



TASK

Express - Web Applications

Visit our website

Introduction

WELCOME TO THE EXPRESS - WEB APPLICATIONS TASK!

Now that you're familiar with the basics of using Node.js, you can start building server-side applications more efficiently using Express. Express is a lightweight web framework. By the end of this task, you will be able to use Express to create a back-end web application that handles routing and can respond to HTTP requests with both static and dynamic content. As a full-stack web developer, you will also need to be able to create custom RESTful web APIs that can be used to store and manipulate data on the back-end of your full-stack application – Express will empower you to do so.

WHAT IS EXPRESS.JS?

Express is the most popular Node web framework. It gives you access to a library of code (Node.js functions), making it easier for you to create web servers with Node. It is also the underlying library for several other popular Node web frameworks. Express is minimal and flexible, providing a robust set of features for web and mobile applications. It is open-source and maintained by the Node.js foundation.

Although Express is a minimalist framework, developers have created compatible middleware packages to address almost any web development problem. You can find a list of middleware packages maintained by the Express team at [Express Middleware](#).

According to Express, “Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle.”

Unlike other frameworks, like Django, Express is an [unopinionated](#) web framework. Unopinionated frameworks have fewer restrictions than opinionated frameworks regarding the best way to glue components together to achieve a goal, or even what components should be used. These frameworks allow you to use the best tools to complete a particular task, but you need to find these tools yourself. Express is very flexible and pluggable and has no “best way” of doing something. You can use almost any compatible middleware in almost any order and you can structure your app as you like.

WHY USE EXPRESS?

You use Express for the same reason you would use any other web framework: to speed up development and decrease the amount of boilerplate code you have to write. We have been using Node to create a web server. There are many aspects of creating a web server for your web application that will be very similar, regardless of the app you are building. We could write all the code we need without Express, but we will use Express to speed up development.

INSTALL EXPRESS

In the previous tasks, you have installed Node.js and learned about NPM. You will now install Express using NPM.

NPM is an extremely important tool for working with Node applications. It is used to fetch any packages (JavaScript libraries) that an application needs for development, testing, and/or production. It can also be used to run tests and Node modules used in the development process. Express is another package you need to install using NPM.

1. Open your terminal or command prompt.
2. Create a directory using the **mkdir** command and then navigate into it using the **cd** command.

```
mkdir myapp  
cd myapp
```
3. Use the **npm init** command to create a package.json file for your application. We manage dependencies using a plain-text definition file named **package.json**. All the dependencies for a specific JavaScript package are listed in this file. The package.json file should contain everything NPM needs to fetch and run your application. The command **npm init** will prompt you to enter several details, such as the package name, version, description, entry point, test command, etc. Please note: **npm init** creates the initial **package.json** file. Enter it now:

```
npm init
```
4. To display the package.json file, enter **cat package.json** or **type package.json** (depending on the operating system you are using) into your terminal. You can also view the package.json file in Sublime or Visual Studio Code.

5. We will now install the Express library in the 'myapp' directory that you created, and save it in the dependencies list of the package.json file. Do this by entering the following command into the terminal or command prompt:
npm install express
6. Enter **cat package.json** or **type package.json** into your terminal again. The dependencies section of your package.json will now appear in the **package.json** file and will contain Express.

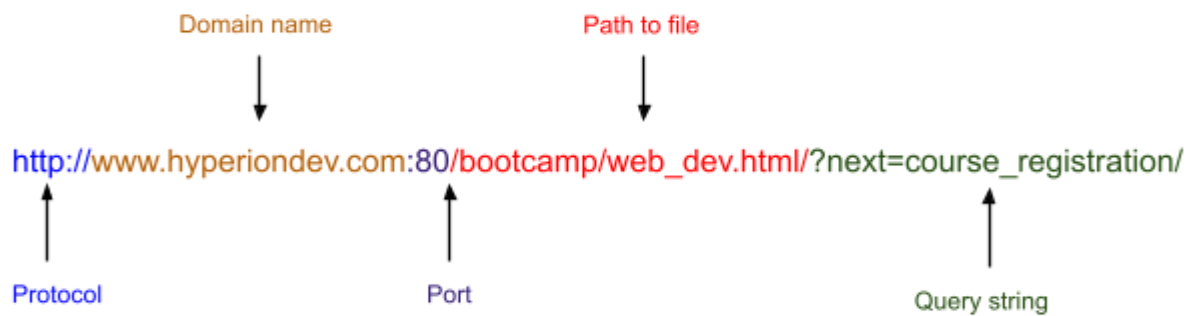
```
cat package.json
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.15.4"
  }
}
```

ROUTING

As we already have Express installed, we can start creating the code for the back-end of our web application. One of the most critical aspects of server-side logic is routing.

One of the most important tasks of a server is to perform routing. Routing refers to determining how an application responds to a client's request to a particular endpoint. The request is made using a URI (or path) and a specific HTTP request method.

Remember that a URL contains a lot of information:



1. It identifies the protocol being used to send information. In the example above, the protocol being used is HTTP.
2. It identifies the domain name of the web server on which the resource can be found, e.g. `www.hyperiondev.com`.
3. It identifies the port on the server. In this example, the port number is given as port 80. In reality, if the default HTTP ports are used (port 80 is the default for HTTP, port 443 for HTTPS) they don't have to be given in the URL.
4. It gives the path to the resource on the web server, e.g. `/bootcamp/web_dev.html`
5. Data can be passed using parameters or using a query string (as shown in the image above).

To perform routing, we are interested in the *path* section of the URL. Routing involves identifying the path that was requested and providing a response based on that path.

With Express there are a few route methods used to perform routing: `get`, `post`, `put`, and `delete`. In this task, we will cover the primary routing methods. The `get` method responds to HTTP `get` requests. An HTTP `get` request is used to request a specific resource.

See a code example that uses the `get` method for routing:

```
app.get('/', function (req, res) {  
  res.send('Hello World!')  
})
```

The `app.get()` method shown above takes two arguments:

- **The path.** In this example, the path is `'/'`, the root route of our app. In development, the URL <http://localhost:8000/> (where the server is running on port 8000) will match the route specified in the `app.get('/', ...)`

method in our code example. If you wanted to add a route handler that would handle the HTTP request using the URL,

<http://localhost:8000/about>, you would have to add an `app.get('/about', ...)` function.

- **A callback function.** The callback function that is passed as the second argument to the `app.get()` method acts as a *route handler*. In other words, in this example, when a get request is made to the homepage of the web app, a response object that simply contains the text “Hello World!” will be sent from my server to the browser.

Now that we understand the basics of what routing is, let's create a server with Express!

SPOT CHECK 1

Let's see what you can remember from this section.

1. What is the main reason for using Express?
2. What is routing?

CREATE A SERVER USING EXPRESS

In the root directory of your 'myapp' directory (the directory you created when you installed Express), create a file called **app.js** and copy the code below into it:

```
const express = require('express');
const app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(8000, function () {
  console.log('Example app listening on port 8000!');
});
```

The code above is a simple “Hello World” Express web application. Let's analyse this code line by line:

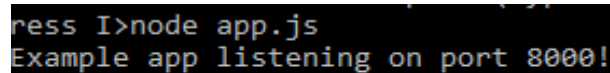
- **Line 1:** `require()` is called to import the 'express' module.
- **Line 2:** Create an object called **app** by calling the top-level `express()` function. This object represents our Express application. The app object contains important methods that we use to create our server. Notice two of

the methods in the code above: **get()** and **listen()**. You will also learn about **app.use()** in this task.

- **Line 4:** Create a route handler that will respond to requests using the **app.get()** routing method.
- **Line 8:** The **app** object also includes the **listen()** method. The **listen()** method specifies what port our **app** object (application server) will listen to HTTP requests on. The **app.listen()** method returns an **http.Server** object.

To start the server, call **node** with the script in your terminal or command prompt (remember to make sure that you navigate to the **myapp** folder first).

Type **node app.js** in the terminal as per the example below.



```
ress I>node app.js
Example app listening on port 8000!
```

Now use your web browser to navigate to <http://127.0.0.1:8000/>. You should see the string “Hello World!” displayed in your browser!

SERVING STATIC FILES WITH EXPRESS

Even dynamic web applications may have certain static components to display. With Express, it is easy to serve static resources by using the **express.static** built-in middleware function. Remember that a middleware function is a function that has access to the request and response objects.

To allow your app to serve static files, simply do the following:

1. Create a folder in your project directory to store the static files you want to serve. This folder is usually called “public”.
2. Add to this folder all the static resources you want your app to make available (images, HTML files, etc.). Make sure that these files have meaningful names because, by default, you will use the file name of the resource to access it. Don't store any files that you don't want users of your app to see in this directory!!
3. Add the following code to your **app.js** file:

```
app.use(express.static('public'));
```

We use **app.use()** to include any built-in middleware functions we need in our app. For a list of other built-in middleware functions for Express, see [here](#).

Once you start your server (type `node app.js` in the terminal), you should be able to access these static resources from the browser. For example, if you added an HTML file called “example.html” to the public folder, you could access it with this URL: <http://localhost:8000/example.html> (where you have created a server that listens for HTTP requests on port 8000).

ENVIRONMENT VARIABLES

When creating back-end apps, it is important to be able to access *environment variables*. So far, the only variables we have used have been variables we have created using the keywords `const` or `let`. However, the back-end code needs to be able to access variables created and set in the environment in which the code is run (i.e. variables set by the server). Node.js allows us to access these variables using `process.env`. To see some of the environment variables stored on your PC, try adding the following line of code to your `app.js` file and running it in the terminal.

```
console.log('The value of process.env is:', process.env);
```

When your code is being executed on a web server (after deployment), an important environment variable that will be set on the server is the port number on which your application server will listen for HTTP requests. To get the port number from the environment variables instead of hardcoding it, we use the following code:

```
const PORT = process.env.PORT || 8000;
app.listen(PORT, () => {
  console.log(`Server is listening on port ${PORT}`);
});
```

You should also use `process.env` for other values you don't want to hardcode and don't want in your codebase (like API keys).



Take note:

Notice how we use a string literal and an arrow function to write code more efficiently in the code example above. Remember, these are introduced as improvements to JavaScript with ES6.

Also, notice that the variable **PORT** is entirely in uppercase capital letters. This is a common naming convention

recommended by style guides for constant variables that can't change once a value has been assigned. See more naming conventions recommended by Google for JavaScript [here](#).

NODEMON

It can be time-consuming to restart your server every time you make changes to your code! You can use Nodemon to enable server restarts on file changes. After typing the command shown below into the command line interface, you should see that your package.json file now includes a reference to Nodemon in its devDependencies section.

```
npm install --save-dev nodemon
```

To run an app using Nodemon, type **nodemon name_of_file** in the terminal instead of **node name_of_file**.

ADD A START SCRIPT

It is recommended that you add a script to your **package.json** file for every application that you create that specifies how to start your app. This is done as shown in the image below:



```
"main": "app.js",  
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "start": "nodemon app.js"  
},
```

To start the application, we then type **npm start**. This will use the instruction specified in the start script in the package.json file to run the code.

SPOT CHECK 2

Let's see what you can remember from this section.

1. What method do you use to import the 'express' module?
2. What method do you use to create a route handler that will respond to requests?
3. What does Nodemon do?



A note from the HyperionDev Team

There's a reason that true full-stack web developers are sometimes referred to as unicorns. It's the rarity factor. A genuine full-stack web developer has such a diverse range of skills that they are hard to find. Let's look in more detail at what these developers do and why it's so hard to find full-stack web developers [here](#).

MORE ON RESTFUL APIS

Let's look at the other end of the API: the back-end.

According to [IBM](#), RESTful web services are based on the following principles:

- RESTful web services expose resources using URIs.
- Resources are manipulated using PUT, GET, POST, and DELETE operations.
 - PUT creates a new resource
 - DELETE deletes a resource.
 - GET retrieves the current state of a resource.
 - POST transfers a new state onto a resource.
- Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.
- Interaction with resources is stateless. State information is exchanged using techniques like URI rewriting, cookies, hidden form fields, and embedding state information in response messages.

CREATE A CUSTOM RESTFUL API USING EXPRESS

A RESTful web API is code that is written to respond to HTML PUT, GET, POST, and DELETE requests. To create a RESTful API, we are going to write JavaScript functions using Express and Node to handle each of these requests.

Express has built-in middleware routing functions to handle each of these HTTP request methods. Earlier in this task, we already used the **app.get()** function to respond to HTTP GET requests with the specified URL path ('/'). The app object contains methods to handle each of the HTTP verbs: **app.post()**, **app.get()**, **app.put()**, and **app.delete()**.

Like the **app.get()** method, each of these methods takes two arguments:

- The **route**. These methods are used to perform routing. Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- A **callback** function. The callback function that is passed as the second argument to the **app.get/post/put/delete()** methods acts as a route handler.

Each route handler that we write will be used to either **create** data (e.g. create a JSON file), **read** data, **update** data, or **delete** data. These operations are often referred to as CRUD (Create, Read, Update, and Delete) operations.

Each CRUD operation can be accessed using a corresponding HTTP request as shown in the table below:

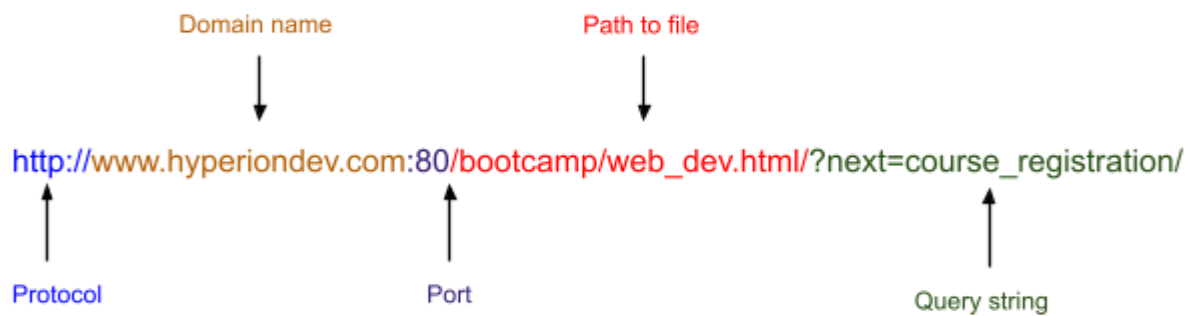
HTTP verb	CRUD operation	Express method	Description
Post	Create	app.post()	Used to submit some data about a specific entity to the server.
Get	Read	app.get()	Used to get a specific resource from the server.
Put	Update	app.put()	Used to update a piece of data about a specific object on the server.
Delete	Delete	app.delete()	Used to delete a specific object.

You create an API that receives a URI with an HTTP request and use the appropriate Express routing middleware to call the corresponding functions that handle the CRUD operations.

If we are going to be able to add and update data on our servers though, we need a way to be able to pass our data from the browser to the server.

PASSING DATA THROUGH TO THE SERVER USING THE REQUEST OBJECT

An important role of the server is to receive necessary data that is passed through from the browser. This data can be passed from the client to the server using the URL. Let's review the makeup of a URL:



Data can be passed using **parameters** (as shown in the image below) or using a **query string** (also shown in the image above).

With query strings, data are passed as key-value pairs (`?key=value&key2=value2`), e.g. `?next=course_registration` as shown above.



The image directly above illustrates what a URL may look like if parameters are added to the URL. In the example above, the parameter '2315' may represent the id of a student at HyperionDev.

To access the data passed through using the URL, we use the **req** object that is passed through as an argument to the **app.post** or **app.put** route handler. The **req.params** command is used to get parameters from a URL, and **req.query** is used to get data from a query string.

In the example below, see how the code **req.query.name** is used to get the value of **the key-value pair** where the key is 'name'.

```
app.post('/', (req, res) => {
  fileHandler.writeFile('person.json', `{name: ${req.query.name}}`, (err) => {
    if (err) throw err;
    res.send('File created!');
  });
});
```

URL: localhost:3000?name=Gareth

See in the example below how the code `req.params.name` is used to get the value of **the parameter 'name'** that is defined in the route argument of the `app.put()` method.

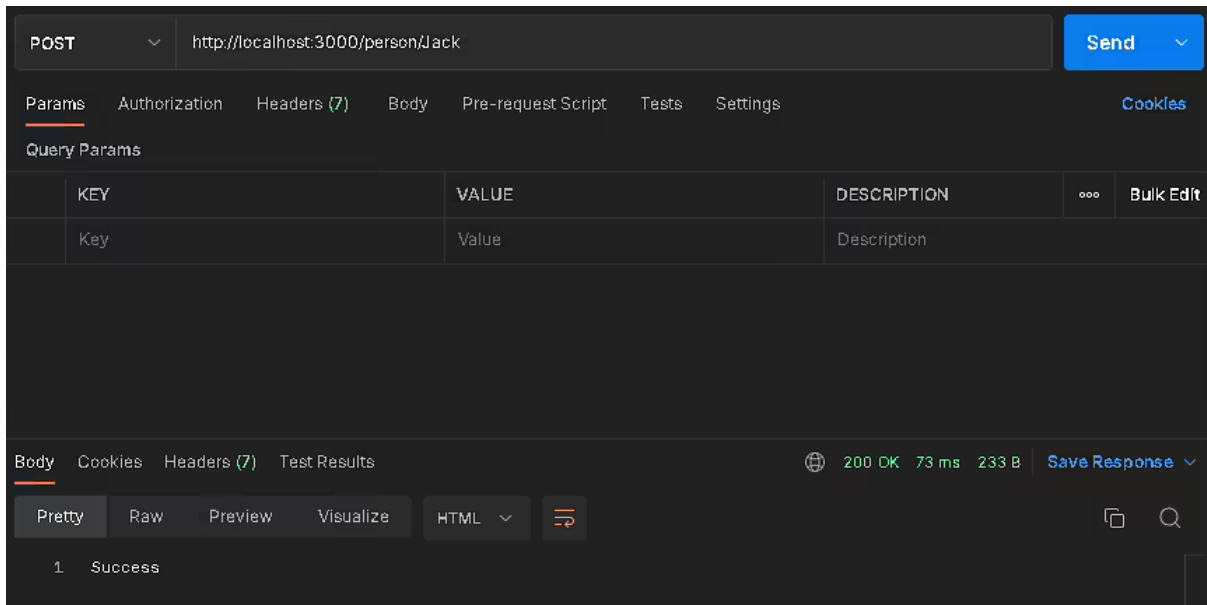
```
app.put('/:name', (req, res) => {
  fileHandler.writeFile('person.json', `{name: ${req.params.name}}`, (err) => {
    if (err) throw err;
    res.send('File updated!');
  });
});
```

URL: localhost:3000/Sue

Have a look at the code example that accompanies this task to see this code in action. Remember that you should be using JSON data (**.json**) when creating an API. JSON data can be defined by double quotes around both the key and the value.

TEST YOUR API USING POSTMAN

You will later learn how to use your front-end to pass data through to the server but, for now, we can test that our server is sending and receiving data using Postman. Postman is a free API development environment. You can see how Postman works and download it [here](#). The next image shows the results you could expect from using Postman to send a post request with the parameters shown in the image to the server that we configured with the code above.



Some IDEs, like [IntelliJ](#) for instance, also have built-in functionality for testing APIs.

SPOT CHECK 3

Let's see what you can remember from this section.

Match up the following:

1. Post	a. Used to update a piece of data about a specific object on the server.
2. Get	b. Used to delete a specific object.
3. Put	c. Used to submit some data about a specific entity to the server.
4. Delete	d. Used to get a specific resource from the server.

Instructions

- The Express related tasks involve you creating apps that need some modules to run. These modules are located in a folder called 'node_modules', which is created when you run the following command from your command line: **npm install** or similar. Please note that this folder typically contains thousands of files which, if you're working directly from Dropbox, has the potential to *slow down Dropbox sync and possibly your computer*. As a result, please follow this process when creating/running such apps:
 - Create the app on your local machine (outside of Dropbox) by following the instructions in the compulsory task.
 - When you're ready to have a reviewer review the app, please delete the node_modules folder.
 - Compress the folder and upload it to Dropbox.
 - Your reviewer will, in turn, decompress the folder, install the necessary modules and run the app from their local machine.
- Read through the example files that accompany this task before attempting the lab. To execute the code in the example files:
 - Copy the example folder to your local computer.
 - Navigate to the folder that contains the example files from your command line interface.
 - Use the command line interface to type **npm install** to install all the needed dependencies (including Express). This command will use the **package.json** file in the example folder to see which dependencies to install.
 - Use the command line interface to type **npm start** to start this application.
- Remember that the primary goal of this task is to get to grips with writing code for the **back-end** of your web application. Therefore, even though your app will display output in the browser, your main concern should be the server-side functionality and not the appearance of the front-end.

Compulsory Task 1

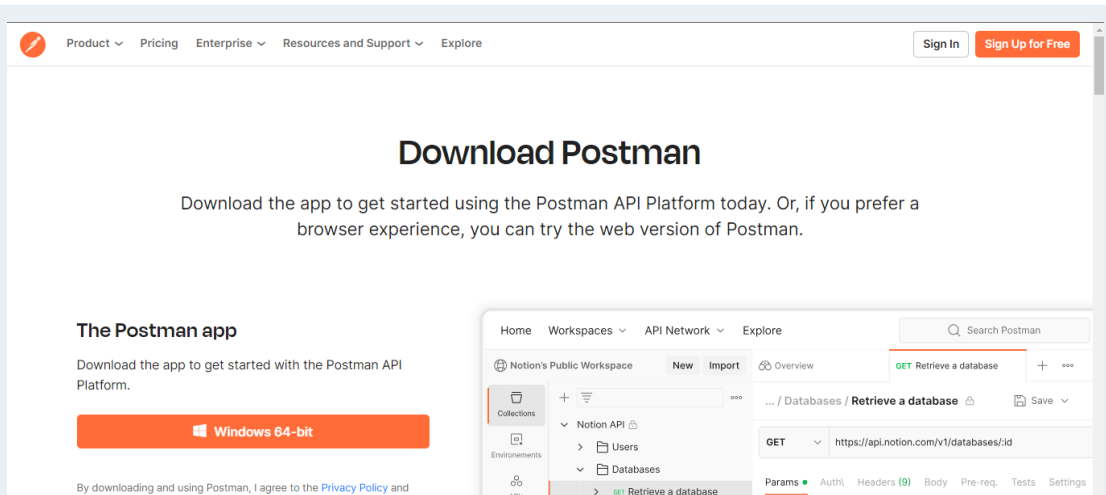
Follow these steps:

- Create an application project folder called **my_first_express_app**.
- Create another subdirectory called **public** that contains two static html files called **about.html** and **contact_us.html**. Feel free to reuse any html files you have created before.
- Create a file called **person.json** that describes a person. E.g. of json: {"name": "Tom Jones", "email": "tom@gmail.com", "gender": "male"}
- Create a server that will do the following:
 - Display "Welcome" and the name of the person that is read from the file **person.json** at URL `http://localhost:3000/`.
 - Display the static HTML page, **about.html** at URL `http://localhost:3000/about.html`
 - Display the static HTML page, **contact_us.html** at URL `http://localhost:3000/contact_us.html`
 - Display the message "Sorry! Can't find that resource. Please check your URL" if the user enters an unknown path. Help [here](#).
- Enable the server to restart on file changes.
- You should be able to start the server using `npm start`.

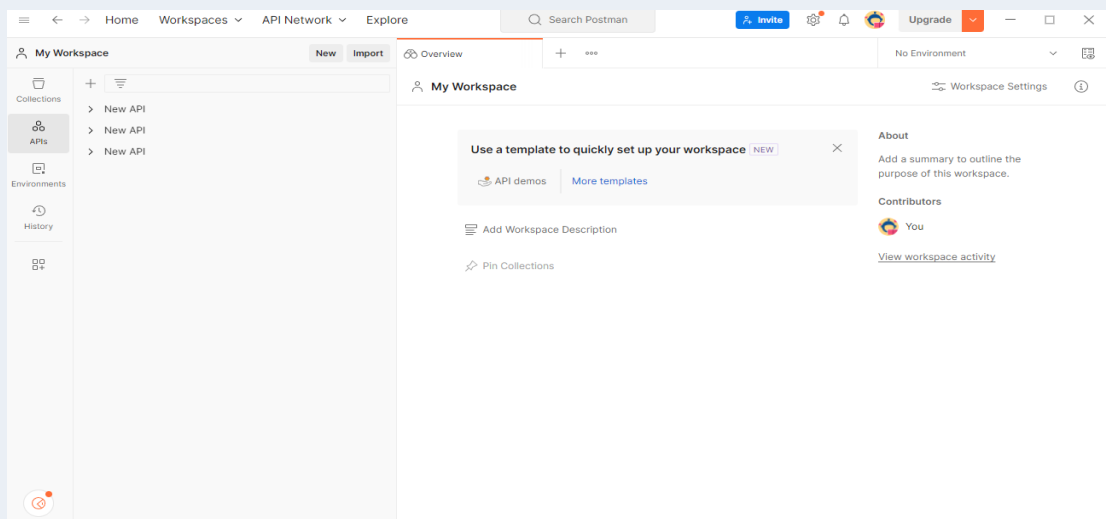
Compulsory Task 2

Follow these steps:

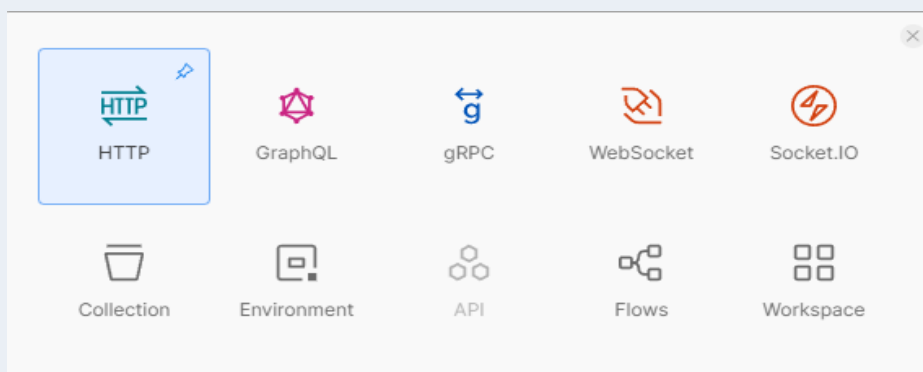
- Download Postman [here](#). Postman is a powerful tool to understand API calls and work with HTTP requests and responses. To demonstrate your understanding regarding HTTP requests and responses, follow the steps below:
- **Step 1:**
Download and install Postman on your computer:
https://www.postman.com/downloads/?utm_source=postman-home



- **Step 2:**
Launch the Postman application on your computer.

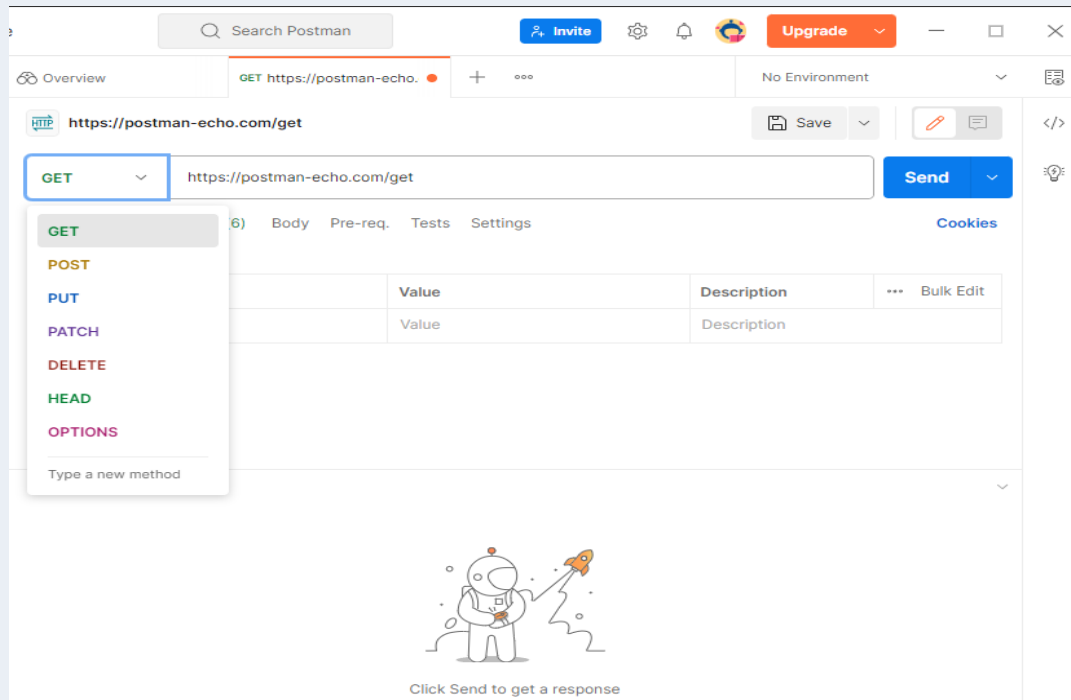


- **Step 3:**
Create a new request by clicking the “New” button and click on the HTTP button.



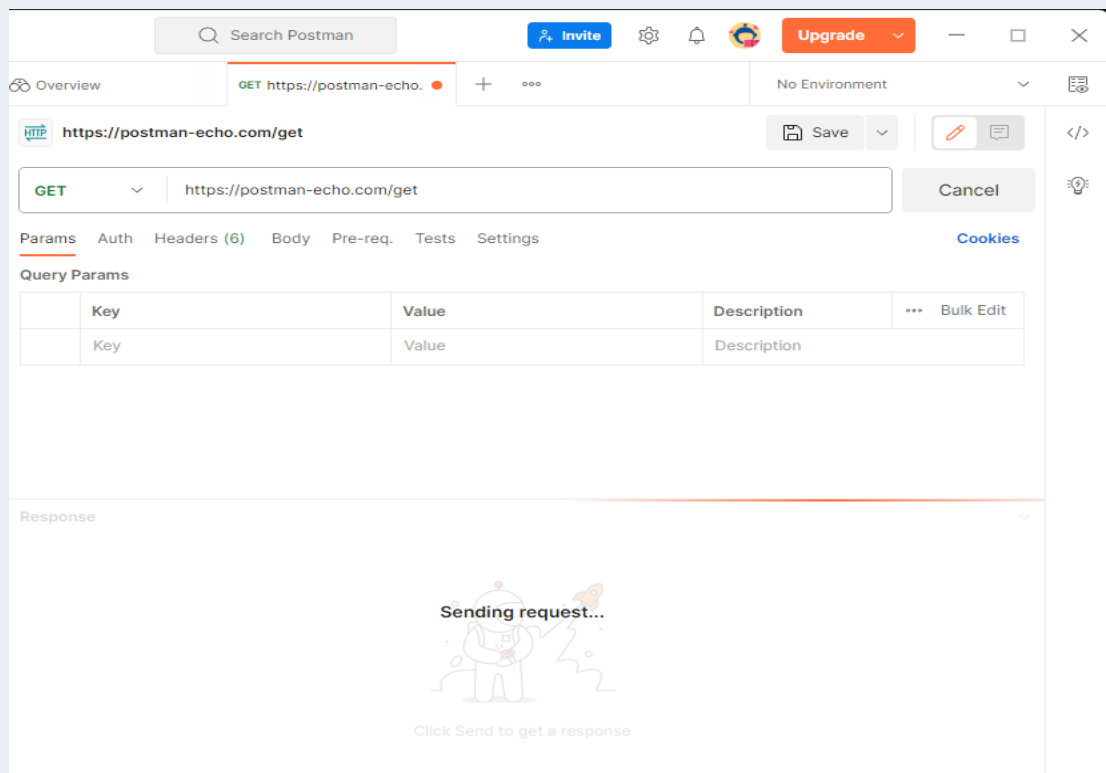
You can choose the request type from the list and enter the URL you wish to examine. You can use this URL for testing purposes:

<https://postman-echo.com/get>



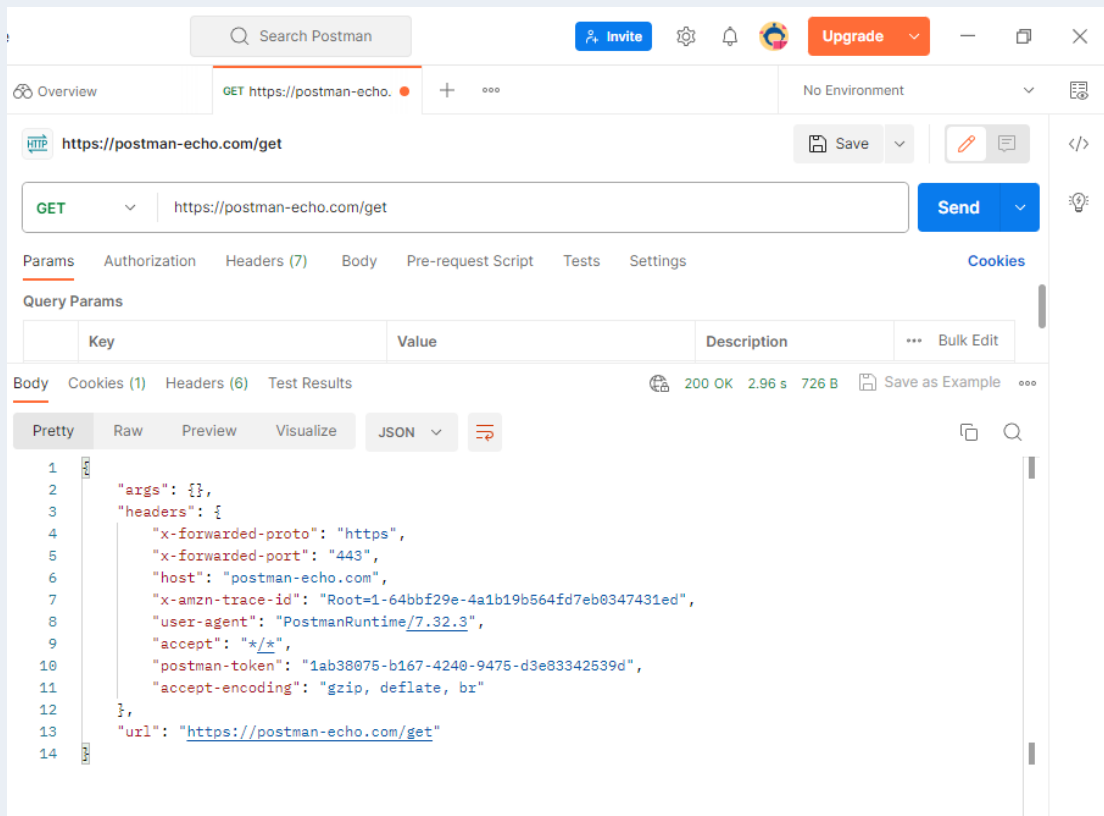
- **Step 4:**

Send the request to the server by clicking the “Send” button.

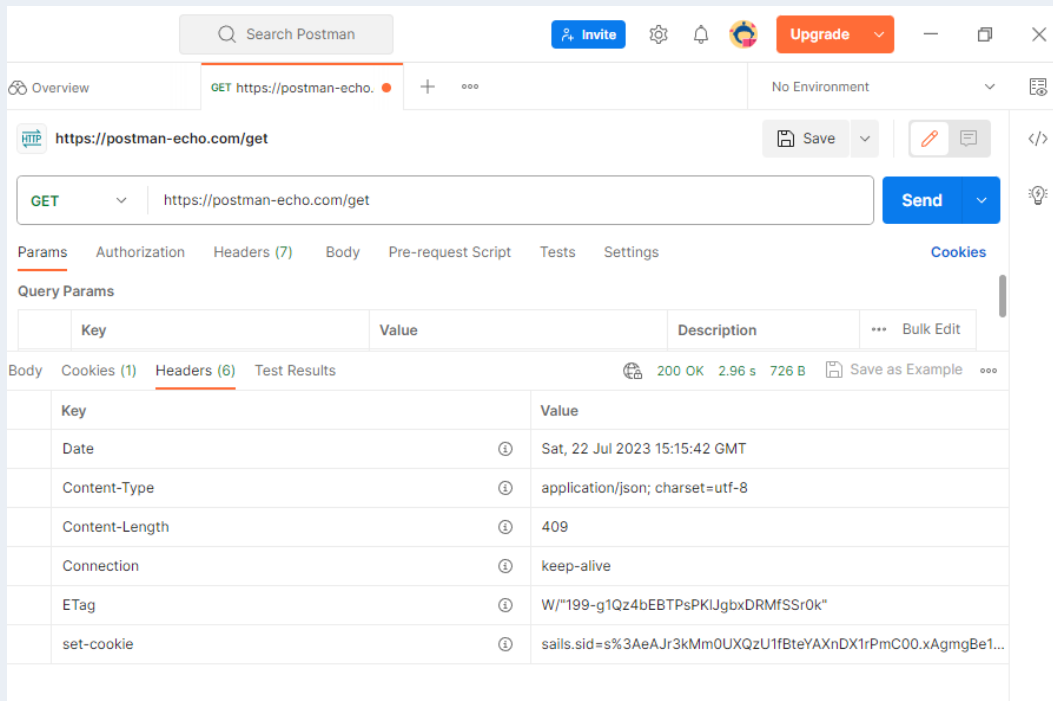


- **Step 5:**

Observe the response received in the Postman terminal including the headers, response body, status code, etc.



You can see the response headers as well.



Now that you're a bit more familiar with Postman, let's continue with the lab:

- Copy the example folder that accompanies this task to your local PC. In your command line interface, navigate to this folder and type **npm install**.
- Run the **people_server.js** file by typing **npm start**.
- Test the Restful API created in `people_server.js` with Postman. Create a folder called '**Screenshots**' in the folder for this task in Dropbox and insert screenshots (make sure each screenshot includes the response) of how you have used Postman to test this API to demonstrate your proficiency in the following:
 - Make an HTTP Post request to the API with the query string **?name=Jack** (e.g. **http://localhost:3000/person/?name=Jack**)
Make sure you enter the correct port number in the URL you use for testing, and that there are no trailing white spaces at the end of your request - either of these oversights could result in an error.
 - Make an HTTP put request to the API with the URL containing the parameter value "Samantha" for Jack to update their name.
(**http://localhost:3000/person/?name=Jack&newName=Samantha**)
 - Make an HTTP Get request to the API.
 - Make an HTTP Delete request to the API.

Things to look out for:

Make sure that you delete 'Node_modules' before submitting the code.

SPOT CHECK 1 ANSWERS

1. Express is used to speed up development and decrease the amount of boilerplate code you have to write.
2. Routing refers to determining how an application responds to a client's request to a particular endpoint.

SPOT CHECK 2 ANSWERS

1. `require()`
2. `.get()`
3. Nodemon enables server restarts on file changes.

SPOT CHECK 3 ANSWERS

1. Post	c. Used to submit some data about a specific entity to the server.
2. Get	d. Used to get a specific resource from the server.
3. Put	a. Used to update a piece of data about a specific object on the server.
4. Delete	b. Used to delete a specific object.

Completed the task(s)?

Ask an expert to review your work!

[Review work](#)



Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.

