
TASK

# Express – Web Applications

Visit our website

# Introduction

## WELCOME TO THE EXPRESS – WEB APPLICATIONS TASK!

Now that you are familiar with the basics of using Node.js, it is time to explore building server-side applications more efficiently with Express. Express is a popular, lightweight framework for Node.js that simplifies routing, handling HTTP requests, and managing static and dynamic content. In this task, you will learn to create a back-end web application using Express to handle these essential functions.

## UNDERSTANDING WEB COMMUNICATION

Web communication is the process through which clients (like browsers) and servers (like your back-end application) exchange information. This is achieved primarily through HTTP requests. The client sends a request to the server, and the server responds, often with data or HTML content. This request-response cycle is fundamental to how web applications function and enables interactions such as loading a web page, submitting a form, or accessing data.

## TYPES OF WEB COMMUNICATION

In web development, communication between a client and a server is essential. There are several types of architectures for managing this communication, such as **SOAP**, **GraphQL**, and **REST**. However, REST (Representational State Transfer) has become one of the most widely adopted methods for structuring APIs, due to its simplicity and flexibility.

Since REST is popular and well-suited for web applications, we will focus on RESTful principles in this task as you learn to build custom API endpoints with Express.

## 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. Middleware functions act as building blocks that handle tasks like processing requests, adding security, or parsing data before sending a response. 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."

Express is an **unopinionated** web framework. It offers more flexibility, allowing you to choose and integrate your own tools and components without strict guidelines. You can use compatible middleware in any order and structure your app however you prefer.

In Express, the "request" is the information sent by a client (like a browser) to the server, asking for a specific resource or sending data. The "response" is the server's reply, which can be data, an HTML page, or a status message, sent back to the client.

## 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

You will need to refer to the '**Additional Reading – Installation, Sharing, and Collaboration**' guide to complete your Express installation. That guide will walk you through everything you need.

## ROUTING

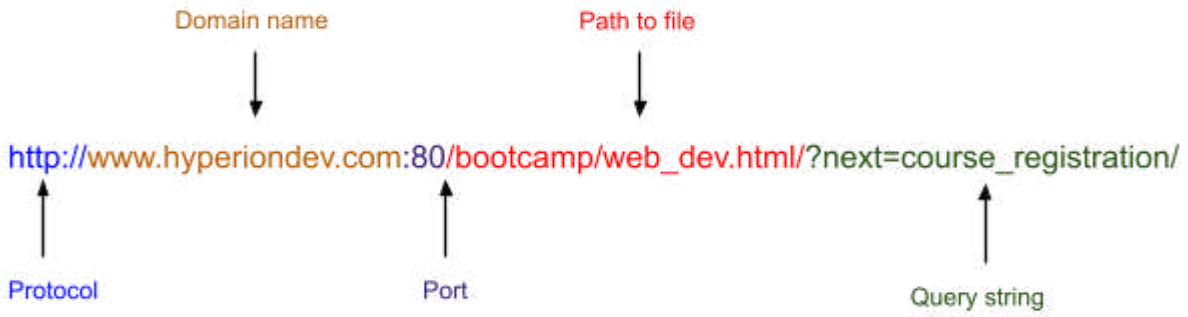
After installing Express, we can start creating the code for the back end of our web application. One of the most critical aspects of server-side logic is routing.

### What is routing?

One of the most important tasks of a server is to perform routing. Routing refers to how an application responds to requests made by clients. In this context, the client typically refers to a web browser or any application which sends requests to your server. For example, when you enter a URL in your browser and hit enter, your browser acts as a client by requesting information from the server.

Each request made by the client uses a URI (or path) along with a particular HTTP request method (such as GET, POST, etc.).

**Breaking down URLs**




As highlighted in the example URL above, a URL contains important information broken down below.

1. **Protocol**: It identifies the protocol being used to send information. In the example above, the protocol being used is HTTP.
2. **Domain name**: It identifies the domain name of the web server on which the resource can be found, e.g. www.hyperiondev.com.
3. **Port**: 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 do not have to be given in the URL.
4. **Path**: It gives the path to the resource on the web server, e.g. /bootcamp/web_dev.html.
5. **Parameters or query strings**: 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.

**Express route methods**

With Express, there are a few route methods used to perform routing: **GET**, **POST**, **PUT**, **PATCH**, 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 `"/"`, which is known as the "**root**" route of our application. The root route represents the base URL of your web application. 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!

## CREATE A SERVER USING EXPRESS

In the root directory of your Express project (the directory you created when you installed Express), create a new file called **index.js**. If you named your project directory differently or are using an existing NPM-initialised project, make sure you are working within that directory and copy the code below into it:

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

app.get("/", (req, res) => {
  res.send("Hello World!");
});

app.listen(8000, () => {
  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 index.js` in the terminal as per the example below:

```
node index.js
```

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 Static Files Folder:** In your project's root directory (where Express is installed), create a folder to store the static resources you want to serve. By convention, this folder is often named "public", but you can choose any name you like.

2. **Add Static Files:** 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. To avoid conflicts, organise files in subfolders (.eg. /images or /html) rather than directly in the root. This prevents static files, like `example.html` from unintentionally displaying when you access the root endpoint (`"/"`).

   Only include files in the public folder that you want users to access, such as images, CSS, and client-side JavaScript; keep sensitive files like configuration files or **API keys** outside this folder in a secure location to protect them from public access.

3. **Configure Express to Serve Static Files:** In your **index.js** file, add the following line of code to include the express.static middleware:

   ```
   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**.

Start the server by running `node index.js` in the terminal. This setup currently uses port 8000, but you may set a different port as needed within your application. For example, if you have a file called **example.html** in the **public** folder and your server is listening on port 8000, you can view it at: **http://localhost:8000/example.html**

## ENVIRONMENT VARIABLES

When building back-end applications, accessing environment variables is crucial for securely managing sensitive data like API keys, database credentials, and port numbers. These variables are set outside the code, typically by the server or operating system. So far, we have used variables defined directly in our code using `const` or `let`. However, back-end applications often need variables defined within the environment where the app is running. Node.js allows access to these through `process.env`.

To simplify working with environment variables, we can use the popular **dotenv** package, which loads environment variables from a `.env` file into `process.env`. This approach keeps sensitive information out of your codebase and centralises configuration.

**Setting Up dotenv**

1. **Install dotenv** by running the following command in your terminal within the project directory

```
npm install dotenv
```

2. **Create a .env file** in the root directory of your project. Inside this file, you can set your environment variables like this:

```
PORT=8000
```

3. **Load environment variables** from the .env file by adding the following line at the top of your **index.js** file:

```
require("dotenv").config()
```

   This command reads the variables in .env and adds them to process.env so you can use them in your application.

4. **Accessing the PORT variable:** Now, instead of hardcoding the port number, you can retrieve it from the environment variables with this code:

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

Using environment variables keeps sensitive information, like API keys, secure and allows you to change configuration details without modifying the source code. For example, when deploying to a web server, the server's environment will determine the port number, API endpoints, and other critical values without needing any changes to your codebase.

**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 cannot change once a value has been assigned. See more naming conventions recommended by Google for JavaScript **here**.

## NODEMON

When working on a Node.js application, it can become time consuming to restart your server every time you make changes to the code. **Nodemon** solves this problem by automatically restarting the server when it detects changes.

First, make sure your terminal is open in the same directory as your index.js and package.json files. Once you are in the correct directory, install Nodemon by running the following command:

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

Running this command will update the package.json file to include Nodemon in the devDependencies section, which indicates that Nodemon is a tool which is only needed during development.

Once Nodemon has been installed, instead of using the `node` command to run your application, you can use the nodemon command. For example, to run your application (assuming that the entry file is index.js), you would type the following:

```
nodemon index.js
```

Nodemon will now monitor the files in your project directory and automatically restart the server when any changes have been detected.

## ADD A START SCRIPT

To make it easier to start your application, you can add a script to your package.json file. This script can be used to define how to start the application and

allows you to simply run `npm start` in the terminal instead of typing the entire Nodemon command.

Here's how you can add the script in your package.json file:

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

Now, to start your application, you would only need to type the following command in the terminal (make sure that you are in the same directory as your index.js and package.json files):

```
npm start
```

A note from the

**HyperionDev Team**

There's a reason that true full-stack web developers are sometimes referred to as unicorns. It is 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**.

## A LOOK AT RESTFUL APIS

When we talk about APIs (Application Programming Interfaces) on the web, we are often referring to RESTful APIs. REST is a set of principles for designing networked applications, making it easy for different systems to communicate over HTTP. APIs designed with REST are known as RESTful APIs, which allow us to interact with data on a web server by using HTTP methods. Understanding RESTful APIs is essential for back-end development and is key to building applications that handle data dynamically.

A RESTful API operates by exposing resources – such as data on users, products, or any relevant content – through URLs, known as URIs (Uniform Resource

Identifiers). Each type of HTTP request is linked to a specific operation on these resources, forming a pattern called CRUD:

- **GET** (Read): Retrieves a resource's data.
- **POST** (Create): Adds a new resource.
- **PUT** (Update): Updates existing resource data.
- **DELETE** (Delete): Removes a resource.

These resources are accessed using URIs, and data exchanged via RESTful APIs can be formatted in JSON, HTML, XML, plain text, PDF, JPEG and other formats. By being stateless, RESTful APIs keep each interaction independent, with any required state information passed through techniques like URL rewriting or cookies, without storing session details on the server.

## CREATE A CUSTOM RESTFUL API USING EXPRESS

Express makes creating a RESTful API straightforward by providing built-in methods to handle each HTTP request type. 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.

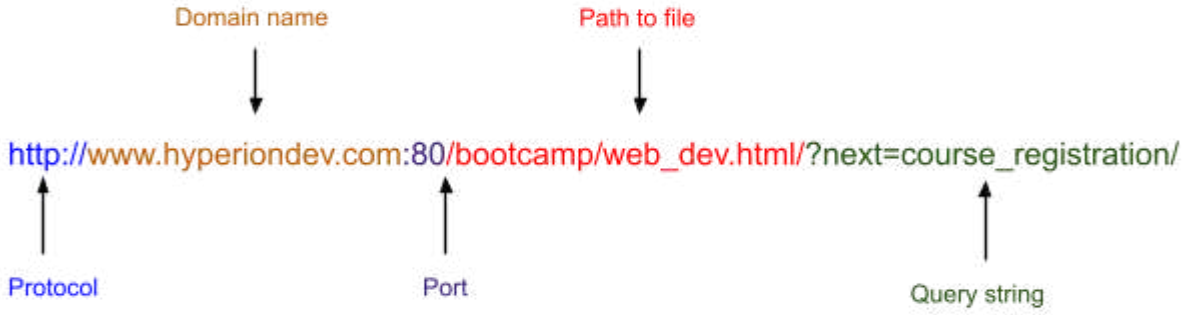
## PASSING DATA THROUGH TO THE SERVER USING THE REQUEST OBJECT

An important role of the server is to receive necessary data that is passed through from the browser. Express provides the **req** (request) object to access data sent from a client to the server, and it includes several useful properties for handling data passed through URLs. Here's a deeper look at two key properties of req: **req.query** and **req.params**

### req.query

**Query strings** contain key-value pairs appended to the end of a URL, introduced by a `?` and separated by `&` for multiple pairs. They are primarily used to send optional data, such as search terms or filters, in a GET request.

Example:




The `req.query` property accesses data passed via query strings. This property contains an object where each query string key maps to its corresponding value. This is especially useful for GET requests that need extra data.

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

Assume the URL is the above example:

```
app.get("/", (req, res) => {
  // Extract data from the query string
  const next = req.query.next; // 'course_registration' comes from
the query string 'next=John'

  res.send(`Result: ${next}`);
});
```

**req.params**

URL parameters, on the other hand, embed data directly within the URL path itself. They're useful for routing to specific resources (such as user profiles or product details) and are typically used in RESTful routes.


The image above shows an example URL with a parameter added to it. In this case, the parameter **2315** could represent the ID of a student at HyperionDev.

In Express, you can retrieve values passed as URL parameters using the `req.params` property. This property contains an object where each parameter name (defined in the route) is a key, and the actual value provided in the request URL is the corresponding value.

For example, if the URL is **/portal/2315**, then `req.params.id` will retrieve **2315**.

In the example below, `req.params.id` is used to access the `id` parameter from the URL:

```
// Route handler for a GET request to the root path ('/:id)
app.get('/portal/:id, (req, res) => {
    // Access the id parameter
    const id = req.params.id;

    // Create a greeting message using the id
    const greeting = `Hello, student with ID ${id}!`;

    // Send the greeting message as a response
    res.send(greeting);
});
```

**Combining req.query and req.params**

It's possible to use both `req.query` and `req.params` in the same route to handle a combination of URL parameters and query strings. This approach is useful when a specific resource (e.g., a user by ID) requires additional filtering or sorting criteria.

For instance, you might have a route like `/users/:id`, where `id` identifies the user, and `req.query` allows for optional parameters like sort or filter. This way, req.params can retrieve the main identifier (`id`), while `req.query` handles additional options.

Example:

```
app.get("/users/:id", (req, res) => {
    const userId = req.params.id;       // Captures the user ID from the
URL
    const sortBy = req.query.sort;      // Captures additional sorting
options
    res.send(`User ID: ${userId}, Sort By: ${sortBy}`);
```

```
});
```

Have a look at the code example that accompanies this task to see some similar 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.

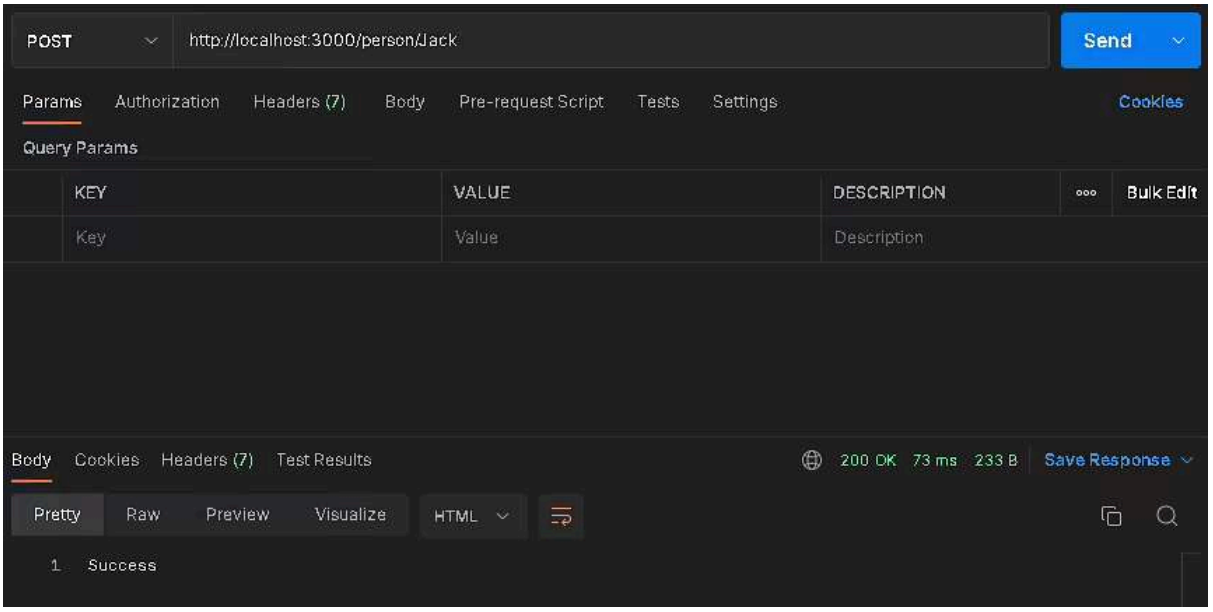
JSON is widely used in APIs because it organises data into simple, readable key-value pairs. This format allows data to be structured clearly, making it ideal for communication between clients and servers.

Using JSON ensures that data shared through our API is consistent and easy to process. This structured format enhances the API's ability to work smoothly with different applications, promoting compatibility and efficiency.

## SOME NOTES ON USING POSTMAN

Before using a front end to pass data to the server, Postman offers an easy way to interact directly with your server and test API functionality. Postman is a powerful tool for testing APIs, as it lets you send requests and view responses without writing additional code, which is helpful for debugging and validating your API's behaviour.

In this example, Postman is used to send a POST request to the server we configured. In the image, you will see parameters in the request that match the data structure expected by the server. This allows us to verify that the server handles and responds to these requests as intended. With Postman, you can also test other request types (e.g., GET, PUT, DELETE) and send parameters as query strings, in the body, or as URL parameters.




# Instructions

- The Express 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 has the potential to **slow down** your computer. As a result, please follow this process when creating/running such apps:
    - Create the app on your local machine by following the instructions in the auto-graded task.

- 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.

## Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give the task your best attempt and submit it when you are ready.

After you click "Request review" on your student dashboard, you will receive a 100% pass grade if you've submitted the task.

When you submit the task, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for this task, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you've done that, feel free to progress to the next task.

---

# Auto-graded 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. Resource on generalised error handling **here**.

- Enable the server to restart on file changes.
- You should be able to start the server using `npm start`.

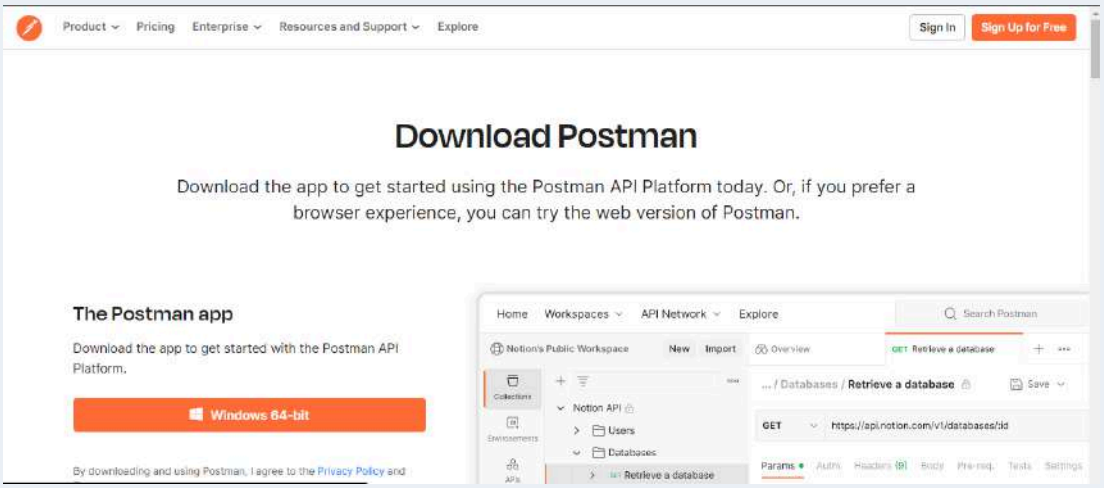
Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

# Auto-graded task 2

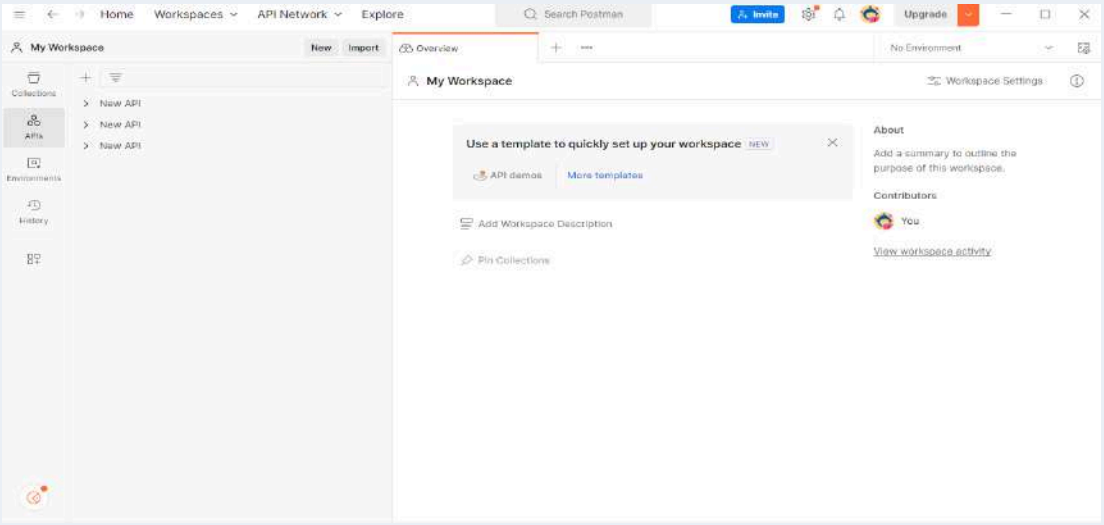
Follow these steps:

- 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:

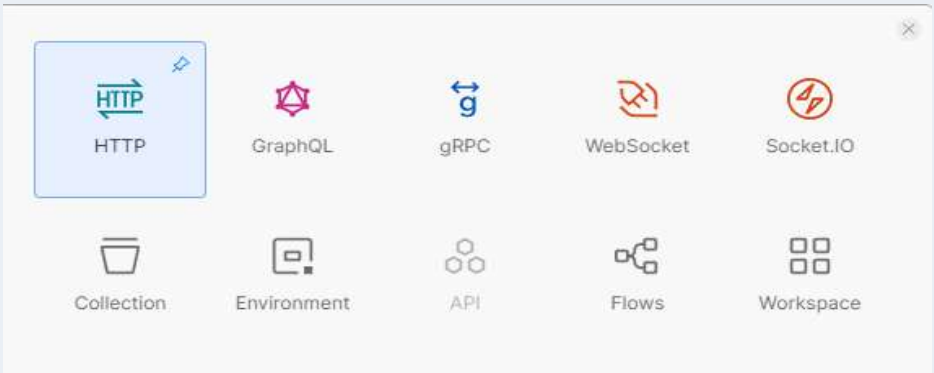
  - **Step 1:**
    Download and install Postman on your computer:
    **https://www.postman.com/downloads/?utm_source=postman-home**



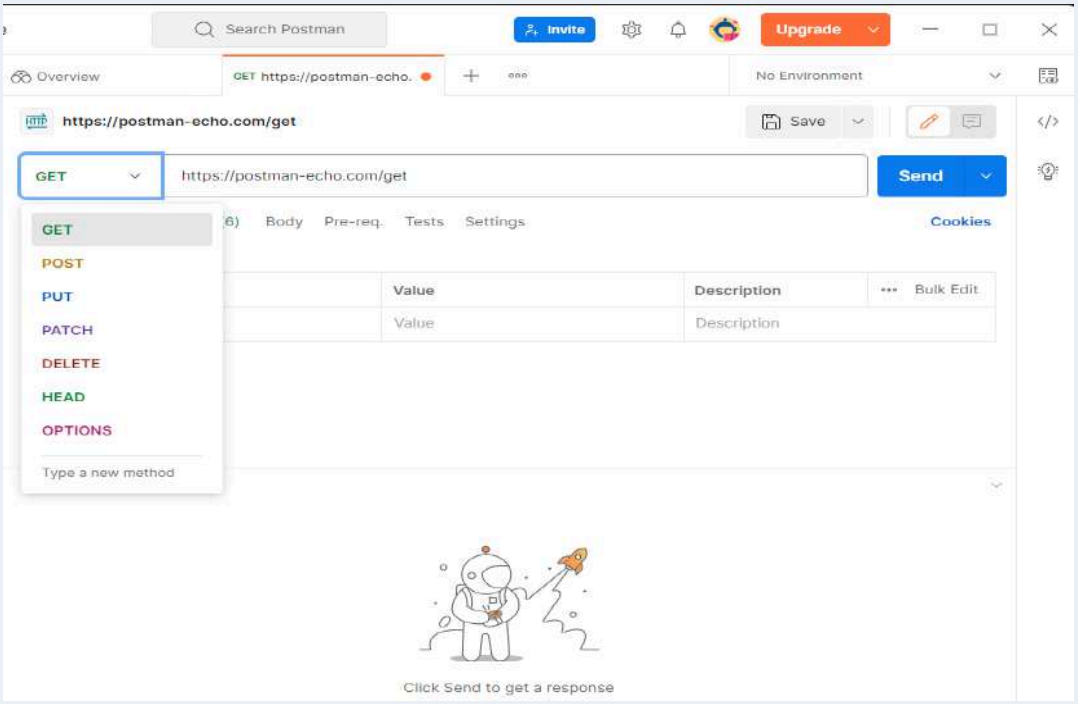
- **Step 2:**
  Launch the Postman application on your computer.



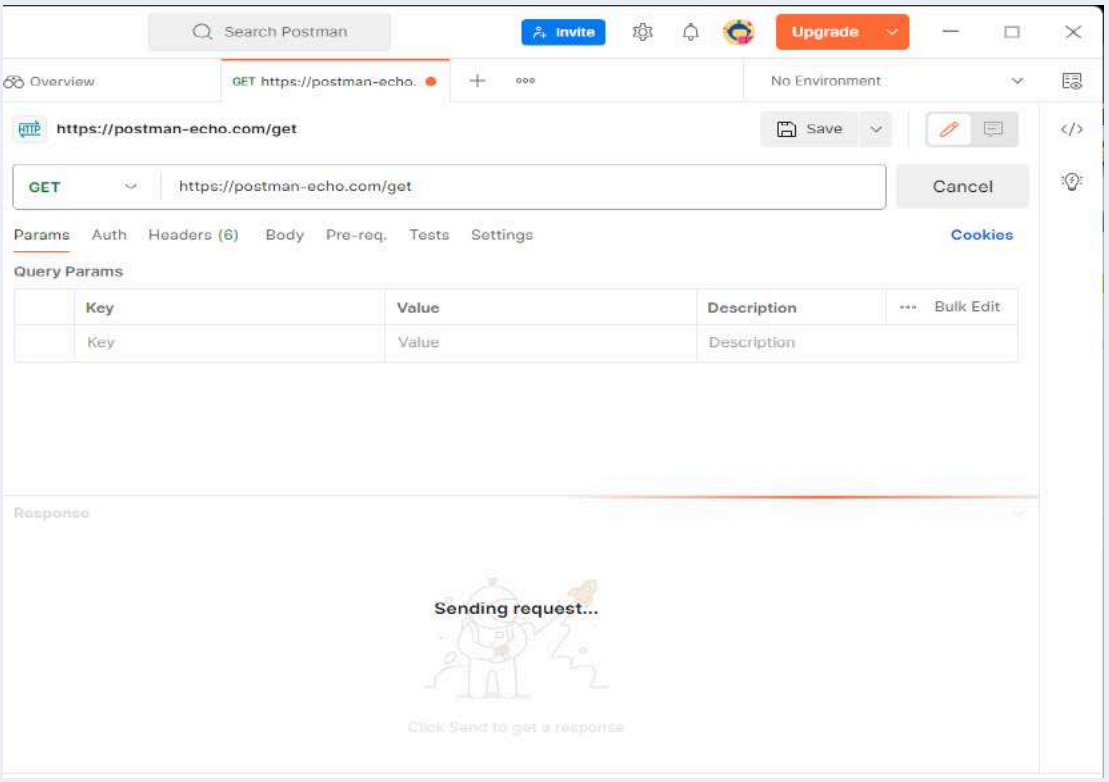

- **Step 3:**
  Create a new request by clicking the “New” button and click on the HTTP button.



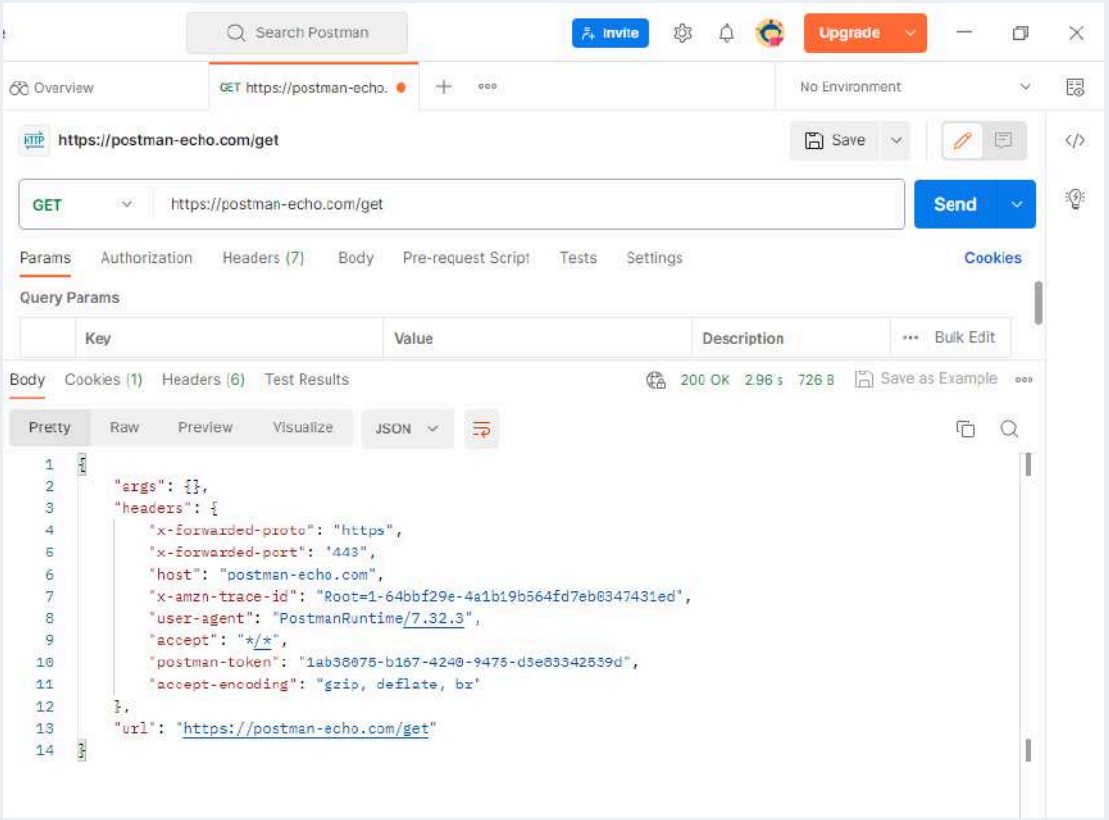

You can choose the request type from the list and enter the URL you wish to examine. You can use this URL for testing purposes:
**https://postman-echo.com/get**



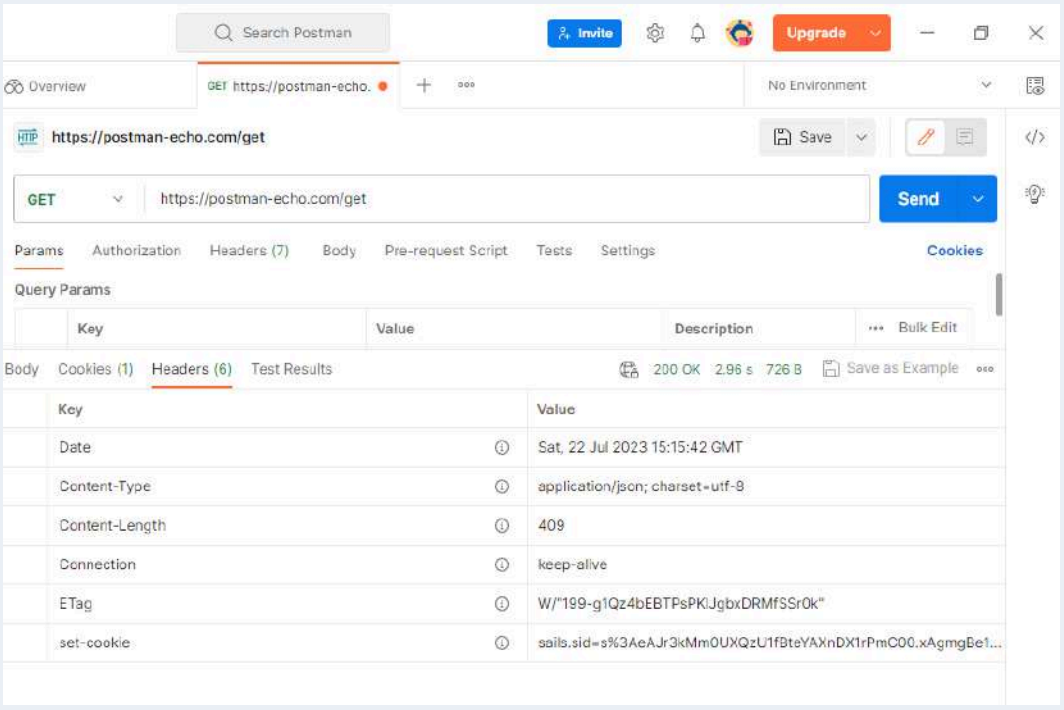

- **Step 4:**
  Send the request to the server by clicking the “Send” button.



- **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 task:

- Copy the example folder that accompanies this task to your local PC. Open a command line interface, and navigate to the folder where the example files are located. Then, follow these steps:
    - Install dependencies: Type `npm install` in the command line. This command will install all required dependencies listed in the **package.json** file, including Express.
    - **Run the server**: Start the **people_server.js** file by typing `npm start` in the command line.
        - The npm start command uses the start script specified in the **package.json** file to run the server. The start script is often configured to execute `node people_server.js` or a similar command, which launches the server.
        - Once the server is running, you should see a message indicating it's live on a specific port.

- Test the Restful API created in **people_server.js** with Postman. Create a folder called '**Screenshots**' in the folder for this task 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 to check if a specific person exists.
    - Make an HTTP DELETE request to the API to remove a specific person.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

## Things to look out for:

Make sure that you delete ‘Node_modules’ before submitting the code.


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.