA2NzRsUy9Sd2J0YVNDTmUva0pIQ0NwbTJqOU5YZHgxeE12NXVlOUhnMzBWMGNyWmdOTUhS

}
```

Having created the token, we can use it inside the `Authorization` header using the form `Bearer ACCESS_TOKEN`.

## Creating Permissions and Validations Middleware

The first thing we should define is who can use the `users` resource. These are the scenarios that we'll need to handle:

- Public for creating users (registration process). We will not use JWT for this scenario.
- Private for the logged-in user and for admins to update that user.
- Private for admin only for removing user accounts.

Having identified these scenarios, we will first require a middleware that always validates the user if they are using a valid JWT. The middleware in `/common/middlewares/auth.validation.middleware.js` can be as simple as:

```
exports.validJWTNeeded = (req, res, next) => {
    if (req.headers['authorization']) {
        try {
            let authorization = req.headers['authorization'].split(' ');
            if (authorization[0] !== 'Bearer') {
                return res.status(401).send();
            } else {
                req.jwt = jwt.verify(authorization[1], secret);
                return next();
            }
        } catch (err) {
            return res.status(403).send();
        }
    } else {
        return res.status(401).send();
    }
};
```

We will use HTTP error codes for handling request errors:

- HTTP 401 for an invalid request
- HTTP 403 for a valid request with an invalid token, or valid token with invalid permissions

We can use the bitwise AND operator (bitmasking) to control the permissions. If we set each required permission as a power of 2, we can treat each bit of the 32bit integer as a single permission. An admin can then have all permissions by setting their permission value to 2147483647. That user could then have access to any route. As another example, a user whose permission value was set to 7 would have permissions to the roles marked with bits for values 1, 2, and 4 (two to the power of 0, 1, and 2).

The middleware for that would look like this:

```
exports.minimumPermissionLevelRequired = (required_permission_level) => {
   return (req, res, next) => {
       let user_permission_level = parseInt(req.jwt.permission_level);
       let user_id = req.jwt.user_id;
       if (user_permission_level & required_permission_level) {
           return next();
       } else {
           return res.status(403).send();
       }
   };
};
```

The middleware is generic. If the user permission level and the required permission level coincide in at least one bit, the result will be greater than zero and we can let the action proceed, otherwise the HTTP code 403 will be returned.

Now, we need to add the authentication middleware to the user's module routes in `/users/routes.config.js`:

```
app.post('/users', [
   UsersController.insert
]);
app.get('/users', [
   ValidationMiddleware.validJWTNeeded,
   PermissionMiddleware.minimumPermissionLevelRequired(PAID),
   UsersController.list
]);
app.get('/users/:userId', [
   ValidationMiddleware.validJWTNeeded,
   PermissionMiddleware.minimumPermissionLevelRequired(FREE),
   PermissionMiddleware.onlySameUserOrAdminCanDoThisAction,
   UsersController.getById
]);
app.patch('/users/:userId', [
   ValidationMiddleware.validJWTNeeded,
   PermissionMiddleware.minimumPermissionLevelRequired(FREE),
   PermissionMiddleware.onlySameUserOrAdminCanDoThisAction,
   UsersController.patchById
]);
app.delete('/users/:userId', [
   ValidationMiddleware.validJWTNeeded,
   PermissionMiddleware.minimumPermissionLevelRequired(ADMIN),
   UsersController.removeById
]);
```

This concludes the basic development on our REST API. All that remains to be done is to test it all out.

## Running and Testing with Insomnia

Insomnia is a decent REST client with a good free version. The best practice is, of course, to include code tests and implement proper error reporting in the project, but third-party REST clients are great for testing and implementing third-party solutions when error reporting and debugging the service is not available. We'll be using it here to play the role of an application and get some insight into what is going on with our API.

To create a user, we just need to POST the required fields to the appropriate endpoint and store the generated ID for subsequent use.

The API will respond with the user ID:

We can now generate the JWT using the `/auth/` endpoint:

We should get a token as our response:

Grab the `accessToken`, prefix it with `Bearer ` and add it to the request headers under `Authorization`:

If we don’t do this now that we have implemented the permissions middleware, every request other than registration would be returning HTTP code 401. With the valid token in place, though, we get the following response from `/users/:userId`:

Also, as was mentioned before, we are displaying all fields, for educational purposes and for sake of simplicity. The password (hashed or otherwise) should never be visible in the response.

Let’s try to get a list of users:

Surprise! We get a 403 response.

Our user does not have the permissions to access this endpoint. We will need to change the `permissionLevel` of our user from 1 to 7 manually in MongoDB and then generate a new JWT.

After that is done, we get the proper response:

Next let’s test the update functionality by sending a `PATCH` request with some fields to our `/users/:userId` endpoint:

We expect a 204 response as confirmation of a successful operation, but we can request the user once again to verify.

Finally, we need to delete the user. We’ll need to create a new user as described above (don’t forget to note the user ID) and make sure that we have the appropriate JWT for an admin user.

Sending a `DELETE` request to `/users/:userId` we should get a 204 response as confirmation. We can, again, verify by requesting `/users/` to list all existing users.

## Conclusion

With the tools and methods covered in this tutorial, you should now be able to create simple and secure REST APIs on Node.js. A lot of best practices that are not essential to the process were skipped so don't forget to:

- implement proper validations (e.g. make sure that user email is unique)
- implement unit testing and error reporting
- prevent users from changing their own permission level
- prevent admins from removing themselves
- prevent disclosure of sensitive information (e.g., hashed passwords)

You can get the source code on my GitHub.

**Related:** 5 Things You Have Never Done with a REST Specification

TAGS JavaScript Node.js Express REST

Marcos Henrique da Silva
Freelance JavaScript Developer

### ABOUT THE AUTHOR

Marcos is passionate about REST architecture, Agile development methodology, and the JavaScript language for programming. He's been working with IT since 2003, and for the past few years, he's been working nearly exclusively with software development and focusing on web applications. He specializes in JavaScript (using frameworks like Angular 1 and Node.js) and Java EE. Marcos has also worked with native Android apps and PHP.

Hire Marcos

World class articles, delivered

Enter your email

Sign Me Up

weekly.

Subscription implies consent to our privacy policy

## Toptal Developers

Magento Developers

Mixed Reality Developers

Mobile App Developers

.NET Developers

Node.js Developers

PHP Developers

Python Developers

React.js Developers

Ruby Developers

Ruby on Rails Developers

Salesforce Developers

Scala Developers

Software Architects

Software Developers

Unity or Unity3D Developers

Virtual Reality Developers

Web Developers

WordPress Developers

View more

---

Join the Toptal® community. Hire a Developer OR Apply as a Developer

Hire the top 3% of freelance talent™