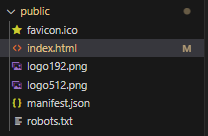
**React and Python Full Stack App Course**

**Creating a React App**[**https://create-react-app.dev/docs/getting-started/**](https://create-react-app.dev/docs/getting-started/)

Once created, you can run **npm start** and the app should start in a localhost tab.

**Changing Favicon**

There are different ways to edit the favicon icon. Favicon is not actually a single image.

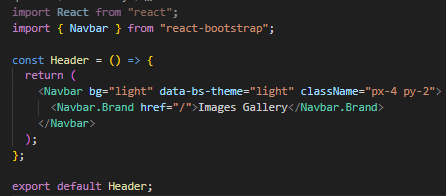
In the public folder, there are files such as **favicon.ico** or **logo.png**.

In the manifest.json, there is an array of different icons that are available for usage, and different client devices, different web browsers could access different icons.

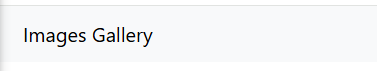
If a web browser requests an icon with size 32x32, then the application will service the relevant icon file.

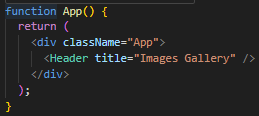
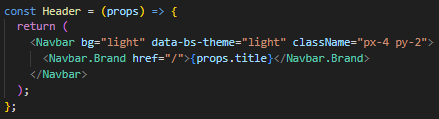
**Using React Boostrap**<https://react-bootstrap.netlify.app/docs/getting-started/introduction>

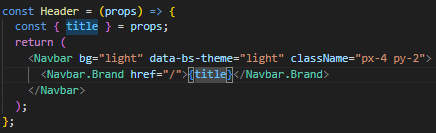
Before importing Bootstrap components, we need to import the specific stylesheet.



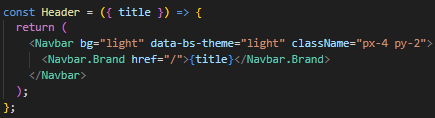
Use a basic navbar design from the link provided (make sure to add spacing if required).

You will get a very subtle whiteish grey navigation bar as a header once you import and display the JSX component in App.js.

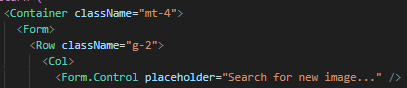
**Using Props in React Components**

Here, we pass the title prop through props into the Header arrow function component, which uses it.

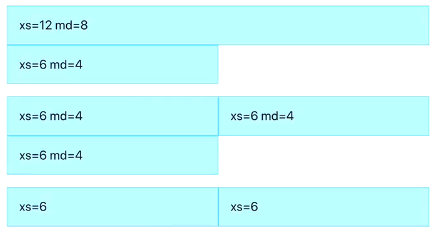
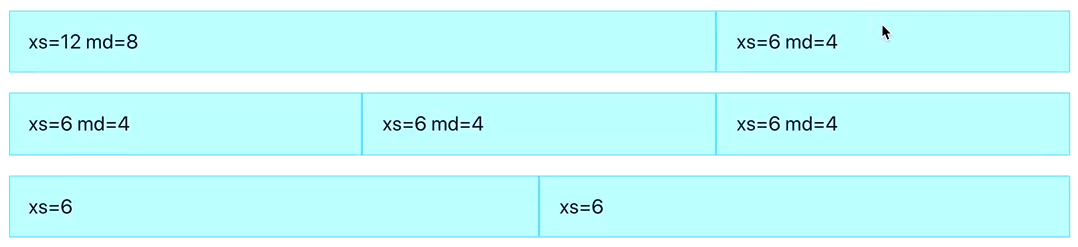
You can also extract the title prop using destructuring and then pass the prop name to the relevant place without dot notation.



Finally, the best way of doing it is to simply destructure the prop name itself in curly braces in the function, and pass this to the relevant place in the component.

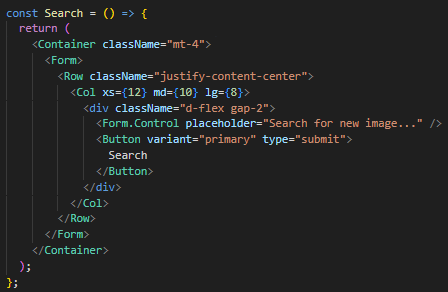
**React Bootstrap Styling**For example, **mt-4** or **g-2**.

* **Container  
  *container****:* A responsive fixed-width container  
  ***container****-fluid:* Always takes 100% of the width  
  ***container****-{breakpoint}:* Fixed-width container at the specified breakpoint, and fluid below it.
* **Row  
  *gx-0*** to ***gx-5:***Horizontal gutter (column spacing)  
  ***gy-0*** to ***gy-5:*** Vertical gutter (row spacing)
* **Col  
  *xs, sm, md, lg, xl, xxl:*** Control column width for each screen size. Accepts numbers 1-12, ***auto*** *or* ***true*** *(equal width).*
* **Bootstrap Breakpoints (used in Container, Row, Col)  
  *xs*** *–* extra small (<576px)  
  ***sm*** *–* small (>= 576px)  
  ***md***– medium (>=768px)  
  ***lg*** – large (>=992px)  
  ***xl***– extra large (>= 1200px)  
  ***xxl***– extra extra large (>=1400px)
* **Spacing Utilities (Margin and Padding)  
  *m, p:***margin, padding  
  ***t, b, s, e, x, y:***top, bottom, start, end, horizontal, vertical  
  ***0-5:***size scale (0 = 0, 1 = 0.25rem, …, 5 = 3rem)  
  ***auto:*** use auto value
* **Text Utilities  
  *text-start:*** Left-aligned text  
  ***text-end:*** Right-aligned text  
  ***text-center:*** Centre-aligned text  
  ***text-uppercase:*** Transform text to uppercase  
  ***text-muted:*** Muted (grey) text colour  
  ***fw-bold***: Font-weight bold  
  ***fs-4:*** Font size heading scale (1-6)
* **Display and Flex Utilities  
  *d-none:*** display: none  
  ***d-flex:*** display: flex  
  ***d-inline:*** display: inline  
  ***flex-column:*** flex direction column  
  ***justify-content-center:*** centre children horizontally  
  ***align-items-center:*** centre children vertically
* **Width/Height  
  *w-100:*** width: 100%  
  ***w-auto:*** width: auto  
  ***h-50***: height: 50%  
  ***min-vh-100:*** min-height: 100vh
* **Responsive Variants**If you were to give the **className=“mt-3 mt-md-5”**, this would mean that the margin-top would be 3 on all screen sizes, but overrides this to 5 when the screen size is md or up.

****You can also mix and match breakpoints to create different grids depending on the screen size. The screen is split into 12 imaginary columns (1-12).  
When the screen is of medium size, the columns will take up (e.g.) 2/3rds and 1/3rd of the screen respectively (md=8, md=4)

When the screen is super small (xs), then column 1 will take up the whole of the screen and column 2 will take up half of the screen (xs=12, xs=6).

**Xs** and **md** are not both applied at the same time, it is only when the screen size changes that they kick into action with their specified proportions.

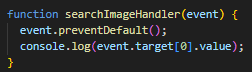
**Explaining my Code**

**Mt-4 –** adds a top spacing between the container and whatever is above it.

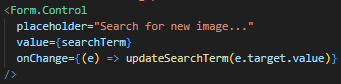
**Justify-content-center –** Centres the content of the row (the col below) in the horizontal space.

**Col xs md lg –** Responsive width that gets narrower on larger screens, helping centre and contain the form nicely.

**D-flex gap-2 –** input and button are laid out in a row (side by side), with 0.5rem space between.

**Why you need to use State in forms****This is incorrect.** We are accessing the form value, which is fine, but this approach couples the logic to the structure of the DOM, meaning the value itself lives in the DOM and not in React’s memory.

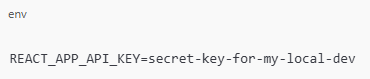
Instead, we need to use state, which causes React to track every keystroke, and update its value in React’s memory. This not only allows you to track the input more closely, but you can add features, such as enabling/disabling buttons based on input, resetting the input after submission and validate/transform the input as the user types.

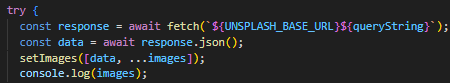
In the search component’s form, we see its value is equal to the current state. This is updated whenever a change is detected.

However, if **onChange** was commented out, we would be unable to type anything in the search bar as its value is equal to state, and if we can’t update state, its value cannot be updated.

**Using UnsplashAPI**<https://unsplash.com/developers>

**Different ENV files**

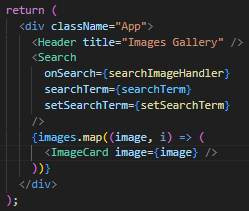
* **.env.local –** this is used only in your local development – by default it is in the .gitignore file so you can store any secrets here, such as API keys.  
    
  
* When deploying an application, you need to add the hidden API keys manually to your hosting provider’s environment variables settings.
* For default values or non-sensitive config that can be safely committed to version control, you can use the **.env** file, for instance:  
  ****

**How set state works when adding to arrays**

Each time an image is searched, it is added to the array. However, it is only when you search for the next image, you see the array update with the previous search.

This is because it is **asynchronous**, so logging the images object in the same statement will not display the updated state yet. Move the console log underneath the state statements.

**Using map to map over elements in the array**

****The images array contains multiple image objects.

We map over each object, and create a JSX component that takes the image as a parameter and creates an image card for it.

**Requests and Responses**

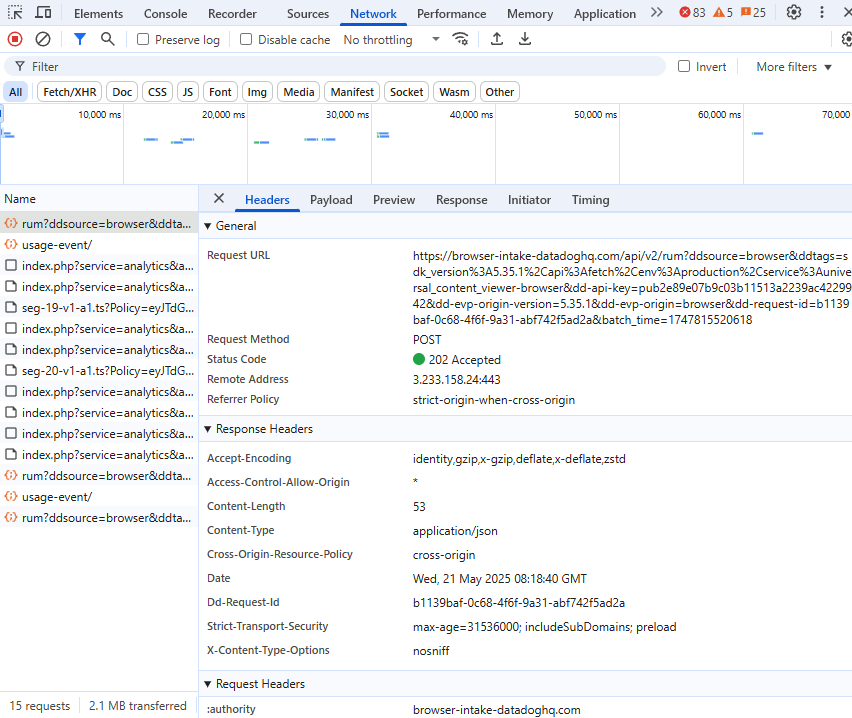
**Request –** sent from the client to the server

* **URL –** Identifier of the specific resource
* **Method –** GET, POST, DELETE and others
* **Headers –** Cookie, User-Agent, Authorisation and others
* **Data (body) –** (Optional) - JSON, Form Data or others

**Response –** sent from the server to the client

* **Status line –** Protocol version, status code, status text
* **Headers –** Content-Type, Date, Server, Set-Cookie and others
* **Data (body) –** (Optional) – JSON, Document, Stylesheet, PNG and others

**Understanding Requests from Browser**

****In the browser, under Network, you can click on any event name, and it will break down the type of request it was under the ‘Headers’ tab.

**Request Summary**

* **Request URL –** The full endpoint being called
* **Request Method –** HTTP method used (e.g. GET, POST)
* **Status Code –** (e.g. 200 OK) – the request succeeded
* **Remote Address –** The IP address and port of the server

**Request Headers**These are sent from your browser to the server (Client 🡪 Server)

* **Accept –** Tells the server what content types the client can handle
* **Content-type –** The format of the request body (e.g. application/json)
* **Cookie –** All relevant cookies tied to your session and identity
* **User-agent –** Your browser type, OS etc…
* **:authority, :method, :path, :scheme –** Pseudo-headers used in HTTP/2 for routing
* **Content-length –** Size of the request body in bytes

**Response Headers**These are the server’s metadata in response (Server 🡪 Client)

* **Access-control-allow-origin –** Controls CORS (Cross-Origin Resource Sharing) – allows web pages to access resources from a different domain that the one serving the page.
* **Content-type –** the format of the response body (e.g. application/json)
* **Cf-cache-status –** From Cloudflare – DYNAMIC means content is generated dynamically.
* **Strict-transport-security –** Enforces HTTPS (prevents downgrade to HTTP)
* **X-content-type-options –** Prevents MIME type sniffing (nosniff)
* **X-robots-tag –** Tells search engines not to index this content (nofollow)
* **Cf-ray, server, date –** Cloudflare-specific debug/trace info

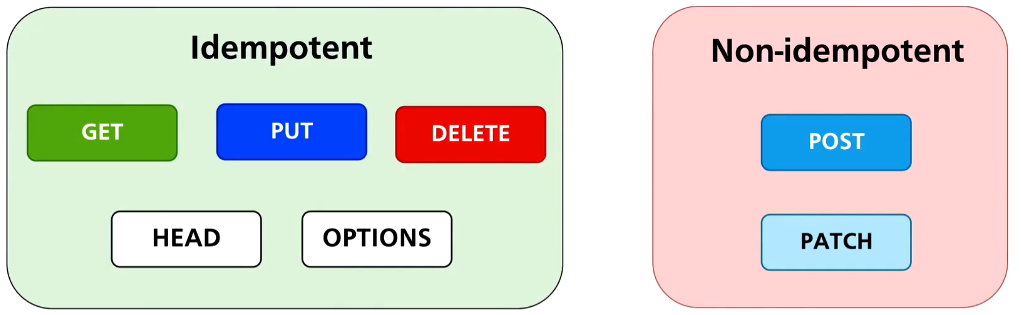
**Cookies**These carry authentication tokens, session state, preferences, and are sent automatically with the request due to the browser’s handling of sessions.

* **\_Secure-next-auth.session-token –** your ChatGPT session
* **Cf\_clearance, \_cf\_bm –** Cloudflare cookies used for bot protection and rate-limiting

**Security Context**

* **CORS Headers –** access-control-\* - ensure requests come from allowed origins.
* **Referrer Policy, Strict-Transport-Security,** and **Cross-Origin-Opener-Policy** are all mechanisms to harden security against XSS, data leaks, or clickjacking.

**Idempotent HTTP Methods**Idempotent operations produce the same result when executed multiple times.

**GET –** calling GET once or 100 times will return the same data and won’t change anything on the server.

**PUT –** sending the same PUT request multiple times has the same effect as sending it once – it keeps overwriting the resource with the same data.

**DELETE –** deleting something once or multiple times has the same end result: the resource is gone. Even if it’s already deleted, repeating the request doesn’t change anything more.

**HEAD –** is read-only and doesn’t change anything.

**OPTIONS –** just asks for information and doesn’t change anything.

**POST –** calling POST multiple times usually creates multiple resources. Each call can have a different effect.

**PATCH –** It **can** be idempotent, if the same PATCH request results in the same change every time. But it’s not **guaranteed** to be idempotent like PUT.

**What are HEAD and OPTIONS?**

* **HEAD –** similar to GET, but only asks for headers – not the actual content. For example, checking if a page exists or how big a file is, without downloading it.
* **OPTIONS –** Asks the server what methods are allowed for a resource. For example – “what can I do with this URL?”

**Using Pipenv and its Virtual Environment**

**Advantages of Pipenv**

1. **Automatic Virtual Environment Management**

* Pipenv creates a separate, isolated Python environment for each project.
* Prevents version conflicts between dependencies used in different projects.
* You don’t need to manually create/activate virtual environments, Pipenv does it for you.

1. **Simplified Dependency Management**

* Pipfile replaces *requirements.txt*and separates:  
  [packages] 🡪 for production dependencies  
  [dev-packages] 🡪 for development tools (e.g. linters, test runners)

1. **Lock Files for Reproducibility**

* Pipfile.lock ensurs consistent installs across machines and deployments.
* Guarantees exact versions, preventing “it works on my machine” bugs

1. **Cleaner and Easier to Use**

* Commands like *pipenv install*, *pipenv uninstall*, and *pipenv shell* are more user-friendly than juggling pip + virtualenv manually.
* Easy to activate a shell in the environment: *pipenv shell*

1. **Built-in Safety Checks**

* Automatically checks for known security vulnerabilities in the installed packages (*pipenv check*)

**Advantages of Flask**Flask is a micro web framework for Python. It’s minimal, but highly extensible.

1. **Lightweight and Minimal**

* Comes with only the essentials: routing, request/response handling, and templating.
* You can add only what you need (ORM, form validation, etc…)

1. **Modular and Flexible**

* No rigid structure – great for small projects, APIs, and even larger apps if architected carefully.
* Easy to integrate with other libraries like SQLAlchemy, Jinja2, WTForms, etc…

1. **Quick to Develop**

* Ideal for rapid prototyping and MVPs.
* Simple to get a “Hello, World” server running in minutes.

1. **Extensive Ecosystem**

* Huge number of plugins and extensions (*Flask-Login, Flask-Migrate* etc…)
* Large and active community with lots of tutorials and third-party tools.

1. **WSGI-Compatible and Deployment-Ready**

* Runs on any WSGI-compliant server (e.g. Gunicorn, uWSGI)
* Plays well with Docker, Heroku, AWS, and other deployment platforms.

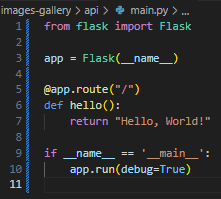
**Pipenv + Flask = Ideal for Many Python Web Projects**

* Clean project isolation
* Easier collaboration and deployment
* Fast development cycle
* Security and reproducibility

**Setting Python Interpreter to .venv**This allows you to run your code using the interpreter inside **.venv** and not the global or Anaconda Python.

You can install and manage packages inside **.venv**, so dependencies are isolated for this project only.

**Basic Flask App**

****If you run **python main.py**, a development server will start, where you will see “Hello, World!”

**App** is your flask application. **\_\_name\_\_** tells flask where to find resources such as templates and static files. It helps Flask to know whether you’re running the app directly or importing it from somewhere else.

**App.route()** – when the user visits the homepage, run the function below.

If **name == main** checks whether the script is being run directly (not imported), then starts the development server. **Debug=True** enables auto-reload on changes and gives error messages.

**Understanding Flask(\_\_name\_\_)**In every Python file, there’s a special built-in variable called **\_\_name\_\_.**

* If the file is run **directly** (e.g. python main.py).  
  **\_\_name\_\_ == “\_\_main\_\_”**
* If the file is imported as a module (import main)  
  **\_\_name\_\_ == “main”**

This allows your file to do two things:

* Be run directly as a script (and start the server).
* Be imported as a module (without starting the server).

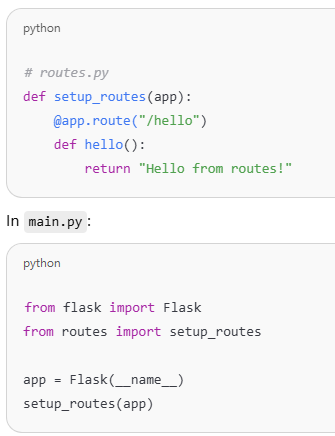
The Flask app can live in any file – main.py, app.py, server.py – what matters is how you run it and how you structure it.

If you run **python main.py**, the app.run() line will execute and Flask starts the server.

If you **import** **main**, then main.py’s code will run, but **\_\_name\_\_ == “main”** and not **\_\_main\_\_**, so app.run() does not execute (due to the logic).

**Why run main.py code without starting the server?**

1. **Testing –** you may want to write unit tests for your Flask routes without starting the server.
2. **Modular Code –** now main.py holds the Flask instance and is imported by other files, so you don’t want it starting a server every time it is imported.
3. **Deployment –** When deploying with tools such as Gunicorn or uWSGI, you don’t use python main.py, you run **gunicorn main:app**, meaning main gets imported, and **Gunicorn** runs the app, not the **app.run()** line.
4. **Command-Line Scripts or CLI Utilities –** sometimes your file might include functions or CLI tools for setup, debugging, database migrations etc… You want the option to import and reuse these functions without starting the server every time.

**What are other files doing with main.py?**They don’t just need to find resources using **Flask()**, but often need to attach themselves to the Flask app.

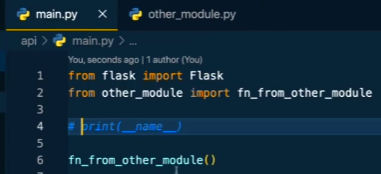
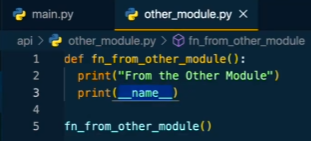
Here, **routes.py** uses the app passed to it. It doesn’t create a new app, but **adds routes** to the existing one.

If every file made its own **Flask(\_\_name\_\_)** instance, then you’d have multiple apps and they wouldn’t share routes, configuration etc…

So instead, you create the app once, in **main.py**, and everything else plugs into it.

There are also other files that define objects (such as **SQLAlchemy** creating a database object), that don’t know anything about the app. In this case it is imported into **main.py**, which initialises everything.

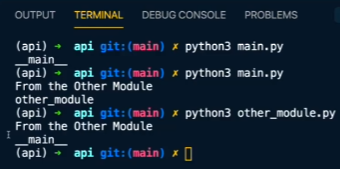
Flask only uses the **\_\_name\_\_** of the file where **Flask(\_\_name\_\_)** is called (usually **main.py** or **app.py**).

If you import the other module to **main.py**, and run the function, it will execute and print the name of the **other\_module**.

This is because **main.py** calls the external function, which executes from within the **other\_module**. It will return the name **other\_module.**

However, if you run the function from the **other\_module** directly, it will return **main.py** as the \_\_name\_\_ instead.

When a file is run directly, **\_\_name\_\_ == “\_\_main\_\_”.**

When a file is imported as a module, **\_\_name\_\_ == “module\_name”.**

The **\_\_name\_\_** variable is not derived from the file name when run directly. It is only set to the filename when the file is imported. Hence, when we run **main.py**, the **\_\_name\_\_** becomes the name of the imported file.

**Running a Flask App**

By initialising the Flask app and environment, you can run the app.

However, as your code contains this block, you can simply run **python** **main.py**.



**App Routes**

****This **hello** function is executed whenever the URL path ends with **/hello**, and will return a 404, if any other path is entered that isn’t specified.

**App.run()**

**App.run()** starts the development server if the script is run directly.

**Host=“0.0.0.0” 🡪** Makes the app accessible on all network interfaces, not just localhost. This means the app can be reached from any device that can connect to your computer’s IP address (assuming no firewalls block it).

**Port=5050 🡪** Runs the app on port 5050 instead of the default 5000

**Debug=True 🡪** Enables debug mode, which auto-restarts the server on code changes and provides a debugger if an error occurs.

**Host=“0.0.0.0”**

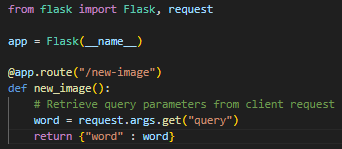
* Binds the app to all available network interfaces on your machine.
* Allows the app to be accessed using:  
  🡪 Localhost or 127.0.0.1  
  🡪 Your private IP address on the LAN (e.g. 192.168.1.20)  
  🡪 Your public IP (if your machine is internet-facing and ports are open)
* Useful for:  
  🡪 Testing from another device on the same network  
  🡪 Running in docker or a CM, where Flask needs to be exposed  
  🡪 Allowing access from a phone, tablet, or other computers
* **Host=“127.0.0.1”**🡪 Binds only to the loopback interface  
  🡪 App is only reachable from the same machine  
  🡪 Can’t be accessed from another device or VM

**Connecting another device to your app**

1. **Run your Flask app with host=“0.0.0.0”**
2. **Find your computer’s local IP address**- Open Command Prompt  
   - Type: *ipconfig*  
   - Loof for **IPv4 Address** (e.g. 192.168.1.100)
3. **Connect from another device that is on the same Wi-Fi network**- Open a browser and go to:  
   **http://<your-computer\_ip>:5050**
4. **Check firewall settings**Make sure your system firewall allows connections on port 5050.  
   - On Windows, allow Python through the firewall when prompted.

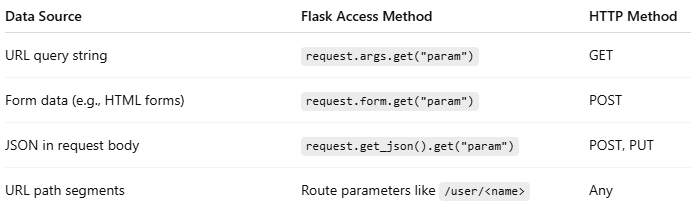
****

**Creating a New Image Endpoint**

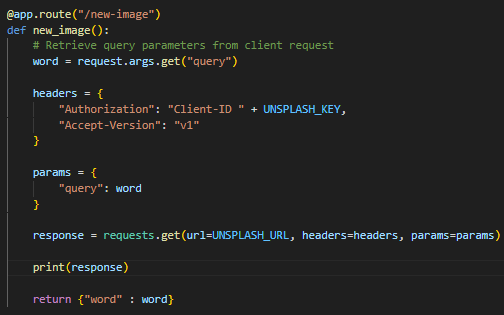
****

The **new\_image** code retrieves a query parameter from the URL. **Request.args** is a dictionary-like object that contains GET parameters.

It tries to get the value of the **query** parameter from the URL and then returns a JSON.

**Request.args.get()** only pulls data from the URL query string.

As you can see, there are different Flask access methods that correspond to retrieving data from other parts of the request, using different data sources.

**Making Requests Using Headers**<https://pypi.org/project/requests/>

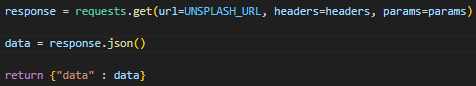
We can import **requests** and then state our different request headers in **requests.get()**.

The headers object is a dictionary that contains the authorisation key and the accept version (required for unsplash).

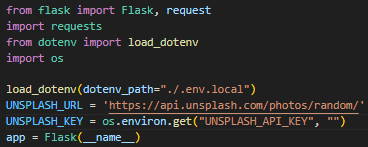
The params dictionary contains the different parameters we pass to the request, in this case, our **query** parameter.

The **response**, followed by **response.text**.

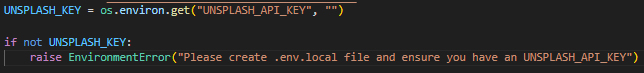




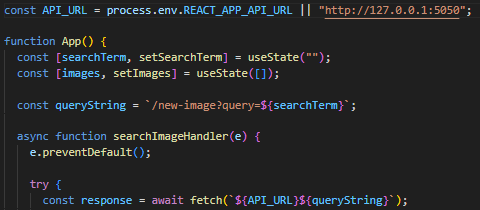
You can also return the JSON file by using **response.json()**.

**Getting API Key from local.env**

To get the API key from local.env, we need to use **os.environ.get()**. However, since the local.env is ignored in gitignore by default, we need to use **dotenv** and **load\_dotenv** to make it visible.

We provide the relative path to the file using **load.env**.

To prevent your app from starting if there is no API key, you can write an if statement and **raise EnvironmentError**, which will display a message in the console when there is no API key found, instead of starting the app.

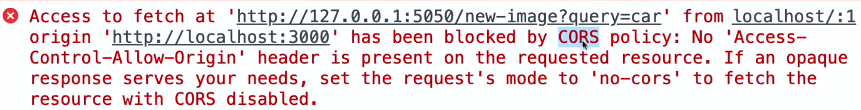
**Querying the backend API from the frontend application**

By setting the API\_URL, you can make it equal to an environment variable, or to a default value.

Make sure the **queryString** matches the path in the backend application.

The variable must have REACT\_APP before it.

When you’re ready to use the application, make sure the backend is running with **python main.py**, and also start the frontend with **npm start**.



When you search using the frontend, you may get an error like this in console, despite postman having no issue fetching what you want. The server has blocked the request by CORS policy.

**What is CORS**Cross-Origin Resource Sharing

Our Frontend is running on the local IP address 127.0.0.1 with port 3000, but the API, whilst sharing the same local IP address uses port 5050.

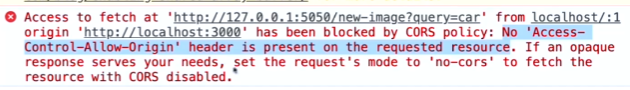
The web browser makes a request to the frontend application, and connects to port 3000, which is calls the origin (what you entered in the web browser).

When a web browser attempts to access other resources located at other URLs, then this is called a **Cross-Origin Request**.

The reason for the browser blocking the cross-origin request is that our API application is not configured to allow Cross-Origin requests.

However, the request we made from the frontend application to Unsplash API was also a cross-origin request that didn’t get rejected.

**So, what is the difference between the two.**

**Comparing Responses from Flask API and Unsplash API**

No ‘Access-Control-Allow-Origin’ header is present on the requested resource.

This means that the web browser expected to see a header called **Access-Control-Allow-Origin** in the response from the server, but the Flask server did not set such a key in the header of the response.

This response is from a successful query from the frontend app only.

The **access control allow origin** \* means that the Unsplash server allows any other origins to access it. You can configure your web server to only allow specific origins, but most cases use **any**.

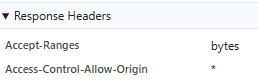
With such a header in place, the web browser will process actual responses from the server and will generate any cross-origin error.

In addition, **access-control-allow-methods** also specifies which methods are allowed for a particular URL, and this is set by the server.

We must modify our Flask application to enable CORS, after which the Flask server will set the **access-control-allow-origin** header in the response to the clients, which will no longer be rejected.

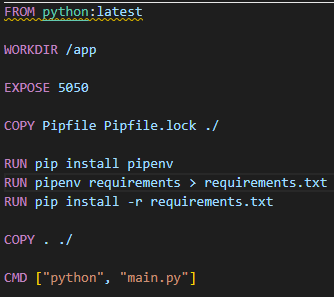
**Enabling CORS in the Flask App**<https://pypi.org/project/flask-cors/>

After importing CORS, you can add it as a wrapper around **app**. Then, if you use the frontend app, it should now successfully use Flask to query the API.

In the case of this app, we see that there is a \* as the value to **Access-Control-Allow-Origin**.

This accepts all URL endpoints.

**Storing Image Data in a Database**This will allow us to refresh the app and avoid the images disappearing when we refresh the application.

**Dockerfile**

**FROM** **python:latest** – uses the latest official Python image from Docker Hub as the base image. This contains Python and Linux so you can run Python applications in a consistent environment.

**WORKDIR /app** – sets the working directory inside the container to **/app**. Any subsequent commands (**COPY, RUN, CMD**) will be run from this directory, which helps organise files.

**EXPOSE 5050 –** Informs Docker that the container will listen on port 5050 at runtime. This doesn’t actually publish the port (you still need **-p** in **docker run**).

**COPY Pipfile Pipfile.lock ./** - copies Pipfile and Pipfile.lock from your local machine to the container’s working directory (**/app**). These files define your app’s Python dependencies (similar to package.json in Node.js).

**RUN pip install pipenv –** installs pipenv, a Python dependency management tool. It is used to manage your app’s dependencies using the Pipfile and Pipfile.lock files.

**RUN pipenv requirements > requirements.txt –** converts locked dependencies in Pipfile.lock into a requirements.txt file. While Pipenv is used for development, Docker containers often prefer **pip install -r requirements.txt** for simplicity and speed.

**RUN pip install -r requirements.txt –** Installs Python packages listed in the generated requirments.txt using pip. This step installs your app’s dependencies in the container environment.

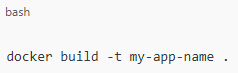
**COPY . ./** **-** copies the entire contents of your local project directory into the container’s working directory. This makes you application code available in the container.

**CMD [“python”, “main.py”] –** defines the default command to run when the container starts. In this case, it runs **python main.py** ,which is the entry point of your application.

**Building a Docker Image for the API Service**Navigate to the working directory (api), and run **‘docker build .’**

When you run **docker build .**, docker starts building an image using the Dockerfile in your current directory.

Afterwards, you now have a Docker image created with all your app code and dependencies. Docker will tag it as **latest** unless you specify a name/tag using:

****

Once built, you can run the container with:



This starts a container from your image and maps port 5050 on your machine to port 5050 in the container.

**Running the Docker container**

**Docker run –** starts a new container

**-p 5050:5050 –** maps ports between your host machine and the container

**Images-gallery-api –** the name of the Docker image you previously built

If you want to run another container on ports 5050:5050, you won’t be able to and will be met with an error. This is because port 5050 is already taken up by a container so cannot host another.

You can also run **docker run -p 5051:5050 images-gallery-api**. This will mean you are running the container at port 5050 in your host port 5051.

Each container runs in isolation and has its own port 5050. This is a common technique when running multiple instances of the same app (or different apps using the same internal port).

You are effectively running two separate backend instances of the same app/image, each in its own isolated container.

Each has its own process, memory, and environment, can serve independent requests, listens on its own host port, and can even be configured differently if needed (e.g. by passing different env vars or mounting different volumes).

This means you could use them for load testing, development vs production, or A/B testing. You could also route traffic to them selectively (e.g. with Nginx, Traefik or Docker Compose).

**Traefik**Traefik is a modern reverse proxy and load balancer designed specifically for containerised environments like Docker. It sits in front of your backend containers and acts as a smart router.

* Receives incoming HTTP/HTTPS requests
* Routes them to the appropriate backend container (based on hostname, path, port etc…)
* Supports load balancing, SSL, middleware and service discovery
* **Reverse Proxy –** forwards frontend requests to backends
* **Load Balancing –** distributes traffic across multiple backend containers
* **Automatic HTTPS –** integrates with Let’s Encrypt
* **Auto-Discovery –** detects containers via Docker labels (no static config)
* **Dashboard –** visual interface to monitor routes and services

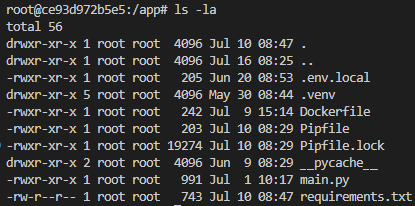
**Why use Traefik?**

* You want automatic routing as containers start/stop
* You want to avoid hardcoding ports in your frontend
* You want clean URLs (**/api/images** instead of [**http://localhost:5050**](http://localhost:5050))
* You need **SSL/TLS** or load balancing

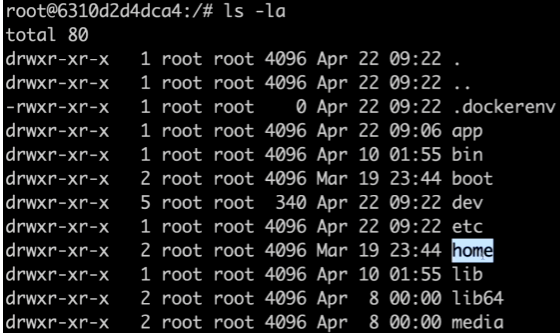
**Exploring Docker Container**

Running **docker ps** will reveal the running container. You can then run **docker exec -it amazing\_murdock bash**, which will open bash inside the container.

Here, you can see the files that have been copied from the **api** folder in the project.

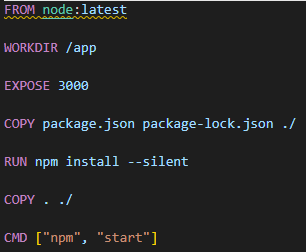
If you run **ls -la** inside this container, you can see all of the files inside **app**, including hidden files such as **.venv** and **.env.local**.

These should not be made public, so should be added to a Dockerignore file.



If you navigate outside of the **app** folder, you can see all of the different folders inside your container.

You can see **app** is one of many. This is why it is important to create **workdir** **app**, as otherwise all of the image files would be lost amongst all these other files.

**Creating and running the frontend docker image**

Using **node:latest** can lead to unexpected results if the image is updated and breaks compatibility.

**Expose 3000** tells Docker that the app inside the container will listen on port 3000. It does not actually open the port, but is mainly for documentation and can help tools like Docker Compose.

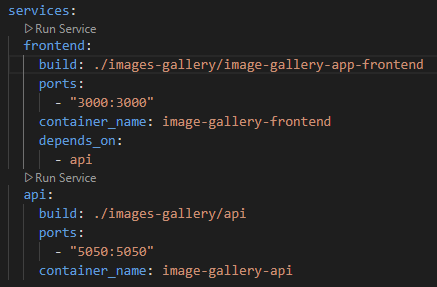
Copy the package files to define the Node.js project dependencies.

**--silent** supresses most of the log output, which is optional for cleaner builds.

**COPY . ./ -** copies the rest of your app’s source code from your local directory into the container’s **/app** directory. Happens after dependencies are installed to leverage Docker layer caching.

**CMD [“npm”, “start”] –** tells Docker to run npm start when the container launches.

**Creating a Docker Compose file**This allows you to start multiple containers using just one command, **docker compose up**. It is also possible to build or re-build custom images for particular containers before their start.

****The **Docker Compose** file:

1 – Builds **frontend** from its container name and maps to port 3000.

2 – Builds **api** from its container and maps to port 5050.

3 – **Frontend** depends on **api**, which will therefore be built first.

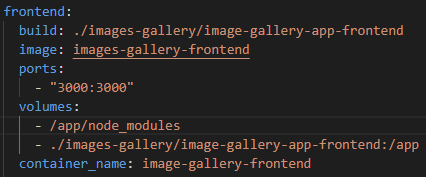
**Running containers with Docker Compose command**Using the command ‘docker compose up’, so long as the Docker Compose file has no issues, the docker engine will start running and building the containers.

**Docker Desktop Issues**

* Windows + R
* **services.msc**
* Docker Desktop Service 🡪 Start

**Updating changes in real time**When running in Docker, we are essentially running static images, so any changes we make to the code/visuals is not reflected when we refresh the local server.

We want this feature enabled, which is possible by using volumes.

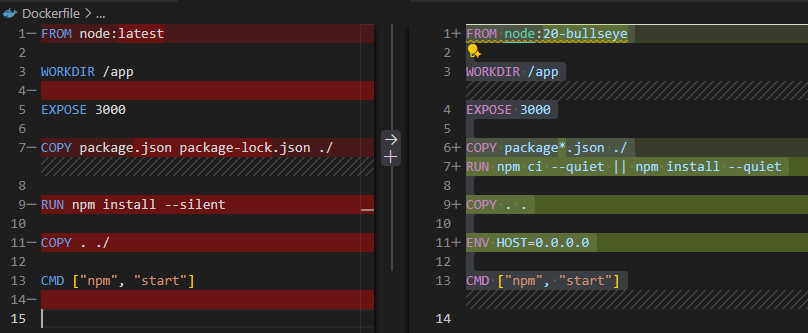
Volumes are matched to the containers, not images, so the images remain unchanged.

We can map our local frontend folder on our computers to the app folder inside the container. After this instruction, the contents of the app folder inside the container will be overwritten by the contents of the frontend folder, and immediately reflected inside the container.

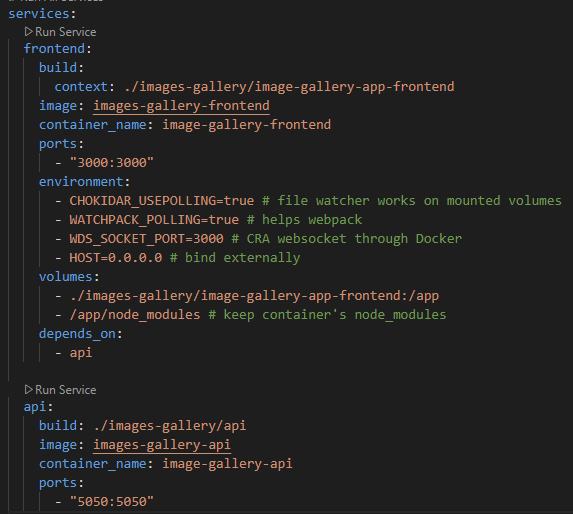
We also need to add **/app/node\_modules** to ensure that we keep the node modules folder inside the app folder inside the container. Contents of this node modules folder will not be overwritten by the contents of the node modules folder from the frontend. This ensures live code reloading works, and prevents deletion of installed node modules in the container.

**What happens if node modules in the container gets overwritten?**

1. **Missing dependencies –** the container’s node modules folder gets replaces by an empty or mismatched one from your host machine. The app can’t find the required packages and starts with errors such as: **Error: Cannot find module ‘expess’**.
2. **Platform Mismatch –** if you had node modules on your host machine and mounted it into a Linux container, native modules may be incompatible, resulting in errors such as **Module did not self-register properly.**
3. **Hot Reload Breaks –** your app might reload but crash instantly due to the missing or broken dependencies.

**Updating Dockerfile to allow for hot reload**

* **Base image –** pinning to bullseye, makes it more stable and easier to debug. Sticking with **latest** can result in issues and surprise breakages.
* **Dependency install –** easier way to grab both package and package-lock files. **Npm ci** is faster and installs exact versions listed in package-lock. It will fail if package-lock is missing or doesn’t match the package.json exactly.  
    
  Builds are fast, reproducible and deterministic.
* **Network/dev server binding –** ENV HOST=0.0.0.0 ensures the dev server listens on all interfaces so your browser on the host can reach it **(important for hot reload)**.

**Updated docker-compose for hot reload:**

**Environment variables:**CHOKIDAR and WATCHPACK – make file watchers work even when file system events don’t propagate well in Docker.

WDS\_SOCKET – fixes WebSocket issues with Create React App dev server.

HOST=0.0.0.0 – makes the React dev server accessible from outside the container.

**Volumes order –** the order matters because later mounts override earlier ones. In this version, you mount your source code into **/app** and protect **node\_modules** with an anonymous volume. If it were the other way around, the code mount **(./…:/app)** overwrites **/app** completely, including the **node\_modules** directory.

**Environment Variables Explained**

* **CHOKIDAR\_USEPOLLING=true –** Chokidar is the file-watching library used by Create React App and Webpack Dev Server.  
  Normally, file watching uses OS-level file system events (fast and efficient), however, in Docker (with mounted volumes), those events sometimes don’t propagate correctly.  
  Setting this forces Chokidar to use **polling** instead, which checks periodically for changes.  
  **This ensures hot reloading works reliably when you edit frontend files on your host machine.**
* **WATCHPACK\_POLLING=true –** Watchpack is another file-watching utility that Webpack relies on. Like Chokidar, it can miss file changes inside Docker volume mounts.  
  This variable tells Webpack to also use polling instead of native events.   
  **This provides redundancy – helps endure Webpack’s rebuild triggers on file changes.**
* **WDS\_SOCKET\_PORT=3000 –** CRA’s Webpack Dev Server (WDS) uses a WebSocket connection between browser and server for hot module replacement (HMR).  
  By default, this socket may try to use the container’s internal networking, which can break when running inside Docker.  
  Setting this ensures that the socket connection uses **port 3000**.   
  **This prevents hot reload issues where the frontend builds but the browser doesn’t update.**
* **HOST=0.0.0.0 –** By default, CRA’s dev server binds to localhost (inside the container). This would only be accessible from inside the container itself, not from your host machine.  
  Setting **HOST** tells it to listen on **all network interfaces**, making the app accessible from outside the container (via localhost:3000 on your host).  
  **Without this, you might not be able to access the app from your browser.**

**Chokidar vs Watchpack**When you edit a file:

1. Webpack (via Watchpack) needs to notice – rebuild the bundle.
2. CRA (via Chokidar) needs to notice – tell the dev server and HMR system to push the changes to the browser.

If either one misses the change:

* If Watchpack misses it – your code never recompiles, so nothing updates.
* If Chokidar misses it – your app rebuilt, but CRA never pushes the update to the browser.

They both must be in polling mode inside Docker as they’re covering different parts of the chain.

**What is a bundle?**When you write a React app, you have many files (e.g. App.js), assets (CSS, images, fonts) and modern JavaScript features.  
The browser can’t just read it all directly, so Webpack takes everything and produces a bundle:

* A small set of files that the browser can understand and load.
* All of your code and dependencies get merged, transformed, and optimised into that bundle.

**In your case,** when you edit a file in your project:

* **Webpack** (with Watchpack) sees the change and rebuilds the bundle.
* **CRA** (with Chokidar) makes sure the dev server reloads the browser so it gets the new bundle.

**How to check inside the container in command prompt:**

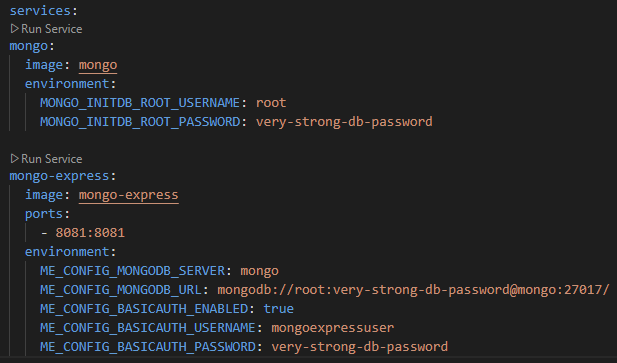
1. Open Powershell or Command Prompt
2. Run: **docker exec -it [name of image] sh**
3. **cd /app**
4. **ls**
5. **cat [name of file]**

This will enable you to check the contents of the container image from command line. When you update a file, you can run **cat [name of file]** again to check if your changes have been received by the image.

You can also check to see if it’s been updated in **localhost:3000** or **localhost:5050** depending on where you’ve sent the frontend or backend.

**Why we need Mongo**Currently, we use the frontend to query the API which connects with unsplash API. Whenever we restart the frontend, we lose the images we retrieved.

We need to save the images to a database, which the frontend accesses through Flask API. We can use the Mongo Express GUI to view the database outside of the CLI and to insert records directly if we want.

The values **USERNAME** and **PASSWORD** are needed so that other apps like mongo-express can connect securely.

You’ll be able to access the mongo-express UI at [**http://localhost:8081**](http://localhost:8081)

**ME\_CONFIG\_MONGODB\_SERVER: mongo –** tells mongo-express to connect to the mongo service defined above.

**ME\_CONFIG\_MONGODB\_URL –** full connection string with protocol username, password, host and port.

**ME\_CONFIG\_BASICAUTH\* -** enables basic authentication for the mongo-express web UI, so you need the username and password to login. This protects the dashboard from being publicly accessible with no password.

**Start container images without logs**Docker compose up -d

**Using MongoDB Shell and Mongo-Express GUI**

*docker exec -it full-stack-project-react-and-python-mongo-1 mongosh -u root -p very-strong-db-password*

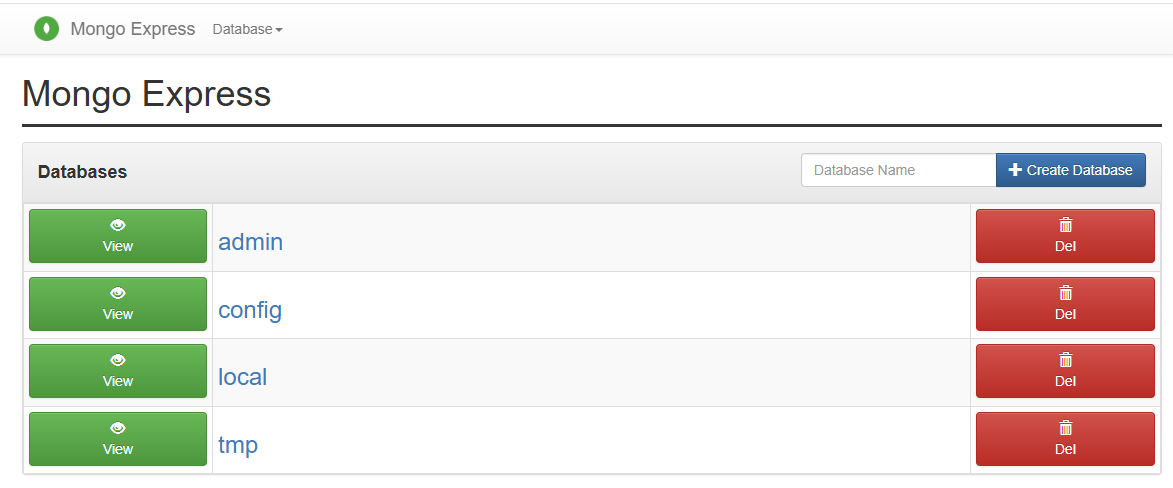
By running this, you can enter the mongo shell, where you can perform different MongoDB operations:

*show dbs*

You can see that there are 3 default databases. You can also switch to a particular database and analyse collections there.

*use admin*

*show collections*

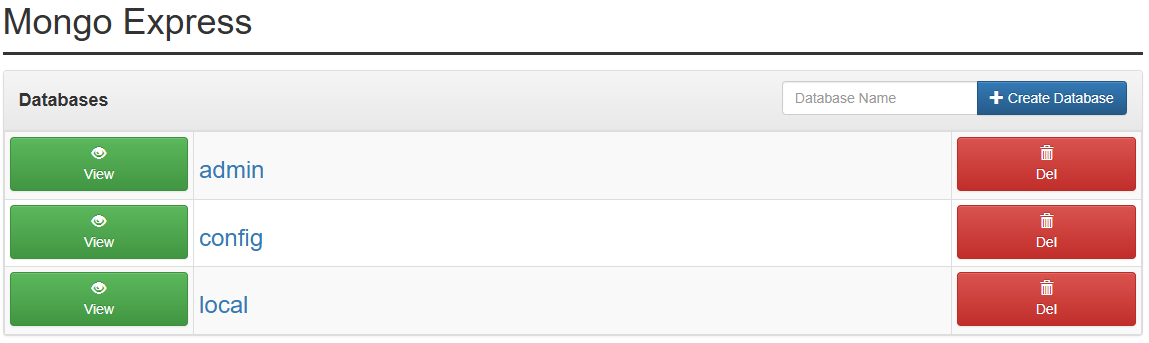
This navigates to the **admin** database and then displays the collections within it. We could create new test databases here inside of the mongo shell, create new collections and insert some sample documents, but this isn’t convenient as it’s in the CLI.

Mongo Express works very similarly to Compass, but with a worse interface. You can create new databases, collections and insert data. If you wanted to go back to the shell and view the persons collection inside **tmp**, you could:

*use tmp*

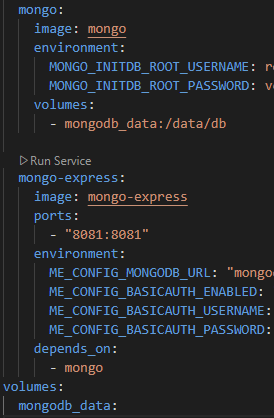
*db.persons.find()*

This would display all of the entries stored in this collection of this database.

**MongoDB Data is Deleted After Docker-Compose Restart**

When you run **docker compose down** and then **up**, you create brand new containers and the old containers are removed. This is why the data is removed after running **down**.

To resolve this, we need to add **volumes mapping**. Any container that is started using this mongo image will utilise the same volume, making the MongoDB data persistent.



In **volumes** at the bottom of the yaml file, we create a custom volume inside Docker.

We then map the custom volume to the **data/db** folder inside the **mongo** container.

Any container that is started by docker-compose will be connected to **mongodb\_data** volume.

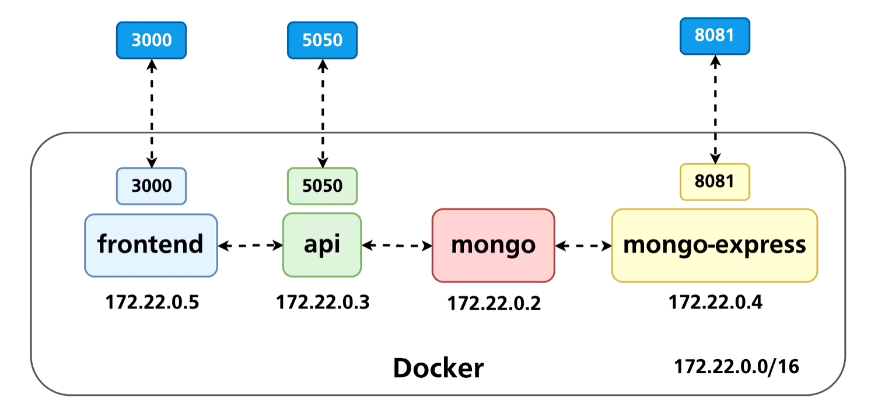
Any data that is created by one container inside this volume will be reusable by other containers that are connected to the same volume.

*The volume declaration at the bottom tells Docker Compose to ‘create or reuse if already exists’ a named volume called* ***mongodb\_data****.*

***Mongodb\_data:/data/db –*** *this line mounts the named volume inside the mongo container at* ***/data/db*** *(where MongoDB stores its database files). This means that your database files are stored* ***outside the container’s filesystem****.*

*This means that if you stop/recreate the container, your data survives. If you didn’t use a volume, the data would disappear when the container is removed.*

**Communication Between Docker Containers**Docker-compose creates a bridge network automatically.

A bridge network is a site-specific range of IP addresses.

For example, you might see a range such as **172.22.0.0/16.**

16 is the network mask for this particular network and each container that is included in the docker compose file gets a unique IP address from this range.

There is also a default gateway for each container, which usually ends with 1 in the last octet. It is the first usable IP address in the network.

The fact that Mongo doesn’t have a port does not stop the other containers from communication with it. They all belong to the same network and can communicate with each other using their IP addresses.

**Exploring Networking Between Docker Containers**By typing ***docker network ls****,* you can see which network was created for the containers. It will tell you the **network id**, **name**, **driver**, and **scope**.

By typing ***docker network inspect [NAME/ID]***, you’ll be able to see the subnet and the gateway numbers:

* **Subnet –** range of IP addresses that Docker assigns to containers connected to that network. Subnets ensure containers on the same network can communicate with each other directly.
* **Gateway –** the IP address inside the subnet that Docker assigns to the **bridge interface** of the network. Containers use this gateway to reach outside their network.
* **Bridge Interface –** the virtual router inside your computer that connects all the containers together and provides the gateway.

Each container gets an IP from the subnet. They all connect to the bridge. The bridge holds the gateway address and allows them to talk to each other and the outside world.

Further down the output, you can see information about other containers

By typing ***docker network inspect [NAME] | findstr IPv4Address****,* you can see the list of IP addresses for the containers of that particular network (e.g. frontend).

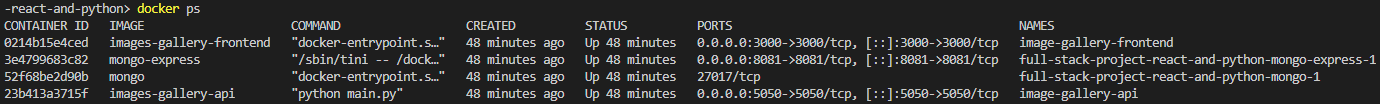
For example:

**Full-stack-project-react-and-python\_default** is simply the name of the Docker network that was auto-created by Docker Compose for your project.

It is not a container or an image, but a network that links all your containers together:

* Frontend
* Api
* Mongo
* Mongo-express

In Docker Desktop, it looks like there’s only 1 container running, when reality it’s the UI grouping together the 4 different containers.

By typing **docker ps**, you can view information on all the containers you have running:  
By typing ***docker exec -it images-gallery-api sh***, you can dive deeper into a specific container:

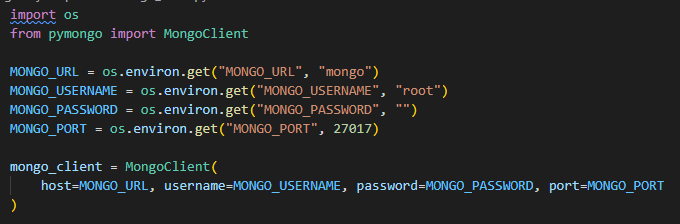
* **Hostname –** returns the container ID of the container you’ve entered
* **Hostname -i –** returns the IP of the container you’ve entered

**What is Ping in Docker?**Ping is a way of asking “Hey, are you there?” over the network. If you give it a name like **mongo**, or and IP address (e.g. *ping mongo*), it will send a tiny test message. If the other side replies, you know it’s reachable.

Ping helps check if containers can see each other on the network. If **ping mongo** works, your API container can probably also connect to Mongo using its service name.

This means when you are in ***images-gallery-api*** *sh*, you can check if it is able to communicate with the other containers.

**What is Pymongo?**Contains tools for interacting with MongoDB from Python. It will enable us to save our API images to mongo as there is currently no speaking between the API and Mongo.

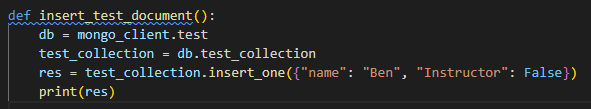
**MongoClient File**

**Os –** used to read environment variables

**MongoClient –** the main client class from pymongo that lets you connect to a MongoDB server.

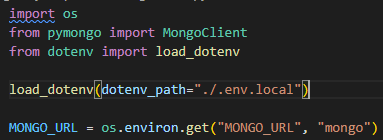
**MONGO\_URL –** defaults to “mongo” (often the service name in Docker Compose)

**MONGO\_USERNAME –** defaults to “root”

**MONGO\_PASSWORD –** defaults to 27017 (the standard MongoDB port)

This function inserts a simple test document into MongoDB.

1. Accesses (or creates if none) the database named **test**. In MongoDB, databases are created lazily; they don’t actually exist until you insert data.
2. Accesses (or creates) the collection inside the database. Collections in MongoDB are similar to tables in SQL, but schema-less (documents can have different fields).
3. Inserts a single document into the collection. The document here is a Python dictionary, which pymongo converts to BSON. MongoDB will automatically add an **\_id** field if you don’t provide one.

In the **mongo\_client** file, we need to import **dotenv**. This is because we have imported the mongo\_client into **main.py**, which is loaded before the dotenv line is executed, meaning we wouldn’t be able to access environment variables.

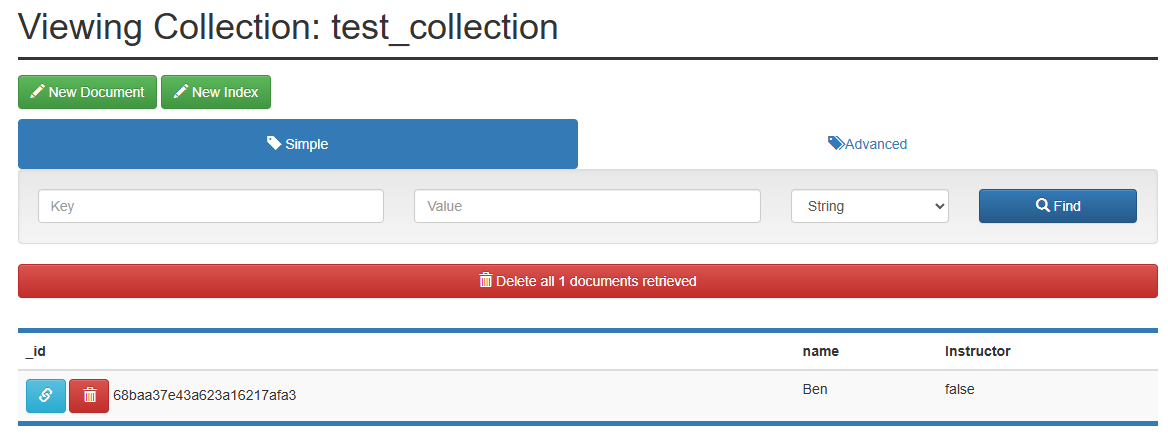
**Rebuilding services after changes**

*docker compose up --build -d api*

The name of the service (api), should be the same as what is specified in the docker-compose configuration file.

**Checking if a file has been inserted**

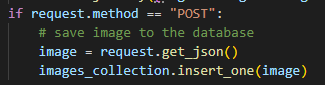
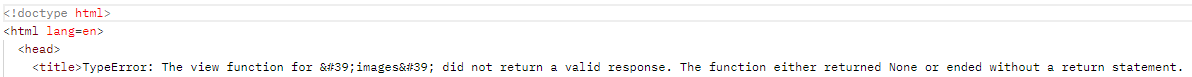
*Docker logs image-gallery-api*

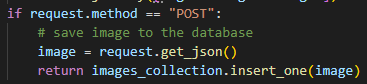


**Reusing Mongo Client**We will utilise the mongo\_client we created and import it into other modules. It does not make sense to create multiple instances of the client in different modules each time you require connection to MongoDB.

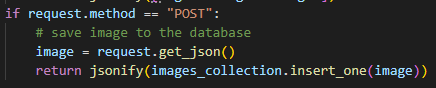
It makes sense to reuse only a single instance, which is why we created a separate mongo\_client module.

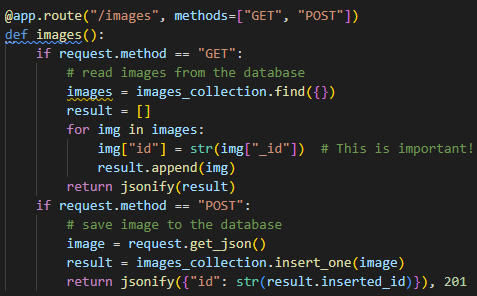
**API Endpoints**At the moment, there is already a new-image API endpoint with a single GET method available.  
We also need an endpoint with two methods, GET and POST, which will retrieve all images saved in MongoDB, but also save a new image to this collection.

******Adding ‘images’ endpoint and handling GET requests**When we try to insert data into the database using Postman, we get a 500 error, saying the function returned ‘None’ or ended without a return statement.

Other error messages could include a TypeError, where the type returned should be a string, dictionary or tuple.



Trying to jsonify the object results in another TypeError saying the object isn’t JSON serializable. This is because it isn’t clear how python can convert Mongo ID, **ObjectId(123456789)**, to a type such as string.

**@app.route() –** tells Flask whenever a request is made to ‘/images’, to use the function below to handle it. Allow both GET and POST requests.

**Images = images\_collection.find() –** queries the MongoDB collection for all image documents. Returns a cursor over all documents. A cursor is an object that lets you iterate over the results of a database query. In this case, **images** is now a cursor.

**Img[“id”] = str(img[“\_id”]) –** adds an **id** field to the image dictionary. Sets it to the string version of MongoDB’s **\_id** field (since **\_id** is an ObjectId, which isn’t JSON serializable). Now each image has an “id” field that’s JSON friendly.

**Return jsonify(result) –** converts the results list to JSON and returns it as the HTTP response. The client will receive an array of images with **id** fields.

HTTP APIs send data as plain text, not as Python objects. **Jsonify** converts Python data to a JSON string so that browsers, JavaScript, Postman etc… can understand. It also sets the response’s **Content-Type** header to **application/json**, so the client knows what to expect.

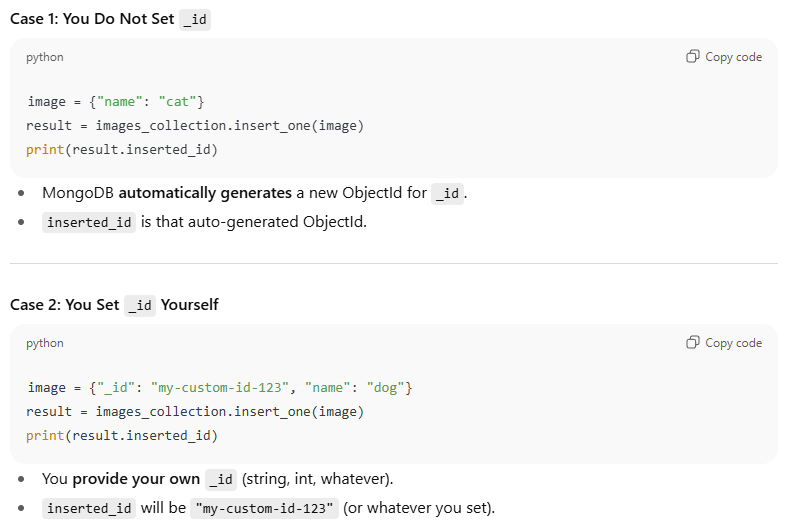
**Image = request.get\_json() –** reads the incoming JSON body of the request and parses it into a Python dictionary called **image**.

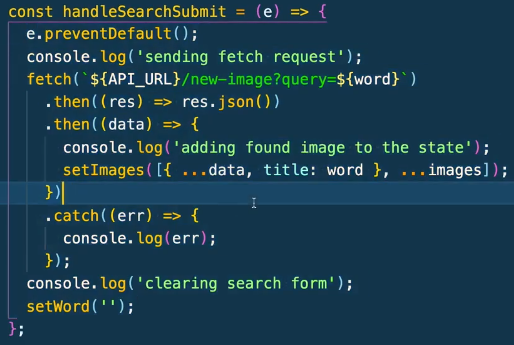
**Result = images\_collection.insert\_one(image) –** inserts the new image into the MongoDB collection. Insert\_one returns an object with information about the insert, including the new document’s **\_id**.

**Return jsonify ({“id”: str(result.inserted\_id)}), 201 –** returns a JSON response containing the new document’s **id** (converted to string). The 201 means ‘created’. It needs to be converted back to JSON after you’re done with the Python object.

You convert it back to JSON, but need to make sure the ID is string again.

**Result.inserted\_id** is not a method, but a property of the object returned by **insert\_one.** It is either the value set by MongoDB, or one you set yourself.

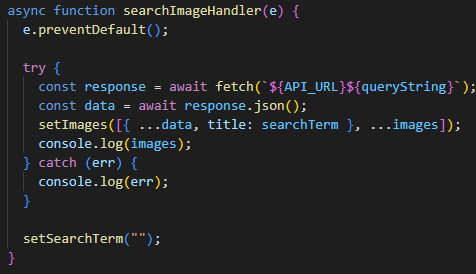


**Awaiting the API call**

In this API call, the console logs fire in this order:

1. Sending fetch request
2. Clearing search form
3. Adding found image to the state

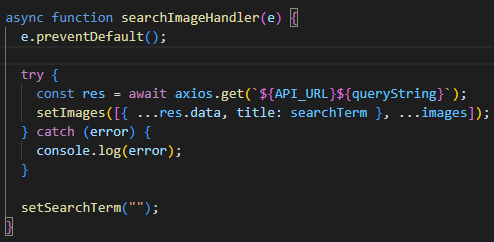
This means that using **fetch then catch**, does not wait for the processed result of the fetch request.



Instead, you can try-catch, which will wait for the response before executing code.

Older browsers don’t have as much ability to execute **await** statements, which may pose a problem sometimes.

**You can also use axios, which has other advantages.**

Using axios means you don’t need **response.json()** as it happens automatically.

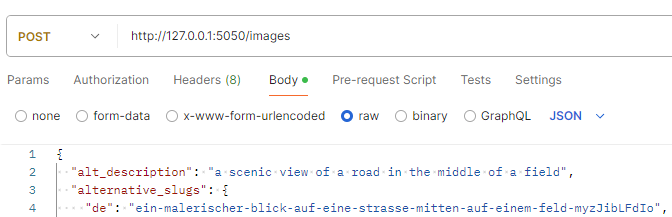
It throws errors for 404s or 500s, whereas Fetch only throws for network errors.

Axios has built-in features for timeout, cancel requests, set headers, interceptors etc…

Axios works on older browsers, whereas Fetch is not supported in some old browsers.

Axios lets you run code before a request is sent or after a response is received (good for auth tokens, logging).

**Using Postman to Insert Image Objects**

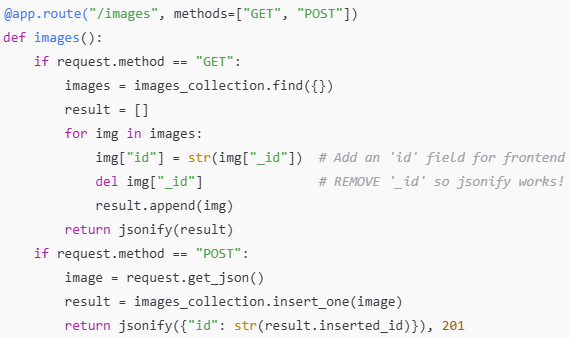
****Select **POST** 🡪 Enter the images endpoint 🡪 In **Body**, select **raw** 🡪 insert image object 🡪 if successful, you’ll get a **201 CREATED** response.

**Retrieving Saved Image when App Loads**

If you’re getting a **network error** when trying to fetch images from the backend, meaning your API call to 5050/images fails, your **images** state is likely empty or undefined.

This could mean that your frontend and backend aren’t talking to each other (maybe one isn’t running), or another issue.

One issue encountered was:  
**TypeError – Object of type ObjectId is not JSON serializable**

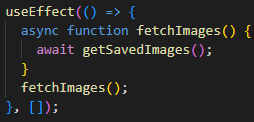
The backend is using MongoDb, so the images have an **\_id** field of type **ObjectId**. This crashes your backend endpoint **/images**, so the frontend gets a network error as the backend never sends a valid response.

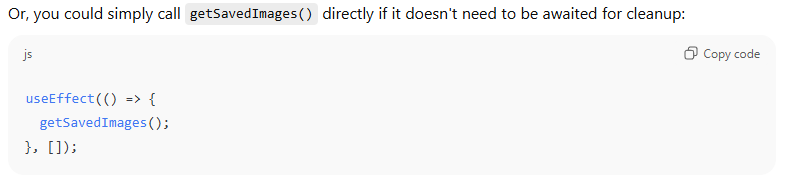
In the **for loop**, an **id** is inserted that equals the **id** of the **\_id** field but as a string.

The **\_id** field is then removed from the object so that the result can be converted into JSON.

E.g. **ObjectId(‘123456789’)** is converted into **“123456789”**.



****We use **useEffect** to fetch the images only when the page loads for the first time.

It must be an async function. For some reason, an async arrow function didn’t work here. This is the best way of doing it anyway.

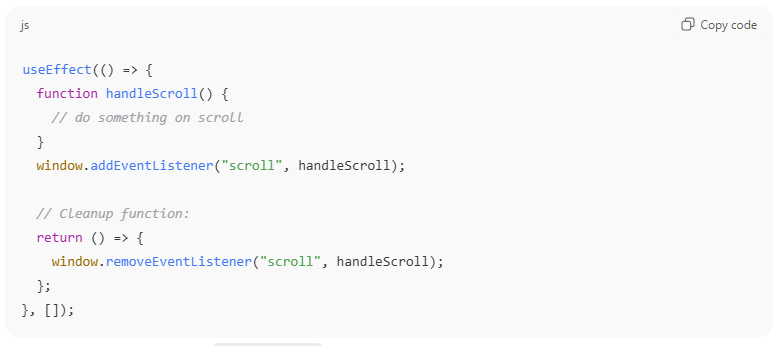
This option calls **getSavedImages()**, which is already an async function. Since **useEffect** doesn’t care about the returned promise (unless it’s a cleanup function), it just fires and forgets.



You will get an error if you have an image object without a title. You can’t convert **undefined** to upper case, so you need to make it optional and if it doesn’t exist, set it equal to an empty string.

This could also apply to other fields in the object that you use in your code, so **defensive programming** is always best.

**What is a cleanup function?**A special function inside React’s **useEffect**, which you can optionally return from your effect. React will call this function when your component unmounts, or before re-running the effect (if dependencies change).



In this example, a window event listener, React will run the initial function after the initial render.

The effect code runs once, or every time dependencies change. Every time you scroll, the handler you provided will run.

Cleanup happens when your component unmounts (e.g. navigation to a different page, or conditionally removing the component). If your dependencies change, React will call the cleanup function to update with the new dependencies.

**Saving Images in the Database from the Frontend App**

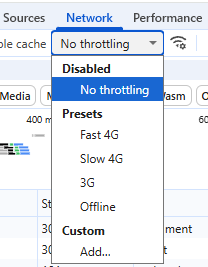
You can link this to an **onClick** event listener on a button, in this case ‘Save’.

The images array is scanned and the image found by its ID. It is then given a ‘saved’ property.

The image is saved to the database.

If the backend responds with an inserted\_id, update the images array in the React state.

**Simulating Slow Internet**

****You can simulate slow internet by navigating to the network tab and selecting one of the slow presets (e.g. 3G).

This allows you to judge the performance of the app when connection is slow, and will enable you to iron out any issues that arise with slow internet.

For instance, with slow internet, when fetching the images from the database, the welcome screen is displayed. We can replace this with a loading spinner to indicate that images are being fetched from the database.