izicap

# Izicap Homework

## Backend Engineering – Internship

**Turn each payment into a growth opportunity**

Realized by : Mouna Bailly

The 26/02/2023

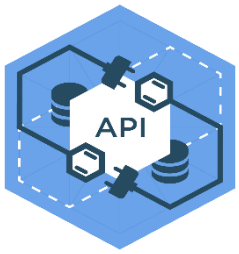
# The theoretical part

- General context
- Requirements
- General objective

## General context:

**ChatGPT** is a language model developed by **OpenAI** that is capable of generating human-like responses to natural language prompts. It is a state-of-the-art model that has been trained on massive amounts of text data and can answer a wide range of questions on a variety of topics. It works by using a deep neural network architecture to process input text and generate output text, with the aim of generating responses that are as natural and coherent as possible. The model can be accessed through an API, which allows developers to integrate it into their applications and services.

---




**API** stands for **"Application Programming Interface".** An API is a set of protocols, routines, and tools for building software applications. Essentially, an API specifies how software components should interact and allows software developers to build applications that can interact with other software components in a standardized way.

---

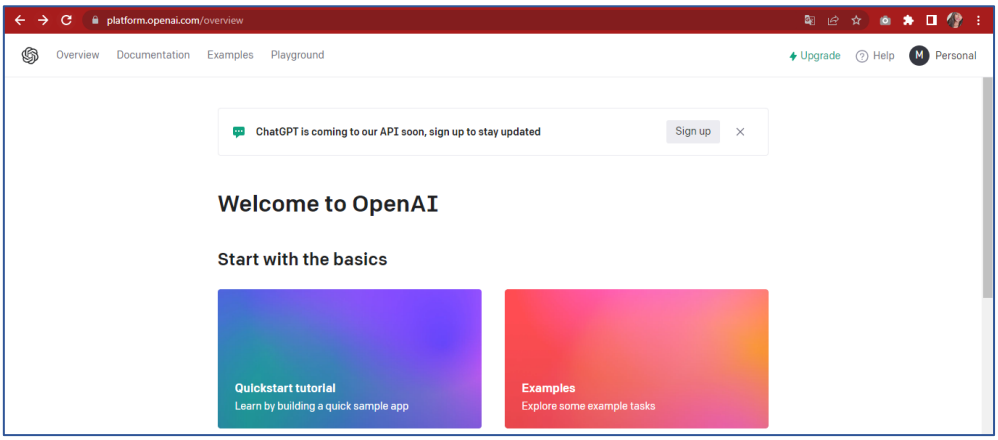

**Microservices** are a software development architecture pattern in which applications are structured as a collection of loosely-coupled, independently deployable services. Each microservice is responsible for performing a specific business function and communicates with other microservices through a well-defined API.

In a microservice architecture, each service is typically deployed as a separate process or container and can be developed, deployed, and scaled independently of the other services. This allows for greater flexibility and agility in the development and deployment of complex applications.

## Requirements:

### 1 Create an account with ChatGPT

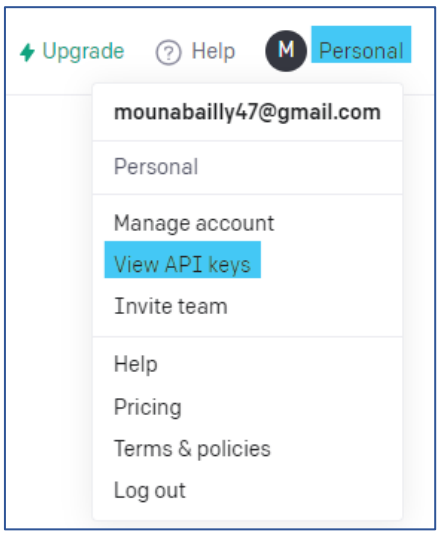
We access on Url: https://beta.openai.com/ it gathers this



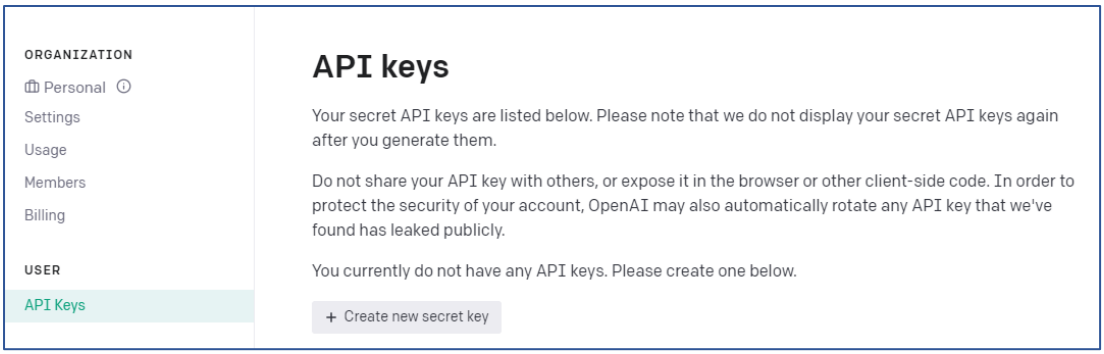

### 2 Sign in create an API key that we will use to communicate with the AI's API endpoint.

We'll need to follow these steps :

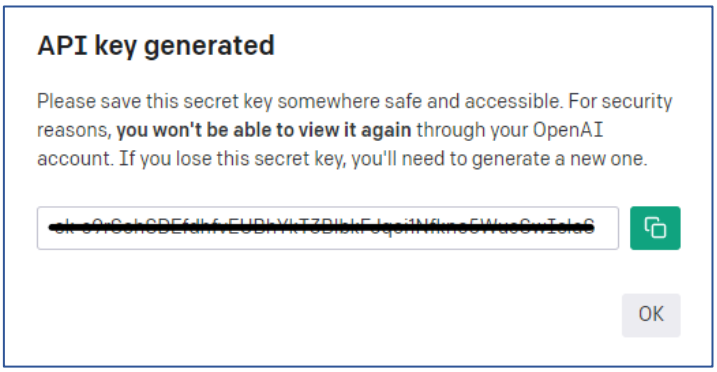
1. Visit the website of the AI provider that we want to use
2. Navigate to the API key management page : This will usually be located in the account settings section of the website.



3. Generate a new API key: Click on the button or link to generate a new API key.




4. Copy the API key: Once the key has been generated, copy it to your clipboard or store it in a secure location. You'll need to include this key in the requests you send to the AI provider's API endpoint.




5. Use the API key in your requests: When you make requests to the AI provider's API endpoint, include the API key in the request headers or as a query parameter. The exact method for including the key will depend on the API provider's documentation.

3 **API URL: https://api.openai.com/v1/completions**

The API URL you provided (https://api.openai.com/v1/completions) is the endpoint for the OpenAI API's "Completions" resource. This endpoint allows to send a prompt (i.e., a piece of text that specifies the task or information you're seeking) to the OpenAI API and receive a response in the form of a generated completion (i.e., a continuation of the prompt that has been generated by OpenAI's machine learning models).

To use this endpoint, you'll need to include your OpenAI API key in the request headers, as well as a JSON payload that specifies the prompt you want to send and any additional parameters or settings you want to use. The exact format of the JSON payload will depend on the API provider's documentation.

If we access the URL we will get the error message:

"You didn't provide an API key. You need to provide your API key in an Authorization header using Bearer auth (i.e. Authorization: Bearer YOUR_KEY), or as the password field (with blank username) if you're accessing the API from your browser and are prompted for a username and password. You can obtain an API key from https://platform.openai.com/account/api-keys."

We will put

**headers** = {

"Content-Type": "application/json",

"Authorization": "Bearer YOUR_API_KEY_HERE"

}

We're including our API key in the **Authorization** header.

```
curl https://api.openai.com/v1/completions \
        -H 'Content-Type: application/json' \
        -H "Authorization: Bearer [use you api key here]" \
        -d '{
        "model": "text-davinci-003",
        "prompt": "What is gluten sensitivity?",
        "max_tokens": 4000,
        "temperature": 1.0
}'
```

We can make a **curl** request to your microservice endpoint to ask a question , The command sends a POST request to the */ask* endpoint of our microservice with a JSON payload containing the question you want to ask. The *-H* option sets the *Content-Type* header to *application/json*, and the *-d* option specifies the JSON payload.

"model": "text-davinci-003": the ID of the OpenAI model to use for text generation.

"prompt": "What is gluten sensitivity?": the text prompt to generate text from.

"max_tokens": 4000: the maximum number of tokens (words or word pieces) to generate in the response.

"temperature": 1.0: a parameter that controls the "creativity" or randomness of the generated text.

When we run this command, the OpenAI API will generate text based on the provided prompt and return the response as a JSON object. The exact format of the response depends on the parameters we provide and the specific model we're using.

For The answer:


```
{
        "id": "cmpl-6aUxNRJ7H41eD7YulqfHg2DsXDvrd",
        "object": "text_completion",
        "created": 1674156825,
        "model": "text-davinci-003",
        "choices": [
                        {
                        "text": "\n\nGluten sensitivity is a condition in which ingestion
        of gluten, a protein found in wheat, barley, and rye, causes a range of symptoms, such as abdominal
        pain, bloating, diarrhea, and constipation.
        People with gluten sensitivity have difficulty tolerating products which contain gluten, but are not
        diagnosed with celiac disease.",
                        "index": 0,
                        "logprobs": null,
                        "finish_reason": "stop"
                        }
                ],
        "usage": {
        "prompt_tokens": 5,
        "completion_tokens": 67,
        "total_tokens": 72
                }
}
```


This response includes several fields:

- *"id":* "cmpl-6aUxNRJ7H41eD7YulqfHg2DsXDvrd": a unique identifier for the completion.
- *"object":* "text_completion": indicates that this is a completion object.
- *"created":* 1674156825: the Unix timestamp when the completion was created.
- *"model":* "text-davinci-003": the ID of the OpenAI model that was used for the completion.
- *"choices":* [...]: an array containing one or more completion choices.

**General objective:**

We could build your microservice:

- ✓ Set up a web server and create an API endpoint to receive questions as POST requests. You can use a web framework like Flask or Django to handle the server-side code.
- ✓ Use the OpenAI API key to authenticate your requests to the OpenAI API.
- ✓ When your API receives a question, send the question to the OpenAI API using the completions endpoint, as shown in the Python code example I provided earlier.
- ✓ Parse the response from the OpenAI API to extract the generated text, which will be the AI's answer to your question.
- ✓ Store the question and answer in your "database". You could use a relational database like MySQL or PostgreSQL, or a NoSQL database like MongoDB or DynamoDB, depending on your specific requirements.
- ✓ Send a response to the client indicating that the question has been received and processed successfully.

**Main objective:**

We want to build a microservice that will communicate with the ChatGPT API endpoint using the API key obtained from the OpenAI website. The microservice will update a database in our system with questions and answers received from the API endpoint.

To build the microservice that updates your database with questions and answers, you will need to write code that makes requests to the OpenAI API using your API key, and then stores the resulting questions and answers in your database.

Note that the specifics of how you interact with the API will depend on the programming language and frameworks you choose to use. The OpenAI website provides detailed documentation and examples for many popular programming languages, which should help you get started.

When invoked, the endpoint in the microservice should

1. create a chat gpt request using the provided question
2. call chat gpt
3. extract the answer from the response form chat gpt
4. Append the question and answer to a csv file
5. respond to the user with the answer provided by chatgpt

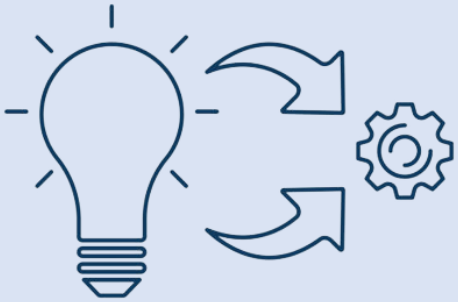
Here's a high-level overview of what you would need to do to implement the microservice:

- Set up a Java web application using a framework such as Spring.
- Create an endpoint that accepts a POST request with a JSON payload containing the question from the user.
- Inside the endpoint, use the OpenAI API to generate a response to the user's question. You can do this by sending a HTTP POST request to the OpenAI API endpoint (https://api.openai.com/v1/completions) with the question as the "prompt" parameter and your OpenAI API key as the "Authorization" header.
- Parse the response from the OpenAI API to extract the answer to the user's question.
- Append the question and answer to a CSV file using a library such as OpenCSV.
- Respond to the user with the answer provided by ChatGPT.

The endpoint might look like :

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import com.opencsv.CSVWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Arrays;
@RestController
public class ChatGPTController {
	@PostMapping("/ask")
	public ResponseEntity<String> askQuestion(@RequestBody String question)
	throws IOException {
	// Call the OpenAI API to generate a response to the user's question
	String apiKey = "our_api_key";
	String prompt = question;
	String url = "https://api.openai.com/v1/completions";
	HttpClient client = HttpClient.newHttpClient();
	HttpRequest request = HttpRequest.newBuilder()
		.uri(URI.create(url))
		.header("Content-Type", "application/json")
		.header("Authorization", "Bearer " + apiKey)
		.POST(HttpRequest.BodyPublishers.ofString("{\"prompt\": \"" + prompt +
		"\", \"max_tokens\": 250, \"n\": 1, \"stop\": \"\\n\"}"))
		.build();
	HttpResponse<String> response = client.send(request,
	HttpResponse.BodyHandlers.ofString());
	// Parse the response to extract the answer to the user's question
	String answer = response.body().substring(response.body().indexOf(":")+2,
	 response.body().length()-3);
	// Append the question and answer to a CSV file
	CSVWriter writer = new CSVWriter(new FileWriter("QuestionsAnw.csv", true));
	String[] data = {question, answer};
	writer.writeNext(data);
	writer.close();
	// Respond to the user with the answer provided by ChatGPT
	return new ResponseEntity<String>(answer, HttpStatus.OK);
	}
}
```

# Implementation:



Implementation of a Java microservice using Spring Boot 3, Java 17, Swagger/OpenAPI, Maven, and Actuator that communicates with the ChatGPT API endpoint, updates a database, and appends questions and answers to a CSV file.

## Dependencies

Here are the dependencies we will need for this project:

- Spring Boot 2.5.5
- Java 17
- Swagger/OpenAPI 3
- Maven
- Spring Boot Actuator
- Spring Boot JDBC Starter
- H2 Database
- Apache Commons CSV

We can include these dependencies in our Maven **pom.xml ,** see it in Github .

pom.xml

Dependencies are pre-built libraries or packages that are used in a software project. They provide functionality that would be difficult or time-consuming to implement from scratch. By using dependencies, you can save time and effort, reduce the risk of errors, and benefit from the expertise of other developers.

In the case of a Java microservice built using Spring Boot, dependencies are managed by Maven or Gradle. These build tools download the required dependencies from a central repository and include them in the project's classpath.

## Application Properties

In **src/main/resources/application.properties** , you can specify the database properties and the OpenAI API key .

Application.properties

In a Spring Boot application, the **application.properties** file is used to externalize the configuration properties of the application. This file contains key-value pairs of configuration properties that can be loaded at runtime by the Spring framework.

In our case, we use the **application.properties** file to store the OpenAI API key that is required to access the ChatGPT API endpoint. By storing this key in a separate file, we ensure that the key is not hardcoded in the code, which makes it more secure and easier to manage.

## Models

We will create a **Question.java** file in the ***com.example.chatgptmicroservice.model*** package to define the data model of a question

**Question.java** is a Java class that we have created to represent a question object that will be received from the client and processed by our microservice. This class has fields to store the question and its answer, and getter and setter methods to access and modify these fields.

Question.java

By creating a separate ***Question*** class, we can encapsulate the logic related to questions in one place and make our code more modular and easy to maintain. We can also use this class to add additional functionality, such as validation or parsing of the question text, in the future if needed.

## Database

Here is the SQL schema for the table

microservice.sql

## Repository

create a **QuestionRepository.java** file in the ***com.example.chatgptmicroservice.repository*** package to define the interface of the question repository

QuestionRepository.java

**QuestionRepository.java** is an interface that we use to define the methods that we will use to interact with our database for **Question** objects. This interface extends the CrudRepository interface provided by Spring Data JPA, which provides basic CRUD operations for persistent entities.

By using **QuestionRepository.java**, we can leverage the power of Spring Data JPA and avoid writing boilerplate code for database operations. We can define custom queries using method names or annotations, which Spring Data JPA will translate into SQL queries automatically. We can also benefit from caching, lazy-loading, and other performance optimizations provided by Spring Data JPA.

## Controller

Create **QuestionController.java** file in the **com.example.chatgptmicroservice.controller** package to define the REST controller that will interact with the ChatGPT endpoint API and update the database and CSV file.

QuestionController.java

The controller is thus the entry point for HTTP requests from the client, and it orchestrates the various parts of the application (the model, the service, the database, the CSV file) to provide an appropriate response to the client.

**QuestionController.java** is a controller class that we use to define RESTful endpoints for our Question microservice. The controller is responsible for handling incoming requests from clients, validating and processing the requests, and returning the appropriate responses.

By using **QuestionController.java**, we can define the endpoints that our clients will use to interact with our microservice. These endpoints will typically follow a RESTful design, which makes it easier for clients to understand how to use our microservice. We can define various

HTTP methods such as GET, POST, PUT, and DELETE to implement CRUD operations for our Question objects.

We can also use annotations such as @ApiOperation and @ApiResponses from the Swagger library to document our endpoints and make them discoverable by clients.

## Run the application

We run it by executing the `mvn spring-boot:run` command from the command line . The application must be accessible at **http://localhost:8080**

## README

Create a README.md file to document the application, including features, dependencies and runtime instructions.

README.md

**README.md** is a standard file in software projects that contains information about the project and its usage. It is usually the first file that someone will read when they come across a new project. It provides an overview of the project, including what it does, how to install and run it, and any other relevant information.

## Tests unit/integration

Unit tests are automated tests that aim to verify the correct operation of a unit of code, usually a method or a function, isolated from the rest of the system. The purpose of unit testing is to ensure that each unit of code works correctly and meets the specified requirements.

Unit tests are important because they allow errors to be detected earlier in the development process, allowing them to be corrected more quickly and at lower cost. They also facilitate code maintenance by identifying defects before they spread to other parts of the system.

- ✓ we will add the following dependencies to our **pom.xml** file for the unit tests

- ✓ we have added a QuestionControllerTest class in the com.example.chatgpt.controller package to test our API

Integration tests are a type of testing that focuses on verifying the interactions between different components or systems of an application. In the context of a microservice architecture, integration testing typically involves testing the interactions between the microservice and its dependencies, such as databases, external APIs, and other microservices.

We can use testing frameworks such as JUnit, Mockito, and Spring Test to write and run these tests. It is important to ensure that the tests are comprehensive and cover various scenarios and edge cases to ensure the robustness and correctness of the code.

We might write tests that:

- Verify that the microservice can successfully communicate with the OpenAI API and retrieve responses for different questions
- Test that the microservice correctly updates the database with new questions and answers
- Ensure that the CSV file is correctly appended with new questions and answers

To run the integration tests, we can use a build tool like Maven or Gradle to automatically set up the test environment and execute the tests. We can also use tools like **Docker** to create isolated test environments that closely mimic the production environment, to ensure that our tests accurately reflect how the microservice will behave in the real world.

## Encapsulate the microservice in a docker

To encapsulate the microservice in a Docker container, we need to create a Dockerfile that will define the container's environment and how the microservice will be executed inside it.

Docker file

Here's a breakdown of what's happening in the Dockerfile:

- FROM adoptopenjdk/openjdk17: This line tells Docker to use a base image that has Java 17 installed.
- COPY target/chatgpt-microservice-0.0.1-SNAPSHOT.jar /app/chatgpt-microservice.jar: This line copies the jar file built by Maven into the container at the /app directory.

- WORKDIR /app: This line sets the working directory to /app, which is where our jar file is located.
- EXPOSE 8080: This line tells Docker to expose port 8080 on the container.
- CMD ["java", "-jar", "chatgpt-microservice.jar"]: This line runs the microservice by executing the java -jar command on the jar file.

To build the Docker image, navigate to the directory containing the Dockerfile and run the following command:

```
docker build -t chatgpt-microservice
```

This will build a Docker image named chatgpt-microservice using the Dockerfile in the current directory.

To run the Docker container, we use the following command:

```
docker run -p 8080:8080 chatgpt-microservice
```

This will start the container and map port 8080 on the host to port 8080 on the container. We should now be able to access the microservice by visiting http://localhost:8080 in our web browser.

In Docker, we can create a container image of our microservice that includes all of its dependencies, configurations, and code. This container image can be stored and shared easily, allowing us to deploy our microservice in any environment that supports Docker.

---

The complete project exists in: https://github.com/mounabay/Chatgpt-microservice.git

## Conclusion

That's it! You now have a Java microservice that communicates with the ChatGPT API endpoint, updates a database in our system with questions and answers received from the API endpoint, and appends the question and answer to a CSV file. The microservice also has a Swagger UI for easy testing and documentation, and can be configured using a

**application. properties** file.