Michael Herman

Software Developer

-

Search

Navigate…

Developing a RESTful API With Node and TypeScript

Nov 5th, 2016 7:38 am

This tutorial details how to develop a RESTful API with NodeJS, ExpressJS, and TypeScript using test-driven development (TDD).

We will be using:

- NodeJS v7.0.0
- ExpressJS v4.14.0
- TypeScript v2.0.6

Additionally, we will use *tsconfig.json* to configure the project, Gulp to automate transpilation, and d.ts for managing typings with `npm`.

Updates:

- 03/12/2017: Updated the *gulpfile.js* and *package.json*

Contents

Project Setup

To start, we need to set a means to transpile TypeScript into JavaScript that works well with Node. Enter the *tsconfig.json* file. This is similar to a *package.json* or *.babelrc* or really any project-level configuration file you may use. As you can probably guess, it will configure the TypeScript compiler for the project.

Make a new directory to hold the project, and add a *tsconfig.json* file:

```
$ mkdir typescript-node-api
$ cd typescript-node-api
$ touch tsconfig.json
```

We'll use a pretty basic configuration for today:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs"
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

Here:

- in `compilerOptions` we tell TypeScript that we'll be targeting ES2015 and that we'd like a CommonJS style module as output (the same module style that Node uses)
- the `include` section tells the compiler to look for *.ts* files in the "src" directory
- the `exclude` section tells the compiler to ignore anything in "node_modules"

  NOTE: Review the TypeScript docs if you want more info on all the options you can define in the *tsconfig.json* file, They are the same options that you can pass directly to the TypeScript compiler wrapper.

Add a "src" directory:

```
$ mkdir src
```

Before moving any further, let's make sure this configuration works like we expect using the TypeScript compiler wrapper. Create a *package.json* and install TypeScript:

```
$ npm init -y
$ npm install typescript@2.0.6 --save-dev
```

Create a new file called *test.ts* within the "src" directory and add the following:

```
console.log('Hello, TypeScript!');
```

Finally, let's run this one-liner through the compiler. From the project root, run the `tsc` that we installed above in our test file with:

```
$ node_modules/.bin/tsc
```

Given no arguments, `tsc` will first look at *tsconfig.json* for instruction. When it finds the config, it uses those settings to build the project. You should see a new file inside of "src" called *test.js* with the same line of code in it. Awesome!

Now that the compiler is installed and working, let's change up the config to make things easier on ourselves. First, we'll add an `outDir` property to the `compilerOptions` of `tsconfig.json` to tell TypeScript to place all of our transpiled JavaScript into a different directory rather than compiling the files right next to their source *.ts* files:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "dist"
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

Remove the *test.js* file from the "src" folder. Now, run the compiler again, and you'll see that `test.js` is delivered to the `dist` directory.

This is much nicer, but let's take it one step further. Instead of returning to the terminal after each change and manually running the compiler each time let's automate the process with Gulp:

```
$ npm install gulp@3.9.1 gulp-typescript@3.1.1 --save-dev
```

> **NOTE:** You'll also want to globally install `gulp` to trigger Gulp tasks easily from the command line: `npm install -g gulp@3.9.1`

Add a *gulpfile.js* to the root of the directory. This is where we'll automate the compiling of our source files:

1. Pull in the *tsconfig.json* and pass it to `gulp-typescript` for configuration
2. Tell `gulp-typescript` to transpile our project and deliver it to "dist"
3. Tell Gulp to watch our source *.ts* files, so that our transpiled JavaScript automatically gets rebuilt upon file changes

Add the code:

```
const gulp = require('gulp');
const ts = require('gulp-typescript');
const JSON_FILES = ['src/*.json', 'src/**/*.json'];

// pull in the project TypeScript config
const tsProject = ts.createProject('tsconfig.json');

gulp.task('scripts', () => {
  const tsResult = tsProject.src()
  .pipe(tsProject());
  return tsResult.js.pipe(gulp.dest('dist'));
});

gulp.task('watch', ['scripts'], () => {
  gulp.watch('src/**/*.ts', ['scripts']);
});

gulp.task('assets', function() {
  return gulp.src(JSON_FILES)
  .pipe(gulp.dest('dist'));
});

gulp.task('default', ['watch', 'assets']);
```

To test this out, remove *dist/test.js* and run `gulp` from the project root. You'll see Gulp start up, and *test.js* should be compiled again and placed into "dist". Awesome! Our project is now configured.

Let's move on to working with Express…

Express Config

For our Express server, we'll use the express-generator as our template. We'll start with what would be the "bin/www" file and create an HTTP server, initialize it, and then attach our Express app to it.

Install Express along with debug (to provide some nice terminal output while developing):

```
$ npm install express@4.14.0 debug@2.2.0 --save
```

In TypeScript, when you install third-party packages, you should also pull down the package's type definitions. This tells the compiler about the structure of the module that you're using, giving it the information needed to properly evaluate the types of structures that you use from the module.

Before TypeScript 2.0, dealing with *.d.ts* (type definition) files could be a real nightmare. The language had a built in tool, `tsd`, but it was a bear to work with and you had to decorate your TypeScript files with triple-slash comments to pull declarations into your file. Then typings came along and things were much better, but there were still some issues and now you had another separate package manager to manage in your project.

With TypeScript 2.0, TypeScript definitions are managed by `npm` and installed as scoped packages. This means two things for you:

1. Dependency management is simplified
2. To install a type module, prefix its name with `@types/`

Install the type definitions for Node, Express, and debug:

```
$ npm install @types/node@6.0.46 @types/express@4.0.33 @types/debug@0.0.29 --save-dev
```

With that, we’re ready to create the HTTP server. Rename *src/test.ts* to *src/index.ts*, remove the console log, and add the following:

```
import * as http from 'http';
import * as debug from 'debug';

import App from './App';

debug('ts-express:server');

const port = normalizePort(process.env.PORT || 3000);
App.set('port', port);

const server = http.createServer(App);
server.listen(port);
server.on('error', onError);
server.on('listening', onListening);

function normalizePort(val: number|string): number|string|boolean {
  let port: number = (typeof val === 'string') ? parseInt(val, 10) : val;
  if (isNaN(port)) return val;
  else if (port >= 0) return port;
  else return false;
}

function onError(error: NodeJS.ErrnoException): void {
  if (error.syscall !== 'listen') throw error;
  let bind = (typeof port === 'string') ? 'Pipe ' + port : 'Port ' + port;
  switch(error.code) {
    case 'EACCES':
      console.error(`${bind} requires elevated privileges`);
      process.exit(1);
      break;
    case 'EADDRINUSE':
      console.error(`${bind} is already in use`);
      process.exit(1);
      break;
    default:
      throw error;
  }
}

function onListening(): void {
  let addr = server.address();
  let bind = (typeof addr === 'string') ? `pipe ${addr}` : `port ${addr.port}`;
  debug(`Listening on ${bind}`);
}
```

If you’re using an editor with rich TypeScript support, it’s not going to appreciate `import App from './App';`, but just ignore it for now. The rest of this file is pretty straightforward:

1. Use `debug` to set up some terminal logging for the app
2. Get a port value from the environment, or set a default port number of 3000
3. Create the HTTP server, and pass `App` to it (this will be our Express app)
4. Set up some basic error handling and a terminal log to show us when the app is ready and listening

Since this file will start the app, let's also add a `"start"` script to `package.json` for convenience:

```
"scripts": {
  "start": "node dist/index.js"
},
```

Before we can start the app up, let's make the *App.ts* file that we referenced on in *index.ts*. It's also a good time to go ahead and install the dependencies we'll use in the Express application.

```
$ touch src/App.ts
$ npm install express@4.14.0 body-parser@1.15.2 morgan@1.7.0 --save
$ npm install @types/body-parser@0.0.33 @types/morgan@1.7.32 --save-dev
```

Inside of *App.ts* let's create the `App` class to package up and configure our Express server. An instance of `App` will:

- Hold a reference to our instance of Express
- Automatically configure any middleware that we want to use
- Attach any routers/route handlers that we create

Essentially, it's going to bootstrap the app and deliver it to the call to `http.createServer` in *index.ts*.

App.ts:

```
import * as path from 'path';
import * as express from 'express';
import * as logger from 'morgan';
import * as bodyParser from 'body-parser';

// Creates and configures an ExpressJS web server.
class App {

  // ref to Express instance
  public express: express.Application;

  //Run configuration methods on the Express instance.
  constructor() {
    this.express = express();
    this.middleware();
    this.routes();
  }

  // Configure Express middleware.
  private middleware(): void {
    this.express.use(logger('dev'));
    this.express.use(bodyParser.json());
    this.express.use(bodyParser.urlencoded({ extended: false }));
  }

  // Configure API endpoints.
  private routes(): void {
    /* This is just to get up and running, and to make sure what we've got is
     * working so far. This function will change when we start to add more
     * API endpoints */
    let router = express.Router();
    // placeholder route handler
    router.get('/', (req, res, next) => {
      res.json({
        message: 'Hello World!'
      });
    });
    this.express.use('/', router);
  }

}

export default new App().express;
```

Here's a quick rundown:

- The `App.express` field holds a reference to Express. This makes it easier to access `App` methods for configuration and simplifies exporting the configured instance to *index.ts*.
- `App.middleware` configures our Express middleware. Right now we're using the `morgan` logger and `body-parser`.
- `App.routes` will be used to link up our API endpoints and route handlers.

  NOTE: If you have a text editor with rich TypeScript support, the error in *index.ts* should have disappeared.

Currently, there's a simple placeholder handler for the base URL that should return a JSON payload with `{ "message": "Hello World!" }`. Before writing more code, let's make sure that we're starting with a working, listening, and hopefully responding server. We're going to use httpie for this quick sanity check.

Compile, and then run the server:

```
$ gulp scripts
$ npm start
```

To test, open a new terminal window and run:

```
$ http localhost:3000
```

If everything has gone well, you should get a response similar to this:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 26
Content-Type: application/json; charset=utf-8
X-Powered-By: Express

{
    "message": "Hello World!"
}
```

The server is listening! Now we can start building the API.

The API

Since we're good developers (and citizens), let's utilize TDD (test-driven development) while we build out the API. That means we want to set up a testing environment. We'll be writing our test files in TypeScript, and using Mocha and Chai to create the tests. Let's start by installing these to our `devDependencies`:

```
$ npm install mocha@3.1.2 chai@3.5.0 chai-http@3.0.0 --save-dev
$ npm install @types/mocha@2.2.32 @types/chai@3.4.34 @types/chai-http@0.0.29 --save-dev
```

If we write out tests in *.ts* files, we'll need to make sure that Mocha can understand them. By itself, Mocha can only interpret JavaScript files, not TypeScript. There are a number of different ways to accomplish this. To keep it simple, we'll leverage `ts-node`, so that we can provide TypeScript interpretation to the Mocha environment without having to transpile the tests into different files. `ts-node` will interpret and transpile our TypeScript in memory as the tests are run.

Start by installing `ts-node`:

```
$ npm install ts-node@1.6.1 --save-dev
```

Now, in `package.json`, add a `test` script to run mocha with the `ts-node` register:

```
"scripts": {
  "start": "node dist/index.js",
  "test": "mocha --reporter spec --compilers ts:ts-node/register 'test/**/*.test.ts'"
},
```

With the environment all set up, let's write our first test for the "Hello World" route we created in *App.ts*. Start by adding a "test" folder to the route, and add a file called *helloWorld.test.ts*:

```
import * as mocha from 'mocha';
import * as chai from 'chai';
import chaiHttp = require('chai-http');

import app from '../src/App';

chai.use(chaiHttp);
const expect = chai.expect;

describe('baseRoute', () => {

  it('should be json', () => {
    return chai.request(app).get('/')
    .then(res => {
      expect(res.type).to.eql('application/json');
    });
  });

  it('should have a message prop', () => {
    return chai.request(app).get('/')
    .then(res => {
      expect(res.body.message).to.eql('Hello World!');
    });
  });

});
```

In the terminal, run `npm test` you should see both test pass for the `baseRoute` describe block. Excellent! Now we can test our routes as we build out the API.

First Endpoint

Our API will be delivering data on superheros, so we'll need to have a datastore for the API to access. Rather than setting up a full database, for this example let's use a JSON file as our "database". Grab the data here and save it to a new file called *data.json* in the "src" folder.

With this little store of data, we'll implement a CRUD interface for the superhero resource. To start, let's implement an endpoint that returns all of our superheros. Here's a test for this endpoint:

```
import * as mocha from 'mocha';
import * as chai from 'chai';
import chaiHttp = require('chai-http');

import app from '../src/App';

chai.use(chaiHttp);
const expect = chai.expect;

describe('GET api/v1/heroes', () => {

  it('responds with JSON array', () => {
    return chai.request(app).get('/api/v1/heroes')
      .then(res => {
        expect(res.status).to.equal(200);
        expect(res).to.be.json;
        expect(res.body).to.be.an('array');
        expect(res.body).to.have.length(5);
      });
  });

  it('should include Wolverine', () => {
    return chai.request(app).get('/api/v1/heroes')
      .then(res => {
        let Wolverine = res.body.find(hero => hero.name === 'Wolverine');
        expect(Wolverine).to.exist;
        expect(Wolverine).to.have.all.keys([
          'id',
          'name',
          'aliases',
          'occupation',
          'gender',
          'height',
          'hair',
          'eyes',
          'powers'
        ]);
      });
  });

});
```

Add this to a new file called *test/hero.test.ts*.

To summarize, the test asserts that:

- the endpoint is at `/api/v1/heroes`
- it returns a JSON array of hero objects
- we can find Wolverine, and his object contains all the keys that we expect

When you run `npm test`, you should see this one fail with a `Error: Not Found` in the terminal. Good. This is expected since we haven't set up the route yet.

It's finally that time: Let's implement our CRUD routes!

To start, create a new folder `src/routes` and create a new file inside the directory named *HeroRouter.ts*. Inside of here, we'll implement each CRUD route for the superhero resource. To hold each route, we'll have a `HeroRouter` class that defines the handler for each route, and an `init` function that attaches each handler to an endpoint with the help of an instance of `Express.Router`.

```
import {Router, Request, Response, NextFunction} from 'express';
const Heroes = require('../data');

export class HeroRouter {
  router: Router

  /**
   * Initialize the HeroRouter
   */
  constructor() {
    this.router = Router();
    this.init();
  }

  /**
   * GET all Heroes.
   */
  public getAll(req: Request, res: Response, next: NextFunction) {
    res.send(Heroes);
  }

  /**
   * Take each handler, and attach to one of the Express.Router's
   * endpoints.
   */
  init() {
    this.router.get('/', this.getAll);
  }

}

// Create the HeroRouter, and export its configured Express.Router
const heroRoutes = new HeroRouter();
heroRoutes.init();

export default heroRoutes.router;
```

We also need to modify the `routes` function of `App` to use our new `HeroRouter`. Add the import at the top of *App.ts*:

```
import HeroRouter from './routes/HeroRouter';
```

Then add the API endpoint to `private routes(): void`:

```
// Configure API endpoints.
private routes(): void {
  /* This is just to get up and running, and to make sure what we've got is
   * working so far. This function will change when we start to add more
   * API endpoints */
  let router = express.Router();
  // placeholder route handler
  router.get('/', (req, res, next) => {
    res.json({
      message: 'Hello World!'
    });
  });
  this.express.use('/', router);
  this.express.use('/api/v1/heroes', HeroRouter);
}
```

Now run `npm test` and ensure that our tests pass:

```
baseRoute
  ✓ should be json
  ✓ should have a message prop

GET api/v1/heroes
  ✓ responds with JSON array
  ✓ should include Wolverine
```

Second Endpoint

Now we're really rolling! Before moving on though, let's break the process down since we'll be repeating it to create and attach each of our route handlers:

1. Create a method on `HeroRouter` that takes the arguments of your typical Express request handler: `request`, `response`, and `next`.
2. Implement the server's response for the endpoint.
3. Inside of `init`, use `HeroRouter`'s instance of the Express `Router` to attach the handler to an endpoint of the API.

We'll follow this same workflow for each endpoint, and can leave `App` alone. All of our `HeroRouter` endpoints will be appended to `/api/v1/heroes`. Let's implement a `GET` handler that returns a single hero by the `id` property. We'll test the endpoint by looking for Luke Cage, who has an `id` of 1.

```
describe('GET api/v1/heroes/:id', () => {

  it('responds with single JSON object', () => {
    return chai.request(app).get('/api/v1/heroes/1')
      .then(res => {
        expect(res.status).to.equal(200);
        expect(res).to.be.json;
        expect(res.body).to.be.an('object');
      });
  });

  it('should return Luke Cage', () => {
    return chai.request(app).get('/api/v1/heroes/1')
      .then(res => {
        expect(res.body.hero.name).to.equal('Luke Cage');
      });
  });

});
```

And the route handler:

```
/**
 * GET one hero by id
 */
public getOne(req: Request, res: Response, next: NextFunction) {
  let query = parseInt(req.params.id);
  let hero = Heroes.find(hero => hero.id === query);
  if (hero) {
    res.status(200)
      .send({
        message: 'Success',
        status: res.status,
        hero
      });
  }
  else {
    res.status(404)
      .send({
        message: 'No hero found with the given id.',
        status: res.status
      });
  }
}

/**
 * Take each handler, and attach to one of the Express.Router's
 * endpoints.
 */
init() {
  this.router.get('/', this.getAll);
  this.router.get('/:id', this.getOne);
}
```

Run the tests!

```
baseRoute
  ✓ should be json
  ✓ should have a message prop

```

```
GET api/v1/heroes
  ✓ responds with JSON array
  ✓ should include Wolverine
    ✓ responds with single JSON object
    ✓ should return Luke Cage
```

What's Next?

For the hero resource, we should have endpoints for updating a hero and deleting a hero, but we'll leave that for you to implement. The structure that we've set up here should guide you through creating those last endpoints.

Once the hero resource is implemented, we could add more resources to the API easily. To follow the same process we would:

1. Create a new file inside of *src/routes* to be the router for the resource.
2. Attach the resource router to the Express app inside of the `routes` method of `App`.

Now you're up and running with Express and TypeScript 2.0. Go build something! You can grab the code from the typescript-node-api repo. Cheers!

Authored by Michael Herman Nov 5th, 2016 7:38 am node

5K

« Token-Based Authentication with Node Building a RESTful API with Node, Flow, and Jest »

Comments

Featured Comment

Bryce C ↪ Hugo Enrique Virgen Herrera • 4 months ago

Hi Hugo, Michael - I was to get past this issue by simply removing the test file pattern in package.json eg to become:

"test": "mocha --reporter spec --compilers ts:ts-node/register"

I think mocha then just defaults to read files in the /test directory. I previously tried various changes to the pattern to no avail.

2 ^ | ˅ • Share ›

86 Comments Michael Herman 1 Login

Recommend 15 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Gytis Greitai • 5 months ago

Nice article. You can go one step further by using Typescript async/await

```
describe('GET api/v1/heroes/:id', () => {
  it('should return Luke Cage', async () => {
    const res = await chai.request(app).get('/api/v1/heroes/1')
    expect(res.body.hero.name).to.equal('Luke Cage');
  });
});
```

3 ^ | ˅ • Reply • Share ›

Maxime ROBERT • 8 months ago

Great article, thank you for that! TDD rocks and I hope I'll feel comfortable soon enough to program like that all the time :)
I've seen one interesting repo on Github https://github.com/pleerock... that might interest you. I haven't tried yet but seems promising!
Cheers

3 ^ | ˅ • Reply • Share ›

Kishore Relangi ➦ Maxime ROBERT • 8 months ago

I see examples for TDD approach missing in the routing-controllers.

^ | ˅ • Reply • Share ›

Alejandro Hidalgo • 6 months ago

Michael, Amazing example and so clear to follow. Thank you very much for publish!

2 ^ | ˅ • Reply • Share ›

Alexander Wong • 7 months ago

Some questions for your tests:
1. Why is mocha imported when it's not being used?
2. I tried forcing the test to fail by manually changing the expect value, however the test seemingly indicates that it passes. Is this an existing bug, or can you replicate this on your environment? See:

#Instead of res.type I used 'foobar'

it('should be text/html', () => {

chai.request(app).get('/')
.then(res => {
expect('foobar').to.eql('text/html');
});
});

# When running npm test:

baseRoute
✓ should be text/html
✓ should have a message prop

see more

2 ^ | ⌄ • Reply • Share ›

michaelherman Mod ➦ Alexander Wong • 7 months ago

We are using mocha. It's being use when we run "npm test".

1 ^ | ⌄ • Reply • Share ›

Timo Zöller ➦ Alexander Wong • 6 months ago

I had the same problem. It turns out you must return from the test callback.

Instead of
it('should be JSON', () => {
chai.request(app) ...

use
it('should be JSON', () => {
return chai.request(app)

^ | ⌄ • Reply • Share ›

michaelherman Mod ➦ Timo Zöller • 6 months ago

updated the post and code. thanks!

^ | ⌄ • Reply • Share ›

ArcaneWater • 8 months ago

Hey great article thanks! Would be great too if you can show same examples on Koa2 as a lot of programmers are jumping on Koa2.

3 ^ | ⌄ • Reply • Share ›

Pedro Vagner ➦ ArcaneWater • 7 months ago

I've see no advantages in using Koa2 layer.

^ | ⌄ • Reply • Share ›

ArcaneWater ➦ Pedro Vagner • 7 months ago

Koa is actively developing . Since most of devs from express are now at Koa

• Reply • Share ›

michaelherman Mod ArcaneWater • 8 months ago

+1 for Koa!! I have a few more Express articles in the works but after that I will be switching over to Koa.

• Reply • Share ›

Pinal Bhatt • 19 hours ago

Really nice article. But it would be really nice if we can even write gulpfile.js in Typescript. any way how can this be done?

1 • Reply • Share ›

michaelherman Mod Pinal Bhatt • 18 hours ago

Try https://medium.com/@pleeroc...

• Reply • Share ›

Seth Johnson • 10 days ago

This is great! Thanks!

1 • Reply • Share ›

Jack Harrison • a month ago

Great Job: I went through several attempts before finding this to be the closest to what I wanted as a base setup. This is still a relatively new stack approach and cheers to you for putting one together that provides a great starting point.

1 • Reply • Share ›

Kehinde Onadipe • 3 months ago

Great article, thanks!!!

1 • Reply • Share ›

Kevin Aung • 3 months ago

Thanks for the article! This is by far the cleanest node/express/ts example I've seen.

1 • Reply • Share ›

Nicolás Wernli • 6 months ago

Great article man! Congrats!

1 • Reply • Share ›

Dmitri Pavlutin • 7 months ago

Thank you! Very interesting and detailed.

1 • Reply • Share ›

Awesome! I had to change tsconfig target to es5, then worked fine.

1 ⌃ | ⌄ • Reply • Share ›

Andrei Rînea ➦ Filipe • 4 months ago

Strange, mine works in ES6, even tried in ES2017 and still works.

1 ⌃ | ⌄ • Reply • Share ›

Shai Badihi ➦ Filipe • 3 months ago

Hi,
This is because you've got:
D:\typescript-node-api\dist\index.js:1
(function (exports, require, module, __filename, __dirname) { import * as http from 'http';
^^^^^^
SyntaxError: Unexpected token import

?

⌃ | ⌄ • Reply • Share ›

Thomas Thai • 8 months ago

Thanks for the tutorial and moving toward writing them for Typescript.

1 ⌃ | ⌄ • Reply • Share ›

ole frank Jensen • 8 months ago

Great article, thanks a lot :)

1 ⌃ | ⌄ • Reply • Share ›

hieubui • 4 days ago

I use
"scripts": {
"start": "gulp watch scripts \n node dist/index.js"
}
to watch for a change and rebuilds + reruns if updating happened. Is it right?

⌃ | ⌄ • Reply • Share ›

Kheenan • 18 days ago

This is a great tutorial.
I used it before but now I'm having a problem that I never encountered before.

My chai tests always return 404 for all routes. The routes are accessible and fine otherwise. It just does not work within the test. it says 404 for all routes.

Does anybody know how to deal with this?

⌃ | ⌄ • Reply • Share ›

michaelherman Mod ↪ Kheenan • 17 days ago

Happy to help. Can you list out the things you've tried in terms of debugging? Also, make sure you are using the *exact* same versions for the dependencies as I am.

Reply • Share ›

Kheenan ↪ michaelherman • 17 days ago

Thanks for responding. I did roll back to the same dependencies with no success, I started a new project with the latest version of each dependency with no problems. It is clear that the problem is with that previous project. I was pretty early in the project so I just abandoned it. But I would hate to encounter the same problem later and not know how to deal with it. Rather than waste your time I'll just assume I did some nonsense somewhere. If somebody else complains then maybe we could revisit it.

Reply • Share ›

Renaldo van Dyk • a month ago

Hi, How do I call the actual API endpoint. I tried localhost:3000/api/v1/heroes/ but it responds with "Cannot GET ....." and the same for localhost:3000/api/v1/heroes/1 the only route that is working is localhost:3000 that responds with {"message":"Hello World!"}

Reply • Share ›

michaelherman Mod ↪ Renaldo van Dyk • a month ago

That's the correct endpoint. Try `http://localhost:3000/api/v1/heroes` (without the `/` at the end). If that's not working then there's something else happening on your end. Add `console.logs` to the route handler to debug. Perhaps the database is not set up right?

Reply • Share ›

Renaldo van Dyk ↪ michaelherman • a month ago

All the tests pass, doesn't that mean the database is set up right? I tried adding http and removed the / but no luck. I'll add some logs to try and debug. But what is baffling me is the fact that the tests pass.

Reply • Share ›

michaelherman Mod ↪ Renaldo van Dyk • a month ago

Yes, if the tests pass then it should work. I'm not sure what's happening on your end. I just tested it out and it works.

Reply • Share ›

Renaldo van Dyk ↪ michaelherman • a month ago

I found the issue, and it is quite silly. I forgot to compile my code so the routes never changed in the javascript dist files. I won't make that mistake again. One more thing that is bothering me is the data.json file only gets delivered to the dist folder once, if I change the name of the json file and compile my code it doesn't get reflected in the dist folder.

^ | v • Reply • Share ›

Renaldo van Dyk ➦ michaelherman • a month ago

Hi Michael, so the I tested it with your data and your routes and it works perfectly. If I change it up a bit to suit my use case it breaks as soon as I change the route in App.ts here is an example.
App.ts:

this.express.use('/api/v1/myroute', HeroRouter);

I replace all the heroes routes in the tests and HeroRouter stays the same. All the tests pass but when I call localhost:3000/api/v1/myroute in my browser I get the "Cannot Get ..." Error. Is there something I need to change in my HeroRouter to correspond to the new route name? I don't understand why this is happening.

^ | v • Reply • Share ›

Tobias Kiefer • 2 months ago

Nice Article :) One Question: Is it possible to get complex datatypes (like custom classes) from the body of the Rest Call?

^ | v • Reply • Share ›

michaelherman Mod ➦ Tobias Kiefer • 2 months ago

Yes. You may want to use an interface. Check out http://json2ts.com/

^ | v • Reply • Share ›

Tobias Kiefer ➦ michaelherman • 2 months ago

Yes that is clear, but my question is if I could pass such an object in the request body. At the moment you pass just an integer within the url, but what if i would want to add e.g a new hero and but in json format in the request body. Am I able to get this value?

^ | v • Reply • Share ›

michaelherman Mod ➦ Tobias Kiefer • 2 months ago

Use a POST request for this

^ | v • Reply • Share ›

Shai Badihi • 3 months ago

gr8 explanation. you've told to ignore import App from './App'; line but it doesn't really work for me ignoring it. btw all the import lines does not recognize.. I changed them to
//import * as http from 'http';
const http = require('http');
const debug = require('debug');
//import * as debug from 'debug';

//import App from './App';

const App = require('./App');

Please assist.

Reply • Share ›

michaelherman Mod → Shai Badihi • 3 months ago

What happens if you use `import` and then transcompile the code? Do you get an error?

Reply • Share ›

Hugo Enrique Virgen Herrera • 4 months ago

problems in "the API" step, then trying to execute test by typing "npm test" it says:

```
$ npm test

> tnesp@1.0.0 test C:\Users\hugo virgen\htdocs\tnesp
> mocha --reporter spec --compilers ts:ts-node/register 'test/**/*.test.ts'

Warning: Could not find any test files matching pattern: 'test/**/*.test.ts'
No test files found
npm ERR! Test failed. See above for more details.
```

This problem even happens cloning your example repo. please help!

Reply • Share ›

Bryce C → Hugo Enrique Virgen Herrera • 4 months ago

Hi Hugo, Michael I am faced with the same error, also on Windows 10 (node 7.2.1, npm3.10.10). The error occurs also testing against a clone of the sample's repository. Have you made any further progress? Thanks, Bryce.

Reply • Share ›

Hugo Enrique Virgen Herrera → Bryce C • 3 months ago

actually, i created a repo tah wordk fine with windows 10 (changes are in Development branch):

https://github.com/virgenhe...

2 • Reply • Share ›

Bryce C → Hugo Enrique Virgen Herrera • 3 months ago

Awesome thanks Hugo, I see the surrounding ' chars needed removal from the "test" path pattern (for Win10 only?).

1 • Reply • Share ›

Louk Man → Bryce C • 2 months ago

Thanks a lot for this solution! This solve the problem on Windows 10! :)

1 • Reply • Share ›

michaelherman Mod ➦ Hugo Enrique Virgen Herrera • 4 months ago

Not sure what's happening on your end, environment-wise. Are you sure the directory structure is correct? You may need to update the globs in the test script within package.json - 'test/**/*.test.ts'

• Reply • Share ›

Hugo Enrique Virgen Herrera ➦ michaelherman • 4 months ago

in addition, i ran `npm run test` and that command gives me a more descriptive error resume:

```
$ npm run test

> tnesp@1.0.0 test C:\Users\hugo virgen\htdocs\tnesp
> mocha --reporter spec --compilers ts:ts-node/register 'test/**/*.test.ts'

Warning: Could not find any test files matching pattern: 'test/**/*.test.ts'
No test files found

npm ERR! Windows_NT 10.0.10586
npm ERR! argv "C:\\Program Files\\nodejs\\node.exe" "C:\\Program
Files\\nodejs\\node_modules\\npm\\bin\\npm-cli.js" "run" "test"
npm ERR! node v6.10.1
npm ERR! npm v3.10.10
npm ERR! code ELIFECYCLE
npm ERR! tnesp@1.0.0 test: `mocha --reporter spec --compilers ts:ts-node/register
'test/**/*.test.ts'`
```

see more

• Reply • Share ›

Luciano ➦ Hugo Enrique Virgen Herrera • 2 days ago

to solve the error remove quotes between pattern test/**/*.test.ts

```
"scripts": {
"start": "node dist/index.js",
"test": "mocha --reporter spec --compilers ts:ts-node/register test/**/*.test.ts"
},
```

1 • Reply • Share ›

Hugo Enrique Virgen Herrera ➦ michaelherman • 4 months ago

could be sonthing related with my devEnv?
win10, git bash, node: 6.10.1 (also tested with 7 with same results), npm 3.10.10?

• Reply • Share ›

Load more comments