Rest APIs - Module 5

Noroff Guide

Noroff Education

Table of Contents

Module overview

Introduction

Welcome to Module 5 of REST APIs.

If anything is unclear, check your progression plan or contact a tutor on Discord.

Module structure	**Estimated workload**
5.1. Lesson and task Project functionalities	8 hours
5.2. Lesson and task Authentication	8 hours
5.3. Lesson and task Project documentation	8 hours
5.4. Lesson and task Tests	8 hours
5.5. Lesson and task Self-study	8 hours

Learning outcomes

In this module, we are covering the following knowledge learning outcomes:

→ The candidate has knowledge of industry-standard JavaScript frameworks that can be used to build RESTful services.
→ The candidate has knowledge of techniques, principles, and tools used to implement API authentication.

- → The candidate has insight into REST principles and the features of a RESTful service.
- → The candidate can update their vocational knowledge of industry-standard JavaScript REST frameworks.

In this module, we are covering the following skill learning outcomes:

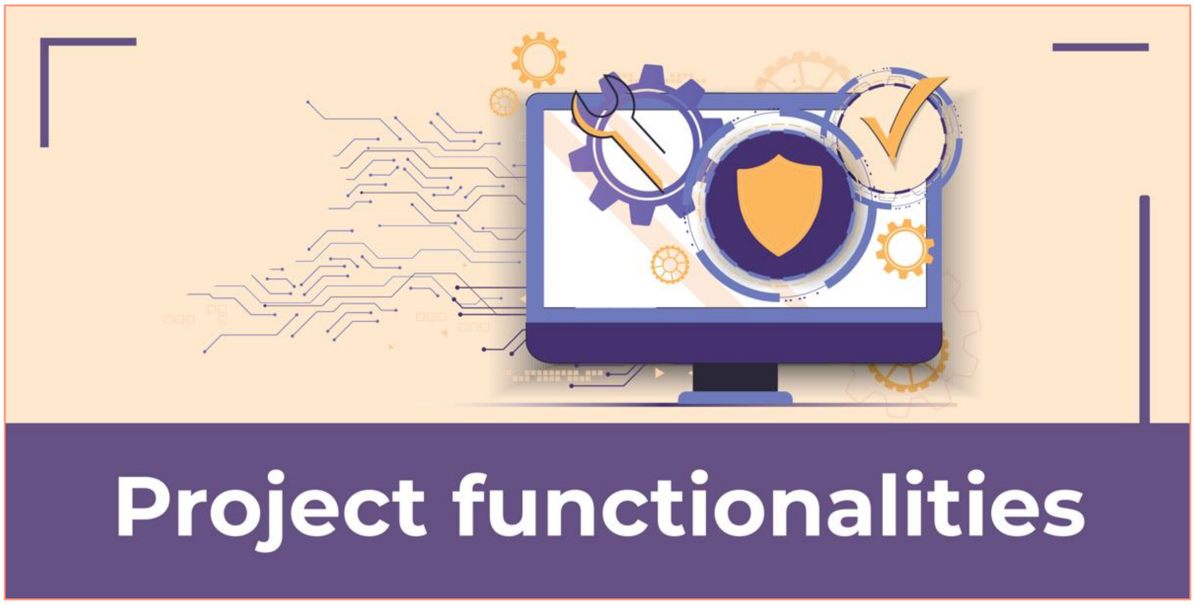
- → The candidate can apply vocational knowledge of RESTful principles to describe the features that identify a service as RESTful.
- → The candidate masters creating RESTful solutions in JavaScript that use common HTTP methods.
- → The candidate masters the Postman software (or open-source alternatives) to test and document a REST API.
- → The candidate can apply vocational knowledge to implement API authentication in new and existing RESTful solutions.
- → The candidate can study an existing database solution and design an appropriate REST service to expose desired data.

In this module, we are covering the following general competence learning outcomes:

- → The candidate can develop RESTful solutions from database integration to completed REST endpoints.
- → The candidate can carry out work as a REST API developer, maintainer, or tester.

5.1. Lesson - Project functionalities

Introduction



In this module, we will use previously learnt knowledge to create a REST API that could be published and used by external users. Up to this point, we have implemented mainly web applications, using the REST principles to create proper HTML views. In this module, we will use JSON to represent the response – we will not focus on creating the web application but rather on exposing the correct data.

In this lesson, we will implement most of the API functionalities. We will set up our project, install the necessary packages, and implement all API endpoints for non-authenticated users.

In this section, a database connection will not be needed – this will be added in future lessons. The API will simulate a calculator, and all request handlers for non-authenticated users will operate only on the data provided by the client as route parameters.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcomes:

- → The candidate has knowledge of industry-standard JavaScript frameworks that can be used to build RESTful services.
- → The candidate has insight into REST principles and the features of a RESTful service.

In this lesson, we are covering the following skill learning outcomes:

- → The candidate masters creating RESTful solutions in JavaScript that use common HTTP methods.
- → The candidate can study an existing Database solution and design an appropriate REST service to expose desired data.

In this lesson, we are covering the following general competence learning outcomes:

- → The candidate can develop RESTful solutions from database integration all the way to completed REST endpoints.
- → The candidate can carry out work as a REST API developer, maintainer, or tester.

Project overview

In this module, we will create the Calculator API.

This API will provide the following endpoints:

- /add/:number1/:number2 – Adds two numbers provided as route parameters.
- /subtract/:number1/:number2 – Subtracts two numbers provided as route parameters.

- /multiply/:number1/:number2 – Multiplies two numbers provided as route parameters.
- /divide/:number1/:number2 – Divides two numbers provided as route parameters.

All these endpoints will be available for non-authenticated users. Their implementation will be the focus of this lesson. Each request's response will be in JSON format.

The application will store data about users and the last result returned to them in the database. The database will be set up in the following lessons. Authenticated users will have access to the following four endpoints:

- /previous/add/:number1 – Adds one number provided as a route parameter to the last result returned to the user.
- /previous/subtract/:number1 – Subtracts one number provided as the route parameter from the last result returned to the user.
- /previous/multiply/:number1 – Multiplies one number provided as the route parameter by the last result returned to the user.
- /previous/divide/:number1 – Divides the last result returned to the user by one number provided as the route parameter.

The calculator is meant to work on integers. If we aren't able to convert the numbers provided as parameters to integers, we will return an error. There are possible scenarios where the parameters are correct, but the result will not be an integer – when number1 is not a divisor of number2 (for example, 9/2 = 4.5 – both numbers are integers, but the result is not). In such cases, the result will be rounded, and we will provide information about this in the response.

The application will be implemented in Express. In this lesson, we will implement a basic API. In the following lessons, we will set up a database (using Sequelize ORM) and create detailed tests and documentation. Documentation will be created in Postman. As the application is meant for external users, we will also use Postman to test the application after each lesson.

Project setup

Let’s create a new folder and name it “restproject”. We will create our new project in this folder. Next, open the folder in the VSCode terminal and generate a new NPM project with Express generator with the command:

```
npm express-generator --view=ejs
```

Install all the required dependencies with the command:

```
npm install
```

Install the dotenv package so that we will be able to use the .env file (which stores our environment variables):

```
npm install dotenv
```

And install this package at the beginning of the generated app.js file:

```
require('dotenv').config()
```

Now, create the .env file on the same level as the app.js file, and set the PORT configuration in it to 3000 (if it’s not already set to 3000):

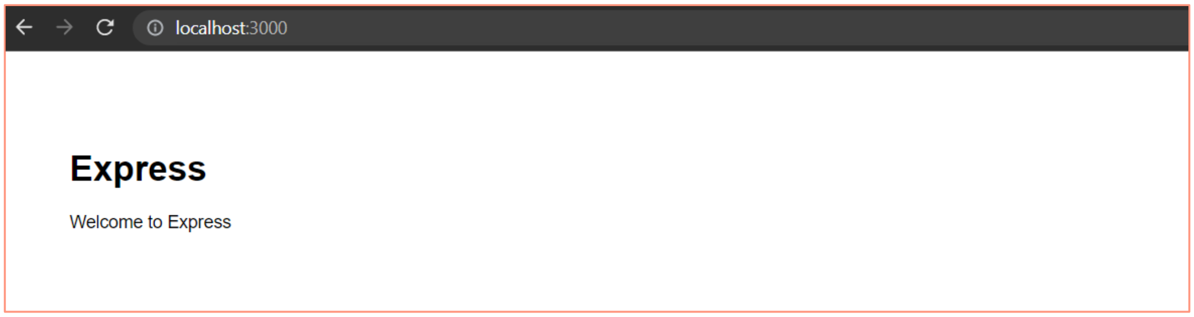
.env:

```
PORT = "3000"
```

Try to run the application with the command:

```
npm start
```

Access the page http://localhost:3000. You should see the Express application result:



Express

Welcome to Express

API implementation

Let’s start implementing the API. For now, we assume that the data provided by users is always correct, so no data validation needs to be implemented – we will focus on error handling in the next section.

Let’s create four router files in the /routes folder:

- routes/add.js
- routes/subtract.js
- routes/multiply.js
- routes/divide.js

Our API is small (there are not many endpoints to handle), so putting all the logic in the main router file (index.js) was a possibility; however, to reinforce good development practices, we separated the logic into different router files.

Let’s implement a handler for each file. We will send the status code 200 (we assume the data will always be correct, so the response should always reflect this) and send the result as JSON – remember that a status code of 200 (OK) means that the request has succeeded.

In our handlers, we can see that the GET requests receive our integers as route parameters in the req object. Knowing that the route parameters will always be strings, we need to convert these parameters to integers (using parseInt) before performing operations on them:

routes/add.js:

```
var express = require('express');
var router = express.Router();

router.get('/:number1/:number2', function(req, res, next) {
  res.status(200).json(parseInt(req.params.number1) +
parseInt(req.params.number2));
});

module.exports = router;
```

routes/subtract.js:

```
var express = require('express');
var router = express.Router();

router.get('/:number1/:number2', function(req, res, next) {
  res.status(200).json(parseInt(req.params.number1) -
parseInt(req.params.number2));
});

module.exports = router;
```

routes/multiply.js:

```
var express = require('express');
var router = express.Router();

router.get('/:number1/:number2', function(req, res, next) {
  res.status(200).json(parseInt(req.params.number1) *
parseInt(req.params.number2));
});

module.exports = router;
```

routes/divide.js:

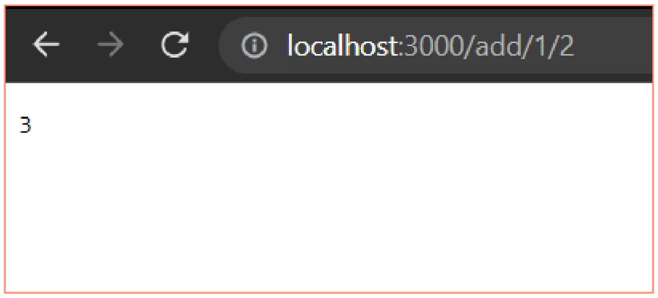
```
var express = require('express');
var router = express.Router();

router.get('/:number1/:number2', function(req, res, next) {
  res.status(200).json(parseInt(req.params.number1) /
parseInt(req.params.number2));
});

module.exports = router;
```

Next, let's include these files in the main app.js file. We can do this with the following code:

app.js:

```
var addRouter = require('./routes/add');
var subtractRouter = require('./routes/subtract');
var multiplyRouter = require('./routes/multiply');
var divideRouter = require('./routes/divide');

app.use('/add', addRouter);
app.use('/subtract', subtractRouter);
app.use('/multiply', multiplyRouter);
app.use('/divide', divideRouter);
```

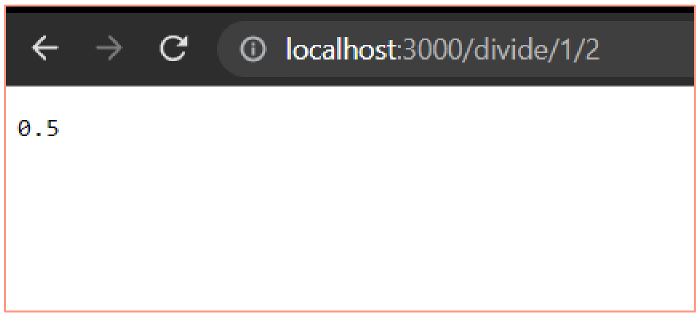
After running the application, almost everything should work properly:



localhost:3000/add/1/2

```
3
```

However, the /divide endpoint is not working as specified yet, as it returns a float, not an integer:



localhost:3000/divide/1/2

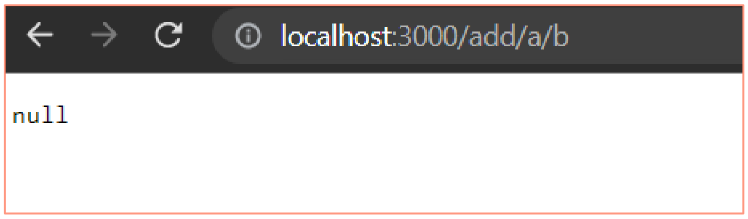
```
0.5
```

This will be fixed in the next lesson.

Error handling

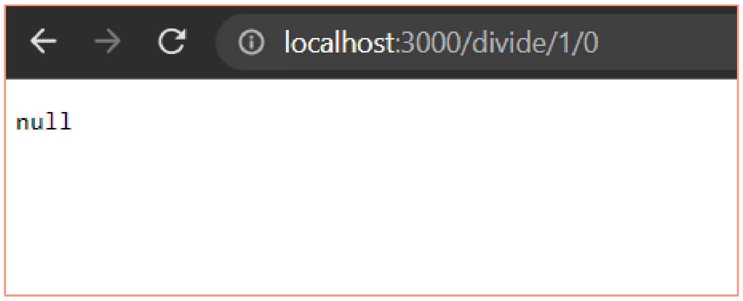
In this section, we will consider scenarios where data provided by the user is not correct. Currently, in such cases, null is returned in the response. For example:

http://localhost:3000/add/a/b



```
null
```

http://localhost:3000/divide/1/0



```
null
```

Additionally, our JSON response currently returns only calculation results. We should return results with some context so that anybody using the API will know what data is being returned. When developing an API for external users, following some specifications for JSON response structure is a good idea. JSend is one of the most popular specifications of this kind. It can be viewed on the page linked in the READ section below.

READ

Page: JSend on GitHub.

A JSend response has the following format:

```
{
    status : "success",
    data : {
        "post" : { "id" : 1, "title" : "A blog post", "body"
: "Some useful content" }
     }
}
```

- **Status** is either “success”, “fail”, or “error” – it informs us whether the request has been processed properly.
- **Data** is an object and contains returned information.

We could manually structure responses to follow the JSend specification; however, using the existing JSend external package is easier.

Let’s install the jsend package with the command:

```
npm install jsend
```

Next, let’s use JSend in each created router file.

In our files:

- add.js
- subtract.js
- multiply.js
- divide.js

The following code applies:

```
const jsend = require('jsend')
//...
router.use(jsend.middleware);
```

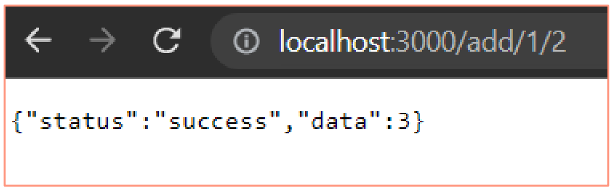
And in implementation, use res.jsend.success() instead of res.status(200).json().

Example of the final add.js file:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();

router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const result = parseInt(req.params.number1) +
parseInt(req.params.number2);
    res.jsend.success(result);
});

module.exports = router;
```

Now, implement the JSend format in the other router files by yourself.

After testing, we will get the following result in the case of a successful request:



```
{"status":"success","data":3}
```

Besides the success() method, JSend also provides us with the following methods:

- fail() – Used when data provided in the request is wrong.
- error() – Used when there was an error on the server side and the server couldn't return a successful response.

Let's add validation for the route parameters and use the fail() method in the case that the parameters are not valid:

add.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();

router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    const result = number1 + number2;
    res.jsend.success(result);
});

module.exports = router;
```

subtract.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();

router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    const result = number1 - number2;
    res.jsend.success(result);
});

module.exports = router;
```

multiply.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();

router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    const result = number1 * number2;
    res.jsend.success(result);
});

module.exports = router;
```

In the case of divide.js, we also need to check if the result is an int and that number2 is not 0 (as diving by 0 would give an undefined result). To communicate that the result is rounded (the original result was not an integer), let’s return the object instead of one field (in the case of success):

add.js / subtract.js / multiply.js:

```
  res.jsend.success({"result": result});
```

divide.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();

router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    if(number2 == 0) {
        return res.jsend.fail({"number2": "number2 cannot be
0"});
    }
    const result = number1 / number2;
    if (Number.isInteger(result)) {
        res.jsend.success({"result": result});
    }
    else {
        res.jsend.success({"result": Math.round(result),
"message": "Result has been rounded, as it was not an
integer."});
    }
});

module.exports = router;
```

We check the same conditions as in previous cases (if number1 and number2 are numbers), but additionally, we also check if number2 is 0 (we cannot divide by 0) and if the result is an integer (we have to return an integer, but the result of dividing two integers does not have to be an integer). When the result is not an integer, we round it to the closest integer.

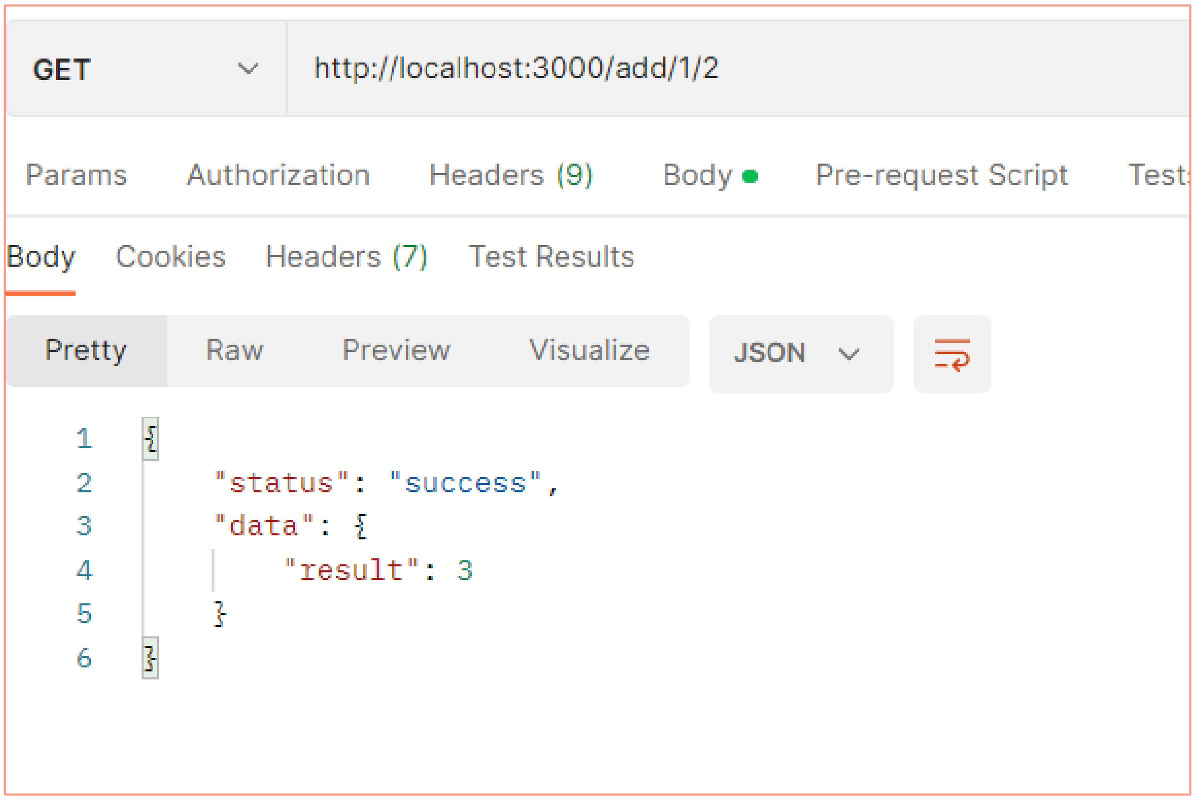
We will test this API in the next section.

Testing with Postman

Let’s open Postman and test each endpoint.

Make one request that is supposed to return a success:

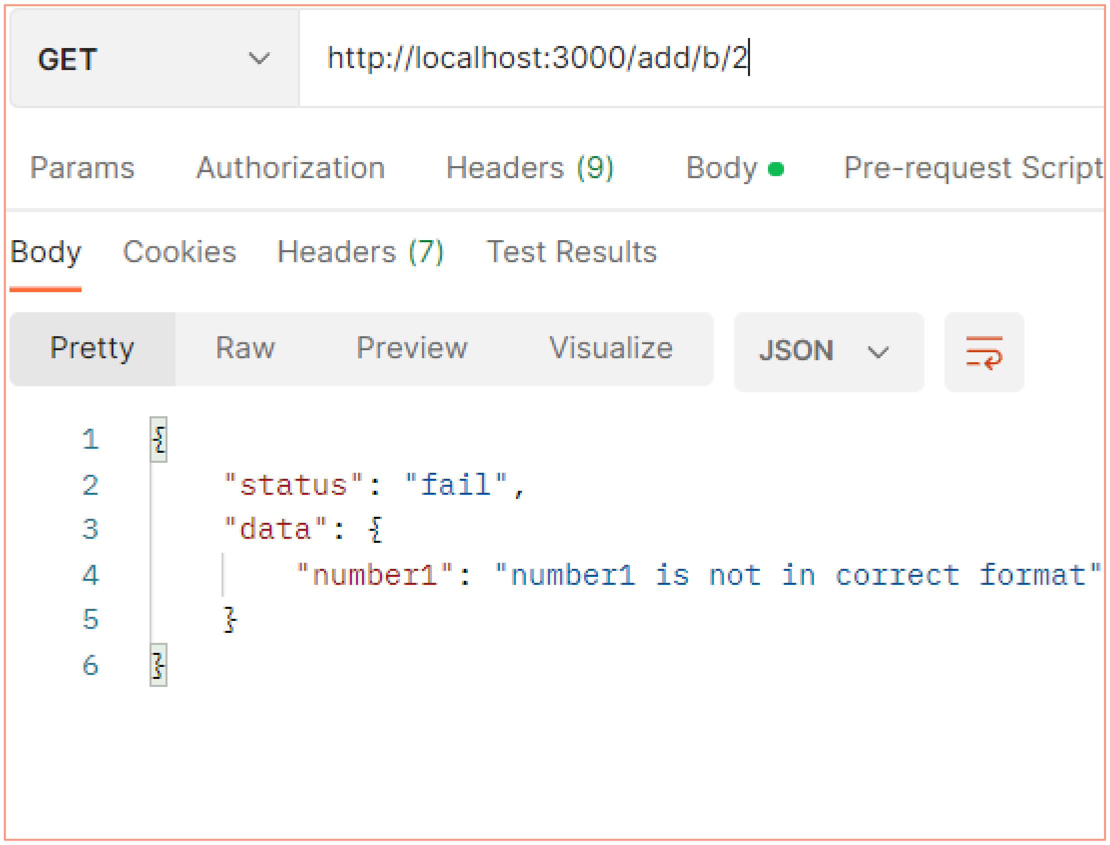
http://localhost:3000/add/1/2



```
{
    "status": "success",
    "data": {
        "result": 3
    }
}
```

And make one request that is supposed to fail:

http://localhost:3000/add/b/2

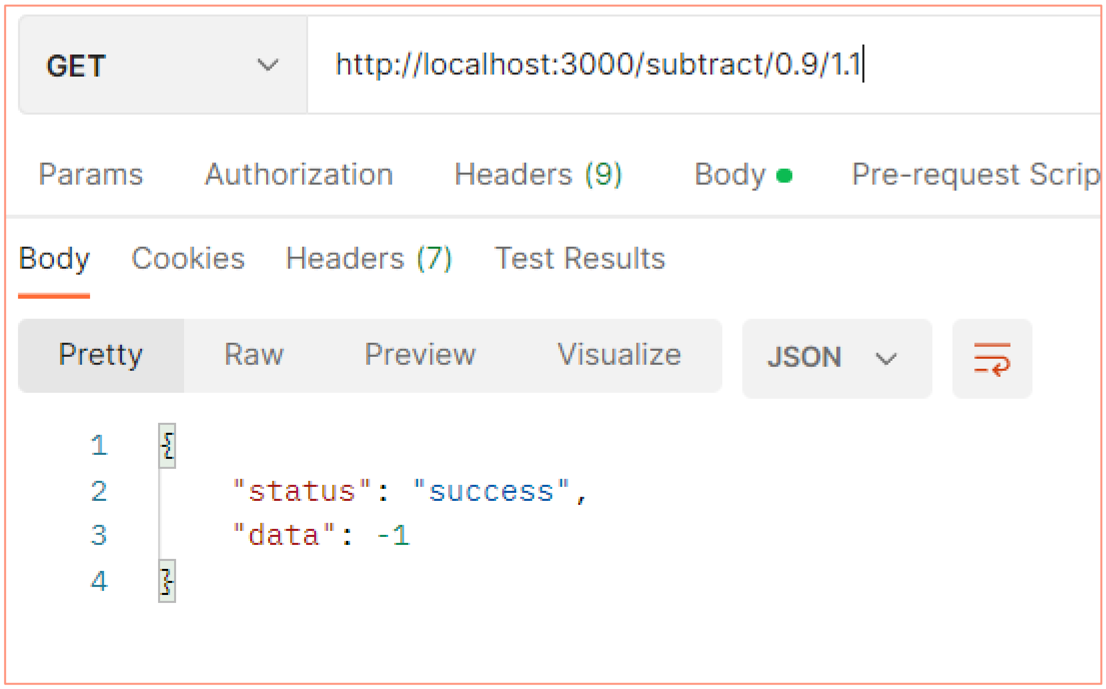


```
http://localhost:3000/add/b/2
```

```
{
    "status": "fail",
    "data": {
        "number1": "number1 is not in correct format"
    }
}
```

Also, test a case to make sure that an integer is returned:

http://localhost:3000/subtract/0.9/1.1



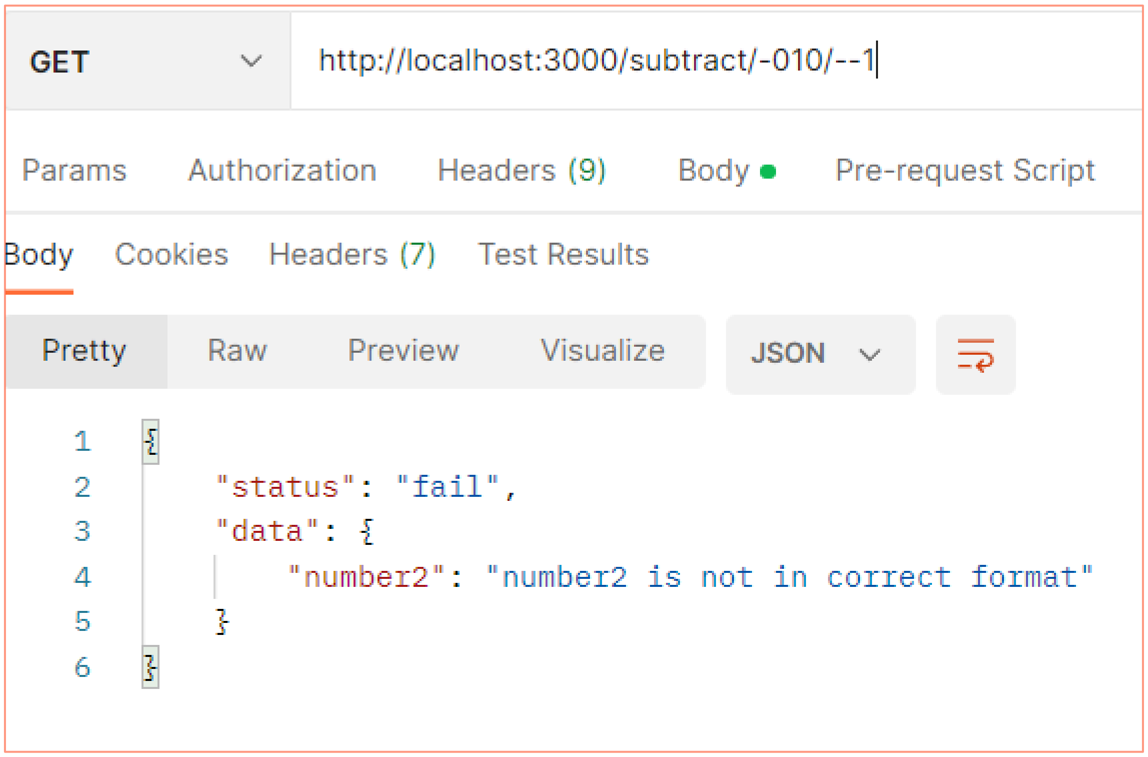
```
http://localhost:3000/subtract/0.9/1.1
```

```
{
    "status": "success",
    "data": -1
}
```

Notice that parameters are always cut to the integer part, not rounded.

Test cases where an incorrect format is given to the parameters:

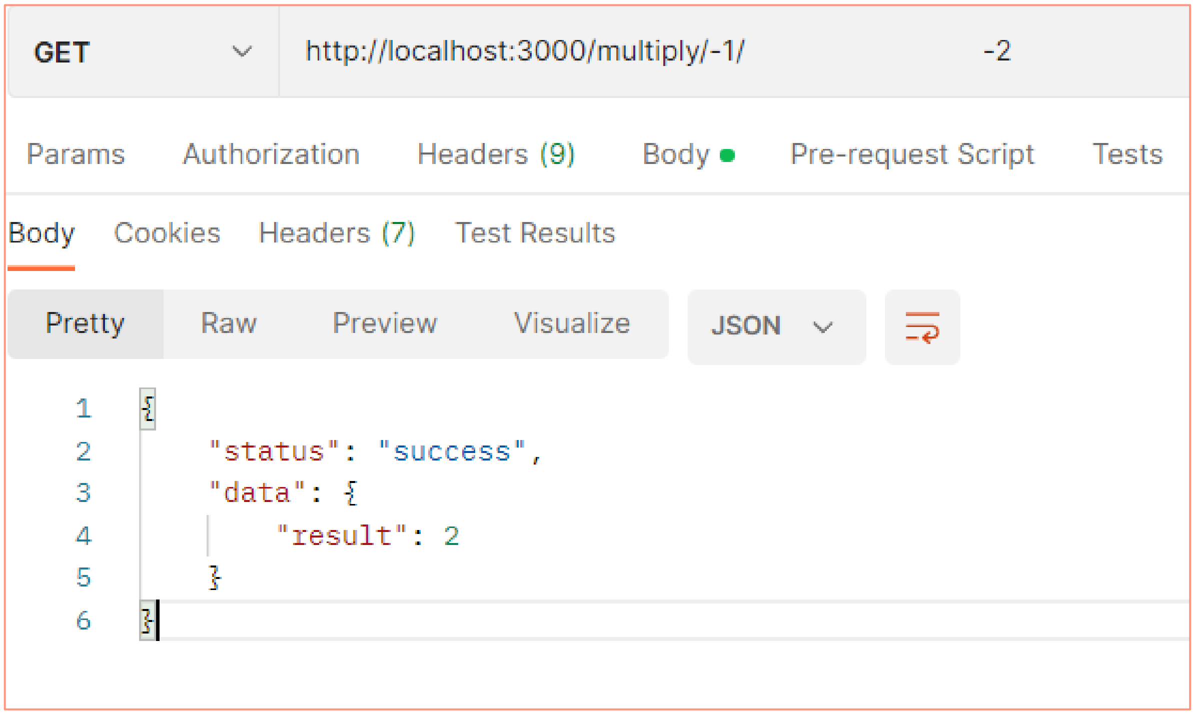
http://localhost:3000/subtract/-010/--1



```
{
    "status": "fail",
    "data": {
        "number2": "number2 is not in correct format"
    }
}
```

Notice that -010 is correct and is equal to -10. However, --1 is not correct and cannot be parsed to an integer.

http://localhost:3000/multiply/-1/ -2

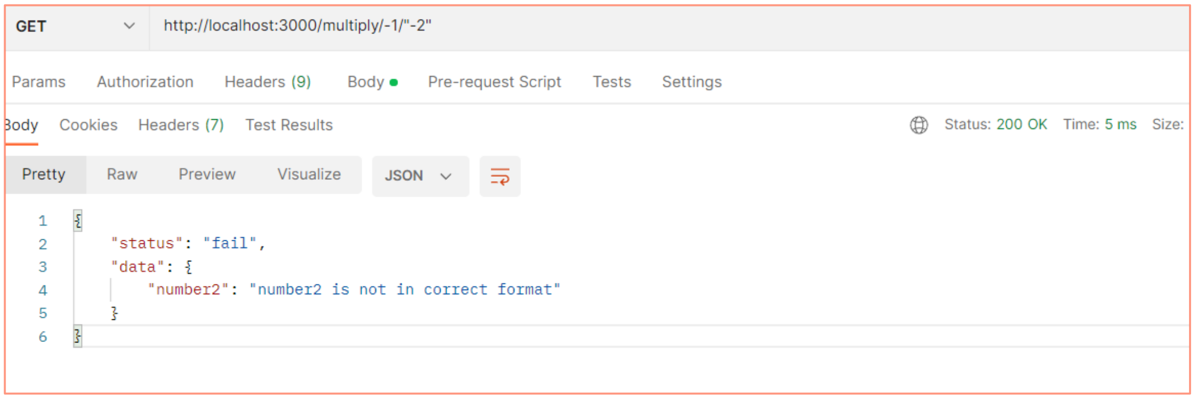


http://localhost:3000/multiply/-1/   -2

```
{
    "status": "success",
    "data": {
        "result": 2
    }
}
```

The application works correctly even if there are redundant spaces in the URL.

http://localhost:3000/multiply/-1/"-2"

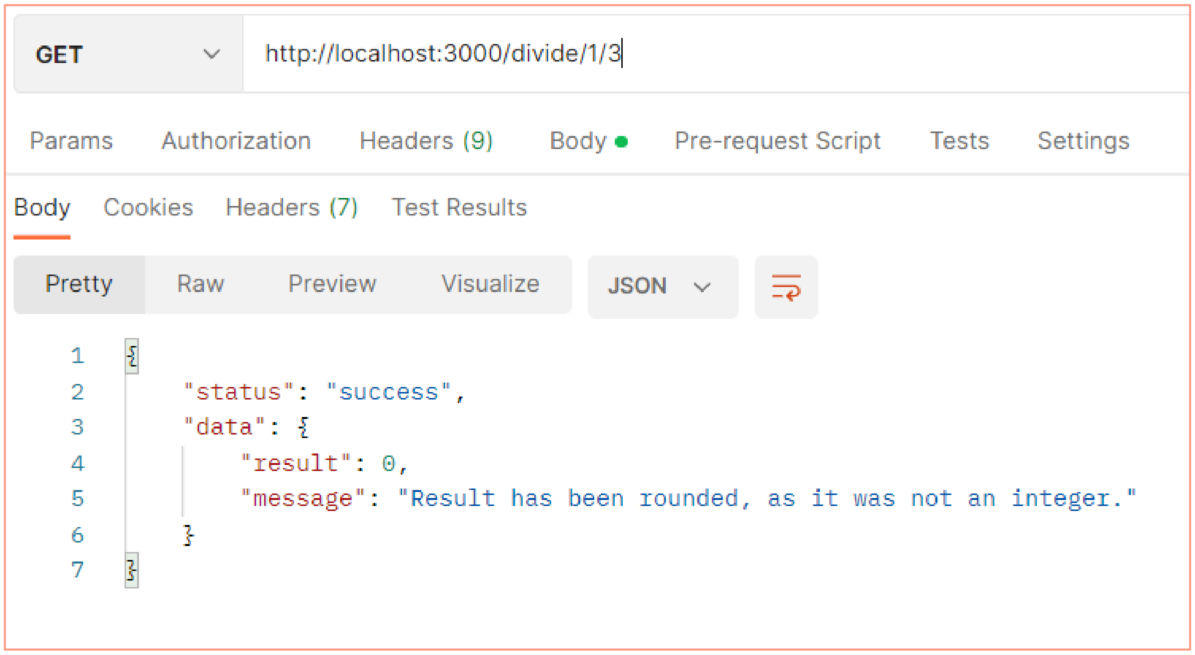


http://localhost:3000/multiply/-1/"-2"

Status: 200 OK Time: 5 ms Size:

```
{
    "status": "fail",
    "data": {
        "number2": "number2 is not in correct format"
    }
}
```

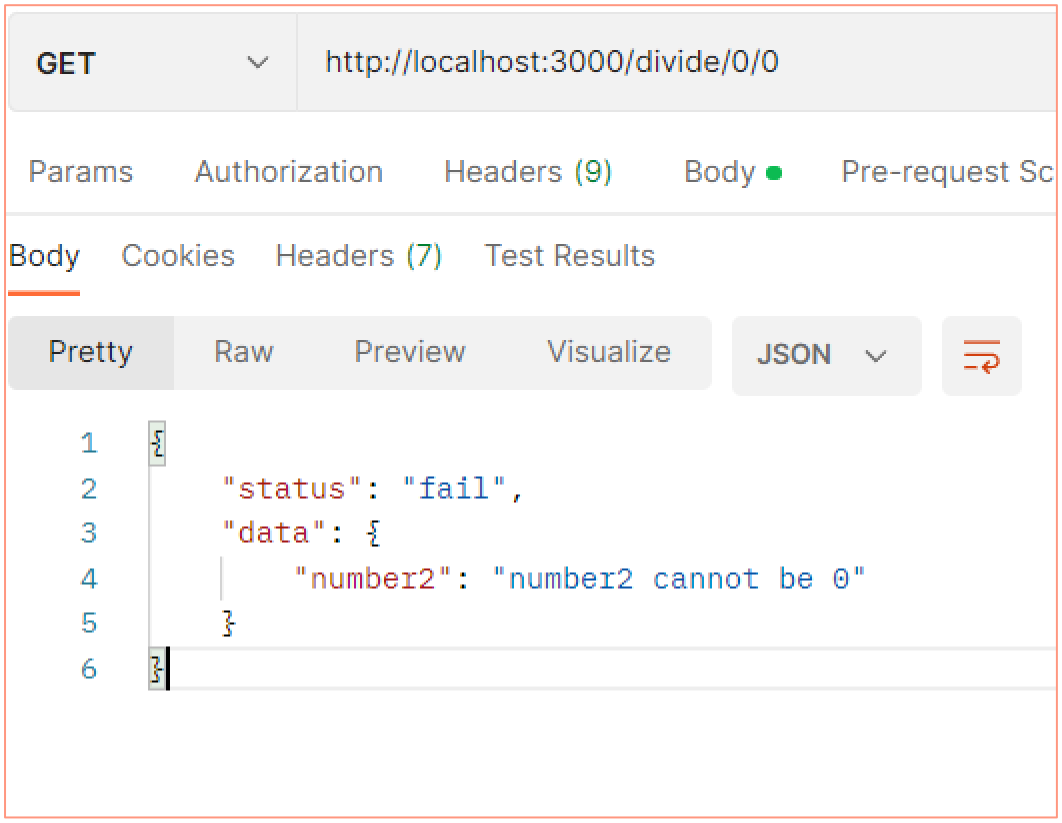
http://localhost:3000/divide/1/3



```
{
    "status": "success",
    "data": {
        "result": 0,
        "message": "Result has been rounded, as it was not an integer."
    }
}
```

The result of 1/3 is not an integer, so we got a success, but we also got a message about the result being rounded.

http://localhost:3000/divide/0/0



```
{
    "status": "fail",
    "data": {
        "number2": "number2 cannot be 0"
    }
}
```

All these tests ensure that our API is working correctly. They are also great examples to put into the API documentation – we will return to this aspect in the following lessons.

What did I learn in this lesson?

This lesson provided the following insights:

- How to set up the REST API.
- How to add error handling to the API handlers.
- How to structure JSON response according to the JSend specification.

References

	INFO Please note that the reference list below was compiled when writing the course content to credit the sources used. Therefore, some links that worked at the time of writing may not work now or may have changed. However, accessing these references is not a requirement for this module. Note that all the required reading is specified in the READ blocks in the content.

Expressjs.com. (2017). *Express 4.x - API Reference*. [online] Available at: https://expressjs.com/en/api.html.

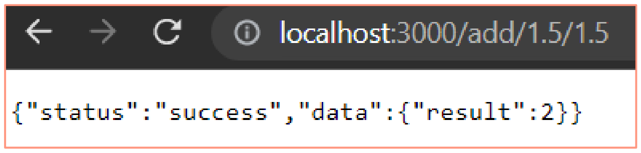
GitHub. (2023). *JSend*. [online] Available at: https://github.com/omniti-labs/jsend [Accessed 13 Mar. 2023].

5.1. Lesson task - Project functionalities

The task

In this lesson, we implemented the basic functionalities of our API. In this task, you need to change the application so that it supports floats instead of integers. In the following lessons, we will use integers, so remember to change it back.

Currently, http://localhost:3000/add/1.5/1.5 returns 2, as 1.5 is converted to 1:



```
{"status":"success","data":{"result":2}}
```

Change the application (all the endpoints, not only /add) so that in this case, it will return 3 instead.

5.2. Lesson - Authentication

Introduction

In the previous lesson, we implemented a simple REST API available to all users. That API simulates a calculator and handles basic operations on integers.

In this lesson, we will add a new functionality available only to authenticated users. We will store the result of the last requested operation in a MySQL database and enable some endpoints to use this stored value instead of the first number.

Authentication will use a JSON Web Token, so we will be able to log in from Postman and send authorised requests without initialising the session.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

→ The candidate has knowledge of techniques, principles, and tools used to implement API authentication.

In this lesson, we are covering the following skill learning outcome:

→ The candidate can apply vocational knowledge to implement API authentication in new and existing RESTful solutions.

In this lesson, we are covering the following general competence learning outcomes:

→ The candidate can develop RESTful solutions from database integration all the way to completed REST endpoints.
→ The candidate can carry out work as a REST API developer, maintainer, or tester.

Database setup

The last time we implemented authentication with a JSON Web Token, we used a local array to store the users. This solution was temporary – to demonstrate how JSON Web Token authentication works. In this lesson, we will store user data in a MySQL database.

We will generate our database structure (tables and fields) using Sequelize.

Install the necessary packages with the command:

```
npm install sequelize mysql mysql2
```

Open MySQL Workbench and create a new database – name it ‘Calculator’.

Add a standard user named ‘ProjectAdmin’ to the database and give them admin permissions.

Create the .env file at the same level as app.js and fill it with data:

```
HOST = "localhost"
ADMIN_USERNAME = "ProjectAdmin"
ADMIN_PASSWORD = "0000"
DATABASE_NAME = "Calculator"
DIALECT = "mysql"
PORT = "3000"
```

Create the ‘models’ folder in the root of the application. We will create two models: one for users and one for their results. Each user will have a name, an email, an encrypted password, and a salt used for encryption. Each result will consist of an operation name and a value.

Now, let’s add files containing this information to our models folder:

models/index.js:

```
const Sequelize = require('sequelize')
const fs = require("fs")
const path = require("path")
const basename = path.basename(__filename);
const sequelize = new Sequelize(process.env.DATABASE_NAME,
process.env.ADMIN_USERNAME, process.env.ADMIN_PASSWORD,
{  host: process.env.HOST, dialect: process.env.DIALECT });
const db = {}
db.sequelize = sequelize
fs.readdirSync(__dirname)
  .filter(file => {
    return (file.indexOf('.') !== 0) && (file !== basename)
&&
      (file.slice(-3) === '.js');
    })
  .forEach(file => {
    const model = require(path.join(__dirname,
file))(sequelize,
      Sequelize);
    db[model.name] = model;
  });
Object.keys(db).forEach(modelName => {
  if (db[modelName].associate) {
    db[modelName].associate(db);
  }
});
module.exports = db
```

models/User.js:

```
module.exports = (sequelize, Sequelize) => {
    const User = sequelize.define('User', {
        Name: {
            type: Sequelize.DataTypes.STRING,
            allowNull: false
        },
        Email: {
            type: Sequelize.DataTypes.STRING,
            allowNull: false
        },
        EncryptedPassword: {
            type: Sequelize.DataTypes.BLOB,
            allowNull: false
        },
        Salt: {
            type: Sequelize.DataTypes.BLOB,
            allowNull: false
        },
    },{
        timestamps: false
    });
    User.associate = function(models) {
        User.hasOne(models.Result);
    };
    return User
}
```

models/Result.js:

```
module.exports = (sequelize, Sequelize) => {
    const Result = sequelize.define('Result', {
        OperationName: Sequelize.DataTypes.STRING,
        Value: Sequelize.DataTypes.INTEGER
    },{
        timestamps: false
    });
    Result.associate = function(models) {
        Result.belongsTo(models.User);
    };
    return Result
}
```

One user has one result – this is a type of one-to-one relationship.

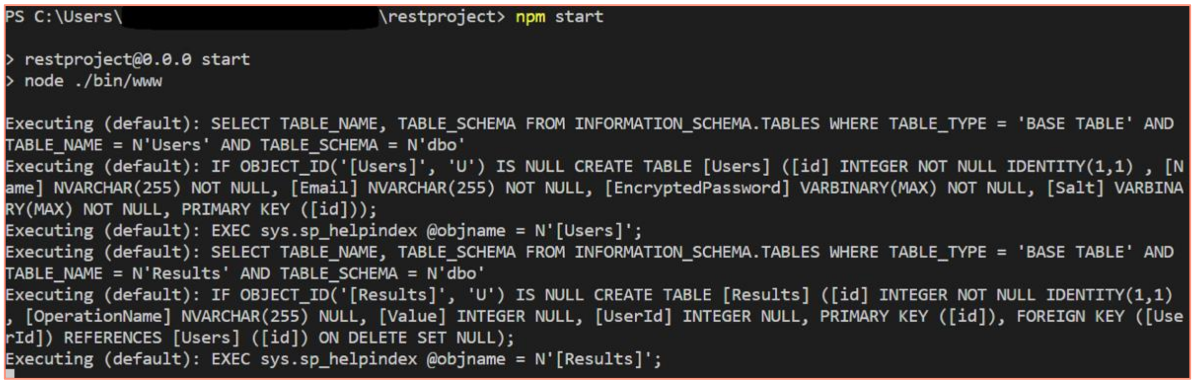
Initialise the database in app.js after calling the dotenv package but before the Express initialisation:

```
require('dotenv').config()
var db = require("./models");
db.sequelize.sync({ force: false })
var app = express();
```

Run the application with the command:

```
npm start
```

We should see that the tables have been generated in our Calculator database:



```
PS C:\Users\[redacted]\restproject> npm start

> restproject@0.0.0 start
> node ./bin/www

Executing (default): SELECT TABLE_NAME, TABLE_SCHEMA FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_TYPE = 'BASE TABLE' AND
TABLE_NAME = N'Users' AND TABLE_SCHEMA = N'dbo'
Executing (default): IF OBJECT_ID('[Users]', 'U') IS NULL CREATE TABLE [Users] ([id] INTEGER NOT NULL IDENTITY(1,1) , [N
ame] NVARCHAR(255) NOT NULL, [Email] NVARCHAR(255) NOT NULL, [EncryptedPassword] VARBINARY(MAX) NOT NULL, [Salt] VARBINA
RY(MAX) NOT NULL, PRIMARY KEY ([id]));
Executing (default): EXEC sys.sp_helpindex @objname = N'[Users]';
Executing (default): SELECT TABLE_NAME, TABLE_SCHEMA FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_TYPE = 'BASE TABLE' AND
TABLE_NAME = N'Results' AND TABLE_SCHEMA = N'dbo'
Executing (default): IF OBJECT_ID('[Results]', 'U') IS NULL CREATE TABLE [Results] ([id] INTEGER NOT NULL IDENTITY(1,1)
, [OperationName] NVARCHAR(255) NULL, [Value] INTEGER NULL, [UserId] INTEGER NULL, PRIMARY KEY ([id]), FOREIGN KEY ([Use
rId]) REFERENCES [Users] ([id]) ON DELETE SET NULL);
Executing (default): EXEC sys.sp_helpindex @objname = N'[Results]';
```

Authentication

Let’s start by implementing services for the created models.

Create the services models and files: UserService.js and ResultService.js.

Implement methods to get one record by ID – getOne(), and to add a new record to the table – create():

services/UserService.js:

```
class UserService {
    constructor(db) {
        this.client = db.sequelize;
        this.User = db.User;
    }

    async getOne(email) {
        return this.User.findOne({
            where: {Email: email}
        })
    }

    async create(name, email, encryptedPassword, salt) {
        return this.User.create({
            Name: name,
            Email: email,
            EncryptedPassword: encryptedPassword,
            Salt: salt
        })
    }
}

module.exports = UserService;
```

services/ResultService.js:

```
class ResultService {
    constructor(db) {
        this.client = db.sequelize;
        this.Result = db.Result;
    }

    async getOne(userId) {
        return this.Result.findOne({
            where: {UserId: userId}
        })
    }
    async create(operationName, value, userId) {
        const model = this.Result;
        return model.findOne({
            where: {UserId: userId}
        }).then(function(result) {
            if(result) {
                return model.update({
                    OperationName: operationName,
                    Value: value,
                }, {where: {UserId: userId}})
            }
            else {
                return model.create({
                    OperationName: operationName,
                    Value: value,
                    UserId: userId
                })
            }
        });
    }

module.exports = ResultService;
```

These services will be useful later in this lesson.

In ResultService, we first check whether the user has already set the result. If so, we update the operation name and value; otherwise, we add the new record to the table.

In this section**,** we will use methods from UserService to implement user authentication. Users will be able to sign up and log in. We will also use the “crypto” package to encrypt the user’s password. This will be done the same way passport authentication was implemented in previous lessons.

Create the auth.js router file. Here, link app.js:

```
var authRouter = require('./routes/auth');
//...
app.use('/', authRouter);
```

Now, let’s move to the created auth.js router file. Include all the necessary packages:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var crypto = require('crypto');
var UserService = require("../services/UserService")
var userService = new UserService(db);
var bodyParser = require('body-parser')
var jsonParser = bodyParser.json()
router.use(jsend.middleware);
```

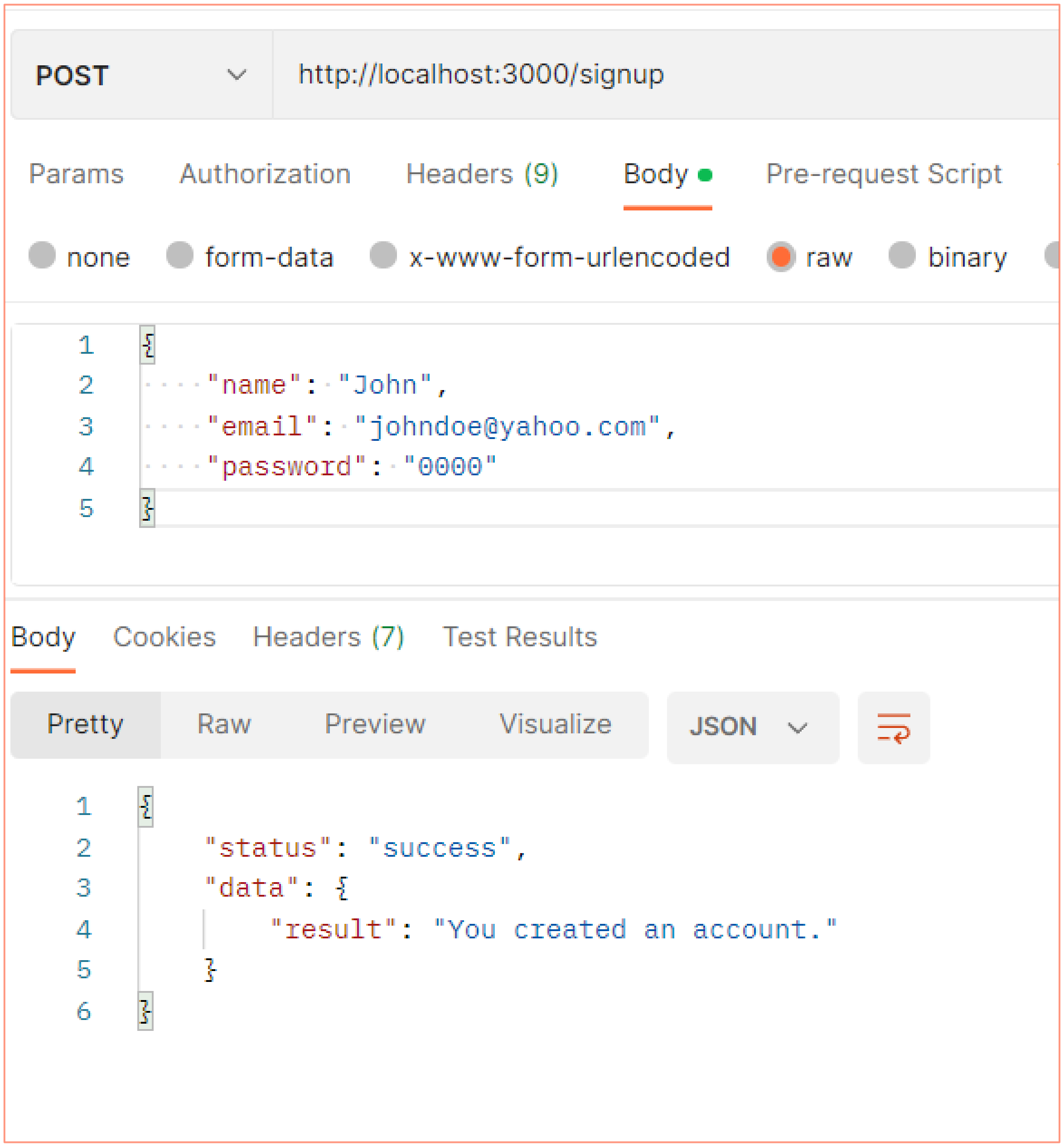
We use JSend to return JSON responses following the JSend specification. We access user services to get and create users. The crypto package is used to generate passwords for new users. Lastly, we use a body parser to read the JSON request’s body.

Let’s implement the /signup endpoint in auth.js:

```
  router.post("/signup", async (req, res, next) => {
    const { name, email, password } = req.body;
    var salt = crypto.randomBytes(16);
    crypto.pbkdf2(password, salt, 310000, 32, 'sha256',
function(err, hashedPassword) {
      if (err) { return next(err); }
      userService.create(name, email, hashedPassword, salt)
      res.jsend.success({"result": "You created an
account."});
    });
});
```

We get the name, email, and password from the body parameters. Then, we use the crypto package to create an encrypted password for the user. Finally, we save the data in the Users table and return the response with JSend.

Let’s test the signup with Postman:



```
http://localhost:3000/signup
```

```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password": "0000"
}
```

```
{
    "status": "success",
    "data": {
        "result": "You created an account."
    }
}
```

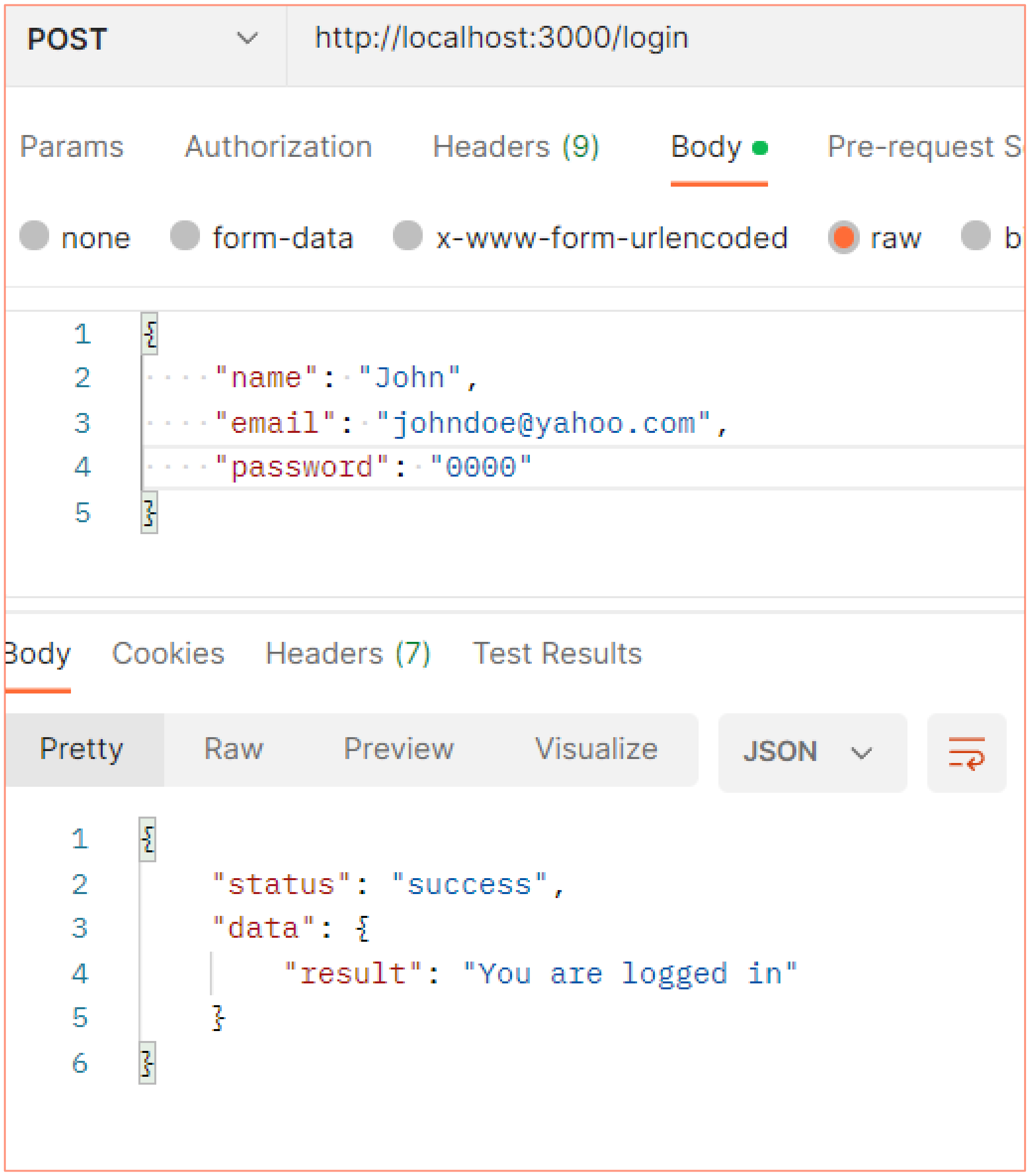
Next, we need to implement the login functionality. The user should provide an email and a password in the request's body. Then, we will encrypt the password and compare it with the actual one from the database. For now, let's just return information about correct credentials in JSON – we will generate a JWT token in the next section.

auth.js:

```
router.post("/login", jsonParser, async (req, res, next) => {
    const { email, password } = req.body;
    userService.getOne(email).then((data) => {
        if(data === null) {
            return res.jsend.fail({"result": "Incorrect email
or password"});
        }
        crypto.pbkdf2(password, data.Salt, 310000, 32,
'sha256', function(err, hashedPassword) {
          if (err) { return cb(err); }
          if (!crypto.timingSafeEqual(data.EncryptedPassword,
hashedPassword)) {
              return res.jsend.fail({"result": "Incorrect
email or password"});
          }
          res.jsend.success({"result": "You are logged in"});
        });
    });
});
```

We look for a user based on the email and return a response with JSend based on whether the login attempt was successful.

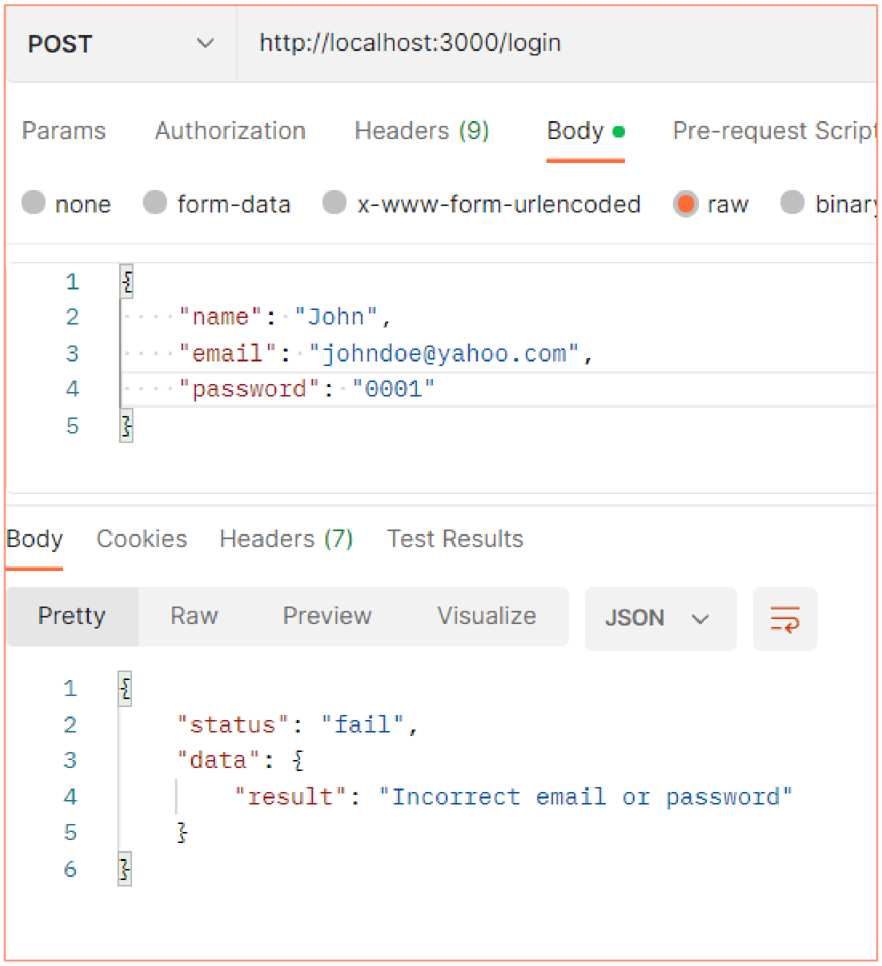
Let’s test it from Postman with a successful request:


POST http://localhost:3000/login

```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password": "0000"
}
```

```
{
    "status": "success",
    "data": {
        "result": "You are logged in"
    }
}
```

And with an unsuccessful request:



```
http://localhost:3000/login
```

```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password": "0001"
}
```

```
{
    "status": "fail",
    "data": {
        "result": "Incorrect email or password"
    }
}
```

Generating a JWT token

In this section, we will generate a JWT token for the user after a successful login.

For this, we need the jsonwebtoken package, so let’s install it with the command:

```
npm install jsonwebtoken
```

And include it in the auth.js router file:

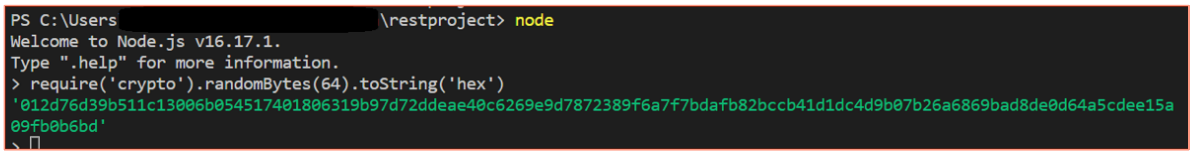
```
var jwt = require('jsonwebtoken')
```

Now, let’s generate the token secret and place it in the .env file. To do this, open the VSCode terminal and run the node with the command:

```
node
```

Then, use the crypto package to generate the token. To do this, place the following command in the node terminal:

```
require('crypto').randomBytes(64).toString('hex')
```



```
PS C:\Users[redacted]\restproject> node
Welcome to Node.js v16.17.1.
Type ".help" for more information.
> require('crypto').randomBytes(64).toString('hex')
'012d76d39b511c13006b054517401806319b97d72ddeae40c6269e9d7872389f6a7f7bdafb82bccb41d1dc4d9b07b26a6869bad8de0d64a5cdee15a
09fb0b6bd'
> 
```

Copy the token shown in the terminal and place it in the .env file:

```
TOKEN_SECRET =
012d76d39b511c13006b054517401806319b97d72ddeae40c6269e9d78723
89f6a7f7bdafb82bccb41d1dc4d9b07b26a6869bad8de0d64a5cdee15a09f
b0b6bd
```

Let's use this token to generate the JWT token for a logged-in user. In the auth.js file, replace the line returning information about the successful login with the code:

auth.js:

```
           let token;
          try {
            token = jwt.sign(
              { id: data.id, email: data.Email },
              process.env.TOKEN_SECRET,
              { expiresIn: "1h" }
            );
          } catch (err) {
            res.jsend.error("Something went wrong with
creating JWT token")
          }
          res.jsend.success({"result": "You are logged in",
"id": data.id, email: data.Email, token: token});
```

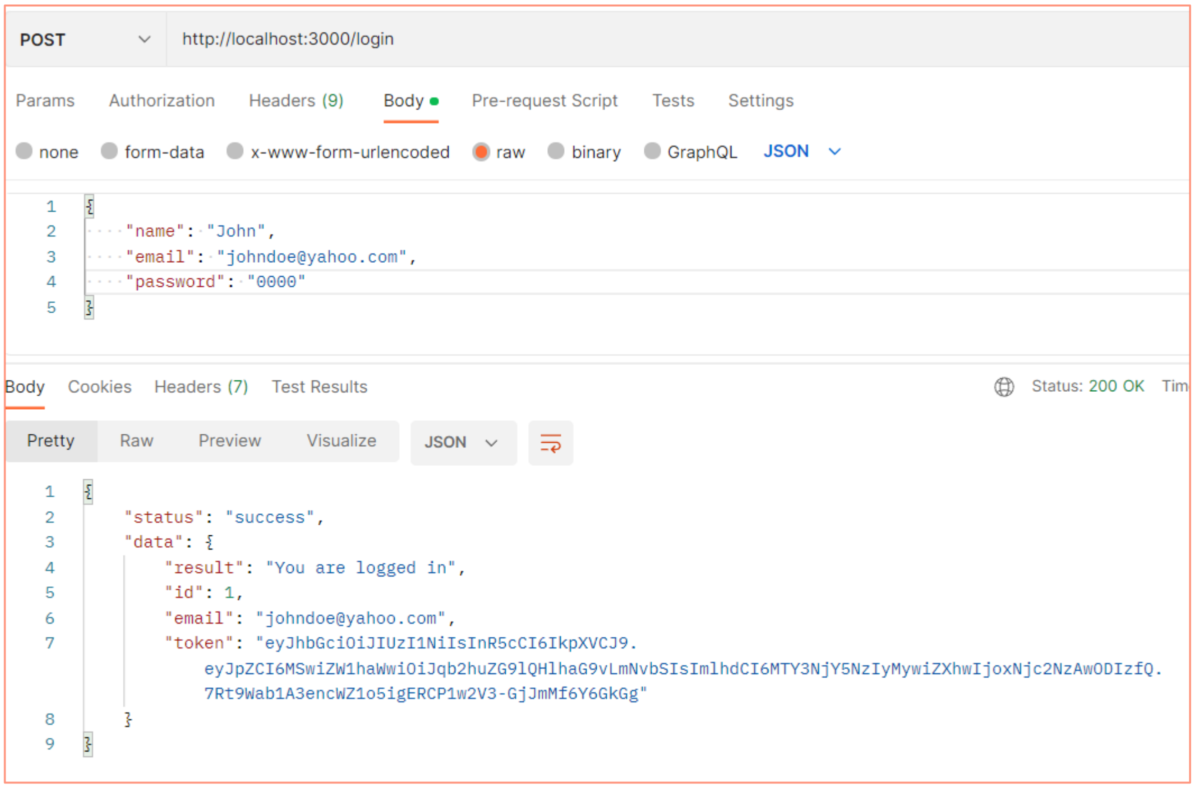
The JWT token has an expiration time – we set it to one hour in the line:

```
{ expiresIn: "1h" }
```

After that time, the JWT token is no longer valid, and the verify() method will return the error that the token has expired.

The JWT token stores information about the ID and email of the user. We use the sign() method from the jsonwebtoken package to create a new JWT token. In the case of an error, we send a message with the JSend error() method. If everything goes correctly, we send the token in the response.

Let's test it with Postman:



```
http://localhost:3000/login
```

```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password": "0000"
}
```

```
{
    "status": "success",
    "data": {
        "result": "You are logged in",
        "id": 1,
        "email": "johndoe@yahoo.com",
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
            eyJpZCI6MSwiZW1haWwiOiJqb2huZG9lQHlhaG9vLmNvbSIsImlhdCI6MTY3NjY5NzIyMywiZXhwIjoxNjc2NzAwODIzfQ.
            7Rt9Wab1A3encWZ1o5igERCP1w2V3-GjJmMf6Y6GkGg"
    }
}
```

Here we can see that the token has been sent in the response.

Our final auth.js file looks as follows:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var crypto = require('crypto');
var UserService = require("../services/UserService")
var userService = new UserService(db);
var bodyParser = require('body-parser')
var jsonParser = bodyParser.json()
var jwt = require('jsonwebtoken')

router.use(jsend.middleware);

router.post("/login", jsonParser, async (req, res, next) => {
    const { email, password } = req.body;
    userService.getOne(email).then((data) => {
        if(data === null) {
            return res.jsend.fail({"result": "Incorrect email
or password"});
        }
        crypto.pbkdf2(password, data.Salt, 310000, 32,
'sha256', function(err, hashedPassword) {
          if (err) { return cb(err); }
          if (!crypto.timingSafeEqual(data.EncryptedPassword,
hashedPassword)) {
              return res.jsend.fail({"result": "Incorrect
email or password"});
          }
          let token;
          try {
            token = jwt.sign(
              { id: data.id, email: data.Email },
              process.env.TOKEN_SECRET,
              { expiresIn: "1h" }
            );
          } catch (err) {
            res.jsend.error("Something went wrong with
creating JWT token")
          }
          res.jsend.success({"result": "You are logged in",
"id": data.id, email: data.Email, token: token});
        });
    });
});
```

```
  router.post("/signup", async (req, res, next) => {
    const { name, email, password } = req.body;
    var salt = crypto.randomBytes(16);
    crypto.pbkdf2(password, salt, 310000, 32, 'sha256',
function(err, hashedPassword) {
      if (err) { return next(err); }
      userService.create(name, email, hashedPassword, salt)
      res.jsend.success({"result": "You created an
account."});
    });
});

module.exports = router;
```

Using the token

In this section, we will use the generated JWT token to recognise the user. We will implement endpoints depending on the user's previous result.

First, we need to set the previous result at the end of each current request handler. We can retrieve data from the token using the jsonwebtoken verify() method.

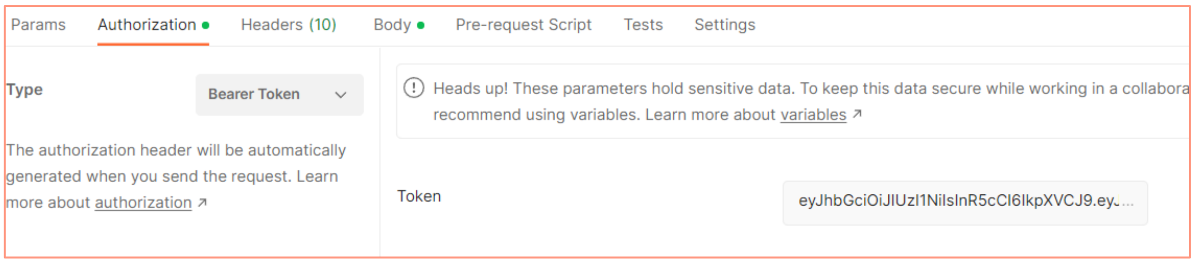
Let's see this in the case of the /add endpoint:

add.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var ResultService = require("../services/ResultService")
var resultService = new ResultService(db);
var jwt = require('jsonwebtoken')

router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    const result = number1 + number2;
    const token = req.headers.authorization?.split(' ')[1];
    if(token) {
        const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
        resultService.create("add", result, decodedToken.id);
    }
    res.jsend.success({"result": result});
});

module.exports = router;
```

After calculating the result, we check whether the token was set in the authorization header. If it was, we decode it and use the result service to save the result to the database.

Let's test it with Postman. First, log in and copy the token to the Authorization section. Choose the Bearer Token option:



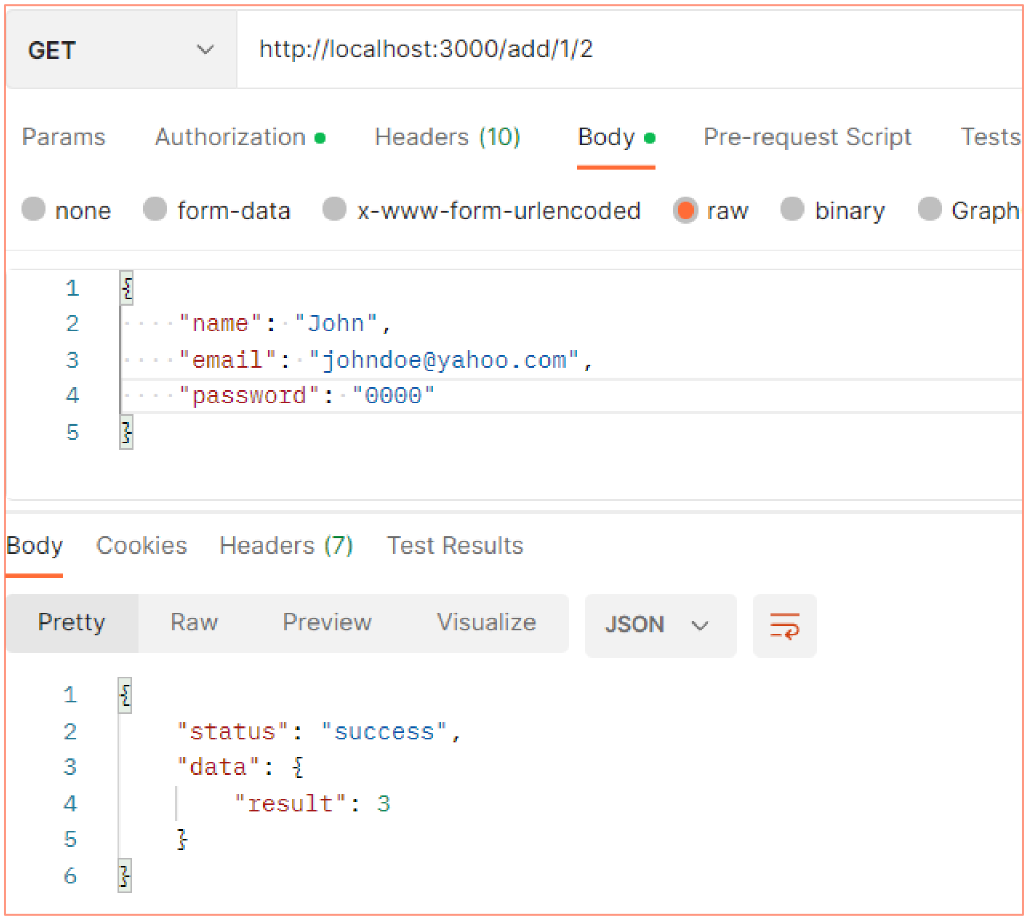
Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collabora
recommend using variables. Learn more about variables ↗

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey...

Next, make a request to http://localhost:3000/add/1/2:

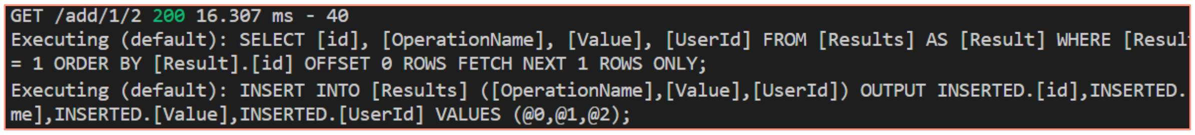


http://localhost:3000/add/1/2

```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password": "0000"
}
```

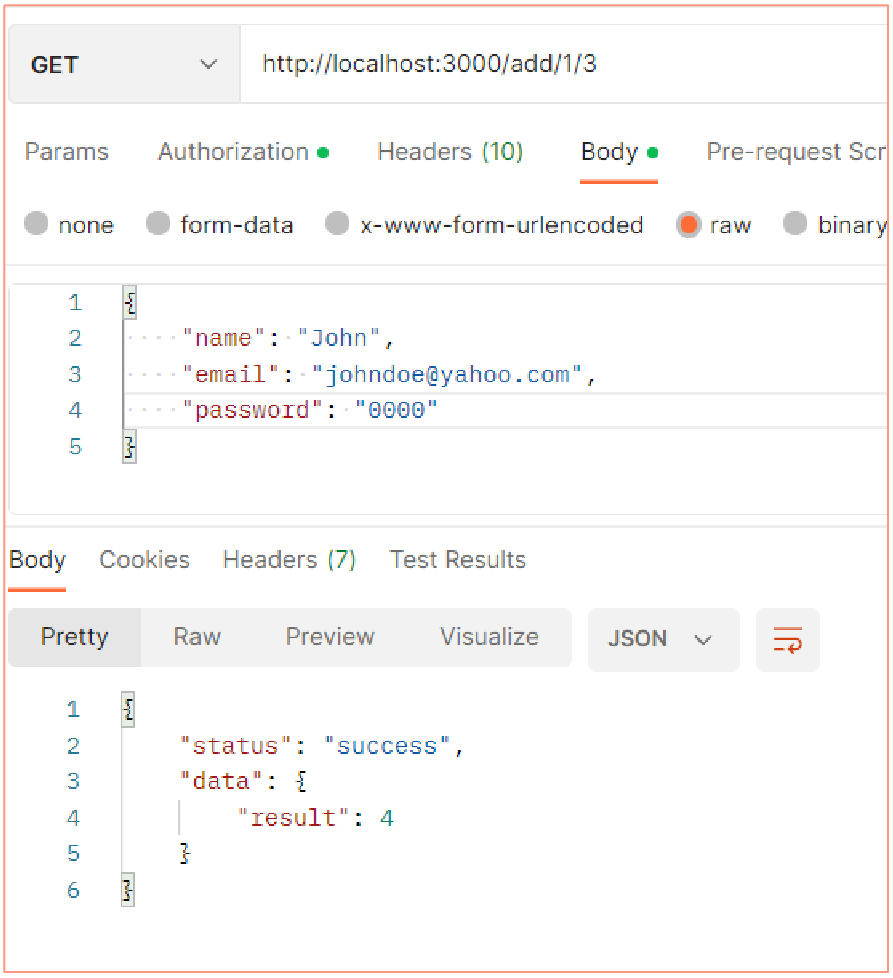
```
{
    "status": "success",
    "data": {
        "result": 3
    }
}
```

We can see in the logs (or from MySQL Workbench) that the record has been correctly added to the results table:



```
GET /add/1/2 200 16.307 ms - 40
Executing (default): SELECT [id], [OperationName], [Value], [UserId] FROM [Results] AS [Result] WHERE [Resul
= 1 ORDER BY [Result].[id] OFFSET 0 ROWS FETCH NEXT 1 ROWS ONLY;
Executing (default): INSERT INTO [Results] ([OperationName],[Value],[UserId]) OUTPUT INSERTED.[id],INSERTED.
me],INSERTED.[Value],INSERTED.[UserId] VALUES (@0,@1,@2);
```

Now, let's make a request to http://localhost:3000/add/1/3:

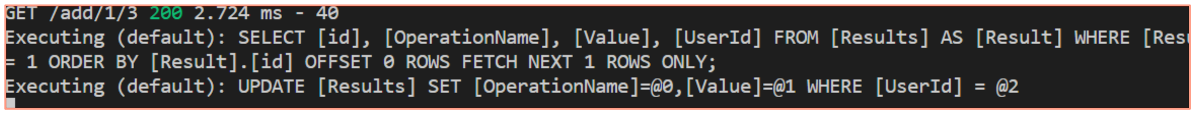


http://localhost:3000/add/1/3

```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password": "0000"
}
```

```
{
    "status": "success",
    "data": {
        "result": 4
    }
}
```

We can see that the record has been updated instead of inserted:



```
GET /add/1/3 200 2.724 ms - 40
Executing (default): SELECT [id], [OperationName], [Value], [UserId] FROM [Results] AS [Result] WHERE [Res
= 1 ORDER BY [Result].[id] OFFSET 0 ROWS FETCH NEXT 1 ROWS ONLY;
Executing (default): UPDATE [Results] SET [OperationName]=@0,[Value]=@1 WHERE [UserId] = @2
```

Try to implement the same behaviour (save the operation to the database if the user was authenticated and it finished successfully) for the /subtract, /multiply, and /divide endpoints on your own. The results are presented below:

subtract.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var ResultService = require("../services/ResultService")
var resultService = new ResultService(db);
var jwt = require('jsonwebtoken')
```

```
router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    const result = number1 - number2;
    const token = req.headers.authorization?.split(' ')[1];
    if(token) {
        const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
        resultService.create("subtract", result,
decodedToken.id);
    }
    res.jsend.success(result);
});

module.exports = router;
```

multiply.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var ResultService = require("../services/ResultService")
var resultService = new ResultService(db);
var jwt = require('jsonwebtoken')

router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    const result = number1 * number2;
    const token = req.headers.authorization?.split(' ')[1];
    if(token) {
        const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
        resultService.create("multiply", result,
decodedToken.id);
    }
    res.jsend.success({"result": result});
});

module.exports = router;
```

divide.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var ResultService = require("../services/ResultService")
var resultService = new ResultService(db);
var jwt = require('jsonwebtoken')

router.use(jsend.middleware);
router.get('/:number1/:number2', function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    if(number2 == 0) {
        return res.jsend.fail({"number2": "number2 cannot be
0"});
    }
    const result = number1 / number2;
    const token = req.headers.authorization?.split(' ')[1];
    if(token) {
        const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
        resultService.create("divide", Math.round(result),
decodedToken.id);
    }
    if (Number.isInteger(result)) {
        res.jsend.success({"result": result});
    }
    else {
        res.jsend.success({"result": Math.round(result),
"message": "Result has been rounded, as it was not an
integer."});
    }
});

module.exports = router;
```

Having done that, we can move on to implement four new endpoints. Create the previous.js router file, in which we will create these handlers for the endpoints:

- /add
- /subtract
- /multiply
- /divide

Link it in app.js:

```
var previousRouter = require('./routes/previous');
//...
app.use('/previous', previousRouter);
```

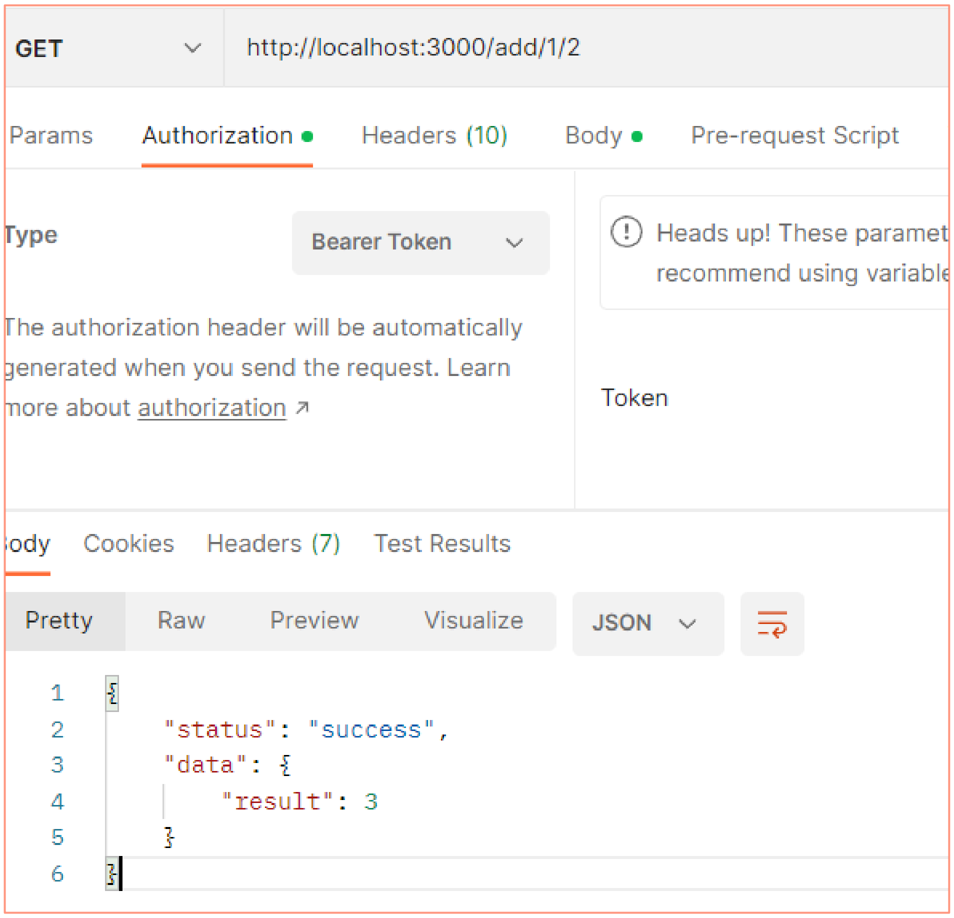
Let’s implement the /previous/add/:number1 endpoint:

previous.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var ResultService = require("../services/ResultService")
var resultService = new ResultService(db);
var jwt = require('jsonwebtoken')

router.use(jsend.middleware);
router.get('/add/:number1', async function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    if (decodedToken == null) {
        return res.jsend.fail({"result": "JWT token is
invalid"});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value + number1;
    resultService.create("add", result, decodedToken.id);
    res.jsend.success({"result": result, "previousOperation":
previous.OperationName, "previousValue": previous.Value});
});

module.exports = router;
```

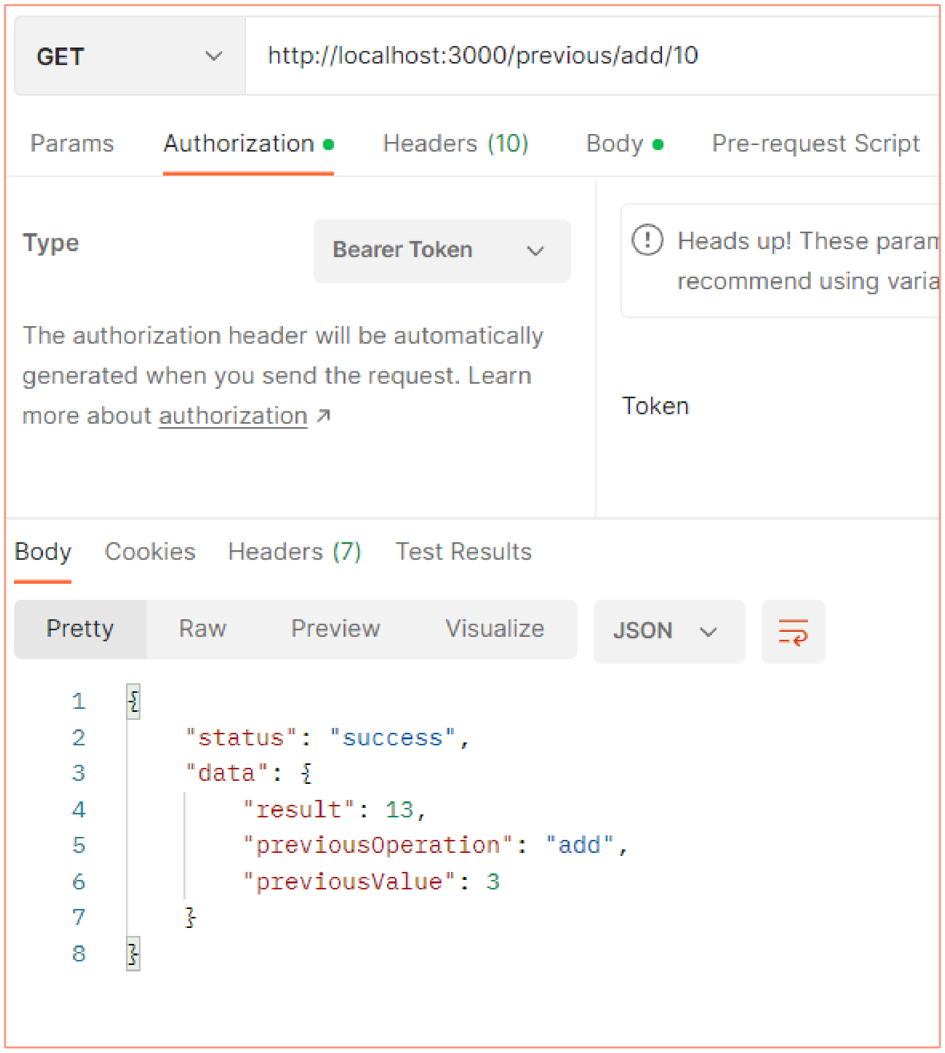
We take number1 from the route and parse it to an integer using parseInt(). Then, we decode the JWT token using verify(). If it is not valid, we use the fail() method. If everything works correctly, we use the result service to get the previous value from the database. We then use it to calculate the result. Next, we save the result in the database and send the response to the user.

Let's see this in a Postman example. First, we add two numbers:



```
{
    "status": "success",
    "data": {
        "result": 3
    }
}
```

Then we add 10:


GET http://localhost:3000/previous/add/10

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These param
recommend using varia

Token

```
{
    "status": "success",
    "data": {
        "result": 13,
        "previousOperation": "add",
        "previousValue": 3
    }
}
```

Now, implement the rest of the endpoints. The final previous.js file, including all the endpoints, is presented below:

previous.js:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var ResultService = require("../services/ResultService")
var resultService = new ResultService(db);
var jwt = require('jsonwebtoken')

router.use(jsend.middleware);
router.get('/add/:number1', async function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    if (decodedToken == null) {
        return res.jsend.fail({"result": "JWT token is
invalid"});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value + number1;
    resultService.create("add", result, decodedToken.id);
    res.jsend.success({"result": result, "previousOperation":
previous.OperationName, "previousValue": previous.Value});
});

router.get('/subtract/:number1', async function(req, res,
next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    if (decodedToken == null) {
        return res.jsend.fail({"result": "JWT token is
invalid"});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value - number1;
    resultService.create("subtract", result,
decodedToken.id);
    res.jsend.success({"result": result, "previousOperation":
previous.OperationName, "previousValue": previous.Value});
});

router.get('/multiply/:number1', async function(req, res,
next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    if (decodedToken == null) {
        return res.jsend.fail({"result": "JWT token is
invalid"});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value * number1;
    resultService.create("multiply", result,
decodedToken.id);
    res.jsend.success({"result": result, "previousOperation":
previous.OperationName, "previousValue": previous.Value});
});

router.get('/divide/:number1', async function(req, res, next)
{
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    if(number1 == 0) {
        return res.jsend.fail({"number1": "number1 cannot be
0"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    if (decodedToken == null) {
        return res.jsend.fail({"result": "JWT token is
invalid"});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value / number1;
    resultService.create("divide", Math.round(result),
decodedToken.id);
    if (Number.isInteger(result)) {
        res.jsend.success({"result": result,
"previousOperation": previous.OperationName, "previousValue":
previous.Value});
    }
    else {
        res.jsend.success({"result": Math.round(result),
"previousOperation": previous.OperationName, "previousValue":
previous.Value, "message": "Result has been rounded, as it
was not an integer."});
    }
});
```

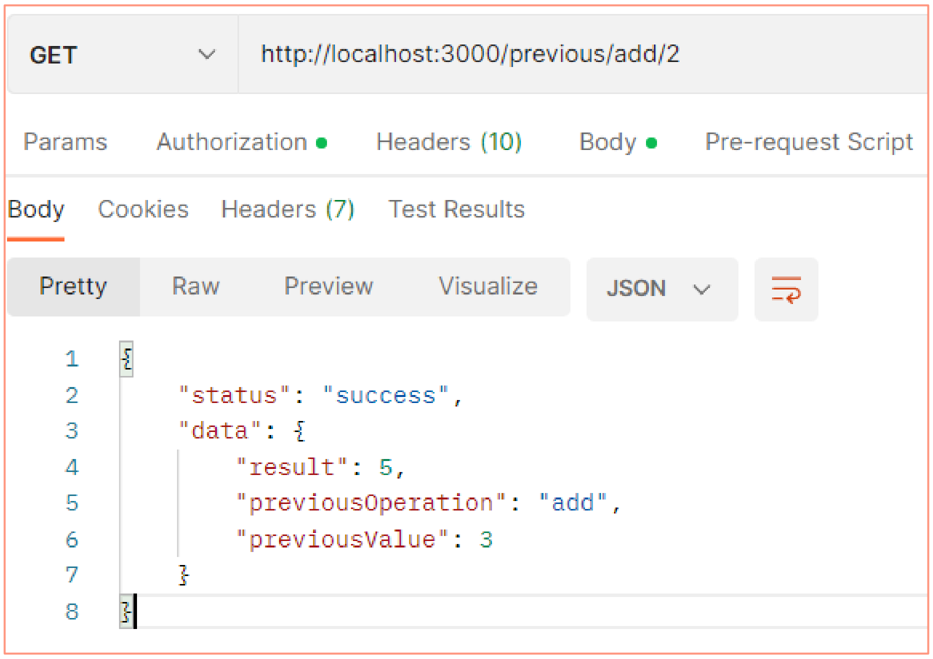
```
module.exports = router;
```

Testing with Postman

Let's test the newly created endpoints in Postman.

First, set the JWT token in Postman as in the previous examples. Then, make a request to the /add/1/2 endpoint. It will return 3, and it has already been tested. Knowing that 3 is saved as the previous value, let's test all of the new endpoints:

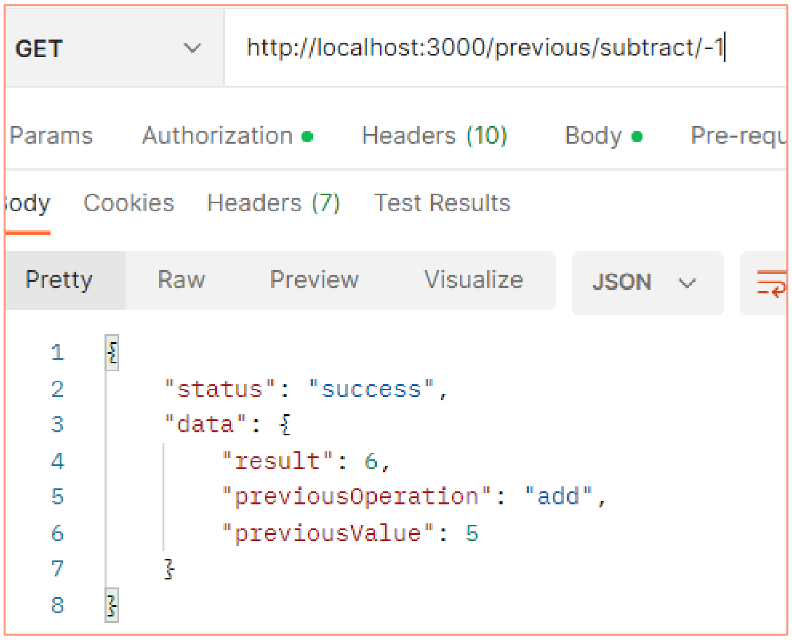
/previous/add/2:



```
{
    "status": "success",
    "data": {
        "result": 5,
        "previousOperation": "add",
        "previousValue": 3
    }
}
```

3 + 2 = 5

/previous/subtract/-1:

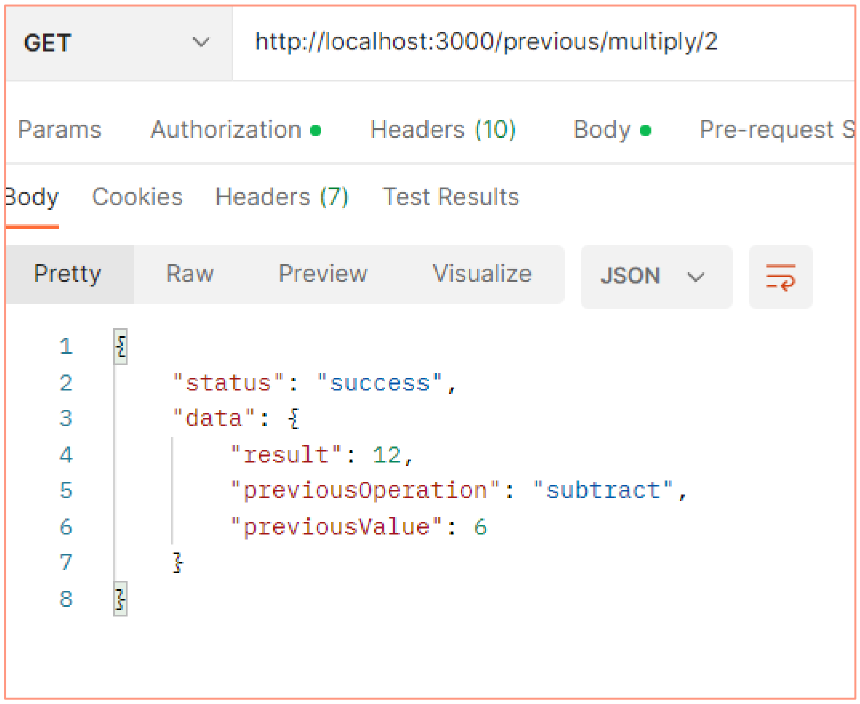


http://localhost:3000/previous/subtract/-1

```
{
    "status": "success",
    "data": {
        "result": 6,
        "previousOperation": "add",
        "previousValue": 5
    }
}
```

5 – (-1) = 6

previous/multiply/2:

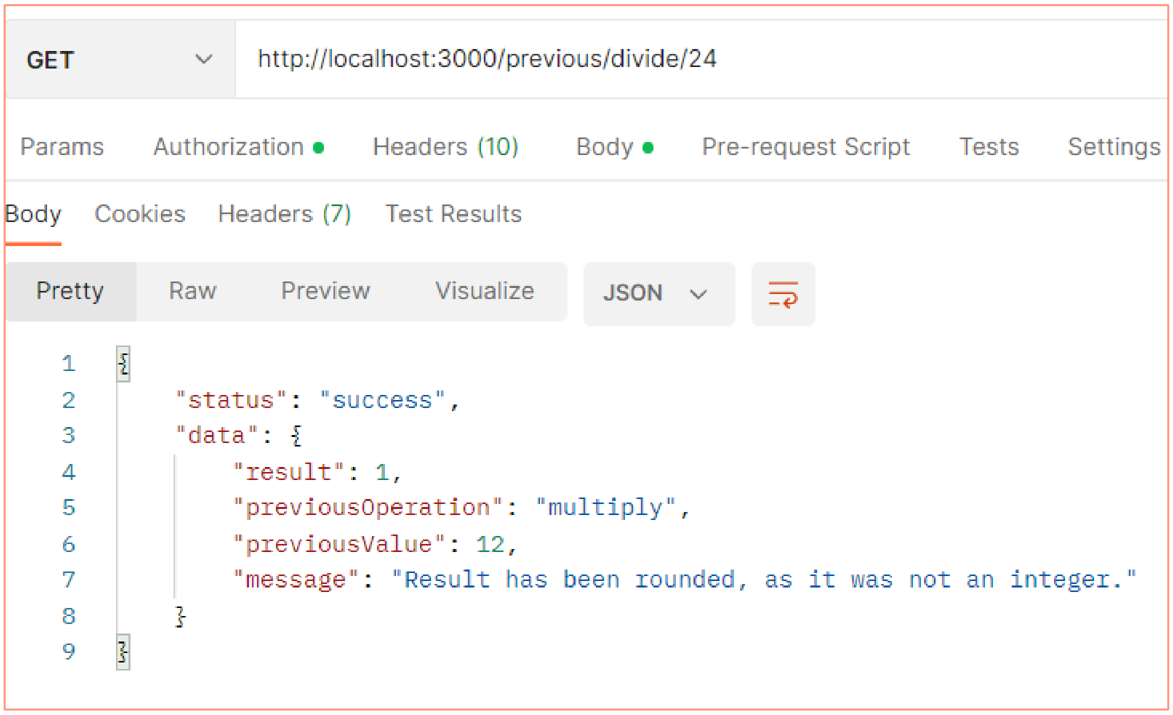


http://localhost:3000/previous/multiply/2

```
{
    "status": "success",
    "data": {
        "result": 12,
        "previousOperation": "subtract",
        "previousValue": 6
    }
}
```

6*2 = 12

previous/divide/24:



```
http://localhost:3000/previous/divide/24
```

```json
{
    "status": "success",
    "data": {
        "result": 1,
        "previousOperation": "multiply",
        "previousValue": 12,
        "message": "Result has been rounded, as it was not an integer."
    }
}
```

12 / 24 = 0.5 ~ 1

We can see that all the endpoints work as expected.

What did I learn in this lesson?

This lesson provided the following insights:

- How to use one-to-one relationships.
- How to use JWT authentication in practice.
- How to use authentication to personalise users' experience.

References

INFO

Please note that the reference list below was compiled when writing the course content to credit the sources used. Therefore, some links that worked at the time of writing may not work now or may have changed. However, accessing these references is not a requirement for this module. Note that all the required reading is specified in the READ blocks in the content.

sequelize.org. (n.d.). *Getting Started* | *Sequelize.* [online] Available at: https://sequelize.org/docs/v6/getting-started/.

5.2. Lesson task - Authentication

The task

In this lesson, we learnt how to create and use JWT authentication to personalise an API. In this task, you need to implement the following functionality (only for the sake of this task; remove it before the next lesson):

If the previous result is a multiplicity of the user email length and the request was successful, send the message “You are lucky” in the JSON response. Apply this only to requests to the endpoints using the “previous” value.

5.3. Lesson - Project documentation

Introduction

Our Calculator API is working correctly, but if we want other users to be able to use it, we should provide proper documentation. Currently, it is not likely that a user will know:

- The route parameters must be named number1 and number2
- The calculator works only with integers
- What application behaviour to expect in the case of an invalid request

Proper documentation can solve all of the abovementioned issues and more. We will create new documentation in Postman.

First, a new collection will be created from scratch, including our API requests. We will do this manually – we could base it on the Swagger documentation generated with external software, but this was already covered in previous lessons.

Next, we will add at least one negative and one positive request example to the collection, write a proper introduction, and publish the documentation.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

→ The candidate has knowledge of industry-standard JavaScript frameworks that can be used to build RESTful services.

In this lesson, we are covering the following skill learning outcome:

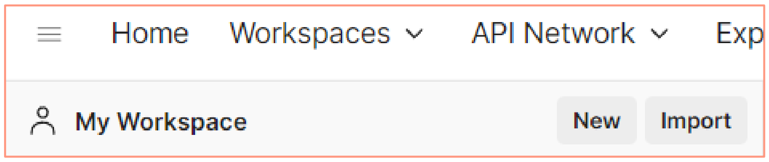
→ The candidate masters the Postman software (or open-source alternatives) to test and document a REST API.

In this lesson, we are covering the following general competence learning outcome:

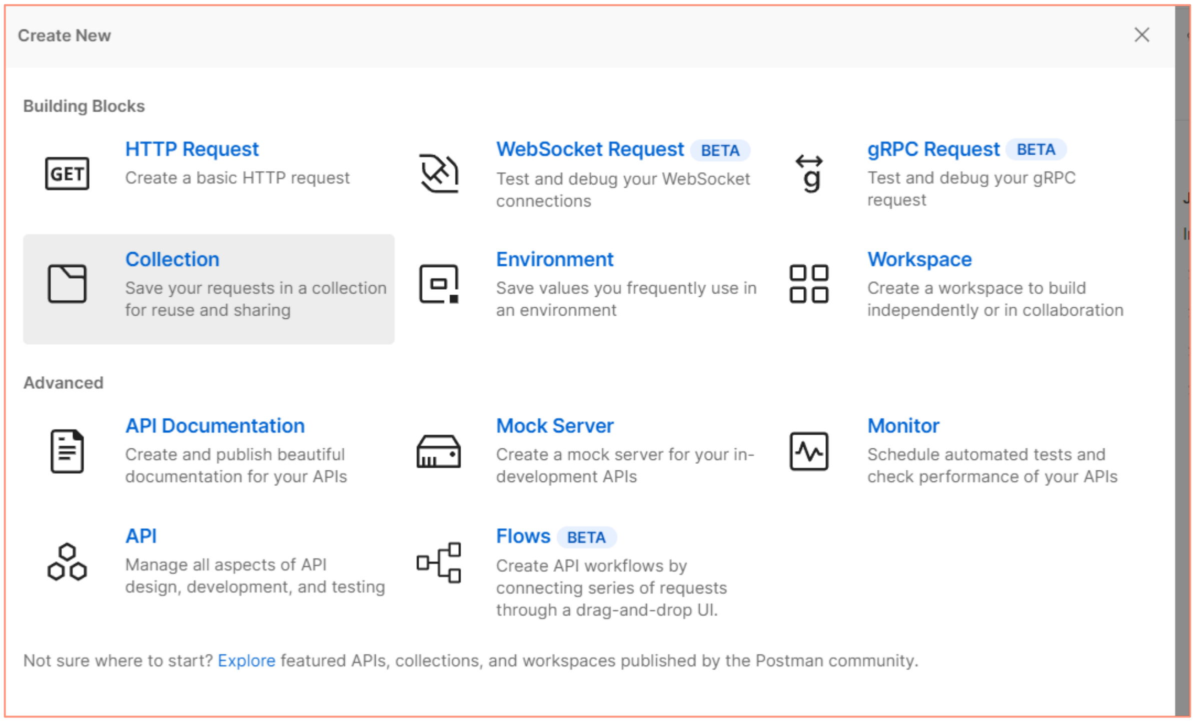
→ The candidate can carry out work as a REST API developer, maintainer, or tester.

Creating a collection

Let's create Postman documentation for our application. Open Postman and open any workspace (the view from where you send requests). To create a new collection, click the "New" button:



Choose the "Collection" option (we could also create the documentation by choosing the API documentation option; however, we would have less space for customisation):


Create New

Building Blocks

HTTP Request Create a basic HTTP request

WebSocket Request BETA Test and debug your WebSocket connections

gRPC Request BETA Test and debug your gRPC request

Collection Save your requests in a collection for reuse and sharing

Environment Save values you frequently use in an environment

Workspace Create a workspace to build independently or in collaboration

Advanced

API Documentation Create and publish beautiful documentation for your APIs

Mock Server Create a mock server for your in-development APIs

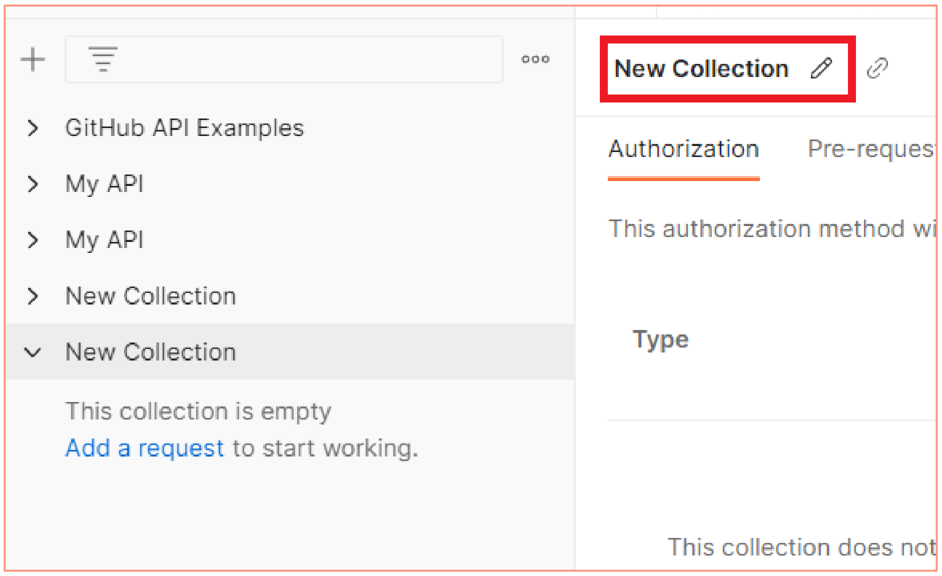
Monitor Schedule automated tests and check performance of your APIs

API Manage all aspects of API design, development, and testing

Flows BETA Create API workflows by connecting series of requests through a drag-and-drop UI.

Not sure where to start? Explore featured APIs, collections, and workspaces published by the Postman community.

Name the collection “Calculator” – change the name in the space indicated by the red border below:

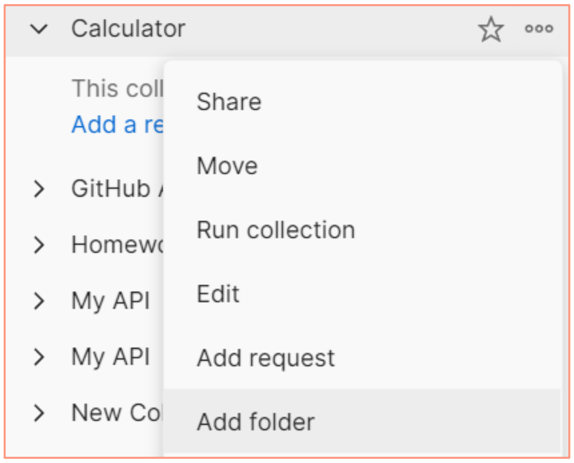


This collection is empty
Add a request to start working.

This authorization method wi

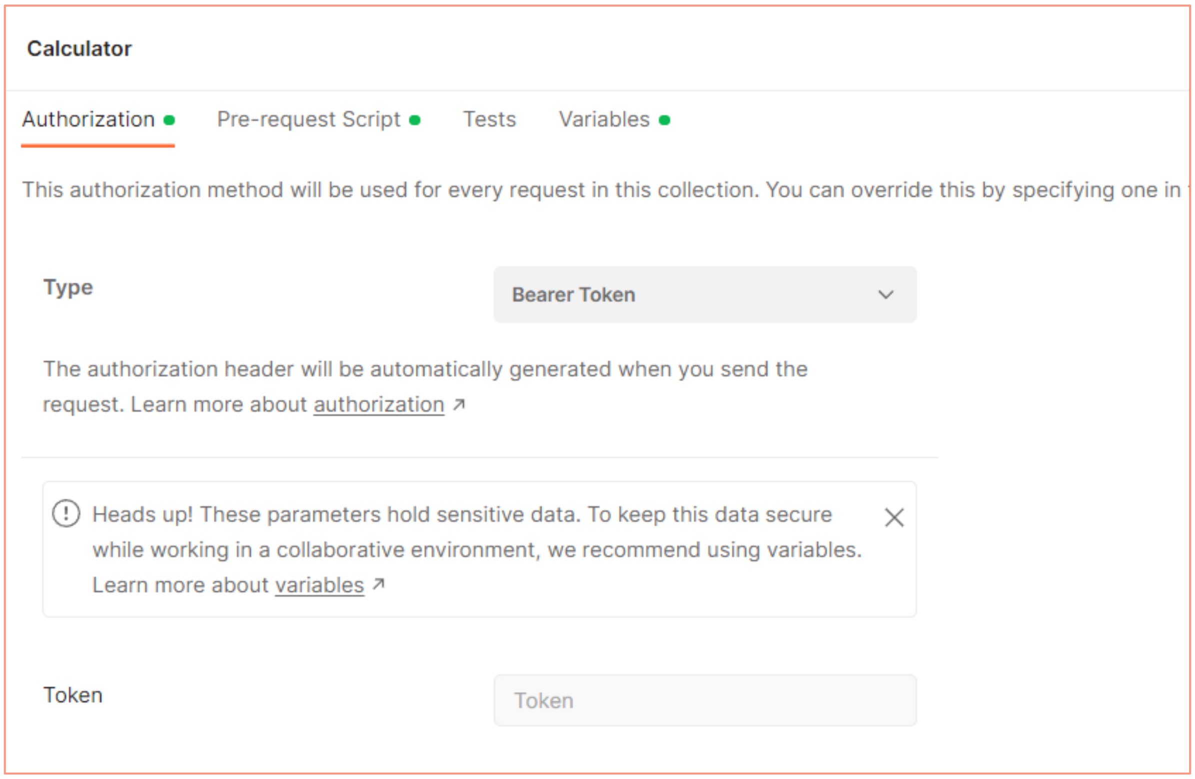
This collection does not

Add three folders to the collection by clicking the triple dot icon and choosing “Add folder”:



- Auth – Here we will add endpoints responsible for authentication.
- Basic operations – Here we will add options available without authentication.
- Operations with previous value – Here we will add options available only after authentication.

After clicking on the “Calculator” tab, we will see a window with authorization, pre-request scripts, tests, and variables. For now, set the authorization type to “Bearer Token”, but the token field should not be filled yet:



Calculator

Authorization ● Pre-request Script ● Tests Variables ●

This authorization method will be used for every request in this collection. You can override this by specifying one in

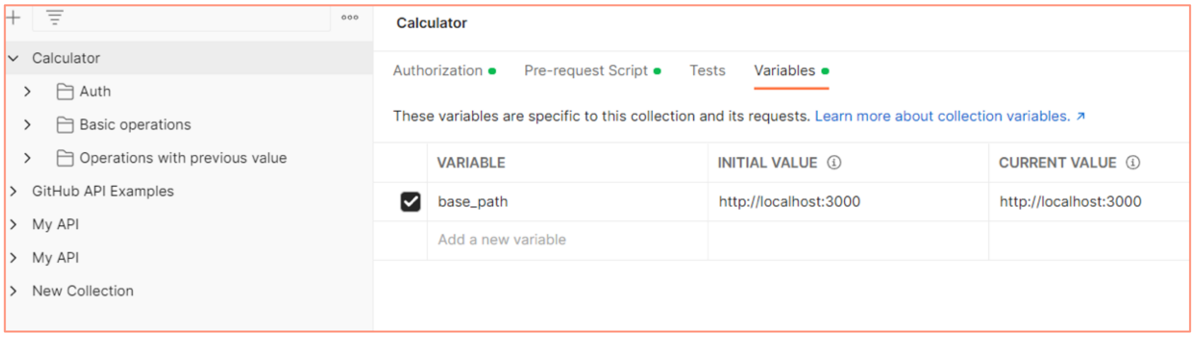
Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables ↗

Token: Token

Next, let's add a variable for the base host path to our application (http://localhost:3000):



These variables are specific to this collection and its requests. Learn more about collection variables. ↗

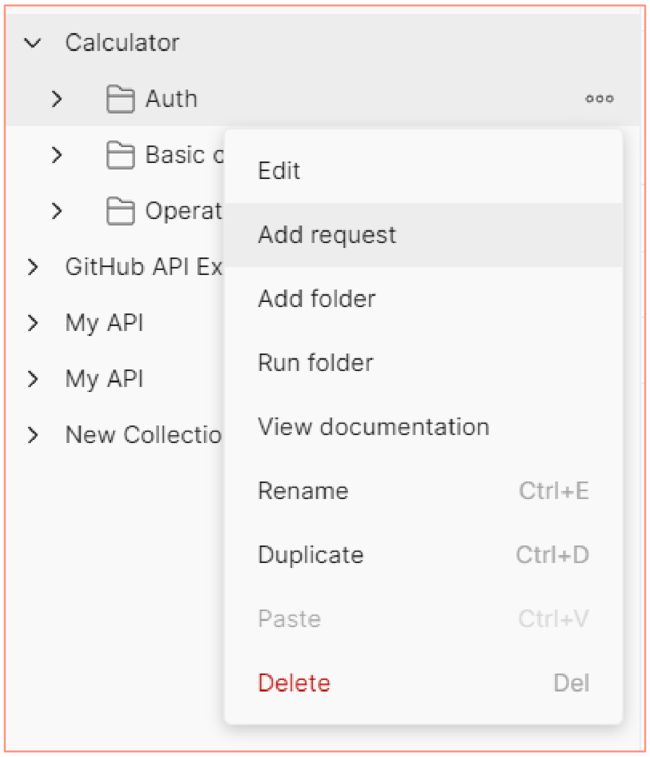
	VARIABLE	INITIAL VALUE	CURRENT VALUE
✓	base_path	http://localhost:3000	http://localhost:3000
	Add a new variable		

To access it, we will call it with double brackets – {{base_path}}. We can access it everywhere (for example, in the URL and request body) in the Calculator collection.

The data set here is available from all child requests and folders, so we will be able to use this variable in the next section, where we will add requests to the folders.

Adding endpoints

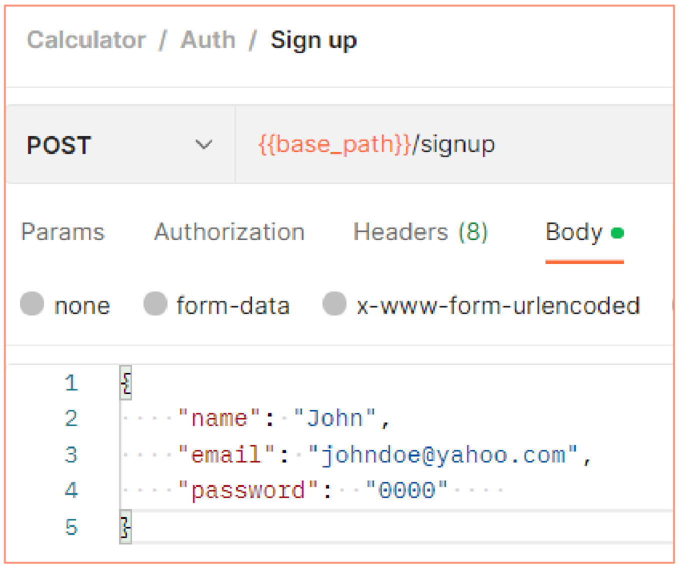
Let's open the created folders and add the implemented endpoints. To add a request to a folder, click the triple dot icon and choose the "Add request" option:



Add the following requests to the collection:

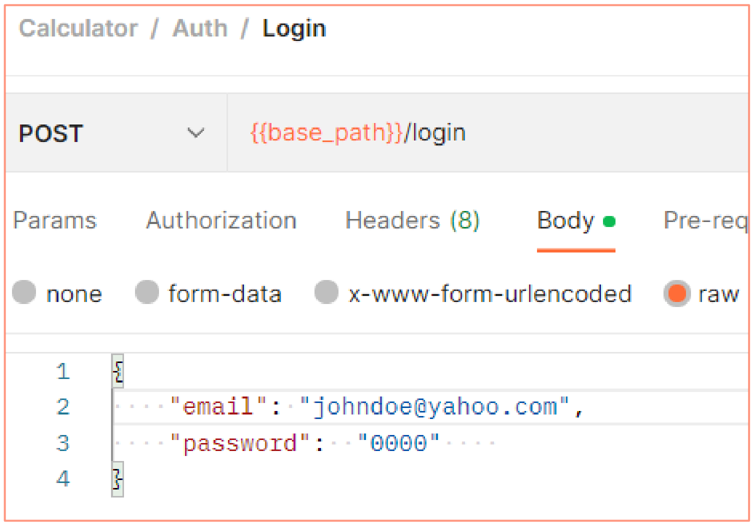
- Auth:
  - POST /login – Log in
  - POST /signup – Sign up
- Basic operations:
  - GET /add/:number1/:number2 – Add
  - GET /subtract/:number1/:number2 – Subtract
  - GET /multiply/:number1/:number2 – Multiply
  - GET /divide/:number1/:number2 – Divide
- Operations with the previous value:
  - GET /previous/add/:number1 – Add to previous
  - GET /previous/subtract/:number1 – Subtract from previous
  - GET /previous/multiply/:number1 – Multiply previous
  - GET /previous/divide/:number1 – Divide previous

Let's start with the Auth folder. Our requests should look as follows:



```
{{base_path}}/signup
```

```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password":  "0000"    
}
```


Calculator / Auth / Login

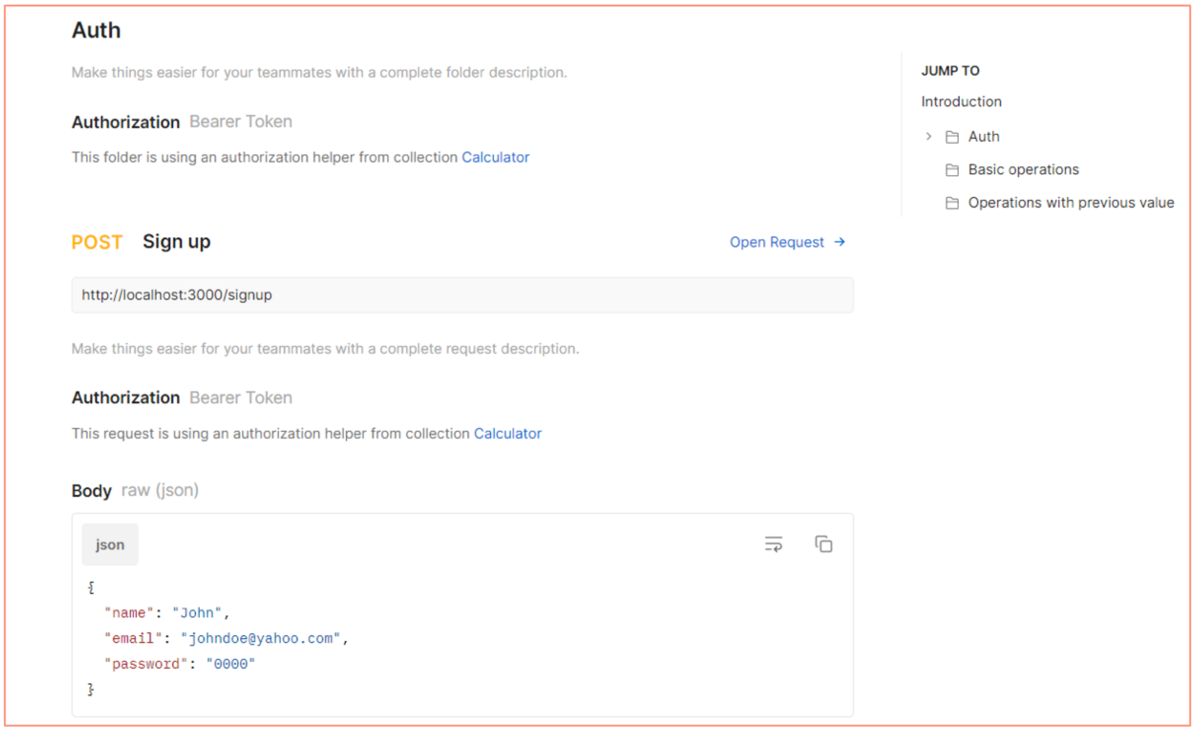
POST {{base_path}}/login

Params Authorization Headers (8) Body Pre-req

none form-data x-www-form-urlencoded [✓ raw]

```
{
    "email": "johndoe@yahoo.com",
    "password": "0000"
}
```

We can see what this folder looks like in the current documentation by right-clicking on the Auth folder and choosing “View documentation”:


Auth

Make things easier for your teammates with a complete folder description.

Authorization Bearer Token

This folder is using an authorization helper from collection Calculator

JUMP TO

Introduction

Auth

Basic operations

Operations with previous value

POST Sign up Open Request →

http://localhost:3000/signup

Make things easier for your teammates with a complete request description.

Authorization Bearer Token

This request is using an authorization helper from collection Calculator

Body raw (json)

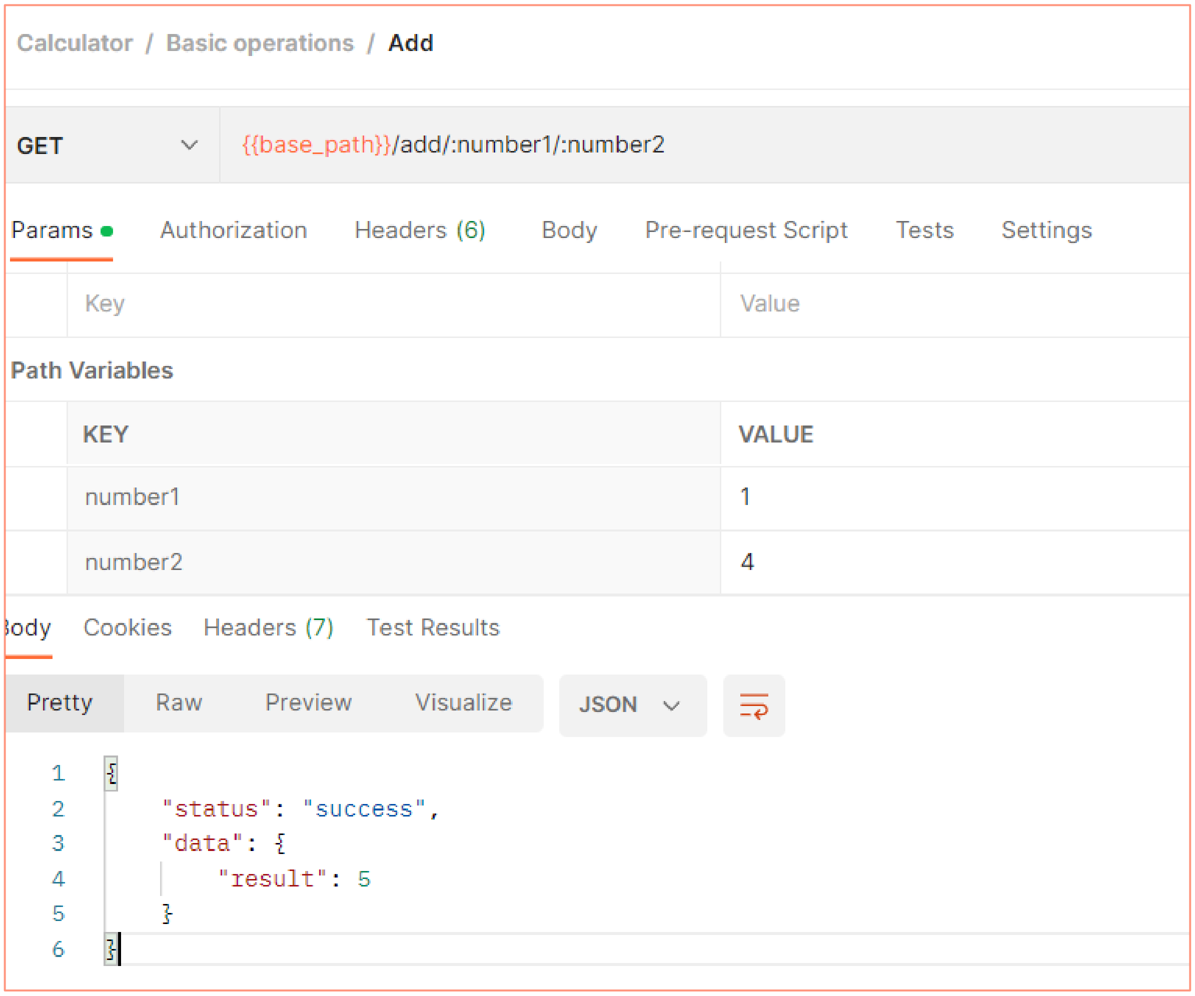
json

```
{
  "name": "John",
  "email": "johndoe@yahoo.com",
  "password": "0000"
}
```

It already contains the most important information – we will improve it in the following sections.

Next, let’s add requests for the rest of the folder. Below you’ll see /add and /previous/add (the rest are very similar):

Basic operations / Add:



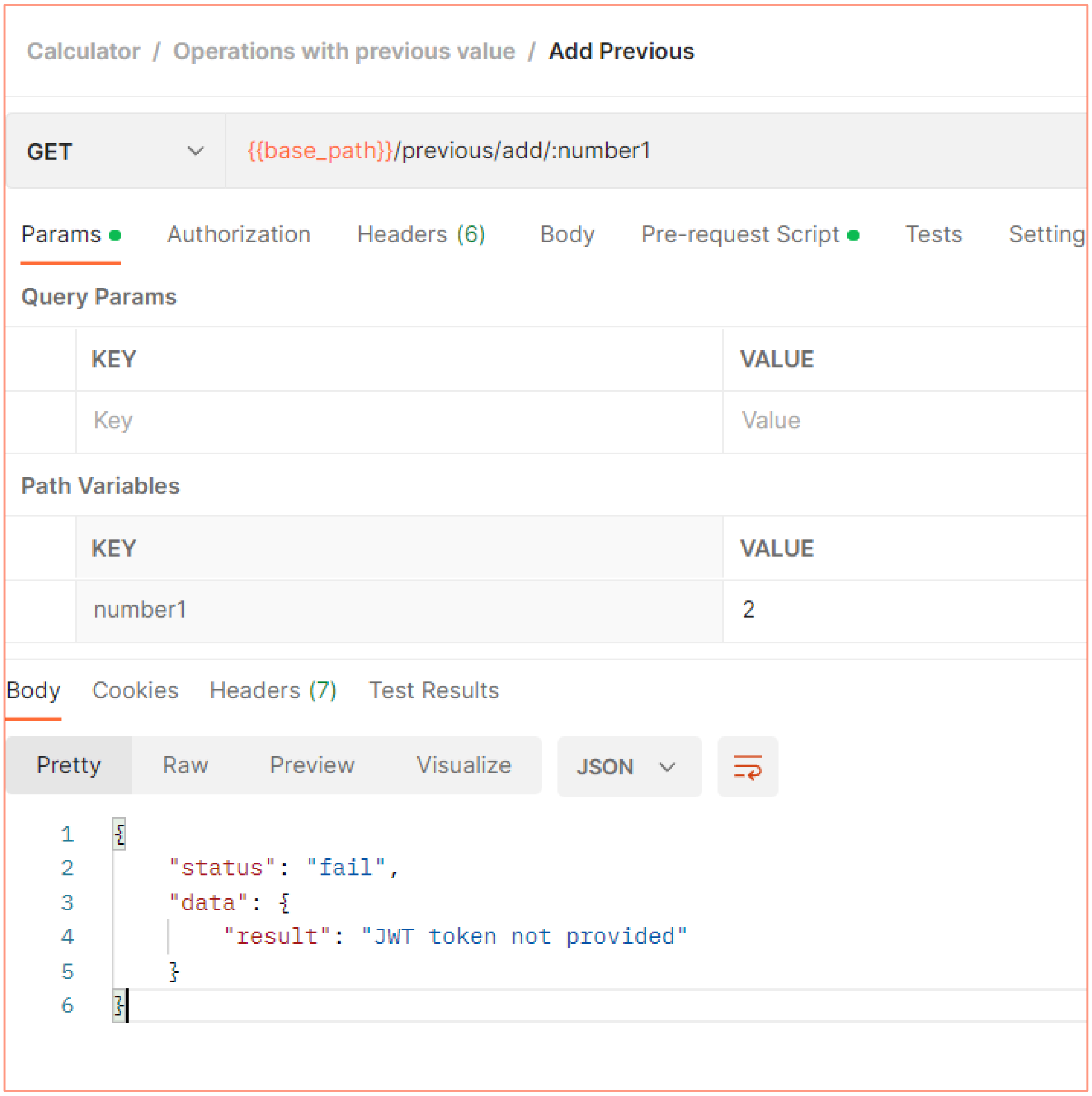
```
GET {{base_path}}/add/:number1/:number2
```

KEY	VALUE
number1	1
number2	4

```
{
    "status": "success",
    "data": {
        "result": 5
    }
}
```

In the case of requests to the /previous endpoint, they are more complex, as we need to authenticate ourselves before sending the request. We will skip this part for now and assume that we are logged in – we will focus more on this in the next section, where examples of each request will be created.

For now, it looks as follows:


Calculator / Operations with previous value / Add Previous

GET {{base_path}}/previous/add/:number1

Query Params

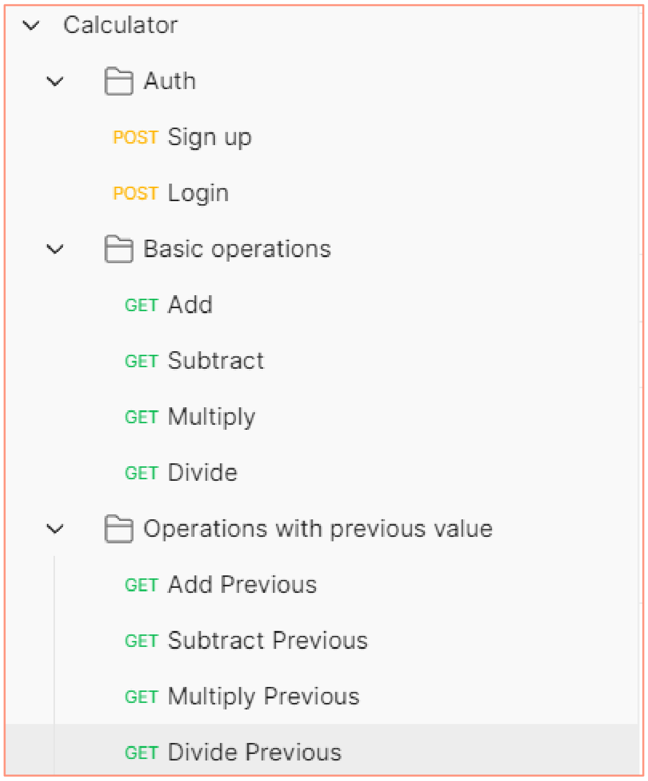
KEY	VALUE
Key	Value

Path Variables

KEY	VALUE
number1	2

```
{
    "status": "fail",
    "data": {
        "result": "JWT token not provided"
    }
}
```

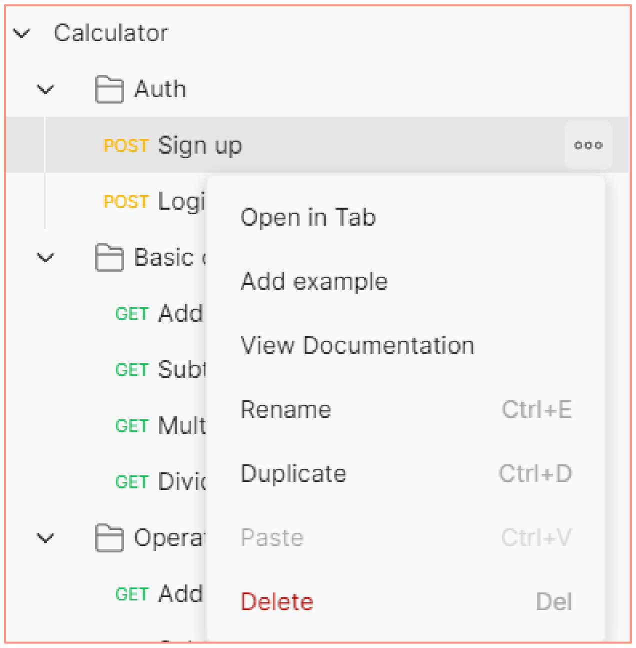
After adding all the requests, our collection should look as follows:



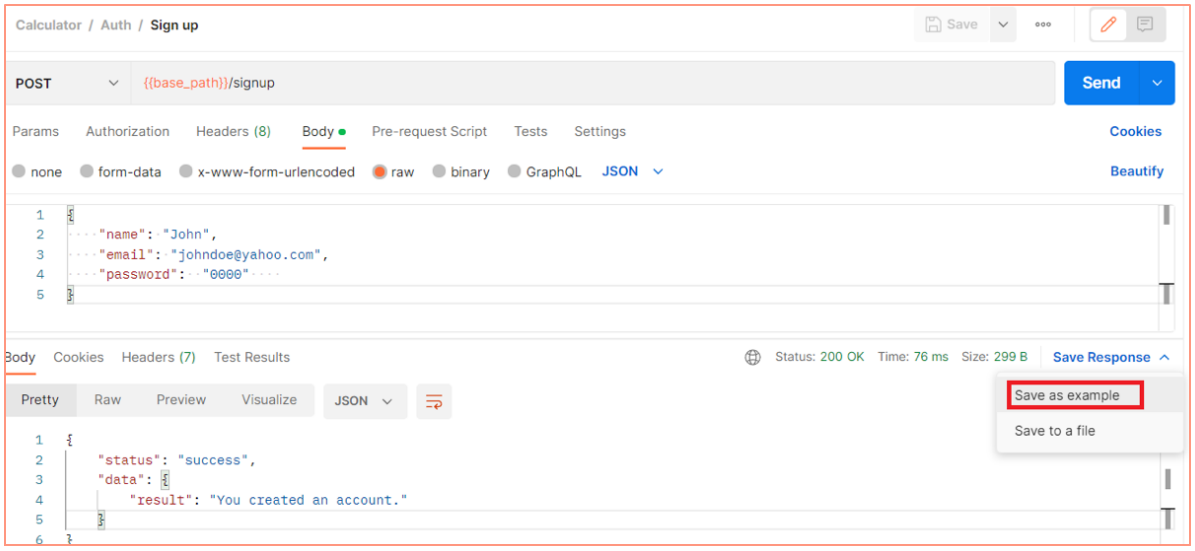
Adding examples

Examples are very useful to include in documentation. They show how to execute a request and what response can be expected, so readers don't necessarily have to read the entire endpoint description.

We can add as many examples to our requests as we want. One way to add an example is to right-click on the request and choose the “Add example” option:



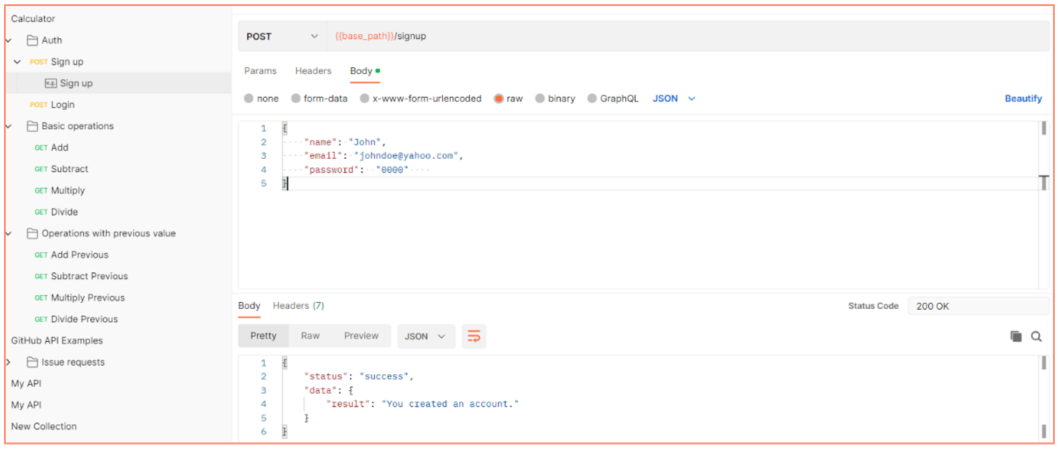
However, there is an easier way – we can open the request in a tab, send it as usual, and save the response as an example. To do this, we need to click the "Save response" option on the right and choose "Save as example":



```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password": "0000"
}
```

```
{
    "status": "success",
    "data": {
        "result": "You created an account."
    }
}
```

We should see the saved example:



POST {{base_path}}/signup

```
{
    "name": "John",
    "email": "johndoe@yahoo.com",
    "password":  "0000"
}
```

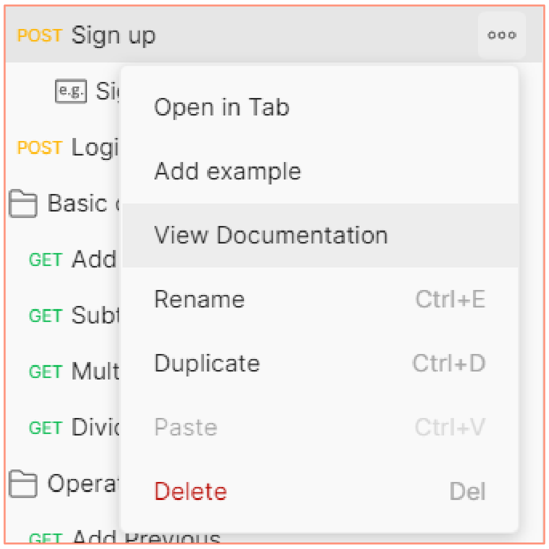
Body Headers (7) Status Code 200 OK

```
{
    "status": "success",
    "data": {
        "result": "You created an account."
    }
}
```

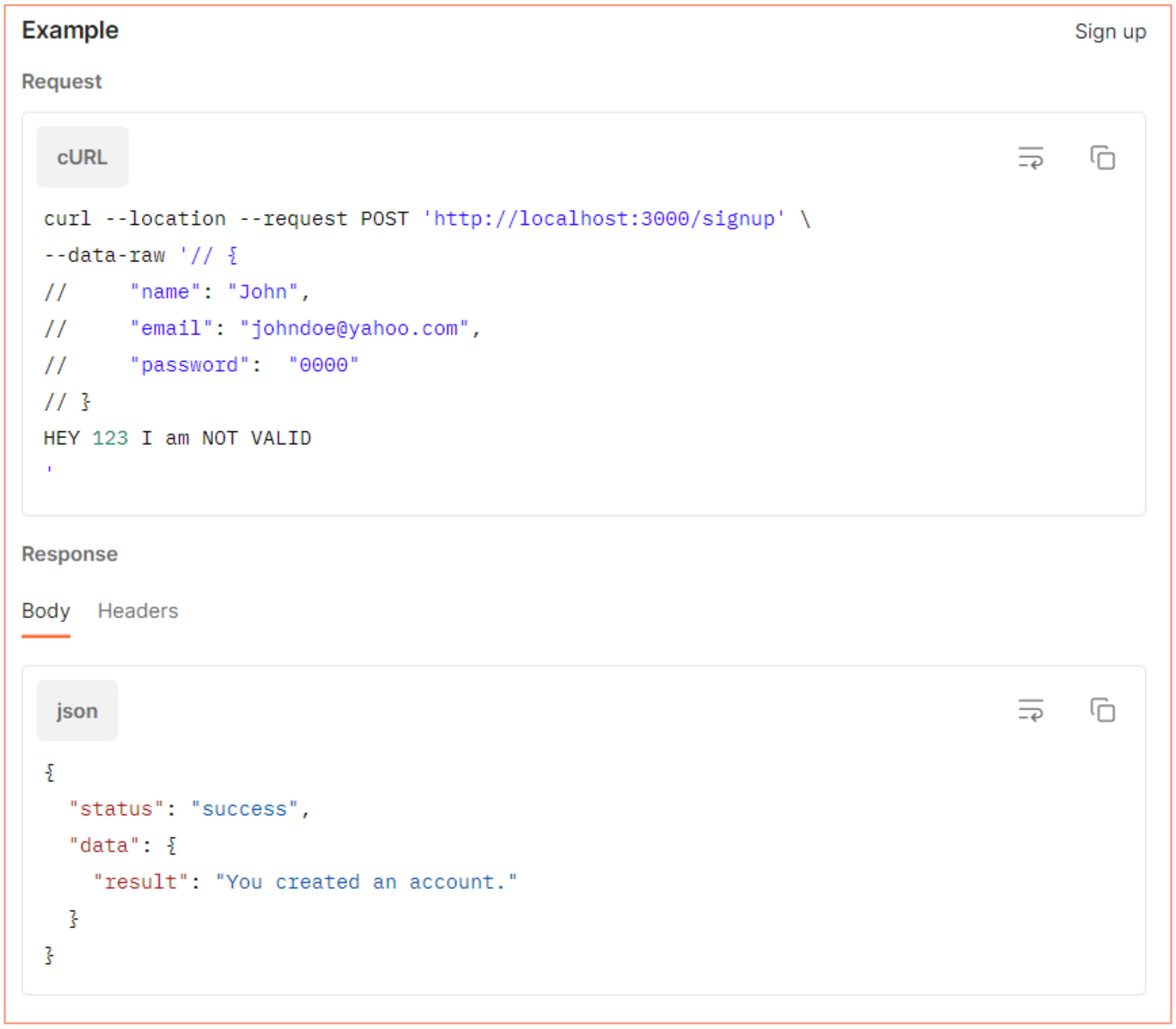
Notice that there is no option to send the request. The only purpose of examples is to improve the documentation – the examples do not even have to work properly. To demonstrate this, temporarily change the body of the example to invalid JSON (comment out the correct JSON and add some invalid text):

```
// {
//    "name": "John",
//      "email": "johndoe@yahoo.com",
//      "password":   "0000"
// }
HEY 123 I am NOT VALID
```

Next, open the documentation view:



After scrolling to the example, we should see that the response is still correct, even though a request with such a body would never work:



```
curl --location --request POST 'http://localhost:3000/signup' \
--data-raw '// {
//      "name": "John",
//      "email": "johndoe@yahoo.com",
//      "password":  "0000"
// }
HEY 123 I am NOT VALID
'
```

```
{
  "status": "success",
  "data": {
    "result": "You created an account."
  }
}
```

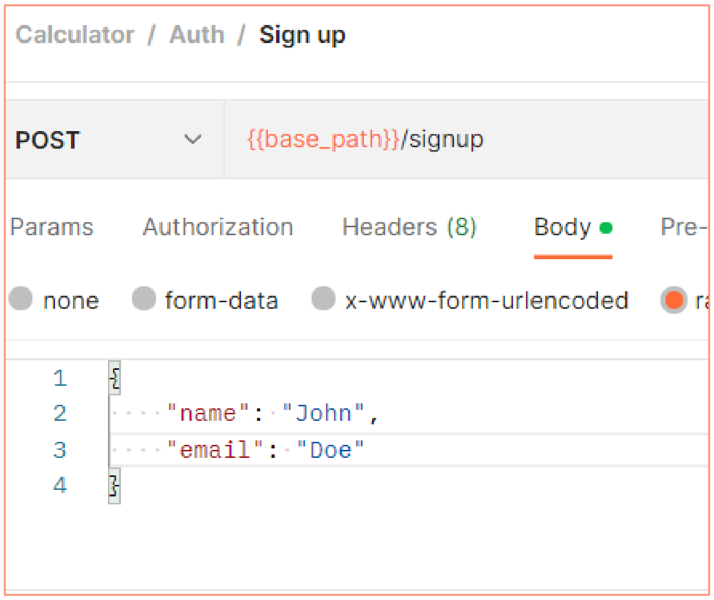
If this POST request were made to the API, an error would occur (the body is not valid, so it would not be processed properly by the server) – but for the purposes of the documentation, the response will be shown.

Let's revert these changes and create two request examples for each endpoint (by saving the responses) – one valid and one invalid. As request examples only demonstrate a response and don't send it actively, we don't need to worry about our JWT token expiration – so in the case of Operations with the previous value (the successful examples), log in first and set the proper header.

Now let's create one positive and negative request for each endpoint.

- Positive request – Returns the expected data.
- Negative request – Shows a possible failure response.

While creating the negative request for /signup, we received an error that stopped our application:



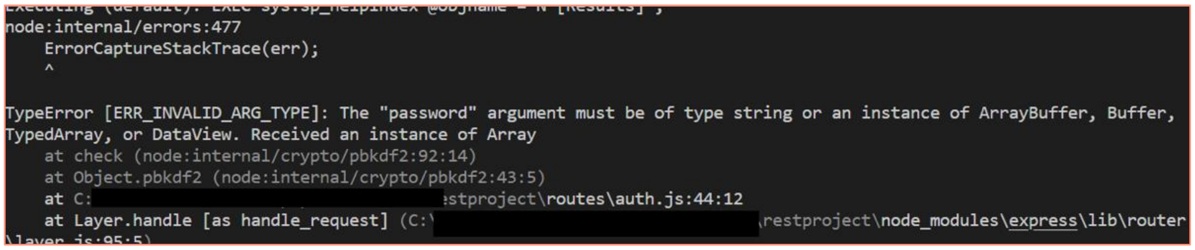
Calculator / Auth / Sign up

POST {{base_path}}/signup

Params Authorization Headers (8) Body Pre-

none form-data x-www-form-urlencoded ra

```
{
    "name": "John",
    "email": "Doe"
}
```



```
Executing (default): EXEC sys.sp_helpindex @objname = N'[Results]';
node:internal/errors:477
    ErrorCaptureStackTrace(err);
    ^

TypeError [ERR_INVALID_ARG_TYPE]: The "password" argument must be of type string or an instance of ArrayBuffer, Buffer,
TypedArray, or DataView. Received an instance of Array
    at check (node:internal/crypto/pbkdf2:92:14)
    at Object.pbkdf2 (node:internal/crypto/pbkdf2:43:5)
    at C:[redacted]stproject\routes\auth.js:44:12
    at Layer.handle [as handle_request] (C:\[redacted]\restproject\node_modules\express\lib\router
\layer.js:95:5)
```

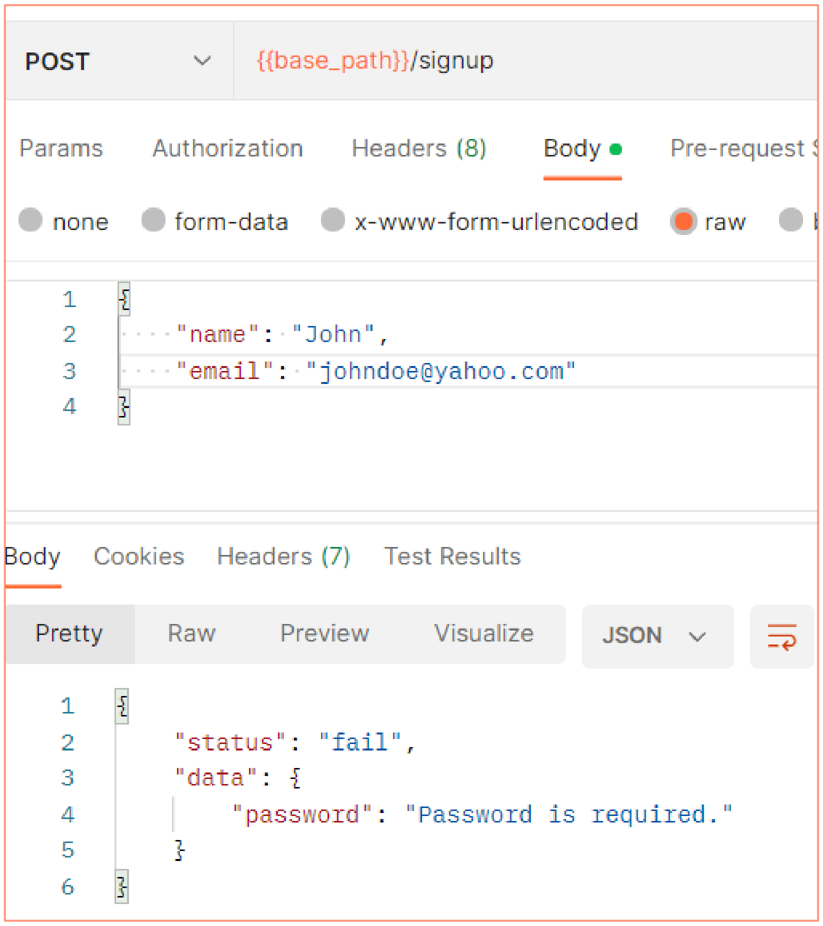
This error occurred because we have no validation on signup besides encrypting the password. Let’s add validation by checking if all fields were provided and whether the user of the provided email actually exists in the database:

auth.js:

```
  router.post("/signup", async (req, res, next) => {
    const { name, email, password } = req.body;
    if (name == null) {
      return res.jsend.fail({"name": "Name is required."});
    }
    if (email == null) {
      return res.jsend.fail({"email": "Email is required."});
    }
    if (password == null) {
      return res.jsend.fail({"password": "Password is
required."});
```

```
    }
    var user = await userService.getOne(email);
    if (user != null) {
      return res.jsend.fail({"email": "Provided email is
already in use."});
    }
    var salt = crypto.randomBytes(16);
    crypto.pbkdf2(password, salt, 310000, 32, 'sha256',
function(err, hashedPassword) {
      if (err) { return next(err); }
      userService.create(name, email, hashedPassword, salt)
      res.jsend.success({"result": "You created an
account."});
    });
});
```

Now, let’s go back to the negative example for signup:



```
{
    "name": "John",
    "email": "johndoe@yahoo.com"
}
```

```
{
    "status": "fail",
    "data": {
        "password": "Password is required."
    }
}
```

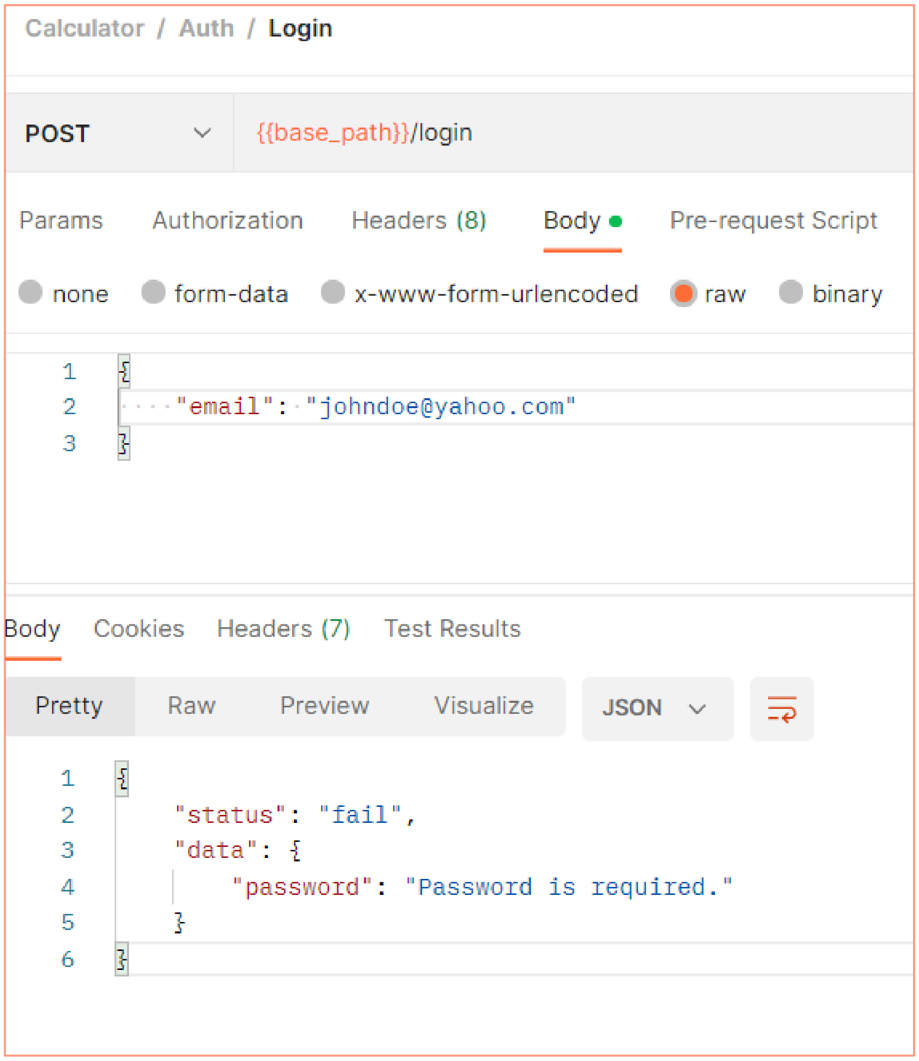
Save it as an example.

The /login endpoint has a similar problem to /signup in terms of values not being provided. Add validation to check that the email and password are not nulls:

auth.js:

```
router.post("/login", jsonParser, async (req, res, next) => {
    const { email, password } = req.body;
    if (email == null) {
      return res.jsend.fail({"email": "Email is required."});
    }
    if (password == null) {
      return res.jsend.fail({"password": "Password is
required."});
    }
    userService.getOne(email).then((data) => {
        if(data === null) {
            return res.jsend.fail({"result": "Incorrect email
or password"});
        }
        crypto.pbkdf2(password, data.Salt, 310000, 32,
'sha256', function(err, hashedPassword) {
          if (err) { return cb(err); }
          if (!crypto.timingSafeEqual(data.EncryptedPassword,
hashedPassword)) {
              return res.jsend.fail({"result": "Incorrect
email or password"});
          }
          let token;
          try {
            token = jwt.sign(
              { id: data.id, email: data.Email },
              process.env.TOKEN_SECRET,
              { expiresIn: "1h" }
            );
          } catch (err) {
            res.jsend.error("Something went wrong with
creating JWT token")
          }
          res.jsend.success({"result": "You are logged in",
"id": data.id, email: data.Email, token: token});
        });
    });
});
```

Save the following request as a negative example:



POST {{base_path}}/login

none | form-data | x-www-form-urlencoded | [✓ raw] | binary

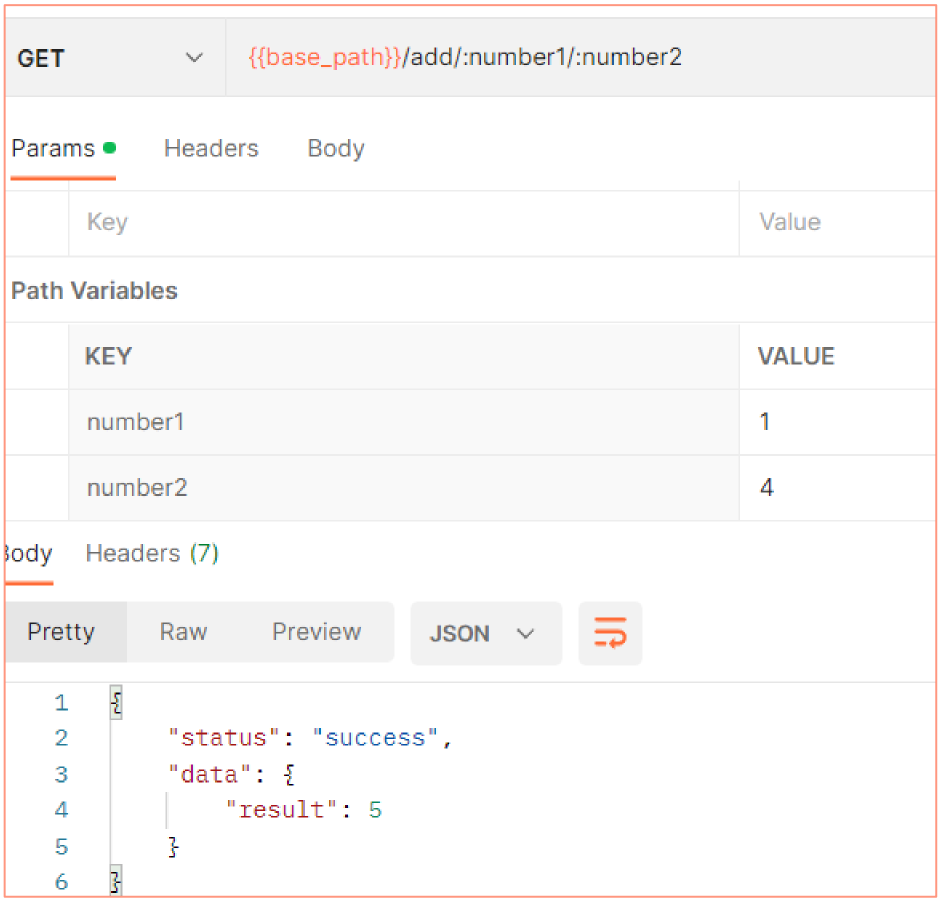
```
{
    "email": "johndoe@yahoo.com"
}
```

JSON

```
{
    "status": "fail",
    "data": {
        "password": "Password is required."
    }
}
```

For Add, create examples similar to the following:

Add:



{{base_path}}/add/:number1/:number2

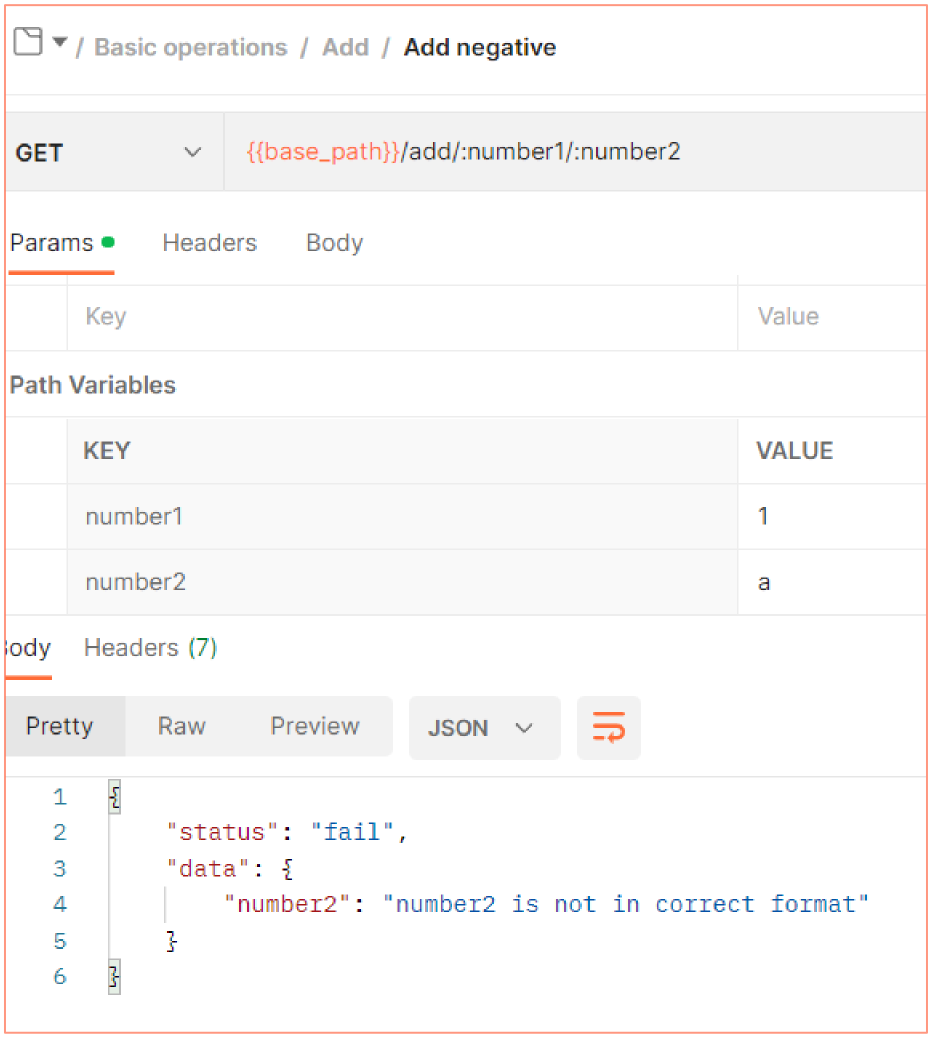
Key	Value

Path Variables

KEY	VALUE
number1	1
number2	4

```
{
    "status": "success",
    "data": {
        "result": 5
    }
}
```

Add negative:



GET {{base_path}}/add/:number1/:number2

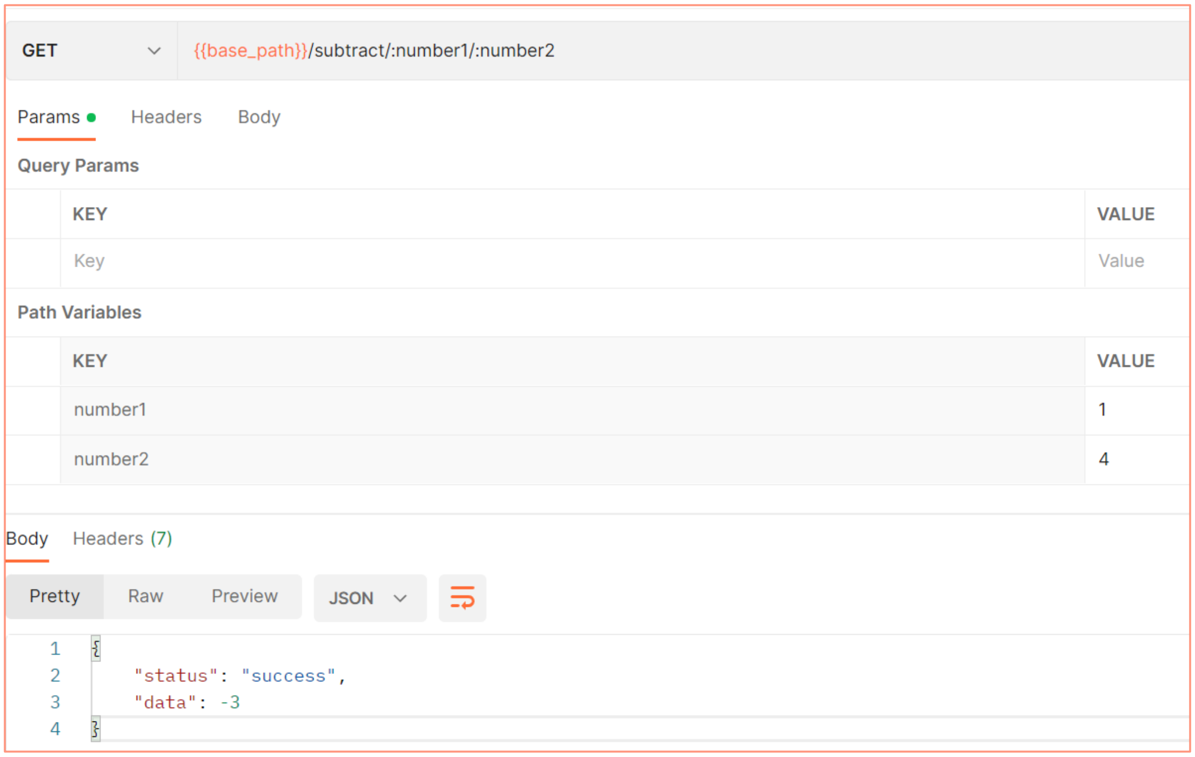
Path Variables

KEY	VALUE
number1	1
number2	a

```
{
    "status": "fail",
    "data": {
        "number2": "number2 is not in correct format"
    }
}
```

Create similar examples for Subtract, Multiply, and Divide:

Subtract positive:


GET {{base_path}}/subtract/:number1/:number2

Params Headers Body

Query Params

KEY	VALUE
Key	Value

Path Variables

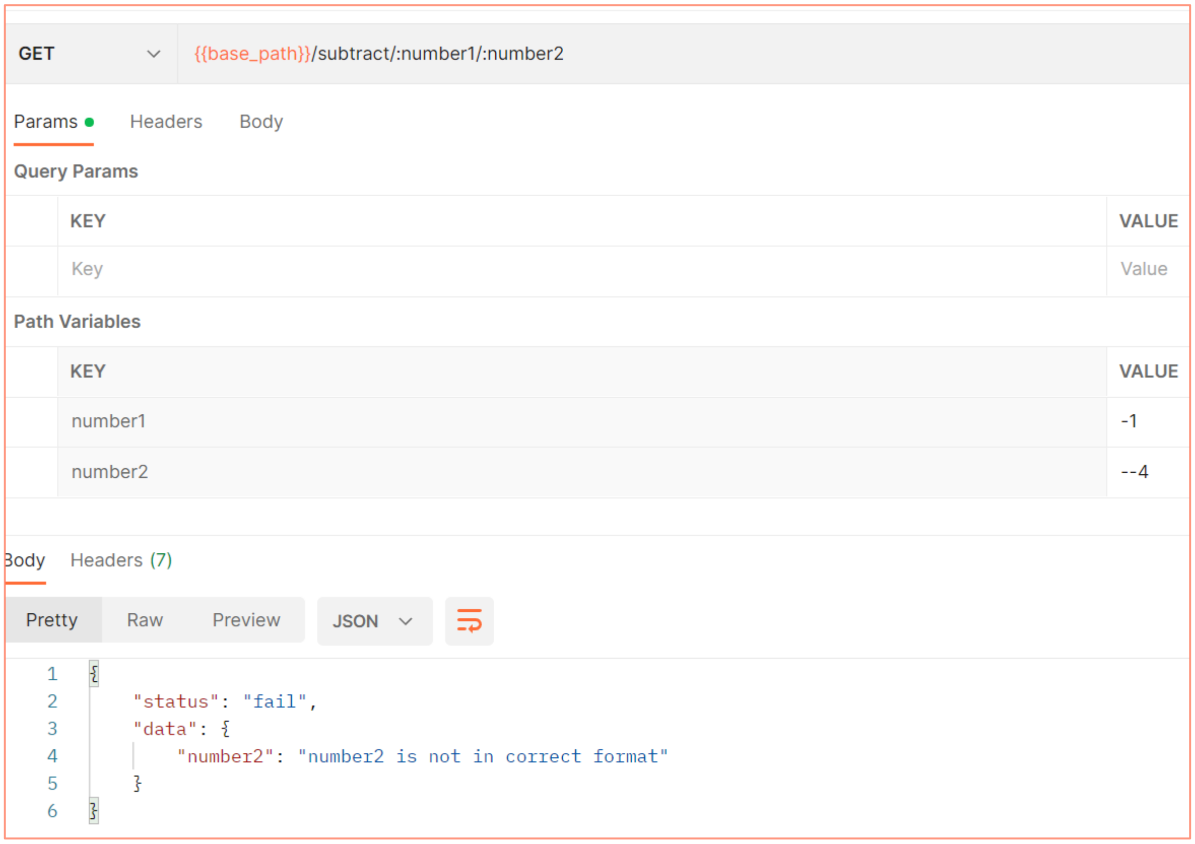
KEY	VALUE
number1	1
number2	4

Body Headers (7)

Pretty Raw Preview JSON

```
{
    "status": "success",
    "data": -3
}
```

Subtract negative:


GET {{base_path}}/subtract/:number1/:number2

Params Headers Body

Query Params

KEY	VALUE
Key	Value

Path Variables

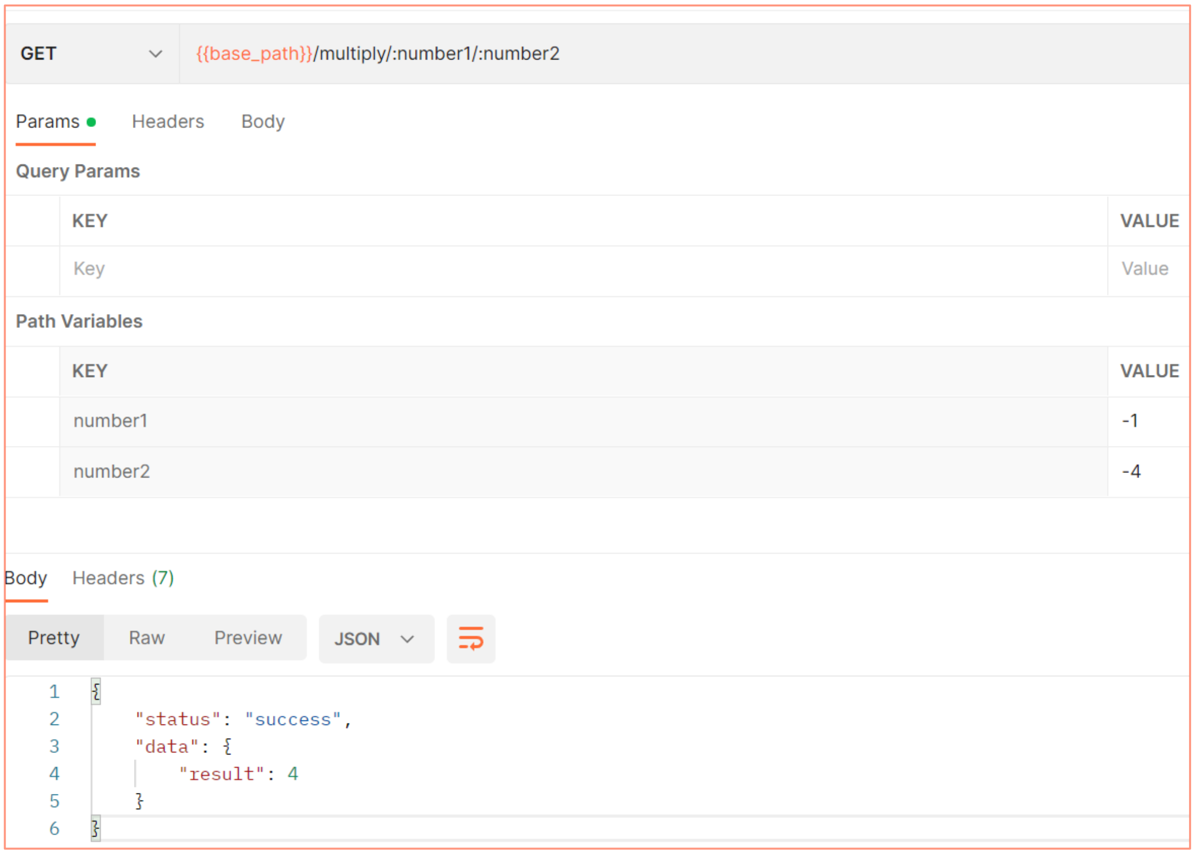
KEY	VALUE
number1	-1
number2	--4

Body Headers (7)

Pretty Raw Preview JSON

```
{
    "status": "fail",
    "data": {
        "number2": "number2 is not in correct format"
    }
}
```

Multiply positive:



GET {{base_path}}/multiply/:number1/:number2

Query Params

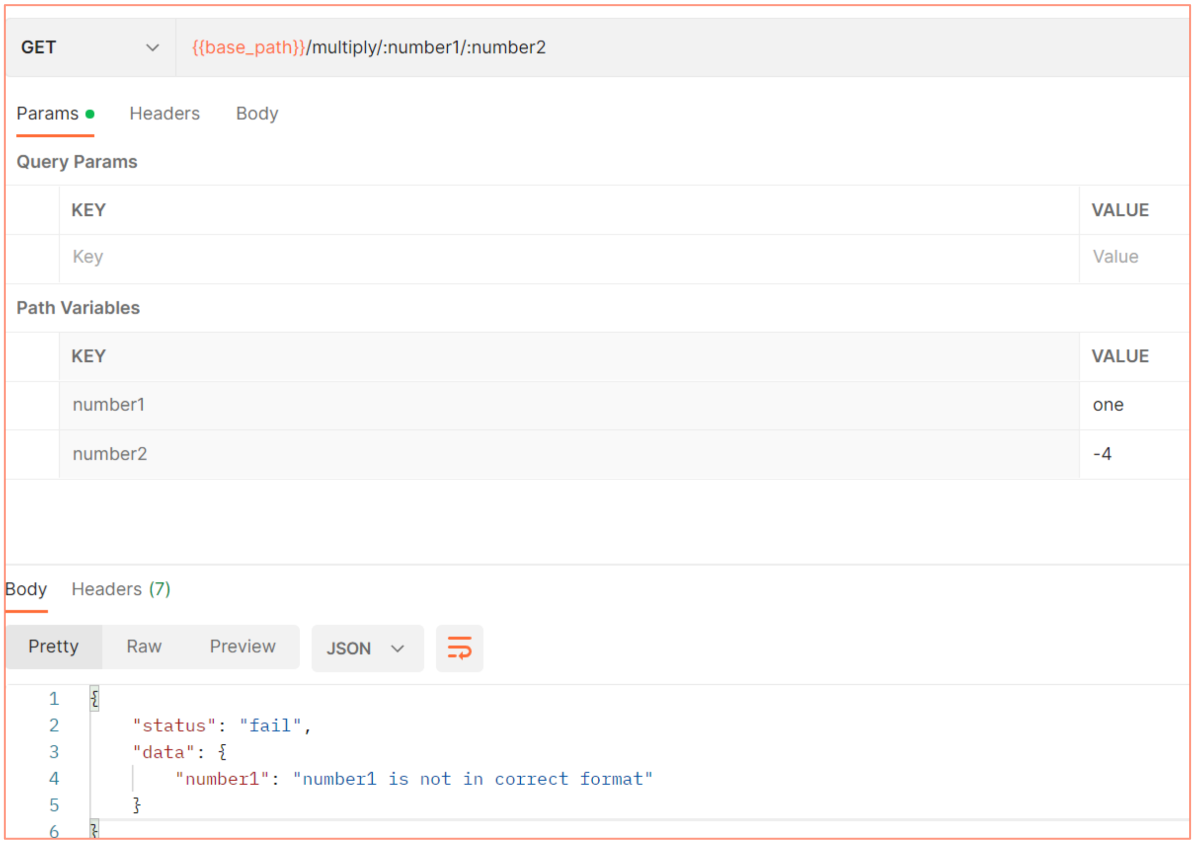
KEY	VALUE
Key	Value

Path Variables

KEY	VALUE
number1	-1
number2	-4

```
{
    "status": "success",
    "data": {
        "result": 4
    }
}
```

Multiply negative:



GET {{base_path}}/multiply/:number1/:number2

Query Params

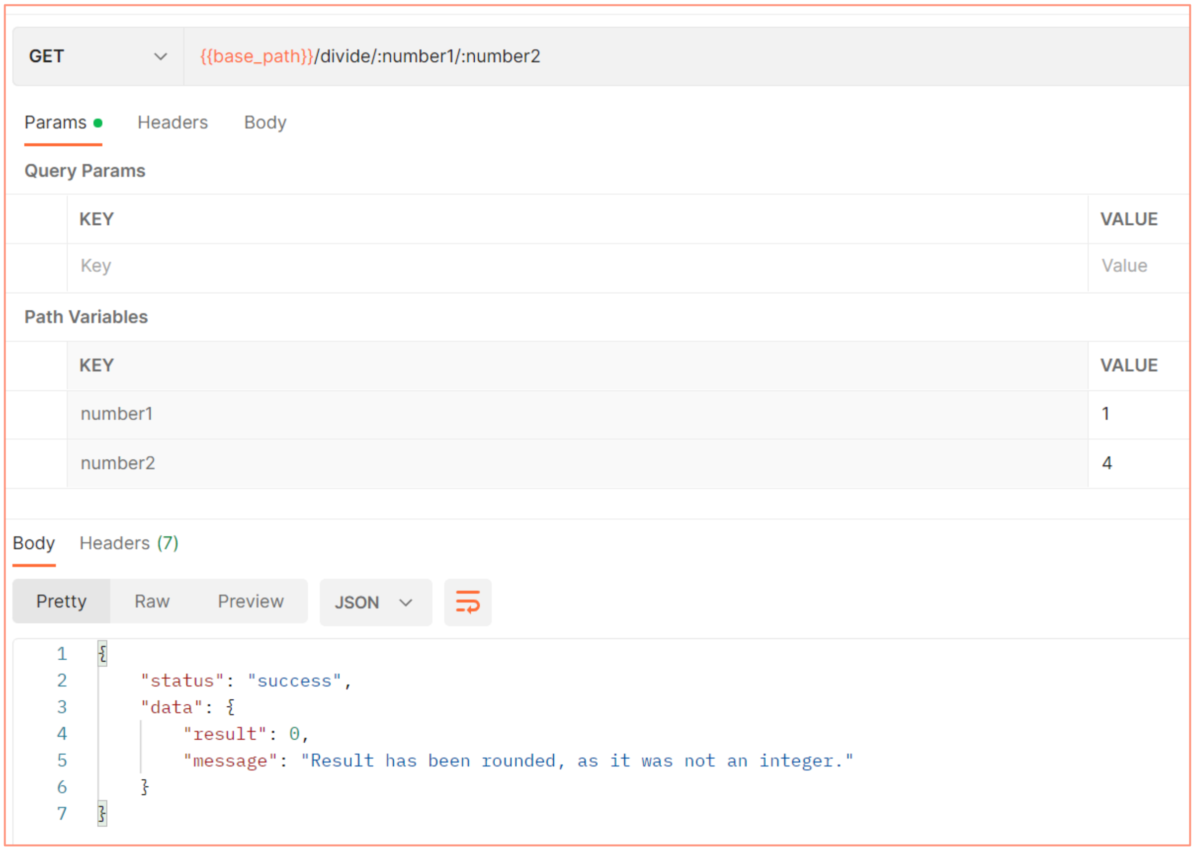
KEY	VALUE
Key	Value

Path Variables

KEY	VALUE
number1	one
number2	-4

```
{
    "status": "fail",
    "data": {
        "number1": "number1 is not in correct format"
    }
}
```

Divide positive:



GET {{base_path}}/divide/:number1/:number2

Query Params

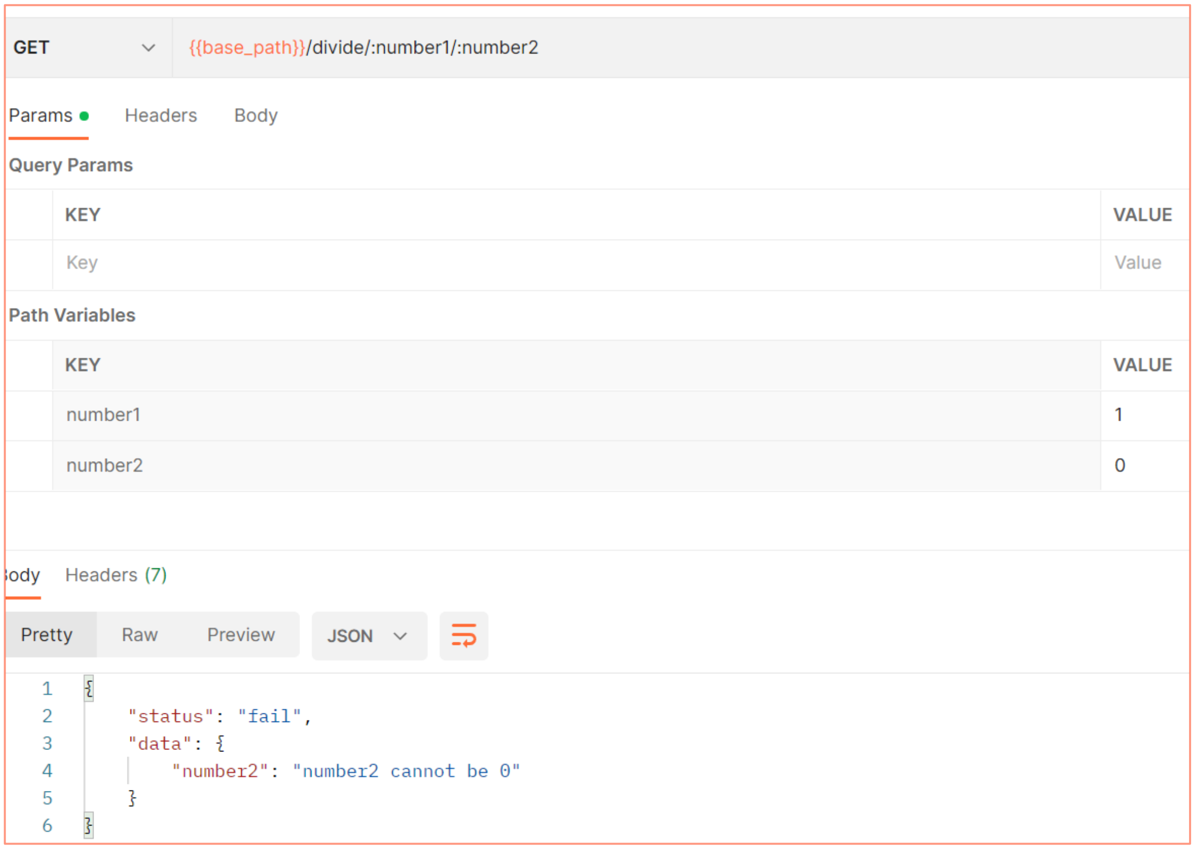
KEY	VALUE
Key	Value

Path Variables

KEY	VALUE
number1	1
number2	4

```
{
    "status": "success",
    "data": {
        "result": 0,
        "message": "Result has been rounded, as it was not an integer."
    }
}
```

Divide negative:



GET {{base_path}}/divide/:number1/:number2

Params Headers Body

Query Params

KEY	VALUE
Key	Value

Path Variables

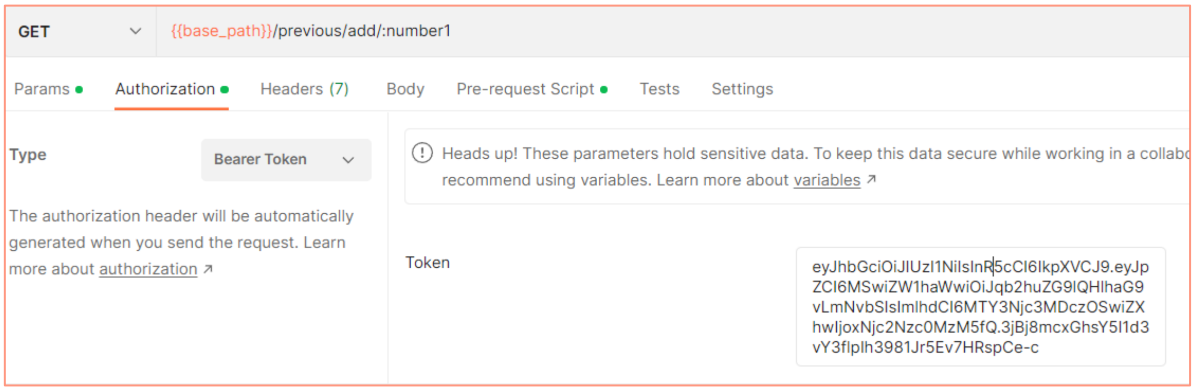
KEY	VALUE
number1	1
number2	0

Body Headers (7)

Pretty Raw Preview JSON

```
{
    "status": "fail",
    "data": {
        "number2": "number2 cannot be 0"
    }
}
```

For Add Previous, set the Authentication header first (send the /login request and copy the header):



GET {{base_path}}/previous/add/:number1

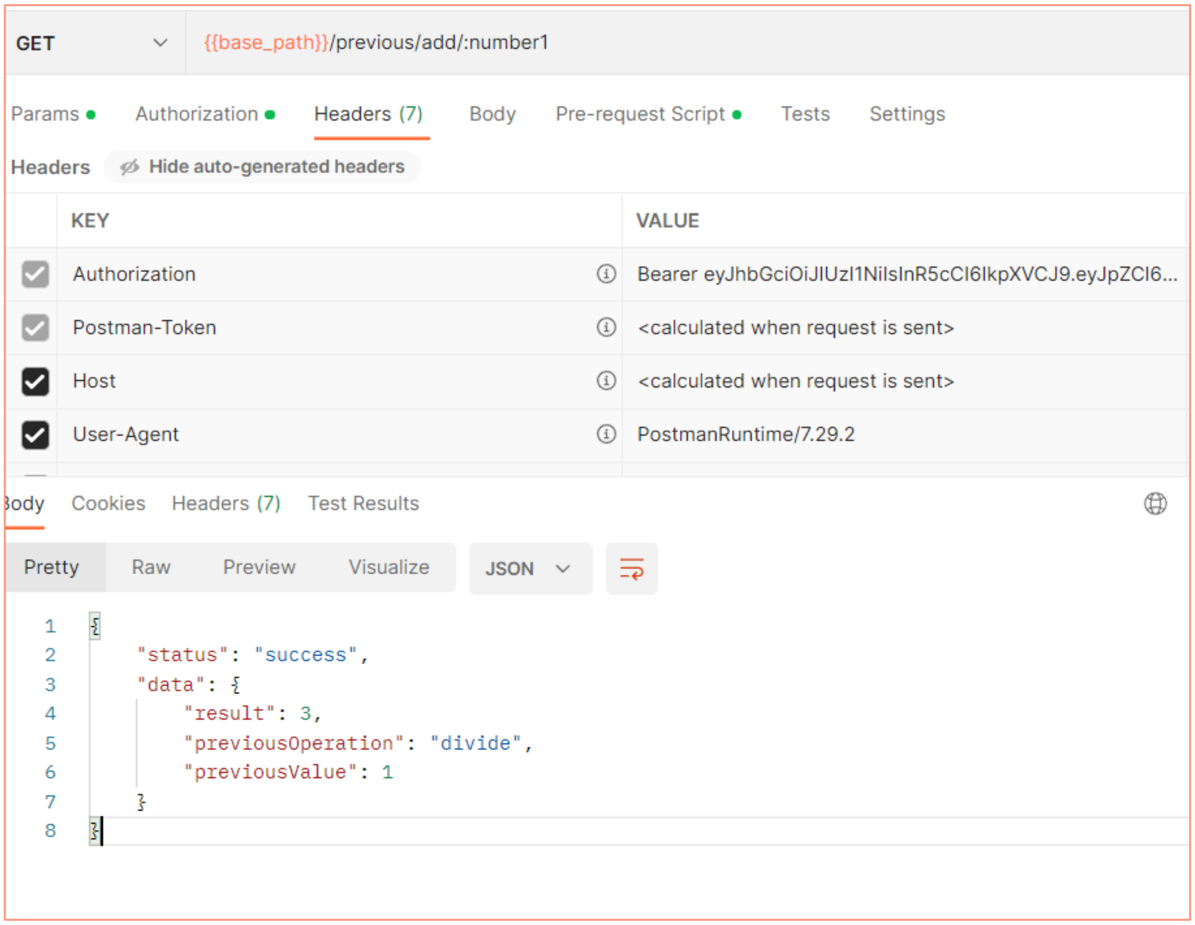
Params Authorization Headers (7) Body Pre-request Script Tests Settings

Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collabo recommend using variables. Learn more about variables ↗

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiZW1haWwiOiJqb2huZG9lQHlhaG9vLmNvbSIsImlhdCI6MTY3Njc3MDczOSwiZXhwIjoxNjc2Nzc0MzM5fQ.3jBj8mcxGhsY5I1d3vY3fIplh3981Jr5Ev7HRspCe-c



GET {{base_path}}/previous/add/:number1

Params ● Authorization ● Headers (7) Body Pre-request Script ● Tests Settings

Headers Hide auto-generated headers

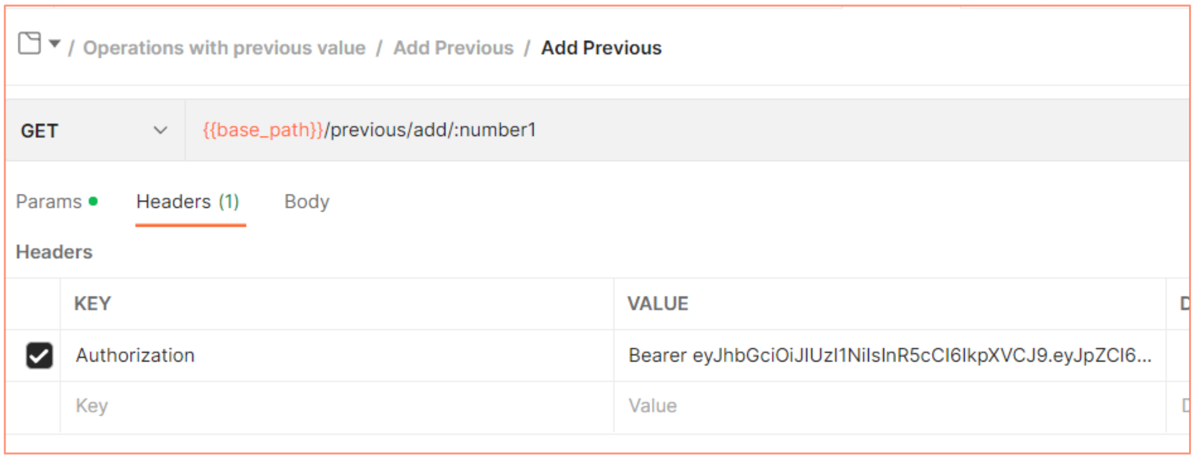
	KEY	VALUE
✓	Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6...
✓	Postman-Token	<calculated when request is sent>
✓	Host	<calculated when request is sent>
✓	User-Agent	PostmanRuntime/7.29.2

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "status": "success",
    "data": {
        "result": 3,
        "previousOperation": "divide",
        "previousValue": 1
    }
}
```

In the Example view, add the header by copying it from the request’s Headers section:



/ Operations with previous value / Add Previous / **Add Previous**

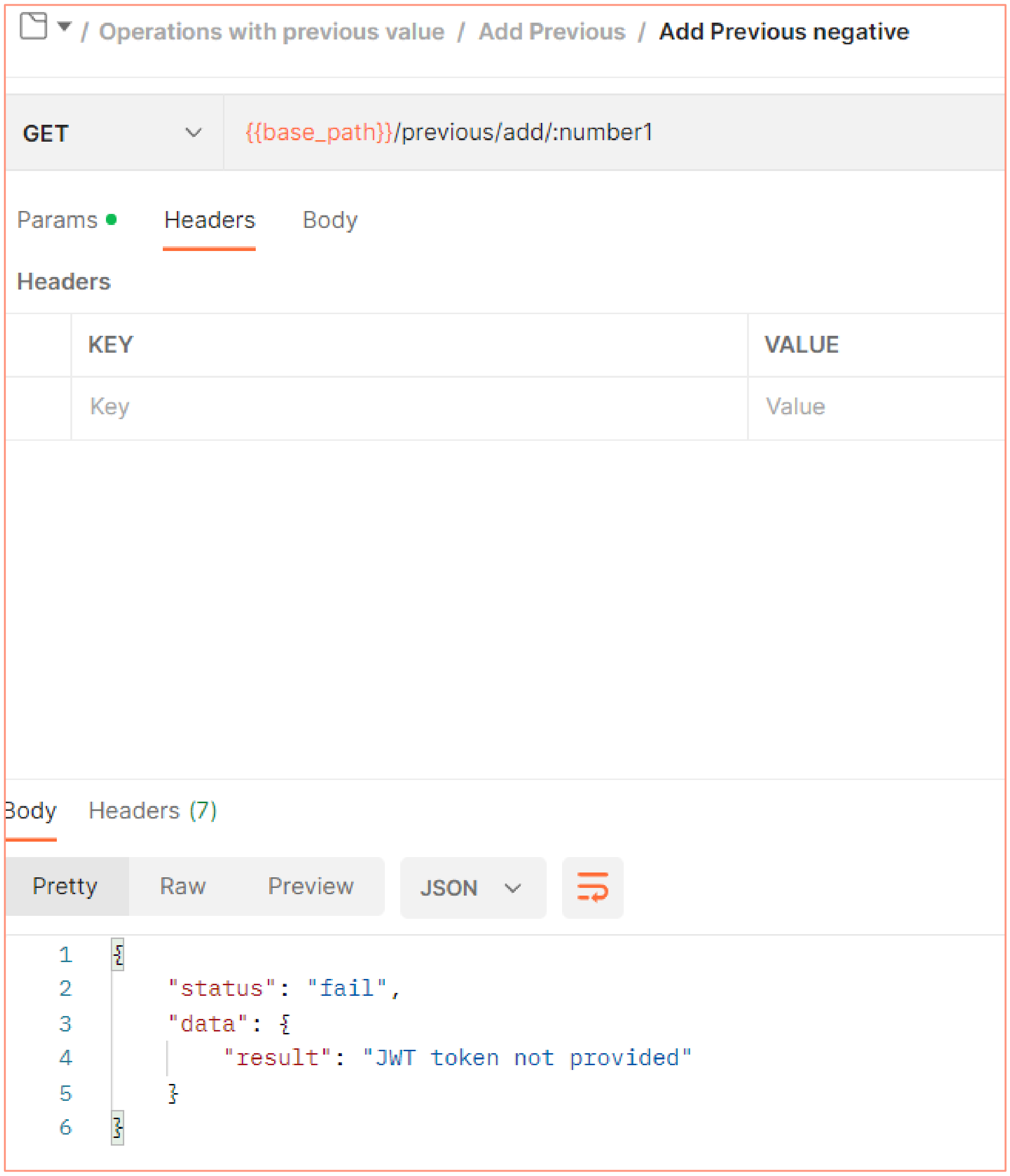
GET {{base_path}}/previous/add/:number1

Params ● Headers (1) Body

Headers

	KEY	VALUE
✓	Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6...
	Key	Value

For requests in this folder, negative requests might just be those without a header:



/ Operations with previous value / Add Previous / Add Previous negative

GET {{base_path}}/previous/add/:number1

Params ● Headers Body

Headers

KEY	VALUE
Key	Value

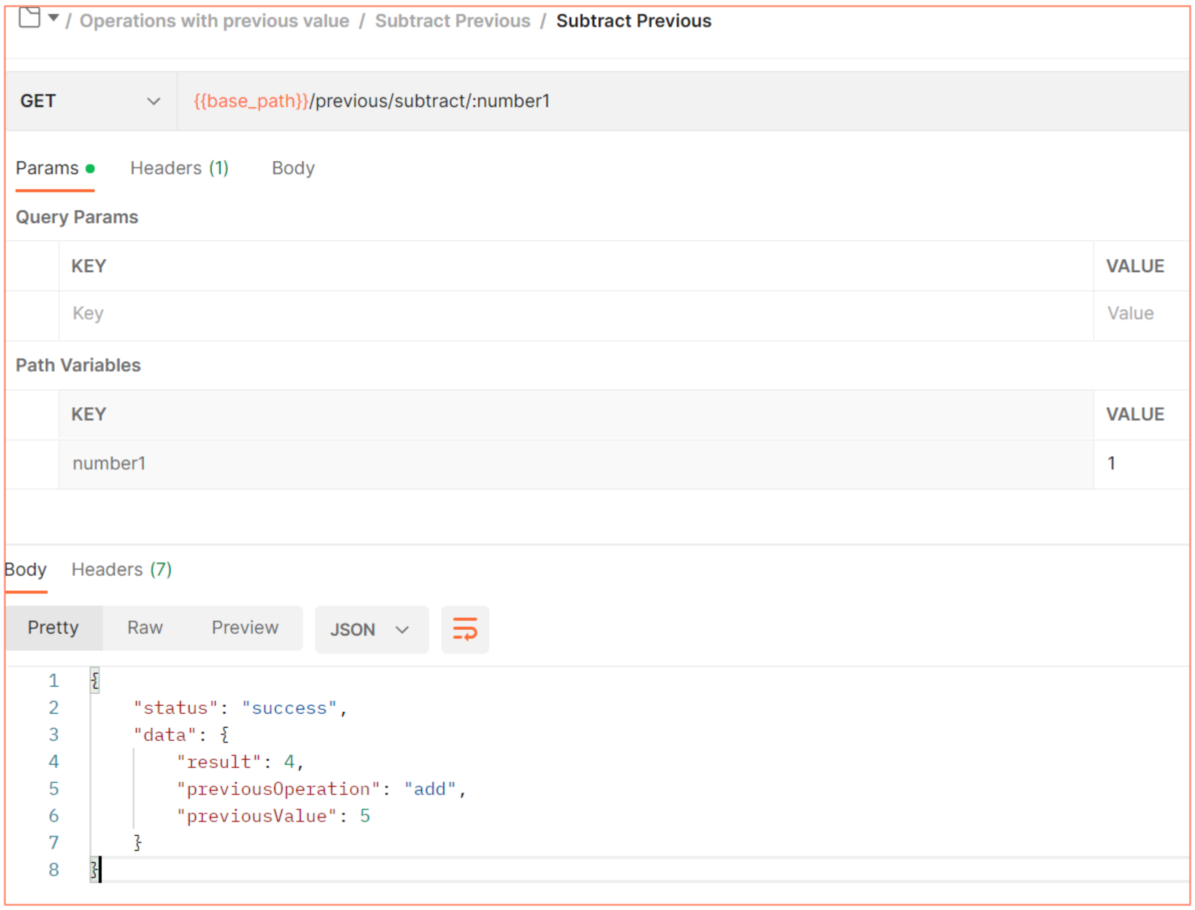
Body Headers (7)

Pretty Raw Preview JSON

```
{
    "status": "fail",
    "data": {
        "result": "JWT token not provided"
    }
}
```

Create similar examples for Subtract Previous, Multiply Previous, and Divide Previous:

Subtract previous positive:


/ Operations with previous value / Subtract Previous / Subtract Previous

GET {{base_path}}/previous/subtract/:number1

Query Params

KEY	VALUE
Key	Value

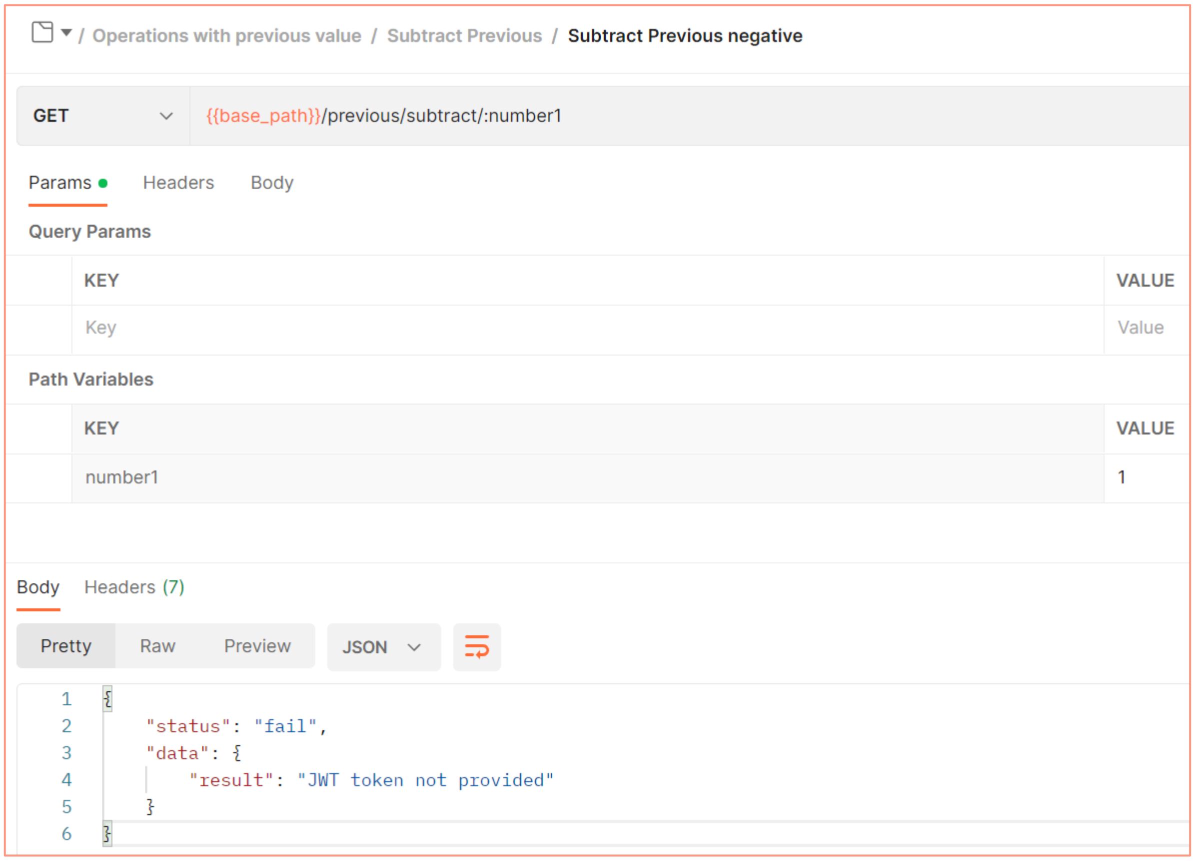
Path Variables

KEY	VALUE
number1	1

Body
Headers (7)
Pretty
Raw
Preview
JSON

```
{
    "status": "success",
    "data": {
        "result": 4,
        "previousOperation": "add",
        "previousValue": 5
    }
}
```

Subtract previous negative:


/ Operations with previous value / Subtract Previous / **Subtract Previous negative**

GET {{base_path}}/previous/subtract/:number1

Query Params

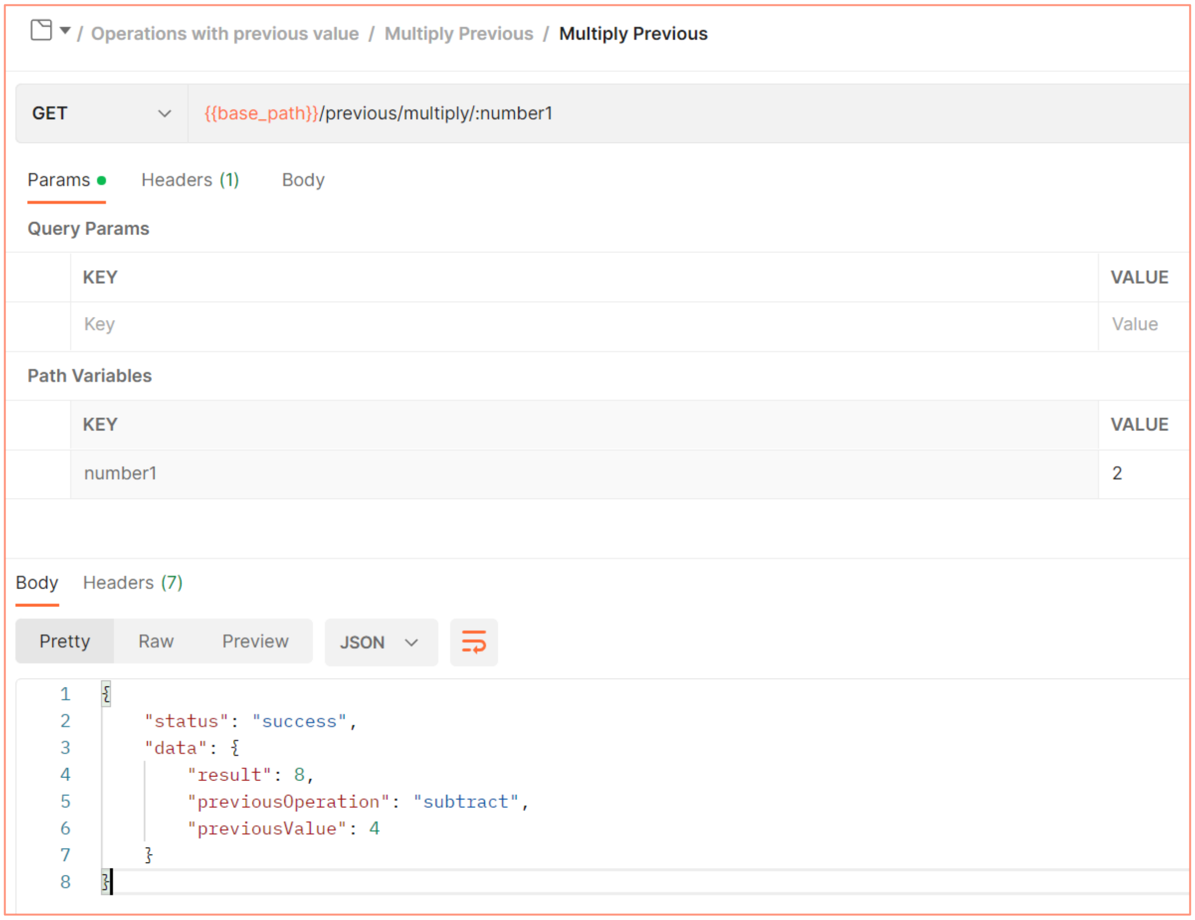
KEY	VALUE
Key	Value

Path Variables

KEY	VALUE
number1	1

```
{
    "status": "fail",
    "data": {
        "result": "JWT token not provided"
    }
}
```

Multiply previous positive:



/ Operations with previous value / Multiply Previous / **Multiply Previous**

GET {{base_path}}/previous/multiply/:number1

Params ● | Headers (1) | Body

Query Params

KEY	VALUE
Key	Value

Path Variables

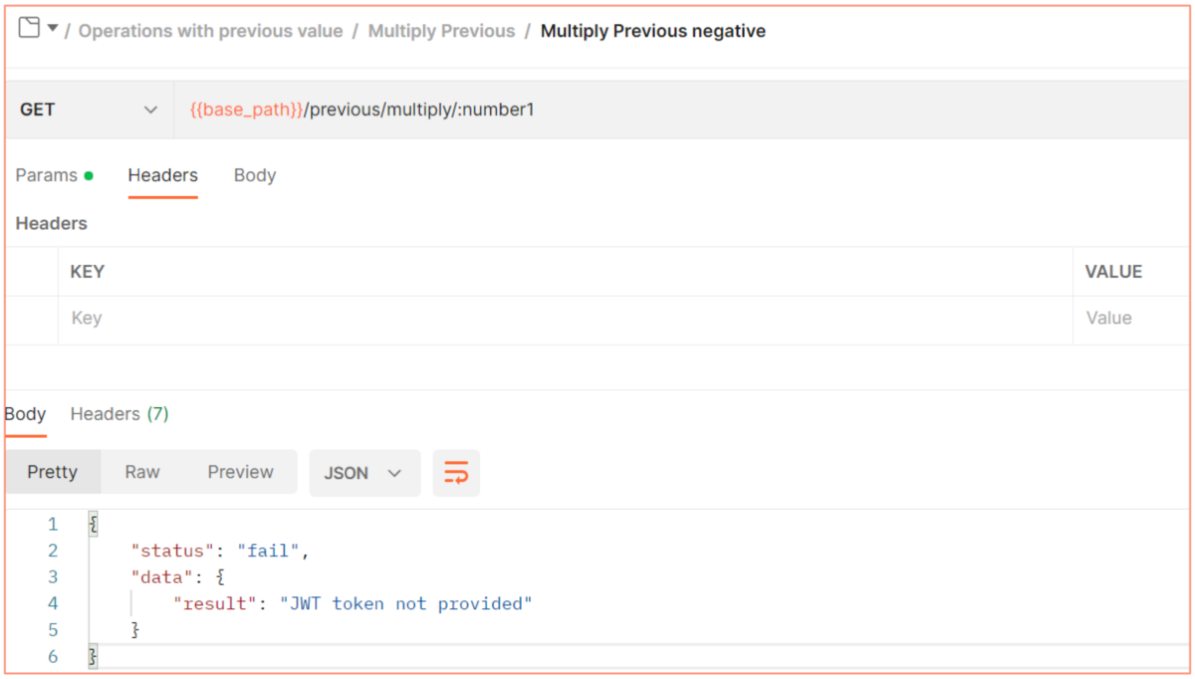
KEY	VALUE
number1	2

Body | Headers (7)

Pretty | Raw | Preview | JSON

```
{
    "status": "success",
    "data": {
        "result": 8,
        "previousOperation": "subtract",
        "previousValue": 4
    }
}
```

Multiply previous negative:



/ Operations with previous value / Multiply Previous / **Multiply Previous negative**

GET {{base_path}}/previous/multiply/:number1

Params ● | Headers | Body

Headers

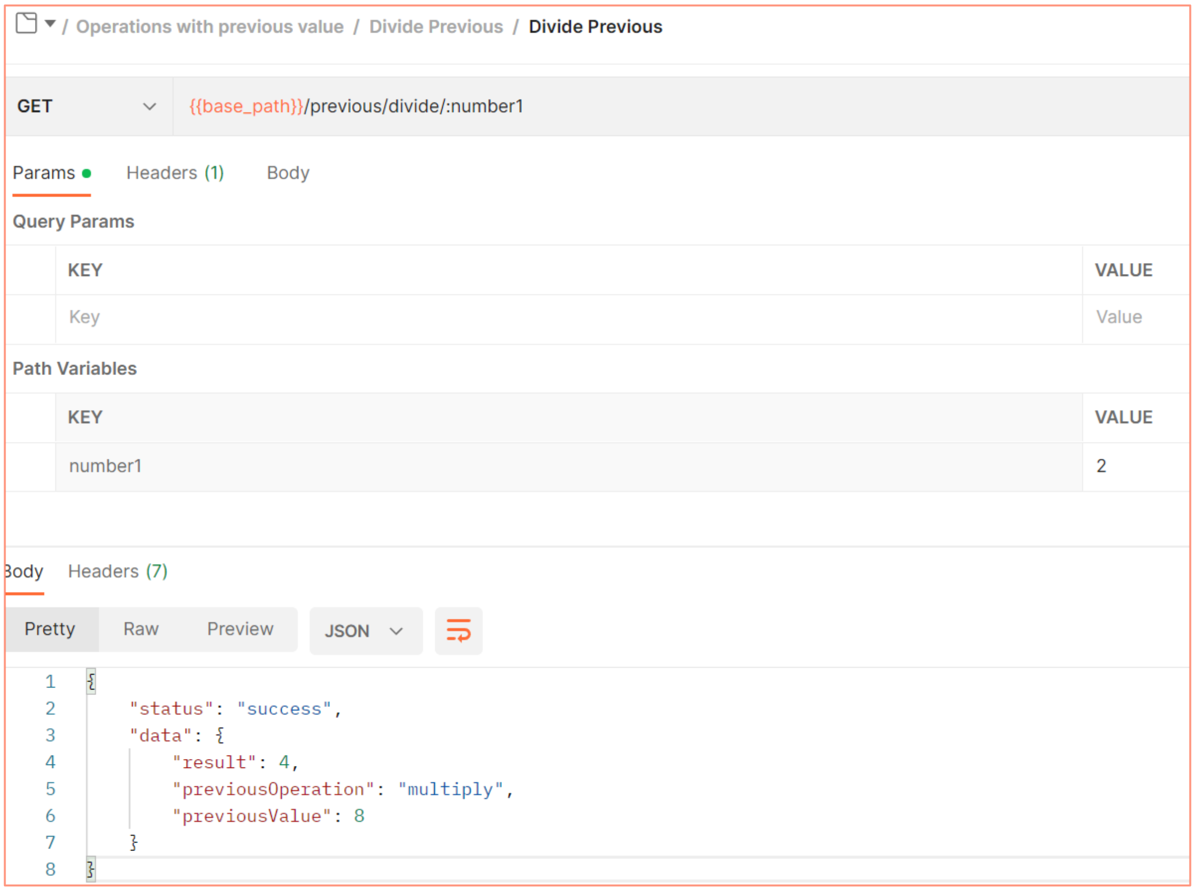
KEY	VALUE
Key	Value

Body | Headers (7)

Pretty | Raw | Preview | JSON

```
{
    "status": "fail",
    "data": {
        "result": "JWT token not provided"
    }
}
```

Divide previous positive:


/ Operations with previous value / Divide Previous / **Divide Previous**

GET {{base_path}}/previous/divide/:number1

Params ● Headers (1) Body

Query Params

KEY	VALUE
Key	Value

Path Variables

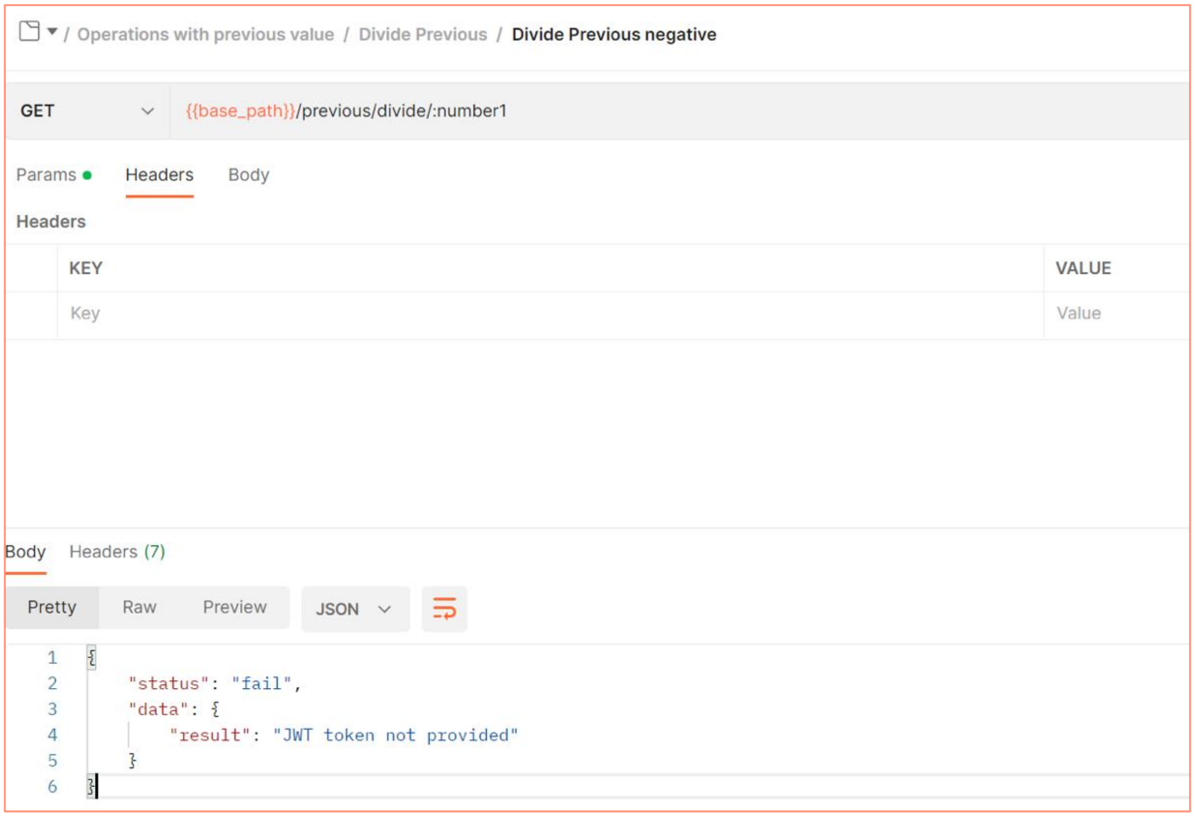
KEY	VALUE
number1	2

Body Headers (7)

Pretty Raw Preview JSON

```
{
    "status": "success",
    "data": {
        "result": 4,
        "previousOperation": "multiply",
        "previousValue": 8
    }
}
```

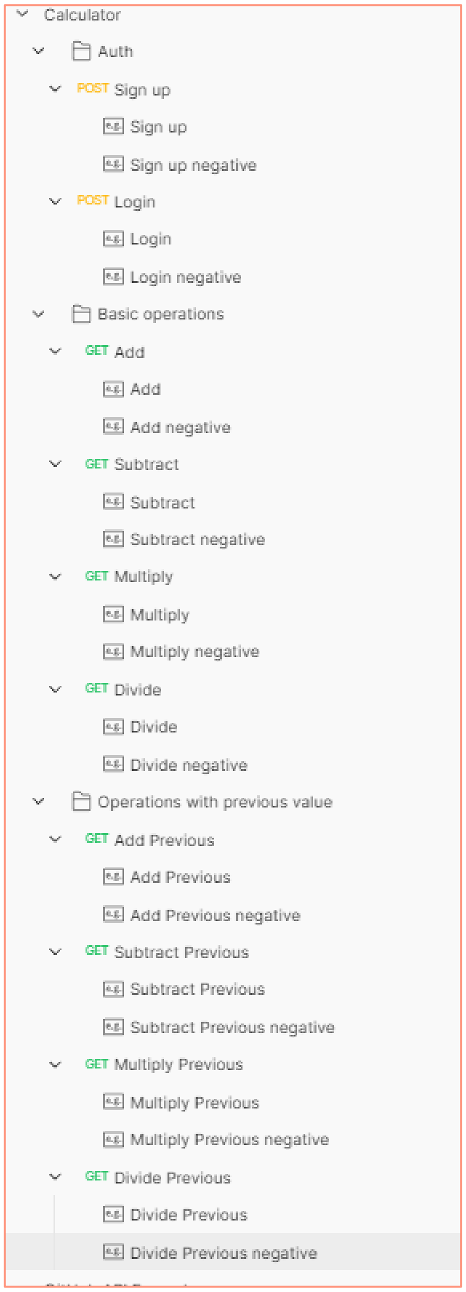
Divide previous negative:



```
{{base_path}}/previous/divide/:number1
```

```
{
    "status": "fail",
    "data": {
        "result": "JWT token not provided"
    }
}
```

After finishing all the examples, your Calculator collection should look as follows:

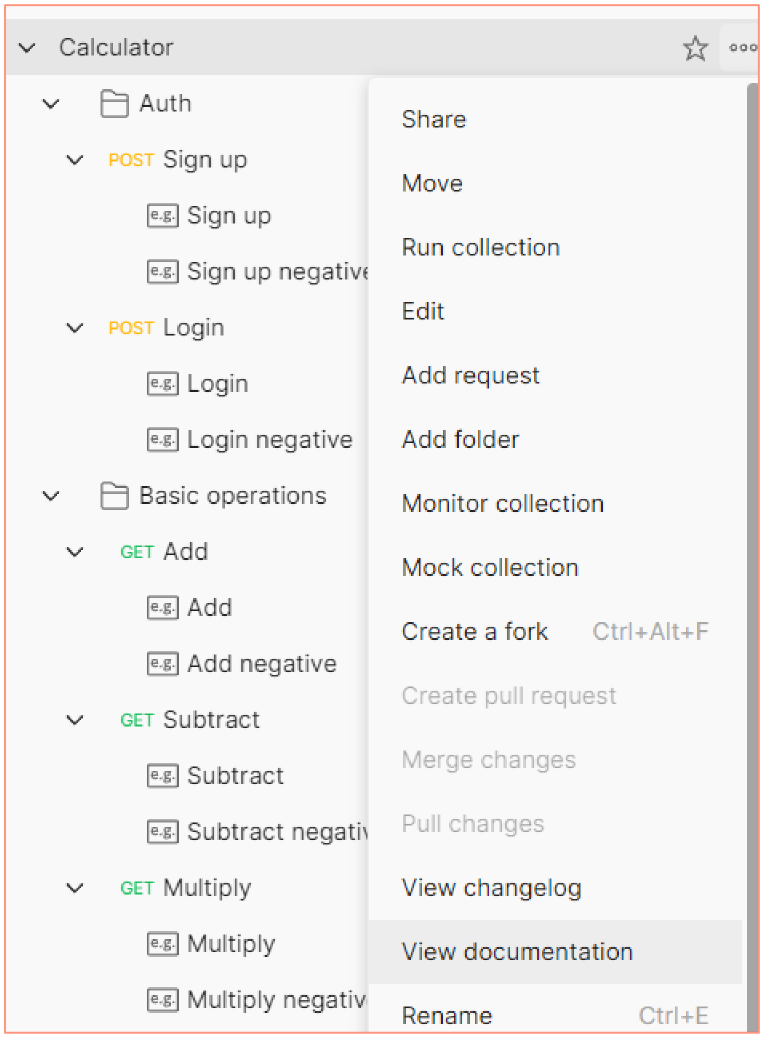


- Calculator
  - Auth
    - POST Sign up
      - Sign up
      - Sign up negative
    - POST Login
      - Login
      - Login negative
  - Basic operations
    - GET Add
      - Add
      - Add negative
    - GET Subtract
      - Subtract
      - Subtract negative
    - GET Multiply
      - Multiply
      - Multiply negative
    - GET Divide
      - Divide
      - Divide negative
  - Operations with previous value
    - GET Add Previous
      - Add Previous
      - Add Previous negative
    - GET Subtract Previous
      - Subtract Previous
      - Subtract Previous negative
    - GET Multiply Previous
      - Multiply Previous
      - Multiply Previous negative
    - GET Divide Previous
      - Divide Previous
      - Divide Previous negative

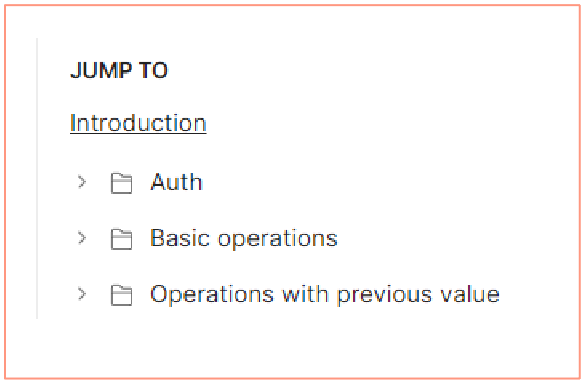
Adding descriptions

Currently, our documentation has a large amount of content and examples. However, requests are described only by examples, which is not necessarily sufficient. A standard description helps us to specify the endpoint’s goal.

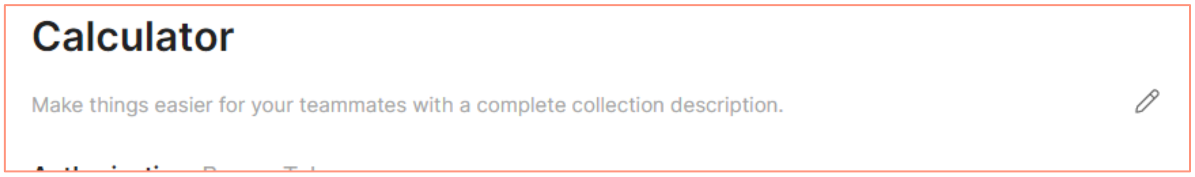
Let’s open the current documentation view by right-clicking on the Calculator collection and choosing “View documentation”:



On the right, we will see our documentation's table of contents. Let's start from the beginning by clicking on "Introduction":



In each area where we should, Postman reminds us to add a description with the following communication:


Calculator

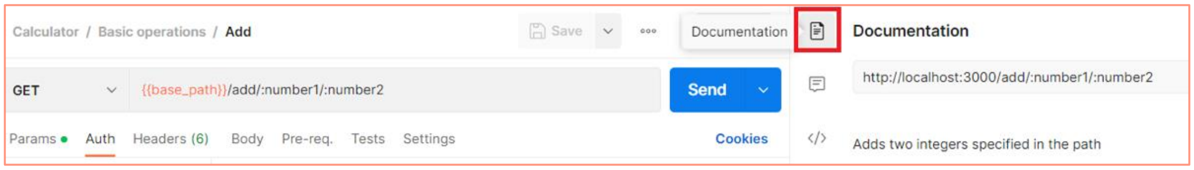
Make things easier for your teammates with a complete collection description.

We can click the pencil icon to edit the documentation and add the description.

In this section, you need to go through the documentation and fill in each area similar to the one presented above. In the next section, you will have access to the final published documentation, so you will be able to compare the results.

The other way to add/change an endpoint's description is to do so from the requests and folders level.

To do this, open any endpoint (for example, Add) and click the documentation icon in the menu on the right (marked with a red square in the picture below):



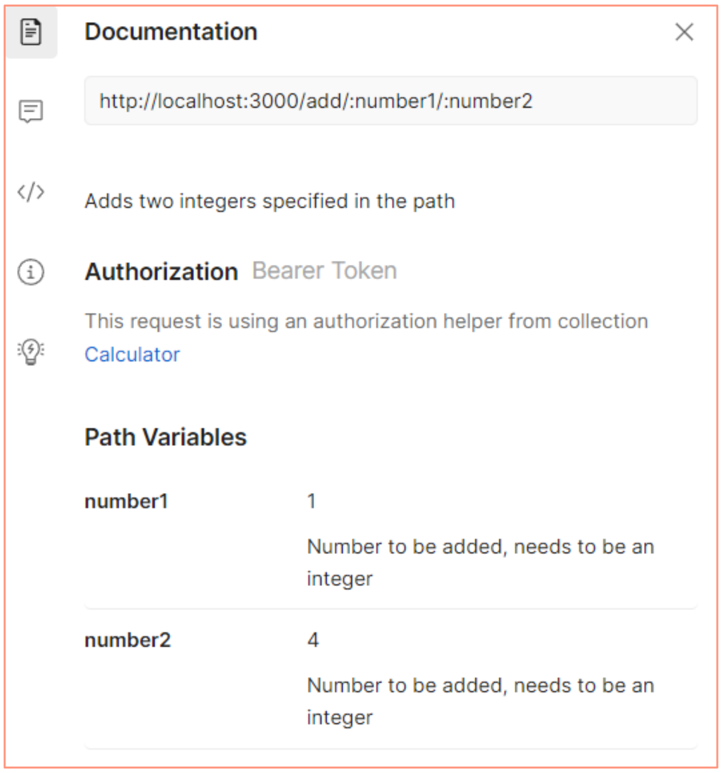
{{base_path}}/add/:number1/:number2

Documentation

http://localhost:3000/add/:number1/:number2

Adds two integers specified in the path

After opening the section, we can edit the documentation chapter connected to the opened endpoint:


Documentation

http://localhost:3000/add/:number1/:number2

Adds two integers specified in the path

Authorization Bearer Token

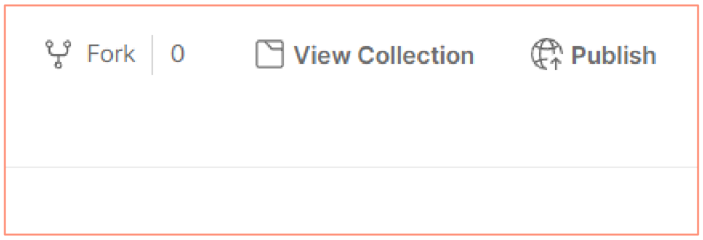
This request is using an authorization helper from collection Calculator

Path Variables

number1	1 Number to be added, needs to be an integer
number2	4 Number to be added, needs to be an integer

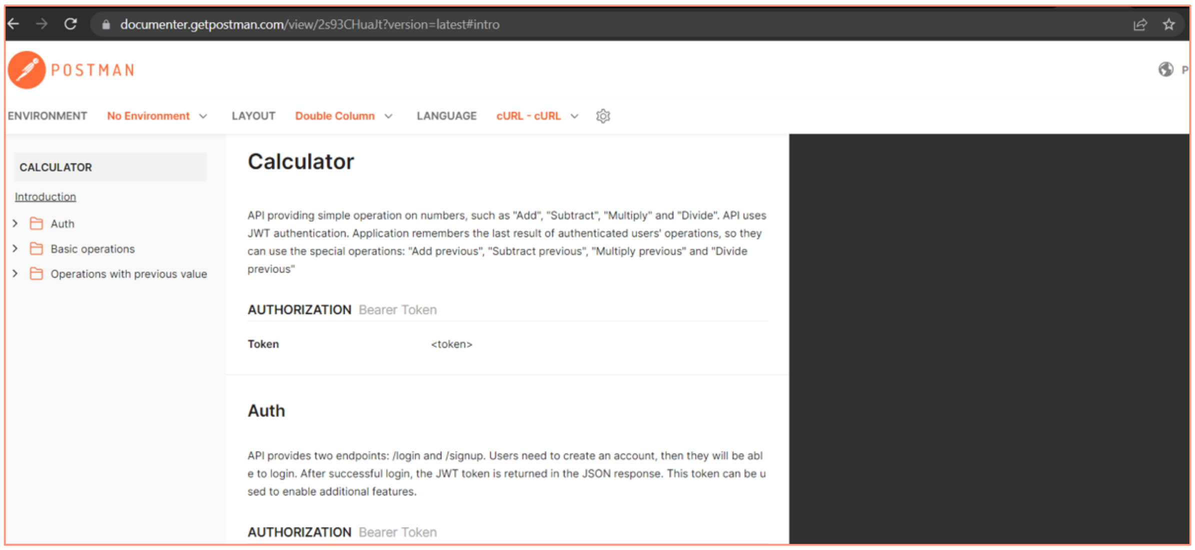
Publishing documentation

After adding proper descriptions, our documentation should be ready to publish. Open the documentation view the same way as in the last section. In the top-right corner, you should see a publish button – click it:



The new window should open in your web browser. Use the default settings and publish the application.

This lesson's documentation is available to view on Postman (linked in the READ section).


documenter.getpostman.com/view/2s93CHuaJt?version=latest#intro

POSTMAN

ENVIRONMENT No Environment LAYOUT Double Column LANGUAGE cURL - cURL

CALCULATOR
Introduction
Auth
Basic operations
Operations with previous value

Calculator

API providing simple operation on numbers, such as "Add", "Subtract", "Multiply" and "Divide". API uses JWT authentication. Application remembers the last result of authenticated users' operations, so they can use the special operations: "Add previous", "Subtract previous", "Multiply previous" and "Divide previous"

AUTHORIZATION Bearer Token

Token <token>

Auth

API provides two endpoints: /login and /signup. Users need to create an account, then they will be abl e to login. After successful login, the JWT token is returned in the JSON response. This token can be u sed to enable additional features.

AUTHORIZATION Bearer Token

Read the documentation and compare it in detail with your version.

READ

Documentation: Calculator on Postman.

What did I learn in this lesson?

This lesson provided the following insights:

- How to create a new Postman collection for our API.
- How to add detailed request examples for our API.
- How to create detailed descriptions for our API and publish the documentation.

References

	INFO Please note that the reference list below was compiled when writing the course content to credit the sources used. Therefore, some links that worked at the time of writing may not work now or may have changed. However, accessing these references is not a requirement for this module. Note that all the required reading is specified in the READ blocks in the content.

Postman Learning Center. (n.d.). *Introduction.* [online] Available at: https://learning.postman.com/docs/getting-started/introduction/.

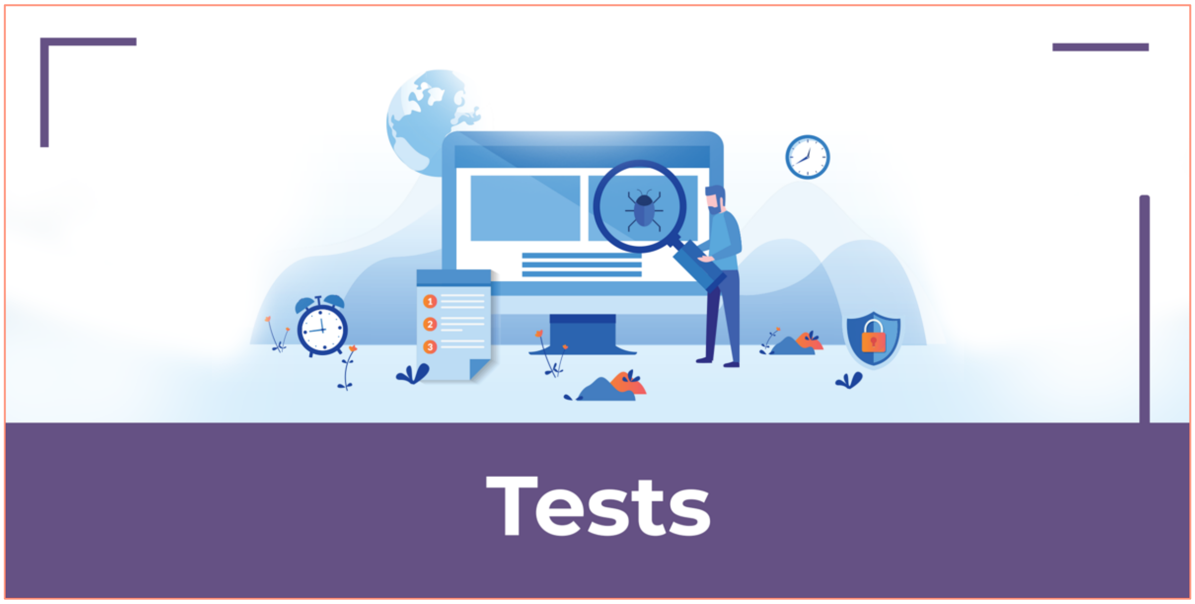
5.3. Lesson task - Project documentation

The task

In this lesson, we created detailed Postman documentation for our API. In this task, you need to generate simple Swagger documentation for our API (you don't need to describe fields with comments). The documentation should be available at http://localhost:3000/doc.

5.4. Lesson - Tests

Introduction



We have already implemented the Calculator API and created Postman documentation for it. We found some minor bugs – the application crashed when data from the request's body was different than expected. We fixed most of these bugs by adding validation. In this lesson, we will implement simple tests for our API and look for other potential bugs.

First, we will consider which tests fit our application. Next, we will implement them using Node libraries (such as Jest or Supertest).

In the end, we will summarise the entire project and think about which things could have been done differently.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- → The candidate has knowledge of industry-standard JavaScript frameworks that can be used to build RESTful services.

In this lesson, we are covering the following skill learning outcomes:

- → The candidate masters creating RESTful solutions in JavaScript that use common HTTP methods.
- → The candidate masters the Postman software (or open-source alternatives) to test and document a REST API.

In this lesson, we are covering the following general competence learning outcome:

- → The candidate can carry out work as a REST API developer, maintainer, or tester.

Testing overview

Our API implements very simple functionalities. To implement proper unit tests, we would need to export the methods executing operations on the two numbers to external files, test these files, and use the exported methods in the router files instead.

Currently, in app.js, we use the following code:

```
const number1 = parseInt(req.params.number1);
if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const number2 = parseInt(req.params.number2);
    if(isNaN(number2)) {
        return res.jsend.fail({"number2": "number2 is not in
correct format"});
    }
    const result = number1 + number2;
```

In one function, we parse parameters to integers and add them. Theoretically, the router handler should contain only those functions that directly handle requests to the endpoints. To fix this, let's create the external file numberHelper.js with methods such as:

```
function add(number1, number2) {
    return number1 + number2;
}

function parseParameters(number1, number2) {
    const intNumber1 =  parseInt(number1);
    const intNumber2 =  parseInt(number2);
    if(isNaN(intNumber1)) {
        return {number1: "number1 is not in correct format"};
    }
    if(isNaN(intNumber2)) {
        return {number2: "number2 is not in correct format"};
    }
    return {number1: intNumber1, number2: intNumber2}
}
```

Then, we should be able to unit-test these methods. You can try to refactor some of the endpoints this way and create simple unit tests; however, we will not implement unit testing in this lesson. These functionalities are very simple, and for the more complex ones, we use external packages, so there is not much room for possible mistakes.

Rather than testing single methods, we will test entire endpoints. Tests should not modify the database, so in theory, we should create a test database just for the tests. However, as this is not a real-world solution, we will use the already created database and the test user to test all the functionalities based on the previous result.

In these endpoints' (integration) testing, we will use Jest and Supertest.

Besides that, we will implement more tests in Postman and try to think of edge cases we could have forgotten. An edge case is a possible but unusual scenario where our application might behave in an unexpected way.

Integration tests

We will use Jest and Supertest. Install them with the command:

```
npm install jest supertest
```

Change the package.json file to use Jest in testing – use the following script:

```
"scripts": {
    "start": "node ./bin/www",
    "test": "jest"
  },
```

Create the "tests" folder at the same level as the app.js file. Create the basicRoutes.test.js file in that folder. Here we will test all basic operations available to non-authenticated users.

In basicRoutes.test.js, include the necessary packages as well as the files to be tested:

```
const express = require("express");
const request = require("supertest");
const app = express();
require('dotenv').config()

const bodyParser = require("body-parser");

const addRoutes = require("../routes/add");
const subtractRoutes = require("../routes/subtract");
const multiplyRoutes = require("../routes/multiply");
const divideRoutes = require("../routes/divide");

app.use(bodyParser.json());
app.use("/add", addRoutes);
app.use("/subtract", subtractRoutes);
app.use("/multiply", multiplyRoutes);
app.use("/divide", divideRoutes);
```

Next, let’s create one positive and one negative test for the /add endpoint:

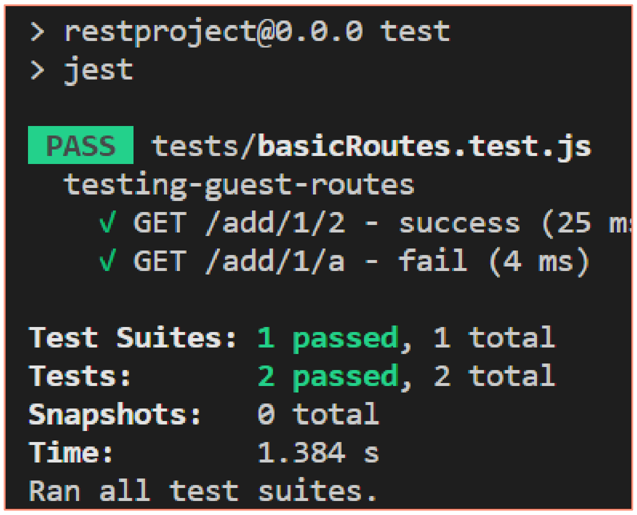
```
describe("testing-guest-routes", () => {
  test("GET /add/1/2 - success", async () => {
    const { body } = await request(app).get("/add/1/2");
    expect(body).toEqual({
        "status": "success",
        "data": {
            "result": 3
        }
      });
  });

  test("GET /add/1/a - fail", async () => {
    const { body } = await request(app).get("/add/1/a");
    expect(body).toEqual({
        "status": "fail",
        "data": {
            "number2": "number2 is not in correct format"
        }
      });
  });
});
```

We use the Supertest package to make requests to our API easily. We check that the response’s body is exactly what was expected.

Run the test with the command:

```
npm test
```

Both the positive and the negative test should pass:



```
> restproject@0.0.0 test
> jest

 PASS  tests/basicRoutes.test.js
  testing-guest-routes
    √ GET /add/1/2 - success (25 m
    √ GET /add/1/a - fail (4 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.384 s
Ran all test suites.
```

Now, implement similar tests for /subtract, /multiply, and /divide:

```
test("GET /subtract/1/2 - success", async () => {
    const { body } = await
request(app).get("/subtract/1/2");
    expect(body).toEqual({
        "status": "success",
        "data": {
            "result": -1
        }
      });
  });

  test("GET /subtract/1/a - fail", async () => {
    const { body } = await
request(app).get("/subtract/1/a");
    expect(body).toEqual({
        "status": "fail",
        "data": {
            "number2": "number2 is not in correct format"
        }
      });
  });

  test("GET /multiply/1/2 - success", async () => {
    const { body } = await
request(app).get("/multiply/1/2");
    expect(body).toEqual({
        "status": "success",
        "data": {
            "result": 2
        }
      });
  });

  test("GET /multiply/1/a - fail", async () => {
    const { body } = await
request(app).get("/multiply/1/a");
    expect(body).toEqual({
        "status": "fail",
        "data": {
            "number2": "number2 is not in correct format"
        }
      });
  });

  test("GET /divide/2/1 - success", async () => {
    const { body } = await request(app).get("/divide/2/1");
    expect(body).toEqual({
        "status": "success",
        "data": {
            "result": 2
        }
      });
  });

  test("GET /divide/1/0 - fail", async () => {
    const { body } = await request(app).get("/divide/1/0");
    expect(body).toEqual({
        "status": "fail",
        "data": {
            "number2": "number2 cannot be 0"
        }
    });
  });
```

Let’s move on to testing operations based on the previous result. Create a new test file in the same folder and name it integration.test.js. We will need all our functionalities to test operations on previous values properly.

First, we will log in to the app and assign a generated token to the variable.

Next, we will make the request to the /add endpoint (we already know it works, as it has been tested) to make sure that the previous value is set. We need to remember to set the correct authorization header.

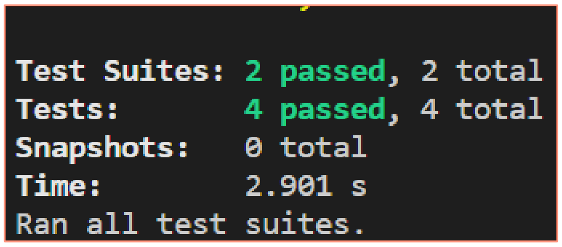
Finally, we will call the /previous/add/1 endpoint to increment the previous result by 1.

The code responsible for this is presented below:

integration.test.js:

```
const express = require("express");
const request = require("supertest");
const app = express();
require('dotenv').config()

const bodyParser = require("body-parser");

const previousRoutes = require("../routes/previous");
const authRoutes = require("../routes/auth");
const addRoutes = require("../routes/add");

app.use(bodyParser.json());
app.use("/previous", previousRoutes);
app.use("/add", addRoutes);
app.use("/", authRoutes);

describe("testing-guest-routes", () => {
  //test user is already created and has credentials email:
"johndoe@yahoo.com", password:"0000"
  let token;
  test("POST /login - success", async () => {
    const credentials = {
      email: "johndoe@yahoo.com",
      password: "0000"
    }
    const { body } = await
request(app).post("/login").send(credentials);
    expect(body).toHaveProperty("data");
    expect(body.data).toHaveProperty("token");
    token = body.data.token
  });

  test("GET /previous/add - success", async () => {
    await request(app).get("/add/1/2").set('Authorization',
'Bearer ' + token);
    const { body } = await
request(app).get("/previous/add/1").set('Authorization',
'Bearer ' + token);
    expect(body).toHaveProperty("data");
    expect(body.data).toHaveProperty("previousValue");
    expect(body.data).toHaveProperty("result");
    expect(parseInt(body.data.result)).toBe(parseInt(body.dat
a.previousValue) + 1);
  });

});
```

After running the test (with the “npm test” command), all the tests are passed:



```
Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        2.901 s
Ran all test suites.
```

ACTIVITY

As an exercise, implement tests for the /previous/subtract, /previous/multiply, and /previous/divide endpoints (they will be similar to the one presented above).

Solution: Click here to reveal the answer.

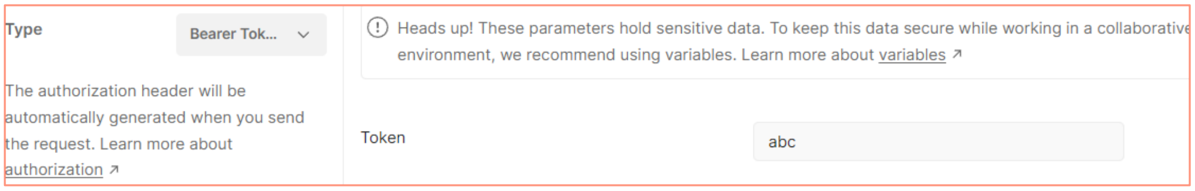
Other tests

We have created many tests in our application (not only in this lesson but in the previous one, too) and used Postman many times to ensure that our API is working correctly.

However, not every bug is easy to find.

For example, we didn't test what happens when the JWT token expires or when we provide an invalid token. Currently, the application works correctly only if the JWT token is valid or if it is not present at all. If the token is not present, only requests to endpoints that do not rely on the "previous" value should work. Let's use Postman to test this behaviour.

Let's use an invalid JWT – to do so, type random text (for example, "abc") in the Token field in Postman:



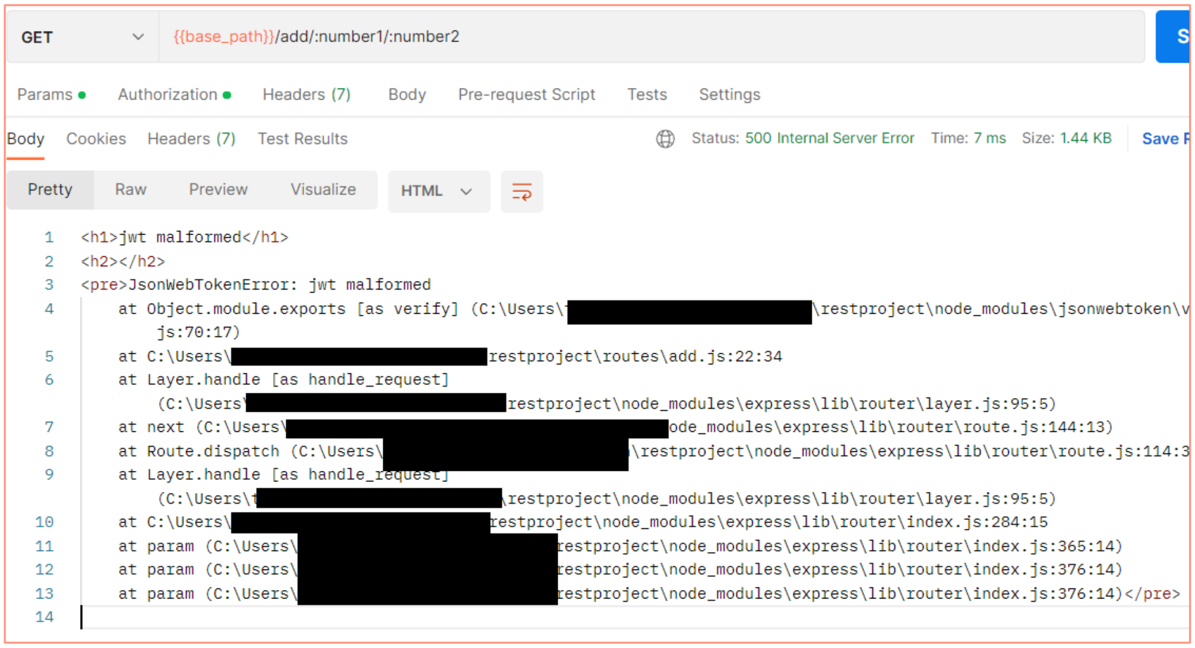
Type: Bearer Tok...

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables ↗

Token: abc

We get the result:



```
<h1>jwt malformed</h1>
<h2></h2>
<pre>JsonWebTokenError: jwt malformed
    at Object.module.exports [as verify] (C:\Users\[redacted]\restproject\node_modules\jsonwebtoken\v
        js:70:17)
    at C:\Users\[redacted]restproject\routes\add.js:22:34
    at Layer.handle [as handle_request]
        (C:\Users\[redacted]restproject\node_modules\express\lib\router\layer.js:95:5)
    at next (C:\Users\[redacted]ode_modules\express\lib\router\route.js:144:13)
    at Route.dispatch (C:\Users\[redacted]\restproject\node_modules\express\lib\router\route.js:114:3
    at Layer.handle [as handle_request]
        (C:\Users\[redacted]restproject\node_modules\express\lib\router\layer.js:95:5)
    at C:\Users\[redacted]restproject\node_modules\express\lib\router\index.js:284:15
    at param (C:\Users\[redacted]restproject\node_modules\express\lib\router\index.js:365:14)
    at param (C:\Users\restproject\node_modules\express\lib\router\index.js:376:14)
    at param (C:\Users\restproject\node_modules\express\lib\router\index.js:376:14)</pre>
```

Now, let’s check a case where the token is valid but has expired. To get an expired token, we can either wait until our token expires (we set the expiration time to 1 hour) or temporarily change the expiration time and create a new token. Let’s try the second option:

Stop the application and open the auth.js router file.

Temporarily change the line

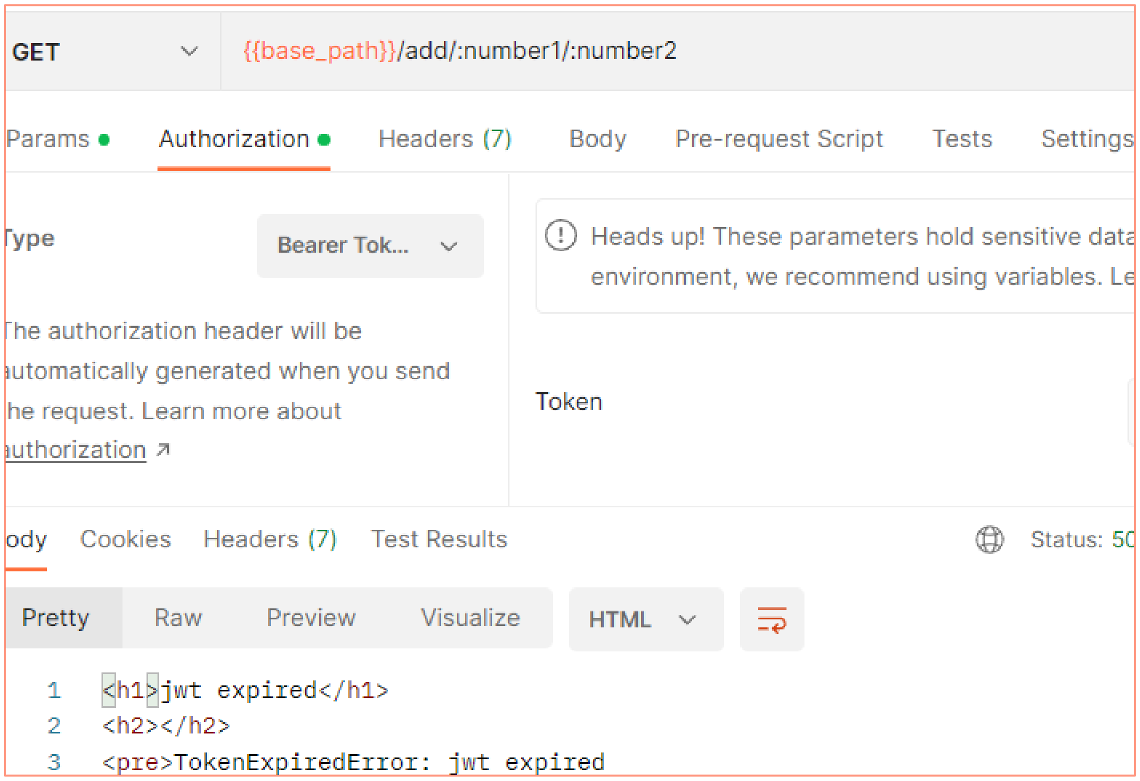
```
{ expiresIn: "1h" }
```

to

```
{ expiresIn: "20s" }
```

Start the application, log in to the application with Postman, and replace the Token field with the new token.

We should get very similar results:

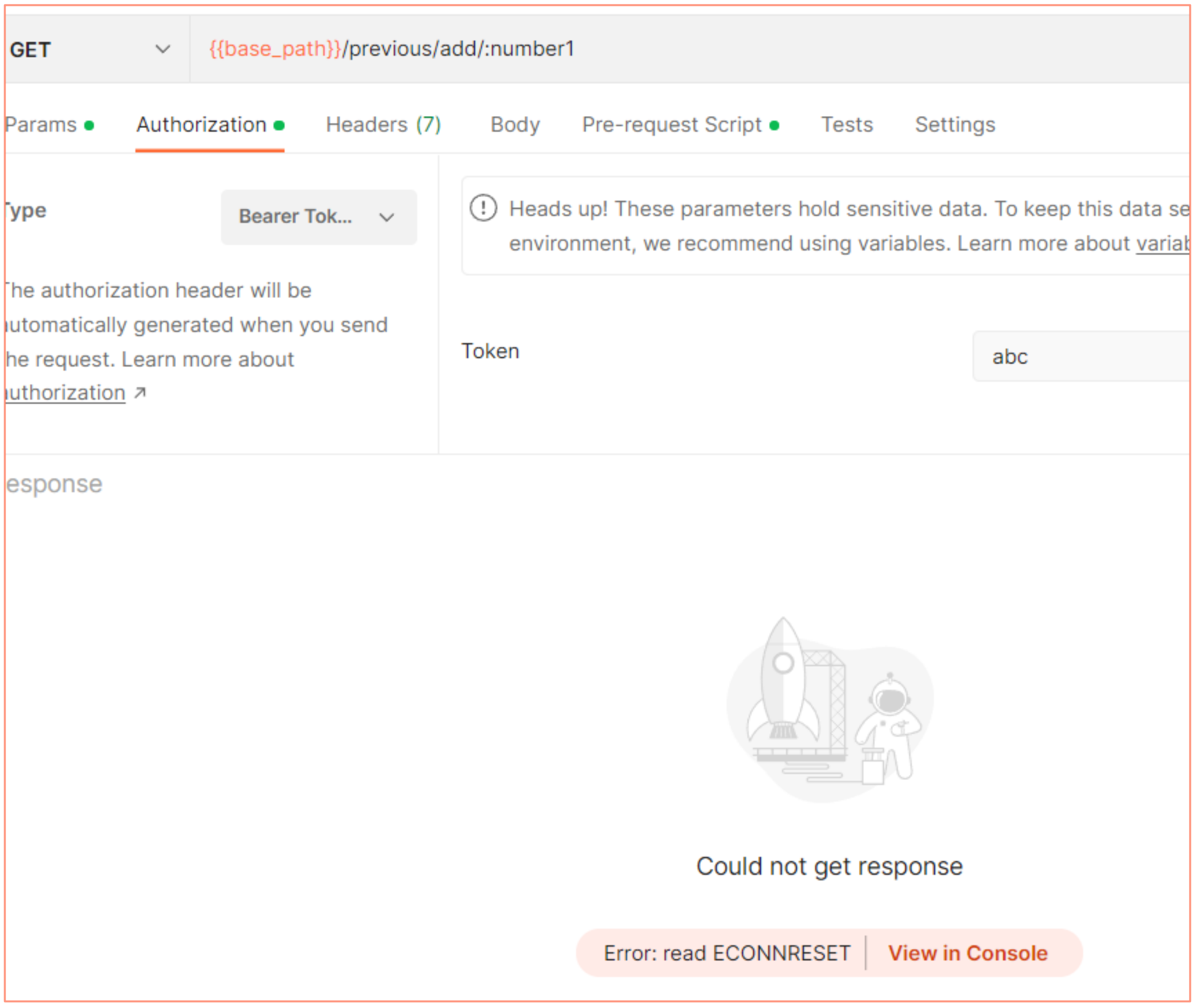


Heads up! These parameters hold sensitive data
environment, we recommend using variables. Le

The authorization header will be
automatically generated when you send
the request. Learn more about
authorization

```
<h1>jwt expired</h1>
<h2></h2>
<pre>TokenExpiredError: jwt expired
```

Let’s see how it works with the /previous methods:


GET {{base_path}}/previous/add/:number1

Params ● Authorization ● Headers (7) Body Pre-request Script ● Tests Settings

Type Bearer Tok...

he authorization header will be
utomatically generated when you send
he request. Learn more about
uthorization ↗

Heads up! These parameters hold sensitive data. To keep this data se
environment, we recommend using variables. Learn more about variab

Token abc

esponse

Could not get response

Error: read ECONNRESET | View in Console

In this case, the application completely crashes (the same as when we provide an expired token).

Now that we have identified the bugs, let’s see how to fix them.

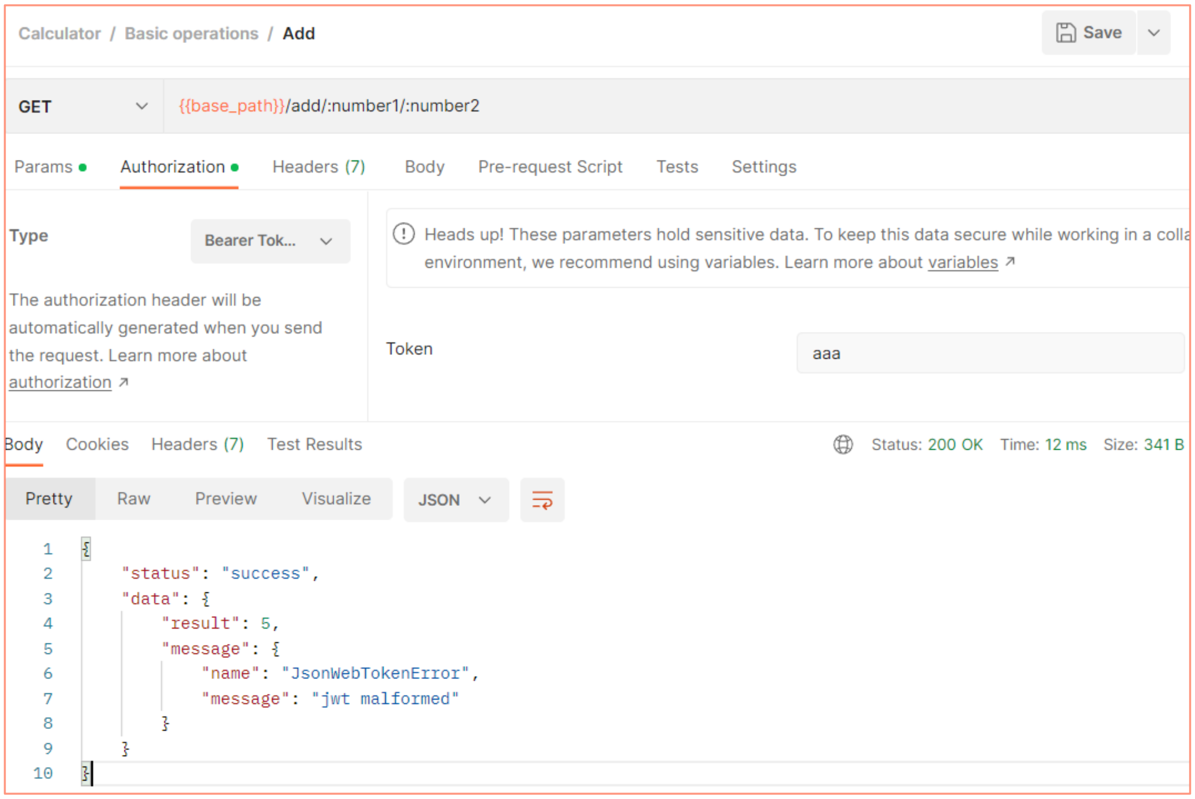
In the case of basic operations, we always want to return the response, regardless of the JWT token. To do this, let’s use a try/catch block.

Replace the code in the add.js file:

```
    if(token) {
        const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
        resultService.create("add", result, decodedToken.id);
    }
    res.jsend.success({"result": result});
```

With:

```
if(token) {
        try {
            const decodedToken = jwt.verify(token, 
process.env.TOKEN_SECRET );
            resultService.create("add", result, 
decodedToken.id);
        }
        catch(err) {
            res.jsend.success({"result": result, "message": 
err});
        }
    }
    res.jsend.success({"result": result});
```


Calculator / Basic operations / Add

Save

GET {{base_path}}/add/:number1/:number2

Params ● Authorization ● Headers (7) Body Pre-request Script Tests Settings

Type: Bearer Tok...

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a colla
environment, we recommend using variables. Learn more about variables ↗

Token: aaa

Body Cookies Headers (7) Test Results — Status: 200 OK Time: 12 ms Size: 341 B

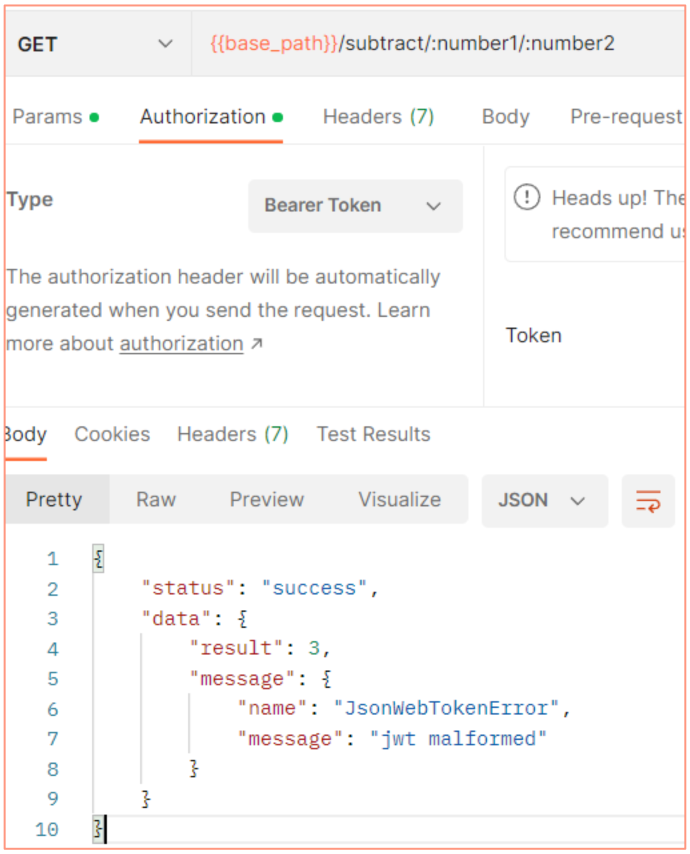
Pretty Raw Preview Visualize JSON

```
{
    "status": "success",
    "data": {
        "result": 5,
        "message": {
            "name": "JsonWebTokenError",
            "message": "jwt malformed"
        }
    }
}
```

The result is correct, but the message about the error is also included. Now, fix endpoints /subtract, /multiply, and /divide in the same way:

subtract.js:

```
    if(token) {
        try {
            const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
            resultService.create("subtract", result,
decodedToken.id);
        }
        catch(err) {
            res.jsend.success({"result": result, "message":
err});
        }
    }
    res.jsend.success({"result": result});
```



GET {{base_path}}/subtract/:number1/:number2

Params Authorization Headers (7) Body Pre-request

Type Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! The recommend u

Token

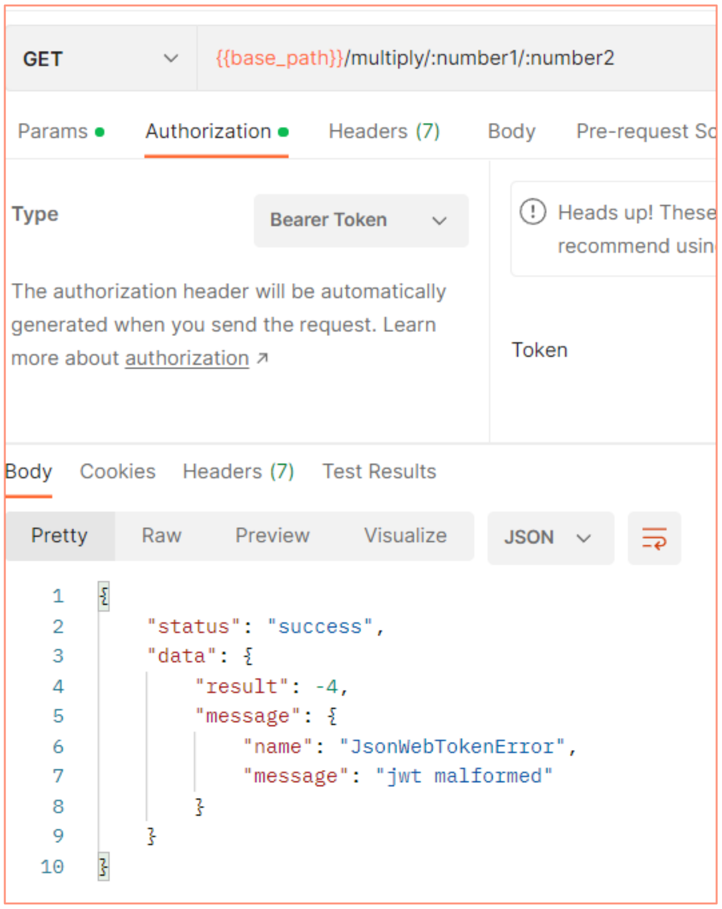
Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "status": "success",
    "data": {
        "result": 3,
        "message": {
            "name": "JsonWebTokenError",
            "message": "jwt malformed"
        }
    }
}
```

multiply.js:

```
 if(token) {
        try {
            const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
            resultService.create("multiply", result,
decodedToken.id);
        }
        catch(err) {
            res.jsend.success({"result": result, "message":
err});
        }
    }
    res.jsend.success({"result": result});
```


GET {{base_path}}/multiply/:number1/:number2

Params ● Authorization ● Headers (7) Body Pre-request Sc

Type Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These
recommend usin

Token

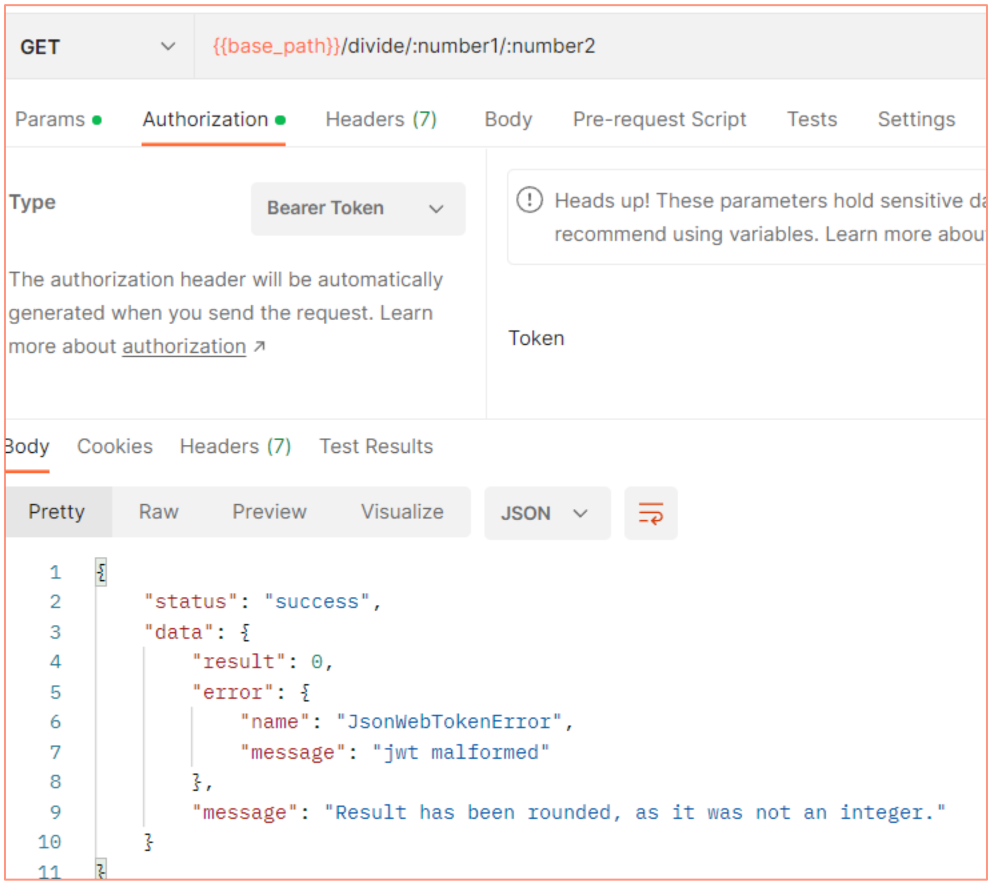
Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "status": "success",
    "data": {
        "result": -4,
        "message": {
            "name": "JsonWebTokenError",
            "message": "jwt malformed"
        }
    }
}
```

divide.js:

```
    if(token) {
        try {
            const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
            resultService.create("divide",
Math.round(result), decodedToken.id);
        }
        catch(err) {
            if (Number.isInteger(result)) {
                res.jsend.success({"result": result, "error":
err});
            }
            else {
                res.jsend.success({"result":
Math.round(result), "error": err, "message": "Result has been
rounded, as it was not an integer."});
            }
        }
    }
    if (Number.isInteger(result)) {
        res.jsend.success({"result": result});
    }
    else {
        res.jsend.success({"result": Math.round(result),
"message": "Result has been rounded, as it was not an
integer."});
    }
```



The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

```
{
    "status": "success",
    "data": {
        "result": 0,
        "error": {
            "name": "JsonWebTokenError",
            "message": "jwt malformed"
        },
        "message": "Result has been rounded, as it was not an integer."
    }
}
```

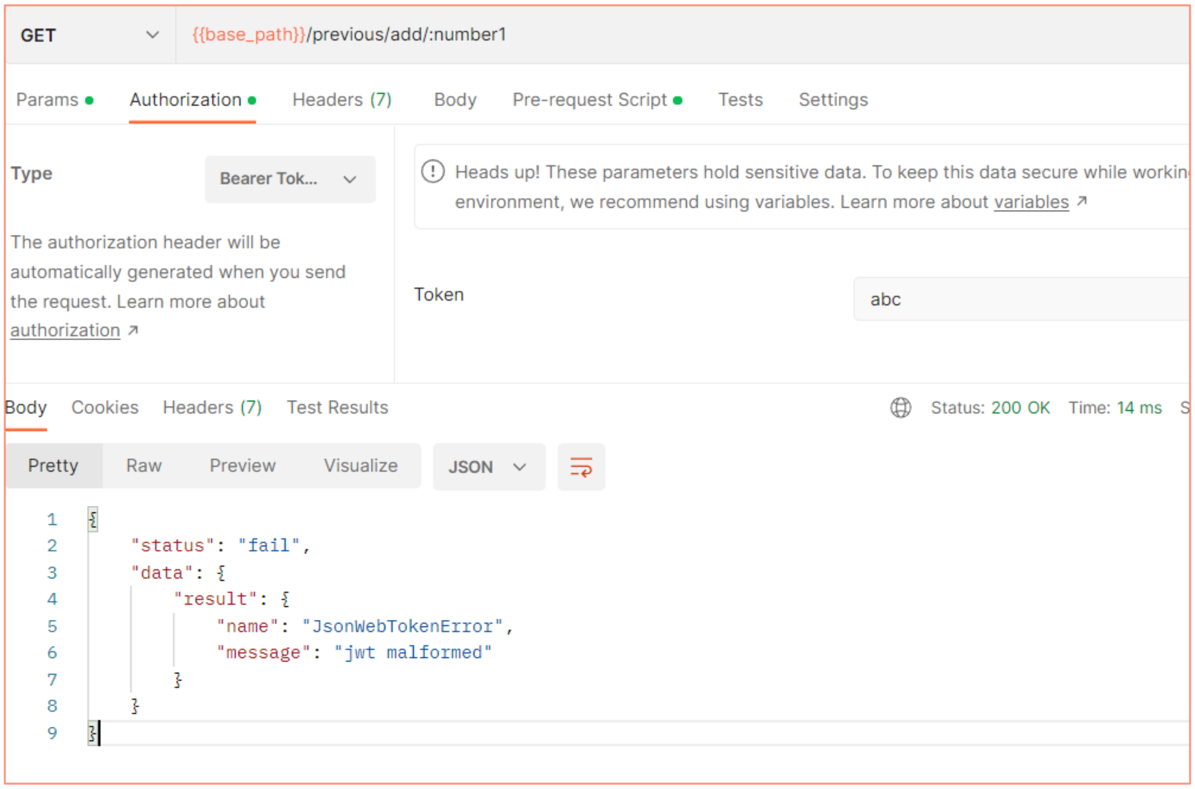
A similar issue occurs with the /previous endpoints as well. To fix this, let's replace the code:

```
    const decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    if (decodedToken == null) {
        return res.jsend.fail({"result": "JWT token is
invalid"});
    }
```

With:

```
    let decodedToken;
    try {
        decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    }
    catch(err) {
        return res.jsend.fail({"result": err});
    }
```

We should see the result:



GET {{base_path}}/previous/add/:number1

Type: Bearer Tok...

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while workin environment, we recommend using variables. Learn more about variables ↗

Token: abc

Status: 200 OK Time: 14 ms

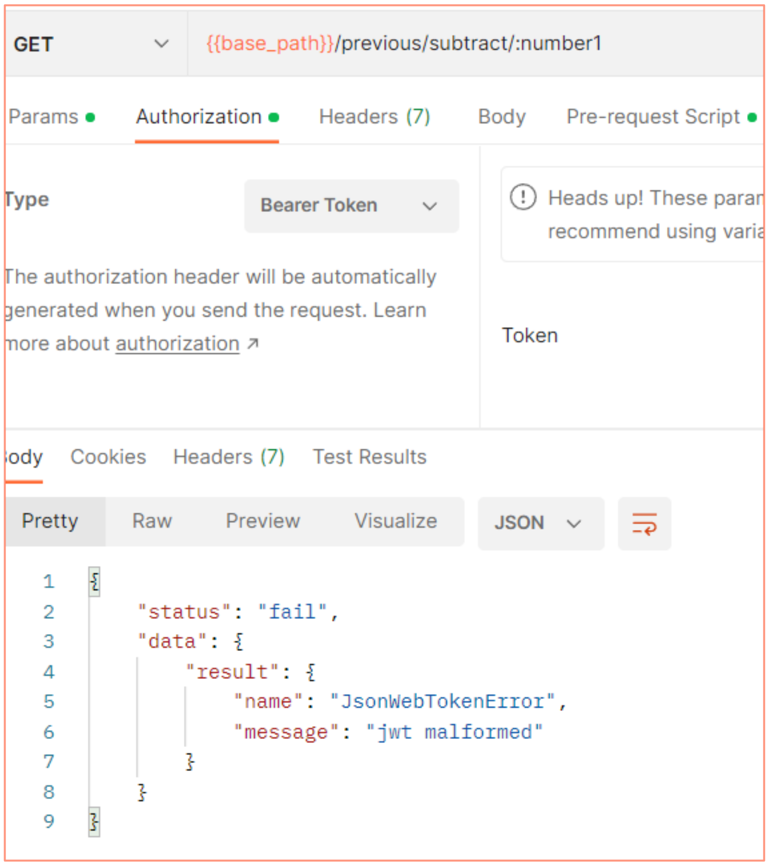
```
{
    "status": "fail",
    "data": {
        "result": {
            "name": "JsonWebTokenError",
            "message": "jwt malformed"
        }
    }
}
```

Make sure that this change is made for all four endpoints.

Entire previous.js router file:

```
var express = require('express');
var jsend = require('jsend');
var router = express.Router();
var db = require("../models");
var ResultService = require("../services/ResultService")
var resultService = new ResultService(db);
var jwt = require('jsonwebtoken')

router.use(jsend.middleware);
router.get('/add/:number1', async function(req, res, next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    let decodedToken;
    try {
        decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    }
    catch(err) {
        return res.jsend.fail({"result": err});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value + number1;
    resultService.create("add", result, decodedToken.id);
    if( result % decodedToken.email.length )
    res.jsend.success({"result": result, "previousOperation":
previous.OperationName, "previousValue": previous.Value});
});

router.get('/subtract/:number1', async function(req, res,
next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    let decodedToken;
    try {
        decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    }
    catch(err) {
        return res.jsend.fail({"result": err});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value - number1;
    resultService.create("subtract", result,
decodedToken.id);
    res.jsend.success({"result": result, "previousOperation":
previous.OperationName, "previousValue": previous.Value});
});

router.get('/multiply/:number1', async function(req, res,
next) {
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    let decodedToken;
    try {
        decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    }
    catch(err) {
        return res.jsend.fail({"result": err});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value * number1;
    resultService.create("multiply", result,
decodedToken.id);
    res.jsend.success({"result": result, "previousOperation":
previous.OperationName, "previousValue": previous.Value});
});

router.get('/divide/:number1', async function(req, res, next)
{
    const number1 = parseInt(req.params.number1);
    if(isNaN(number1)) {
        return res.jsend.fail({"number1": "number1 is not in
correct format"});
    }
    if(number1 == 0) {
        return res.jsend.fail({"number1": "number1 cannot be
0"});
    }
    const token = req.headers.authorization?.split(' ')[1];
    if(!token) {
        return res.jsend.fail({"result": "JWT token not
provided"});
    }
    let decodedToken;
    try {
        decodedToken = jwt.verify(token,
process.env.TOKEN_SECRET );
    }
    catch(err) {
        return res.jsend.fail({"result": err});
    }
    const previous = await
resultService.getOne(decodedToken.id);
    const result = previous.Value / number1;
    resultService.create("divide", Math.round(result),
decodedToken.id);
    if (Number.isInteger(result)) {
        res.jsend.success({"result": result,
"previousOperation": previous.OperationName, "previousValue":
previous.Value});
    }
    else {
        res.jsend.success({"result": Math.round(result),
"previousOperation": previous.OperationName, "previousValue":
previous.Value, "message": "Result has been rounded, as it
was not an integer."});
    }
});

module.exports = router;
```

subtract:

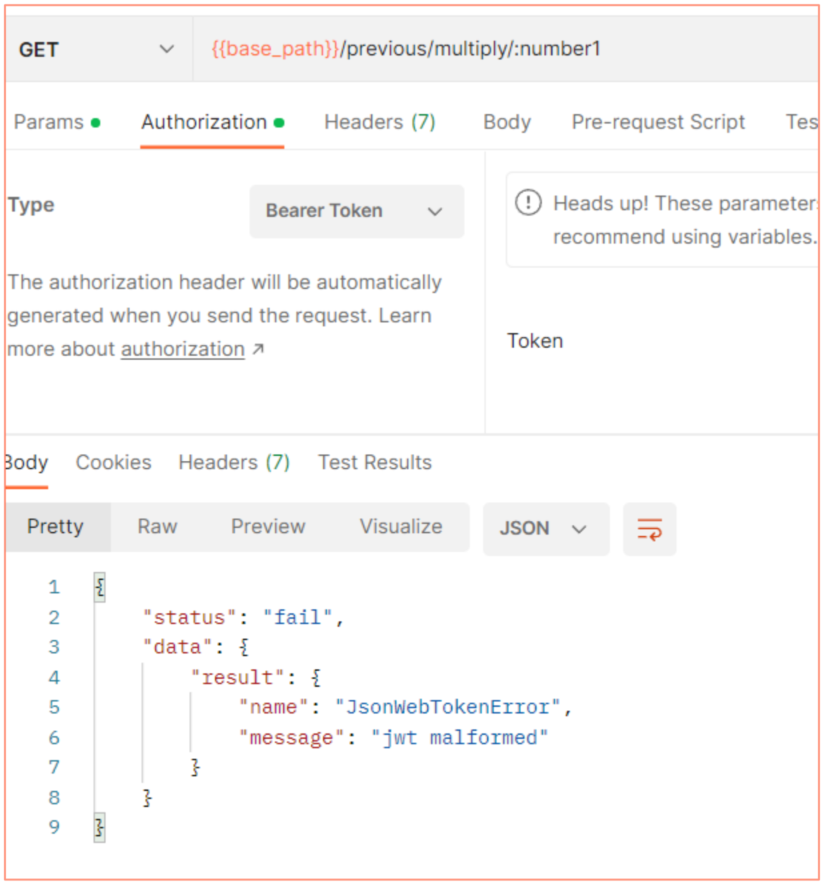


{{base_path}}/previous/subtract/:number1

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

```
{
    "status": "fail",
    "data": {
        "result": {
            "name": "JsonWebTokenError",
            "message": "jwt malformed"
        }
    }
}
```

multiply:


GET {{base_path}}/previous/multiply/:number1

Type: Bearer Token

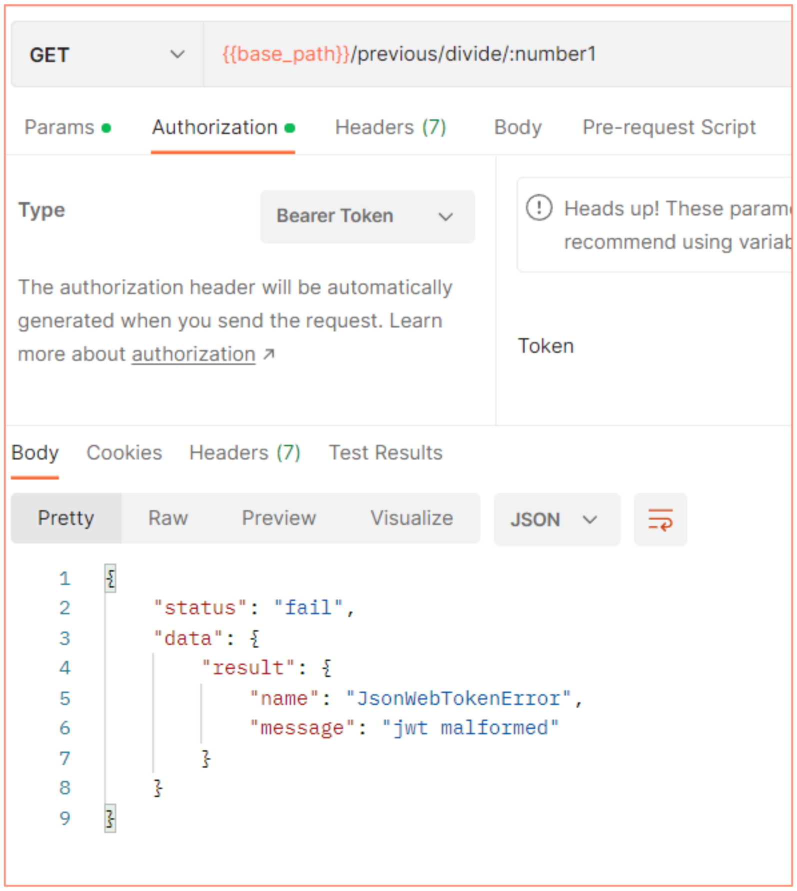
The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameter
recommend using variables.

Token

```
{
    "status": "fail",
    "data": {
        "result": {
            "name": "JsonWebTokenError",
            "message": "jwt malformed"
        }
    }
}
```

divide:



```
{
    "status": "fail",
    "data": {
        "result": {
            "name": "JsonWebTokenError",
            "message": "jwt malformed"
        }
    }
}
```

Summary

In this course, we implemented an API that is not a web application. In such cases, JWT authentication is much more convenient than a session. We don't have to generate a JWT token on login, though – often, it is generated after registration and lasts much longer than one hour.

We created our first complete documentation with multiple examples for each request. We also implemented integration tests to ensure that sequences of the most common actions always work. We can run these tests with one simple test command.

In bigger applications, it might be a good idea to use unit testing. In our application, there was not much to test, as arithmetic operations don't necessarily need that. If we want to introduce unit testing, we should restructure our code into smaller parts instead of having the entire logic in router files and services.

After these lessons, students should be able to create their own RESTful applications – web applications as well as external APIs.

What did I learn in this lesson?

This lesson provided the following insights:

- How to determine whether an application should be covered with unit tests.
- How to create integration tests with JWT authentication.
- How to search for bugs in an application.

References

	INFO Please note that the reference list below was compiled when writing the course content to credit the sources used. Therefore, some links that worked at the time of writing may not work now or may have changed. However, accessing these references is not a requirement for this module. Note that all the required reading is specified in the READ blocks in the content.

jestjs.io. (n.d.). *Getting Started · Jest.* [online] Available at: https://jestjs.io/docs/getting-started.

5.4. Lesson task - Tests

The task

In this lesson, we implemented tests for our API and fixed minor bugs. In this task, you need to test the signup method. However, as tests shouldn't affect our database, you should implement the method in the user service to delete the user of the provided email. In your test, you should create a new user, log in to the application, make at least one call to any of the /previous operations, and delete the user at the end.

5.5. Lesson - Self-study

Introduction

This is a self-study lesson which consolidates the REST APIs project.

Materials

READ

1. Page: 4.x API on Express.
2. Page: Overview on Postman Learning Center.
3. Page: Getting Started on Jest.

5.5. Lesson task - Self-study

The task

In this task, you need to add the /sqrt/:number and /previous/sqrt endpoints, which return the square root of the number. These should behave similarly to the /divide endpoint – they should always return an integer, and if the result was rounded, the response should contain a message about that. Remember to update the tests and documentation to cover these new endpoints as well.

Noroff
Education