****

**STUDENT:** ΙΟΑΝΝΙS TSALOUMAS

**MODULE:** AGILE & CONTINUOUS SOFTWARE ENGINEERING

**SUPERVISOR:** KONSTANTINOS MAKEDOS

**RN:** SEY24032

**EMAIL:** itsaloumas@athtech.gr

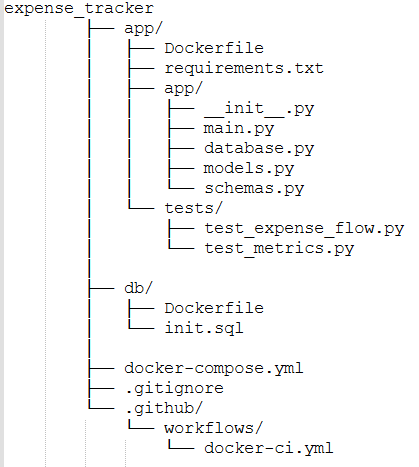
**Spring Semester 2024-2025**

In this project—created to fulfill the course’s requirements—an expense tracker is developed using the Python programming language and, more specifically, the Fast‑API framework to implement all the endpoints supporting the required CRUD operations.

FastAPI is a web framework for developing APIs with Python, designed to be easy to use, quick to develop with, and highly efficient. One of FastAPI’s features is that it provides an interactive environment for testing and exploring endpoints via Swagger. It also offers Pydantic for data validation and parsing, leveraging Python’s type hints to ensure that the data flowing through the API is in the correct format.

Additionally, the SQLAlchemy toolkit was used, providing a flexible way to connect to the application’s back‑end database. SQLAlchemy is a widely used Python toolkit and ORM (Object‑Relational Mapper) that simplifies communication and interaction with databases. Through the SQLAlchemy ORM, Python classes are mapped to database tables, making it easier to work with databases in an object‑oriented environment.

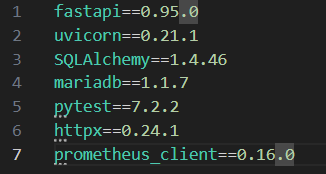
Let’s first take a concise look at the files in the Git repository. Below we can see the root of the folders and the files of the project.



* app/Dockerfile
* app/requirements.txt
* app/app/\_\_init.py\_\_
* app/app/database.py
* app/app/models.py
* app/app/schemas.py
* app/app/main.py
* app/tests/test\_expense\_flow.py
* app/tests/test\_metrics.py
* db/Dockerfile
* db/init.sql
* docker-compose.yml
* .gitignore
* .github\workflows\docker-ci.yml

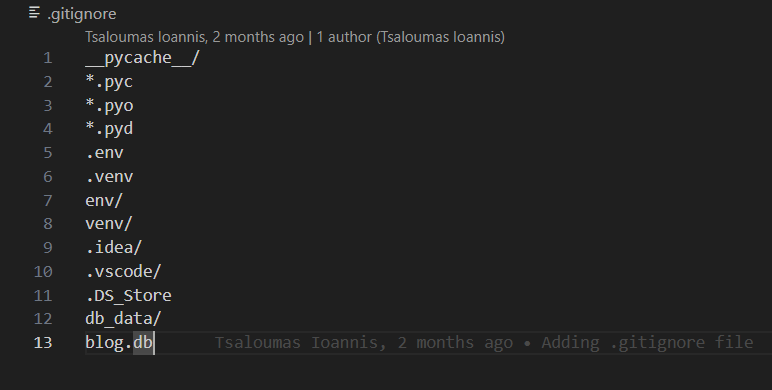
Let’s now examine in detail the contents of each of the files:

* **app\requirements.txt**



This file lists the versions of the various libraries that the system running the application must have. We supply these in the Dockerfile when building the image, so they’ll be available inside the container.

* **.gitignore**

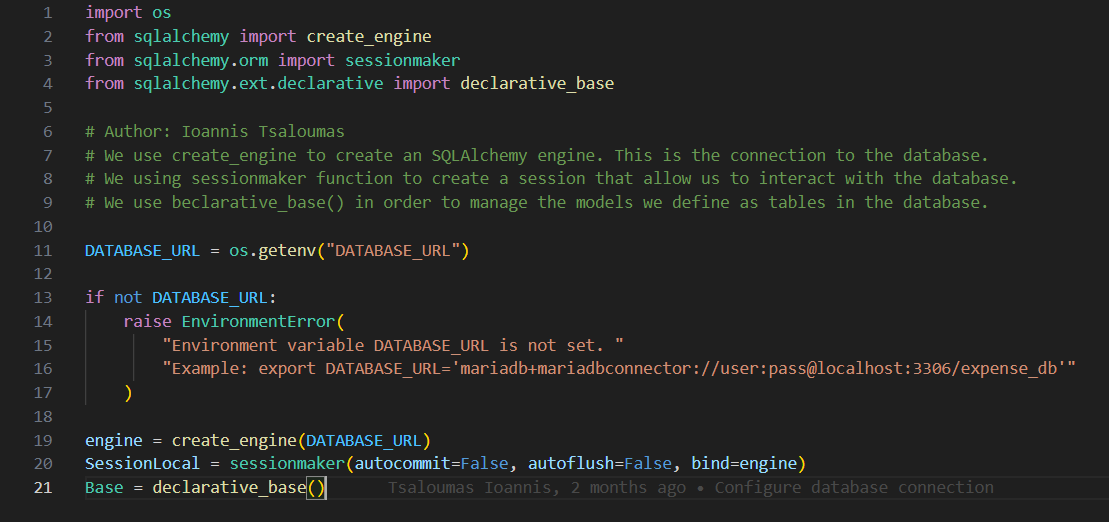


A .gitignore file lists filename patterns that Git should **ignore**—it tells Git not to track, stage, or commit any files or directories that match those patterns in order to keep e.g., virtual-env folders, build artifacts, secrets out of our repository so they don’t clutter history or leak to others.

* **app\app\\_\_init\_\_.py**

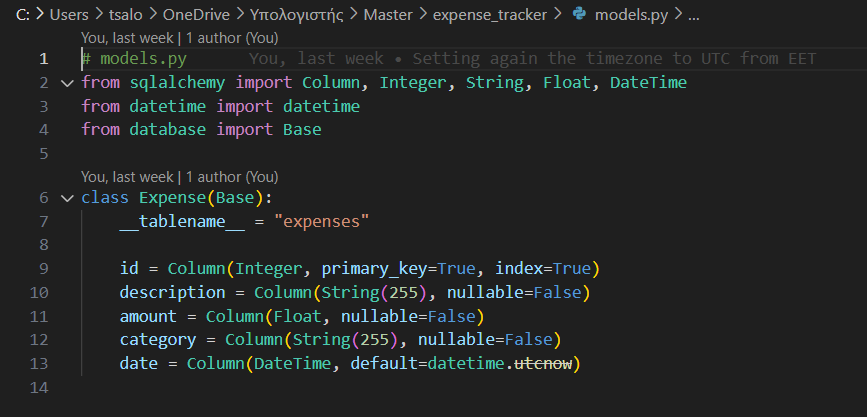
This is a file that remains empty and serves to turn the folder into a Python package.

* **app\app\database.py**



In database.py we set up the application’s communication with its database. Specifically, once we define the DATABASE\_URL (in our case read from an environment variable), we pass it into create\_engine to build the object that manages all connections to the database. Next, we declare SessionLocal via sessionmaker—each session acting as its own little “dialogue” with the database, where you can perform inserts, updates, and so on. Finally, we define Base, the “foundation” of all our ORM models (for example, an Expense class). By inheriting from Base, SQLAlchemy knows that our class corresponds to a table in the database.

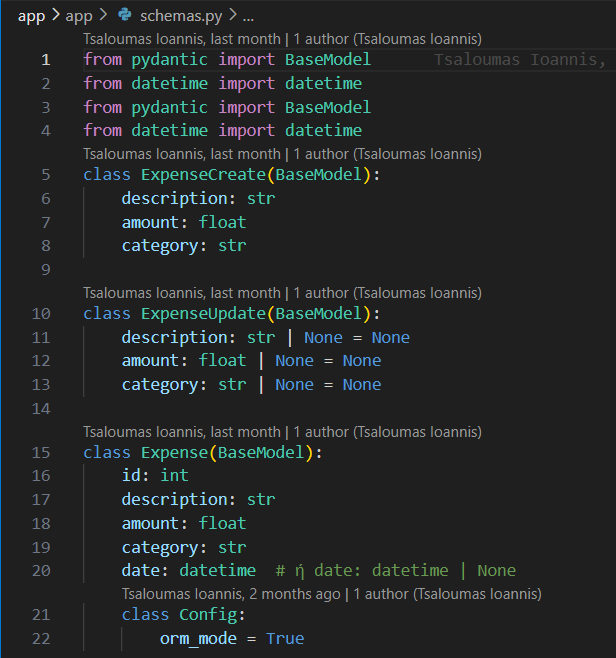
* **app\app\models.py**



In models.py we “sketch out” our table in Python—defining the ORM (Object‑Relational Mapping) model that mirrors the database’s structure. We declare an Expense class that inherits from Base (the very same declarative\_base() you set up in database.py), so SQLAlchemy instantly knows: “Ah, this is a table.” By setting \_\_tablename\_\_ = "expenses" we give our table its name, and then we list each column as a class attribute. SQLAlchemy can now take any Expense object you create in Python and translate it into a row in the expenses table, and vice versa.

* **app\app\schemas.py**

In schemas.py you’ll find the Pydantic models used to validate and serialize the data flowing in and out of the API. Pydantic ensures that any data received or sent via the API matches these types—for example, that description is a string, amount is a float, and so on.



The two classes, ExpenseCreate and ExpenseUpdate, are Pydantic schemas that define what data our API expects when creating or updating an expense, without needing to send the id or the date. In ExpenseCreate only description, amount, and category are included, which are required, and it is used in **POST /expenses** so that the user sends exclusively what is needed for a new expense. In ExpenseUpdate, the same fields are all optional, allowing a partial update in **PUT /expenses/{id}**, without having to refill the entire object.

With the Config class, Pydantic can read data directly from SQLAlchemy objects (ORM objects), and if we take an Expense object from the database, it can be easily converted into an Expense from schemas.py to be returned by the API.

* **app\app\main.py**

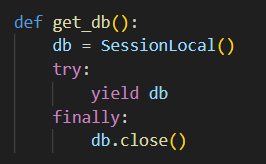
This is the main file of our application. After importing the necessary libraries, we create all the tables in the database based on the ORM models defined in models.py.



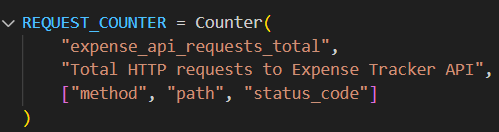
Base.metadata holds information about all the models—such as table structures, column names, etc.—while the create\_all() method updates or creates the tables in the database (if they don’t already exist), using the engine configured in database.py. Then we create an instance of the FastAPI application, which serves as the central “server” that will handle HTTP requests and route them to the appropriate endpoints.



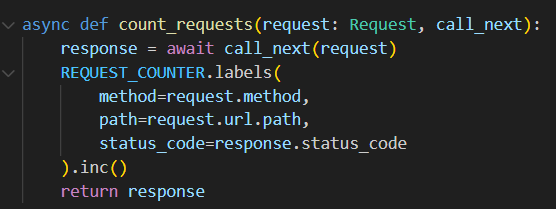
All the operations (endpoint definitions) that follow are added to this object.  
We define the get\_db() function:



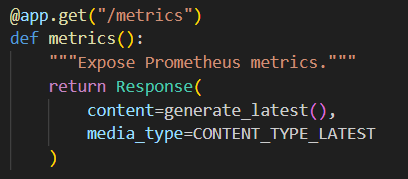
Essentially, it’s a generator that creates and returns a database session (via SessionLocal) for each request that requires access to the database. Αlso we have the request counter:



This defines a Prometheus counter metric named expense\_api\_requests\_total, used to track the total number of HTTP requests made to the Expense Tracker API. The metric helps monitor API usage over time. Then we have an HTTP middleware that intercepts every incoming request. It increments the REQUEST\_COUNTER before passing the request along, allowing all requests to be automatically counted as shown below:



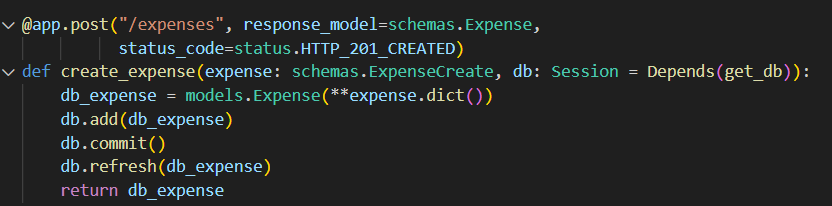
Also we have the /metrics endpoint that exposes the Prometheus metrics. When called, it returns the current values of all registered metrics in a format readable by Prometheus.



At this point we start to implement all the CRUD actions with the (following) relevant GET, PUT, POST, DELETE requests.

🡪Create

From the @app.post("/expenses") decorator on create\_expense, we designate that this endpoint handles POST requests at the /expenses path. The response\_model parameter indicates that the response will conform to the Expense Pydantic schema.

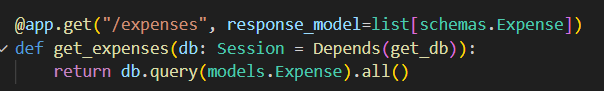


By setting the status code to **201 Created**, it signals that a new record has been successfully created.

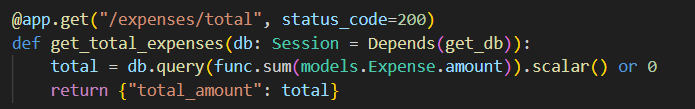
🡪Read

Five different endpoints that handle GET requests have been implemented.

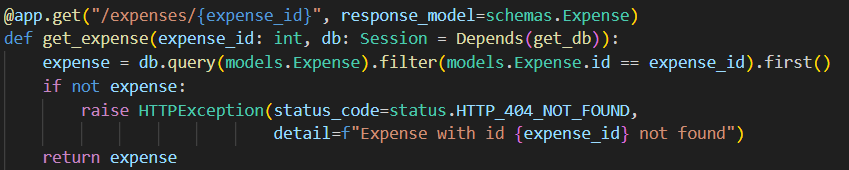
The first handles GET requests at the **/expenses** path and returns a list of Expense objects.



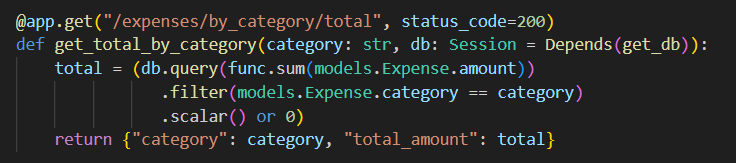
The second endpoint returns a list of the total expenses as recorded in the database.



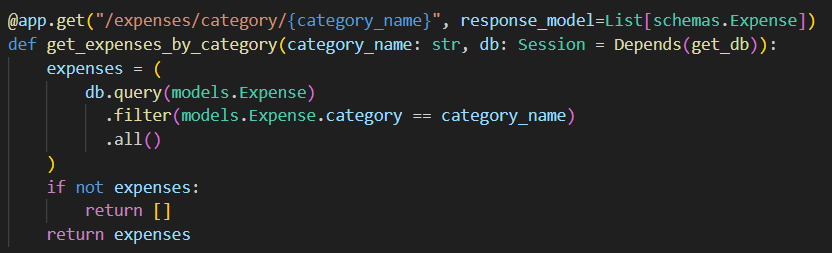
The third endpoint handles GET requests for a specific expense, based on its ID.



The 4th endpoint returns the tomal amount of all expenses for a given category.

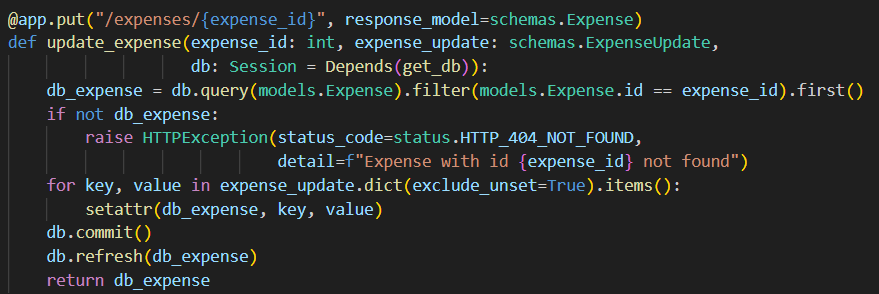


The 5th endpoint returns a list of all expenses for a given category.



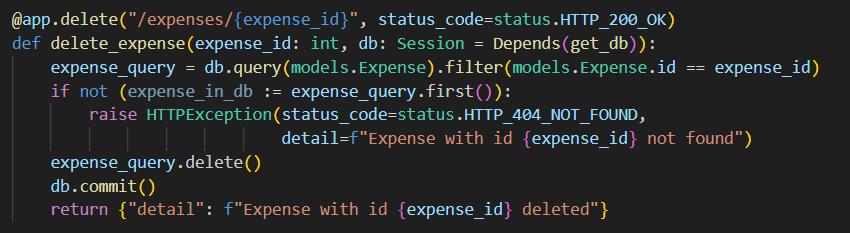
🡪Update

This endpoint handles PUT requests to update an existing expense. It expects the ID of the expense to be updated and the new data, then checks whether that expense already exists in the database—raising a 404 error if not. If it does exist, it locates the record and applies the update using the update() method with exclude\_unset=True, so that only the fields provided in the request are changed. Finally, it commits the changes, refreshes the object from the database, and returns it.



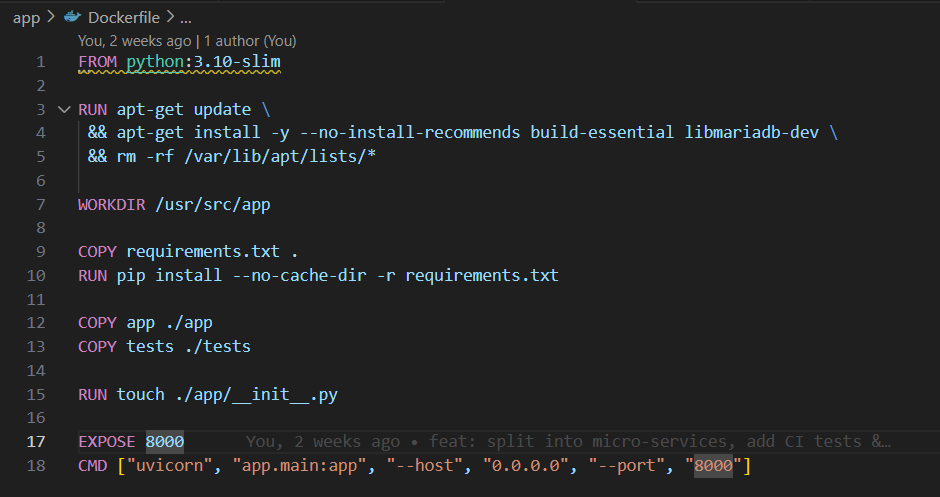
🡪Delete

This endpoint deletes an expense corresponding to the given ID. It takes the ID and performs a query to locate the record. If the record doesn’t exist, it raises a 404 exception; if it does, it deletes the record using the delete() method and then commits the transaction.

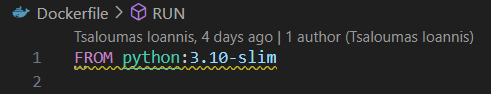


* **app/Dockerfile**

The Dockerfile in the /app project directory is used to define how the FastAPI application should be packaged into a Docker image. Here's the version of the Dockerfile:



Here’s an explanation of each section of the Dockerfile:



This line specifies the base image for our Docker image. In this case, we’re using the Python 3.10 slim image as the parent image.



We install build-essential and libmariadb-dev to enable the compilation of everything required by the MariaDB connector.



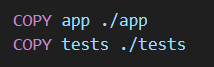
This line sets the working directory inside the container to /usr/src/app. Subsequent commands will be executed in this directory.



With this command we take the requirements.txt file from our build context (the folder where we actually run docker build) and places it into the container’s current working directory (which we’ve set to /usr/src/app with WORKDIR).



This command installs the Python packages specified in requirements.txt using the pip package manager. The --no-cache-dir flag prevents caching of the downloaded files, which can help reduce the image size.



With those commands we bring our entire local app/ directory into the image at /usr/src/app/app and the local tests/ folder into /usr/src/app/tests.



We create an empty \_\_init\_\_.py file in the app directory, ensuring Python recognizes it as a package.



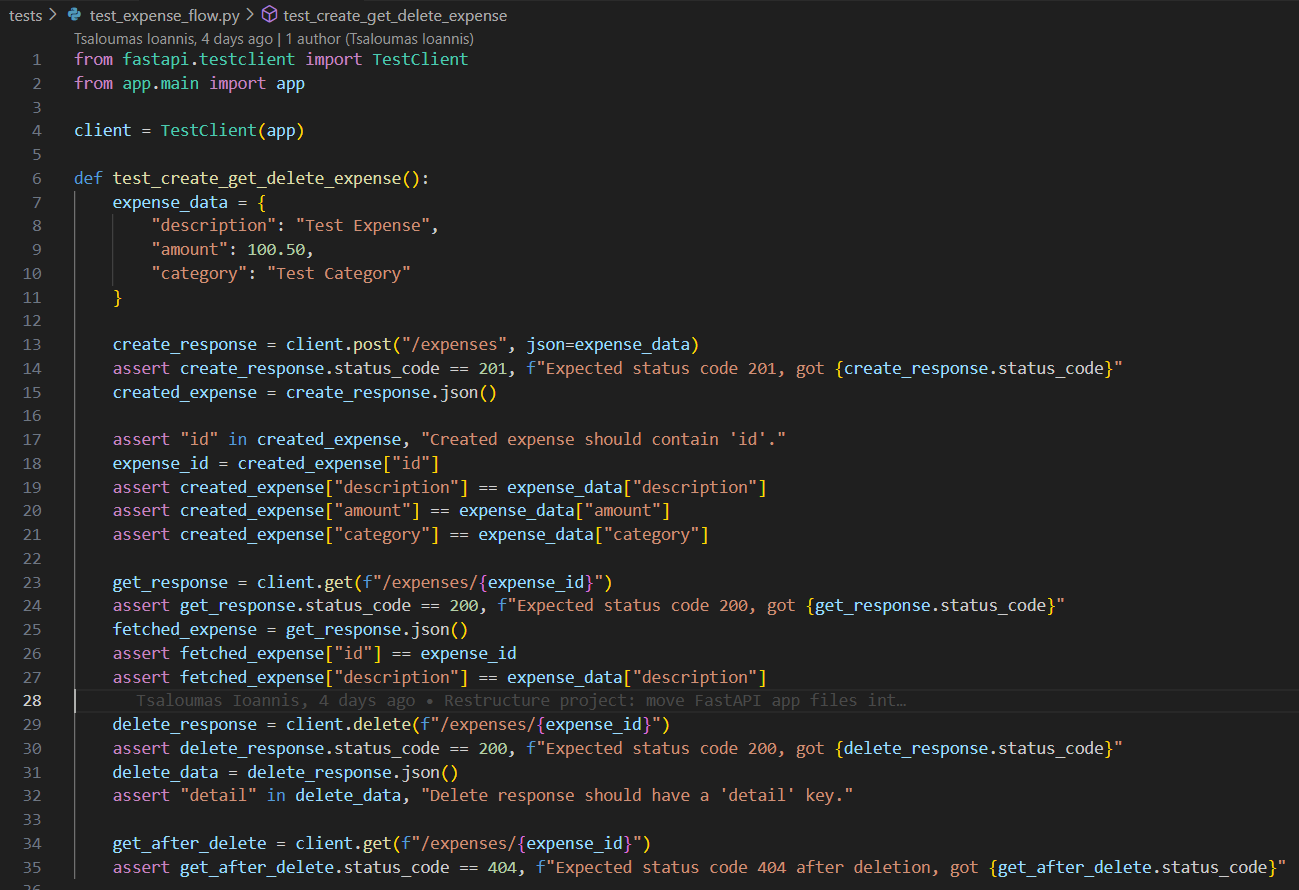
Here, we’re exposing port 8000 from the container. This allows communication to the application running inside the container over port 8000.



Finally, this command specifies the default command to run when the container is launched. It uses uvicorn (ASGI server) to run the main FastAPI application (app.main:app) on host 0.0.0.0 and port 8000.

Below we have the files under the app/tests folder:

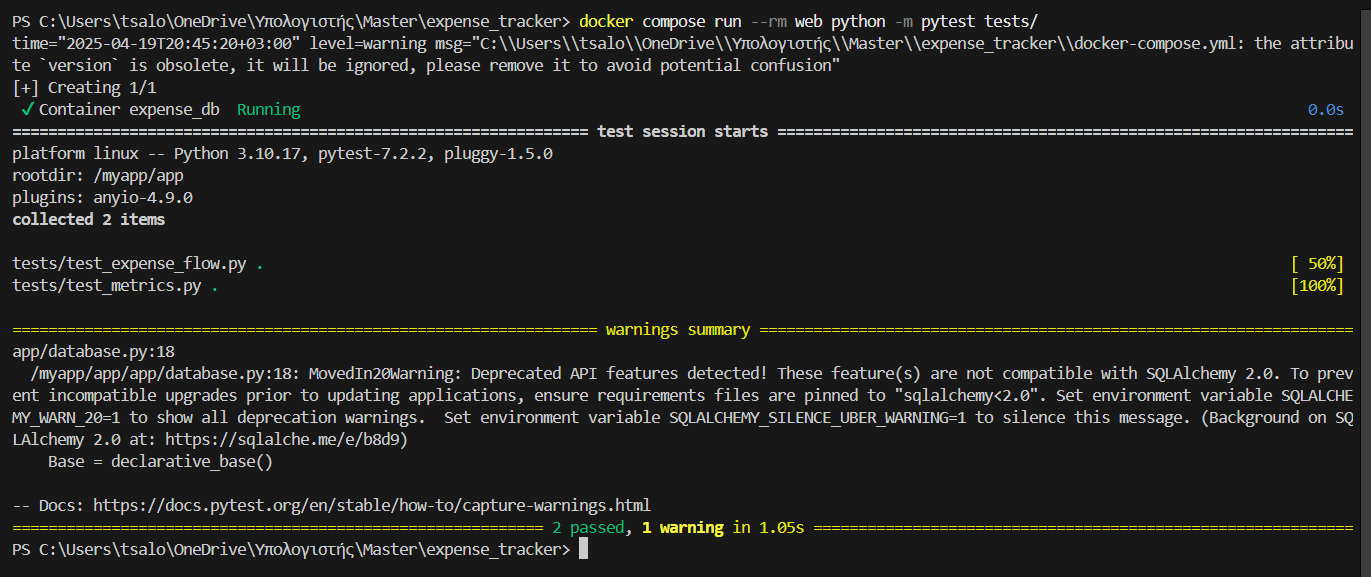
* **app/tests/test\_expense\_flow.py**



The small script spins up the FastAPI application inside an in‑memory TestClient, so we can send HTTP requests without standing up a real server. In the test\_create\_get\_delete\_expense test we first build an expense\_data object with description, amount, and category, we send a POST request to /expenses and assert that the server returns 201 along with the same data we sent.Next, we retrieve that same expense via GET using the ID returned by the creation step, verifying that the fields were indeed stored correctly. We then send a DELETE request to remove it, confirming we receive a 200 status code and a confirmation message.Finally, issuing GET again on the same ID ensures the record is gone for good, since the server responds with 404 Not Found.

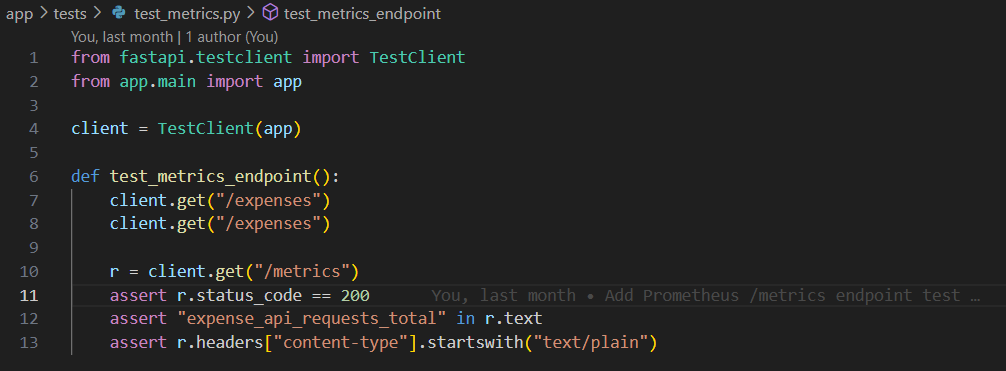
Open a terminal and type the command::

**>** docker compose run --rm web python -m pytest tests/



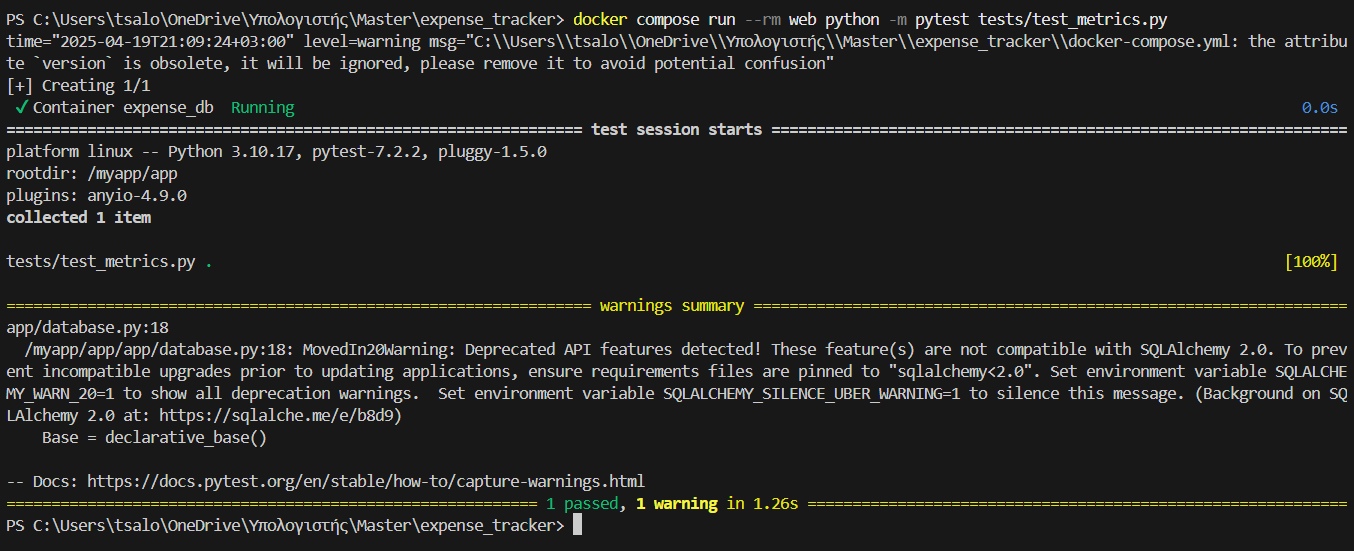
All tests passed (2 passed), but a single warning appeared, telling us that we’re using SQLAlchemy API features that will be removed in version 2.0 and recommending we lock our dependency to sqlalchemy<2.0.

* **app/tests/test\_metrics.py**

****

In this directory (app/tests) we also have the app/tests/test\_metrics.py. This test verifies the Prometheus /metrics endpoint in a FastAPI application. It uses TestClient to simulate two calls to the /expenses endpoint and then checks the /metrics response. The test ensures that the metrics endpoint returns a 200 status, includes the custom metric expense\_api\_requests\_total, and responds with a plain text content type. This helps confirm that API usage metrics are correctly exposed for monitoring.

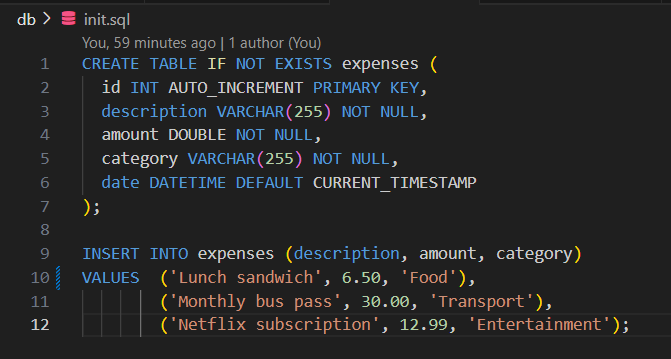
Likewise, for the metrics with Prometheus, below we see that one test was collected and passed successfully.



Below we have the files under the db directory. There are two files in this directory:

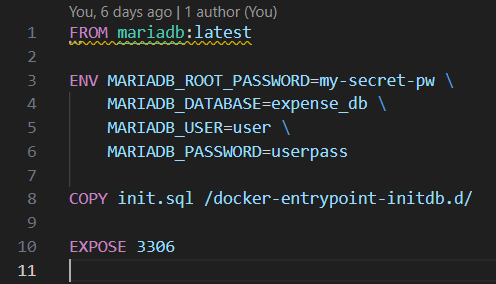
* **db/sql.init**

Under the db folder we have the sql.init file. In the sql.init file we create a table named expenses (if it doesn’t already exist). This table includes an auto-incrementing primary key id, and fields for description, amount, category, and date. All fields are required except date, which defaults to the current timestamp. This setup helps store detailed records of individual expenses. Also in this file we seed initial expenses for demonstration puproses.



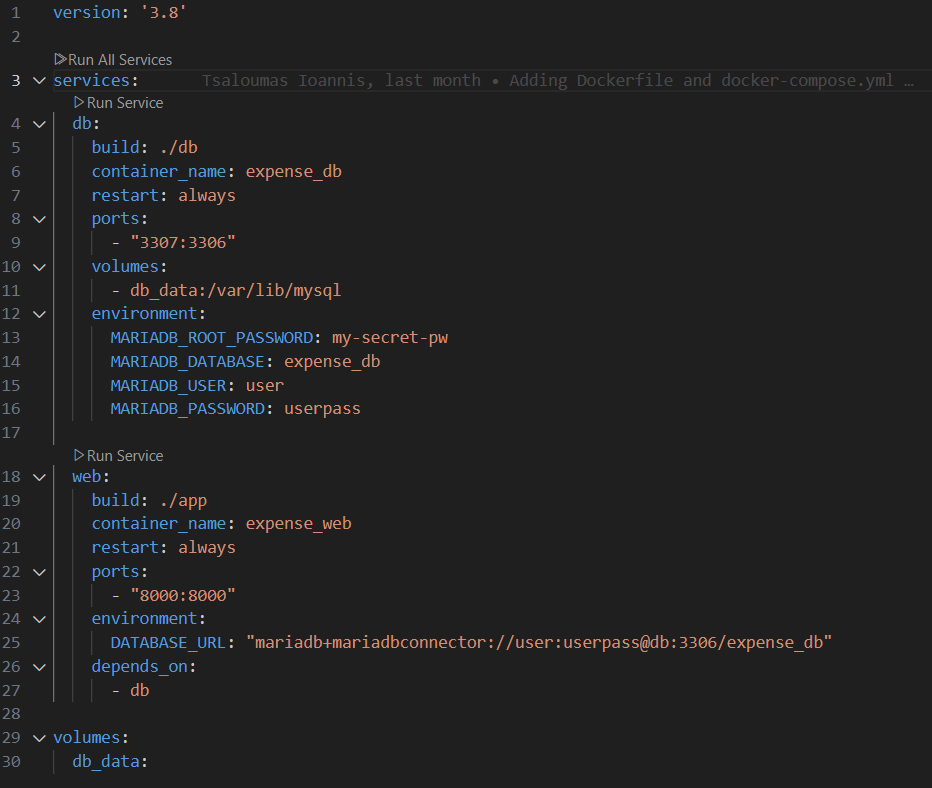
* **db/Dockerfile**

The other Dockerfile under the db folder is responsible for mariadb. In this file we set the **base image** for the container and we also set the **environment variables** used by the MariaDB image during initialization. Then we copy the custom SQL script inside the container to pre-load tables, insert initial data etc. Finally in this file we tell Docker that the container listens on port 3306 (the default MariaDB port).



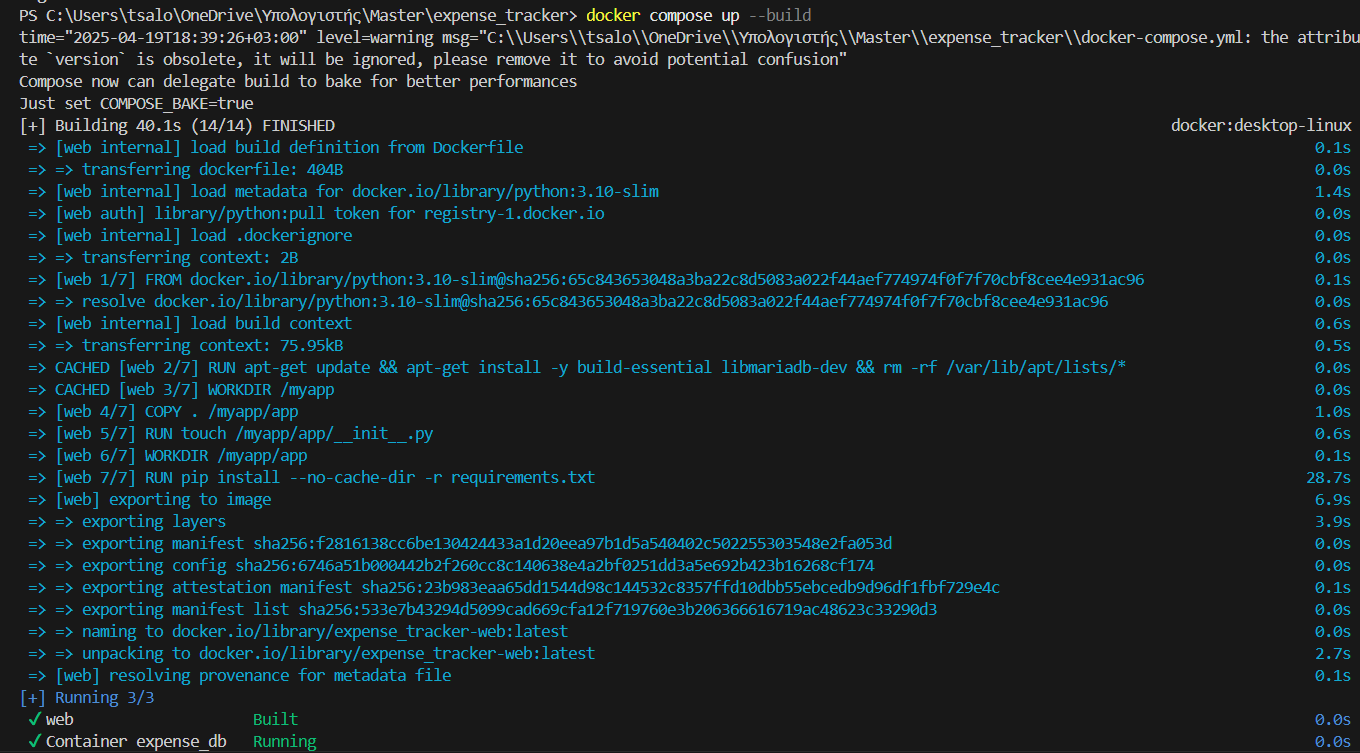
* **docker-compose.yml**

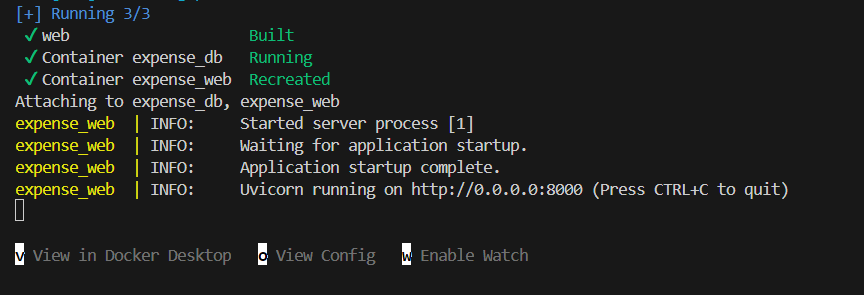
The docker-compose.yml defines two separate microservices, each with its own Dockerfile and build context. With the first service, we automatically create the database schema (based on the environment variables that configure the user, password, and database name), while the second builds the image from a local Dockerfile, defines the database connection via the DATABASE\_URL variable, and exposes port 8000 so that we can call the API endpoints from outside. Finally, the depends\_on setting ensures that the web service will first wait for MariaDB to start before attempting to connect.



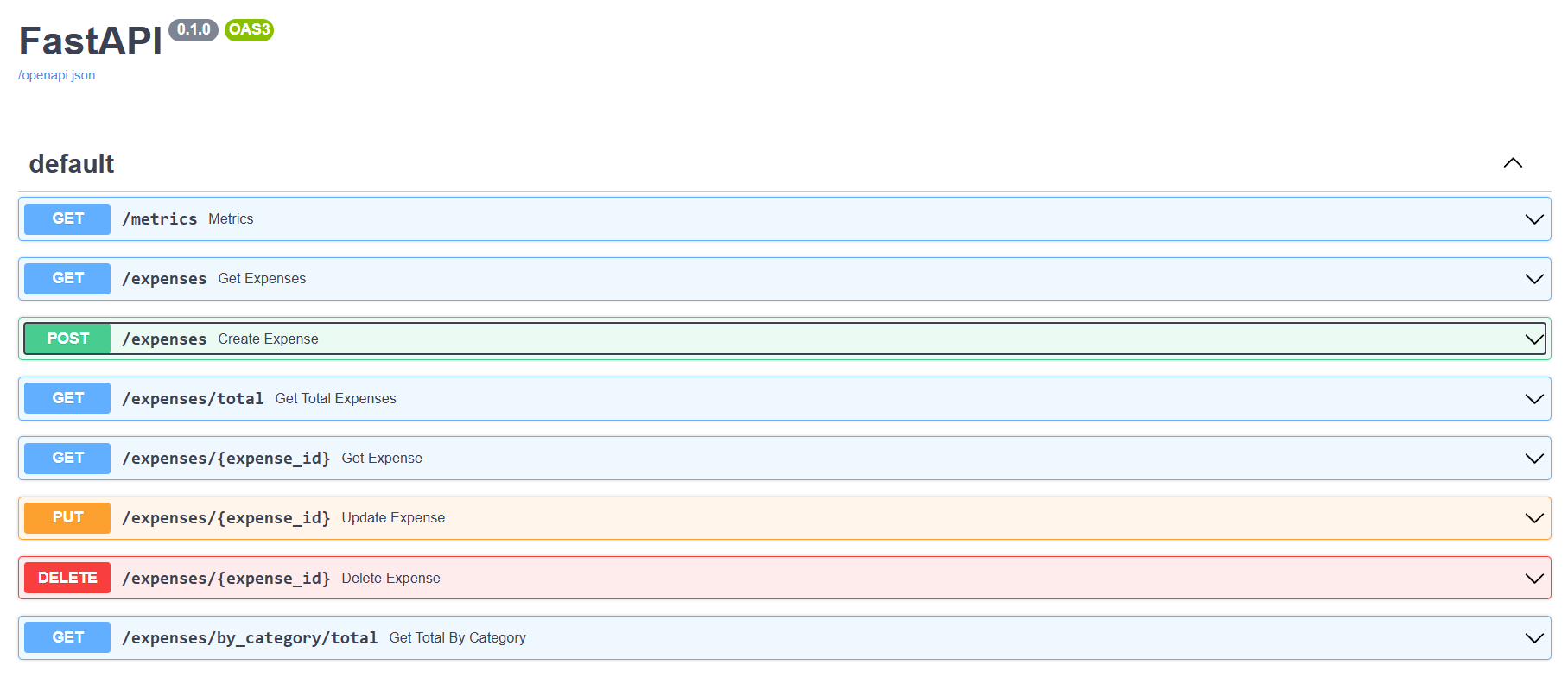
At this point, let’s do a small demo by spinning up a container locally on our machine.  
We run the command: docker **compose up –build**

With this command, Docker Desktop starts all the necessary processes to build the image, as we see below:

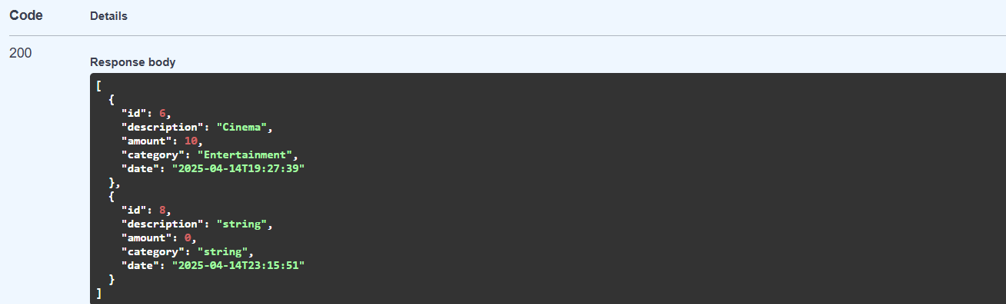
****

****

As we can see, the server has started and is awaiting requests at <http://0.0.0.0:8000>.

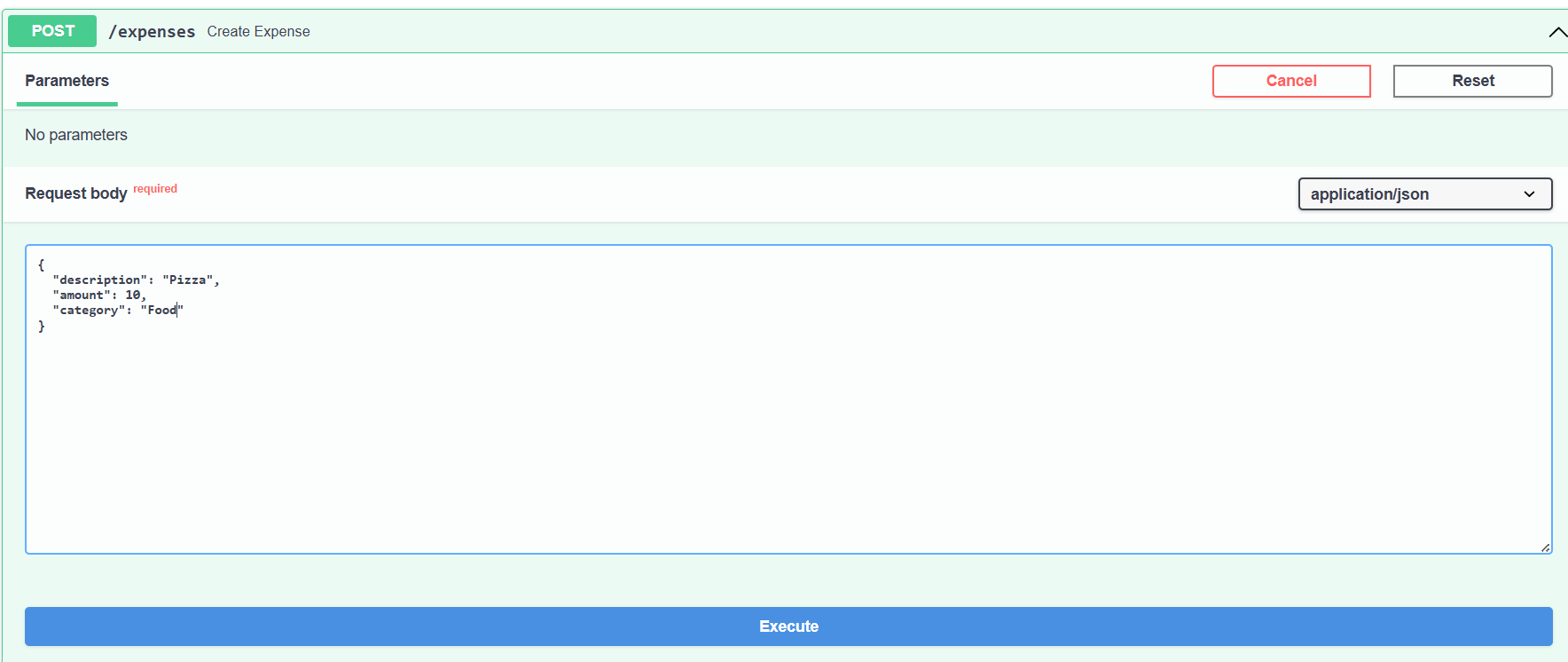
****

First, let’s execute a GET request. We open that specific request and then click **Try it out** and **Execute**. The result is what we see on our screen:

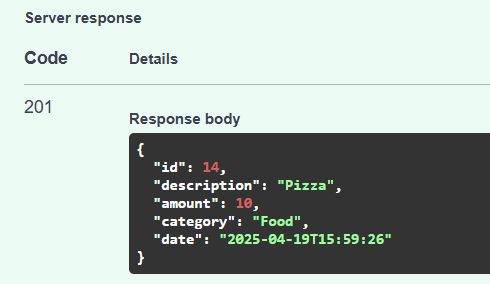
****

As we can see, our server returns **201**, which is the status code we set in the code for post requests, and in the response body we receive the two objects we created during development.

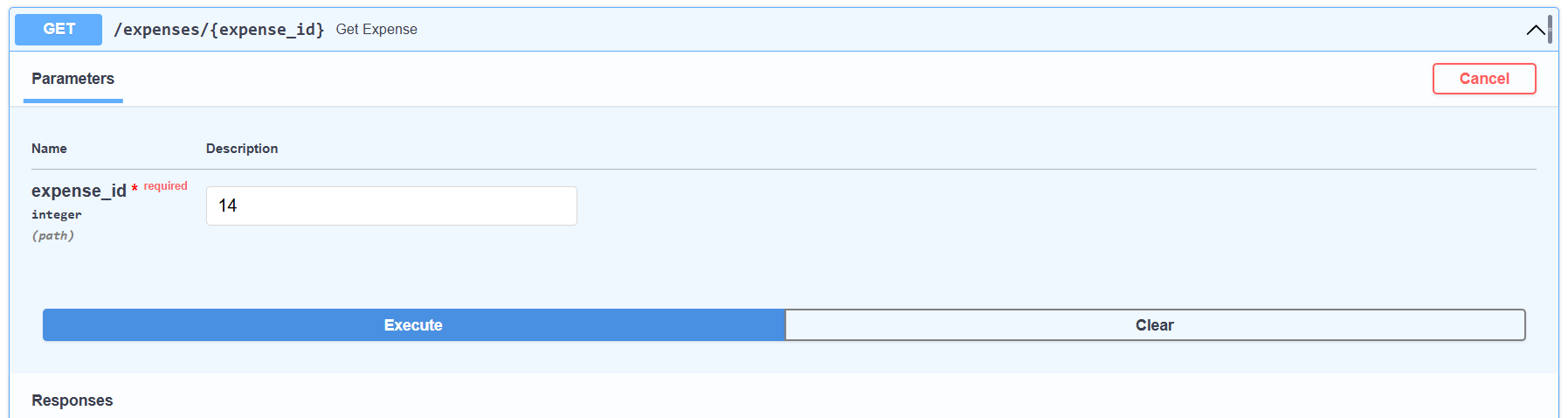
Now let’s create a new object: after selecting the corresponding **POST** request, we fill in the fields in the request body accordingly.

****

Then we click Execute. Below we see the server’s response. The id has been set as an auto‑increment, and the date field shows the current timestamp in UTC.



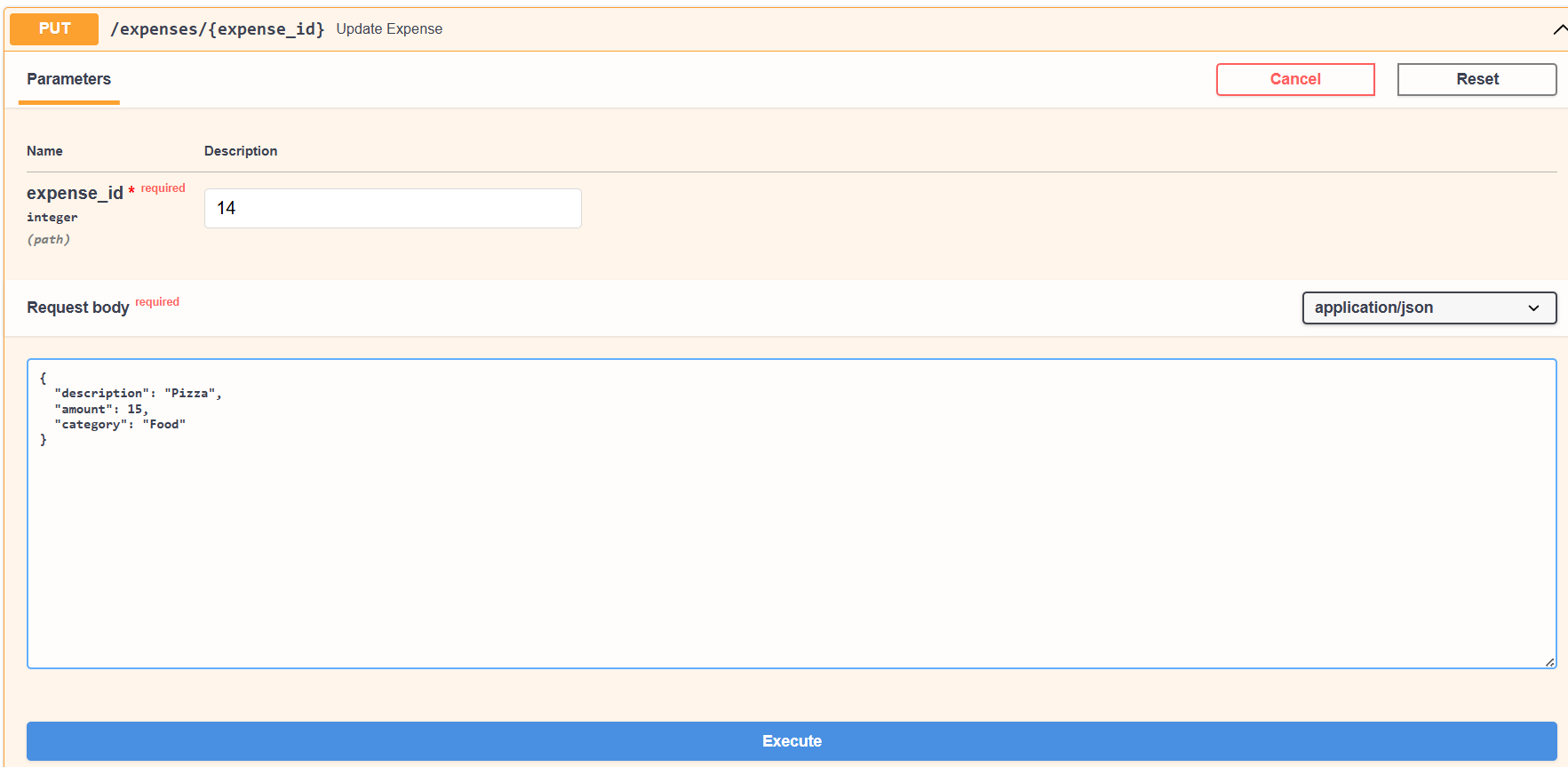
Similarly, to retrieve that expense we can use any of the GET requests. Let’s say by providing the id 14:



We click on **GET /expenses/{expense\_id}**, then click **Try it out** at the top right, enter **14** in the expense\_id field, and press **Execute**.



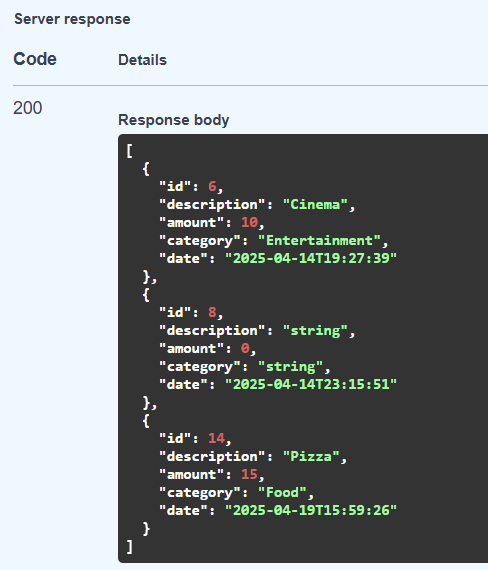
Let’s say we now want to change the pizza’s amount to 15 dollars.  
We go to the corresponding **PUT** request, click **Try it out** as before, and then fill in the fields:



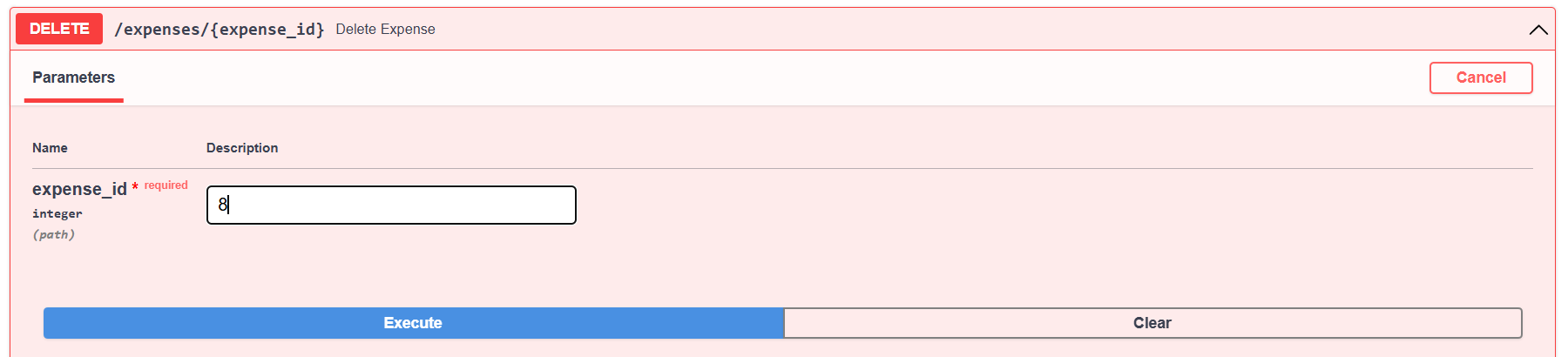
After we’ve changed the amount to 15, we click **Execute**.



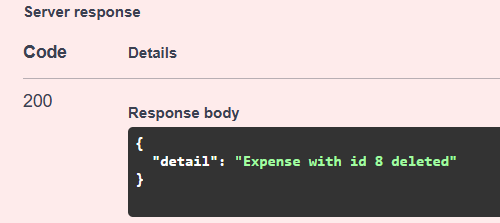
After changing the amount to 15, we click **Execute**. As we can see, the value has now been updated to 15. We can confirm this by making a GET request as well.



Finally, let’s say we want to delete the expense with id 8, which has the description “string.” We select the corresponding DELETE endpoint and enter 8.



We click **Execute**.



As we can see, we receive a message indicating that everything went well and the object was deleted.

If we then perform a GET to retrieve all expenses, we observe that there is no expense with ID 8.

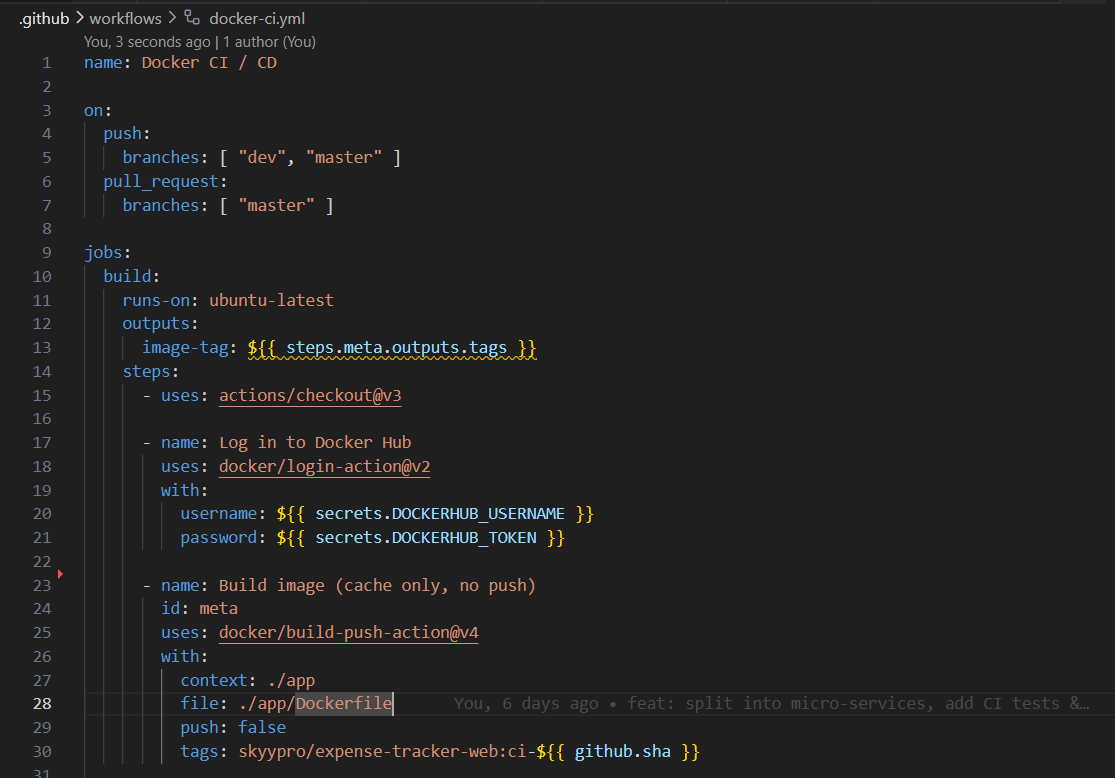


Regarding the CI-CD part of the coursework in order to securely authenticate with Docker Hub, we create a Docker access token with read and write access in our Docker Hub account settings. After that we add this token as a secret named DOCKERHUB\_ACCESS\_TOKEN in our GitHub repository.

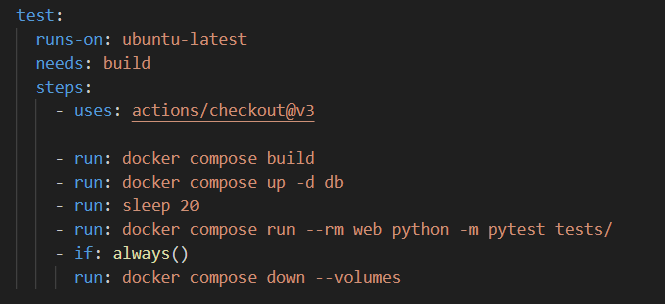
With our GitHub Actions workflow and Docker Hub access token set up, every time we push changes to the main branch, GitHub Actions will automatically build our FastAPI project into a Docker image and push it to Docker Hub.

* **.github\workflows\docker-ci.yml**

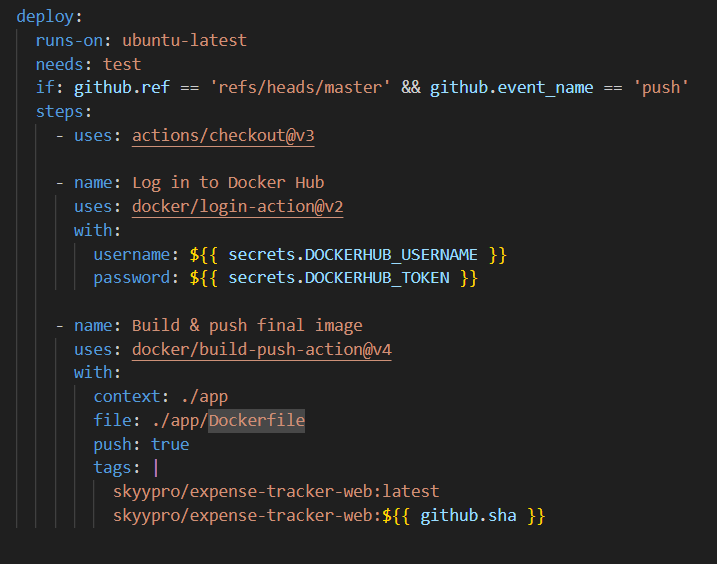
We create a .github/workflows folder in our root directory and place a ci-cd.yml file inside it. In that YAML file, we first define the events that will trigger the workflow—in our case, when a pull request is opened or updated against master. Next we write the first job, called build, which runs on an Ubuntu runner, checks out the code, logs into Docker Hub using our secrets (DOCKERHUB\_USERNAME and DOCKERHUB\_TOKEN), and builds the image without pushing it—thereby preserving an initial build cache stage.



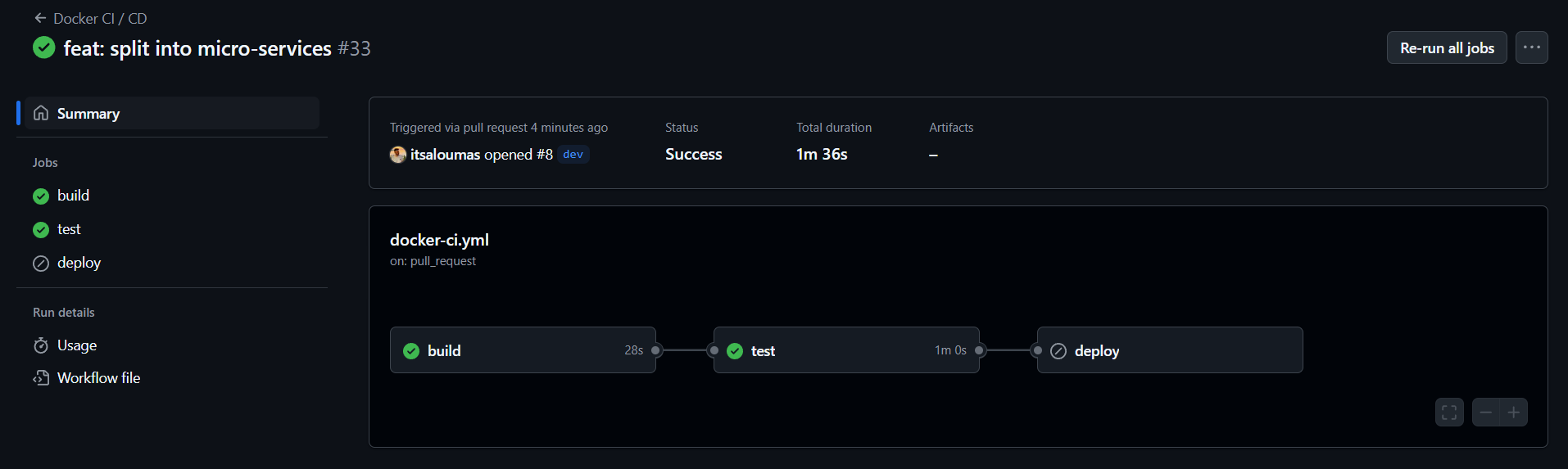
Then, in the test job (which depends on build), we again check out the code, rebuild the containers, start only the database in detached mode, wait a few seconds for it to become ready, and run our pytest tests inside the web container.



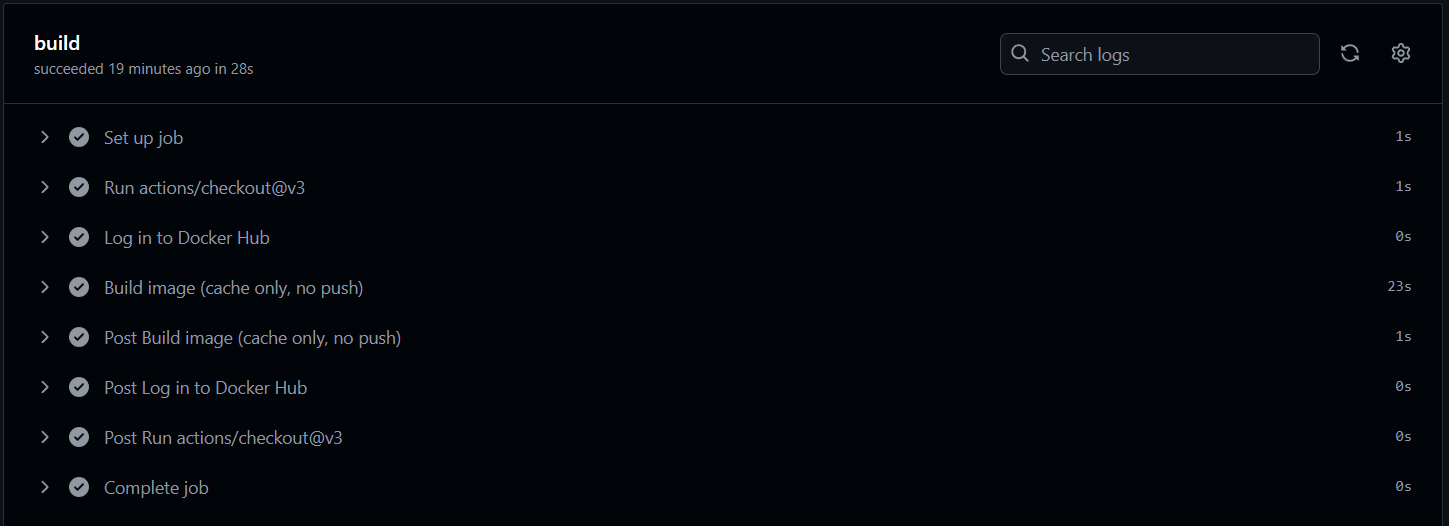
Finally, once all tests pass—and only if we’re on the master branch—the **deploy** job rebuilds the image, pushes it to Docker Hub with the latest tag and the commit SHA, ensuring that every merge into our production branch automatically produces a tested, ready‑to‑use Docker image. This gives us a complete CI/CD pipeline that automates building, testing, and delivering our application.



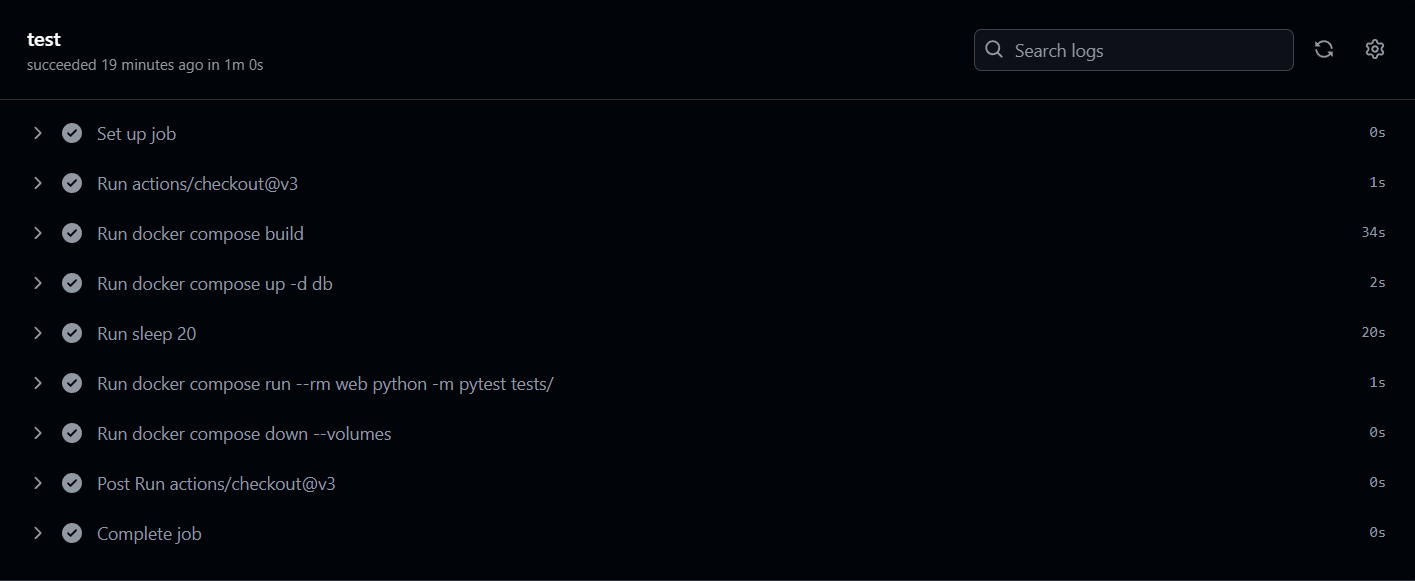
Once we finish the development of our application, we first push to the /dev branch. As soon as we push to /dev, the build and test stages of our application begin to run, as we can see below.

****

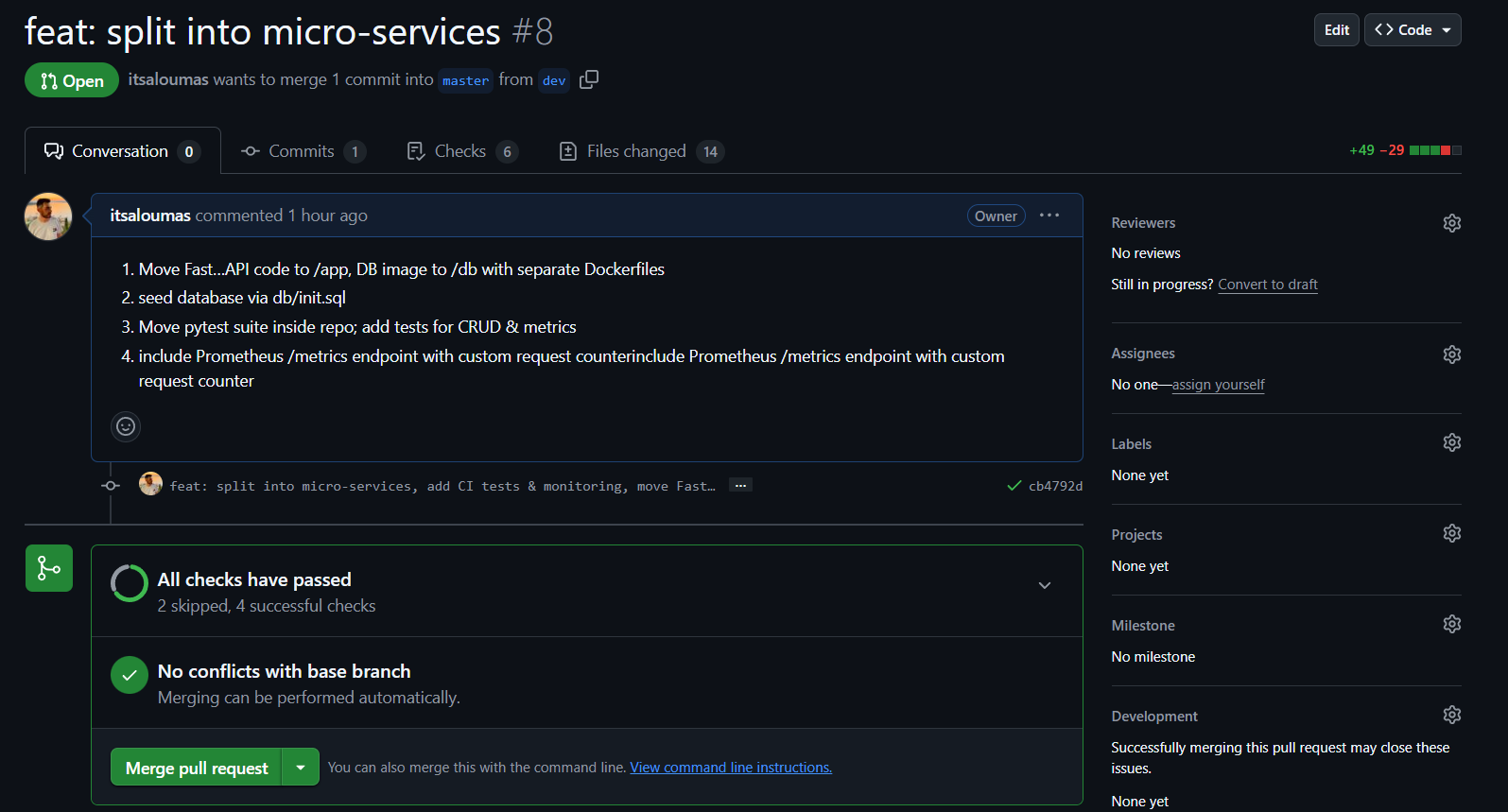
As we can see, the build and test actions were completed successfully. The build stage creates the image of the web microservice using the docker/build-push-action. It does not perform a push; it simply ensures that the Dockerfile is syntactically correct and that all dependencies are resolved.

****

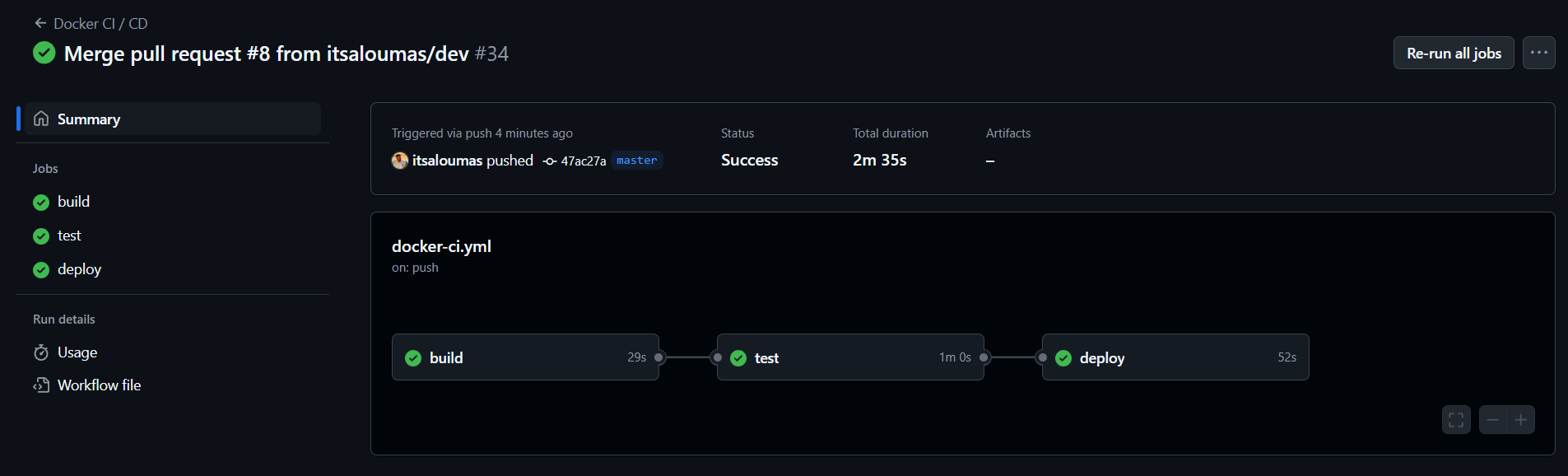
The test stage builds the entire docker-compose.yml setup (web + db). It starts the database (docker compose up -d db), then runs the unit tests using pytest inside the web image, and finally shuts everything down, cleaning up the volumes.

****

The deploy stage is executed only after a merge into the master branch. It rebuilds the image (using cache) and then performs a Docker push to Docker Hub (skyypro/expense-tracker-web:latest along with a hash tag), making it available for production. Therefore, in order to trigger the deploy process, we need to merge the pull request on GitHub by clicking the corresponding button, as shown below:

****

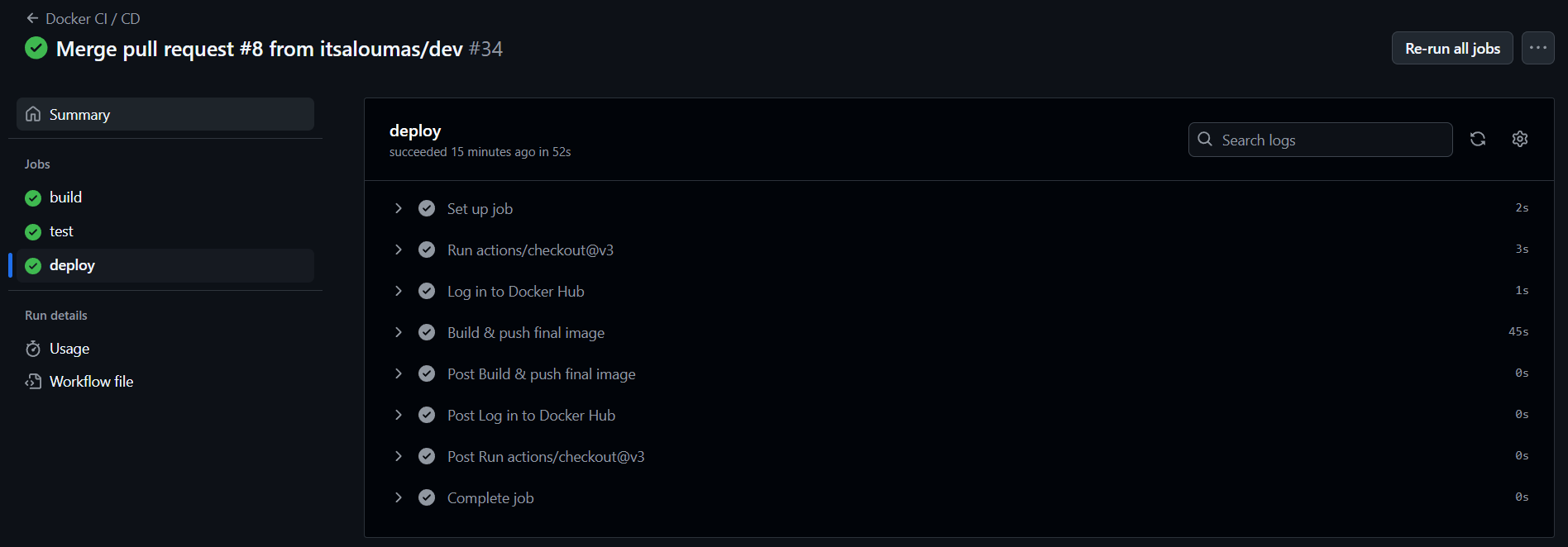
The logic of “running *only* on master” is configured inside the workflow file, where we’ve defined if: github.ref == 'refs/heads/master' && github.event\_name == 'push'. This way, after development, we first push to the dev branch, ensure that the build and test stages pass successfully, then create a Pull Request ⇾ merge into master, and only then is the deploy stage triggered.

****

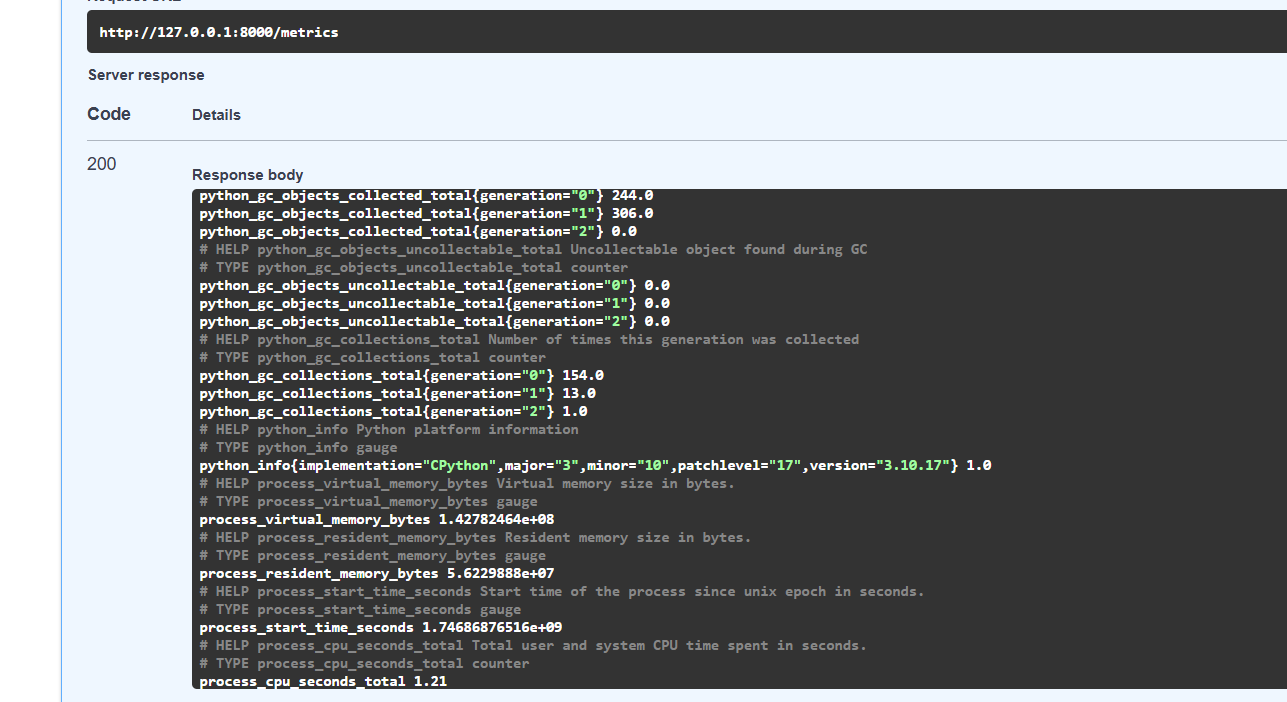
In the deploy stage, we use the secrets DOCKERHUB\_USERNAME and DOCKERHUB\_TOKEN to authenticate and perform the Docker push to Docker Hub with two tags:

•skyypro/expense-tracker-web:latest

• skyypro/expense-tracker-web:${{ github.sha }}



Finally, for the monitoring part, a metrics endpoint has been implemented through which we can expose both default metrics (Python-/process-related) for our application. If we hit the metrics endpoint, we receive the following as the response body:



From the image above, we can see metrics related to the application, such as:

* **process\_resident\_memory\_bytes**: indicates how much RAM is being used by the Uvicorn/FastAPI process.
* **process\_cpu\_seconds\_total**: shows the total CPU time consumed by the process.
* **expense\_api\_requests\_total**: this counter increases with every HTTP request that passes through our middleware. In our case, it is already at 3.0.