

Worklab 1: Setup of the frontend project and repository

- [Worklab 1: Setup of the frontend project and repository](#)
 - [Initial task: Create the frontend directory](#)
 - [Task 1: Setup the frontend skeleton](#)
 - [directory creation, project initialization](#)
 - [Dependencies installation](#)
 - [Frontend creation](#)
 - [First steps to a chatbot](#)
 - [Adding chat history](#)
 - [Ollama, or how I stopped worrying about a ChatGPT subscription and started using llms locally](#)
 - [Installation](#)
 - [Model Selection](#)
 - [Run a Model](#)
 - [Start Ollama as a server](#)
 - [Interact with the Model:](#)
 - [The frontend HTTP client, how to use HTTPX and pydantic](#)
 - [HTTPX](#)
 - [Pydantic](#)
 - [Setup of the REST service](#)
 - [Frontend update](#)
 - [Task 2: Setup the remote git repository](#)
 - [Task 3: Setup pre-commit GitHub Actions](#)
 - [What are GitHub Actions](#)
 - [Setup pre-commit](#)
 - [Setup the CI linting pipeline](#)
 - [Conclusion](#)

Tasks 1,2, and 3 are more or less independent of each other and can be done in the order you like.

Initial task: Create the frontend directory

First of all, let's create the frontend directory.

- `mkdir frontend`

And go in this directory `mkdir frontend`, all the reste of the worklab will be done inside this directory.

Task 1: Setup the frontend skeleton

directory creation, project initialization

Let's create our app skeleton with `uv`. We will use python 3.12 to work inside it.

- `uv init --app --python 3.12.`

You will see that `uv` has created a basic directory skeleton with:

1. a `.gitignore` file,
2. a `.python-version` file,
3. a `hello.py` python script,
4. a `pyproject.toml` TOML file,
5. a `README.md` markdown file.

The `.gitignore` is related to git and allows us to list all the files we do not want to track, like files coming from compilation, tests reports, credentials, large files and so on. More on that in the Task 1 if you haven't started by it. This also means that `uv` has already initialized a local git repository.

The `.python-version` is an internal `uv` file that permits it to track the version of python that has been used to create this app directory. You shouldn't touch it.

The `README.md` file is where you should put everything that is relevant to the project. How to run it, what's its purpose, basic documentation, how is it tested and so on.

The `hello.py` and `pyproject.toml` files **represent a basic python app**. As it is right now, although not really interesting, this is a fully functional python app. You can see this by running `python hello.py` in your terminal.

While we're at it, let's create our makefile right now. Create a file called `makefile`, either via your IDE UI or with the terminal with the command

- `touch makefile`.

As said earlier, a makefile is a great way to centralize every command that your project might need. Just for the test, let's write the command to call our `hello.py` module.

```
.PHONY: hello
hello:
    python hello.py
```

The `.PHONY` keyword is here to ensure that your shell clearly understands that the command you are writing, `make hello`, refers to a command of your makefile and not to a command of your OS.

- `make hello` will then run `python hello.py`.

While not really useful here, when you have multiple commands with multiple arguments, the `makefile` starts to be really useful.

Now, let's start writing our frontend.

Dependencies installation

We will use `streamlit` to build our frontend. This python library allows us to write frontend in pure python. While not as powerful as a full fledged frontend framework like `vue.js`, `react`, or `angular`, the learning curve is less abrupt and it's a great library to write Proofs Of Concept.

Let's add it to our project, remember that with `uv` you have to type the following command.

- `uv add streamlit`.

This first command has done three things.

1. It has updated the `pyproject.toml` with the following section.

```
dependencies = [  
    "streamlit>=1.41.1",  
]
```

Stating that starting from now, your app will need to install a version of streamlit superior or equal to `1.41.1` to work.

2. It has created a `uv.lock` file. This is an internal file to `uv` which tracks all the subdependencies installed when you install the "real dependencies" that are stored in the `dependencies` section of the `pyproject.toml` file. This allows `uv` to have reproducible build and you should not touch it manually.
3. It has created the python virtual environment dedicated to this project with the `.venv` directory. By default, all `uv` commands run in this virtual environment, whether you activate it or not.

Questions:

- Say you want to run a command that needs the virtual environment to be activated and you do not want, or can't, use `uv`. How do you activate it ?
- You want to install the version `1.40.0` of streamlit. How would you `remove` the `1.41.1` version and `add` this specific version ?

Let's start working on our frontend. Create a `main.py` file at the root of the `frontend` directory and let's work in it.

Frontend creation

By convention, the usual abbreviation to work with `streamlit` is `st`. Let's import it.

```
import streamlit as st
```

Use `st.set_page_config` to set the title and icon for your Streamlit app and add a title to your app using `st.title`.

Python

```
import streamlit as st  
  
# Set page title
```

```
st.set_page_config(
    page_title="JuniaGPT",
    page_icon="🚀",
)

st.title("JuniaGPT 🚀")
```

Just with this, you already have a frontend. You can run it in your browser in two ways, whether or not you want to activate your virtual environment.

1. You do not want to activate your venv, you can run the following command in your terminal: `uvx streamlit run main.py`.
2. You have activated your virtual environment, you can then simply use `streamlit run main.py`.

Questions:

1. `uvx` is similar to the `uv` command `uv run`. In fact, you can also run your frontend with the command `uv run streamlit run main.py`. What's the difference between these two commands?
2. By default, streamlit runs on port `8501`. If this port is already used on your laptop, what would the extra argument to pass to the above command to change the port where the streamlit frontend is running?
3. Write a command in your `makefile` to run the frontend.

First steps to a chatbot

We want to be able to discuss with an LLM endpoint, so we need to have a conversation window. This can be done with the following streamlit functions.

- `st.chat_message`
- `st.markdown`
- `st.chat_input`

What do we want? We want to be able to have a conversation with an LLM endpoint, there will be two protagonists:

- the `user`: yourself,
- the `ai/assistant`: the LLM endpoint that will answer to your question.

First, let's make sure we can enter some inputs on our frontend and display it.

```
# Query user prompt
if prompt := st.chat_input("What is your question?", key="user_prompt"):
    with st.chat_message("user"):
        st.markdown(prompt)
```

This is the bare minimum to be able to display some text to your frontend. But this is quite limited, there is no history, everything you write overwrites the last thing. You are now talking to yourself, alone.

Questions:

- Look into the `st.chat_message` api documentation, on the streamlit website and see if you can change the avatar.
- What happens if you remove the context manager with `st.chat_message("user")`: and just write the following.

```
# Query user prompt
if prompt := st.chat_input("What is your question?", key="user_prompt"):
    st.markdown(prompt)
```

Let's now add a parrot. We are going to add a new `chat_message` box, the one for the future ai endpoint. for now it will only repeat what you've written. But you'll be a bit less lonely.

```
# Query user prompt
if prompt := st.chat_input("What is your question?", key="user_prompt"):
    with st.chat_message("user"):
        st.markdown(prompt)

    with st.chat_message("assistant"):
        st.markdown(prompt)
```

With this, you now have a dummy AI chatbot that repeats everything you say. The future seems bright.

Let's now add some history capacity.

Adding chat history

To deal with history, `streamlit` uses what it calls a `session_state`. This is a key/value cache which persists for each user session.

You can access the `session_state` simply by typing `print(st.session_state)` in your frontend and it will appear in your terminal. There not much in it for now.

We are going to store our conversation history in this `session_state`, let's initialize a `messages` key in this dict.

Above the query prompt, write the following.

```
# Initialize chat history and variables
if "messages" not in st.session_state:
    st.session_state.messages = []
```

How does a conversation history works when we work with LLMs ? When you talk with an LLM, all your questions and its answers are stored in a `json` format as a list of dictionnaires like this.

```
[
  {
    "role": "user",
    "content": "your_question"
  },
  {
    "role": "assistant",
    "content": "the_llm_answer"
  },
  {
    "role": "user",
    "content": "your_next_question"
  },
  {
    "role": "assistant",
    "content": "the_next_llm_answer"
  },
]
```

etc. The `role` and `content` keys are standards and are fixed, the `user` value always represents the human, and the `assistant` value always represents the AI. This is thanks to this structure that you are able to ask questions like `what was my previous question ?` to an LLM and it's able to answer it.

So, to be able to do that, we will store our conversation in this format in the `session_state`, and loop over it to display the full history of our conversation.

```
st.set_page_config(
    page_title="JuniaGPT",
    page_icon="🚀",
)

st.title("JuniaGPT 🚀")

# Initialize chat history and variables
if "messages" not in st.session_state:
    st.session_state.messages = []

# Write queries and answers onto the page
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Query user prompt
if prompt := st.chat_input("What is your question?", key="user_prompt"):
    # store the new input from the user in the session_state "messages" key
    st.session_state.messages.append({"role": "user", "content": prompt})

    with st.chat_message("user"):
        st.markdown(prompt)
```

```
with st.chat_message("assistant"):  
    # let's do that for now, this will be useful later on when we'll  
add the LLM endpoint  
    response = prompt  
  
    st.markdown(response)  
  
    # store the new response from the user in the session_state  
"messages" key  
    st.session_state.messages.append(  
        {"role": "assistant", "content": response},  
    )
```

We still have a parrot, but now we have a conversation history. But now we have everything ready to add an LLM endpoint.

First, let's talk about Ollama.

Ollama, or how I stopped worrying about a ChatGPT subscription and started using llms locally

Ollama is an open-source platform that allows you to run large language models (LLMs) locally on your own computer.

What it does:

- **Local Execution:** Instead of relying on cloud services, Ollama enables you to run LLMs directly on your machine (laptop, desktop, etc.).
- **Accessibility:** It simplifies the process of running LLMs, making them more accessible to individuals who may not have the technical expertise or resources to set up complex infrastructure.
- **Customization:** Ollama provides a degree of control over the models, allowing for customization and experimentation.
- **Data Privacy:** By running LLMs locally, you maintain greater control over your data and privacy, as your information isn't transmitted to external servers.

How to run it:

Installation

- [Download the appropriate version of Ollama for your operating system](#) (Windows, macOS, Linux) from the official website.
- Follow the installation instructions provided.

Model Selection

- Ollama supports a variety of pre-trained LLM models. You can see them in the [Models](#) part of their website.
- Choose a model based on your needs and the available resources (CPU/GPU). Smaller models generally have lower resource requirements. Since we want it to be completely local, we will work

with what we call a **Small Language Model**, you have a large choice like `phi3`, `phi4`, developed by Microsoft, or `llama3` developed by Meta, etc.

Run a Model

- Once you have installed Ollama, the easiest way to interact with a model is with the command `ollama run <your_model>` in your terminal. For example `ollama run phi4` will let you interact with the `phi4` model.

Question:

- Look at the model catalog of Ollama, select one or two different models and try to interact with them.

Start Ollama as a server

- To start ollama without running the desktop application, you use the command `ollama serve`.

If you see an error like `"Error: listen tcp 127.0.0.1:11434: bind: address already in use"` on Linux, this is because once you've run the command `ollama run ...` the Ollama server is added as a system service. To stop it you can run `systemctl stop ollama`, maybe with a `sudo`, but this is not necessary. On Linux, you can see its status by typing the command `systemctl status ollama`.

- Once the Ollama server is up and running, you can download a model to interact with with the command `ollama pull <your_model>`.

Interact with the Model:

To interact with Ollama in the server mode, we will have to send him POST requests. For LLMs, there are two ways to interact with them.

1. **Completion:** You send a single question to the LLM and it answers it, without notion of conversation history.
2. **Chat completion:** You have a conversation with the LLM, with history notion, and its answer can be based on your previous questions.

We are interested with the second option.

Questions:

Look at the [api documentation of Ollama](#).

- To interact with the server in a chat completion way, we have to send it some `curl` request. What will be the structure of this request ?
- The answer of an LLM is determined by a parameter called the temperature t , which is a float $0 \leq t \leq 1$. Where in the `curl` request do I have to put the temperature to modify the answer ?
- How can I modify the choice of model in the `curl` request ?

Ok, now that we have setup locally an Ollama server, we want to interact with it via our frontend. To do that we will have to code an HTTP client to interact with the server. This will be done with the combination of two python libraries:

- [HTTPX](#)
- [Pydantic](#)

The frontend HTTP client, how to use HTTPX and pydantic

HTTPX

HTTPX is a powerful and modern Python library for making HTTP requests, offering a combination of performance, flexibility, and ease of use.

- **Modern and Asynchronous:** HTTPX is built on top of the asyncio library, making it inherently asynchronous. This allows for concurrent requests, significantly improving performance in applications with multiple HTTP interactions.
- **Versatile and Feature-Rich:** It supports various HTTP methods (GET, POST, PUT, DELETE, etc.), handles different content types (JSON, XML, text, etc.), and provides features like:
 - **Connection pooling:** Reuses existing connections to the same server, reducing latency.
 - **Automatic decompression:** Handles gzip and deflate compression automatically.
 - **Streaming:** Supports both sending and receiving data in chunks, useful for large files.
 - **HTTP/1.1 and HTTP/2 support:** Adapts to the server's capabilities.
 - **TLS/SSL support:** Securely handles HTTPS connections.
- **User-Friendly API:** HTTPX offers a clean and intuitive API, making it easy to use for both simple and complex requests.
- **Extensible:** You can extend HTTPX's functionality with plugins for features like:
 - **Authentication:** Integrate with various authentication mechanisms (e.g., OAuth2).
 - **Retries:** Implement automatic retries on failed requests.
 - **Caching:** Cache responses to improve performance and reduce server load.
- **Well-maintained and Active Community:** HTTPX is actively maintained and has a growing community, ensuring ongoing development and support.

Pydantic

Pydantic helps you write more robust and maintainable code by ensuring that your data is always valid and in the expected format. It improves the overall quality and reliability of your applications.

- **Data Validation and Parsing:** Pydantic is primarily used for data validation and parsing. It allows you to define data structures (models) using Python type hints. When you create an instance of a model, Pydantic automatically validates the provided data against the defined types and constraints.
- **Data Serialization:** Pydantic can easily serialize your data structures to and from various formats like JSON, dictionaries, and more. This simplifies data exchange between different parts of your application or with external systems.
- **Rich Type Support:** Pydantic supports a wide range of data types, including:
 - **Built-in types:** int, float, str, bool, list, dict, etc.

- **Custom types:** You can create your own custom data types to represent complex structures.
- **Third-party types:** Integrate with other libraries like datetime, UUID, and more.
- **Data Constraints:** You can define constraints on your data, such as:
 - Required fields: Ensure that certain fields are always provided.
 - Default values: Specify default values for optional fields.
 - Length restrictions: Enforce minimum and maximum lengths for strings.
 - Value ranges: Define allowed ranges for numbers.
 - Regular expressions: Validate strings against specific patterns.
- **User-Friendly and Extensible:** Pydantic provides a user-friendly API and is highly extensible. You can easily create custom validators and integrate Pydantic with other parts of your application.

Setup of the REST service

First of all, in the root of the frontend directory, let's create a subdirectory called `rest`, and the three following python files.

- `__init__.py`
- `rest/__init__.py`
- `rest/service.py`

Your frontend directory should have the following structure.

```
.
├── hello.py
├── __init__.py
├── main.py
├── makefile
├── pyproject.toml
├── README.md
├── rest
│   ├── __init__.py
│   └── service.py
└── uv.lock
```

To interact with an LLM via Ollama in a chat completion way, we need to at least have the 3 following parameters.

- The model you want to interact with.
- The temperature you set.
- The conversation history and your new question.

We will wrap the three parameters in a data model with the pydantic `BaseModel` class. Then we will use this class to interact with the Ollama server via a request done with the HTTPX library.

First we will add `httpx` and `pydantic` to our project with `uv`.

- `uv add httpx`

- `uv add pydantic`
- `uv sync`

The `uv sync` command here is used to update our virtual environment with the newly added libraries.

Let's populate our `rest/service.py` file. Start by adding the `Chat` class to be able to converse with Ollama.

```
import httpx
from pydantic import BaseModel, Field

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]]
```

To define a `BaseModel` with `pydantic`, what you do is define the parameters and their type. The `Field` function here is used to define some constraints, default value for our temperature parameter.

Let's also add the skeleton of our http client.

```
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]:
        """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.
```

```

        """
        pass

    def post(
        self,
        chat: Chat,
    ):
        """POST request."""
        pass

```

This class is a wrapper around the `httpx.Client` class with two functions, one internal `_generate_request`, and one we will use to interact with Ollama: `post`.

The `_generate_request` will take the parameters we have passed to the `Chat` class to generate the header, body, and url of our request.

```

def _generate_request(self, chat: Chat) -> tuple[dict, dict, str]:
    """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

```

The `post` function will send our questions to the LLM, with some error catch. Again, this is mainly a wrapper around the `client.post` function of HTTPX. Look at the [Exceptions](#) of HTTPX to understand the `try...except` block. We will talk more about http status code later on.

```

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

```

At the end, your `rest/service.py` script should look like that.

```

import httpx
from pydantic import BaseModel, Field

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]]

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")

```

With a client class initialized.

What we need now is to update the `main.py` script of our frontend to communicate with the LLM.

Frontend update

Let's update the import of `main.py`.

```

import streamlit as st

from rest.service import Chat, client
import httpx

```

We want to be able to modify the temperature parameter of our questions. To do so, streamlit allows us to define `radio buttons` to be able to select between a fixed list of choices. Since we want to keep the middle of the page for our chat, we will also use the possibility to define a `sidebar column`.

```

# Define temperature labels and corresponding values
temperature_mapping = {"Accurate": 0, "Balanced": 0.7, "Creative": 1}
# Let the user chose the temperature category he wants
temperature_choice = st.sidebar.radio(
    label="Model Behavior",
    options=temperature_mapping.keys(),
    index=1,
)
# get the float value associated
temperature = temperature_mapping.get(temperature_choice)

```

Note that we could use a `slider` instead of radio buttons, but this seems less practical.

Now let's update our parrot. The model here is `phi4` but if you have chosen another model, update accordingly.

We populate our `Chat` class with our parameters: model, temperature, and history conversation. and we pass it to the LLM via the `client.post` method.

We check if the return http code is ok, ie a 200 code, if so we retrieve the answer of the LLM `response.json()["message"]["content"]` part of its answer, recall the [structure of a response via the Ollama api](#).

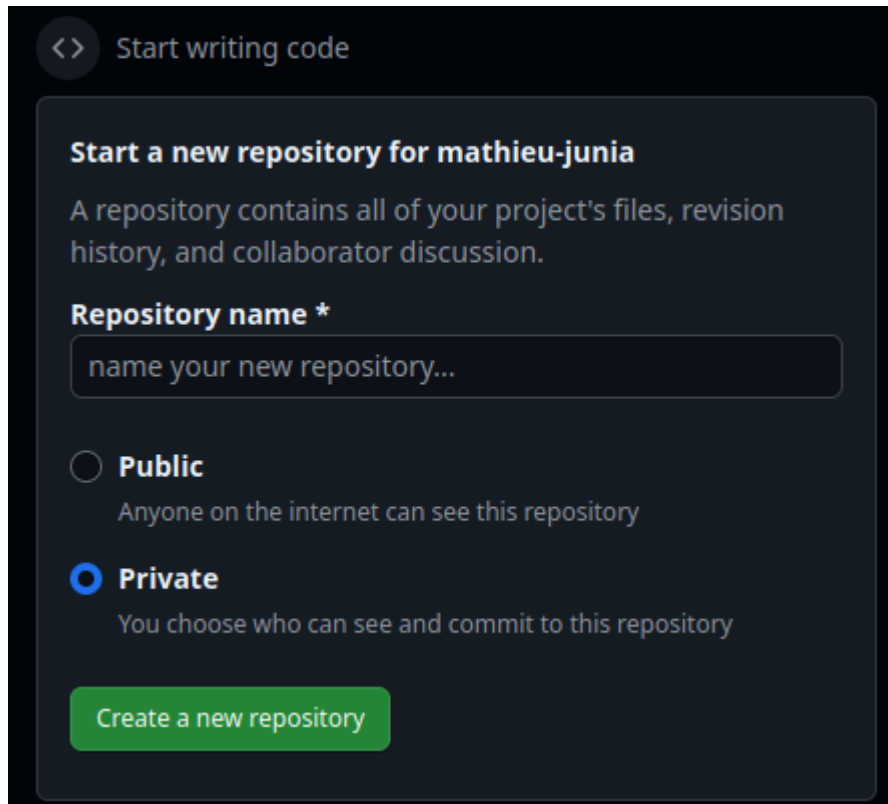
we print it, and finally add it to the history conversation. If something went wrong, we print it.

```
with st.chat_message("assistant"):  
    chat = Chat(  
        model="phi4",  
        temperature=temperature,  
        messages=st.session_state.messages,  
    )  
  
    response = client.post(chat=chat)  
  
    if response.status_code == httpx.codes.OK:  
        message = response.json()["message"]["content"]  
        st.markdown(message)  
  
        st.session_state.messages.append(  
            {"role": "assistant", "content": message},  
        )  
  
    else:  
        st.write("It seems that something broke down 😊")  
        st.write(response.status_code)
```

Congratulations, you now have a fully working chatbot, the easy part is finished.

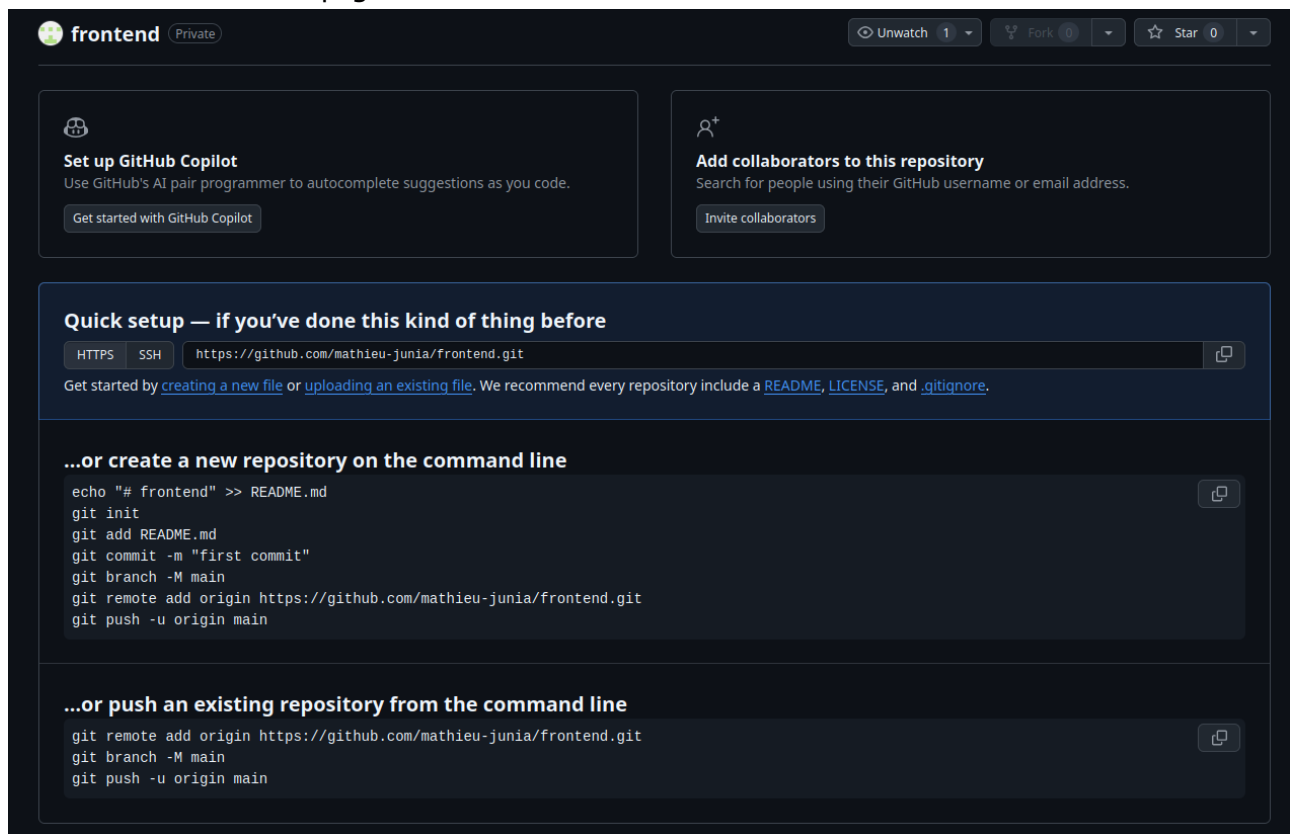
Task 2: Setup the remote git repository

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



The screenshot shows the GitHub interface for creating a new repository. At the top, there is a button labeled 'Start writing code'. Below it, the heading 'Start a new repository for mathieu-junia' is displayed. A descriptive text states: 'A repository contains all of your project's files, revision history, and collaborator discussion.' The 'Repository name' field is required and contains the placeholder text 'name your new repository...'. There are two radio button options: 'Public' (with the description 'Anyone on the internet can see this repository') and 'Private' (with the description 'You choose who can see and commit to this repository'). The 'Private' option is selected. At the bottom, there is a green button labeled 'Create a new repository'.

- You should arrive to this page.



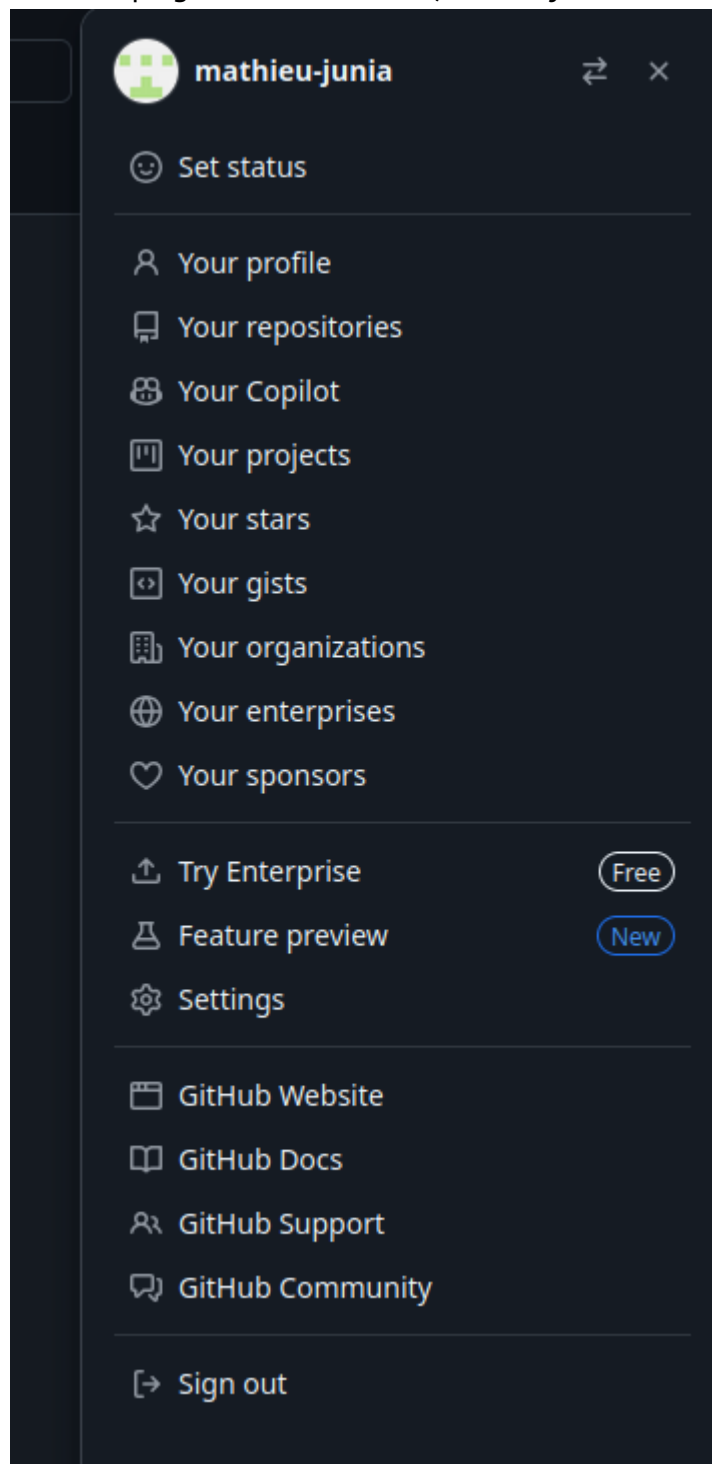
Now before pushing to your remote repository, you need a way to authenticate to GitHub from the terminal. There are usually two possible ways:

- SSH keys
- Personal Access Token

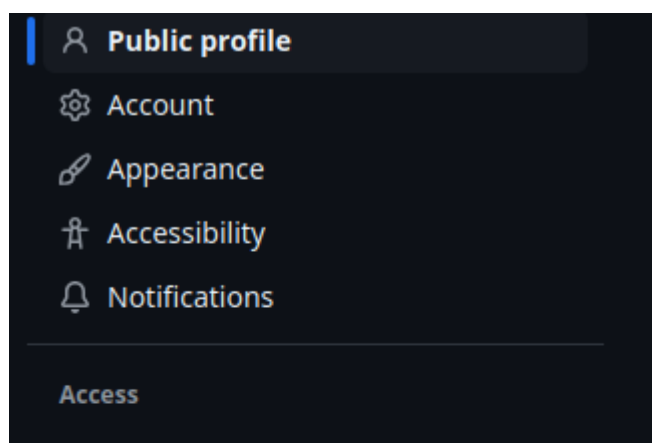
SSH keys allow for a seamless way to push content as you do not have to remember and store the access token, but many companies use private virtual network and as such have shut down the ways to use SSH.

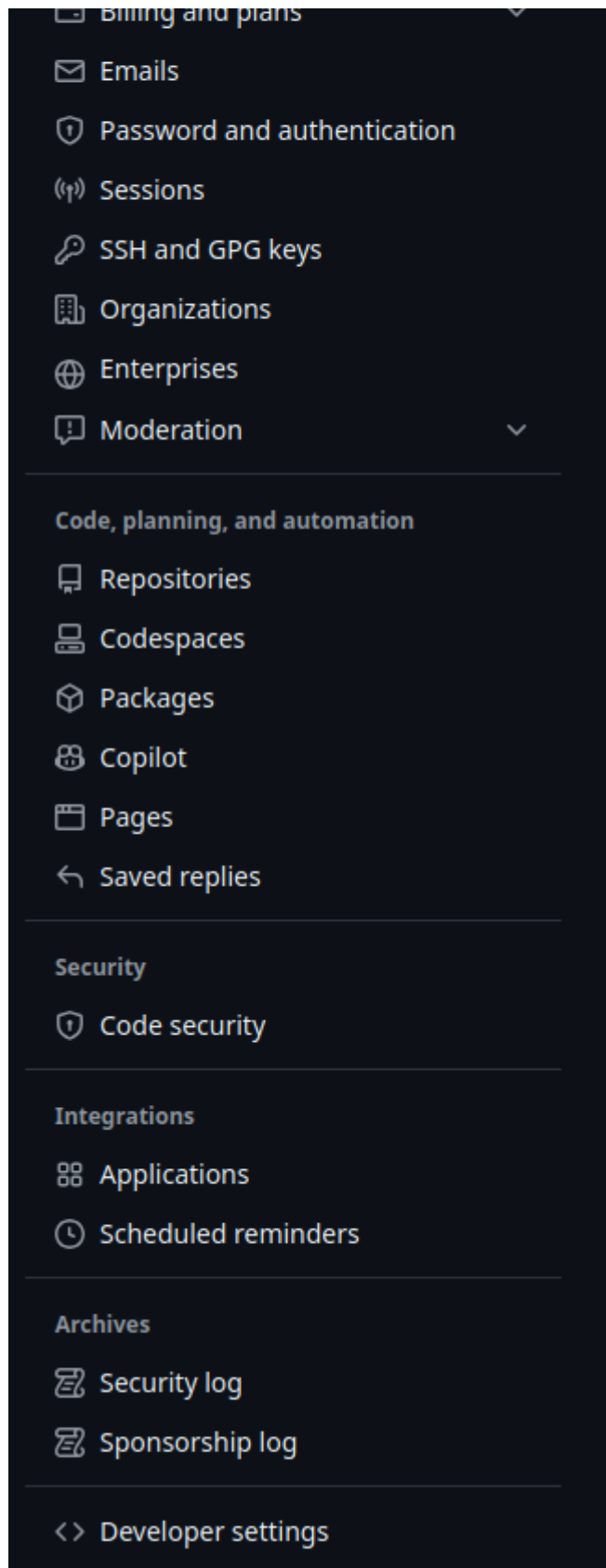
Let's create a Personal Access Token (PAT).

1. On the top right corner of GitHub, click on your icon and select **Settings**.

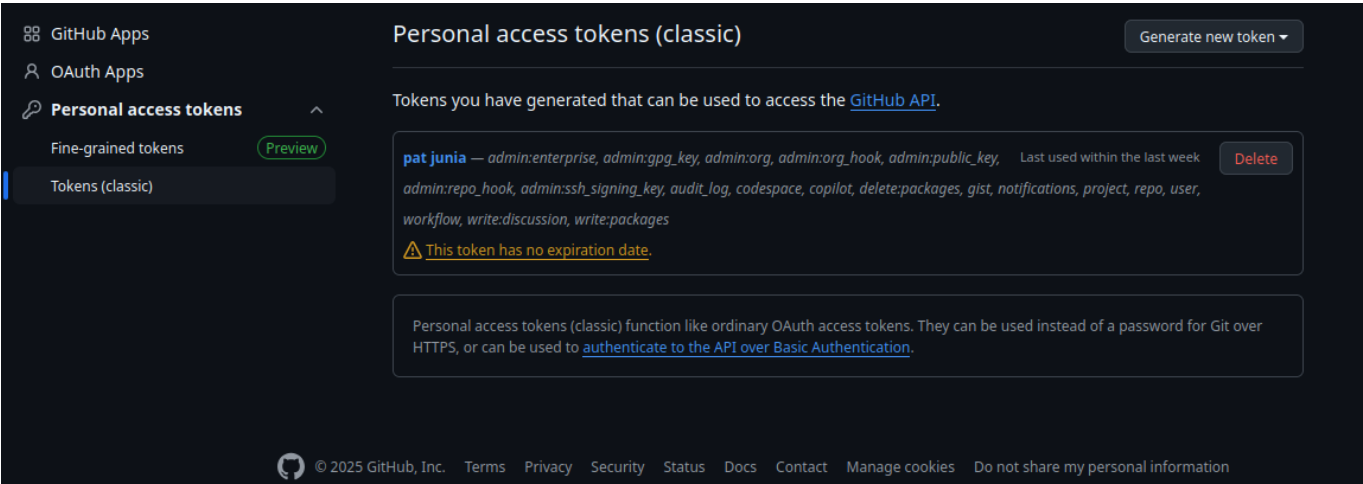


2. On the left menu, go all the way down to **Developer settings**.



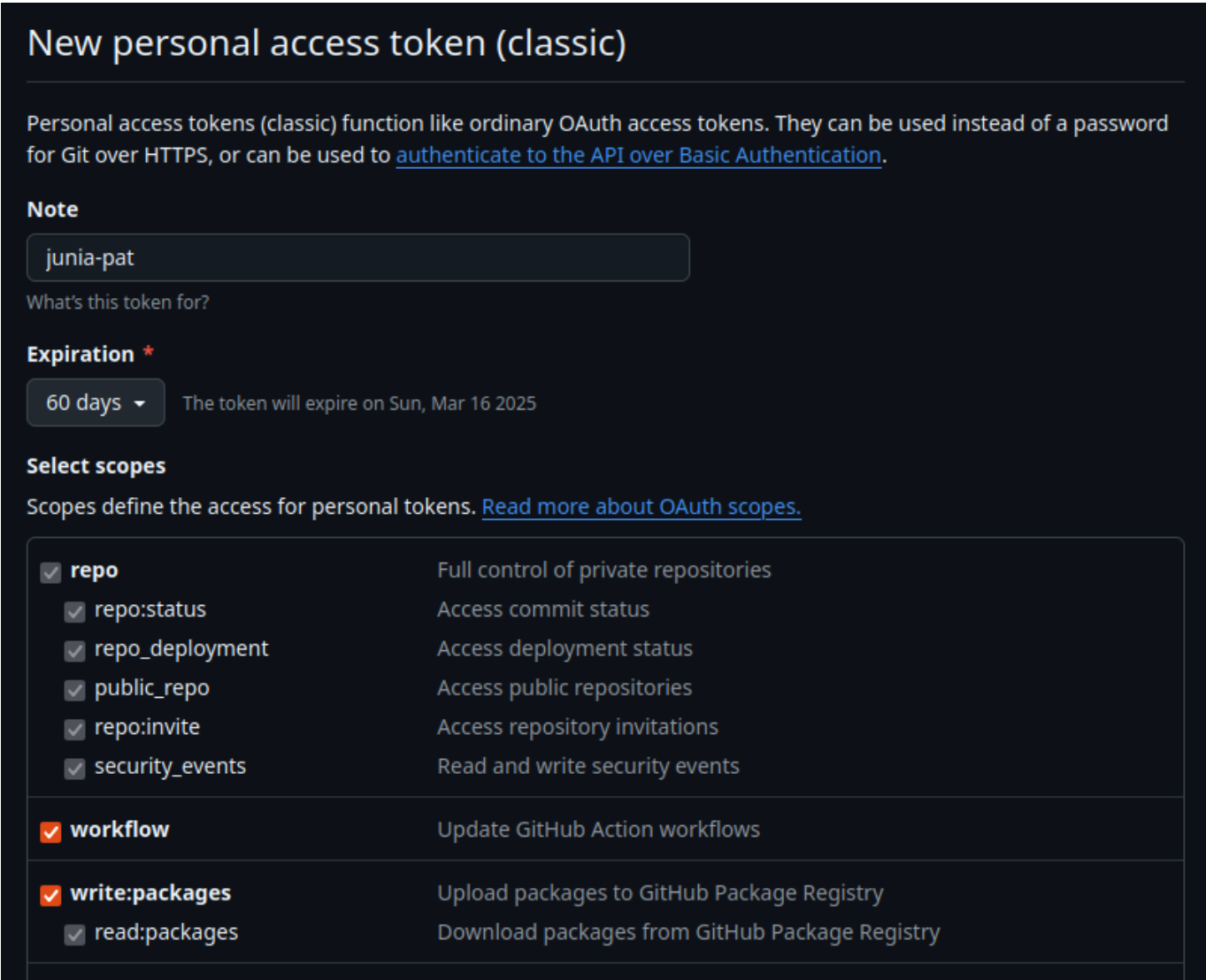


3. Select `Personal access tokens -> Tokens (classic)` and click on `Generate new token -> Generate new token (classic)`.



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 frontend to the remote repository.

- If you have started by this task, you should initialize your local git repository.

At the root of the `frontend` directory, type the following command.

- `git init`

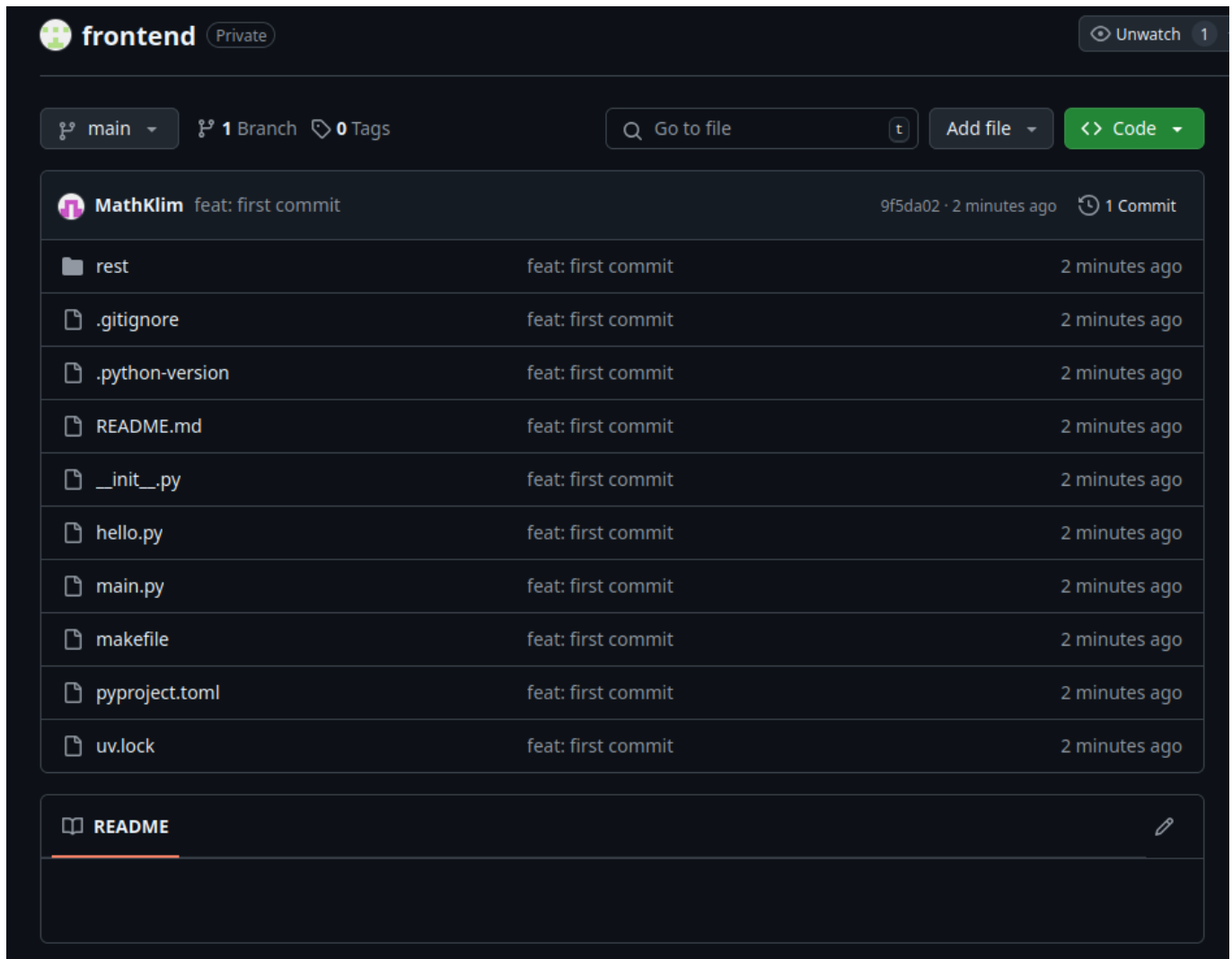
Since we cannot push an empty repository to GitHub, we will follow the first choice.

```
echo "# frontend" >> README.md
git add README.md
git commit -m "feat: first commit"
git branch -M main
git remote add origin https://github.com/your_github_repo_address
git push -u origin main
```

- If you have started by the development of the frontend, you can follow the second choice.

```
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 frontend directory pushed to your GitHub repository.



Task 3: 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.

```
# 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.

```
# 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, wherever 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 `frontend` 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 push 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 on the `Actions` section of your repository.

Conclusion

You have:

- Setup a project
- Develop a frontend for a chatbot
- Setup the minimal pre-commit configuration
- Stored your code in a remote git repository
- Setup a minimal CI pipeline