

# Exercise 1

*Create a simple JSON HTTP server*

## Prior Knowledge

Unix Command Line Shell

Some simple JavaScript (node.js)

## Learning Objectives

Understand the basics of Typescript and create a super simple Web Server

## Software Requirements

node, typescript, yarn

vscode for editing

Creating a node.js program in TypeScript.

1. Node.js is an effective framework for writing server-side programs using the JavaScript language. Typescript is a language that builds on JavaScript to add compile time type safety. In this exercise we are going to create a simple program that returns a random number between 1 and 100.

Because we expect the result to be read by a machine not a human, we will return this as a JSON not as an HTML.

2. Make a directory called ex1. You can do this by starting a terminal window and typing:  
`mkdir ~/ex1`  
`cd ~/ex1`



3. Now we need to create a new node.js project:

```
yarn init -y
```

```
yarn init v1.22.10
warning The yes flag has been set. This will
automatically answer yes to all questions, which may have
security implications.
success Saved package.json
Done in 0.03s.
```

This creates a simple JSON file that the yarn (or npm) uses to keep track of your project. You can look at package.json:

code package.json

```
home > oxsoa > ex1 > {} package.json > ...
1  {
2    "name": "ex1",
3    "version": "1.0.0",
4    "main": "index.js",
5    "license": "MIT"
6  }
7
```

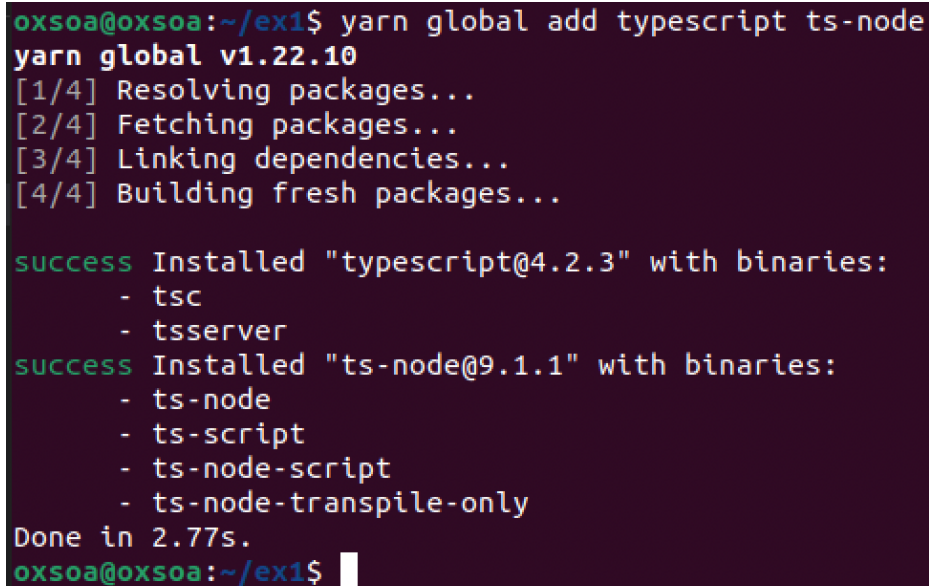
Don't change anything yet!

4. We need to install typescript and typescript support for node (ts-node). We will install them globally (adding the CLIs) and within the project as well.



5. Add typescript and ts-node globally:

```
yarn global add typescript ts-node
```



```
oxsoa@oxsoa:~/ex1$ yarn global add typescript ts-node
yarn global v1.22.10
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Installed "typescript@4.2.3" with binaries:
  - tsc
  - tsserver
success Installed "ts-node@9.1.1" with binaries:
  - ts-node
  - ts-script
  - ts-node-script
  - ts-node-transpile-only
Done in 2.77s.
oxsoa@oxsoa:~/ex1$
```

6. We also want to configure this as a typescript project:

```
tsc --init
```

```
message TS6071: Successfully created a tsconfig.json
file.
```

7. The tsconfig.json file is used to configure the compile options for Typescript. The default options it creates are a bit out of date, so lets edit that:

```
code tsconfig.json
```

Change line 7 to say:

```
"target": "es2017",
```

And change line 17 to specify where compiled JavaScript goes:

```
"outDir": "./dist",
```



Save the file.

8. We also have some dependencies we want in the local project:

```
yarn add express
yarn add --dev @types/express @types/node
```

Express is the Web framework we will use. There are cooler more modern ones such as koa and fastify available, but express is the most widely known and used.

9. It is usual in Typescript to have a src directory to store your code.

In the ex1 directory:

```
mkdir src
```

10. We will call our program “app.ts”. Let’s create an empty file:

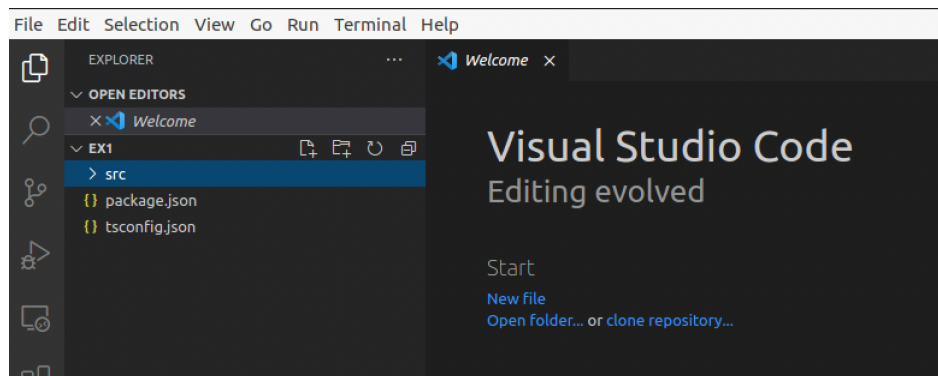
```
touch src/app.ts
```

11. Now let’s open up the current directory in Visual Studio code:

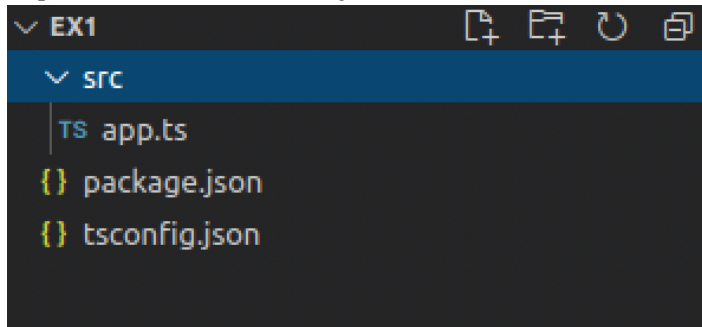
```
code .
```

The reason we want to open the project directory is this lets VSCode know about the tsconfig.json file, so the typescript plugins can work properly.

12. You should see the Visual Studio Code editor window.



13. Expand the src> directory:



14. Open app.ts

15. Type the following code:

The code is at <http://freo.me/ex1-app-ts>

*Please make sure you take time to understand the code. There are some pointers below.*

```
1  import express from "express";
2
3  const app = express();
4
5  // handle JSON body requests
6  app.use(express.json())
7
8  interface Random {
9    random: number;
10 }
11
12 async function get(req:express.Request, res: express.Response) :
13   Promise<express.Response> {
14   const r: Random = {
15     random: Math.floor((Math.random() * 100) + 1)
16   };
17
18   return res.status(200).json(r);
19 }
20
21 app.get("/", get);
22
23 const port = process.env.PORT || 8080;
24 app.listen(port, () =>
25   console.log(`Random server listening at http://localhost:\${port}`))
26 );
```



16. Firstly, notice that there are types

(e.g.

```
random: number
req: express.Request
r: Random
```

)

17. Secondly, unlike JS, we don't use var to declare variables. Instead there are two keywords:

```
let x: Type = ....; // indicates this is a true variable and will be
mutated
const x: Type = ....; // indicates this is never going to change
```

Typescript is pretty good at noticing if you use let when actually there is no mutation later.

18. This code creates an HTTP server (using Express) that responds to any HTTP GET request in the same way. It will instantiate a "Random" object containing a random number and then return that as a JSON string.

19. *Hint:* you can start a terminal window inside Visual Studio Code with Ctrl+`

20. We can compile this code:

```
cd ~/ex1
tsc
```

21. It should have all worked! Check dist/ and you should see a "compiled" app.js file.

22. Just to see types in action, change line 15 of the src/app.js file to read:  
radom: Math.floor((Math.random() \* 100) + 1)



You should see vscode highlight the error, as well as if you try to compile seeing:

```
tsc
src/app.ts:17:9 - error TS2322: Type '{ radom: number; }'
is not assignable to type 'Random'.
  Object literal may only specify known properties, but
  'radom' does not exist in type 'Random'. Did you mean to
  write 'random'?

17         radom: Math.floor((Math.random() * 100) + 1)
      ~~~~~
```

Found 1 error.

23. Change it back again!

24. We could run this code by using the compiled .js file, but then it will be hard to track down the errors because we will see line numbers based on the compiled js, not our original ts. So we can run it using “ts-node” which compiles and runs the typescript directly.

25. `cd ~/ex1`  
`ts-node src/app.ts`

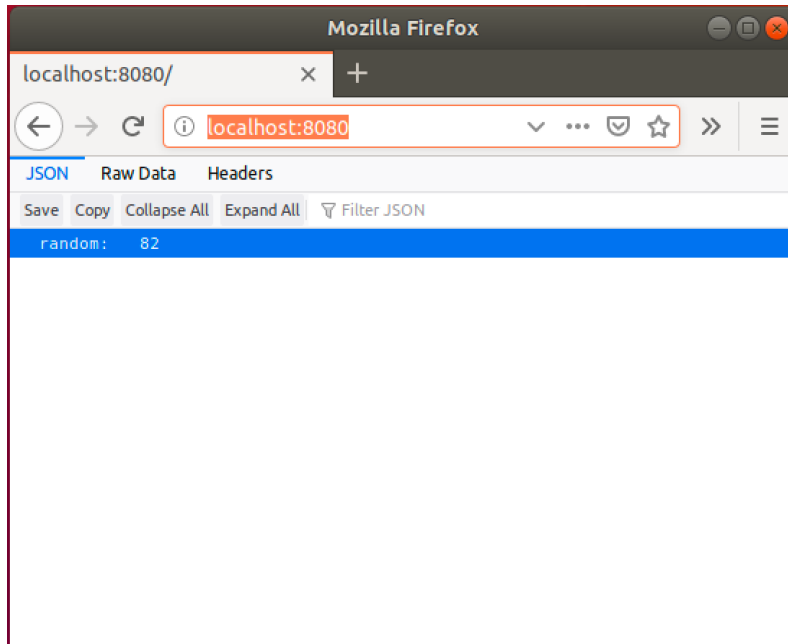
You should see the server respond:

Random server listening at `http://localhost:8080`

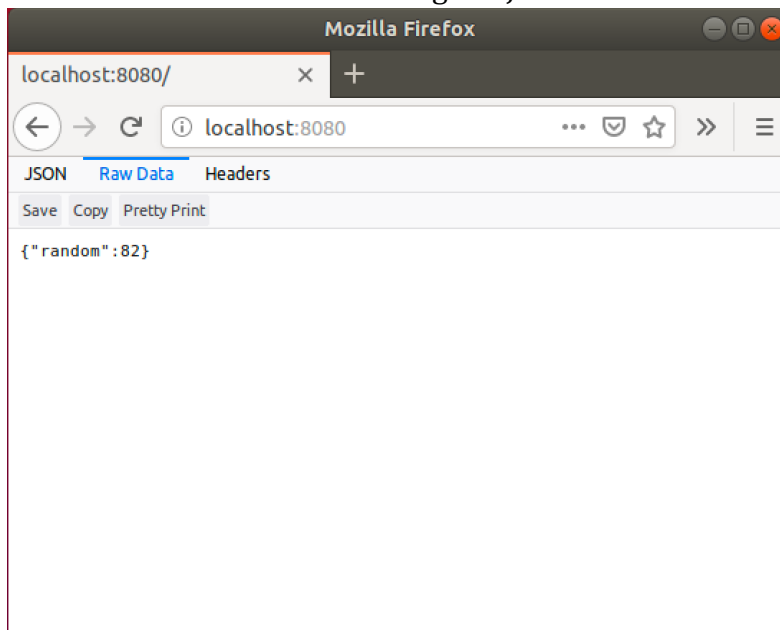
26. You can test this code by pulling up a browser window (e.g. Firefox) and then browsing to <http://localhost:8080>



The result is this:



Click on Raw Data to see the original JSON.



27. However, we do not want a human-/browser-enabled service. We want to call this service from machine-based clients. Let's first try curl (a command-line URL / HTTP tool).





28. **Start a new terminal window** (hint Right-click on the icon) and then type:  
`curl http://localhost:8080`

You should see:

```
curl http://localhost:8080
{"random":71}oxsoa@oxsoa:~/ex1$
```

Hint: Because the HTTP response has no '\n' line ending, the result is a bit hard to read as the next line merges with the output.

29. curl provides a useful debug facility. If you turn on verbose output, you can see the actual network messages as they are sent on the wire:

```
curl -v http://localhost:8080
```

30. You should see output similar to this:

```
* Rebuilt URL to: http://localhost:8080/
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Tue, 24 May 2016 09:04:03 GMT
< Connection: keep-alive
< Content-Length: 13
<
* Connection #0 to host localhost left intact
{"random":33}oxsoa@oxsoa:~/ex1$
```

The lines beginning with > indicate that these are sent to the server and < are received from the service.



31. Now try:

http <http://localhost:8080>

```
oxsoa@oxsoa:~/ex1$ http http://localhost:8080
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 13
Content-Type: application/json; charset=utf-8
Date: Mon, 18 Nov 2019 08:44:59 GMT
ETag: W/"d-Wmm8AE0EUJ7/UdlvLcpb9QqIPYA"
X-Powered-By: Express

{
  "random": 87
}
```

There is also a verbose mode of HTTPie which shows the sent request as well:

```
oxsoa@oxsoa:~/ex1$ http -v http://localhost:8080
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/0.9.8

HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 13
Content-Type: application/json; charset=utf-8
Date: Mon, 18 Nov 2019 08:48:24 GMT
ETag: W/"d-3sCaP8S62kD1XUTpBuzwQ/Q1oZU"
X-Powered-By: Express

{
  "random": 76
}
```

32. We can also run the server in verbose mode. Stop the ts-node.

```
export DEBUG="*:*"
ts-node src/app.ts
```

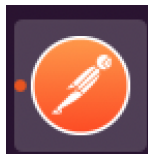


```
oxsoa@oxsoa:~/ex1$ ts-node src/app.ts
express:application set "x-powered-by" to true +0ms
express:application set "etag" to 'weak' +2ms
express:application set "etag fn" to [Function: generateETag] +1ms
express:application set "env" to 'development' +0ms
express:application set "query parser" to 'extended' +0ms
express:application set "query parser fn" to [Function: parseExtendedQueryString] +0ms
express:application set "subdomain offset" to 2 +0ms
express:application set "trust proxy" to false +1ms
express:application set "trust proxy fn" to [Function: trustNone] +0ms
express:application booting in development mode +0ms
express:application set "view" to [Function: View] +0ms
express:application set "views" to '/home/oxsoa/ex1/views' +0ms
express:application set "jsonp callback name" to 'callback' +0ms
express:router use '/' query +1ms
express:router:layer new '/' +0ms
express:router use '/' expressInit +0ms
express:router:layer new '/' +0ms
express:router use '/' jsonParser +1ms
express:router:layer new '/' +0ms
express:router:route new '/' +0ms
express:router:layer new '/' +0ms
express:router:route get '/' +0ms
express:router:layer new '/' +0ms
Random server listening at http://localhost:8080
```

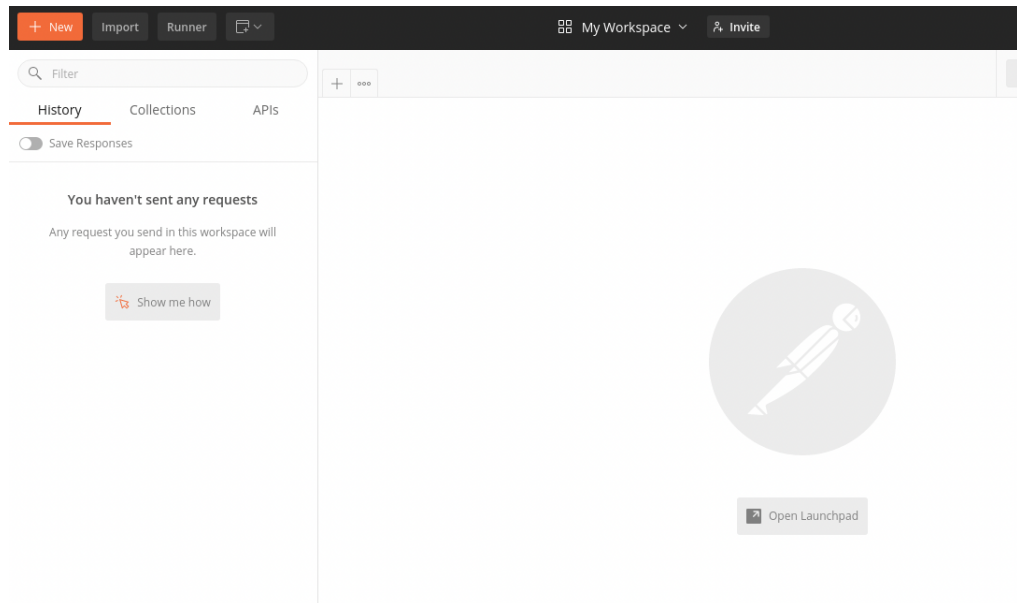
33. Send a request and see the log:

```
express:router dispatching GET / +6s
express:router query : / +0ms
express:router expressInit : / +0ms
express:router jsonParser : / +0ms
body-parser:json skip empty body +0ms
```

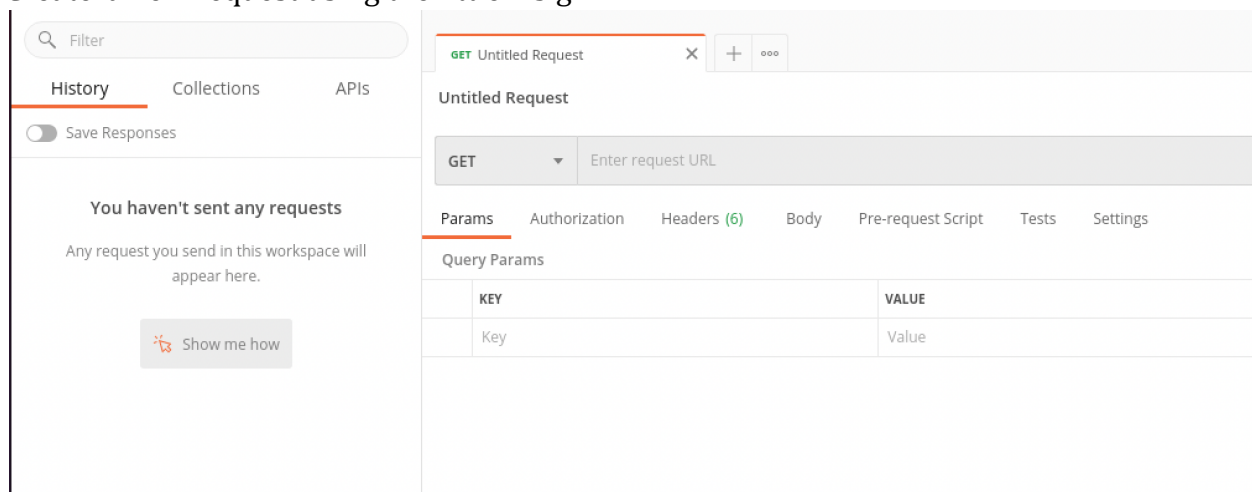
34. Another way of testing this is to use a tool called Postman. Click on the Postman icon:



35. You should see a window like this. There may be some old history in there.



### 36. Create a new request using the little + sign



### 37. Type <http://localhost:8080> into the Request URL field. Choose GET and then click Send. You should see:



The screenshot displays the Postman application window. The top menu bar includes File, Edit, View, and Help. Below the menu is a toolbar with buttons for New, Import, Runner, and a dropdown menu. The main workspace is divided into several sections. On the left, there is a sidebar with a search filter, tabs for History, Collections, and APIs, and a 'Save Responses' toggle. The History tab is active, showing a list of recent requests. The main area is titled 'Untitled Request' and shows a GET request to 'http://localhost:8080'. The 'Params' tab is selected, showing a table for Query Params. The 'Body' tab is also visible, showing a JSON response. The status bar at the bottom indicates 'Status: 200 OK', 'Time: 11 ms', and 'Size: 224 B'. The response body is displayed in the 'Body' tab, showing a JSON object with a 'random' property.

Postman

File Edit View Help

+ New Import Runner

My Workspace Invite

No Environment

Filter

History Collections APIs

Save Responses Clear all

Today

- GET http://localhost:8080
- GET http://localhost:8080
- GET http://localhost:8080

Untitled Request

GET http://localhost:8080

Send Save

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (6) Test Results

Status: 200 OK Time: 11 ms Size: 224 B Save Response

Pretty Raw Preview Visualize JSON

```
1
2  "random": 13
3
```

Find and Replace Console

Bootcamp



### 38. Automated testing of the service

We want this service to meet a set of behavior requirements. To ensure this, we can use a set of tests. There are a number of testing frameworks for SOA services. For this example, we are going to the capabilities of Postman, which includes a nice testing capability.

I have written a test script for this service. It is in JSON format and can be imported into Postman

It is available here: <http://freo.me/ex1-postman>

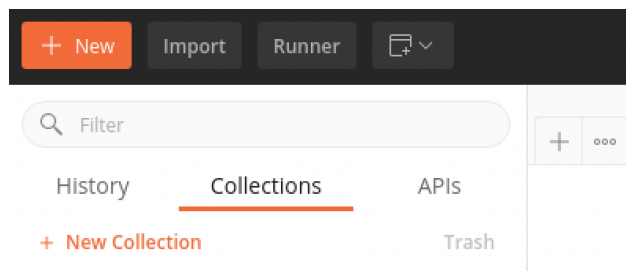
### 39. Start a new terminal window to keep the server running (or use the one you started for curl/http commands.

You can download the Postman collection onto your VM using the following command

```
cd ~/ex1
```

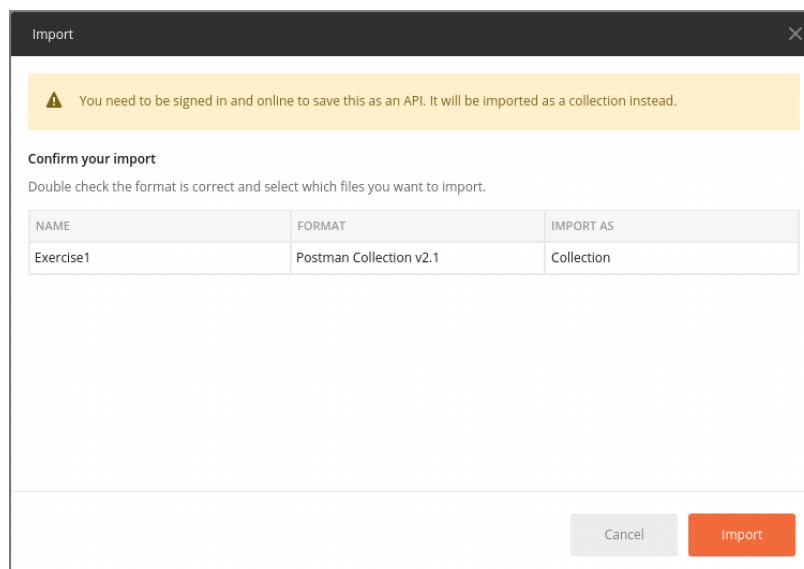
```
curl -L http://freo.me/ex1-postman -o ex1-postman.json
```

### 40. To Import the collection into Postman, first click on Import:



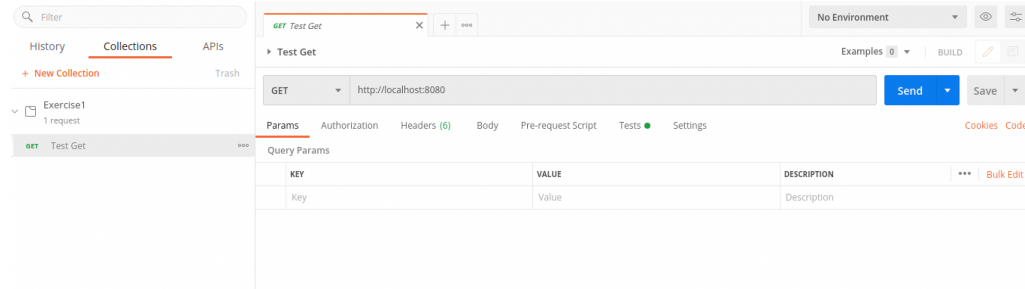
### 41. Select the json file you have downloaded

### 42. You should see:



#### 43. Click **Import**

#### 44. Take a look at the Collection. Expand it and click on the request:



#### 45. Take a look at the test:

Params Authorization Headers (6) Body Pre-request Script **Tests** Settings

```
1 pm.test("RandomService GET test", function() {  
2   pm.response.to.have.status(201);  
3   pm.response.to.be.json;  
4   pm.response.to.be.withBody;  
5   pm.response.to.have.header("Content-Type");  
6   pm.expect(pm.response.headers.get('Content-Type')).to.include('application/json');  
7   const responseJson = pm.response.json();  
8   pm.expect(responseJson.random).to.greaterThan(1).lessThan(100);  
9  
10 });
```

The test does an HTTP GET on the URL and then validates the following aspects:

- The return code is 201
- The Content-Type header is "application/json"
- The JSON type of the result is a tag random, with type number
- The JSON contains a tag called random, with a value >1 and <100

#### 46. The test runs automatically when you execute the request, or you can run all tests in a collection.

#### 47. Run the test.

#### 48. Does the result match your expectations?



49. You can also run the tests using a CLI called newman. Check its installed:

```
yarn global add newman
```

```
yarn global v1.22.10
[1/4] Resolving packages...
warning newman > postman-request > har-validator@5.1.5:
this library is no longer supported
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...
success Installed "newman@5.2.2" with binaries:
  - newman
Done in 3.92s.
```





50. Now try it out:

```
newman run ex1-postman.json
```

51. You will see:

```
oxsoa@oxsoa:~/ex1$ newman run ex1-postman.json
newman

Exercise1

→ Test Get
GET http://localhost:8080 [200 OK, 224B, 80ms]
1. RandomService GET test
```

	executed	failed
iterations	1	0
requests	1	0
test-scripts	1	0
prerequisite-scripts	0	0
assertions	1	1

```
total run duration: 161ms
total data received: 13B (approx)
average response time: 80ms [min: 80ms, max: 80ms, s.d.: 0µs]
```

```
# failure      detail
1. AssertionErr... RandomService GET test
    expected response to have status code 201 but got 200
    at assertion:0 in test-script
    inside "Test Get"
```



52. Before we fix the test, let's get a nicer development experience.

There is a tool called nodemon that we can use to rebuild our code and rerun the tests automatically.

```
yarn global add nodemon
```

53. Now let's add some scripts to our package.json. This snippet is here:

<http://freo.me/ex1-scripts>

```
"scripts": {
  "start": "ts-node src/app.ts",
  "dev": "nodemon --watch . --ext ts --exec \"ts-node src/app.ts\"",
  "test": "newman run ex1-postman.json",
  "test-dev": "nodemon --watch . --ext ts --exec \"sleep 5 && newman run ex1-postman.json\"",
  "build": "tsc",
  "watch": "tsc --watch"
},
```

54. Your package.json should look like (this is a subset!):

```
{
  "name": "ex1",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "scripts": {
    "start": "ts-node src/app.ts",
    "dev": "nodemon --watch . --ext ts --exec \"ts-node src/app.ts\"",
    "test": "newman run ex1-postman.json",
    "test-dev": "nodemon --watch . --ext ts --exec \"sleep 5 && newman run ex1-postman.json\"",
    "build": "tsc",
    "watch": "tsc --watch"
  },
  "devDependencies": {
    "@types/body-parser": "^1.19.0",
    "@types/express": "^4.17.11",
    "@types/node": "^14.14.37"
  }
},
```

55. Now we can automatically run the updated code:

```
yarn run dev
```

56. And in another window, run the tests:

```
yarn run test-dev
```

57. Let's fix the server so that it passes the test, or the test to match the server.

Note: if **you changed the test** to fit the server, then you need to re-export



the JSON!

I'll leave this up to you. Why did you choose to change what you did?

58. Is there anything else wrong with the test spec?

59. Once the tests are passing, this exercise is complete.

*Recap:*

We have created a simple http server that returns a JSON output. We have tested this service in a number of ways – including via browser, Postman, curl and through a proper automated test.

In our next exercise we will create a client for this service.

