FRONTEND

Overview

This project is a frontend application built with Node.js. Follow the steps below to set it up and run it locally.

Prerequisite

- Node.js (Ensure it is installed on your machine)

Getting Started

1. Clone the repository (if you haven't already):
2. git clone amos2025ss04-ai-driven-testing/frontend

cd amos2025ss04-ai-driven-testing/frontend

3. Navigate to the frontend directory:

cd frontend/

4. Install dependencies:

npm install

5. Start the server:

npm run start

6. Open your browser and go to:
7. http://localhost:3000/

BACKEND

This project allows you to easily run a local Ollama container, send prompts to a language model via a Dockerized API, and save the structured response as Markdown.

Requirements

- **Docker** (for running the Ollama container)
  ➔ Install Docker
- **Conda** (optional, for managing the Python environment)
  ➔ Install Anaconda

Files Overview

- environment.yml — Conda environment definition
- main.py — Main script to run a single model: starts the container, sends prompt, and stops the container.
- example_all_models.py — Example script that sends the same prompt to all allowed models.

- llm_manager.py — Handles Docker container management, pulling models/images, sending prompts, and progress reporting.
- allowed_models.json — Config that defines allowed language models.
- prompt.txt — Default input prompt file.
- output-<MODEL_ID>.md — Output file produced for each model.

All files are located inside the backend/ directory.

Setup

1. **(Optional)** Create and activate a Conda environment:
2. conda env create -f backend/environment.yml

conda activate backend

3. Make sure Docker is running on your machine.

Usage

Simply run the main.py script:

python backend/main.py

By default, it reads from backend/prompt.txt, uses the Mistral LLM and writes to backend/output-mistral_7b-instruct-v0.3-q3_K_M.md.

Optional Arguments:

You can also specify a custom prompt file and output file:

python backend/main.py --model 0 --prompt_file ./your_prompt.txt --output_file ./your_output.md

Running All Models

python backend/example_all_models.py

This script does the following:

- Starts each model's container
- Sends the provided prompt (from prompt.txt)
- Saves each response into its own output-<MODEL_ID>.md
- Stops all containers after completion

How It Works

1. The project uses the Docker image ollama/ollama to run language models locally.
2. The LLMManager class in llm_manager.py:
    - Pulls the required Docker image with progress indication.
    - Selects a free port for each container.

    - Waits until the container’s API becomes available.
    - Pulls the selected model inside the container.
    - Sends user prompts to the model endpoint and writes the Markdown-formatted response.
3. allowed_models.json provides a list of allowed models.

Note

- The script automatically pulls the necessary Docker image and model if not already available.
- Each container is started on a free port; the API endpoint for each model is managed automatically.
- On completion, each container is stopped to free up system resources.
- The response is formatted as clean Markdown.

Example

If your prompt.txt contains:

~~~
Write unit tests for the following Python function:


```python
def add_numbers(a, b):
  """
  Adds two numbers together and returns the result.


  Args:
    a (int or float): The first number.
    b (int or float): The second number.


  Returns:
    int or float: The sum of a and b.


  Examples:
    >>> add_numbers(2, 3)
    5
    >>> add_numbers(-1, 1)
    0
~~~

```
    >>> add_numbers(0.5, 0.5)

    1.0

  """

  return a + b
```

Your output.md will look like:

Here is how you can write unit tests for the `add_numbers` function using Python's built-in unittest module and some assertions to check if your code works as expected with test cases from examples provided in docstring.

Make sure that all import statements are correct, including 'unittest'. This example assumes you want a simple set of tests for this specific function:

```python
import unittest

from add_numbers import add_numbers # assuming the file name is "add_numbers" and it's located in same directory as script or pass full path to where your module resides.


class TestAddNumbers(unittest.TestCase):

  def test_positive_integers(self):

     self.assertEqual(add_numbers(2,3),5) # should return the sum of two numbers (i.e., '4') as output: 7 not ('6'). Therefore it fails with this assertion error by comparing actual and expected result here respectively    which is correct i means its working fine


  def test_negative_integers(self):

     self.assertEqual(add_numbers(-1,1),0) # should return the sum of two numbers (i.e., '2') as output: -3 not ('-4'). Therefore it fails with this assertion error by comparing actual and expected result here respectively    which is correct i means its working fine


  def test_decimal(self):

     self.assertEqual(add_numbers(0.5, 2),1) # should return the sum of two numbers (i.e., '3') as output: -4 not ('-8'). Therefore it fails with this assertion error by comparing actual and expected result here respectively    which is correct i means its working fine


if __name__ == "__main__":

  unittest.main() # running all tests in the script  (this line should be at end of your file) if it was a standalone module to run only those test that are above 'TestAddNumbers' otherwise, it will not work because you cannot directly execute python code when this is included as part of another program.
```

This unit tests assumes all numbers being added together must result in the expected sum (positive integers and negative integer). If your use-case might include other inputs such be positive/negative or decimal figures too, then those additional test cases should also exist for that purpose to ensure robustness against edge scenarios – as per requirement.

Requirements

- **Docker** (for running the Ollama container)
  ➔ Install Docker
- **Conda** (optional, for managing the Python environment)
  ➔ Install Anaconda

Files Overview

- api.py — Fast API wrapper
- main.py — Main script to run a single model: starts the container, sends prompt, and stops the container.
- example_all_models.py — Example script that sends the same prompt to all allowed models.
- llm_manager.py — Handles Docker container management, pulling models/images, sending prompts, and progress reporting.
- allowed_models.json — Config that defines allowed language models.
- prompt.txt — Default input prompt file.
- output-<MODEL_ID>.md — Output file produced for each model.

All files are located inside the backend/ directory.

Setup

1. **(Optional)** Create and activate a Conda environment:
2. conda env create -f environment.yml

conda activate backend

3. Make sure Docker is running on your machine.

Usage

Simply run the main.py script:

python backend/main.py

By default, it reads from backend/prompt.txt, uses the Mistral LLM and writes to backend/output-mistral_7b-instruct-v0.3-q3_K_M.md.

Optional Arguments:

You can also specify a custom prompt file and output file:

```
python backend/main.py --model 0 --prompt_file ./your_prompt.txt --output_file ./your_output.md
```

Running the Local API

This project ships with a very small FastAPI wrapper (backend/api.py) that exposes your local Ollama models through HTTP so a future UI can consume them.
Follow the steps below to get it up and running.

1 – Start the server

```
cd backend
```

```
uvicorn api:app --reload          # --port 8000 by default
```

--reload enables hot-reload during development; omit it in production.

You should see something like:

```
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

2 – Available endpoints

Method	Path	Purpose	Body / Query
GET	/models	List all allowed models + whether running	–
POST	/prompt	Ensure container is running, send prompt	{ "model_id": "<id>", "prompt": "<text>" }
POST	/shutdown	Stop & remove a model container	{ "model_id": "<id>" }

Open the automatically generated Swagger UI at:

http://127.0.0.1:8000/docs

Running All Models

```
python backend/example_all_models.py
```

This script does the following:

- Starts each model's container
- Sends the provided prompt (from prompt.txt)
- Saves each response into its own output-<MODEL_ID>.md
- Stops all containers after completion

How It Works

1. The project uses the Docker image ollama/ollama to run language models locally.

2. The LLMManager class in llm_manager.py:
    - Pulls the required Docker image with progress indication.
    - Selects a free port for each container.
    - Waits until the container's API becomes available.
    - Pulls the selected model inside the container.
    - Sends user prompts to the model endpoint and writes the Markdown-formatted response.
3. allowed_models.json provides a list of allowed models.

Note

- The script automatically pulls the necessary Docker image and model if not already available.
- Each container is started on a free port; the API endpoint for each model is managed automatically.
- On completion, each container is stopped to free up system resources.
- The response is formatted as clean Markdown.

Example

If your prompt.txt contains:

~~~
Write unit tests for the following Python function:


```python
def add_numbers(a, b):
    """
    Adds two numbers together and returns the result.


    Args:
        a (int or float): The first number.
        b (int or float): The second number.


    Returns:
        int or float: The sum of a and b.


    Examples:
        >>> add_numbers(2, 3)
~~~

```
    5
    >>> add_numbers(-1, 1)
    0
    >>> add_numbers(0.5, 0.5)
    1.0
  """
  return a + b
```

Your output.md will look like:

Here is how you can write unit tests for the `add_numbers` function using Python's built-in unittest module and some assertions to check if your code works as expected with test cases from examples provided in docstring.

Make sure that all import statements are correct, including 'unittest'. This example assumes you want a simple set of tests for this specific function:

```python
import unittest
from add_numbers import add_numbers # assuming the file name is "add_numbers" and it's located in same directory as script or pass full path to where your module resides.


class TestAddNumbers(unittest.TestCase):
  def test_positive_integers(self):
      self.assertEqual(add_numbers(2,3),5) # should return the sum of two numbers (i.e., '4') as output: 7 not ('6'). Therefore it fails with this assertion error by comparing actual and expected result here respectively    which is correct i means its working fine


  def test_negative_integers(self):
      self.assertEqual(add_numbers(-1,1),0) # should return the sum of two numbers (i.e., '2') as output: -3 not ('-4'). Therefore it fails with this assertion error by comparing actual and expected result here respectively    which is correct i means its working fine


  def test_decimal(self):
      self.assertEqual(add_numbers(0.5, 2),1) # should return the sum of two numbers (i.e., '3') as output: -4 not ('-8'). Therefore it fails with this assertion error by comparing actual and expected result here respectively    which is correct i means its working fine


if __name__ == "__main__":
```

unittest.main() # running all tests in the script (this line should be at end of your file) if it was a standalone module to run only those test that are above 'TestAddNumbers' otherwise, it will not work because you cannot directly execute python code when this is included as part of another program.

This unit tests assumes all numbers being added together must result in the expected sum (positive integers and negative integer). If your use-case might include other inputs such be positive/negative or decimal figures too, then those additional test cases should also exist for that purpose to ensure robustness against edge scenarios – as per requirement.