# Installation

To be able to run this app locally, you will need both Docker and Git installed on your machine. To check if you have each installed, run “docker” and “git” in your terminal separately. If you receive an output that includes “not found” or “not recognized”, then the software needs to be installed.

### Mac with intel chip

1. **Docker:** link
2. **Git:** If not preinstalled, running git --version in the command line will guide you through installing it.

### Mac with M1/M2 chip

1. **Docker:** link
2. **Git:** If not preinstalled, running git --version in the command line will guide you through installing it.

### Windows

1. **WSL:**
    - Open “command prompt”
    - Run the following command: *wsl –update*
    - Wait for it to complete (reach 100%)
2. **Docker**: link
3. **Git**: link

# Quick start

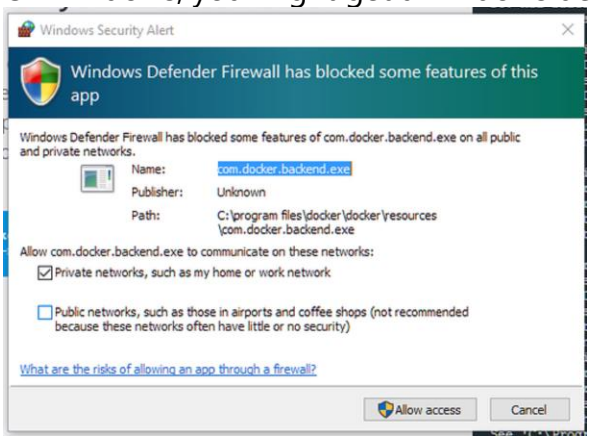
1. **Make sure Docker Desktop is open, if not then open it.** Keep it open and running in the background. You may minimize it.
2. **Clone the app (you only need to do this once):**
    1. Open “terminal” on mac or “command prompt” on windows
    2. Inside the terminal, run this command (paste it and press enter):

        *git clone https://github.com/ralphr123/bex-workshop && cd bex-workshop*
3. **Run the app:** In the same terminal, run the following command:
    1. For windows:

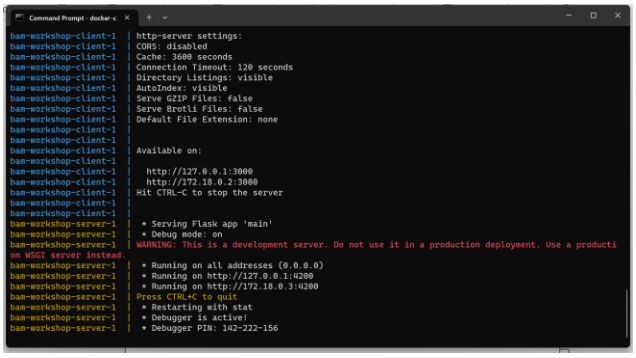
        *set PASSPHRASE=<redacted> && docker-compose up*
    2. For Mac/Linux:

        *export PASSPHRASE=<redacted> && docker-compose up*
4. **Wait for the terminal to stop loading.** This may take several minutes or more.

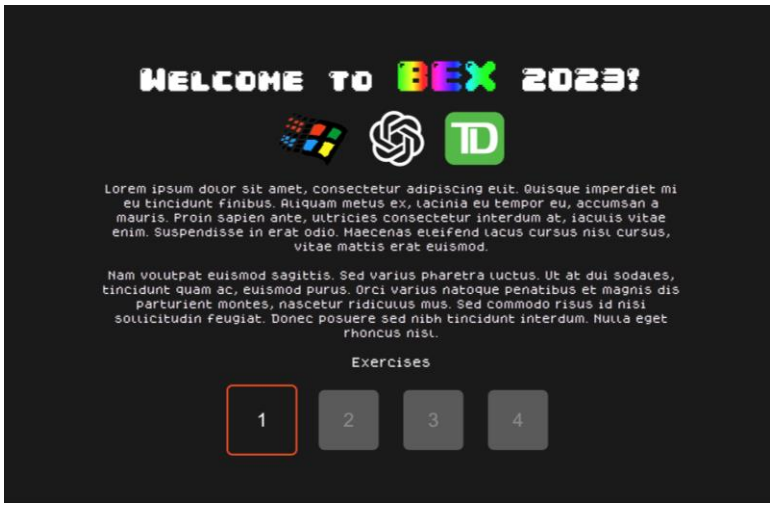
    On windows, you might get a windows defender popup. Click *Allow access*.




You will see **red text** once the app is loaded and ready:



Once the terminal area is finished loading, open http://localhost:3000 on your browser to access the app. If you see a page similar to the following, you're all set up! Otherwise, free to ask for help.




Once you see this page, all setup is complete. Any changes you make to the code will automatically be reflected on website itself, so there is no need to restart the app or refresh the page to test your changes.

If you do need to stop the app:, select the terminal window with your cursor and press CTRL + C. When the app is stopped, it must be rerun with step 3 to see the page again.

# Introduction

As part of this lab, you will be guided through a series of four exercises, to be completed in order. Once you complete an exercise, the next one will unlock on the website. Every time you open an exercise page, you will have a test area in the center, a *Run tests* button on the bottom right, and a *Back* button on the top left.

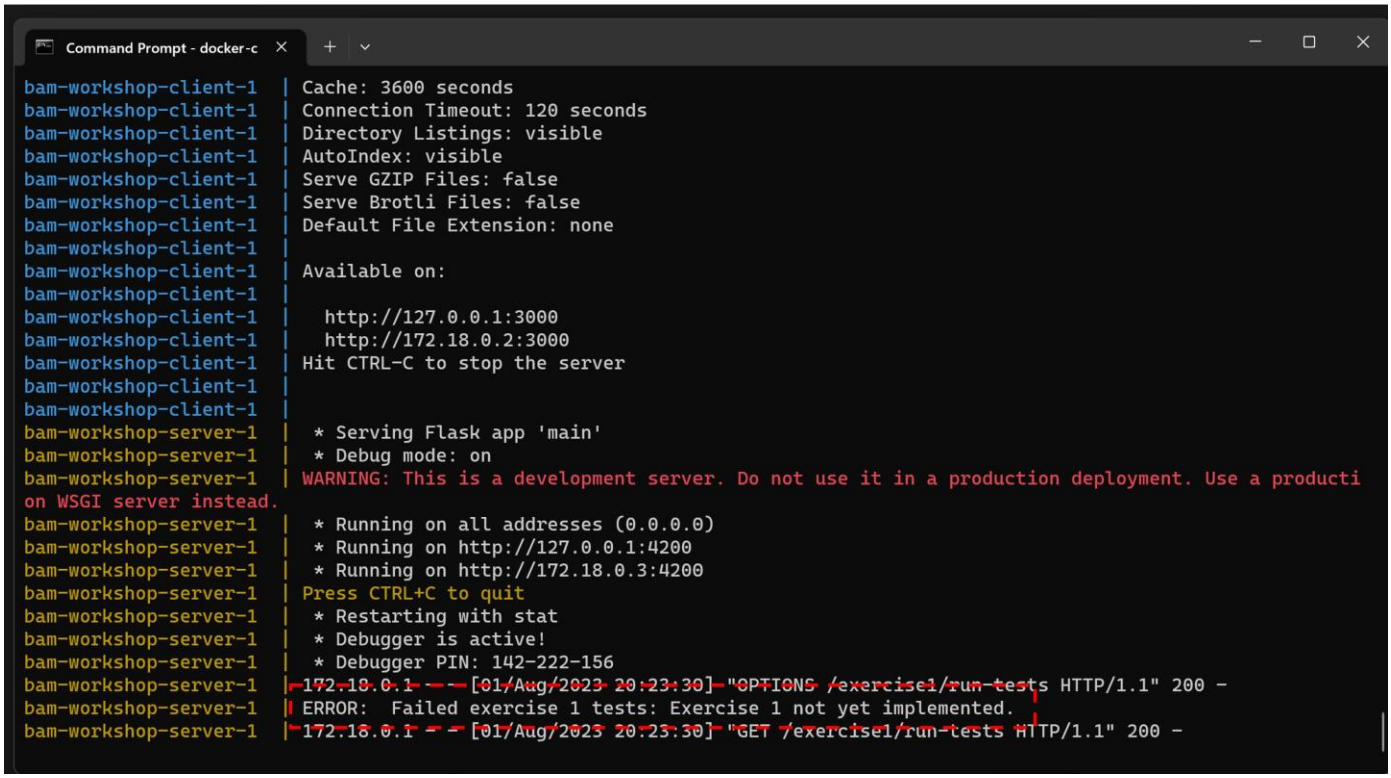

**Back button:** Takes you back to the home page
**Test area:** Test your Python code by interacting with the test area.
**Run tests button:** Ready to verify your code? Click run tests. It will light up green if your implementation is correct, and the exercise is completed. Otherwise you can retry as many times as you'd like.

Once you successfully complete an exercise, feel free to go back to the home page and start the next one whenever you're ready.

# Debugging

When you run your code in the *Test area* or using the *Run tests* button and there is an error, you will see a message saying to check the console. In programming, the console can serve as an information area for programmers. When we write code, we sometimes make mistakes, or something unexpected happens. The console helps by showing messages about those problems. It makes finding and fixing errors and bugs a lot easier!

In this lab, you can check for error messages in the “terminal” or “command prompt” that you ran your start command in. Here is an example of an error message in the console (highlighted in the red dotted line)




Whenever you run tests and there is any sort of error, the first place you should look is in the console.

# Background Information

When calling GPT using code, there are 3 types of messages involved in a conversation with GPT:

1. **User messages**

This is the type of message that humans send to GPT. This could be asking a question, asking for information, or anything else sent to GPT by a person. A user message will trigger a response by GPT.

2. **Assistant messages**

This is the type of message that GPT sends back to humans. This could be an answer to a question, information, or anything else GPT sends back to a person.

3. **System messages**

Most non-developers don't know this type of message, but it can play a crucial role in setting the stage for a conversation. A system message is sent to GPT by a human, like a user message. However, unlike a user message, it will not trigger a response by GPT. Instead, system messages are often used to tell GPT what it's role is in a conversation before the conversation begins. It can also give GPT context or instructions for how to respond to user messages.

**Example**

*System message (Person):* You are a tow truck, you start every message with "Vroom!".
*User message (Person):* What is the capital of France?
*Assistant message (GPT):* Vroom! The capital of France is Paris.

# Exercise 1

**Location: /src/_EDIT_THESE_FILES/exercise1.py**

In this exercise, you will be completing a function that takes in a message, sends the message to GPT-3.5 using Python, and returns the response. Sending a message using the frontend chat box will call the Python function in this file, passing the message as a parameter. When the function is completed correctly, a response will show up in the chat box.

To complete this exercise, you will need to read official Microsoft documentation for how to set up a connection with Azure OpenAI and send a message.

https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/chatgpt?pivots=programming-language-chat-completions#working-with-the-gpt-35-turbo-and-gpt-4-models

The documentation above shows how to send an entire conversation to GPT inside the *messages* array. Take a close look at how each message is being passed:

*{ "role": "…", "content": "…" }*

Each message is passed as a Python dictionary, with the *role* being the type of message (*user, assistant,* or *system*) and the *content* being the message itself. However, keep in mind that only one message is being sent for this exercise. Can you guess which type of message (*user*, *assistant*, or *system*) this is?

Note: the response from the API will be a Python dictionary, not a string. You must access the properties of this dictionary using the square bracket ('[]') syntax to get the response: *response['choices'][0]['message']['content']*

**Instructions:**

1. Set OpenAI variables as shown in the documentation
2. Complete the *sendMessage* function

# Exercise 2

**Location: /src/_EDIT_THESE_FILES/exercise2.py**

In this exercise, you will be completing a class that performs a similar function to exercise 1 but maintains a conversation history. Sending a message using the frontend chat box will call the *sendMessage* function of this class for each message sent. When the function is completed correctly, a full conversation with history will be possible on the frontend.

As you've likely noticed in exercise 1, the *messages* property in the *openai.ChatCompletion.create()* function can take in an array of messages, representing an entire conversation. In the exercise 2 class, you will notice that there also exists a *messages* array as a class property. Every time the *sendMessage(…)* function is called, the message and response should be stored in the messages array. On every call, the entire array (conversation history) will be sent to GPT. That way, GPT will have access to every message and response that has been sent, allowing it to carry out full conversations.

To complete this exercise, the Microsoft documentation from exercise 1 might come in handy.

**Instructions:**

1. Complete the *addUserMessage* function
2. Complete the *addAIMessage* function
3. Complete the *sendMessage* function

# Exercise 3

**Location: /src/_EDIT_THESE_FILES/exercise3.py**

Sending a message using the frontend chat box will use the Python class in this file, calling the *sendMessage* function for each message sent. This is similar to Exercise 2 as it supports a full conversation with history. However, in addition to the chat box, there is a prompt area that takes in a system prompt for GPT. When the function is completed correctly, GPT will respond to messages following the prompt in the prompt area.

In this exercise, you will be completing a class that performs a similar function to exercise 2 but allows for adding system messages. If you recall from the *Background Information* section, a system message instructs GPT on how to respond to *user* messages.

You can reuse the code from Exercise 2 for most of this exercise.

**Instructions:**

1. Complete the *addUserMessage* function (same as exercise 2)
2. Complete the *addAIMessage* function (same as exercise 2)
3. Complete the *sendMessage* function (same as exercise 2)
4. Complete the *sendSystemMessage* function
5. Complete the *getChatMessages* function

# Exercise 4

For this exercise, you are going to be creating a *system* message for GPT to help Donny (the bunny) get through the maze. The message will contain instructions to decode maze codes into readable instructions for Donny to read and follow.

Here is an example of a maze code

*\instr -L 4 -R 1*

Can you guess what it's saying?

The decoded instructions should be in the following format:

*GO <DIRECTION> <NUMBER> TIMES*
*…*

For example:

*GO UP 4 TIMES*
*GO RIGHT 2 TIMES*

Keep in mind the line breaks between the statements, those are important. The prompt should instruct GPT on how to read a maze code and output readable maze instructions from the code. Good luck!