Worklab 2: Setup of the backend project, connect it to the frontend and setup the repository

Setup of the REST API Helloworld

We will use FastAPI to build our REST API backend. FastAPI is a modern, high-performance web framework for building APIs with Python 3.7+.

Its key features are:

- **Speed**:
  - **Leverages ASGI** (Asynchronous Server Gateway Interface) for high concurrency and performance.
  - Significantly faster than many other Python frameworks.
- **Developer Experience**:
  - **Type hints**: Utilizes Python's type hints for automatic data validation and clear code documentation.
  - **Asynchronous support**: Enables efficient handling of multiple requests concurrently, improving overall performance and responsiveness.
- Ease of Use:
  - **Concise and intuitive syntax**: Easy to learn and use, even for beginners.
  - Dependency injection: Simplifies code organization and testing.

- **Production Ready**:
  - **Robust and battle-tested**: Designed for production environments with features like dependency injection, data validation, and security considerations.

In essence, FastAPI aims to:

- **Increase development speed**: By reducing the time spent on writing boilerplate code and debugging.
- **Improve code quality**: Through type hints and automatic data validation.
- **Enhance developer experience**: With excellent documentation and a user-friendly interface.

Key Use Cases:

- **Building RESTful APIs**: For various applications, including web services, microservices, and machine learning models.
- **Creating backend services**: For web applications, mobile apps, and other client-side applications.

All the REST APIs and backend we deploy in production are developped using FastAPI. It's an excellent choice, the documentation is quite good and community is large, so you won't have trouble to find answers to your questions.

Let's create a new project for our backend.

- `mkdir backend`
- `cd backend`
- `uv init --app --python 3.12`

We will need the 4 following libraries: fastapi, uvicorn, httpx, pydantic. So let's add them to the project.

- `uv add fastapi pydantic httpx uvicorn`.

You already know `pydantic` and `httpx` from the previous worklab, `uvicorn` is a web server implementation for Python. It's uvicorn that allows fastapi to run.

Again, we will also add `ruff` and `isort` as dev dependencies.

- `uv add isort ruff --dev`

Before jumping into the REST API that will communicate with Ollama. Let's build a simple helloworld REST API.

The Helloworld of REST API

In the backend directory, create a `main.py` file and write the following code in it.

```python
from fastapi import FastAPI, status


app = FastAPI(title="JuniaGPT", version="1.0.0")

@app.post(
    "/helloworld",
    tags=["helloworld"],
    status_code=status.HTTP_200_OK,
)
def post_hellowolrd() -> dict:
    """the easiest rest api."""
    return {"hello": "world"}
```

This is the simplest REST API you can write. It has a single endpoint `/helloworld` which allows you to a `POST` request, and it returns the following dict in json format `{"hello": "world"}`.

`app = FastAPI(title="JuniaGPT", version="1.0.0")` is the main part of the code, this is this part which defines the FastAPI REST API, you can change the title to your liking, the version should follows the semantic versioning conventions.

The fact that you have defined an endpoint with to a `POST` request is defined by the following decorator

```
@app.post(
    "/helloworld",
    tags=["helloworld"],
    status_code=status.HTTP_200_OK,
)
```

`@app.post` will define a `POST` request, `@app.get` will define a `GET` request and so on. Inside you have to define the value of the endpoint, here `"/helloworld"`, some tags (optional) which helps grouping requests, and a `status_code` which defines the standard status code the endpoint should return if there are no errors.

You can see the full list of parameters of the `FastAPI` class here.

To see you app running, you can use one of the two following commands.

- `uv run uvicorn main:app --host 0.0.0.0 --port 8000 --reload` or
- `uvicorn main:app --host 0.0.0.0 --port 8000 --reload` if you have activated your virtual environment (remember why you need to activate it ?)

The following `curl` command will show that your api is up and running (you need to type it in another terminal than the one you used to run your api).

- `curl -X 'POST' 'http://0.0.0.0:8000/helloworld'`

Let's pimp a little bit our endpoint. We now want it to return `{"hello": "world"}` but with an extra parameter, like a name, so we want it to return for example `{"hello": "world by Mathieu"}`. There are two ways to do that: as part of the URL path, or as a query parameter.

As part of the URL path

To do that, update your code as the following one.

```python
from fastapi import FastAPI, status


app = FastAPI(title="JuniaGPT", version="1.0.0")

@app.post(
    "/helloworld/names/{name}",
    tags=["helloworld"],
    status_code=status.HTTP_200_OK,
)
def post_helloworld(name:str) -> dict:
    """the easiest rest api."""
    return {"hello": f"world by {name}"}
```

As a query parameter

To do that, update your code as the following one.

```python
from fastapi import FastAPI, status


app = FastAPI(title="JuniaGPT", version="1.0.0")

@app.post(
    "/helloworld/",
    tags=["helloworld"],
    status_code=status.HTTP_200_OK,
)
def post_hellowolrd(name:str) -> dict:
    """the easiest rest api."""
    return {"hello": f"world by {name}"}
```

The difference between the two lies in the URL, for the first one you have `"/helloworld/names/{name}"` so the name is part of the URL and to call this endpoint you have to a request like the following one.

- `curl -X 'POST' 'http://0.0.0.0:8000/helloworld/names/Mathieu'`

While in the latter case, this defines a query parameter, so the request will look like the following one.

- `curl -X 'POST' 'http://0.0.0.0:8000/helloworld/?name=Mathieu'`

Query parameters are always identified by the pattern `?key=value` in urls.

Which one is the best ? The best practice for RESTful API design is that **path parameters** are used to **identify a specific resource or resources**, while **query parameters** are used to **sort/filter those resources**, source. Since here we clearly are not sorting or filtering anything, we can stick with the URL path base approach.

Writing the documentation of the API with OpenAPI/Swagger

As any piece of software, a REST API needs to be documented. The documentation of a REST API is very specific and follows some conventions. In particular, the OpenAPI/Swagger convention. You can see the full OpenAPI Specification here, and the website of the OpenAPI Initiative here.

OpenAPI conventions:

- **Standardized Format**: OpenAPI uses a standardized format (typically JSON or YAML) to describe RESTful APIs. This ensures consistency and machine-readability, making it easier for both humans and machines to understand.
- Comprehensive Coverage: It covers all aspects of an API, including:
  - **Endpoints**: URLs, HTTP methods (GET, POST, PUT, DELETE, etc.).
  - **Parameters**: Path, query, header, and body parameters with their data types and descriptions.
  - **Request/Response Bodies**: Structures of request and response payloads, including data models and examples.
  - **Error Handling**: Definitions of possible error responses and their meanings.
  - **Authentication**: Mechanisms for securing API access (e.g., API keys, OAuth).
- **Design-First Approach**: OpenAPI encourages a design-first approach where the API is defined in the specification before implementation. This promotes better planning and reduces the risk of inconsistencies between the API design and its actual implementation.
- **Automated Tooling**: OpenAPI enables the use of automated tools for:
  - **Code Generation**: Generating server-side code (e.g., in Java, Python, Node.js) and client SDKs in various languages.
  - **Interactive Documentation**: Generating interactive API documentation with features like code samples, try-it-out consoles, and visualization tools.
  - **Testing**: Automating API tests based on the OpenAPI specification.
- **Community and Ecosystem**: OpenAPI has a large and active community, with extensive documentation, tutorials, and a wide range of supporting tools and libraries available. This makes it easier to learn, use, and integrate OpenAPI into your development workflow.

Right now I'm sure you do want to write some OpenAPI documentation related to your helloworld api.

```
from fastapi import FastAPI, status


app = FastAPI(title="JuniaGPT", version="1.0.0")

@app.post(
    "/helloworld/names/{name}",
    tags=["helloworld"],
    status_code=status.HTTP_200_OK,
)
def post_helloworld(name:str) -> dict:
    """the easiest rest api."""
    return {"hello": f"world by {name}"}
```

Then, let's go, here is the following OpenAPI documentation for this simple REST API.

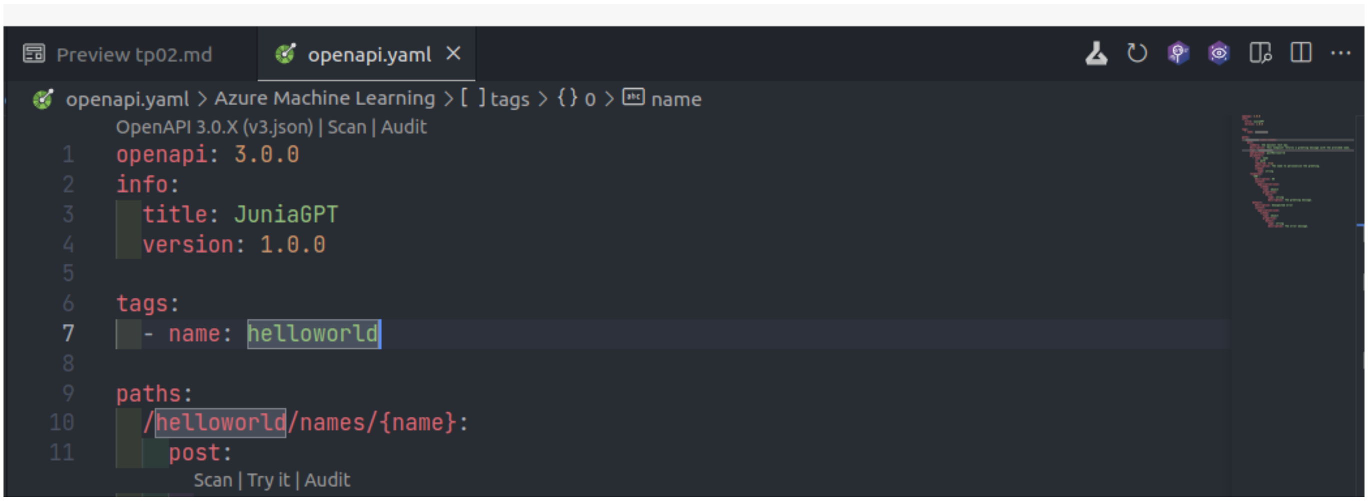
```
openapi: 3.0.0
info:
  title: JuniaGPT
  version: 1.0.0

tags:
  - name: helloworld

paths:
  /helloworld/names/{name}:
    post:
      summary: the easiest rest api.
      description: This endpoint returns a greeting message with the
provided name.
      tags: [helloworld]
      operationId: postHelloworld
      parameters:
        - name: name
          in: path
          required: true
          description: The name to personalize the greeting.
          schema:
            type: string
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  hello:
                    type: string
                    description: The greeting message.
        default:
          description: Unexpected error
          content:
            application/json:
              schema:
                type: object
                properties:
                  detail:
                    type: string
                    description: The error message.
```

You can see in vscode what this documentation looks like by installing the **OpenAPI (Swagger) Editor** extension, the extension ID is the following one: 42Crunch.vscode-openapi.

Once installed, rename your yaml file as `openapi.yaml` and then on the top right corner of the screen, click on the second button starting from the right.

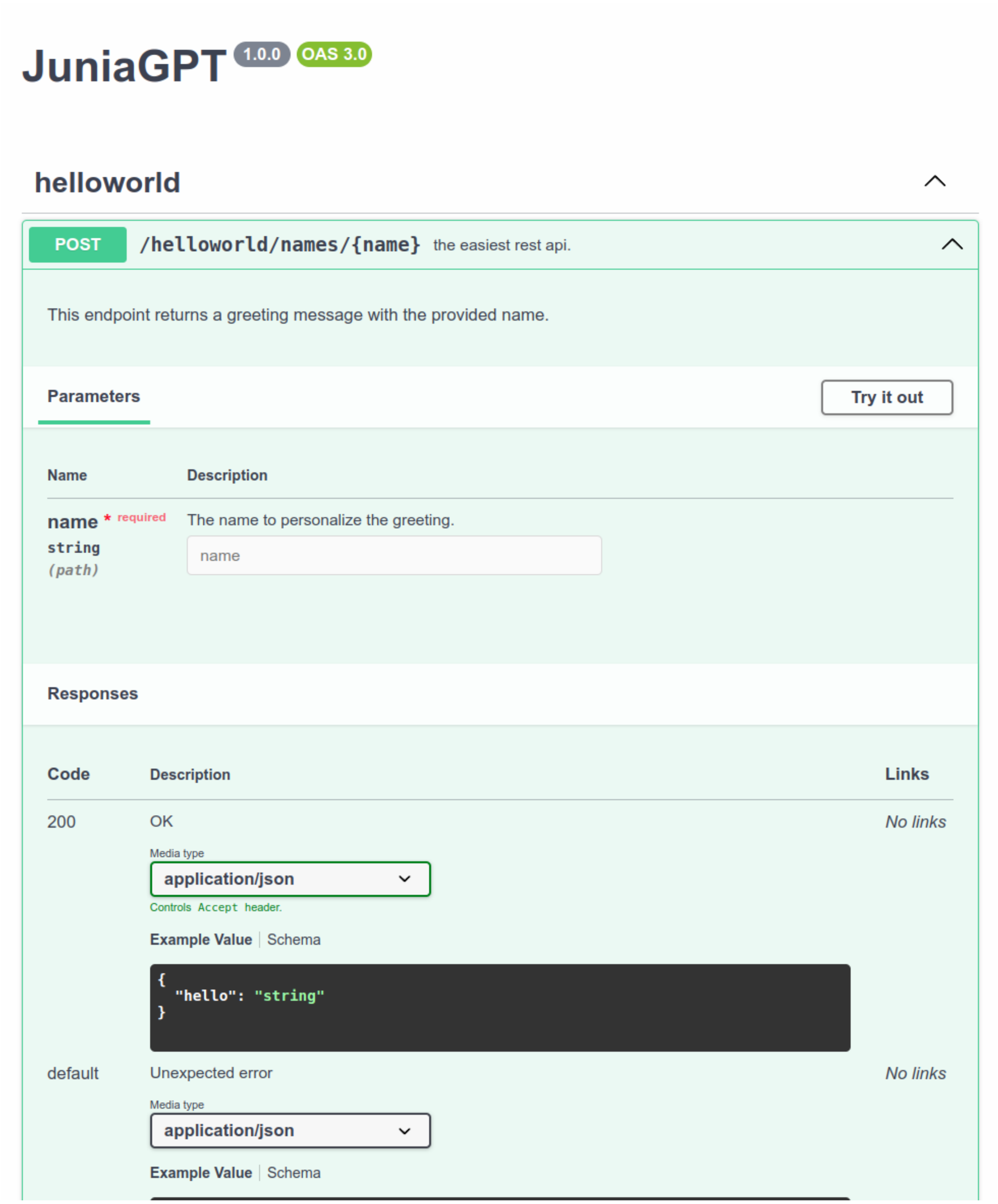

Preview tp02.md
openapi.yaml
openapi.yaml > Azure Machine Learning > [ ] tags > { } 0 > name
OpenAPI 3.0.X (v3.json) | Scan | Audit
Scan | Try it | Audit

```yaml
openapi: 3.0.0
info:
  title: JuniaGPT
  version: 1.0.0

tags:
  - name: helloworld

paths:
  /helloworld/names/{name}:
    post:
```

and you will see the OpenAPI documentation of your API. This should look like this.



/helloworld/names/{name} the easiest rest api.

This endpoint returns a greeting message with the provided name.

Name	Description
name * required string (path)	The name to personalize the greeting. name

Code	Description	Links
200	OK Media type application/json Controls `Accept` header.	No links
default	Unexpected error Media type application/json	No links

```
{
  "hello": "string"
}
```

Pretty cool no ? Well, the problem is that you will have to update this documentation each time you will update your API, moreover the "Try it out" button does not work, because to make it work you would need to be linked to a server hosting your API.

But is it always the case ? Thanks to FastAPI it isn't. But before jumping ti that we need to talk about Pydantic ans its integration with FastAPI.

FastAPI and Pydantic

When you need to send data from a client (let's say, a browser) to your API, you send it as a request body.

A request body is data sent by the client to your API. A response body is the data your API sends to the client.

Your API almost always has to send a response body. But clients don't necessarily need to send request bodies all the time, sometimes they only request a path, maybe with some query parameters, but don't send a body.

To declare a **request body**, you use Pydantic models with all their power and benefits. source

Lets try to use it for our helloworld api. The beautiful thing about Pydantic and FastAPI is that you can use Pydantic for both request body **and** response body, so that you can also do data validation on the response of yout API.

Let's try it, add the following things to your script: a new endpoint to your api and also some classes.

```python
class HelloIn(BaseModel):
    name: str


class HelloOut(BaseModel):
    hello: str
    name: str


@app.post(
    "/helloworld",
    tags=["helloworld"],
    response_model=HelloOut,
    status_code=status.HTTP_200_OK,
)
def post_helloworld_with_pydantic(name: HelloIn):
    """the easiest rest api."""
    world = f"world by {name.name}"

    return HelloOut(hello=world, name=name.name)
```

The code above demonstrates how to leverage Pydantic models for data validation and type checking in a FastAPI application. It also highlights clear documentation and proper response model definition.

- **Data Models**: defines two Pydantic models:
    - **HelloIn**: Represents the input data with a required string field named name.
    - **HelloOut**: Represents the output data with string fields hello and name.
- **Endpoint Definition**:
    - Creates a POST endpoint at the path /helloworld.
    - Associates the endpoint with the "helloworld" tag for better organization.

- **Response Model**:
  - Specifies the expected response model using HelloOut. This ensures the response structure matches the defined schema.
- **Status Code**:
  - Sets the expected HTTP status code to 200 OK for successful requests.
- **Function Implementation**:
  - Defines a function `post_helloworld_with_pydantic` that handles the POST request.
  - Takes an argument of type HelloIn, ensuring type safety and data validation.
  - Constructs the greeting message and assigns it to the hello field.
  - Returns an instance of HelloOut with the constructed message and received name.

if you run `uv run uvicorn main:app --host 0.0.0.0 --port 8000 --reload` to run your api, and then in another terminal you type:

```
curl -X 'POST' \
  'http://0.0.0.0:8000/helloworld' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "name": "Mathieu"
}'
```

you should see a response like `{"hello":"world by Mathieu","name":"Mathieu"}%`.

Request Bodies are mainly used in the case of `POST` requests, as we will see in the section where we connect the api to Ollama.

But wait, we have modified the api right ? So now we have to update the OpenAPI documentation ! Well, this is not the case, because FastAPI in fact generates automatically the documentation in the OpenAPI format.

Redirection to the documentation

When your api is up and running, you can access its documentation in the OpenAPI/Swagger format at the address `0.0.0.0:8000/docs`, or `localhost:8000/docs` depending on your browser. Since we are lazy and we do not want to type the `/docs` part, we add a direction redirection to the doc as soon as we type `0.0.0.0:8000`. You can do this by adding the following piece of code to your `main.py` file.

```
from starlette.responses import RedirectResponse

@app.get(
    "/",
    tags=["startup"],
    description="API startup on documentation page.",
)
def main():
    """Redirect to the api documentaiton at launch."""
    return RedirectResponse(url="/docs")
```

Why should we use a REST layer ?

Using a REST layer as a backend service between your frontend and an endpoint offers several key advantages:

Improved Security

- **Data Protection**: By centralizing data handling and logic on the backend, you can implement robust security measures like authentication, authorization, and data encryption. This protects sensitive user data from potential vulnerabilities in the frontend.
- **Reduced Attack Surface**: Isolating critical business logic on the backend minimizes the attack surface for hackers targeting your application.

Enhanced Scalability

- **Independent Scaling**: The frontend and backend can be scaled independently based on their respective demands. For example, you can scale the backend to handle increased data processing or traffic while keeping the frontend relatively unchanged.
- **Load Balancing**: The backend can be easily load balanced across multiple servers to handle high traffic volumes efficiently.

Improved Maintainability

- **Separation of Concerns**: Clear separation of frontend and backend responsibilities leads to better code organization, making the application easier to maintain and update.
- **Independent Development**: Frontend and backend teams can work independently, allowing for faster development cycles and easier collaboration.

Reusability

API-First Approach: The REST API can be reused by other applications, such as mobile apps or other web services, increasing the value and versatility of your backend logic.

Better Performance

- **Caching**: The backend can implement caching mechanisms to improve data retrieval performance for frequently accessed data.
- **Optimized Data Transfer**: The backend can optimize data transfer by sending only the necessary data to the frontend, reducing network traffic and improving application responsiveness.

Rate limiter

A **rate limiter** is a mechanism that controls the rate at which requests are made to a service. It essentially sets limits on how often certain actions can occur within a specified timeframe.

Rate limiters are a crucial component of many modern applications and services, playing a vital role in ensuring security, stability, and a positive user experience.

Purpose:

- Prevent abuse:
  - DDoS attacks: Protect against Distributed Denial of Service attacks, where a large number of requests overwhelm a system.
  - Brute-force attacks: Limit the number of login attempts to prevent unauthorized access.
  - API abuse: Prevent malicious or unintentional overuse of APIs by limiting the number of requests from a single source.
- Resource management:
  - Ensure fair resource allocation among users.
  - Prevent overloading of servers and maintain system stability.
- Throttling: Control the flow of traffic to prevent sudden spikes in demand from overwhelming the system.

How it works:

- Tracking: Monitors the number of requests within a specific time window (e.g., per second, minute, hour).
- Enforcement, if the number of requests exceeds the defined limit:
  - **Throttling**: Slow down the request rate by delaying responses.
  - **Blocking**: Temporarily or permanently block requests from the source.
  - **Returning error codes**: Send HTTP error codes (e.g., 429 Too Many Requests) to indicate that the rate limit has been exceeded.

Examples:

- API rate limiting: Limiting the number of API calls per second or minute from a single IP address or API key.
- Login attempts: Limiting the number of failed login attempts to prevent brute-force attacks.
- Email sending: Limiting the number of emails that can be sent within a certain time period to prevent spam.

Key benefits of using rate limiters:

** **Improved security**: Protects against various attacks and misuse. ** **Enhanced performance**: Prevents server overload and ensures consistent service availability. ** **Fair resource allocation**: Ensures fair access to resources for all users. ** **Better user experience**: Prevents excessive delays and improves overall application responsiveness.

Connect FastAPI and Ollama

To connect FastAPI to Ollama, we will use the same method we use in for the frontend by defining an HTTP client, we will also use pydantic to define the input and the output of our FastAPI route.

At the root of the backend, let's create a `config` directory and let's add some files in it, so that your directory has the following architecture.

```
.
├── config
│   ├── __init__.py
│   └── schemas.py
├── __init__.py
├── main.py
├── pyproject.toml
├── README.md
└── uv.lock
```

We will have the following pydantic classes to the `config/schemas.py` file.

```python
from pydantic import BaseModel

class InferenceBase(BaseModel):
    """The base class of our pydantic models."""

    pass


class PromptIn(InferenceBase):
    """The class we use to format the inputs of our api."""

    role: str
    content: str

    model_config = {
        "json_schema_extra": {
            "examples": [{"role": "user", "content": "this is a test"}],
        },
    }


class PromptOut(InferenceBase):
    """The output of the REST API."""

    answer: str

class Chat(BaseModel):
    """Base class for what a generic POST request to an LLM should contain.

    * The model you want to use.
    * The temperature.
    * The messages you send.
    """

    model: str
    temperature: float | None = Field(ge=0.0, le=1.0, default=0.7)
    messages: list[dict[str, str]]
```

Aside from the `Chat` class we've already defined in the last wroklab, let's break down the Python code:

1. **Importing Pydantic:**

`from pydantic import BaseModel`: This line imports the BaseModel class from the pydantic library. Pydantic is a powerful library for data parsing and validation. BaseModel serves as the foundation for creating data models that define the structure and constraints of your data.

2. **Defining the Base Class:**

`class InferenceBase(BaseModel)`: This defines a class named `InferenceBase` that inherits from BaseModel. This establishes a common base for all other models within your system, potentially sharing common attributes or methods.

3. **Defining the Input Model:**

- `class PromptIn(InferenceBase)`: This creates a class PromptIn that inherits from InferenceBase. This class represents the structure of the input data for your API.
- `role: str`: This defines a required field named role of type str. It represents the role of the user or system sending the prompt (e.g., "user", "assistant").
- `content: str`: This defines another required field named content of type str. This holds the actual text of the prompt or message.
- `model_config:` This dictionary provides additional configuration for the model.
    - `json_schema_extra:` This key allows you to add extra information to the JSON Schema generated by Pydantic.
    - `examples:` This provides an example of how the PromptIn object should look in JSON format: `{"role": "user", "content": "this is a test"}`. This is helpful for documentation and understanding the expected input structure.

4. **Defining the Output Model:**

- `class PromptOut(InferenceBase):` This creates a class PromptOut that inherits from InferenceBase. This class represents the structure of the output data from your API.
- `answer: str:` This defines a required field named answer of type str. This field will hold the response or output generated by your model.

In Summary:

This code snippet defines three Pydantic models:

- `InferenceBase`: A base class that other models can inherit from.
- `PromptIn`: A model that defines the expected structure of the input data for your API, including the role and content of the prompt.
- `PromptOut`: A model that defines the expected structure of the output data from your API, containing the answer generated by your model.

By using these models, you can:

- **Validate input data**: Ensure that the input received by your API conforms to the expected structure and data types.
- **Generate consistent output**: Ensure that the output from your API adheres to the defined structure.
- **Improve code readability and maintainability**: By defining clear data structures, you make your code more organized and easier to understand.
- **Generate documentation**: Pydantic can automatically generate JSON Schema from your models, which can be used to document your API.

Finally, let's add our route to Ollama. Creat a directory route like this.

```
.
├── config
│   ├── __init__.py
│   └── schemas.py
├── __init__.py
├── main.py
├── pyproject.toml
├── README.md
├── routes
│   ├── __init__.py
│   └── juniagpt.py
└── uv.lock
```

in `routes/juniagpt.py`, add the following code.

```python
from typing import Annotated

from fastapi import APIRouter, Body, status

from config.schemas import PromptIn, PromptOut, Chat
import httpx

router = APIRouter(prefix="/v1")

class LLMClient:
    """The client used to communicate with the backend LLM."""

    def __init__(
        self,
        root_url: str,
    ) -> None:
        self.client = httpx.Client(verify=True)
        self.root_url = root_url

    def _generate_request(self, chat: Chat) -> tuple[dict, dict, str]:  # 
type:ignore
        """Generates the 3 parts necessary for the request via the HTTPX 
library.
```

```
        This function generates the header, body, and url for a POST
request via HTTPX.

        Args:
            chat (Chat): A Chat class.

        Returns:
            tuple[dict, dict, str]: The header, body, and url.
        """
        headers = {
            "accept": "application/json",
            "Content-Type": "application/json",
        }

        body = {
            "model": chat.model,
            "messages": chat.messages,
            "stream": False,
            "options": {"temperature": chat.temperature},
        }

        route = f"http://{self.root_url}/api/chat"

        return headers, body, route  # type: ignore

    def post(
        self,
        chat: Chat,
    ):
        """POST request."""
        headers, body, route = self._generate_request(chat=chat)

        try:
            response = self.client.post(
                url=route,
                headers=headers,
                json=body,
                timeout=180.0,
            )
            response.raise_for_status()
        except httpx.RequestError as exc:
            print(f"An error occurred while requesting
{exc.request.url!r}.")
            raise
        except httpx.HTTPStatusError as exc:
            print(
                f"Error response {exc.response.status_code} while
requesting {exc.request.url!r}."
            )
            raise

        return response
```

```
client = LLMClient(root_url="localhost:11434")


@router.post(
    "/models/{model}/temperature/{temperature}/",
    tags=["chat"],
    response_model=PromptOut,
    status_code=status.HTTP_200_OK,
    summary="Converse with JuniaGPT.",
)
def chat(
    prompts: Annotated[
        list[PromptIn],
        Body(
            examples=[
                [
                    {"role": "system", "content": "You are a helpful 
assistant."},
                    {"role": "user", "content": "this is a test"},
                ],
            ],
        ),
    ],
    model: str,
    temperature: float,
):
    messages = [{"role": prompt.role, "content": prompt.content} for prompt 
in prompts]
    chat = Chat(model=model, temperature=temperature, messages=messages)

    response = client.post(chat=chat)

    message = response.json()["message"]["content"]

    return PromptOut(answer=message)
```

Let's breakdown the provided code:

1. **Imports**:

- `from typing import Annotated`: This imports the Annotated type hint from the typing library. This is used to provide additional metadata to type annotations.
- `from fastapi import APIRouter, Body, status`: This imports the APIRouter, Body, and status classes from the FastAPI library. These are used for building web APIs in Python.
- `from config.schemas import PromptIn, PromptOut, Chat`: This imports the PromptIn, PromptOut, and Chat classes from the config.schemas module. These classes likely the data structures used for the API requests and responses.
- `import httpx`: This imports the httpx library, which is a high-performance HTTP client for Python.

2. **API Router**:

`router = APIRouter(prefix="/v1")` creates an APIRouter object named router with a prefix of `/v1`. This router will handle all the API endpoints defined within it.

3. **LLMClient Class**:

You know it. This class defines a client to interact with a large language model (LLM) backend.

- `def __init__(self, root_url: str) -> None`: This is the constructor for the LLMClient class. It takes the root URL of the LLM backend as input and initializes an httpx.Client object for making HTTP requests.
- `def _generate_request(self, chat: Chat) -> tuple[dict, dict, str]`: This function generates the necessary components for an HTTP POST request to the LLM backend. It takes a Chat object as input and returns a tuple containing the headers, body, and URL for the request.
  - It creates headers specifying JSON acceptance and content type.
  - It constructs the request body based on the provided Chat object, including the model name, messages, and temperature setting.
  - It builds the complete URL for the /api/chat endpoint on the LLM backend server.
- `def post(self, chat: Chat) -> httpx.Response:` This function sends a POST request to the LLM backend using the headers, body, and URL generated by _generate_request. It handles potential errors during the request and returns the HTTP response object.

4. Creating an LLMClient Instance:

`client = LLMClient(root_url="localhost:11434")` creates an instance of the LLMClient class, specifying the root URL of the LLM backend as `localhost:11434`.

5. chat API Endpoint:

- `@router.post(...)`: This defines a POST API endpoint at `/v1/models/{model}/temperature/{temperature}/` using the router. It adds metadata like tags, response model, status code, and a summary for documentation purposes.
- `prompts: Annotated[list[PromptIn], Body(...)]`: This defines the request body parameter named prompts. It expects a list of `PromptIn` objects using the Body decorator. The decorator provides additional details like examples to illustrate the expected format of the request body.
- `model: str:` This defines another path parameter named model that captures the model name from the URL path.
- `temperature: float:` This defines another path parameter named temperature that captures the temperature value from the URL path.
- `messages = [...]`: This processes the list of prompts into a list of dictionaries containing role and content keys based on each `PromptIn` object.
- `chat = Chat(model=model, temperature=temperature, messages=messages)`: This creates a Chat object with the extracted model name, temperature, and processed messages.
- `response = client.post(chat=chat)`: This calls the post method of the client instance, sending the constructed Chat object as input.
- `message = response.json()["message"]["content"]`: This extracts the response message content from the JSON response received from the LLM backend.
- `return PromptOut(answer=message)`: This constructs a PromptOut object with the extracted message content as the answer and returns it as the API response.

All that's left to do is to clean a little bit the `main.py` script, add the router, and and healthcheck route.

```python
from fastapi import FastAPI, status
from starlette.responses import RedirectResponse

from routes import juniagpt

app = FastAPI(title="JuniaGPT", version="1.0.0")


app.include_router(juniagpt.router)


@app.get(
    "/",
    tags=["startup"],
    description="API startup on documentation page.",
)
def main():
    """Redirect to the api documentation at launch."""
    return RedirectResponse(url="/docs")


@app.get(
    "/healthcheck",
    tags=["healthcheck"],
    status_code=status.HTTP_200_OK,
    response_description="ok",
    summary="resume",
)
def get_api_status() -> str:
    """Return Ok if the api is up."""
    return "ok"
```

Your REST API is ready to run with the command `uv run uvicorn main:app --host 0.0.0.0 --port 8000`.

Connect FastAPI and the frontend

To connect our REST api and the frontend, we will have to modify the client we wrote in our frontend in the `rest/service.py`.

1. The `root_url` is now `http://0.0.0.0:8000` or `http://localhost:8000`, depending on your OS.
2. The route of our api is `http://0.0.0.0:8000/v1/models/{model}/temperature/{temperature}/`, so the route in our frontend should be now `f"http://{self.root_url}/v1/models/{model}/temperature/{temperature}/"`
3. The answer is wrapped in a `PromptOut` pydantic class, thus the `message` value in the `main.py` script of our frontend is not `message = response.json()["message"]["content"]` anymore but

`message = response.json()["answer"]`.

Once you've done these modifications, you can test your connections between the services.

1. Be sure that your Ollama server is running.
2. Run your REST api in one terminal with the command `uv run uvicorn main:app --host 0.0.0.0 --port 8000`
3. Run your frontend in another terminal.
4. Connec to your terminal and test it.

Setup the remote git repository

- Go to `https://github.com/` and create an account if do not have one.
- Once done, create a new private repository named `backend`.

Now before pushing to your remote repository, you need a way to authenticate to GitHub from the terminal. There are usually two possible ways:

- SSH keys
- Personal Access Token

SSH keys allow for a seamless way to push content as you do not have to remember and store the access token, but many companies use private virtual network and as such have shut down the ways to use SSH.

Let's create a Personal Access Token (PAT).

1. On the top right corner of GitHub, click on your icon and select `Settings`.
2. On the left menu, go all the way down to `Developer settings`.
3. Select `Personal access tokens` -> `Tokens (classic)` and click on `Generate new token` -> `Generate new token (classic)`.

Give a name to your PAT, Set the expiration date to 60 days (or No expiration if you want to keep it) and select the scopes, ie the permissions given to this PAT. We will the follow the Principle of Least Privilege and give him access to

- `repo`
- `workflow`
- `write:packages`

4. Click on `Generate token` at the bottom of the page. You will see the value of your PAT, copy-paste it and store it somewhere safe because you will be able to see it only once.

Now let's push our backend to the remote repository.

```
git add .
git commit -m "feat: first commit"
git remote add origin https://github.com/your_github_repo_address
git branch -M main
git push -u origin main
```

If everything goes well, you should see the content of the backend directory pushed to your GitHub repository.

Setup pre-commit GitHub Actions

You have to have done tasks 1, and 2 before jumping on this one.

What are GitHub Actions

GitHub Actions is a powerful tool for automating software development workflows right within your GitHub repositories.

GitHub Actions provides a flexible and integrated platform to streamline your development process, improve code quality, and accelerate your software delivery.

What it does:

- **Continuous Integration & Continuous Delivery (CI/CD)**:
    - **Automates builds**: Compiles your code, packages it (e.g., into containers or executables), and runs tests.
    - **Automates deployments**: Deploys your code to various environments like testing, staging, and production.
- **Beyond CI/CD**:
    - **Custom workflows**: Create workflows for any event in your repository, such as:
        - **Issue creation**: Automatically label issues based on keywords.
        - **Pull request events**: Trigger code reviews, run linters, or automatically merge approved PRs.
        - **Releases**: Create and publish releases, including generating changelogs.

Key Concepts:

- **Workflows**: Define a sequence of jobs that run in response to an event.
- **Jobs**: A set of steps that execute in parallel.
- **Steps**: Individual tasks within a job, often represented by scripts or actions.
- **Actions**: Reusable units of code that perform specific tasks. They can be community-built or created by you.
- **Runners**: Machines (virtual or physical) that execute your workflows.

Benefits:

- **Increased efficiency**: Automate repetitive tasks, saving time and reducing errors.
- **Improved quality**: Catch bugs early in the development cycle through automated testing.
- **Faster deployments**: Streamline the release process, getting your software to users faster.
- **Better collaboration**: Facilitate smoother teamwork by automating code reviews and other processes.
- **Flexibility**: Highly customizable to fit your specific project needs.

Setup pre-commit

Now, let's design our first pipeline, this will be a CI pipeline designed to ensure that the classical conventions of code style are enforced.

We will use:

- ruff
- isort
- pre-commit

So let's add them to our project with `uv`, since these libraries are purely used for development, we will install them as optional dependencies. There is already a shortcut for dependencies used only for dev with the argument `--dev`. Type the following commands.

- `uv add ruff --dev`
- `uv add isort --dev`
- `uv add pre-commit --dev`

You will see in the `pyproject.toml` file that a new section has been created:

```
[dependency-groups]
dev = [
    "isort>=5.13.2",
    "pre-commit>=4.0.1",
    "ruff>=0.9.2",
]
```

We are going to use `pre-commit`. Pre-commit is a powerful framework for managing and maintaining multi-language pre-commit hooks. What are pre-commit hooks? They are scripts or commands that execute automatically before you commit changes to your Git repository.

Their primary purpose is to:

- **Enforce code style and formatting**: Ensure consistent code across your project (e.g., using tools like Black, isort, Prettier).
- **Catch errors early**: Identify potential issues like syntax errors, type errors, and security vulnerabilities before they reach your main codebase.
- **Run tests**: Execute unit tests or other checks to ensure code quality and functionality.

How pre-commit helps:

- **Manages hook installation and execution**: You define a list of hooks in a configuration file (.pre-commit-config.yaml), and pre-commit handles installing and running them automatically.
- **Supports various languages**: Works with hooks written in any language, making it versatile for diverse projects.
- **Easy to configure**: The configuration file is simple and user-friendly, allowing you to easily customize hooks and their settings.
- **Improves code quality**: By catching issues early, pre-commit helps maintain a high level of code quality and reduces the risk of introducing bugs.

In essence, pre-commit automates the process of running checks on your code before committing, making it easier to maintain a clean and consistent codebase.

When you're working on a Git project, consider using pre-commit to improve your workflow and enhance the quality of your code.

Let's create the initial config for `pre-commit` with the following command.

- `uv run pre-commit sample-config > .pre-commit-config.yaml`

This commands create the following yaml file.

```yaml
# See https://pre-commit.com for more information
# See https://pre-commit.com/hooks.html for more hooks
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v3.2.0
    hooks:
    -   id: trailing-whitespace
    -   id: end-of-file-fixer
    -   id: check-yaml
    -   id: check-added-large-files
```

As you can see, there are already predefined standards hooks, you can see the full list of all standard hooks here.

`ruff` and `isort` also have pre-commit hooks, here and here (and if you wonder if your favorite python styling library as one, just google "your_lib precommit hook" to see.).

There's no rocket science here, just follow the instructions in the github repos to add them to the config. You'll get the following yaml file.

```yaml
# See https://pre-commit.com for more information
# See https://pre-commit.com/hooks.html for more hooks
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v5.0.0
  hooks:
    - id: trailing-whitespace
    - id: end-of-file-fixer
    - id: check-yaml
    - id: check-added-large-files

- repo: https://github.com/astral-sh/ruff-pre-commit
  rev: v0.9.2
  hooks:
    # Run the linter.
    - id: ruff
      args: [ --fix ]
    # Run the formatter.
```

```
      - id: ruff-format

  - repo: https://github.com/pycqa/isort
    rev: 5.13.2
    hooks:
      - id: isort
        name: isort (python)
```

Now, all that's left to do to use pre-commit is to install it and to update it. Run the following commands.

- `uv run pre-commit install`
- `uv run pre-commit autoupdate`

It's settled, everytime you'll commit a piece of code, all the pre-commit above will automatically run and check if your code is compliant with them. If not, they will automatically modify your code. You will just have to add the modifications to git and commit again to have them tracked.

Setup the CI linting pipeline

We want our pre-commits to be able to run locally, but also when we push code in our remote repository.

Why ? This is to ensure that your code will always be consistent, whereever you are or whoever you work with, your code will always have to pass the same gates and checks. That is an industry standard.

To do that, we have to define a "github action", more commonly called a CI pipeline. CI pipelines are written in yaml, and for github have to be stored in a directory `.github/workflows` located at the root of your project.

Let's write the following yaml file, which we call `linting.yaml`. And put it in the directory `.github/workflows` at the root of the `backend` project.

```
name: Lint on Push and Pull Request

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ] # Remplacer 'main' par votre branche principale

jobs:
  lint:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.12'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install pre-commit

      - name: Run pre-commit checks
        run: |
          pre-commit install
          pre-commit autoupdate
          pre-commit run --all
```

The overall purpose of this action is to automatically run code linting and formatting checks whenever code is pushed to the 'main' branch or a pull request is created/updated targeting the 'main' branch. This helps to maintain code quality and consistency within the repository.

Here's a breakdown of what this pipeline does.

1. **Name**: The action is named "Lint on Push and Pull Request".
2. **Triggers**: The action is triggered on two events:
    - **Push**: When code is pushed to the 'main' branch.
    - **Pull Request**: When a pull request is created or updated targeting the 'main' branch.
3. **Job**: The action defines a single job named 'lint'.
4. **Runner**: The job runs on the 'ubuntu-latest' runner.
5. **Checkout**: The first step checks out the code from the repository using the `actions/checkout@v3` action. This action checks out your repository's code into the workflow's (pipeline's) workspace. This allows your workflow to access and work with the code within your repository.
6. **Python Setup**: The second step sets up the Python environment using the `actions/setup-python@v4` action, specifying Python version '3.12'. This action sets up a specific Python version on the runner environment. This ensures that your workflow uses the desired Python version for your project's requirements.
7. **Install Dependencies**: The third step installs the required dependencies:
    - Upgrades pip to the latest version.
    - Installs the pre-commit library for code formatting and linting.
8. **Pre-commit Installation**: The fourth step installs the pre-commit hooks and updates them to the latest versions.
9. **Run Checks**: The final step executes all pre-commit checks defined in the .pre-commit-config.yaml file.

Now, everything that's left is to commit your change and poush it into your remote repository.

- `git add .`
- `git commit -m "feat: added pre-commit and ci"`
- `git push`

Once pushed, GitHub will automatically detect your pipeline and register it. You will be able to see its behavior ont the `Actions` section of your repository.