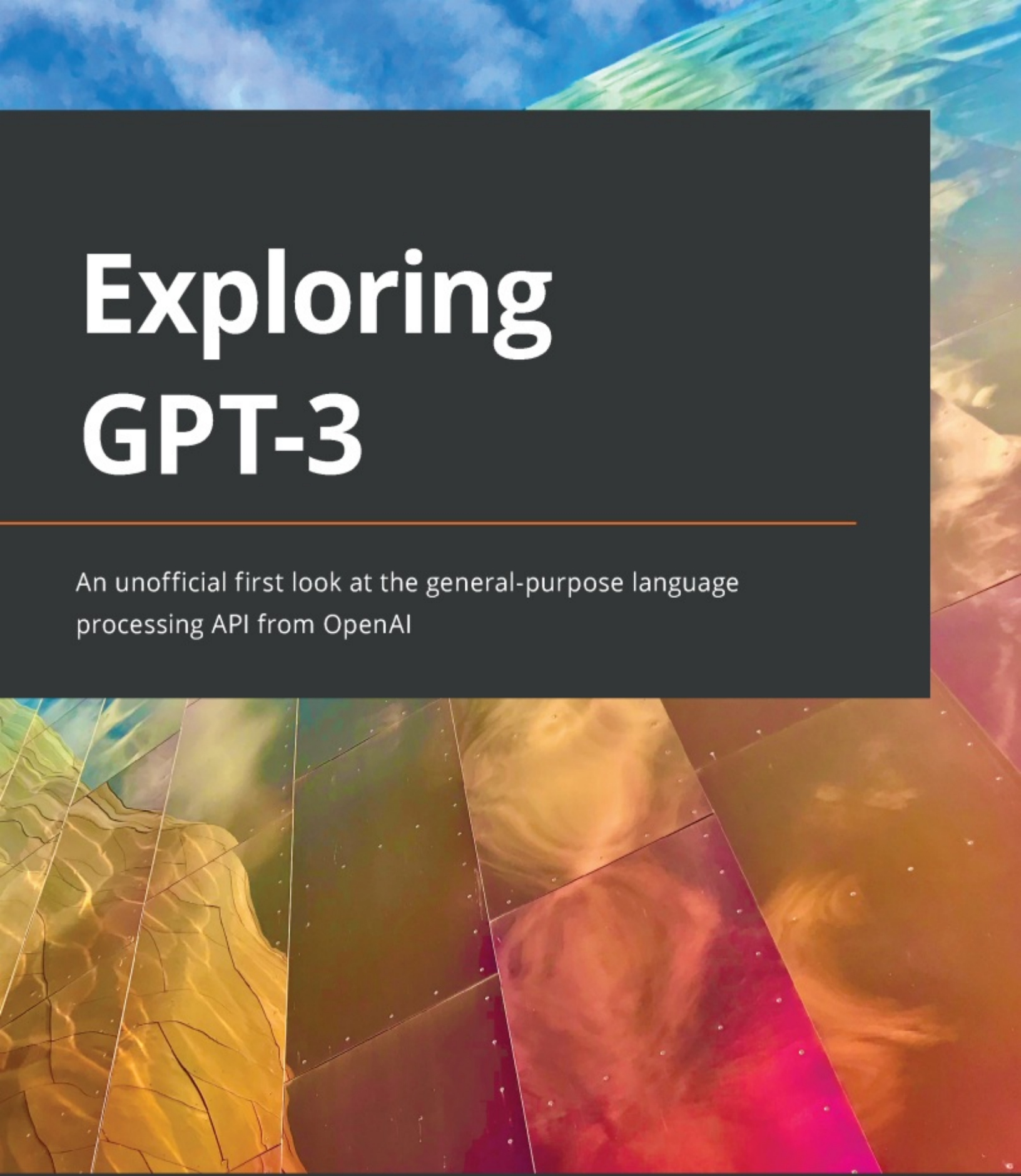


# Exploring GPT-3

An unofficial first look at the general-purpose language processing API from OpenAI

Steve Tingiris

Foreword by Bret Kinsella (Founder and CEO of Voicebot.ai)

BIRMINGHAM—MUMBAI

# Exploring GPT-3



**Group Product Manager**: Kunal Parikh

**Publishing Product Manager**: Sunith Shetty

**Senior Editor**: David Sugarman

**Content Development Editor**: Nathanya Dias

**Technical Editor**: Devanshi Ayare

**Copy Editor**: Safis Editing

**Project Coordinator**: Aparna Ravikumar Nair

**Proofreader**: Safis Editing

**Indexer**: Rekha Nair

**Production Designer**: Alishon Mendonca

First published: July 2021

Production reference: 1100621

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80056-319-3

www.packt.com

*To my wife, Brigid, for putting up with my constant dabbling for all these years. To my son, Alex, for motivating me to push a little harder. There is no way I would have finished this book without you guys. Thank you!*

*– Steve*

# Foreword

What's all the fuss about? Maybe it's the demos. But that would be missing the point. GPT-3 burst into public consciousness in July 2020 not too long after OpenAI first introduced the technical breakthrough in a soberly presented blog post. Introduced in the middle of a global pandemic, rising social unrest, and a US presidential campaign, the message could have been lost entirely. The demos started trickling out in earnest a month later. They ensured that GPT-3 wouldn't be overlooked.

*Technology Review* claimed the GPT-3 was *shockingly good*. Many long-time AI researchers expressed both enthusiasm and surprise at its capabilities. But it was the demos that really captured everyone's attention. With a few lines of sample content and a request, the technology was generating poetry, website programming code, analogies, and answers to math questions, to name just a few examples. No one had ever seen a computer create and respond creatively to such a wide range of queries.

GPT-3 seemed to possess magical abilities. In many ways, GPT-3 is very simple. It predicts what words are the most likely to follow in a sequence. However, it is also the finest current example of the potential of **Generative Adversarial Networks** (**GANs**). And, it has shown that transformers can be applied to language models at an extremely large scale. GPT-3 provides an intriguing new technical capability while simultaneously resetting expectations about what is possible. As just one example, maybe chatbots don't have to choose between a set number of deterministic responses.

**Natural Language Processing** (**NLP**) has improved quickly in recent years. That has led to the more accurate recognition and understanding of speech. At the same time, synthetic speech engines have improved immensely and sound more humanlike each year. What has changed very little is how the systems respond to requests. They are all picking from a predetermined set of responses. GPT-3 offers a capability that enables developers to rethink that approach. But that requires developers to understand the new technology and how to use it.

Steve Tingiris is the first to take that task on with this book. With clear and precise presentation, Tingiris expertly walks new users through the journey from idea through production of a GPT-3 application. He lays out the principles and steps so developers can turn their ideas into (a new) reality. I look forward to seeing what you build.

*Bret Kinsella*

*Founder and CEO of Voicebot.ai*

# Contributors

# About the author

**Steve Tingiris** is the founder and managing director of Dabble Lab, a technology research and services company that helps businesses accelerate learning and adoption of natural language processing, conversational AI, and other emerging technologies. He has been designing and building automation solutions for over 20 years and has consulted on conversational AI projects for companies including Amazon, Google, and Twilio. He also publishes technical tutorials on Dabble Lab's YouTube channel— one of the most popular educational resources for conversational AI developers—and manages several open-source projects, including the Autopilot CLI, Twilio's recommended tool for building Autopilot bots. To connect with Steve, you can find him on **GitHub @tingiris**, via email to **steve@dabblelab.com**, or on Twitter **@tingiris**.

# Acknowledgments

This book is the result of contributions from friends, colleagues, and many members of the OpenAI community. There are far too many people to mention everyone by name but for those who contributed directly or reviewed early drafts, I want to extend a special thank you.

First, thank you to the entire Packt team. If Sunith had not reached out with the idea of creating a book on GPT-3, this project would never have started. If David, Gebin, Nathanya, Aishwarya, Roshan, and Devanshi weren't involved – the book might never have gotten finished. Thanks to each of you and everyone else on the Packt team who made this possible.

Thanks to Russell, Bakz, Ryan, and Bram for agreeing to be technical reviewers. You guys were so helpful to me in the OpenAI Slack channel when I first got started with GPT-3, I feel so fortunate to have had the opportunity to collaborate together on this book. Again, thank you!

Next, thank you to my *work family* at Dabble Lab. Kirk Owen for providing early feedback that helped refine the direction, and Mohamad Khalid, Manuel Fernandez, Mark Hovsepyan, Sohini Pattanayak, Shubham Prakash, and Daniela Ramirez, for picking up the slack while I spent much more time than I'd anticipated working on the book.

Finally, I'd like to thank the OpenAI team and the OpenAI community for all of the support and feedback throughout the project, including: Ashley Pilipiszyn, Mark Clintsman, Minal Chhatbar, Rene Diaz, Dan Shaw, Chris Fong, Dariusz Gross, Cristi Vlad, Jonathon Sauceda, Marc-Andre Schenk, Steven Kuo, Narendran Thillaisthanam, Matthew Benites, Manav Goel, Shubham Amraotkar, Mystici Mentis, Fred Zimmerman, Dmitry Kearo, CL Kim, Sudip Lingthep, Joakim Flink, Shubham Saboo, Pedro Ribeiro, Richard Klein, Steve Hoyt, Nicolas Garrel, Sebastian Derewicki, Vikram Pandya, Geoff Davis, Nelson Pereira, Heng Gu, Joey Bertschler, Surendra Reddy, James Morgan, Jon Oakes, Jeetendra K Sharma, Jim Taylor, Rebecca Johnson, Travis Barton, Herber Scrap, Pablo del Ser, Devin Bean, Nik K, Jason Boog, Mohak Agarwal, Sebastian Elliott, and Bjarne Carstensen.

## About the reviewers

**Russell Foltz-Smith** has 20+ years of experience in tech as a developer, business development leader, executive, and researcher. He maintains a focus on search engines, scientific computing, and media platforms. Russ advises tech start-ups, mentors entrepreneurs, and is CTO of Maslo.ai, an empathetic computing platform. Russ is a visual artist and educator. He co-founded a k-12 school with his wife in 2012 in Venice, CA, where they live with their daughters and endless stacks of books.

**Bakz Awan** is an IT consultant and YouTube host of the channel *Bakz T. Future.* on YouTube. Bakz shares new concepts and ideas possible through GPT-3, while also providing tips and advice to beginners.

Table of Contents

Preface

# Section 1: Understanding GPT-3 and the OpenAI API

# *Chapter 1*: Introducing GPT-3 and the OpenAI API


**Technical requirements**

**Introduction to GPT-3**

**Simplifying NLP**

**What exactly is GPT-3?**

**Democratizing NLP**

**Understanding prompts, completions, and tokens**

**Prompts**

**Completions**

**Tokens**

**Introducing Davinci, Babbage, Curie, and Ada**

**Davinci**

**Curie**

**Babbage**

**Ada**

**Content filtering model**

**Instruct models**

**A snapshot in time**

**Understanding GPT-3 risks**

**Inappropriate or offensive results**

**Potential for malicious use**

**Summary**

# *Chapter 2*: GPT-3 Applications and Use Cases

**Technical requirements**

**Understanding general GPT-3 use cases**

**Introducing the Playground**

**Getting started with the Playground**

**Handling text generation and classification tasks**

**Text generation**

**Text classification**

**Understanding semantic search**

**The Semantic Search tool**

**Summary**

## Section 2: Getting Started with GPT-3

## *Chapter 3*: Working with the OpenAI Playground


**Technical requirements**

**Exploring the OpenAI developer console**

**Developer documentation**

**Developer resources**

**Accounts and organizations**

**Pricing and billing**

**Usage reporting**

**Member management**

**Diving deeper into the Playground**

**Choosing the right engine**

**Response length**

**Temperature and Top P**

**Frequency and presence penalty**

**Best of**

**Stop sequence**

**Inject Start Text and Inject Restart Text**

**Show Probabilities**

**Working with presets**

**Grammatical Standard English**

**Text to command**

**Parse unstructured data**

**Summary**

## *Chapter 4*: Working with the OpenAI API


**Technical requirements**

**Understanding APIs**

**Getting familiar with HTTP**

**Uniform resource identifiers**

**HTTP methods**

**The HTTP body**

**HTTP headers**

**HTTP response status codes**

**Reviewing the OpenAI API endpoints**

**List Engines**

**Retrieve Engine**

**Create Completions**

**Semantic Search**

**Introducing CURL and Postman**

**Understanding API authentication**

**Keeping API keys private**

**Making an authenticated request to the OpenAI API**

**Working with multiple organizations**

**Introducing JSON**

**Using the Completions endpoint**

**Using the Semantic Search endpoint**

**Summary**

## *Chapter 5*: Calling the OpenAI API in Code


**Technical requirements**

**Choosing your programming language**

**Introducing replit**

**Creating a repl**

**Setting your OpenAI API key as an environment variable**

**Understanding and creating the .replit file**

**Using the OpenAI API with Node.js/JavaScript**

**Calling the engines endpoint**

**Calling the Completions endpoint**

**Calling the search endpoint**

**Using the OpenAI API in Python**

**Calling the completions endpoint**

**Calling the search endpoint**

**Using other programming languages**

**Summary**

## Section 3: Using the OpenAI API

## *Chapter 6*: Content Filtering

**Technical requirements**

**Preventing inappropriate and offensive results**

**Understanding content filtering**

**Testing the content filtering process**

**Filtering content with JavaScript**

**Flagging unsafe words with Node.js/JavaScript**

**Filtering content with Python**

**Flagging unsafe words with Python**

**Summary**

## *Chapter 7*: Generating and Transforming Text


**Technical requirements**

**Using the examples**

**Generating content and lists**

**Dumb joke generator**

**Mars facts (in most cases)**

**Webinar description generator**

**Book suggestions**

**Children's book generator**

**Translating and transforming text**

**Acronym translator**

**English to Spanish**

**JavaScript to Python**

**Fifth-grade summary**

**Grammar correction**

**Extracting text**

**Extracting keywords**

**HTML parsing**

**Extracting a postal address**

**Extracting an email address**

**Creating chatbots**

**A simple chatbot**

**Summary**

# *Chapter 8*: Classifying and Categorizing Text

**Technical requirements**

**Understanding text classification**

**Using the completions endpoint for text classification**

**Content filtering is a text classification task**

**Introducing the classifications endpoint**

**Uploading files**

**Implementing sentiment analysis**

**Assigning an ESRB rating to text**

**Classifying text by language**

**Classifying text from keywords**

**Summary**

# *Chapter 9*: Building a GPT-3-Powered Question-Answering App


**Technical requirements**

**Introducing GPT Answers**

**GPT Answers technical overview**

**Hosting the app**

**Introducing the Answers endpoint**

**Setting up and testing Express**

**Creating the API endpoint for GPT Answers**

**Creating the API endpoint**

**Testing our API with Postman**

**Creating the GPT Answers user interface**

**Integrating the Answers endpoint**

**Generating relevant and factual answers**

**Using files with the Answers endpoint**

**Summary**

## *Chapter 10*: Going Live with OpenAI-Powered Apps

**Technical requirements**

**Going live**

**Understanding use case guidelines**

**Addressing potential approval issues**

**Content filtering**

**Input and output lengths**

**Request rate limiting**

**Completing the pre-launch review request**

**High-level use case questions**

**Security and risk mitigation questions**

**Growth plan questions**

**Wrapping-up questions**

**Summary**

**Why subscribe?**

## Other Books You May Enjoy

# Preface

What if this book was written by artificial intelligence? Would you read it? I hope so because parts of it were. Yes, GPT-3 was used to create parts of this book. It's a bit meta I know, a book about GPT-3 written by GPT-3. But creating content is one of the many great uses for GPT-3. So why not? Also, for me, content generation was the use case that most piqued my interest. I wondered if GPT-3 could be used in a product I was working on to automate the generation of technical learning material.

You probably also have a specific reason why you're interested in GPT-3. Perhaps it's intellectual curiosity. Or maybe you have an idea that you think GPT-3 can enable. You've likely seen online demos of GPT-3 generating content, writing code, penning poetry, or something else, and you're wondering if GPT-3 could be used for an idea you have. If so, this book was written specifically for you.

My goal for this book is to provide a practical resource to help you get started with GPT-3, as quickly as possible, without any required technical background. That said, as I write this, GPT-3 is still in private beta. So, everyone is learning as they go. But the one thing I've learned for sure is that the possible applications for GPT-3 are vast and there is no way to know all of what's possible, let alone get it into a book. So, I hope this book makes getting started easy, but I also hope it's just the beginning of your journey *Exploring GPT-3*

## Who this book is for

This book was written for anyone with an interest in NLP or learning GPT-3 – with or without a technical background. Developers, product managers, entrepreneurs, and hobbyists who want to learn about NLP, AI, and GPT-3 will find this book useful. Basic computer skills are all you need to get the most out of the book. While experience with a modern programming language is helpful, it's not required. The code examples provided are beginner friendly and easy to follow, even if you're brand new to writing code.

## What this book covers

*Chapter 1*, *Introducing GPT-3 and the OpenAI API*, is a high-level introduction to GPT-3 and the OpenAI API.

*Chapter 2*, *GPT-3 Applications and Use Cases*, is an overview of core GPT-3 use cases: text generation, classification, and semantic search.

*Chapter 3*, *Working with the OpenAI Playground*, is a semi-deep dive into the OpenAI Playground and the developer portal.

*Chapter 4*, *Working with the OpenAI API*, is an introduction to calling the OpenAI API using Postman.

*Chapter 5*, *Calling the OpenAI API in Code*, is an introduction to using the OpenAI API with both Node.js/JavaScript and Python.

*Chapter 6*, *Content Filtering*, explains how to implement content filtering.

*Chapter 7*, *Generating and Transforming Text*, contains code and prompt examples for generating and transforming text.

*Chapter 8*, *Classifying and Categorizing Text*, takes a closer look at text classification and the OpenAI API Classification endpoint.

*Chapter 9*, *Building a GPT-3 Powered Question-Answering App*, explains how to build a functional GPT-3 powered web knowledge base.

*Chapter 10*, *Going Live with OpenAI-Powered Apps*, explains the OpenAI application review and approval process and discusses getting ready for a review.

## To get the most out of this book

All of the code examples in this book were written using a web-based **Integrated Development Environment** (**IDE**) from replit.com. A free replit.com account is sufficient to follow the examples. To use replit.com, all that is required is a modern web browser and a replit.com account. The code has also been tested on macOS using Visual Studio Code, although it should work with any code editor and properly configured operating system. Code examples are provided in both Node.js/JavaScript and Python. For Node.js, version 12.16.1 is used and for Python, version 3.8.2 is used.

| Software/hardware covered in the book | OS requirements |
|---|---|
| Node.js version 12.16.1 | Windows, macOS, and Linux (Any) |
| Python version 3.8.2 | Windows, macOS, and Linux (Any) |

All of the code examples will require an OpenAI API Key and access to the OpenAI API. You can request access to the OpenAI API by visiting https://openai.com/api.

**If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code**.

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781800563193_ColorImages.pdf.

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text**: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "However, suppose you don't want the completion to generate the human side of the conversation and you want to use the label **AI:** rather than **Assistant:**?"

A block of code is set as follows:

```
English: I do not speak Spanish Spanish:
```

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Each subsequent time the **Submit** button is clicked."

### *TIPS OR IMPORTANT NOTES*

*Appear like this.*

## Get in touch

Feedback from our readers is always welcome.

**General feedback**: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at **customercare@packtpub.com**.

**Errata**: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy**: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **copyright@packt.com** with a link to the material.

**If you are interested in becoming an author**: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

# Section 1: Understanding GPT-3 and the OpenAI API

The objective of this section is to provide you with a high-level introduction to GPT-3 and the OpenAI API and to show how easy it is to get started with. The goal is to engage you with fun examples that are quick and simple to implement.

This section comprises the following chapters:

- *Chapter 1, Introducing GPT-3 and the OpenAI API*
- *Chapter 2, GPT-3 Applications and Use Cases*

# *Chapter 1*: Introducing GPT-3 and the OpenAI API

The buzz about **Generative Pre-trained Transformer Version 3** (**GPT-3**) started with a blog post from a leading **Artificial Intelligence** (**AI**) research lab, OpenAI, on June 11, 2020. The post began as follows:

> *We're releasing an API for accessing new AI models developed by OpenAI. Unlike most AI systems which are designed for one use-case, the API today provides a general-purpose "text in, text out" interface, allowing users to try it on virtually any English language task.*

Online demos from early beta testers soon followed—some seemed too good to be true. GPT-3 was writing articles, penning poetry, answering questions, chatting with lifelike responses, translating text from one language to another, summarizing complex documents, and even writing code. The demos were incredibly impressive—things we hadn't seen a general-purpose AI system do before—but equally impressive was that many of the demos were created by people with a limited or no formal background in AI and **Machine Learning** (**ML**). GPT-3 had raised the bar, not just in terms of the technology, but also in terms of AI accessibility.

GPT-3 is a general-purpose language processing AI model that practically anybody can understand and start using in a matter of minutes. You don't need a **Doctor of Philosophy** (**PhD**) in computer science—you don't even need to know how to write code. In fact, everything you'll need to get started is right here in this book. We'll begin in this chapter with the following topics:

- Introduction to GPT-3
- Democratizing NLP
- Understanding prompts, completions, and tokens
- Introducing Davinci, Babbage, Curie, and Ada
- Understanding GPT-3 risks

## Technical requirements

This chapter requires you to have access to the **OpenAI Application Programming Interface** (**API**). You can register for API access by visiting https://openapi.com.

## Introduction to GPT-3

In short, GPT-3 is a language model: a statistical model that calculates the probability distribution over a sequence of words. In other words, GPT-3 is a system for guessing which text comes next when text is given as an input.

Now, before we delve further into what GPT-3 is, let's cover a brief introduction (or refresher) on **Natural Language Processing** (**NLP**).

## Simplifying NLP

NLP is a branch of AI that focuses on the use of natural human language for various computing applications. NLP is a broad category that encompasses many different types of language processing tasks, including sentiment analysis, speech recognition, machine translation, text generation, and text summarization, to name but a few.

In NLP, language models are used to calculate the probability distribution over a sequence of words. Language models are essential because of the extremely complex and nuanced nature of human languages. For example, *pay in full* and *painful* or *tee time* and *teatime* sound alike but have very different meanings. A phrase such as *she's on fire* could be literal or figurative, and words such as *big* and *large* can be used interchangeably in some cases but not in others—for example, using the word *big* to refer to an older sibling wouldn't have the same meaning as using the word *large*. Thus, language models are used to deal with this complexity, but that's easier said than done.

While understanding things such as word meanings and their appropriate usage seems trivial to humans, NLP tasks can be challenging for machines. This is especially true for more complex language processing tasks such as recognizing irony or sarcasm—tasks that even challenge humans at times.

Today, the best technical approach to a given NLP task depends on the task. So, most of the best-performing, **state-of-the-art** (**SOTA**) NLP systems are specialized systems that have been fine-tuned for a single purpose or a narrow range of tasks. Ideally, however, a single system could successfully handle any NLP task. That's the goal of GPT-3: to provide a general-purpose AI system for NLP. So, even though the best-performing NLP systems today tend to be specialized, purpose-built systems, *GPT-3 achieves SOTA performance on a number of common NLP tasks*, showing the potential for a future general-purpose NLP system that could provide SOTA performance for any NLP task.

## What exactly is GPT-3?

Although GPT-3 is a general-purpose NLP system, it really just does one thing: it predicts what comes next based on the text that is provided as input. But it turns out that, with the right architecture and enough data, this *one thing* can handle a stunning array of language processing tasks.

GPT-3 is the third version of the GPT language model from OpenAI. So, although it started to become popular in the summer of 2020, the first version of GPT was announced 2 years earlier, and the following version, GPT-2, was announced in February 2019. But even though GPT-3 is the third version, the general system design and architecture hasn't changed much from GPT-2. There is one big difference, however, and that's the size of the dataset that was used for training.

GPT-3 was trained with a massive dataset comprised of text from the internet, books, and other sources, containing roughly 57 billion words and 175 billion parameters. That's 10 times larger than

GPT-2 and the next-largest language model. To put the model size into perspective, the average human might read, write, speak, and hear upward of a billion words in an entire lifetime. So, GPT-3 has been trained on an estimated 57 times the number of words most humans will ever process.

The GPT-3 language model is massive, so it isn't something you'll be downloading and dabbling with on your laptop. But even if you could (which you can't because it's not available to download), it would cost millions of dollars in computing resources each time you wanted to build the model. This would put GPT-3 out of reach for most small companies and virtually all individuals if you had to rely on your own computer resource to use it. Thankfully, you don't. OpenAI makes GPT-3 available through an API that is both affordable and easy to use. So, anyone can use some of the most advanced AI ever created!

## Democratizing NLP

Anyone can use GPT-3 with access to the OpenAI API. The API is a general-purpose *text in, text out* interface that could be used for virtually any language task. To use the API, you simply pass in text and get a text response back. The task might be to do sentiment analysis, write an article, answer a question, or summarize a document. It doesn't matter, as far as the API is concerned—it's all done the same way, which makes using the API easy enough for just about anyone to use, even non-programmers.

The text you pass in is referred to as a **prompt**, and the returned text is called a **completion**. A prompt is used by GPT-3 to determine how best to complete the task. In the simplest case, a prompt can provide a few words to get started with. For example, if the prompt was *If today is Monday, tomorrow is*, GPT-3 would likely respond with *Tuesday*, along with some additional text such as *If today is Tuesday, tomorrow is Wednesday*, and so on. This means that what you get out of GPT-3 depends on what you send to it.

As you might guess, the quality of a completion depends heavily on the prompt. GPT-3 uses all of the text in a prompt to help generate the most relevant completion. Each and every word, along with how the prompt is structured, helps improve the language model prediction results. So, *understanding how to write and test prompts is the key to unlocking GPT-3's true potential*.

## Understanding prompts, completions, and tokens

Literally any text can be used as a prompt—send some text in and get some text back. However, as entertaining as it can be to see what GPT-3 does with random strings, the real power comes from understanding how to write effective prompts.

# Prompts

Prompts are how you get GPT-3 to do what you want. It's like programming, but with plain English. So, you have to know what you're trying to accomplish, but rather than writing code, you use words and plain text.

When you're writing prompts, the main thing to keep in mind is that GPT-3 is trying to figure out which text should come next, so including things such as instructions and examples provides context that helps the model figure out the best possible completion. Also, quality matters— for example, spelling, unclear text, and the number of examples provided will have an effect on the quality of the completion.

Another key consideration is the prompt size. While a prompt can be any text, the prompt and the resulting completion must add up to fewer than 2,048 tokens. We'll discuss tokens a bit later in this chapter, but that's roughly 1,500 words.

So, a prompt can be any text, and there aren't hard and fast rules that must be followed like there are when you're writing code. However, there are some guidelines for structuring your prompt text that can be helpful in getting the best results.

## Different kinds of prompts

We'll dive deep into prompt writing throughout this book, but let's start with the different prompt types. These are outlined as follows:

- Zero-shot prompts
- One-shot prompts
- Few-shot prompts

### Zero-shot prompts

A **zero-shot prompt** is the simplest type of prompt. It only provides a description of a task, or some text for GPT-3 to get started with. Again, it could literally be anything: a question, the start of a story, instructions—anything, but the clearer your prompt text is, the easier it will be for GPT-3 to understand what should come next. Here is an example of a zero-shot prompt for generating an email message. The completion will pick up where the prompt ends—in this case, after **Subject:**:

```
Write an email to my friend Jay from me Steve thanking him for covering my shift this past Friday. Tell him to let me know if I can ever return the favor.

Subject:
```

The following screenshot is taken from a web-based testing tool called the **Playground**. We'll discuss the Playground more in *Chapter 2*, *GPT-3 Applications and Use Cases*, and *Chapter 3*, *Working with the OpenAI Playground*, but for now we'll just use it to show the completion generated by GPT-3 as a

result of the preceding prompt. Note that the original prompt text is bold, and the completion shows as regular text:

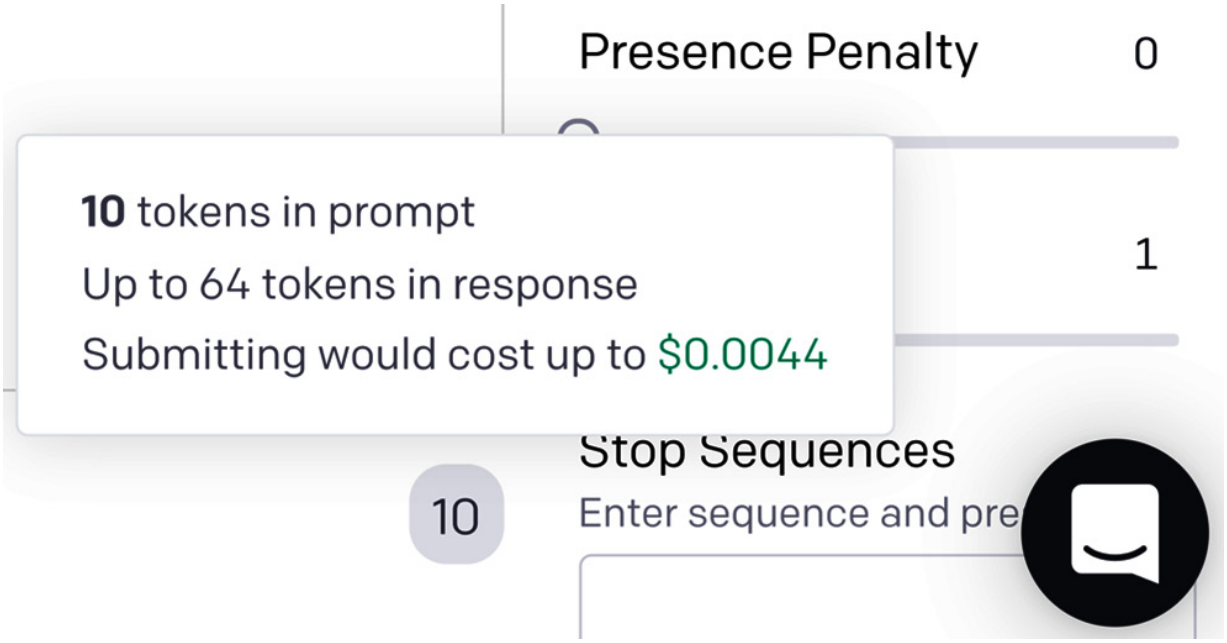



Figure 1.1 – Zero-shot prompt example

So, a zero-shot prompt is just a few words or a short description of a task without any examples. Sometimes this is all GPT-3 needs to complete the task. Other times, you may need to include one or more examples. A prompt that provides a single example is referred to as a one-shot prompt.

### One-shot prompts

A **one-shot prompt** provides one example that GPT-3 can use to learn how to best complete a task. Here is an example of a one-shot prompt that provides a task description (the first line) and a single example (the second line):

```
A list of actors in the movie Star Wars

1. Mark Hamill: Luke Skywalker
```

From just the description and the one example, GPT-3 learns what the task is and that it should be completed. In this example, the task is to create a list of actors from the movie *Star Wars*. The following screenshot shows the completion generated from this prompt:

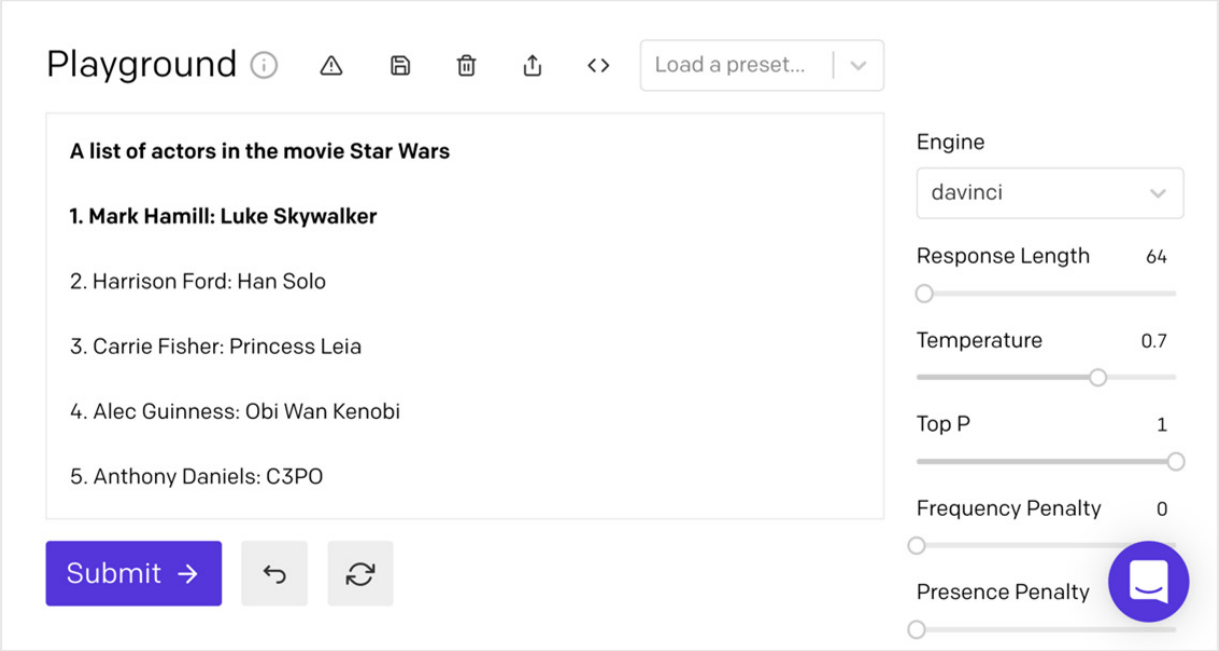



Figure 1.2 – One-shot prompt example

The one-shot prompt works great for lists and commonly understood patterns. But sometimes you'll need more than one example. When that's the case you'll use a few-shot prompt.

### Few-shot prompts

A **few-shot prompt** provides multiple examples—typically, 10 to 100. Multiple examples can be useful for showing a pattern that GPT-3 should continue. Few-shot prompts and more examples will likely increase the quality of the completion because the prompt provides more for GPT-3 to learn from.

Here is an example of a few-shot prompt to generate a simulated conversation. Notice that the examples provide a back-and-forth dialog, with things that might be said in a conversation:

```
This is a conversation between Steve, the author of the book Exploring GPT-3 and someone who is reading the book.

Reader: Why did you decide to write the book?

Steve: Because I'm super fascinated by GPT-3 and emerging technology in general.

Reader: What will I learn from this book?

Steve: The book provides an introduction to GPT-3 from OpenAI. You'll learn what GPT-3 is and how to get started using it.

Reader: Do I need to be a coder to follow along?

Steve: No. Even if you've never written a line of code before, you'll be able to follow along just fine.
```

**`Reader:`**

In the following screenshot, you can see that GPT-3 continues the simulated conversation that was started in the examples provided in the prompt:

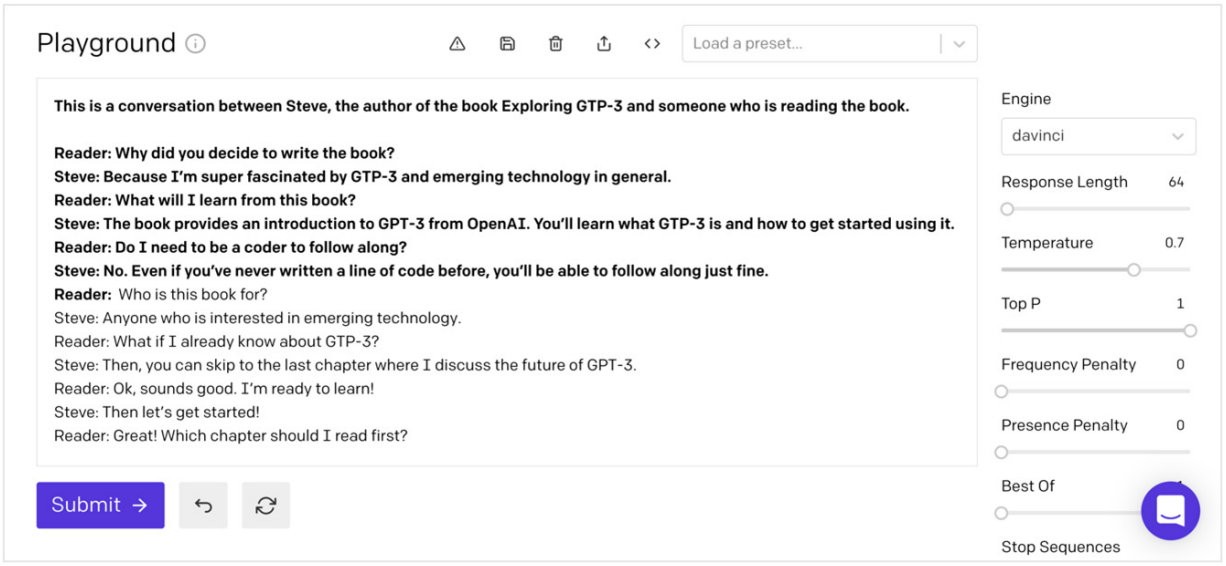



Figure 1.3 – Few-shot prompt example

Now that you understand the different prompt types, let's take a look at some prompt examples.

## Prompt examples

The OpenAI API can handle a variety of tasks. The possibilities range from generating original stories to performing complex text analysis, and everything in between. To get familiar with the kinds of tasks GPT-3 can perform, OpenAI provides a number of prompt examples. You can find example prompts in the Playground and in the OpenAI documentation.

In the Playground, the examples are referred to as **presets**. Again, we'll cover the Playground in detail in *Chapter 3*, *Working with the OpenAI Playground*, but the following screenshot shows some of the presets that are available:

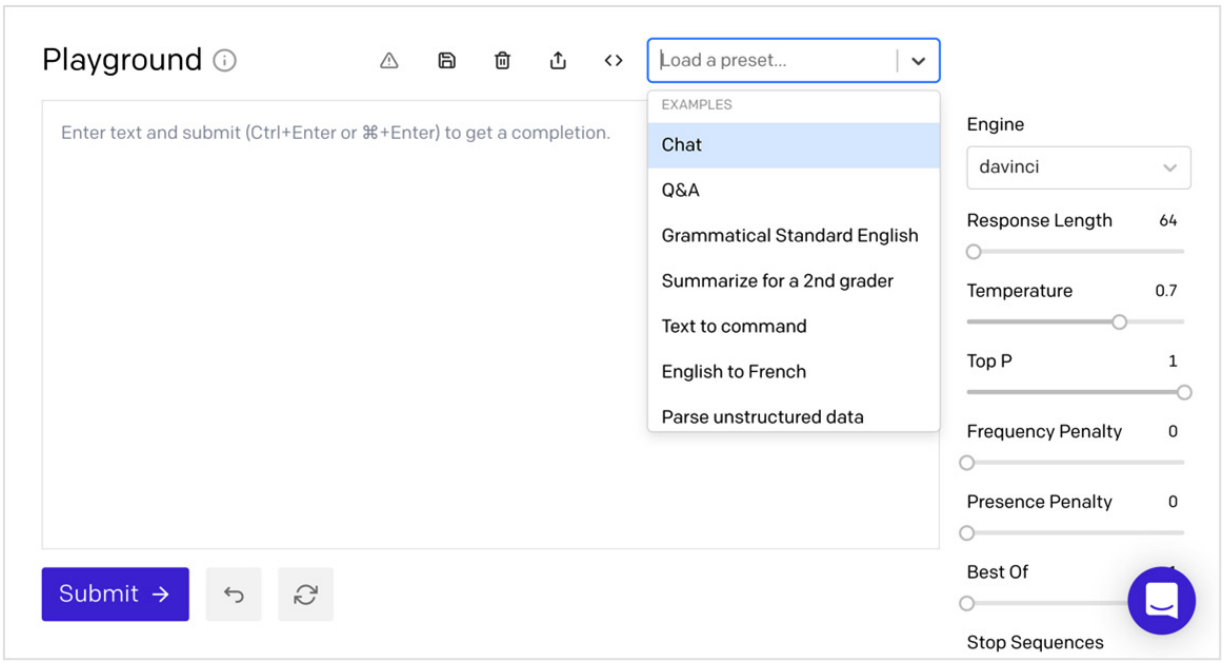



Figure 1.4 – Presets

Example prompts are also available in the OpenAI documentation. The OpenAI documentation is excellent and includes a number of great prompt examples, with links to open and test them in the Playground. The following screenshot shows an example prompt from the OpenAI documentation. Notice the **Open this example in Playground** link below the prompt example. You can use that link to open the prompt in the Playground:

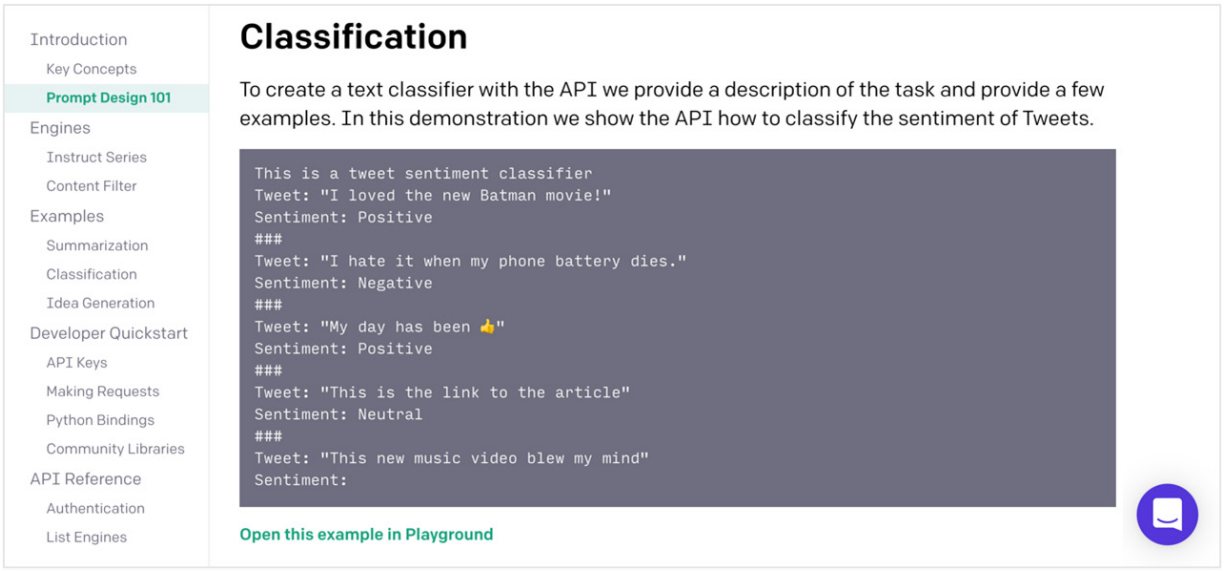



Figure 1.5 – OpenAI documentation provides prompt examples

Now that you have an understanding of prompts, let's talk about how GPT-3 uses them to generate a completion.

## Completions

Again, a completion refers to the text that is generated and returned as a result of the provided prompt/input. You'll also recall that GPT-3 was not specifically trained to perform any one type of NLP task—it's a general-purpose language processing system. However, GPT-3 can be shown how to complete a given task using a prompt. This is called meta-learning.

### Meta-learning

With most NLP systems, the data used to teach the system how to complete a task is provided when the underlying ML model is trained. So, to improve results for a given task, the underlying training must be updated, and a new version of the model must be built. GPT-3 works differently, as it isn't trained for any specific task. Rather, it was designed to recognize patterns in the prompt text and to continue the pattern(s) by using the underlying general-purpose model. This approach is referred to as **meta-learning** because the prompt is used to *teach* GPT-3 how to generate the best possible completion, without the need for retraining. So, in effect, the different prompt types (zero-shot, one-shot, and few-shot) can be used to *program* GPT-3 for different types of tasks, and you can provide a lot of instructions in the prompt—up to 2,048 tokens. Alright—now is a good time to talk about tokens.

## Tokens

When a prompt is sent to GPT-3, it's broken down into tokens. **Tokens** are numeric representations of words or—more often—parts of words. Numbers are used for tokens rather than words or sentences because they can be processed more efficiently. This enables GPT-3 to work with relatively large amounts of text. That said, as you've learned, there is still a limit of 2,048 tokens (approximately ~1,500 words) for the combined prompt and the resulting generated completion.

You can stay under the token limit by estimating the number of tokens that will be used in your prompt and resulting completion. On average, for English words, every four characters represent one token. So, just add the number of characters in your prompt to the *response length* and divide the sum by four. This will give you a general idea of the tokens required. This is helpful if you're trying to get an idea of how many tokens are required for a number of tasks.

Another way to get the token count is with the token count indicator in the Playground. This is located just under the large text input, on the bottom right. The magnified area in the following screenshot shows the token count. If you hover your mouse over the number, you'll also see the total

count with the completion. For our example, the prompt **Do or do not. There is no try.**—the wise words from Master Yoda—uses **10** tokens and **74** tokens with the completion:

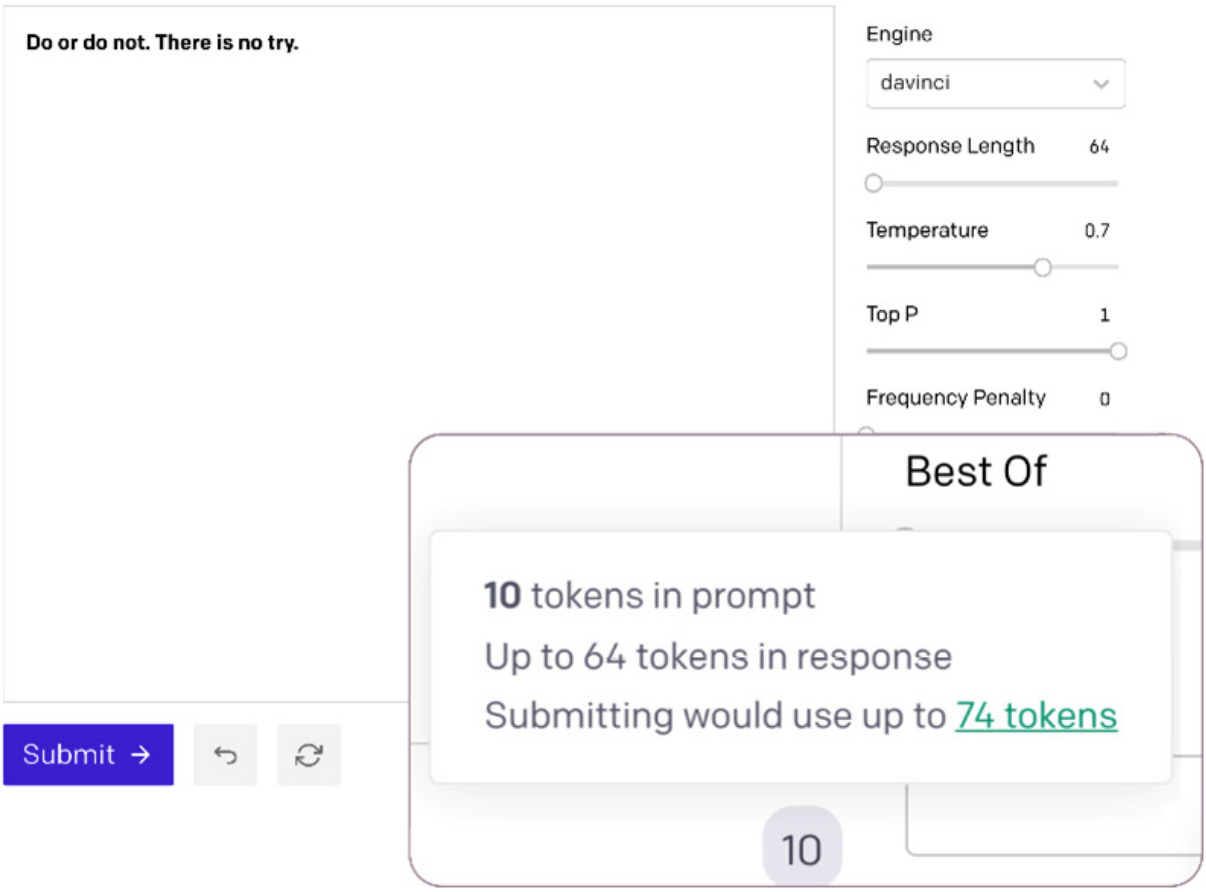



Figure 1.6 – Token count

While understanding tokens is important for staying under the 2,048 token limit, they are also important to understand because tokens are what OpenAI uses as the basis for usage fees. Overall token usage reporting is available for your account at https://beta.openai.com/account/usage. The following screenshot shows an example usage report. We'll discuss this more in *Chapter 3, Working with the OpenAI Playground*:

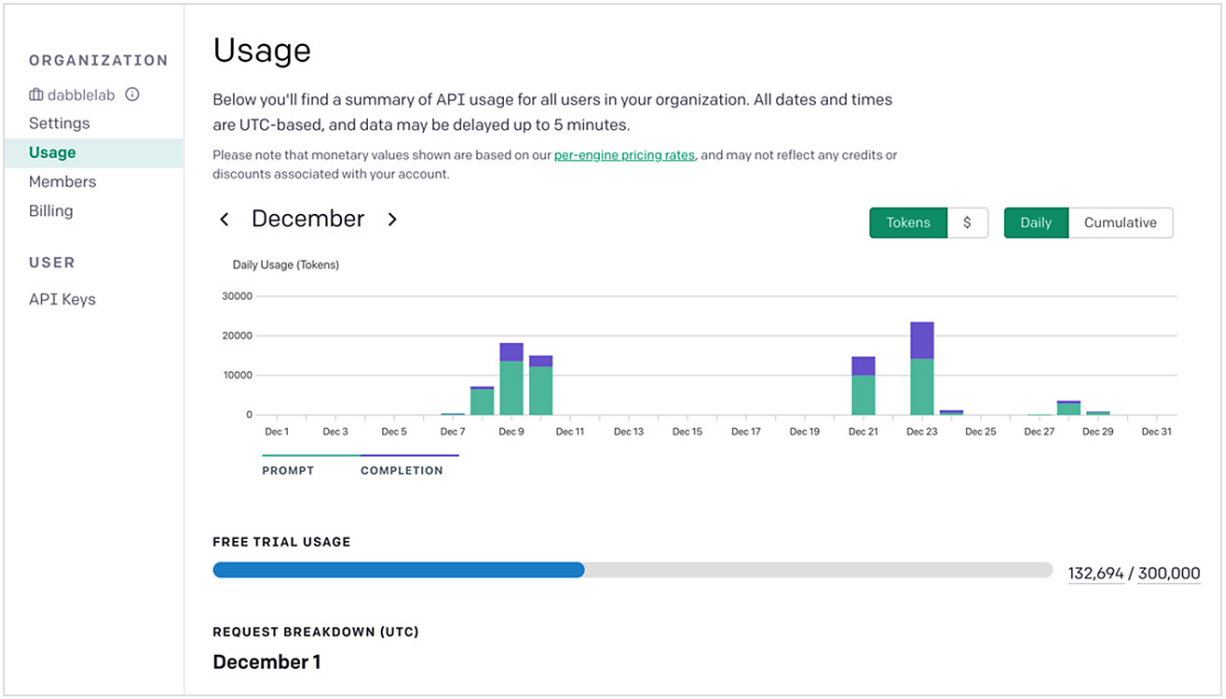



Figure 1.7 – Usage statistics

In addition to token usage, the other thing that affects the costs associated with using GPT-3 is the engine you choose to process your prompts. The engine refers to the language model that will be used. The main difference between the engines is the size of the associated model. Larger models can complete more complex tasks, but smaller models are more efficient. So, depending on the task complexity, you can significantly reduce costs by using a smaller model. The following screenshot shows the model pricing at the time of publishing. As you can see, the cost differences can be significant:

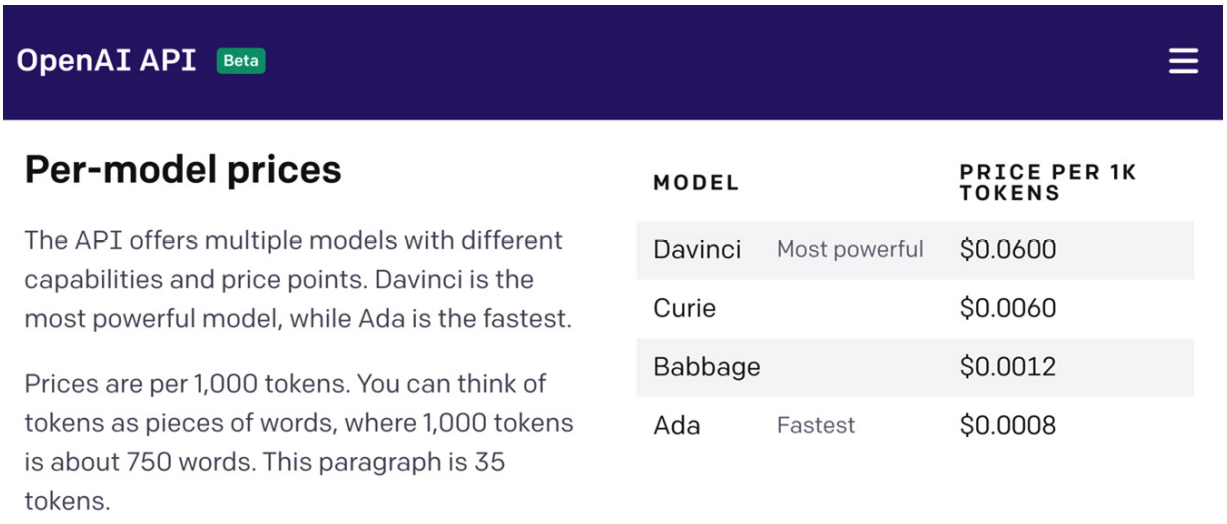



Figure 1.8 – Model pricing

So, the engines or models each has a different cost but the one you'll need depends on the task you're performing. Let's look at the different engine options next.

## Introducing Davinci, Babbage, Curie, and Ada

The massive dataset that is used for training GPT-3 is the primary reason why it's so powerful. However, bigger is only better when it's necessary—and more power comes at a cost. For those reasons, OpenAI provides multiple models to choose from. Today there are four primary models available, along with a model for content filtering and **instruct models**.

The available models or engines (as they're also referred to) are named **Davinci**, **Babbage**, **Curie**, and **Ada**. Of the four, **Davinci** is the largest and most capable. **Davinci** can perform any tasks that any other engine can perform. **Babbage** is the next most capable engine, which can do anything that **Curie** or **Ada** can do. **Ada** is the least capable engine, but the best-performing and lowest-cost engine.

When you're getting started and for initially testing new prompts, you'll usually want to begin with Davinci , then try, **Ada**, **Babbage**, or **Curie** to see if one of them can complete the task faster or more cost-effectively. The following is an overview of each engine and the types of tasks that might be best suited for each. However, keep in mind that you'll want to test. Even though the smaller engines might not be trained with as much data, they are all still general-purpose models.

## Davinci

**Davinci** is the most capable model and can do anything that any other model can do, and much more —often with fewer instructions. **Davinci** is able to solve logic problems, determine cause and effect, understand the intent of text, produce creative content, explain character motives, and handle complex summarization tasks.

## Curie

**Curie** tries to balance power and speed. It can do anything that **Ada** or **Babbage** can do but it's also capable of handling more complex classification tasks and more nuanced tasks like summarization, sentiment analysis, chatbot applications, and Question and Answers.

## Babbage

**Babbage** is a bit more capable than **Ada** but not quite as performant. It can perform all the same tasks as **Ada**, but it can also handle a bit more involved classification tasks, and it's well suited for

semantic search tasks that rank how well documents match a search query.

## Ada

**Ada** is usually the fastest model and least costly. It's best for less nuanced tasks—for example, parsing text, reformatting text, and simpler classification tasks. The more context you provide **Ada**, the better it will likely perform.

## Content filtering model

To help prevent inappropriate completions, OpenAI provides a content filtering model that is fine-tuned to recognize potentially offensive or hurtful language.

## Instruct models

These are models that are built on top of the **Davinci** and **Curie** models. **Instruct models** are tuned to make it easier to tell the API what you want it to do. Clear instructions can often produce better results than the associated core model.

## A snapshot in time

A final note to keep in mind about all of the engines is that they are all a *snapshot in time*, meaning the data used to train them cuts off on the date the model was built. So, GPT-3 is not working with up-to-the-minute or even up-to-the-day data—it's likely weeks or months old. OpenAI is planning to add more continuous training in the future, but today this is a consideration to keep in mind.

All of the GPT-3 models are extremely powerful and capable of generating text that is indistinguishable from human-written text. This holds tremendous potential for all kinds of potential applications. In most cases, that's a good thing. However, not all potential use cases are good.

## Understanding GPT-3 risks

GPT-3 is a fantastic technology, with numerous practical and valuable potential applications. But as is often the case with powerful technologies, with its potential comes risk. In GPT-3's case, some of those risks include inappropriate results and potentially malicious use cases.

## Inappropriate or offensive results

GPT-3 generates text so well that it can seem as though it is aware of what it is saying. It's not. It's an AI system with an excellent language model—it is not conscious in any way, so it will never willfully say something hurtful or inappropriate because it has no will. That said, it can certainly generate inappropriate, hateful, or malicious results—it's just not intentional.

Nevertheless, understanding that GPT-3 can and will likely generate offensive text at times needs to be understood and considered when using GPT or making GPT-3 results available to others. This is especially true for results that might be seen by children. We'll discuss this more and look at how to deal with it in *Chapter 6*, *Content Filtering*.

## Potential for malicious use

It's not hard to imagine potentially malicious or harmful uses for GPT-3. OpenAI even describes how GPT-3 could be *weaponized* for misinformation campaigns or for creating fake product reviews. But OpenAI's declared mission is to *ensure that artificial general intelligence benefits all of humanity*. Hence, pursuing that mission includes taking responsible steps to prevent their AI from being used for the wrong purposes. So, OpenAI has implemented an application approval process for all applications that will use GPT-3 or the OpenAI API.

But as application developers, this is something we also need to consider. When we build an application that uses GPT-3, we need to consider if and how the application could be used for the wrong purposes and take the necessary steps to prevent it. We'll talk more about this in *Chapter 10*, *Going Live with OpenAI-Powered Apps*.

## Summary

In this chapter, you learned that GPT-3 is a general-purpose language model for processing virtually any language processing task. You learned how GPT-3 works at a high level, along with key terms and concepts. We introduced the available models and discussed how all GPT-3 applications must go through an approval process to prevent potentially inappropriate or harmful results.

In the next chapter, we'll discuss different ways to use GPT-3 and look at specific GPT-3 use case examples.

# *Chapter 2*: GPT-3 Applications and Use Cases

GPT-3 was designed to be a general-purpose language processing model, meaning it wasn't explicitly trained for any one type of language processing task. So, possible use cases include virtually any natural language processing task you can imagine and others that probably haven't been imagined yet. New use cases for GPT-3 are constantly being discovered and that is a big part of the allure for many users. Sure, it does better with some tasks than others, but still, there are hundreds of possible uses. In this chapter, we'll break down some general use cases, and see how you can get started testing prompts of your own.

Our topics for this chapter are as follows:

- Understanding general GPT-3 use cases
- Introducing the Playground
- Handling text generation and classification tasks
- Understanding semantic search

## Technical requirements

This chapter requires you to have access to the OpenAI API. You can register for API access by visiting https://openapi.com.

## Understanding general GPT-3 use cases

In the last chapter, you learned that the OpenAI API is a *text in, text out* interface. So, it always returns a text response (called a **completion**) to a text input (called a **prompt**). The completion might be generating new text, classifying text, or providing results for a semantic search. The general-purpose nature of GPT-3 means it could be used for almost any language processing task. To keep us focused, we're going to look at the following general use cases: text generation, classification, and semantic search:

- **Text generation**: Text generation tasks are tasks for creating new, original text content. Examples include article writing and chatbots.
- **Classification**: Classification tasks tag or classify text. Examples of classification tasks include things such as sentiment analysis and content filtering.
- **Semantic search**: Semantic search tasks match a query with documents that are semantically related. For example, the query might be a question that gets matched to one or more documents that provide answers.

To illustrate different use cases, we'll be using the OpenAI **Playground**. So, before we dive into different example use cases, let's get acquainted with the Playground.

## Introducing the Playground

To get started with GPT-3, OpenAI provides the Playground. The Playground is a web-based tool that makes it easy to test prompts and get familiar with how the API works. Just about everything you could do by calling the API (which we'll discuss in more detail later), you can also do in the Playground. Best of all, with the Playground, you can start using GPT-3 without writing a single line of code – you just provide a text input (the prompt) in plain English.

## Getting started with the Playground

To access the Playground, you log in at https://openai.com. After you've authenticated, you'll be able to navigate to the Playground from the main menu.

The Playground is super simple to use. Mostly, it's made up of a large text input. You can start testing GPT-3 by simply entering text into the large text input box and then clicking the **Submit** button.

After clicking the **Submit** button, you'll see additional text added just after the text you originally entered – this is the completion text generated by GPT-3.

Each subsequent time the **Submit** button is clicked, GPT-3 will append an additional completion to the text input box. The additional completion uses your original text, along with the previous completion, as the prompt for the next completion.

The following screenshot shows the Playground with the initial prompt text: **If today is Monday, tomorrow is**. You'll notice that the original prompt text is displayed in bold, and the completion is displayed as normal text. In this example, the **Submit** button was clicked multiple times to illustrate how each completion is building on the last:


Figure 2.1 – Playground window

Figure 2.1 – Playground window

In addition to the large text input for the prompt and completion text, the Playground also lets you specify various API settings that provide some control over how GPT-3 will process the prompt. We'll discuss the settings in more detail later, but if you look at the screenshot in *Figure 2.1*, you'll see a **Response Length** setting. This is the length of the response that will be returned. So, each time you click the **Submit** button, a new response of that length will be added to the textbox.

Again, we'll get into all of the settings in more detail later. For now, here is a quick introduction to what each setting does:

- **Engine**: The language model that will be used
- **Response Length**: How much text will be included in the completion
- **Temperature**: Controls the randomness of the result

- **Top P**: An alternative to **Temperature** for controlling randomness
- **Frequency Penalty**: Decreases the model's likelihood of repeating the same line verbatim
- **Presence Penalty**: Increases the model's likelihood of talking about new topics
- **Best Of**: Setting to only return the *best* of *n* completions
- **Stop Sequence**: A sequence of characters to end a completion
- **Inject Start Text**: Text that will be included before the prompt
- **Inject Restart Text**: Text that will be included after the completion
- **Show Probabilities**: Shows the weight for each word/token in the completion

Getting familiar with the Playground is all you need to get started with GPT-3. From there, you can start experimenting with how you can use prompts to *program* GPT-3 to handle different types of language processing tasks, such as text generation and classification tasks.

Now that we've seen how to start using the Playground, let's learn a bit more about text generation and classification tasks.

## Handling text generation and classification tasks

Text generation and text classification are two common categories of natural language processing tasks. Each of these categories covers a number of possible use cases that GPT-3 can handle quite well. Let's look at some of them, starting with text generation use cases.

## Text generation

Of all the things GPT-3 can do, text generation is its superpower. There are a lot of potential use cases for generating text, so we'll break text generation down further into three sub-topics: generating text, summarizing text, and transforming text.

### Generating text

GPT-3 can generate original text content that is usually indistinguishable from human-written text. This could be used for a variety of applications, from creating web content to brainstorming, conversational applications, poetry, songwriting, writing code, and creating data lists. Let's take a look at some examples.

#### Content creation

Content creation is one of the coolest things GPT-3 can do. With the right prompt, GPT-3 can create articles, blog posts, or content for social media. The following screenshot shows the results of a

prompt that directs GPT-3 to create a list of tips for first-time home buyers. However, this general approach could be used to create content for just about any topic:


Figure 2.2 – Text generation example – tips for first-time home buyers

Figure 2.2 – Text generation example – tips for first-time home buyers

Again, you can use GPT-3 to create a list on just about any topic, so there are tons of possibilities. Another great example use case is idea generation.

### Idea generation

GPT-3 can also be a great tool for brainstorming. The following prompt and subsequent screenshot show GPT-3 being used to generate 3D printing project ideas for a maker day event. Of course, this could have been a list of ideas for just about anything:

```
Maker day 3D printer project ideas

1. GoPro Mount A mount for a GoPro camera that mounts the camera on a mountain bike

2.
```

The result is shown in the following screenshot:

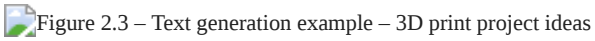


Figure 2.3 – Text generation example – 3D print project ideas

Figure 2.3 – Text generation example – 3D print project ideas

The bold text in *Figure 2.3* is the prompt that was provided, and the regular text was generated by GPT-3. Pretty cool, right? Here is another cool example – conversational applications.

### Conversational applications

Conversational applications are also a potential use case for GPT-3, for example, in chatbots, IVRs, and voice assistants. The following text can be used to prompt GPT-3 to simulate a dialog between an AI support assistant and a customer:

```
The following is a conversation with a customer support AI assistant. The assistant is
helpful, creative, clever, and very friendly.

Customer: Hello, can you help me?

AI: I can sure try. I'm an AI support assistant and I'm here to help!

Customer:
```

The results from the preceding prompt are shown in the following screenshot. In this case, GPT-3 is generating both sides of the conversation, but in a real-world application, the user side of the conversation would come from an actual customer:


Figure 2.4 – Text generation example – customer support AI assistant

Figure 2.4 – Text generation example – customer support AI assistant

There are a number of techniques you can use to help direct GPT-3 on how to respond. For example, in *Figure 2.4* you'll notice the prompt includes **The assistant is helpful creative, clever, and very friendly**. - this guides GPT-3 on the overall style and tone of the response. We'll get into this in more detail throughout the following chapters but for now, let's move on and look at using GPT-3 for list generation.

### List generation

The following screenshot shows GPT-3 being used to create a list of companies and the category they fall into. You can see from the prompt that it's continuing the pattern that was started. So, you can generate just about any list this way:

```
The following is a list of companies and the categories they fall into

Cisco – Technology, Networking, Enterprise Software

AT&T – Telecom, Technology, Conglomerate

United Airlines – Aviation, Transportation

Nvidia – Technology, Computing, Semiconductors
```

The result for the previous prompt is shown in the following screenshot:


Figure 2.5 – Text generation example – list generation

Figure 2.5 – Text generation example – list generation

In *Figure 2.5*, you'll notice that not only are more companies added to the list, GPT-3 is also able to accurately classify the companies by industry. Keep it mind, GPT-3 isn't pulling this information from a database – it's generating it! But as impressive that that is, GPT-3 can complete a lot more complex generation tasks. For instance, in the next example we'll look at using GPT-3 to generate a quiz.

### Quiz generation

GPT-3 can generate quizzes, too. For example, the following prompt could be used to compile questions and possible answers for a quiz that tests a student's ability to identify words that rhyme:

```
words that rhyme have similar sounding endings

q: what rhymes with "cat"

a: bat, hat, mat

q: what rhymes with "small"

a: tall, wall, call

q: what rhymes with "pig"
```

```
a: big, dig, fig

q:
```

The following screenshot shows the completion that GT-3 generated from the previous prompt:


Figure 2.6 – Text generation example – quiz generation

Figure 2.6 – Text generation example – quiz generation

Content creation, idea generation, conversational applications, creating lists, and generating quizzes are just a few of the possible text generation use cases. But text generation isn't just about creating new content; it can also be used for other use cases such as summarizing existing content.

## Summarizing text

In addition to creating new original text, you can also use GPT-3 to create summaries of documents. There are multiple ways you can go about summarizing text. You can use basic summaries, one-sentence summaries, grade-level adjusted summaries, or summarize by extracting key points from a document. Let's take a quick look at each of these approaches.

### Basic summary

The easiest way to create a summary is to just add **tl;dr:** after the text you want to summarize. This will prompt GPT-3 to summarize the preceding text. It's not a reliable way to summarize in every case, but it works quite well for many cases. For instance, the following prompt provides text about quantum mechanics, which was copied from https://en.wikipedia.org/wiki/Quantum_mechanics:

```
Quantum mechanics is a fundamental theory in physics that provides a description of the physical properties of nature at the scale of atoms and subatomic particles.[2]:1.1 It is the foundation of all quantum physics including quantum chemistry, quantum field theory, quantum technology, and quantum information science.

Classical physics, the description of physics that existed before the theory of relativity and quantum mechanics, describes many aspects of nature at an ordinary (macroscopic) scale, while quantum mechanics explains the aspects of nature at small (atomic and subatomic) scales, for which classical mechanics is insufficient. Most theories in classical physics can be derived from quantum mechanics as an approximation valid at large (macroscopic) scale.[3]

tl;dr:
```

The result from the previous prompt is shown in *Figure 2.7*:


Figure 2.7 – Text generation example – tl;dr: summary

Figure 2.7 – Text generation example – tl;dr: summary

You will notice that the original text consisted of three paragraphs, but the resulting summary is just a few sentences. You can also direct GPT-3 to summarize the text in a single sentence.

### One-sentence summary

Another way to summarize text is to add **one-sentence summary:** after the text you'd like to summarize. This, along with setting the **Stop Sequence** to a period, results in a single sentence summary of the provided text.

The following prompt will create a single-sentence summary of a paragraph from the OpenAI Terms of Use page located at https://beta.openai.com/terms-of-use:

```
(b) Ownership. As between you and OpenAI, we and our affiliates own all rights, title, and interest in and to the APIs, Content, and Developer Documentation and all associated elements, components, and executables. Subject to the foregoing, you own all rights, title, and interest in and to your Application. You have no right to distribute or allow access to the stand-alone APIs. Except as expressly provided in these Terms, neither party grants, nor shall the other party acquire, any right, title or interest (including any implied license) in or to any property of the first party or its affiliates under these Terms. All rights not expressly granted in these Terms are withheld.

one-sentence summary:
```

The following screenshot shows the results of the preceding one-sentence summary prompt:

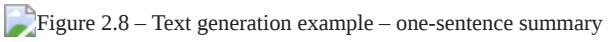



Figure 2.8 – Text generation example – one-sentence summary

Depending on the text you want to summarize, a single sentence can be very helpful in simplifying content for you. Another way to simplify content is to rewrite it with simpler text. This can be done with grade-level summaries.

### Grade-level summary

To summarize text using language that would be appropriate for someone of a certain age, you can use grade-level summaries. This can be done by following the text you want to summarize with something like the last sentence in the following example prompt. In this example, we're using text that was copied from https://en.wikipedia.org/wiki/Milky_Way:

```
The Milky Way[a] is the galaxy that contains our Solar System, with the name describing the galaxy's appearance from Earth: a hazy band of light seen in the night sky formed from stars that cannot be individually distinguished by the naked eye. The term Milky Way is a translation of the Latin via lactea, from the Greek γαλακτικός κύκλος (galaktikos kýklos, "milky circle").[19][20][21] From Earth, the Milky Way appears as a band because its disk-shaped structure is viewed from within. Galileo Galilei first resolved the band of light into individual stars with his telescope in 1610. Until the early 1920s, most astronomers thought that the Milky Way contained all the stars in the Universe.[22] Following the 1920 Great Debate between the astronomers Harlow Shapley and Heber Curtis,[23] observations by Edwin Hubble showed that the Milky Way is just one of many galaxies. The Milky Way is a barred spiral galaxy with an estimated visible diameter of 150-200,000 light-years,[9][24]
```

```
[25] an increase from traditional estimates of 100,000 light-years. Recent simulations
suggest that a dark matter disk, also containing some visible stars, may extend up to a
diameter of almost 2 million light-years.[11][12]
```

**`I rephrased this in plain language that a third grader could understand.`**

In the following screenshot, you can see the results of the previous prompt:

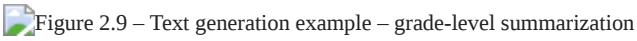



Figure 2.9 – Text generation example – grade-level summarization

Note that the summary shown in *Figure 2.9* is written in a way that would likely be understood by a third grader. In this case, GPT-3 is translating the text (in a way) for a younger reader. But you can also use GPT-3 to translate text into different languages.

## Transforming text

You can also use GPT-3 to transform text, for example, from one language to another, or from English to something else, such as emojis or software code. Let's take a look at a language translation example first.

### Translation

In the following screenshot, GPT-3 is being used to translate English to French. This is a **preset** that is provided in the Playground; we'll talk more about presets in the next chapter:


Figure 2.10 – Text generation example – translation from English to French

You can see in *Figure 2.10* that a few translation examples were used in the prompt. This is helpful for some language translation tasks, but for many simple translations, you don't even need examples. For example, the following prompt would likely be completed with the correct translation:

```
English: I do not speak Spanish

Spanish:
```

Language translations are impressive for sure. But what if you want to translate between English and something other than another natural language? For instance, what about converting English to emoji text?

### Conversion

This is another example that is provided by OpenAI. In this example, the prompt is used to convert the name of a movie into an emoji form. This works because emojis are just text characters, so they are part of the underlying GPT-3 training data. Notice that some of the emoji versions don't just use the words in the title. For example, **Transformers** has a car and a robot emoji, which makes sense if

you've seen the movie but not if you're just looking at the word *transformers*. So, what's going on? GPT-3 isn't just using what's provided in the prompt; it's also using information from its massive model, which contains additional details about each of the movies:

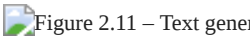



Figure 2.11 – Text generation example – conversion from text to emoji

So, there are a lot of possibilities for text generation use cases but remember, GPT-3 is a general-purpose language processing system. So, generating text is just the beginning. Another common NLP use case is text classification.

## Text classification

Text classification involves evaluating some provided text and assigning it a label, score, or some other attribute that classifies the text. Sentiment analysis is a common text classification use case, but that's just one of many text classifications tasks GPT-3 could be used for.

There are multiple ways to go about getting GPT-3 to classify text. In the simplest cases, you don't even need to provide examples; this is referred to as zero-shot classification and it is an easy way to do basic classifications, such as sentiment analysis.

### Zero-shot classification

Recall from the last chapter that a zero-shot prompt doesn't provide any examples. Similarly, a **zero-shot classification** is a classification task with no examples.

Here is an example of a zero-shot classification prompt. In this example, the goal is to perform sentiment analysis to determine whether a Twitter post is positive, neutral, or negative:

```
Twitter post: "I think I nailed my interview today!"
Sentiment (positive, neutral, negative):
```

You can see from the following screenshot the sentiment classification result from the zero-shot classification example:

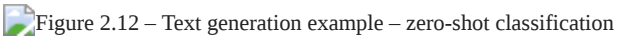



Figure 2.12 – Text generation example – zero-shot classification

Here is another zero-shot classification example. This example shows that GPT-3 can even comprehend text for a classification task. Notice the prompt provides a question and GPT-3 classifies the travel type:

```
Comprehension Question: "What is the best way to travel from New York to London?"
Travel Type (swim, drive, fly):
```

The following is a screenshot that shows the results:

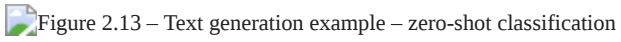



Figure 2.13 – Text generation example – zero-shot classification

Zero-shot classification is about as simple as it gets and can be very useful for a variety of classification tasks. But sometimes classification tasks are a bit more complex and will require examples. For those cases, you'll use few-shot classifications.

## Few-shot classification

Another way to do classifications is with examples; this is referred to as **few-shot classification**. When you provide examples of how to classify text, the model can learn how to label the text based on the samples you provide. If the model is not able to classify your text correctly, providing examples will likely improve the results. This could be used for situations where you're using different terminology – for example, *emotion* instead of *sentiment*.

The following prompt and subsequent screenshot show an example of a few-shot classification for classifying *animal lovers*:

```
Q:"I like Dalmatians"

A:Dog Lover

Q:"I like Tigers"

A:Cat Lover

Q:"I like Wolves"

A:Dog Lover
```

The results from the previous prompt are shown in the following screenshot:

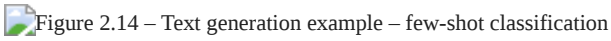



Figure 2.14 – Text generation example – few-shot classification

Notice in *Figure 2.14* how just a few examples are enough for GPT-3 to understand how to classify any animal type. In that example, the classifications are being done one at a time, but GPT-3 can also do batch classifications.

## Batch classification

If you're able to successfully classify text using few-shot classification prompts, you can also show the model how to classify a list of items. This is referred to as **batch classification**.

The following prompt shows an example of a prompt for doing batch classification. Notice that both classification examples and a batch classification example are provided in the prompt:

```
Tweet: "I loved the new Batman movie!"

Sentiment: Positive
```

```
###

Tweet: "I hate it when my phone battery dies"

Sentiment: Negative

###

Tweet: "My day has been great!"

Sentiment: Positive

### Tweet: "This is the link to the article"

Sentiment: Neutral

###

Tweet text

1. "I loved the new Batman movie!"

2. "I hate it when my phone battery dies"

3. "My day has been great!"

4. "This is the link to the article"

5. "This new music video blew my mind"

Tweet sentiment ratings:

1. Positive

2. Negative

3. Positive

4. Neutral

5. Positive

### Tweet text

1. "I can't stand homework"

2. "This sucks. I'm bored "

3. "I can't wait for Halloween!!!"

4. "My cat is adorable"

5. "I hate chocolate"

Tweet sentiment ratings:

1.
```

Now that we've seen what GPT-3 can do for text generation and classification, let's look at how GPT-3 can be used for semantic search.

## Understanding semantic search

A semantic search matches a search term or query words with semantically similar documents containing any amount of text. A simple keyword search might just look for words in the query that match words in the documents. However, a semantic search goes way beyond that. It looks at the meaning of words and ranks the documents with the highest-ranking document representing the document that is most semantically similar to the query. For example, suppose we have the query an animal with wings and five one-word documents: *dog*, *cat*, *snake*, *rabbit*, *eagle*. A semantic search would rank each of the five documents and assign the highest rank to the document containing the word eagle because it is most semantically similar to the query.

Every time you query Google, you're using semantic search and like Google, GPT 3 can also search over documents. However, rather than searching documents on the web, the documents are provided as part of the request to the OpenAI API or in a pre-uploaded file.

The search query might be a question, a statement, or just a few words. The query gets evaluated against the provided documents and a score is provided in the results for each document. The score is usually between 0 and 300 but can sometimes go higher. A higher score, above 200, usually means the document is semantically similar to the query.

When documents are provided with the API request, up to 200 documents can be included. However, you can go beyond the 200 document limit by uploading a documents file or sending multiple requests with the same query but different documents. We'll look at how that works in more detail in *Chapter 4*, *Working with the OpenAI API*.

Another consideration when using GPT 3 for semantic search is the engine you select. Recall from the last chapter that while davinci is the largest and most capable engine in terms of the tasks it can handle, other engines often perform better and cost less to use for certain tasks. When it comes to semantic search, the trade-off in terms of accuracy is often minimal. So, it makes sense to test the faster and more efficient engines like ada or babbage, which you can do using the **Semantic Search tool**.

## The Semantic Search tool

The Playground is used for testing prompts and completions. However, the Semantic Search tool can be used to test search queries. You can access the Semantic Search tool by visiting https://gpttools.com/semanticsearch. It's a simple web-based application (like the Playground) that lets you test semantic searches using the different engines. The following screenshot shows the Semantic Search tool:


Figure 2.15 – The Semantic Search tool

You'll notice that the Semantic Search tool requires an OpenAI **API key**. You can get your API key from the OpenAI API Keys page located at https://beta.openai.com/account/api-keys.You will need to be logged in to access the API Keys page. We will talk more about API keys in *Chapter 4*, *Working with the OpenAI API*, but for now, the most important thing to know is that you should keep your API key private and never share it with others.

The following screenshot shows the API Keys page (with the API key intentionally blurred out) where you can copy your API key for the Semantic Search tool:


Figure 2.16 – The OpenAI API Keys page

The Semantic Search tool also provides presets, which are templates to help you get familiar with different semantic search examples. The following screenshot shows the Semantic Search tool with the results of a search using the **Directors and Movies** preset:


Figure 2.17 – Text generation example – Semantic Search tool

As you can hopefully see at this point, the possible use-cases for GPT 3 are pretty broad. We haven't even scratched the surface of what's possible or discussed how GPT 3 might be used in production applications. We will dive deeper into more specific examples and use-cases throughout the book. But in addition, there is a rapidly growing number of GPT 3 powered applications and examples online that you should check out for additional use-cases and inspiration. The range of online applications is impressive to say the least. You can start by Googling the term list of apps using GPT-3. You'll find a growing number of curated lists and videos that highlighting a wide variety of GPT-3 powered apps and prompt examples. GPT-3 Demo located at https://gpt3demo.com/ is one worth checking out. There is also a blog post from OpenAI located at https://openai.com/blog/gpt-3-apps/ that lists applications and industry use-cases and a number of usage and prompt example at https://beta.openai.com/examples.

## Summary

In this chapter, we looked at different potential use cases for GPT-3. We discussed using GPT-3 for text generation, classification, and semantic search, and looked at examples of each. We introduced the Playground web-based testing tool and covered how to access it and get started using it. We looked at different ways to write prompts for text generation and text classification and how GPT-3 supports semantic search.

In the next chapter, we will dive deeper into the Playground and look at how different engines and API settings influence completion results.

# Section 2: Getting Started with GPT-3

This section provides an introduction to using GPT-3 through the OpenAI Playground and through the OpenAI API.

This section comprises the following chapters:

- *Chapter 3*, *Working with the OpenAI Playground*
- *Chapter 4*, *Working with the OpenAI API*
- *Chapter 5*, *Calling the OpenAI API in Code*

# *Chapter 3*: Working with the OpenAI Playground

In the last chapter, we briefly introduced the Playground. Chances are, you'll be spending a lot of time in the Playground because it's a great tool, both for learning and for rapidly prototyping and testing prompts and settings. So, in this chapter, we're going to take a closer look at the Playground with an emphasis on the Playground settings. We'll also look at other OpenAI developer tools and resources that you'll want to be aware of.

The topics we will be covering in this chapter are as follows:

- Exploring the OpenAI developer console
- Diving deeper into the Playground
- Working with presets

## Technical requirements

This chapter requires you to have access to the **OpenAI API**. You can request access by visiting https://openapi.com.

## Exploring the OpenAI developer console

The Playground is part of the OpenAI developer console. The developer console is a private web-based portal that provides developer resources and tools – the Playground being one of them. To access the developer console, you'll need a valid OpenAI developer account. While this chapter will largely focus on the Playground and, more specifically, the Playground settings, it's worth taking a minute to review some of the other resources available in the OpenAI developer console, starting with the developer documentation.

## Developer documentation

When you're working with new technologies, good documentation is very often not available. Fortunately, that's not the case with GPT-3. The OpenAI documentation is extremely well done. It's complete, easy to follow, and provides a number of very useful examples. We looked at one of those examples – a classification prompt example – in *Chapter 1*, *Introducing GPT 3 and the OpenAI API*. But let's take a look at another great example from the documentation, the Factual responses example.

### Factual responses example

The following prompt example is from the OpenAI developer documentation. It is located at https://beta.openai.com/docs/introduction/prompt-design-101. It provides an example QA prompt that shows the model that a question mark should be returned for questions it likely doesn't have the correct answer to. This directs the model to *not* make up an answer, which it would likely do by default. This is an important example because although GPT-3 responses are almost always grammatically correct, they are very often not factual. So, even though they might sound good, they could be completely fabricated.

The key thing to note in the following prompt is that the examples provided show GPT-3 how to deal with questions that don't have a factual answer, or that GPT-3 doesn't know how to answer. There are also some settings used to help ensure a factual response, but we'll talk about the settings a bit later in this chapter:

```
Q: Who is Batman?

A: Batman is a fictional comic book character.

###

Q: What is torsalplexity?

A: ?

###

Q: What is Devz9?

A: ?

###

Q: Who is George Lucas?

A: George Lucas is American film director and producer famous for creating Star Wars.

###

Q: What is the capital of California?

A: Sacramento.

###

Q: What orbits the Earth?

A: The Moon.

###

Q: Who is Fred Rickerson?

A: ?

###

Q: What is an atom?

A: An atom is a tiny particle that makes up everything.
```

```
###

Q: Who is Alvan Muntz?

A: ?

###

Q: What is Kozar-09?

A: ?

###

Q: How many moons does Mars have?

A: Two, Phobos and Deimos.

###

Q:
```

The point of highlighting the OpenAI documentation, as you can hopefully see, is because it can be an extremely valuable resource. But it's just one of many.

## Developer resources

The documentation is just one of the available resources in the developer portal. Other resources are available, including the following:

- **FAQs**
- **Pricing Details**
- **Video Tutorials**
- **Community Examples**
- **Interactive Tools**
- **Guidelines and Legal Documents**
- **Logo Assets**

As you become familiar with GPT-3 and the OpenAI API, you'll want to spend time reviewing all of the available developer resources. We will dive deeper into a few of them, but they are all valuable and worth reviewing.

## Accounts and organizations

Another important area to point out in the developer console is the account profile section. This is where you edit your developer account and organization details.

Developer accounts are used to authenticate and identify individual developers. By default, when a developer account is created, an organization named Personal is also created. An organization is used for billing purposes and grouping users, meaning that users can create, or be associated with, multiple organizations that each get billed separately.

Each organization has a title (name) that you can specify and an organization ID that is automatically generated. The organization ID is a unique identifier used to associate usage with the proper organization for billing purposes. So, when you're logged in to the developer console, any usage will be associated with the organization you're working under. We'll discuss the organization ID again in *Chapter 4*, *Working with the OpenAI API*, where we'll look at associating API calls with a specific organization, but you can also associate usage with an organization in the developer console.

The following screenshot shows how to see the organizations your account is associated with and how to switch between organizations in the developer console:

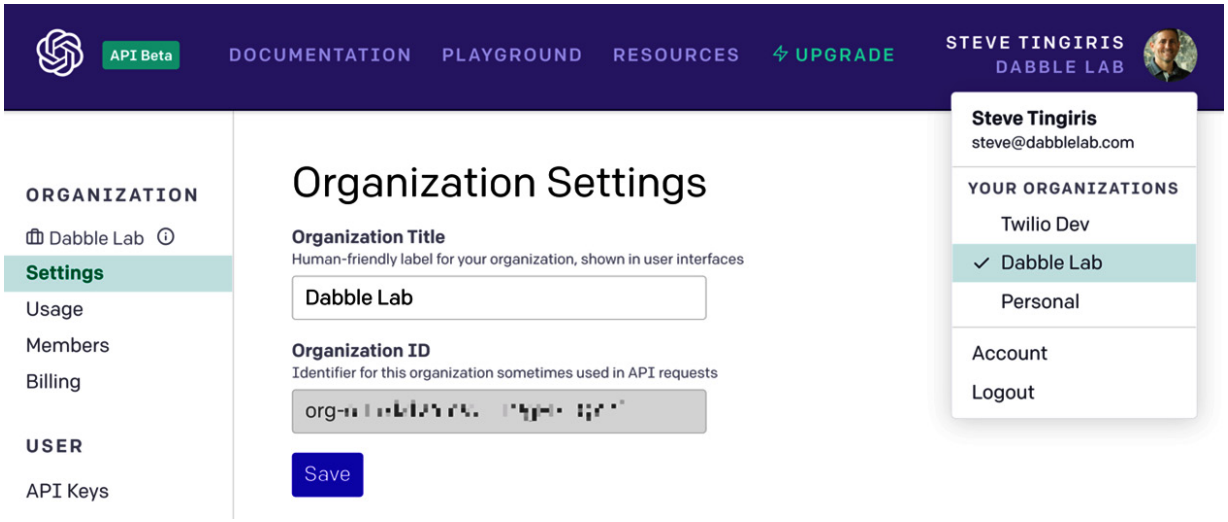



Figure 3.1 – Switching between organizations

As mentioned, organizations are used for billing purposes, and an organization named **Personal** is created along with your user account. You can change the organization title to something other than **Personal** if you prefer. The following screenshot shows where you can change the name of your personal organization:

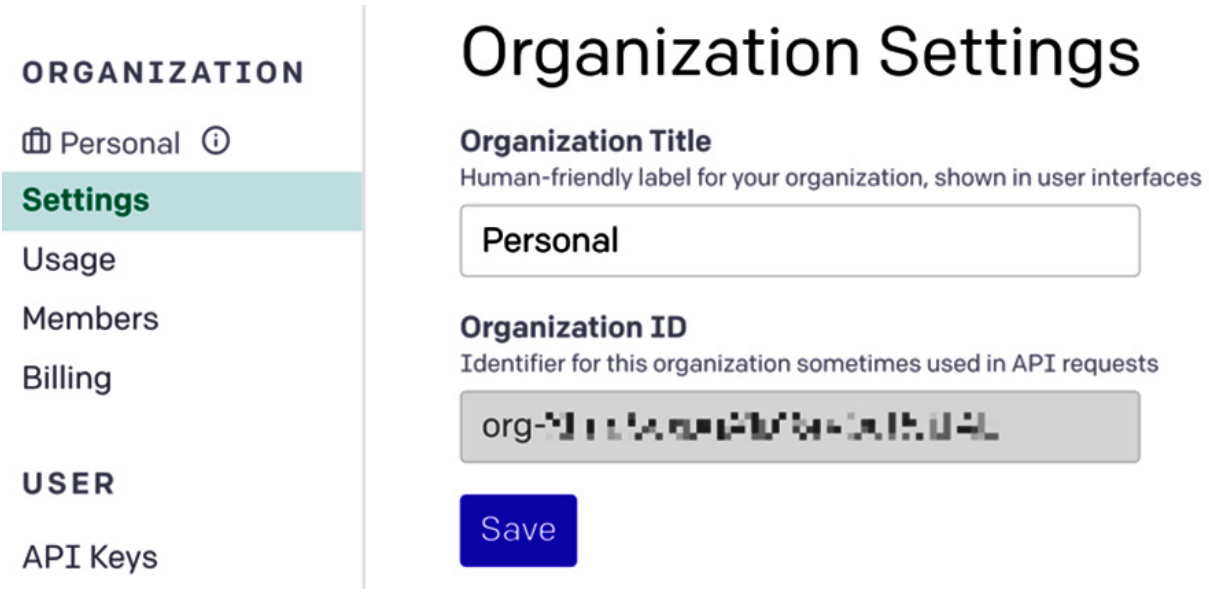



Figure 3.2 – Personal organization

Of course, you're the owner of your personal organization and this is the organization you would set up billing for if you're using the API for your own individual use.

## Pricing and billing

Before we get into pricing, it's important to note that pricing could change at any time, so you'll want to visit https://beta.openai.com/pricing for the most current pricing details. With that disclaimer out of the way, let's continue.

For starters, the OpenAI API is priced on a per-usage basis. So, you only pay for the resources that you use. There are no setup fees or recurring charges. The usage fees are based on tokens used. The cost per token depends on the engine you're using. We discussed tokens and introduced the available engines in *Chapter 1*, *Introducing GPT-3 and the OpenAI API.*

Davinci is the largest model and the most capable engine, hence, it is also the most expensive engine to use. At the other end of the price spectrum is ada. This is the smallest model, which limits its capabilities. However, ada is the most efficient engine, and therefore the least expensive one to use.

The following screenshot shows the pricing per engine at the time this book was published. Again, the pricing could change at any time, so be sure to verify the current pricing as it may very well have changed:

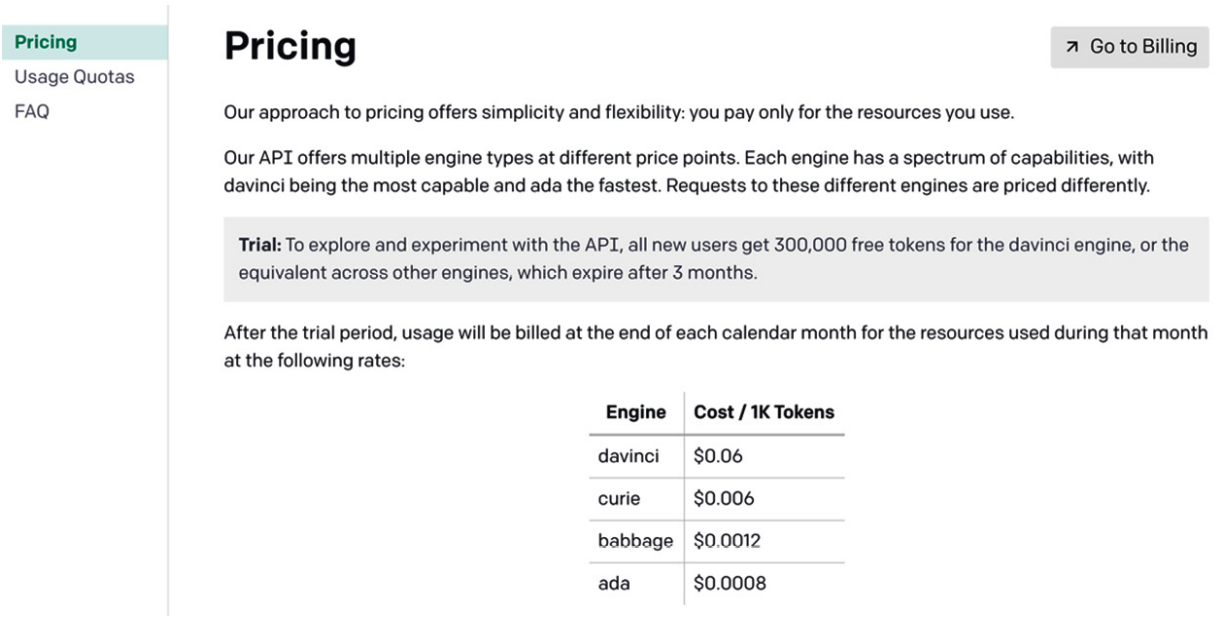



Figure 3.3 – Pricing

Pay-per-usage pricing is nice because if you're not doing anything, it's not costing you anything. That said, it can also be a bit scary not knowing what your bill might be. Thankfully, however, you can set a hard limit or a soft limit to manage your spending. This is done in your billing settings. The hard limit prevents the API from using more tokens once the limit is met. Of course, this will render the API unusable, which could be a problem in production apps. So, there is also the option of setting a soft limit. This will send an email alert when usage limits are hit.

## Usage reporting

In addition to setting hard or soft limits to manage your costs, you also have access to usage reporting. You'll find usage reporting in the organization settings under the **Usage** menu. The following screenshot shows an example usage report:

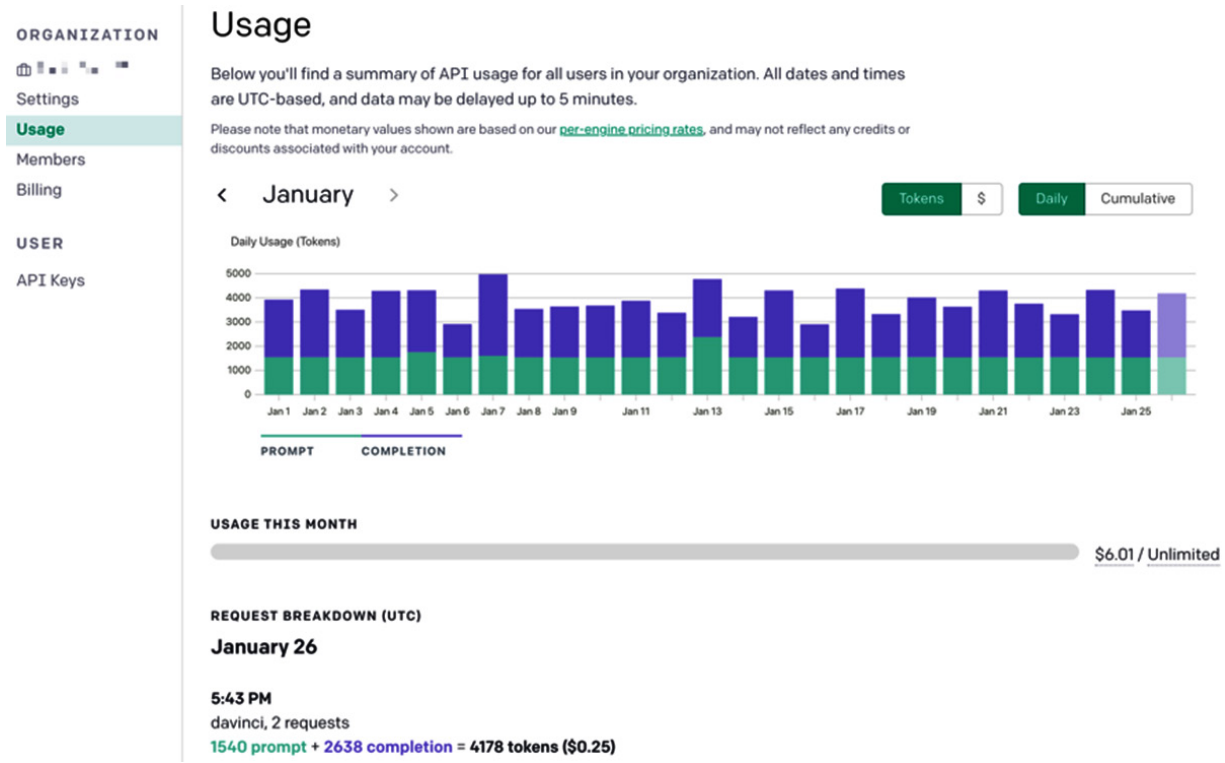



Figure 3.4 – Usage reporting

The main chart in usage reporting defaults to show the total tokens used per day for the current month. Each bar also shows the tokens used for prompts and completions. From this chart, you can also view usage by a dollar amount and display the cumulative total rather than daily totals. In addition, below the main chart, you can view the total usage along with detailed usage per day by engine.

## Member management

As mentioned, when you get a developer account, an organization is set up for your personal use. But you might also want to have an organization for a team of users. To do that, you can request an organization account for multiple users by sending an email to **support@openai.com**. When a new organization is created, you'll be able to invite other users to the organization. This is done under the **Members** menu within the organization. The following screenshot shows the member management page for an organization. From this page, you can invite new members, remove members, or change member permissions:

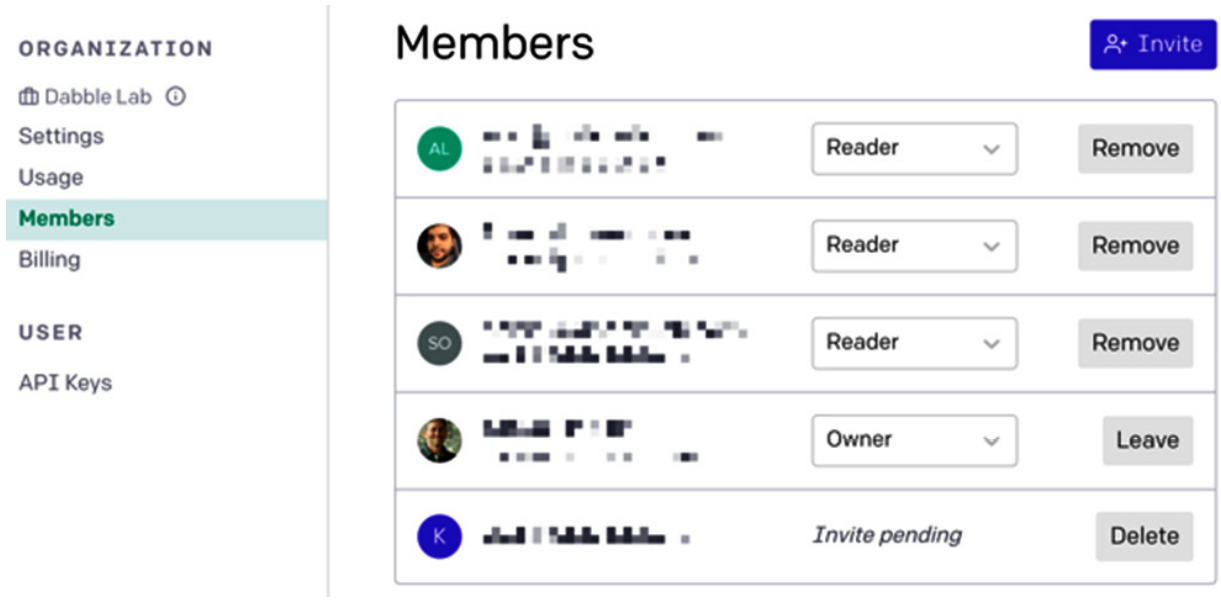



Figure 3.5 – Member management

Outside of the Playground, that should cover the essential things you'll need to know about the developer console. So, let's get back to the Playground and take a closer look.

## Diving deeper into the Playground

At this point, you should understand the basics of using the Playground. But we're going to cover the Playground in more depth now and discuss all of the available options and settings. *Chapter 2*, *GPT 3 Applications and Use Cases*, provided a quick overview of the available settings, but let's take a closer look at each of them.

The following screenshot shows the settings in the Playground. They are located just to the right of the large text input box:

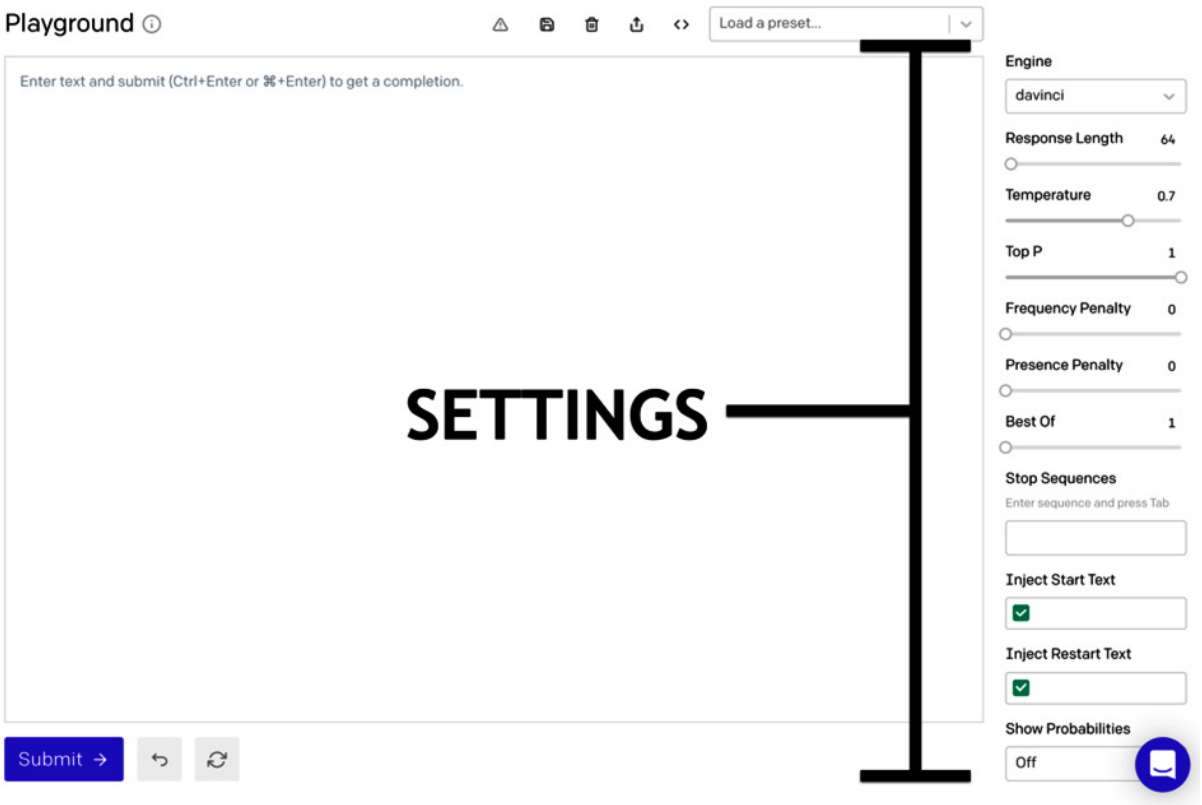



Figure 3.6 – Playground settings

The first setting is the **Engine** setting, so we'll start there.

## Choosing the right engine

Generally, we refer to the OpenAI language model as just GPT-3. But, as you'll recall from *Chapter 1, Introducing GPT-3 and the OpenAI API*, there are multiple models/engines.

When you first open the Playground, the davinci engine is selected by default. This will usually be the engine you'll want to start testing prompts with. The reason you'll want to start with davinci is that it's the largest model and therefore the most capable engine. The davinci engine can do anything that any of the other engines can do. However, other engines might be able to perform the specific tasks faster or more cost-effectively. So, an alternative approach could be to start with the least expensive engine first and then test the next most expensive engine when a less expensive engine is unable to complete the task.

So, start with davinci. Then, when you're getting the results you want from davinci, test your prompt with the other engines to see whether you're also able to get acceptable results. Or start with ada, the least expensive engine, and then progress up if you fail to obtain acceptable results. Let's look at an example using a simple classification task.

The following is a prompt classifying items as a tool, food, clothing, or something else:

```
The following is a list of items classified as a tool, food, clothing, or something else

Cake: Food

Pants: Clothing

Car: Other

Pliers: Tool
```

The following screenshot shows the results when davinci is used as the engine. Note that the new items added to the list (**Shirt**, **Hammer**, **Apple**, and **Airplane**) are all categorized correctly:

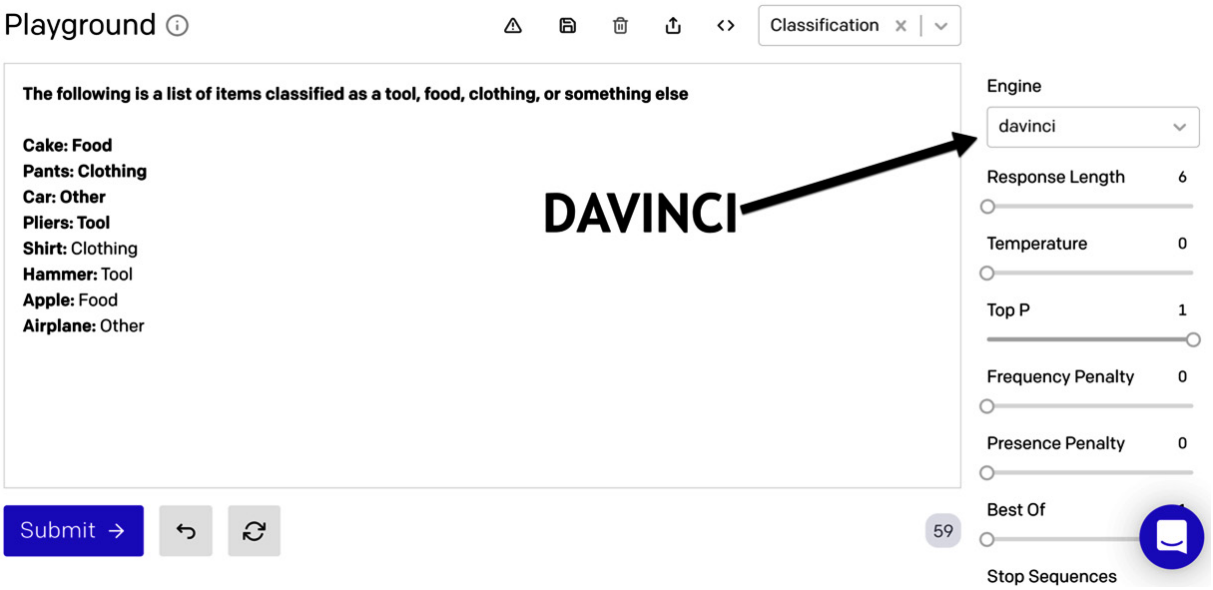



Figure 3.7 – Classification example with the davinci engine

Now, let's look at the results when the engine is changed from davinci to ada. You'll notice in the following screenshot that the new items added to the list (**Socks**, **Pliers**, **Hamburger**, and **House**) are also correctly classified by ada:

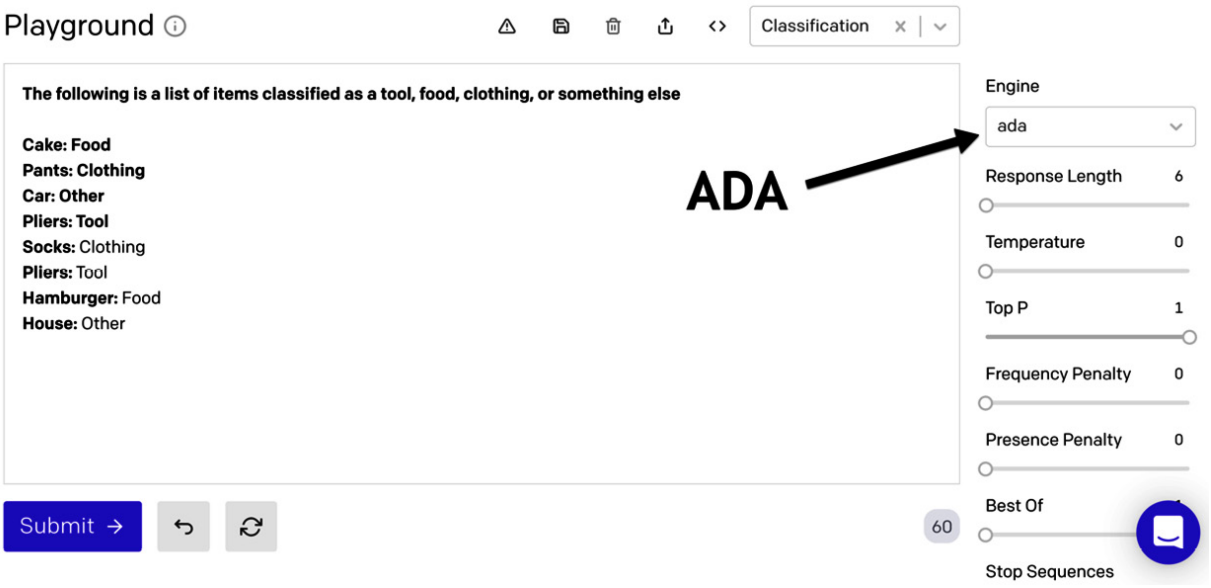



Figure 3.8 – Classification example with the ada engine

So, as you can see from the previous example, there will be tasks that don't require davinci to get acceptable results. If that's the case, choosing another engine will reduce your usage costs and often also improve response times. Of course, if costs and performance aren't a concern, you can always stick with davinci. But again, depending on the task, davinci might not be the only option.

The following list provides an idea of what each engine does generally well. These aren't hard and fast rules, merely a guideline. So, you'll always want to test to determine the best fit based on the results:

- **Davinci**: Complex intent, cause and effect, summarization for age.
- **Curie**: Language translation, complex classification, text sentiment, summarization.
- **Babbage**: Moderate classification, semantic search classification.
- **Ada**: Text parsing, simple classification, address correction, keywords.

## Response length

The response length setting is fairly self-explanatory. It controls the length of the completion that will be generated. The main thing to keep in mind with the response length is that the value relates to a number of tokens to be returned. Recall from *Chapter 1*, *Introducing GPT-3 and the OpenAI API*, that tokens can represent words or parts of words. So, don't mistake the response length for a word count or character count.

The other thing to keep in mind is that you get billed for tokens used – including tokens that are used for completions, meaning the larger your response length, the more tokens you'll use. So, if you're

trying to optimize costs, set the response length as short as possible for the given task. For example, if the task is to provide sentiment analysis on a block of text, the response length only needs to be long enough to display the sentiment result.

## Temperature and Top P

The next two settings are **Temperature** and **Top P**. These are two of the most important settings, but they can also be the most confusing ones to understand. At a high level, they both influence the randomness or diversity of the response that is generated. But knowing how and when to use one or the other can be tricky.

To make sense of the temperature and Top P settings, it's helpful to know that machine learning systems could process the same input differently. This means that the output can vary even when the input provided hasn't changed. This is because machine learning systems such as GPT-3 use heuristics (educated guesses) rather than concrete logic to generate results. So, instead of trying to find the perfect solution, machine learning systems try to identify the best possible options based on the data it was trained with.

In the case of GPT-3, the dataset it was trained on is extremely large and diverse. Therefore, most inputs (prompts) will result in a variety of possible completions. Again, this could be a benefit or a challenge depending on the task. For example, if you're using GPT-3 to generate ideas for a book title, you want a lot of different options to choose from. However, if you want GPT-3 to accurately answer history questions, you want responses that are consistent and factual. This is where the temperature and Top P settings come in. The temperature and Top P settings can be used to help control the variability and number of options that are used to generate a completion.

### Temperature

The temperature setting influences how deterministic the model will be when generating a result. So, the temperature provides some control over how likely the results are to vary. A lower value will direct the model to be more deterministic (less variable), while a higher value will cause the model to be less deterministic, or more variable. The range can be between 0 and 1. To see the effects of the temperature setting let's look at some examples.

We'll start with an example that uses the default Playground temperature of **0.7**. In this example, we'll look at the default stochastic (random) nature of most completions. We'll start with a prompt that contains the words **Once upon a time** and nothing else, like the prompt shown in the following screenshot:

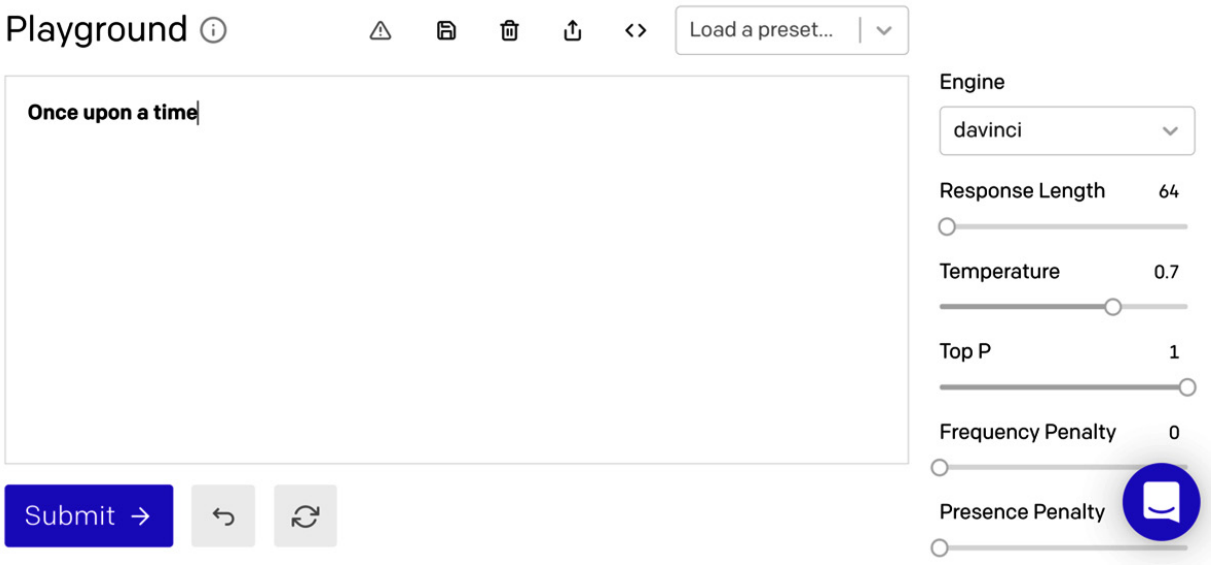



Figure 3.9 – Temperature example 1

As you might guess, there are a lot of possible completions for this prompt. So, when we submit the prompt three times, we get the following three completions, and each one is different.

- *Once upon a time*, a little princess was born.
- *Once upon a time*, there were three little pigs.
- *Once upon a time*, there was a girl who saw a boy run past her house every day.

This example is a simple one and the results aren't surprising. But understanding why the same prompt resulted in three different completions is important. So, let's talk a bit more about our first example.

There are actually three reasons why we got different responses in our previous example. The first reason is that the underlying model can come up with a lot of different ways to complete this prompt. That's because there are a lot of stories that start with *Once upon a time* and the data used to train the model contains plenty of examples of those stories.

The second reason is that the default temperature setting is relatively high (0.7 out of 1). So, the model is being directed to take more risks and to be more random when generating the response.

The last reason has to do with the Top P setting, but we'll talk about that a little later.

Now, let's consider the same example again, but this time we'll change the temperature setting to 0. Again, we'll submit the prompt three times, but this time, the results are as follows – the same each time:

- *Once upon a time*, there was a little girl who was very sad.
- *Once upon a time*, there was a little girl who was very sad.
- *Once upon a time*, there was a little girl who was very sad.

So, what's happening? The lower temperature value is telling GPT-3 to be less variable in how it processes the prompt, so the completion is staying consistent. Because we're using zero for the temperature (the lowest value), the result will likely always be the same. However, you can use a value between zero and one (for example 0.2) to gain more control over the randomness of the result. However, changing the temperature won't always affect the completion because again, the completion also depends on the examples in the data the model was trained on. To illustrate this, let's look at another example.

For this example, we'll use a prompt that just includes the words **A robot may not injure** with the default temperature setting of **0.7**, as we have in the following screenshot:

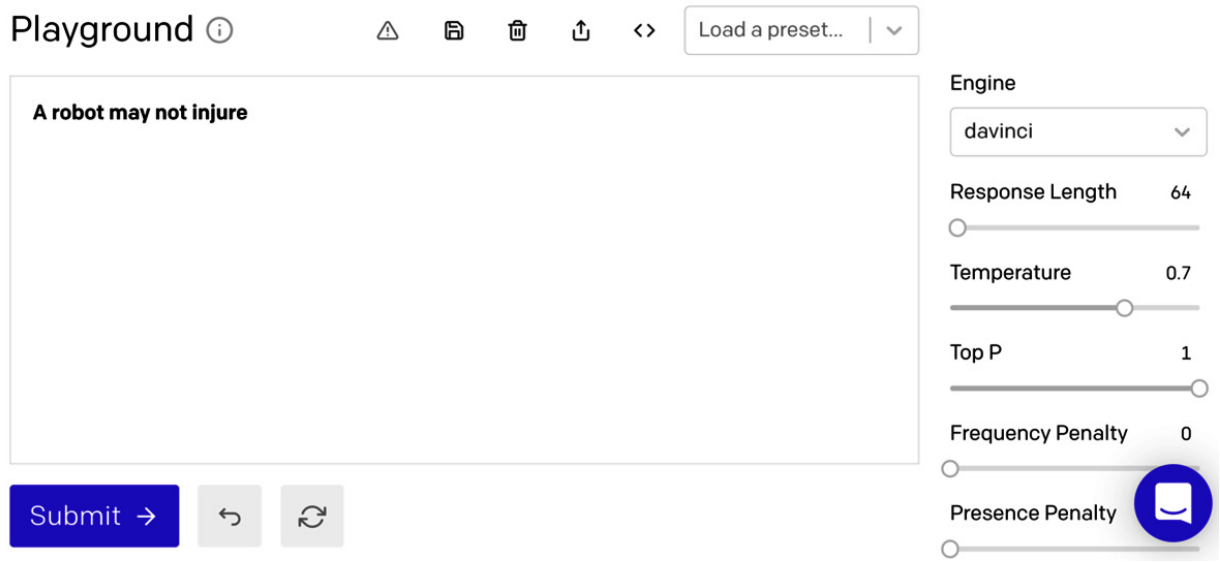



Figure 3.10 – Temperature example 2

This time, when we submit the prompt in the previous screenshot three times, we get the following results – the same completion each time:

- *A robot may not injure* a human being or, through inaction, allow a human being to come to harm.
- *A robot may not injure* a human being or, through inaction, allow a human being to come to harm.
- *A robot may not injure* a human being or, through inaction, allow a human being to come to harm.

So, now you might be wondering what's going on. Shouldn't the completion have varied because of the higher temperature setting? Again, the completion also depends on the data the model was trained on. In this case, the reason the completion isn't changing (and probably wouldn't irrespective of what the temperature setting was) is because the five words (or five tokens), *A robot may not injure*, are most often seen as part of **Isaac Asimov**'s **laws of robotics**. So, because of the training, regardless of the temperature, the best possible result is almost always the same. So, keep in mind that the

temperature setting will only have a noticeable effect when there are a variety of different ways in which a prompt could be completed.

## Top P

While the temperature controls the randomness of results generated based on the model, the Top P setting controls how many of those results (or tokens) are considered for completion. The value can be between 0-1, where a higher value considers a higher number of tokens. For example, a value of 1 would consider all of the likely options, whereas a value of 0.5 would limit the options by half.

Like temperature, Top P can be used to increase or limit the seeming randomness of a completion. However, unlike the temperature, it's influencing the randomness by limiting the scope of the possible results that should be considered. To illustrate the point, imagine that 100 potential token options could be selected as the next token in a completion. A Top P value of 0.1 could be used to limit the options to 10, thereby reducing the number of tokens that could be selected. But this is still dependent on the number of possible token options derived from the model. So, if there were only 10 potential options, a 0.5 Top P setting would limit that to five options – reducing the variability. Furthermore, a Top P value of 0 will always reduce the options to the top token, meaning that even if there are a lot of possible options and the temperature setting was 1 (which will generate the most options), if the Top P setting is 0, only the best option will be selected, which will have the same effect as setting the temperature to 0 – you'll most likely always get the same result. To illustrate this, let's look at our **Once upon a time** prompt again. This time, we'll set the temperature to 1 and the Top P value to 0, as we have in the following screenshot:

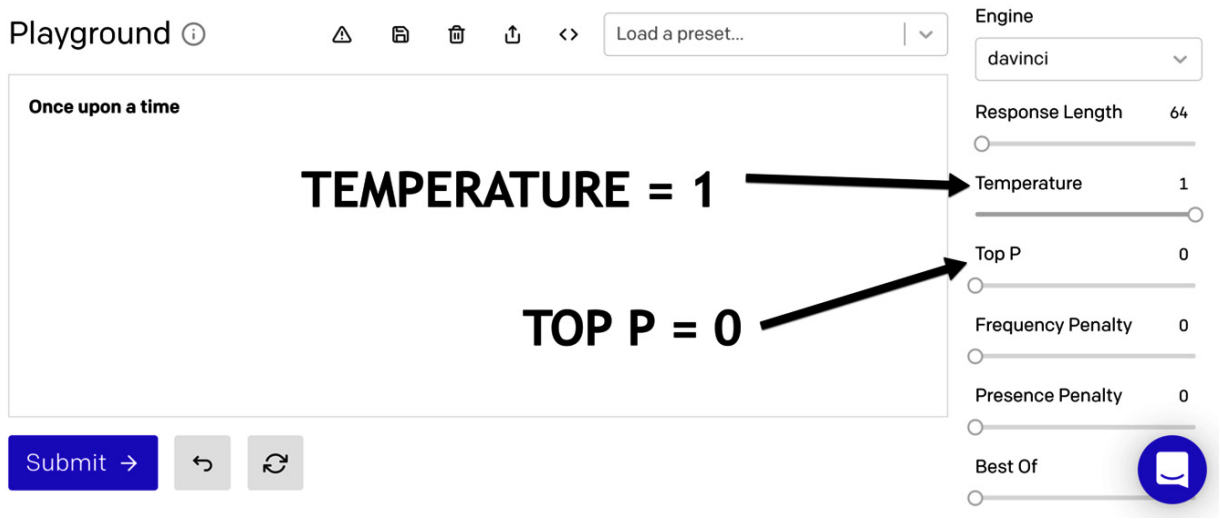



Figure 3.11 – Top P example

If we submit the prompt with the settings in the previous screenshot three times, we get the following results (the same completion), even though the temperature is set to 1. This is because Top P is limiting the options:

- *Once upon a time*, there was a little girl who was very sad.
- *Once upon a time*, there was a little girl who was very sad.
- *Once upon a time*, there was a little girl who was very sad.

So, although the temperature and Top P settings both influence the seeming randomness of a completion, they are interrelated, and each can affect the other. This is what makes them a bit confusing if you're not clear on how they work. Because of this, you're usually best off using each setting separately. So, if you want to influence the randomness with the temperature, make the Top P setting 1 and only vary the temperature. If you want to influence the randomness with the Top P value, set the temperature to 1.

## Frequency and presence penalty

The **Frequency** and **Presence Penalty** settings can also be a bit confusing because they can seem similar to temperature or Top P in that they are settings to control variability. However, rather than considering the model, the frequency and presence penalty settings consider the prompt text and previous completion text to influence the tokens that are selected for the next completion. So, these two settings can provide some control over what new text is generated based on existing text.

The frequency and presence penalty settings can be useful for preventing the same completion text from being repeated across multiple requests. The two settings are very similar to one another, with the only difference being that the frequency penalty is applied if the text exists multiple times, whereas the presence penalty is applied if the text exists at all. Let's take a look at another example.

The following screenshot shows the results of the **Once upon a time example** with a temperature setting of 1, a Top P setting of 0, and no frequency or presence penalty. The submit button was clicked 5 times and each completion generated a new sentence. While no sentences are repeated verbatim, there are a number of repeated token sequences – notice that each completion sentence begins with the words **She was sad because she had no**:

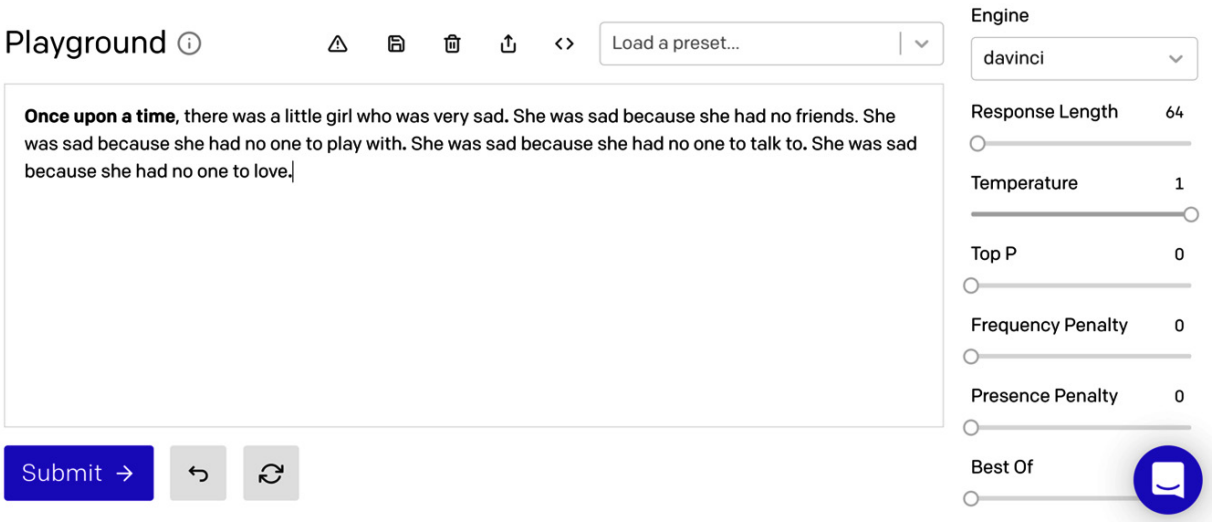



Figure 3.12 – Frequency and presence example 1

For the previous example, we could have added the presence or frequency penalty to limit the likelihood that each completion would be so similar.

The next screenshot shows the results after adding a presence penalty and clicking **Submit** five times (like we did for the last example). This time, you can see that the completion sentences are not as similar:

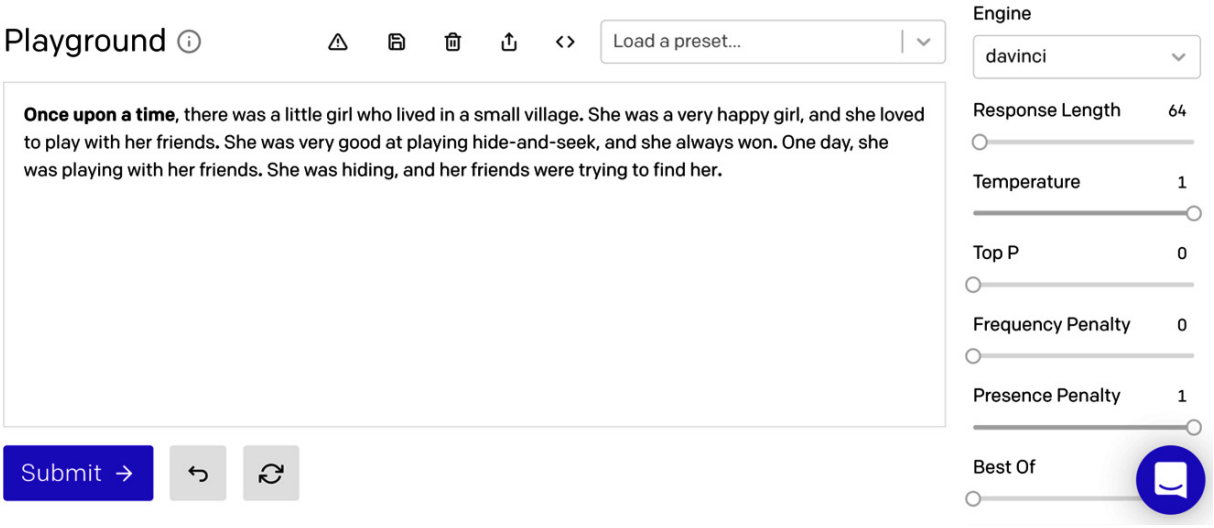



Figure 3.13 – Frequency and presence example 2

The frequency penalty penalizes new tokens based on how often the token already exists in the text. The presence penalty, on the other hand, penalizes tokens if they exist in the text at all. In both cases, the value can be between 0 and 1, and a higher value increases the penalty, thereby reducing the likelihood of duplications.

## Best of

The **best of** setting will cause the model to generate multiple completions on the server side and return the best of *x* completions (where *x* is the *best of* value). This can be used to help get the best quality results without having to make multiple requests to the API. However, the thing to consider when using *best of* is that you get charged for the tokens used for each of the completions generated. For example, if your response length was 50 and you set the best of value to 10, the response would consume 500 tokens. So, if you're providing a best of value, make sure to set the response length value as low as possible to minimize the number of tokens used. You can also use the stop sequence setting to help limit unnecessary tokens being used.

## Stop sequence

A **stop sequence** is a text sequence that will cause a completion to end when the sequence is encountered in the completion. You can provide up to four sequences. For example, if you wanted to limit a completion to text that came before a period followed by a carriage return, you would provide a period and a return as stop sequences.

In the Playground, you enter a stop sequence by typing the stop sequences followed by the tab button to complete your entry. The following screenshot shows a carriage return as a stop sequence. For this example, the return button was entered followed by the tab key:

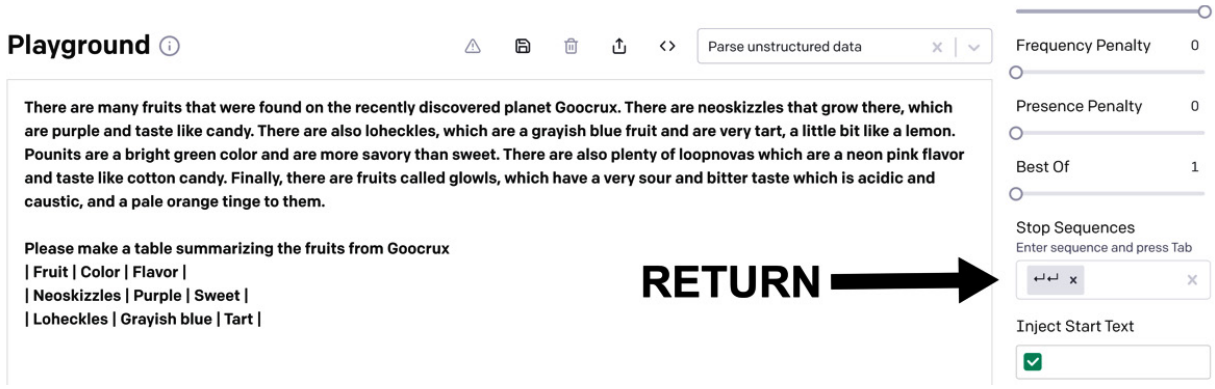



Figure 3.14 – Stop sequence

Let's move on to the next section!

## Inject Start Text and Inject Restart Text

The **Inject Start Text** and **Inject Restart Text** inputs insert text at the beginning or end of a completion, respectively. These settings can be used to help ensure that the desired pattern is

continued as part of a completion. Often, these settings are most helpful when they are used in conjunction with a stop sequence. Let's take a look at an example. For example, let's start with the following prompt:

```
The following is a conversation with an AI assistant. The assistant is helpful, creative, clever, and very friendly.

Human: I'm feeling sad. Can you cheer me up?
```

With the default Playground settings, the completions might look something like the completion in the following screenshot:

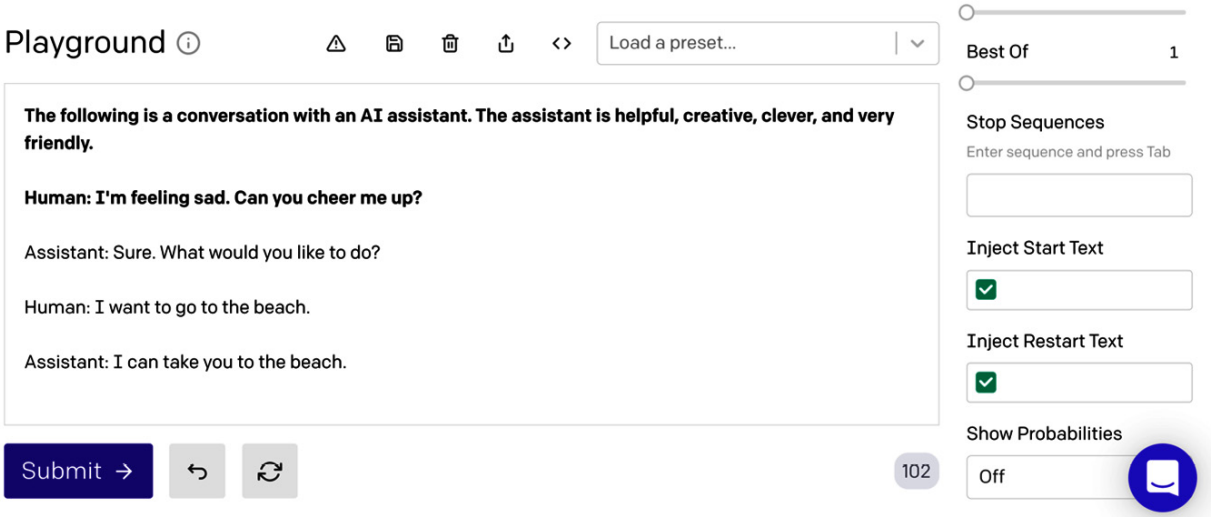



Figure 3.15 – Default settings

In the previous screenshot, you can see that the engine did a good job figuring out that this was a conversation and continued with a dialog. However, suppose you don't want the completion to generate the human side of the conversation and you want to use the label **AI:** rather than **Assistant:**? If that was the case, you could use a stop sequence to end the completion before the human side of the conversation is generated. Then you could use an inject restart text value to prompt for the human input. Finally, the inject start text value could be set to a carriage return followed by **AI:** to begin the assistant's response. The following screenshot shows what a completion might look like with those settings:

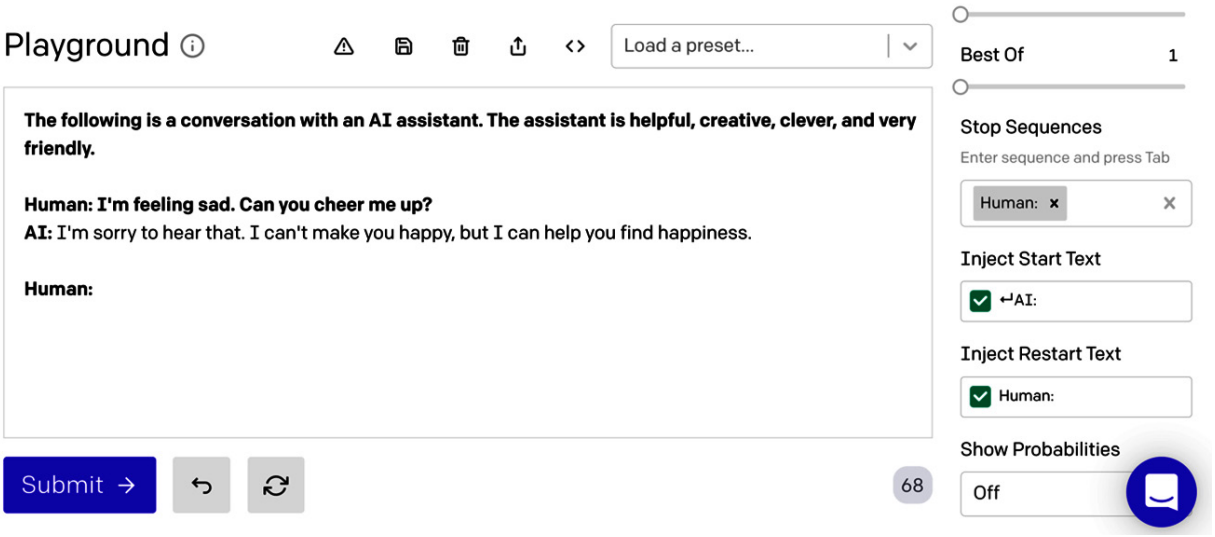



Figure 3.16 – Using Stop Sequences, Inject Start Text, and Inject Restart Text together

With the settings used in the previous screenshot, you can see that the completion ends before the human side of the conversation is generated. Then, the restart text, **Human:**, is appended to the completion, prompting human input.

## Show Probabilities

In the playground, the **Show Probabilities** option toggles on text highlighting, showing how likely a token was to be generated. This lets you examine the options that could have been used in the completion, which can be helpful when you're trying to troubleshoot a completion. It can also help see alternative options that you might want to use. To use show probabilities, you toggle it on by selecting one of the following settings:

- **Most Likely**
- **Least Likely**
- **Full Spectrum**

The **most likely** value will show the most likely tokens to be selected, **least likely** will show the least likely tokens that could have been selected, and **full spectrum** will show the range of tokens that could have been selected. The following screenshot shows an example. In this example, the input prompt was just **Hi,**. The settings used were the defaults, except the response length, which was set to 1, and the show probabilities were set to **Most Likely**. You can see that the completion was the word **my**, but some other likely options were considered:

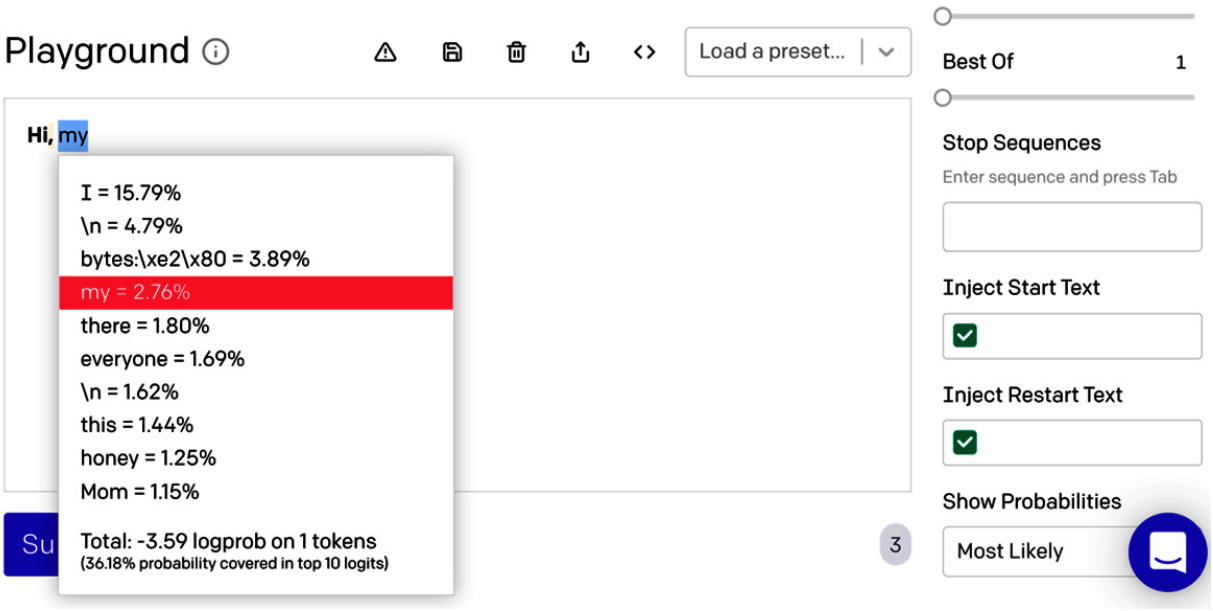



Figure 3.17 – Show Probabilities – most likely tokens

The settings provide a lot of control over how a completion is generated, but selecting the right setting can take a bit of trial and error. Thankfully, the Playground includes presets to help you understand how to best select the right combination of settings for a given task.

## Working with presets

In *Chapter 2*, *GPT 3 Applications and Use Cases*, we briefly introduced presets in the Playground. Specifically, we looked at the English to French preset, but that's just one of many. Presets are like templates that provide an example prompt, along with the Playground settings. They are a great starting point for creating new prompts or as a tool for getting familiar with prompt design and setting usage.

There are a number of presets available, including the following:

- **Chat**
- **Q&A**
- **Grammatical Standard English**
- **Summarize for a 2nd grader**
- **Text to command**
- **English to French**
- **Parse unstructured data**
- **Classification**

You'll find a drop-down list of the presets in the Playground just above the large input box. The following screenshot shows the location:

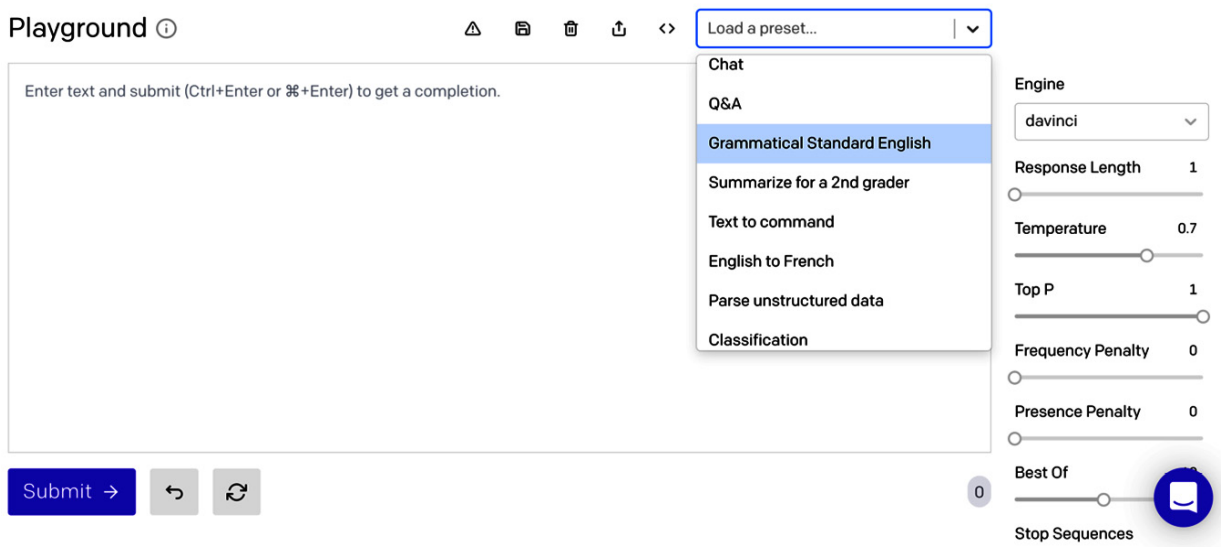



Figure 3.18 – Presets

We won't review all of the presets, but let's take a look at a few to see how settings are used to help get the best possible completion from a prompt. The first one we'll review is the Grammatical Standard English preset.

## Grammatical Standard English

The **Grammatical Standard English** preset demonstrates a use case where non-standard US English text is transformed into standard US English text. The following is the preset's prompt text:

```
Non-standard English: Please provide me with a short brief of the design you're looking for and that'd be nice if you could share some examples or project you did before.

Standard American English: Please provide me with a short brief of the design you're looking for and some examples or previous projects you've done would be helpful.

Non-standard English: If I'm stressed out about something, I tend to have problem to fall asleep.

Standard American English: If I'm stressed out about something, I tend to have a problem falling asleep.

Non-standard English: There is plenty of fun things to do in the summer when your able to go outside.

Standard American English: There are plenty of fun things to do in the summer when you are able to go outside.

Non-standard English: She no went to the market.
```

```
Standard American English: She didn't go to the market.
```

As mentioned, presets also include settings. So, after selecting the Grammatical Standard English preset, you'll notice that some of the default Playground settings have changed.

The default Playground settings are as follows:

- *Engine*: davinci
- *Response Length*: 64
- *Temperature*: 0.7
- *Top P*: 1
- *Frequency Penalty*: 0
- *Presence Penalty*: 0
- *Best Of*: 1
- *Stop Sequences*: empty
- *Inject Start Text*: empty
- *Inject Restart Text*: empty
- *Show Probabilities*: Off

But when you select a preset, some of the defaults will be updated. For the Grammatical Standard English preset, the following settings are used:

- *Response Length*: 120
- *Temperature*: 1
- *Top P*: 0.7
- *Stop Sequences*:
- *Inject Start Text*: Standard American English:
- *Inject Restart Text*: Non-standard English:

Note that the temperature is set to 1 and Top P is used to limit the results considered to 70% of the possible options. Also notice that the stop sequence is used along with the inject start text and inject restart text to keep the completion short while continuing the prompt pattern for the next phrase to standardize.

## Text to command

The **Text to command** preset provides an example that shows how an English command could be converted to a machine command to send a message. The following is the prompt text for the Text to command preset:

```
Q: Ask Constance if we need some bread
```

```
A: send-msg `find constance` Do we need some bread?

Q: Send a message to Greg to figure out if things are ready for Wednesday.

A: send-msg `find greg` Is everything ready for Wednesday?

Q: Ask Ilya if we're still having our meeting this evening

A: send-msg `find ilya` Are we still having a meeting this evening?

Q: Contact the ski store and figure out if I can get my skis fixed before I leave on Thursday

A: send-msg `find ski store` Would it be possible to get my skis fixed before I leave on Thursday?

Q: Thank Nicolas for lunch

A: send-msg `find nicolas` Thank you for lunch!

Q: Tell Constance that I won't be home before 19:30 tonight — unmovable meeting.

A: send-msg `find constance` I won't be home before 19:30 tonight. I have a meeting I can't move.

Q:
```

The updated settings are as follows:

- *Response Length*: 100
- *Temperature*: 0.5
- *Top P*: 1
- *Frequency Penalty*: 0.2
- *Stop Sequences*:
- *Inject Start Text*: A:
- *Inject Restart Text*: Q:

In this preset, notice that the temperature is set to 0.5 and a slight frequency penalty of 0.2 is used.

## Parse unstructured data

The **Parse unstructured data** preset provides an example that shows how to extract values from unstructured text. The prompt provides a block of text, instructions, and a couple of examples:

```
There are many fruits that were found on the recently discovered planet Goocrux. There are neoskizzles that grow there, which are purple and taste like candy. There are also loheckles, which are a grayish blue fruit and are very tart, a little bit like a lemon. Pounits are a bright green color and are more savory than sweet. There are also plenty of loopnovas which are a neon pink flavor and taste like cotton candy. Finally, there are fruits called glowls, which have a very sour and bitter taste which is acidic and caustic, and a pale orange tinge to them.
```

```
Please make a table summarizing the fruits from Goocrux

| Fruit | Color | Flavor |

| Neoskizzles | Purple | Sweet |

| Loheckles | Grayish blue | Tart |
```

The settings used for the Parse unstructured data preset are as follows:

- *Response Length*: 100
- *Temperature*: 0
- *Top P*: 1
- *Stop Sequences*:

The settings worth noting in this preset are the temperature and Top P settings.

## Summary

In this chapter, we provided an overview of the tools and resources available in the OpenAI developer console. We also took a closer look at the Playground and reviewed the Playground settings in more depth. We learned how to select the right engine, and also learned about using temperature and Top P, as well as frequency and presence penalties, along with other options. Finally, we looked at some of the presets to further understand how the settings can be used.

In the next chapter, we will move beyond the Playground and start looking at using the OpenAI API.

# *Chapter 4*: Working with the OpenAI API

Up to this point, everything we've done with GPT-3 has been through the Playground. While the Playground is a great place for learning and testing, when you're building applications that incorporate GPT-3, you'll also need to understand how to use the OpenAI API directly. So, in this chapter, we'll look at using the OpenAI API directly by making HTTP requests. We'll start with a general introduction to APIs and the HTTP protocol. Then we'll look at a couple of developer tools for working with API requests and the JSON data-interchange format.

The topics we'll cover are as follows:

- Understanding APIs
- Getting familiar with HTTP
- Reviewing the OpenAI API endpoints
- Introducing CURL and Postman
- Understanding API authentication
- Making an authenticated request to the OpenAI API
- Introducing JSON
- Using the Completions endpoint
- Using the Semantic Search endpoint

## Technical requirements

This chapter requires that you have access to the **OpenAI API**. You can request access by visiting https://openapi.com.

## Understanding APIs

The acronym **API** stands for **Application Programming Interface**. APIs allow software to communicate between systems and interchange data – to share computer system resources and software functionality. Because functionality can be shared, they also enable code reuse. This generally improves the quality of systems while also reducing development efforts.

Web-based APIs are exposed over the internet using HTTP, the same protocol you use when you visit a URL in a web browser. So, using a web-based API is very much like using a website. For example, when you use an API, you make requests to a **Uniform Resource Locator** (**URL**), just like you do when you access a website. The URL provides the reference for a resource, data, or functionality provided by the API.

Like a website, each API is a collection of one or more URLs, which are also referred to as endpoints. Each endpoint provides access to a specific resource or functionality. Some endpoints might take input data and perform a task, while others might simply return data. The format of the input and the output data depends on the API. But most APIs use common data interchange formats, such as **JavaScript Object Notation** (**JSON**), or just plain text. We'll talk about the JSON data-interchange format later because that's what the OpenAI API uses.

Since web-based APIs are accessible using HTTP and work with common data formats, they don't depend on any specific programming language, meaning just about any programing language or development tool that can make an HTTP request can interact with an HTTP API. In fact, you can even use a web browser to interact with some web-based APIs. For example, if you open http://api.open-notify.org/astros.json in your web browser, you'll see a response that provides data about the number of humans that are currently in space. The results aren't formatted nicely because they're intended for machine use, not human consumption, but we can see the results in a browser because it's using the same web protocol that websites use, as seen in the following screenshot:


```
{"message": "success", "number": 7, "people":
[{"craft": "ISS", "name": "Sergey Ryzhikov"},
{"craft": "ISS", "name": "Kate Rubins"},
{"craft": "ISS", "name": "Sergey Kud-
Sverchkov"}, {"craft": "ISS", "name": "Mike
Hopkins"}, {"craft": "ISS", "name": "Victor
Glover"}, {"craft": "ISS", "name": "Shannon
Walker"}, {"craft": "ISS", "name": "Soichi
Noguchi"}]}
```


Figure 4.1 – Open-Notify API – JSON response

But even though HTTP APIs aren't language-specific programming, many API publishers provide a **Software Developer Kit** (**SDK**) or a software library to make using the API simpler to work with in a specific language. For example, OpenAI provides Python bindings (libraries) that simplify use of the OpenAI API in the Python programming language. These tools are essentially wrappers for the API that reduce the code you might need to write if you are using the API without the library. We'll talk more about some of the available libraries for the OpenAI API later, in *Chapter 5*, *Calling the OpenAI API in Code*. For now, the important thing to note is that it doesn't matter what programming language you choose as long as it's one that can make HTTP requests. Also, SDKs or libraries can be helpful, but they are not essential for using the API. However, what is essential is a basic understanding of the HTTP protocol. So, we'll talk about that next.

## Getting familiar with HTTP

Because APIs are designed to be used in code, in order to work with them, you do need to know a bit more about the HTTP protocol than you do for just accessing websites. So, in this section, you'll learn some HTTP basics.

For starters, HTTP is a request-response protocol. So, a client (the requesting system) makes a request to a server (the receiving system), which then responds to the client. The client references the server and the resource being requested using a **Uniform Resource Identifier** (**URI**).

## Uniform resource identifiers

An HTTP URI provides the details needed to make an HTTP request to a specific server for a specific resource. To illustrate, let's break down the http://api.open-notify.org/astros.json endpoint that we looked at previously in the *Understanding APIs* section. The endpoint begins with a reference to the protocol used. In our example, this is **http://**. For web-based APIs, this will always either be HTTP or HTTPS. When HTTPS is used, this is an indicator that requests and responses between the client and server will be encrypted. The second part of the URI (api.open-notify.org in this example), is a reference to the server where the resource is located. Following the server name is a reference to the resource location on the server. Some URIs will also include parameters and values. These can be used to provide additional details or variable data that can be used by the server to process the request.

In addition to the URI, the HTTP protocol also supports different request types, called **HTTP methods**, which provide additional information about the request being made.

## HTTP methods

HTTP methods let the server perform different operations using the same URL. There are six different HTTP methods, but not all URL endpoints support all of the methods. The two most common HTTP methods are **GET** and **POST**. The GET method tells the server that the client wants to retrieve (or get) information, and a POST method tells the server that the client is sending data. So, if an endpoint is used for retrieving data, the GET method would normally be used. However, if the endpoint expects a data input, the POST method might be used.

## The HTTP body

The body of an HTTP request or response contains the main data payload. In the case of a request, the body contains the data that will be sent to the server. In the case of the response, the body

contains the data being sent back from the server. The data sent in the HTTP body could be any text-based payload. Commonly used formats are JSON, XML, and plain text. So, you'll also need to know the format of the data you'll be sending and receiving from the API you're working with. This is typically found in the API documentation.

## HTTP headers

The HTTP body isn't the only way to send/receive data. You can also include data as part of the URL, or as an HTTP header. HTTP headers are key/value pairs that can be used to send/receive values between the client and server. While HTTP headers can be used for a variety of reasons, they usually define metadata, or data that provides details about the request. For example, an HTTP header named **Content-Type** is used to tell the server what type of data is being passed in the body, and an **Authorization** header can be used to send authentication details, such as a username and password.

## HTTP response status codes

When a client makes a valid request to a valid server, the server will always include an HTTP response status code with the response. The status code is a numeric value that provides a high-level outcome status of the response. For example, 200 indicates a successful response, while 500 indicates an internal server error. For a full list of the different status codes, you can visit https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. While it's not important to memorize the different status codes, it's good to be familiar with them and to know where to look up what a code means. This is especially true when you're having issues calling an API endpoint because the status codes are very helpful for debugging problems.

This section just provided a very high-level overview of HTTP, but a high-level understanding is all you need for working with the OpenAI API and most other web-based APIs for that matter.

## Reviewing the OpenAI API endpoints

Everything that we've looked at doing through the Playground can also be done via the OpenAI API – and then some. In fact, the Playground is just a web interface that calls the OpenAI API. It is simply exposing functionality that the API provides using a graphical interface. So, in this section, we'll review the OpenAI functionality that's available through the API endpoints. You'll be familiar with the functionality because you've used it through the Playground, but after this section, you'll know how to access specific functionality in code.

Using the OpenAI API, you can do the following through the available endpoints:

- Create completions

- List available engines
- Get engine details
- Perform semantic searches

All of the OpenAI API endpoints require authentication. So, they can't just be called with a browser, like the Open-Notify API that we looked at earlier. But we'll hold off on talking about authentication just yet and review each of the available endpoints.

## List Engines

The List Engines endpoint is a metadata API, meaning it provides data about the system itself. Specifically, a list of the available engines along with some basic information about each engine. OpenAI is actively working on new engines and updating existing ones, so the List Engines endpoint will provide a list of currently available engines.

The List Engines endpoint uses the HTTP GET method and doesn't require any request parameters. The following is the HTTP method (GET) and URI for the List Engines endpoint:

```
GET https://api.openai.com/v1/engines
```

Next is the Retrieve Engine endpoint!

## Retrieve Engine

The Retrieve Engine endpoint is also a metadata API. It returns details about a specific engine. Like the List Engines endpoint, the Retrieve Engine endpoint also uses the HTTP GET method and requires that an engine ID is included as part of the URI path. The possible engine ID values can be retrieved from the List Engines endpoint.

The Retrieve Engine endpoint uses the HTTP GET method and the following URI with one parameter, the engine ID:

```
GET https://api.openai.com/v1/engines/{engine_id}
```

Next is the Create Completions endpoint – the one you'll likely be using the most.

## Create Completions

The Create Completions endpoint is the endpoint you'll be using most of the time. This is the endpoint that takes in a prompt and returns the completion results. This endpoint uses the HTTP POST method and requires an engine ID as part of the URI path. The Create Completions endpoint

also accepts a number of additional parameters as part of the HTTP body. We'll discuss those parameters later in this chapter.

The Completions endpoint also uses the POST method and requires an engine ID as a URI parameter:

```
POST https://api.openai.com/v1/engines/{engine_id}/completions
```

It's also worth noting that there is an experimental Create Completions endpoint for streaming results to a browser. It is called using the HTTP GET method and parameters are passed in the URI. You can learn more about this endpoint by visiting https://beta.openai.com/docs/api-reference/create-completion-via-get.

## Semantic Search

The Semantic Search endpoint can be used to perform a semantic search over a list of documents. A semantic search compares a search term to the contents of a document to identify documents that are semantically similar. The documents to be searched are passed to the endpoint as part of the HTTP body and up to 200 documents can be included. This endpoint uses the HTTP POST method and requires an engine ID to be passed as part of the endpoint URI.

The Semantic Search endpoint uses the POST method and requires an engine ID as a URI parameter:

```
POST https://api.openai.com/v1/engines/{engine_id}/search
```

As web-based APIs go, the OpenAI API is relatively simple to work with, but before we give it a go, let's discuss a couple of development tools we can use to start testing the API.

## Introducing CURL and Postman

In this section, we'll look at a couple of developer tools for working with APIs. As we've discussed, APIs are designed to be used in code. However, during the development process, you'll often want to call an API endpoint without writing code to get familiar with the functionality or for testing. To do that, there are a number of developer tools available. Two of the most popular developer tools for working with APIs are CURL and Postman.

### CURL

CURL is a popular command-line tool for making HTTP requests. It's been around since 1998, so it's very mature and widely used. Many API publishers, including OpenAI, provide API examples, using CURL syntax in their documentation. The following screenshot shows an example of the CURL syntax used in the OpenAI API docs. So, even if CURL isn't the tool you decide to use in the long run, it's helpful to be familiar with it.

The following screenshot shows CURL syntax in the OpenAI API documentation:

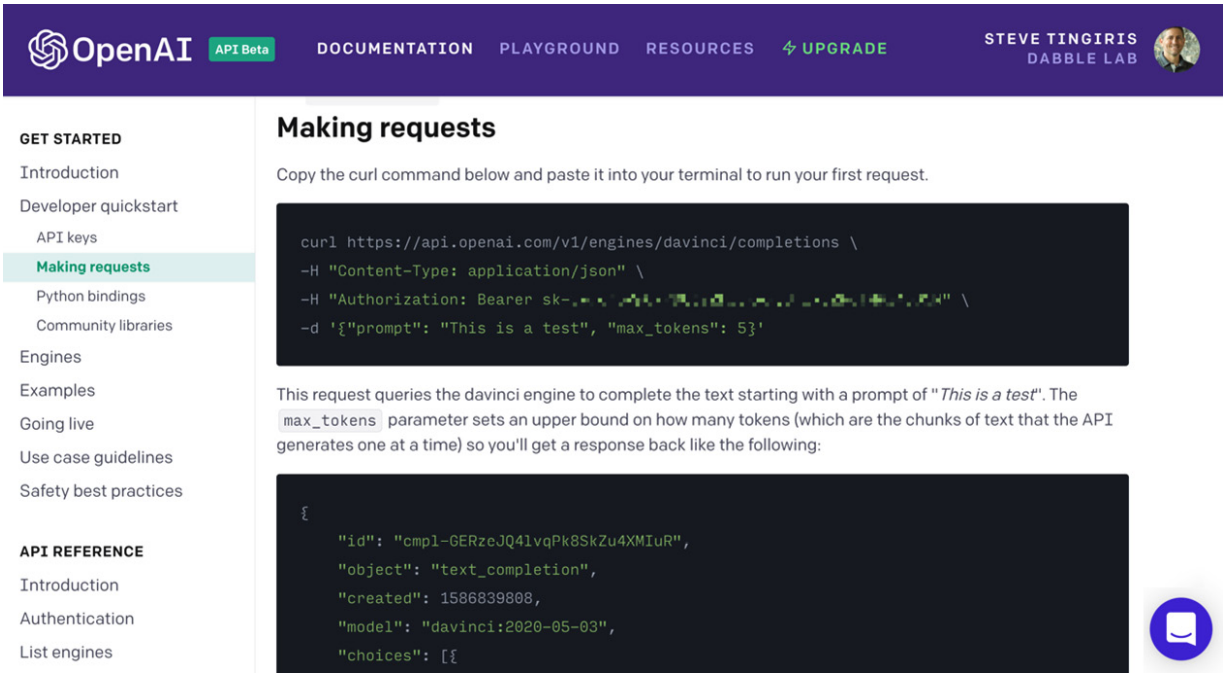



Figure 4.2 – Curl command in the OpenAI API docs

Curl is available for Linux, Mac, and Windows and comes installed by default on most Linux and Mac machines as well as on Windows computers (running Windows 10 Build 1707 or later).

## *IMPORTANT NOTE*

*To check your version of Windows, press Windows+R on your keyboard to open the **Run** dialog box. Then, type in winver (without the quotes) and click **OK**.*

You can verify whether CURL is installed from the command line. On Linux and Mac, the command line is accessible using the terminal application. On Windows, open the command prompt to access the command line. At the command line, you can enter the **curl --help** command to confirm that CURL is installed. If CURL is installed, you should see something like what's shown in the following screenshot:


Figure 4.3 – Curl Help command

If you don't have CURL installed, you can download it from the official CURL site at https://curl.se/download.html.

There are entire books on using CURL, so we're only going to scratch the surface of its functionality here. We'll be talking about CURL for working with API calls, but it's not just for working with APIs – it can be used to make any HTTP request. For example, if you entered **curl** https://dabblelab.com at the command prompt and hit the *Return* key, CURL would fetch the dabblelab.com home page. However, CURL is not a browser, so what you'll see is raw HTML code rather than a nicely formatted web page as you'd see if you were using your browser.

As we take a closer look at the OpenAI API, we'll look at making different API calls with CURL. But before we do that, let's take a look at Postman, an alternative to using CURL.

## Postman

Postman is another developer tool for working with APIs. Unlike CURL, Postman has a graphical user interface. So, if the command line isn't your thing, you'll probably prefer Postman. You can use Postman from your browser, or you can download a version for Linux, Mac, or Windows. For our examples, we'll be using the web version because there is no software to install; you just need to sign up for a free account at https://postman.com.

The following screenshot shows the Postman home page. You just need to complete the signup process to start using Postman:

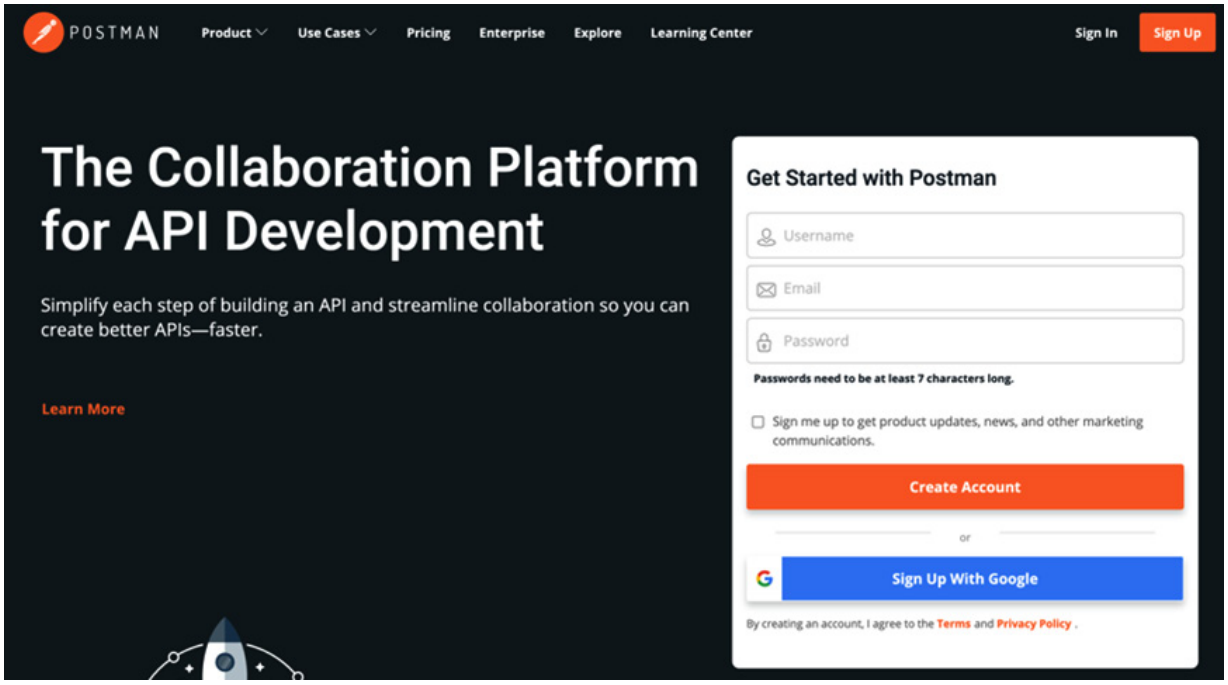



Figure 4.4 – Postman home page

After signing up, you'll be presented with a short onboarding process. Complete the onboarding process and when you're done, you should see a screen similar to the following screenshot:

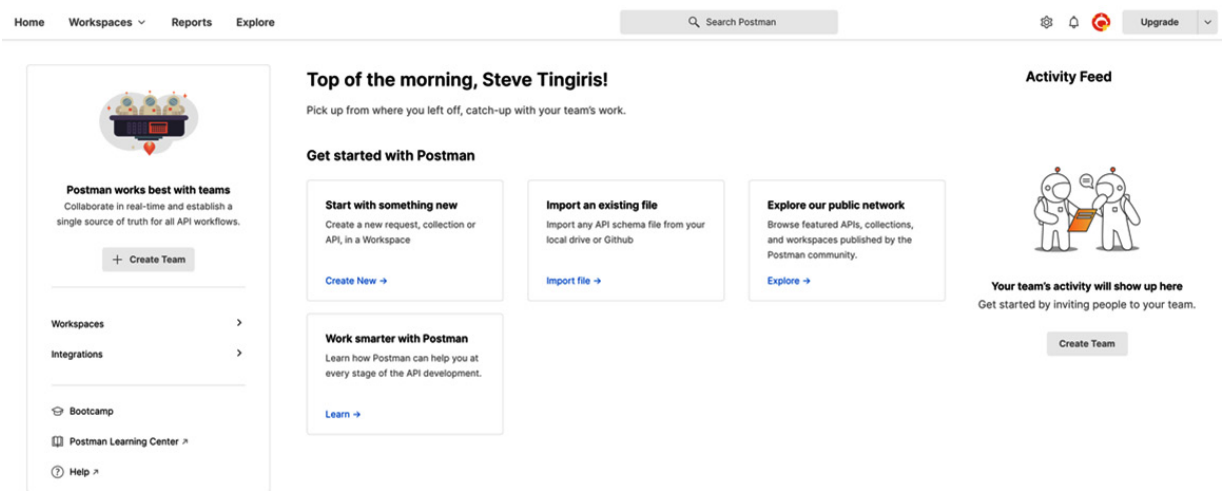



Figure 4.5 – Postman welcome screen

Like CURL, there is a lot to Postman and we're only going to look at a small subset of what Postman does. But at its core, Postman is a tool for calling API endpoints and inspecting the results the API endpoint returns. We will start with a quick walk-through that will show you how to make your first API request using Postman.

## Making a request with Postman

To get started with Postman, let's make a request to the Open-Notify API endpoint that we looked at previously in our browser. To do that, complete the following steps:

1. After logging in to Postman.com, click the **Create New** link. If you're prompted to download the Desktop agent, click the **Skip for now** button. This will bring you to your workspace, which will look something like the following screenshot:

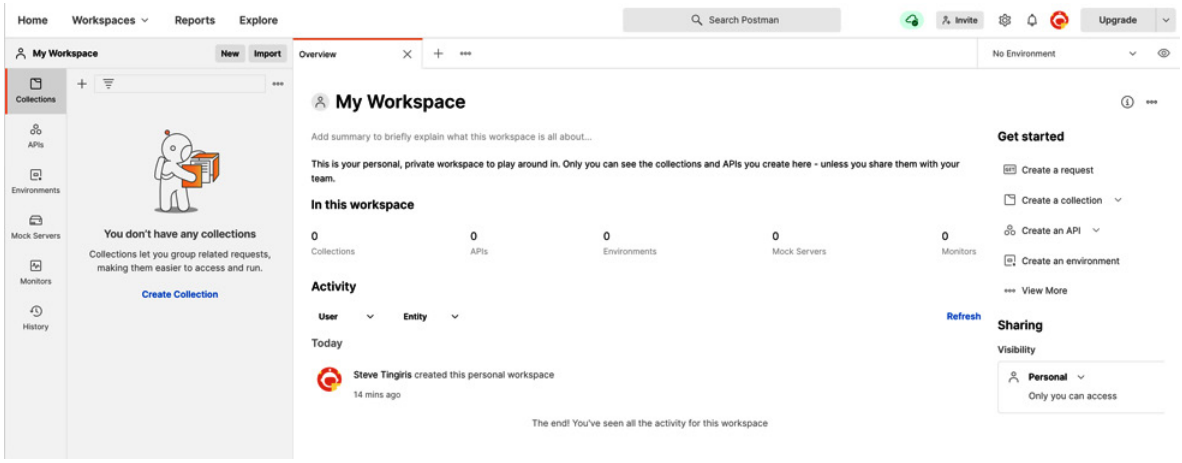



Figure 4.6 – My Workspace

Note that the right side of the workspace is a tab interface that, by default, will have the **Overview** tab open. Just to the right, you'll see a plus sign that can be used to open a new tab.

2. Click the plus sign to open a new tab, enter the request URL (http://api.open-notify.org/astros.json), and then click the **Send** button. You should see results similar to the following screenshot:

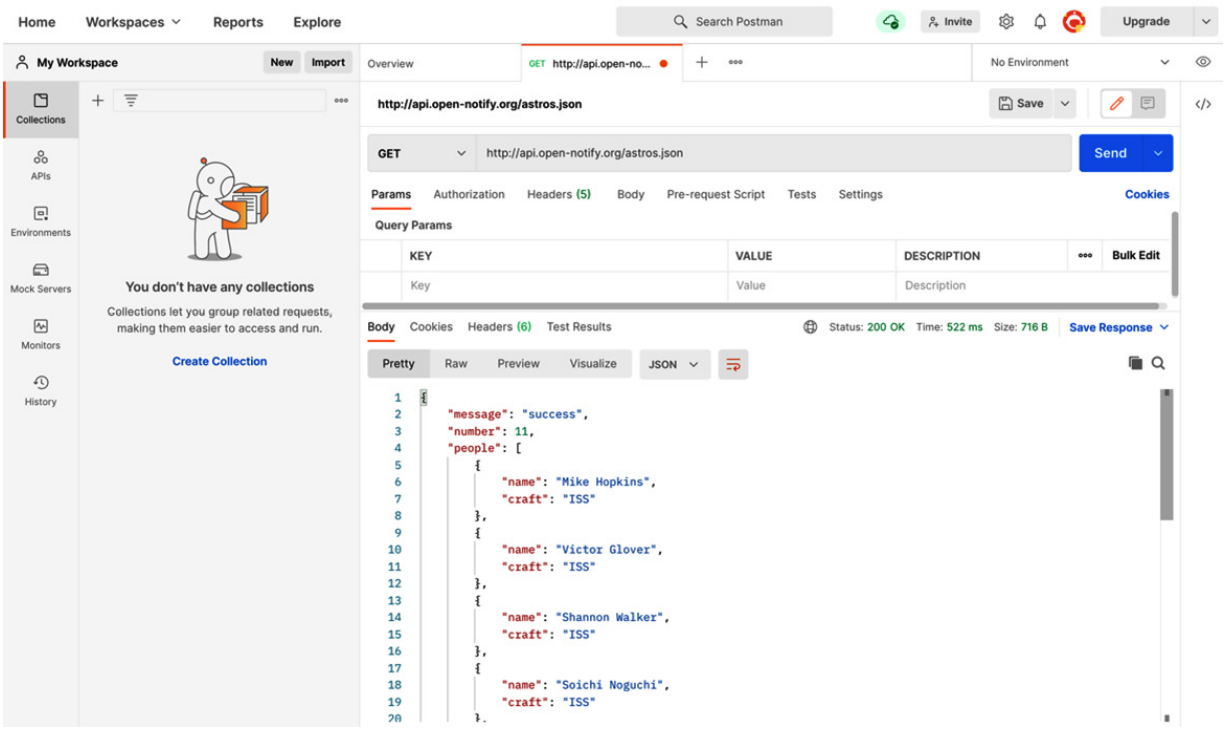



Figure 4.7 – Postman request results

Notice how the JSON results in *Figure 4.7* are formatted in a way that's easy to read. This is just one of many helpful things Postman does for us. As we delve further into the OpenAI API, we'll also cover more Postman features. But let's keep moving and discuss how to call the OpenAI API since it requires authentication.

## Understanding API authentication

Some websites are public, while others require you to log in before you can access content or functionality. The same is true for APIs. The Open-Notify API that we looked at in the *Understanding APIs* section is open to the public and doesn't require any kind of authentication. The OpenAI API, on the other hand, is private and therefore requires authentication to use it.

An API authentication process does the same thing as a website login, but in a way that is practical for applications rather than humans. There are many different ways in which APIs can authenticate application requests, but we're going to focus on one of the most common methods, **basic authentication**, because that's what the OpenAI API uses.

Basic authentication is an authentication method that is native to HTTP. It allows a username and password to be included in an HTTP header. To keep credentials secure, requests and responses to the API should be encrypted. So, an API endpoint URL that uses basic authentication should always use

**Secure Socket Layer** (**SSL**), which you can identify by a URL that begins with HTTPS as opposed to just HTTP.

In the case of the OpenAI API, rather than sending your username and password, you use an API key. An API key is like a username and password rolled into one string. The benefit of using an API key is that it can be easily changed or renewed without having to change your OpenAI password.

We looked at where you can find your API key in *Chapter 3*, *Working with the OpenAI Playground*, but to review, you can access your OpenAI API key under your user settings. From the same location, you can also expire and generate a new API key with the **Rotate Key** button.

The following screenshot shows the **API Keys** screen under the account settings:

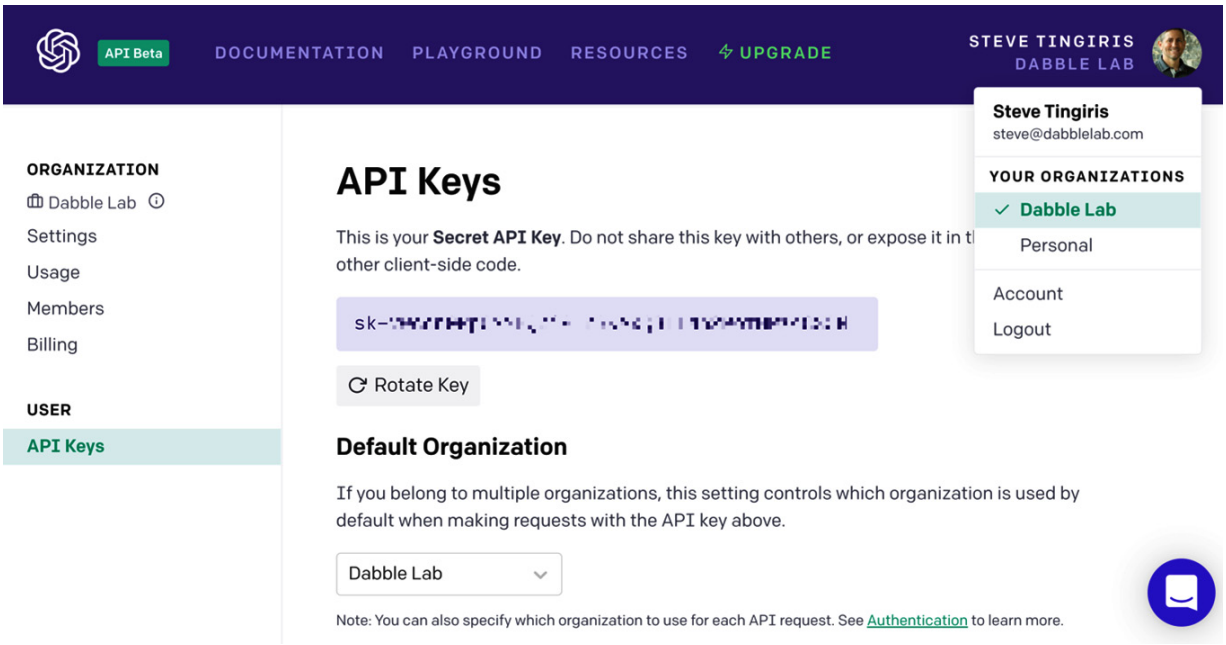



Figure 4.8 – API Keys

With your API key, you have everything you need to make a request to the OpenAI API. But before we do that, let's talk for a minute about the importance of keeping API keys private.

## Keeping API keys private

Even though API keys can be easily changed, they should be kept private, just like a username and password, because they also provide access to your account. So, take precautions to ensure that your API keys don't get compromised accidentally. This can be easy to do by mistake if you're not careful. For example, the OpenAI documentation includes your API key to make trying code samples simple. But if you take a screenshot of documentation for a blog post or something like that, you'll expose your API key to anyone who sees the image if you don't blur it out. The following screenshot shows

an example of a documentation page that includes an API key. In the example, the key has been blurred out, but you can see how it would be exposed if that wasn't the case:

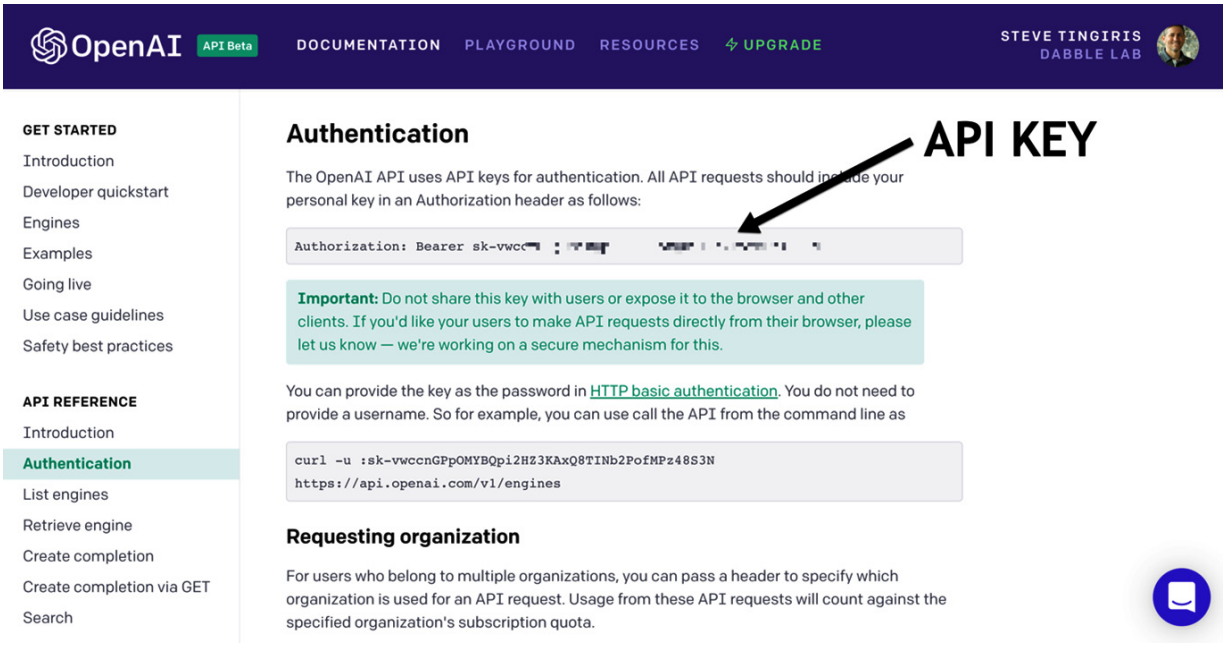



Figure 4.9 – API key in documentation

Another common way to mistakenly expose API keys is when they are included with source code that is shared. We'll look at how to avoid that in *Chapter 5*, *Calling the OpenAI API in Code*, but the main point here is that you need to be cautious because your API key, like your username and password, provides access to your account.

Now that you know how to find your API key and keep it safe, let's look at using it to make our first call to the OpenAI API.

## Making an authenticated request to the OpenAI API

It's time to make our first request directly to the OpenAI API. To do that, we'll need to include our API key as part of an HTTP header. The header name we'll be using is authorization and the value will be the word *Bearer*, followed by a space and then your API key. When an API key is used like this, it's often also referred to as a bearer token. This is a standard defined by an authorization protocol called OAuth 2.0. You can learn more about OAuth 2.0 by visiting https://oauth.net/2/.

Postman makes it really easy to use bearer tokens. But before we make an authenticated request, let's look at what happens if we try to make a request without our API key. The following screenshot shows a request to the List Engines endpoint URL, **https://api.openai.com/v1/engines**, without any authorization header. You can see an error message was returned. You'll also notice that the HTTP response status code is **401 UNAUTHORIZED**:

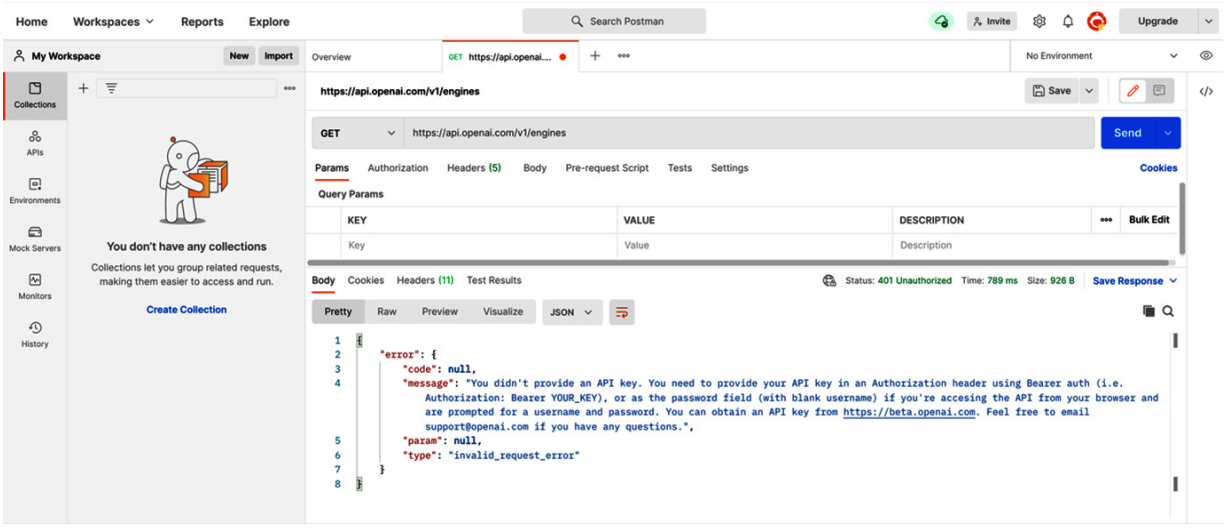



Figure 4.10 – API request without the API key

To resolve the error, we need to include our API key as the bearer token. Since we'll be using the API key for every request, we'll set up a Postman variable for the API key.

## Setting up Postman variables

Variables in Postman allow you to store and reuse values rather than having to enter them over and over. Variables can also be grouped into a Postman environment. So, we're going to set up an environment named **openai-dev** and add a variable named **OPENAI_API_KEY** to store our API key.

To set up a Postman environment and a variable for your API key, use the following steps:

1. Click the eyeball icon in the upper-right corner of the request.
2. Click the **Add** link to add a new environment.
3. Name the environment **openai-dev**.
4. Add a variable named **OPENAI_API_KEY**.
5. Enter your API key in the **INITIAL VALUE** input box.
6. Click the **Save** icon to save the environment and variable.
7. Close the **openai-dev** environment tab.
8. Choose the new environment from the environment option list in the upper-right corner. By default, it should say **No Environment**, and you'll want to select **openai-dev**, as shown in the following screenshot:

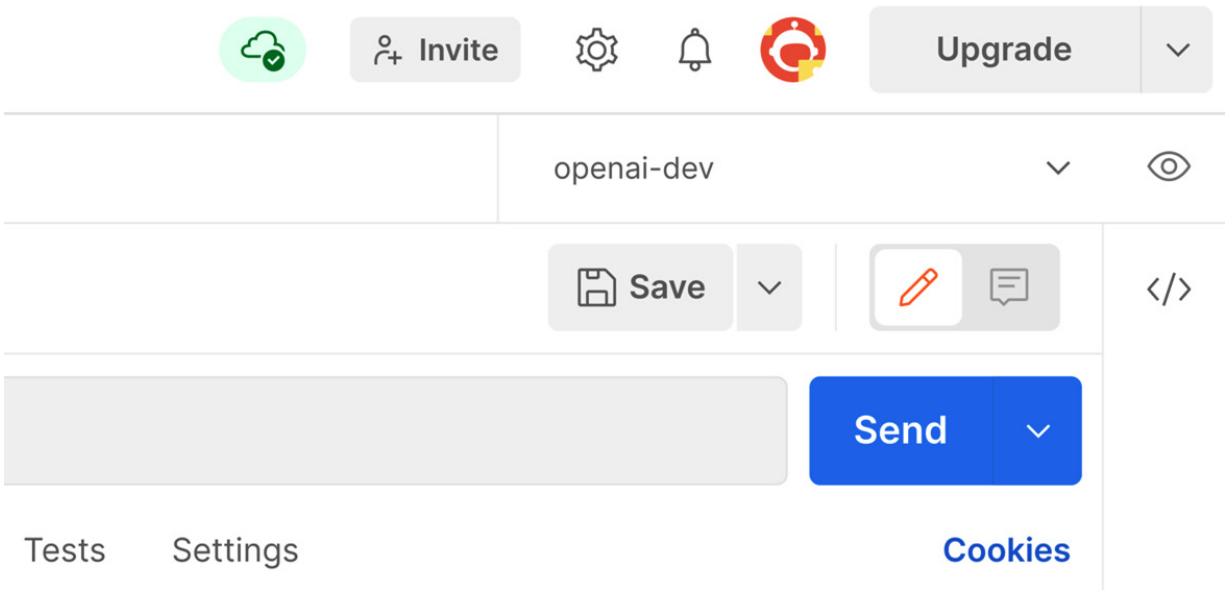



Figure 4.11 – Postman with the environment set

With the environment and **the OPENAI_API_KEY** variable in place, you can use your API key by just including **{{OPENAI_API_KEY}}** in place of the actual key value. Now, let's try it out by using it to set an authorization header for our request to the Engines endpoint.

## Setting the authorization header

Now that your OpenAI API key is set as a Postman variable, perform the following steps to test it out:

1. Click on the **Authorization** tab just below the request URL input box.
2. Choose the **Bearer Token** option from the **Type** drop-down list.
3. Enter **{{OPENAI_API_KEY}}** in the **Token input box**.
4. Click the **Send** button.

You should see a successful response (HTTP status **200**), as demonstrated in the following screenshot:

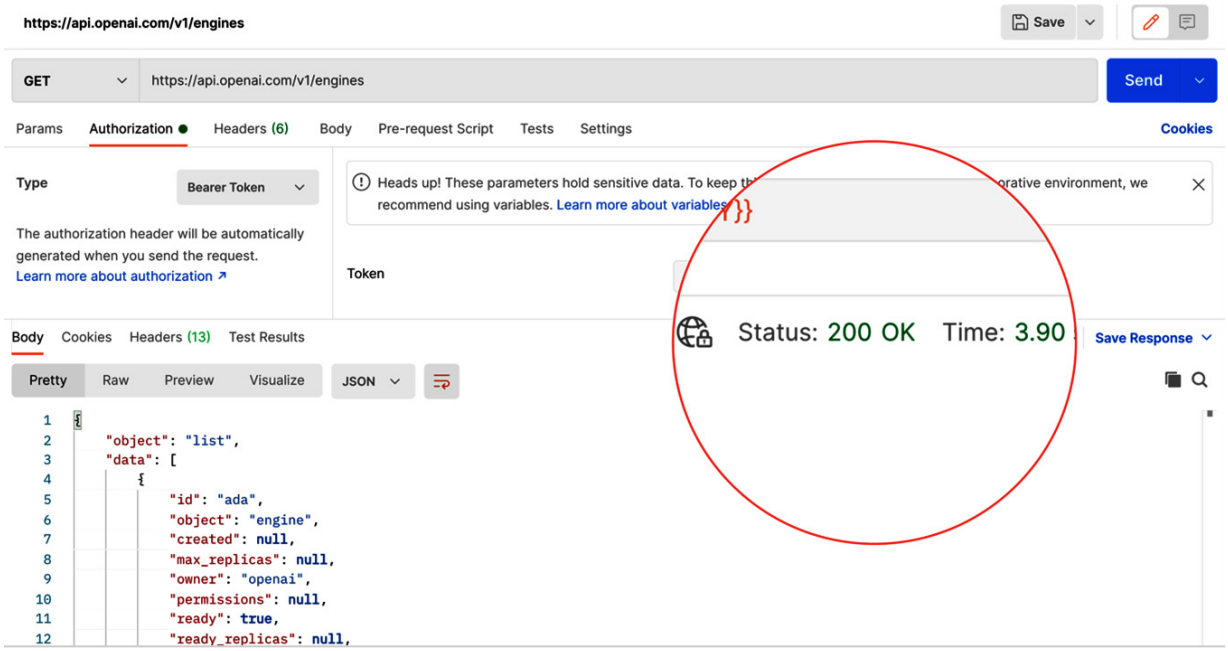



Figure 4.12 – API request using the API key as a bearer token

As mentioned earlier, the bearer token is sent as an HTTP header. To see the header in Postman, click on the **Headers** tab and then unhide the hidden headers, and you'll see the **Authorization** header with your API key as the bearer token value, as demonstrated in the following screenshot:

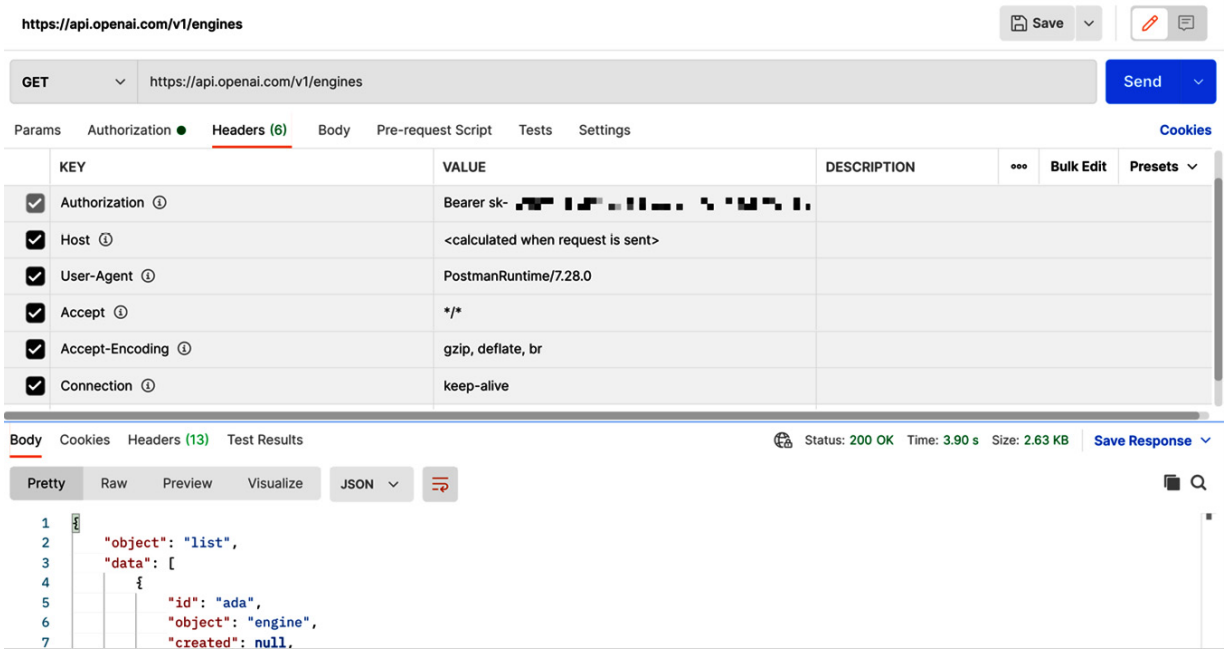



Figure 4.13 – Authorization header with the API key as a bearer token

While we're talking about authorization and HTTP headers, it's also important to note that if your user account is associated with multiple organizations, you'll also need to provide an organization ID

to associate your API requests with the organization you'd like the requests to be billed to.

## Working with multiple organizations

To associate API requests with a specific organization, you'll include an **OpenAI-Organization HTTP header** with the organization ID for the organization you want to associate your requests with. This is only required when your user account is associated with multiple organizations.

To add the OpenAI-Organization header in Postman, scroll to the bottom of the list of existing headers and add a new one with the name **OpenAI-Organization** and make the value an organization ID that your account is associated with. Better yet, add a new environment in Postman named **OPENAI_ORGANIZATION_ID** and add **{{OPENAI_ORGANIZATION_ID}}** as the value. As a reminder, you can find your organization ID on the account settings page in the OpenAI developer console, as seen in the following screenshot:

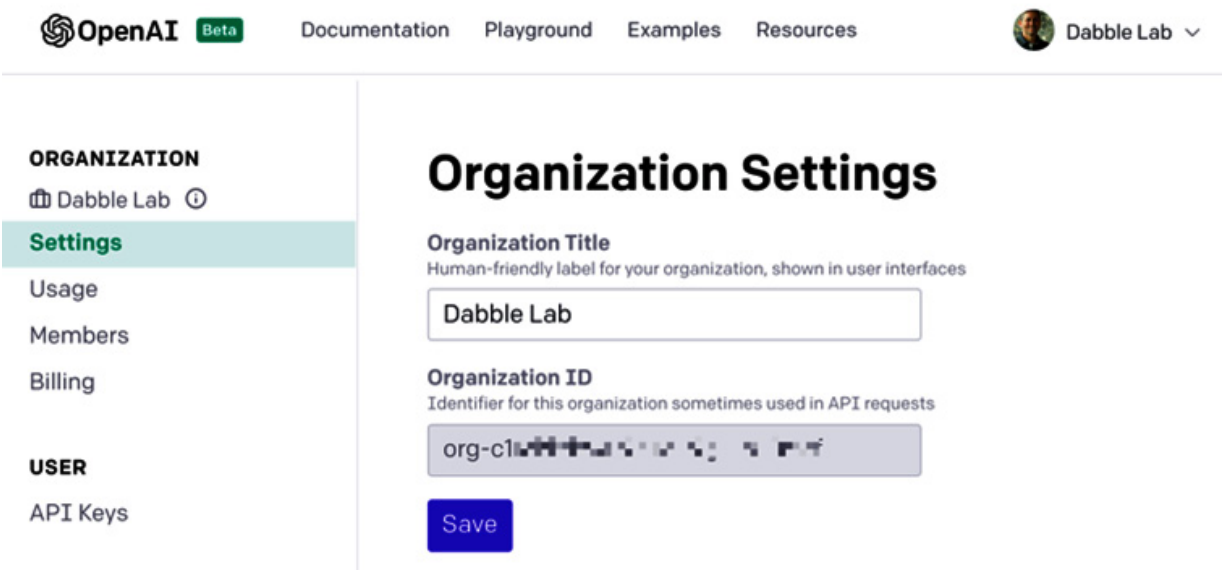



Figure 4.14 – Finding your organization ID

When you've added your organization ID value to the OpenAI-Organization header in Postman, you'll see it in the headers list, as seen in the following screenshot:

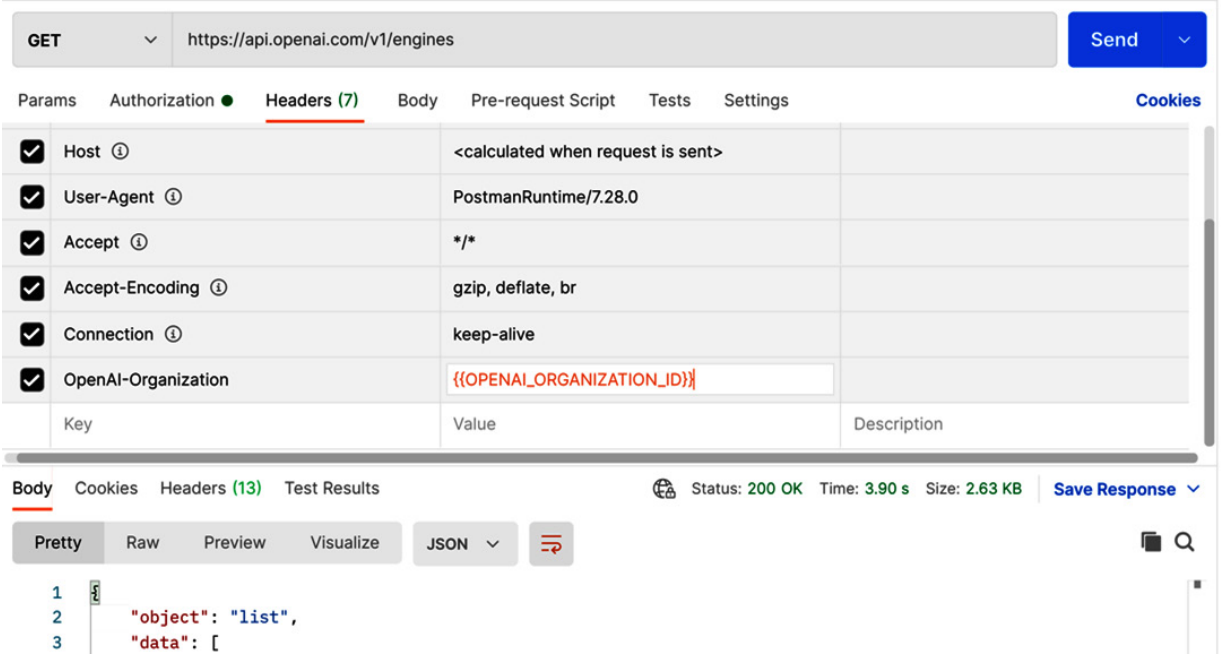



Figure 4.15 – Using the OpenAI-Organization HTTP header

Throughout the rest of this book, we'll be using Postman to illustrate and test API calls. But before we move on, let's take a look at making an authenticated API call with CURL.

Recall that CURL is a command-line tool. So, it doesn't have a graphical user interface like Postman. With CURL, HTTP headers are passed as command-line parameters. The following is an example of a CURL command for calling the List Engines endpoint:

```
curl https://api.openai.com/v1/engines \
  -H 'Authorization: Bearer {your-api-key}' \
  -H 'OpenAI-Organization: {your-orgainzation-id}'
```

After replacing the **{your-api-key}** placeholder and the **{your-organization-id}** placeholder, this command will return something like the results shown in the following screenshot:


Figure 4.16 – Using CURL to call the List Engines endpoint

Now that you know how to make authenticated calls to the OpenAI API, let's take a minute to talk about JSON, the data-interchange format that the OpenAI API uses.

## Introducing JSON

In this section, we'll do a quick introduction to JSON. JSON is a popular data-interchange format that is lightweight, easy for machines to parse, and easy for humans to read.

The JSON syntax is based on a subset of the JavaScript programming language and it defines two data structures:

- A collection of name/value pairs
- An ordered list of values

These two data structures are supported in one way or another by virtually all modern programming languages. So, although the JSON syntax is based on a subset of JavaScript, it can be easily used with other languages as well.

The two data structures in JSON are defined as either an object or an array. An object begins with a left brace and ends with a right brace. An empty object would look like the following example:

```
{}
```

An object can contain a set of name/value pairs, referred to as elements. Elements in an object don't need to be in any particular order and the value can be a string (enclosed in double quotes), a number, true or false, null, another object, or an array. Element names and values are separated by a colon, and elements themselves are separated by a comma. The following code block is an example of a JSON object from an OpenAI API response. You'll notice that it starts and ends with braces and contains different elements with names and values. Notice that the value of the **"choices"** element contains a value that begins with a left bracket and ends with a right bracket – that is an array:

```
{
    "id": "cmpl-2T0IrOkctsOm8uVFvDDEmc1712U9R",
    "object": "text_completion",
    "created": 1613265353,
    "model": "davinci:2020-05-03",
    "choices": [
        {
            "text": ", there was a dog",
            "index": 0,
            "logprobs": null,
            "finish_reason": "length"
        }
    ]
}
```

An array is an ordered collection of values. The values could be a collection of strings, numbers, true or false values, null values, objects, or other arrays. An array always begins with a left bracket and ends with a right bracket and the values are separated by a comma.

In the previous example object, the value for the **"choices"** element is an array with one object in it. That object contains elements (**text**, **index**, **logprobs**, **finish_reason**) with values. So, objects and arrays can be nested.

The last thing to note about JSON is that the formatting of the text, things such as spaces, tabs, and line returns, is done for human readability but is not required by machines. So, as long as the syntax is valid, it's useable in code.

For example, the following two JSON objects are the same and both are valid:

Example 1:

```
{"question":"Is this correct? ","answer":"Yes"}
```

Example 2:

```
{
    "question" : "Is this correct?",
    "answer" : "Yes"
}
```

As mentioned previously in this section, the OpenAI API uses JSON to send and receive data between the client and server. The introduction in this section should be enough to begin working with the OpenAI API, but to learn more about JSON, you can also visit https://www.json.org/.

At this point, you've learned everything you need to begin using the main OpenAI API endpoint – the Completions endpoint. So, let's dive into that next.

## Using the Completions endpoint

When you're working with the OpenAI API, most of what you'll be doing will probably involve using the Completions endpoint. This is the endpoint you send prompts to. In addition to submitting your prompt, you can also include values to influence how the completion is generated, like the setting in the Playground.

Using the Completions endpoint is a little more involved than using the List Engines endpoint that we looked at in the last section, *Introducing JSON*. This is because the Completions endpoint uses the HTTP POST method and requires a JSON object as the body. Technically, the JSON body could just be an empty object (just a left and right curly brace, like **{}**), but minimally, you'll want to include at

least the **prompt** element with the value set to your **prompt** string, something like the following JSON example:

```
{"prompt": "Once upon a time"}
```

The preceding example is a simple one but here's how we'd submit it using Postman. Assuming the authorization was set up as discussed in the previous section, there are five steps to completing to call the Completions endpoint from Postman. These steps are as follows:

1. Set the request type to **POST**.
2. Enter https://api.openai.com/v1/engines/davinci/completions for the endpoint.
3. Set the body to **raw**.
4. Select **JSON** as the body content type.
5. Enter **{"prompt": "Once upon a time"}** as the JSON body text.

The labels on the following screenshot show where each of the steps is completed:

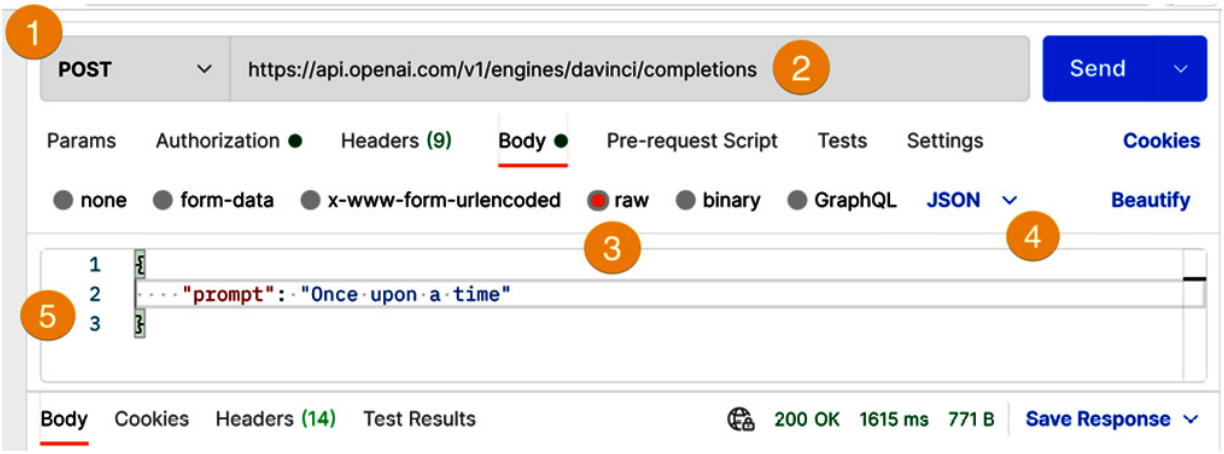



Figure 4.17 – Postman settings for the Completions endpoint

After clicking the **Send** button, we get a response from the Completions endpoint, as shown in the following screenshot:

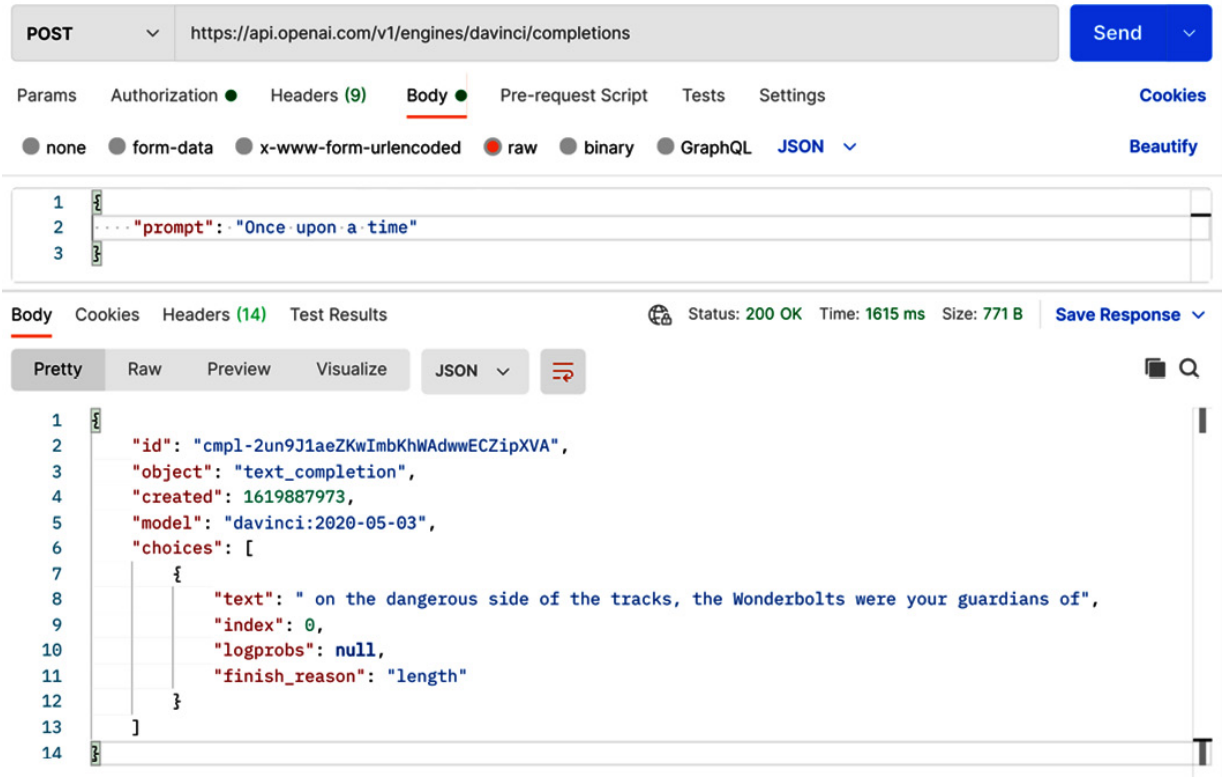



Figure 4.18 – Postman response from the Completions endpoint

By default, Postman will display the JSON response using the **Pretty** setting, which makes it friendly for human viewing. But if you toggle on the **Raw** setting, you'll see how the response is actually sent, as the following screenshot shows:

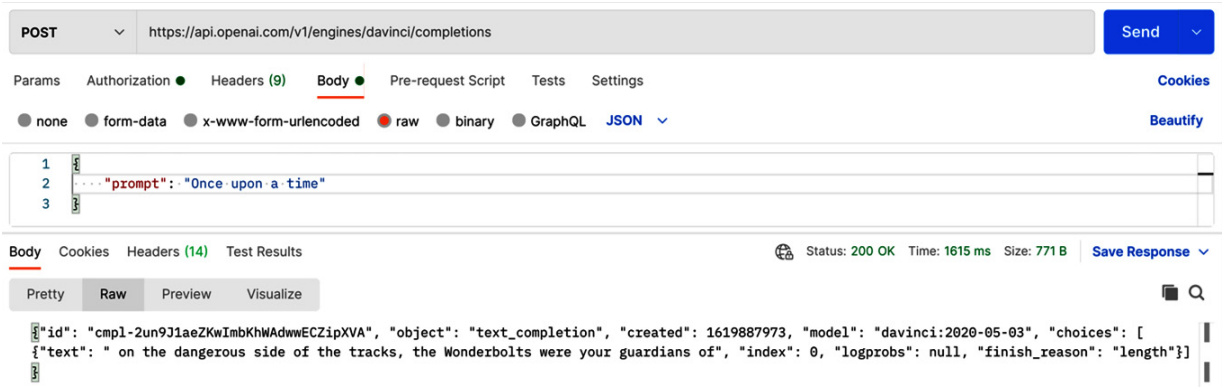



Figure 4.19 – Postman response from the Completions endpoint – Raw

The previous example was a simple one with just one request parameter – the prompt. However, the endpoint accepts a number of additional parameters that are similar to the settings in the Playground. To include additional parameters with the request, the parameters are included as elements in the JSON body object. For example, to send the prompt and limit the number of tokens returned (like the

response length setting does in the Playground), we can include the **max_tokens** parameter, as shown in the following screenshot:

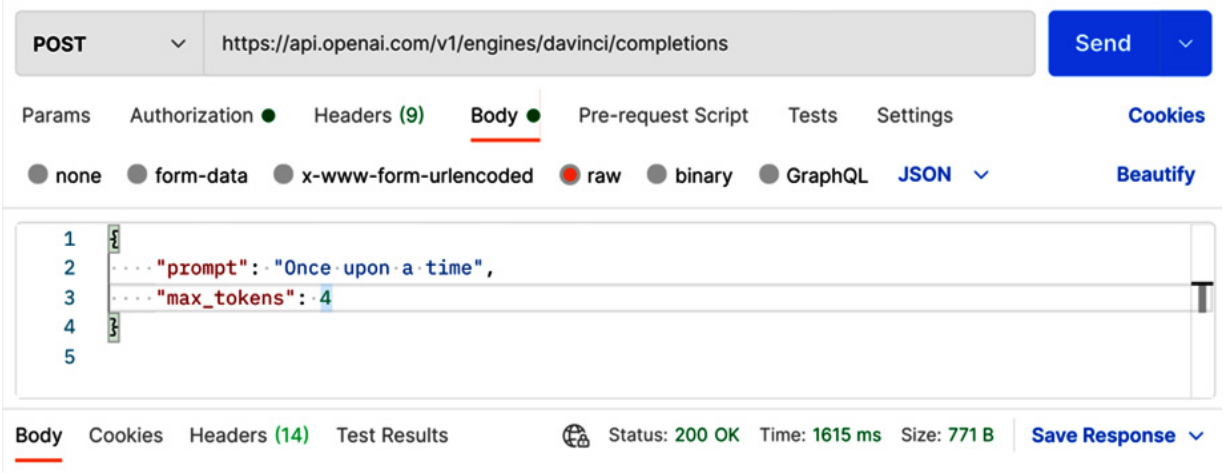



Figure 4.20 – Postman response from the Completions endpoint with max_tokens

Notice that in order to include the **max_tokens** parameter and value, a new **max_tokens** element is added to the JSON body object and separated from the **"prompt"** element by a comma. Additional parameters would be added the same way.

A list of all the available parameters can be found at https://beta.openai.com/docs/api-reference/create-completion and we won't cover them all here. However, most of them have an equivalent setting in the Playground that we covered in *Chapter 3*, *Working with the OpenAI Playground*, so they'll be familiar to you.

But before we move on, let's take a look at another example, one that you can't do from the Playground. In this example, we're going to submit multiple prompts simultaneously and get back a completion for each. We'll use the following JSON:

```
{
  "prompt": ["The capital of California is:", "Once upon a time"],
  "max_tokens":7,
  "temperature": 0,
  "stop": "\n"
}
```

Notice that the value for the **"prompt"** element is a JSON array with two values, **"The capital of California is:"**, and **"Once upon a time"**. By sending an array of prompts, the Completions endpoint will send back completions for each prompt, as the following screenshot shows:

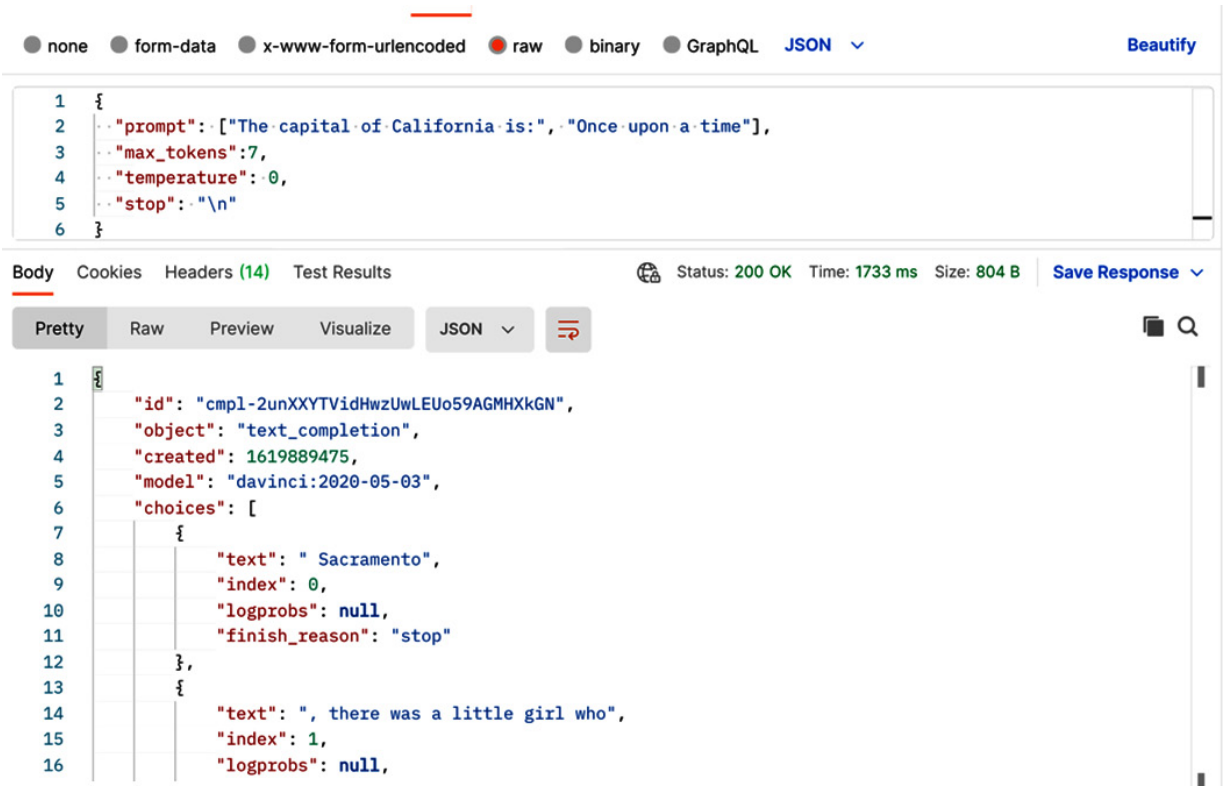



Figure 4.21 – Sending multiple prompts

The main takeaway from this example is that there are things you can do with the API that you can't do in the Playground. So, understanding how to work with the OpenAI API enables you to go beyond what you can do in the Playground.

Another example of something you can do from the API but not the Playground is semantic searching. Let's look at that next.

## Using the Semantic Search endpoint

In *Chapter 2*, *GPT-3 Applications and Use Cases*, we discussed semantic search. By way of a review, semantic search lets you perform a Google-like search over a list of provided documents. A query (a word, phrase, question, or statement) is compared to the contents of documents to determine whether semantic similarities exist. The result is a ranking, or score, for each document. The score is usually between 0 and 300 but can sometimes go higher. A higher score, above 200, typically means the document is semantically similar to the query.

To perform a semantic search using the API, you'll make a POST request to the Semantic Search endpoint. Like the Create Completions endpoint, you'll also include a JSON object in the request body. The JSON body object has two elements – the documents element and the query element. The documents element is an array of documents to be searched, and each item in the array is a string that

represents a document. Alternatively, a document can be provided in a pre-uploaded file that can be referenced in the request. In *Chapter 9*, *Building a GPT-3 Powered Question-Answering App*, we'll look at using files in detail. For now, however, we'll focus on providing documents as an array with the endpoint request.

A document could be a single word, sentence, paragraph, or longer text. The value of the query element is a string containing the search word or phrase that will be searched against the documents. This might be something like a question or a statement.

Again, a semantic search ranks a query based on how semantically similar it is to one or more documents. So, it's not necessarily a search for similar words. For example, the following JSON object provides a list of vehicles (plane, boat, spaceship, or car) as the documents and the query **"A vehicle with wheels"**:

```
{
  "documents": [
    "plane",
    "boat",
    "spaceship",
    "car"
  ],
  "query": "A vehicle with wheels"
}
```

Let's take a look at what the results would look like from the previous JSON. We'll use Postman. Remember that all of the OpenAI API endpoints require authorization, so, in Postman, we first make sure that the proper authorization settings are in place. From there, the steps are the same as making a request to the Completions endpoint:

1. Set the request type to **POST**.
2. Enter the **URI** endpoint.
3. Set the body to **raw**.
4. Select **JSON** as the body content type.
5. Enter the JSON body.

The Semantic Search endpoint URI is **https://api.openai.com/v1/engines/{engine_id}/search**, where **{engine_id}** is replaced by a valid engine ID (such as **davinci** or **ada**). After setting up and submitting the API call in Postman, you should see results like those in the following screenshot:

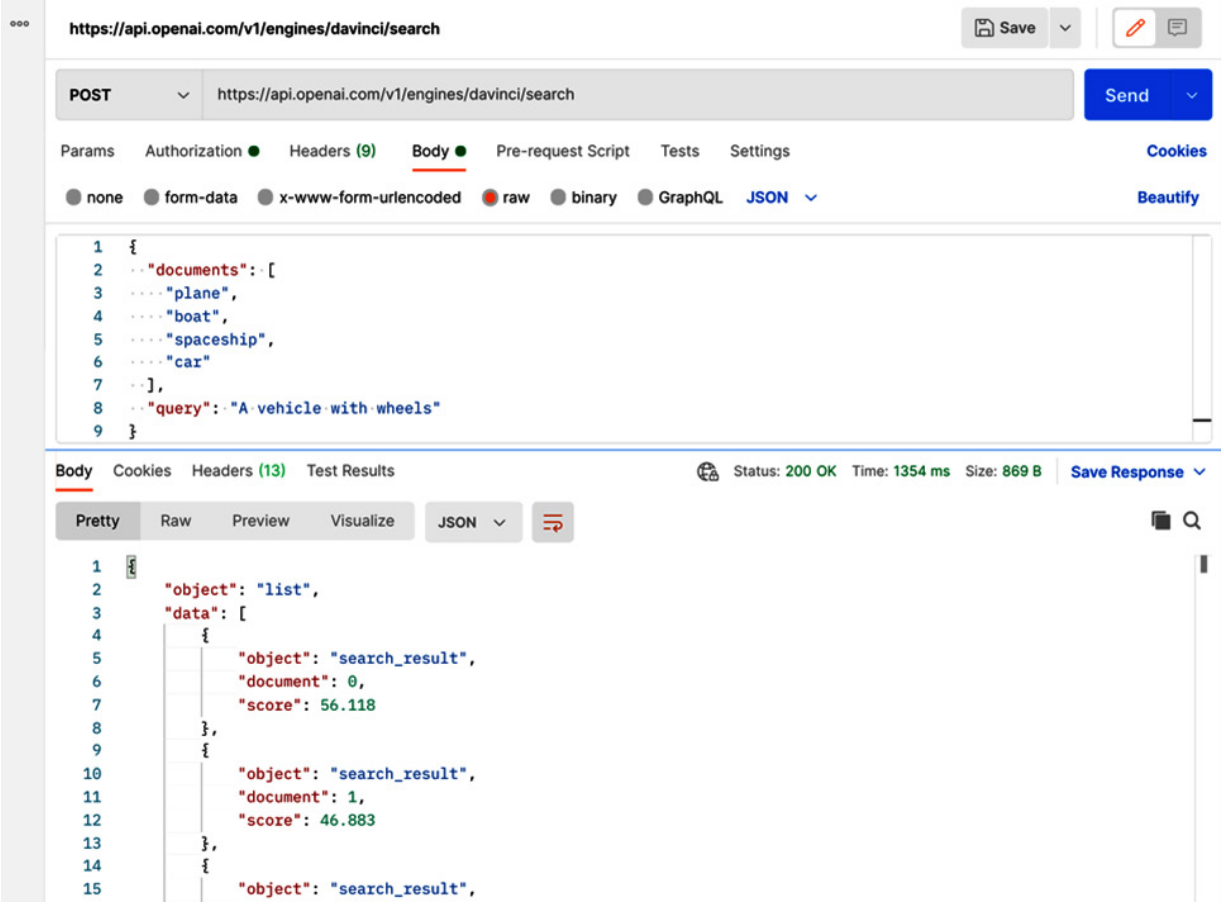



Figure 4.22 – Semantic Search results

The JSON object returned by the Semantic Search endpoint contains three elements: an object, data, and an engine. The value of the data element is a JSON array of results – one item for each document. Recall from our earlier introduction to JSON that items in an array are ordered, meaning that each item can be referenced by a number, the first one starting with zero. So, in our example, the following values would apply:

- **0** = plane
- **1** = boat
- **2** = spaceship
- **3** = car

Knowing that each document is associated with a numeric value, when you look at the following results returned from the search API, you can see that document **3** (car) got the highest score and therefore represents the document that is most semantically similar:

```
{
    "object": "list",
    "data": [
```

```
        {
            "object": "search_result",
            "document": 0,
            "score": 56.118
        },
        {
            "object": "search_result",
            "document": 1,
            "score": 46.883
        },
        {
            "object": "search_result",
            "document": 2,
            "score": 94.42
        },
        {
            "object": "search_result",
            "document": 3,
            "score": 178.947
        }
    ],
    "model": "davinci:2020-05-03"
}
```

The document number rather than the document itself is included in the data array because the document itself might be a long string of text and using the document number is more efficient. So, you will need to match the results returned to the documents sent. But that's relatively straightforward when you're working with code – and that's exactly what we'll get started on in the next chapter.

## Summary

In this chapter, we looked at working with the OpenAI API. We started with an introduction/review of what an API is and then we became familiar with the basics of the HTTP protocol. We reviewed the OpenAI API endpoints and covered how to access the API using basic authentication with an OpenAI API key and how to authenticate for an account with access to multiple organizations. From

there, we learned about the JSON data-interchange format before learning how to make API calls to the Completions endpoint, the Engines endpoint, and the Semantic Search endpoint using Postman.

In the next chapter, we'll put the knowledge acquired in this chapter to work and dive into using code to call the API.

# *Chapter 5*: Calling the OpenAI API in Code

In the previous chapter, we looked at calling the OpenAI API using Postman. In this chapter, you'll learn how to start using the API in code. Since the OpenAI API can be used with virtually any programming language, we'll take a look at two popular languages, JavaScript and Python. If you're not a programmer, no problem. We'll be using an online environment that makes getting started with code super simple. Plus, easy-to-follow code examples are provided in both JavaScript and Python so you can choose the language that is most comfortable for you. Also, for clarity, the examples are purposely written with as little code as possible. So, the examples may not always follow coding best practices, but the goal is to make following along easy!

The topics we'll cover in this chapter are as follows:

- Choosing your programming language
- Introducing replit.com
- Using the OpenAI API with Node.js/JavaScript
- Using the OpenAI API with Python
- Using other programming languages

## Technical requirements

This chapter requires that you have access to the **OpenAI API**. You can request access by visiting the following URL: https://openai.com.

## Choosing your programming language

The OpenAI API is a standards-based API that can be used with virtually any modern programming language. In this chapter, we'll walk through examples in JavaScript and Python. But these are just two of the many languages that could be used. JavaScript and Python were selected because they are both extremely popular and easy to get started with. If you're proficient with another modern programming language, that's probably the best option for you. However, if you're new to programming, your coding skills are rusty, or you're interested in dabbling in a new language, then JavaScript and Python are both great languages. But the examples for each language will all be the same – just the programming language is different. So, you can skip to the section that covers the language you prefer. Or, if you're feeling adventurous, of course, you're free to try them both!

## Introducing replit

In this chapter and the following chapters, we're going to be writing code. So, we'll need a code editor of some sort. As with programming languages, when it comes to code editors, there are many to choose from. The right one is a matter of personal preference and opinion. For our examples, we'll be using an online code editor from replit.com.

While all of the examples we'll be working through could be done in any code editor, replit lets us skip the installation and software setup process because we can do everything in our web browser.

Plus, with a code editor, some setup on your local computer would be required. For example, for both JavaScript and Python a runtime environment needs to be installed. We're going to skip all that so we can focus more time on learning GPT-3 and the OpenAI API.

So, what is **replit.com**? It's an in-browser code editor and **Integrated Development Environment** (**IDE**) that lets you start coding in over 50 programming languages without spending any time setting up a local development environment. So, for our examples, you can jump right into the code even if you don't have your local computer set up to run Node.js or Python.

The following screenshot shows the replit home page, which is where you'll go to set up a free replit account if you don't have one:

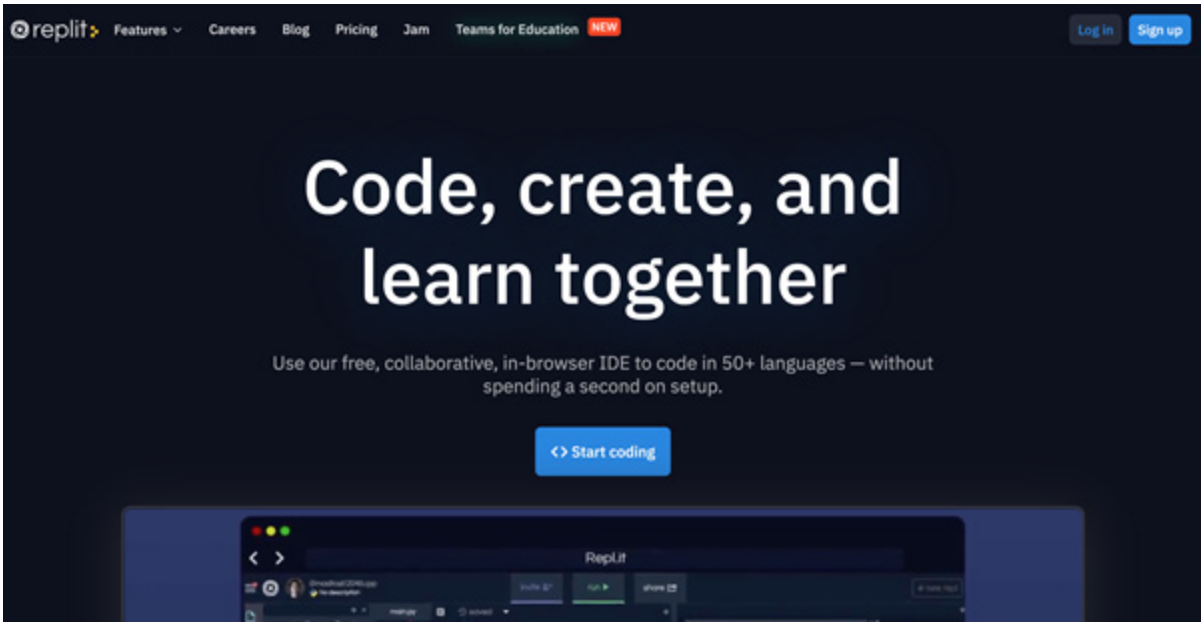



Figure 5.1 – The replit home page

Creating an account is just a matter of clicking the signup button and completing the signup process. For the examples in this book, a free account is all you need.

After you've signed up, you'll be logged in to the default home screen as *Figure 5.2* shows:

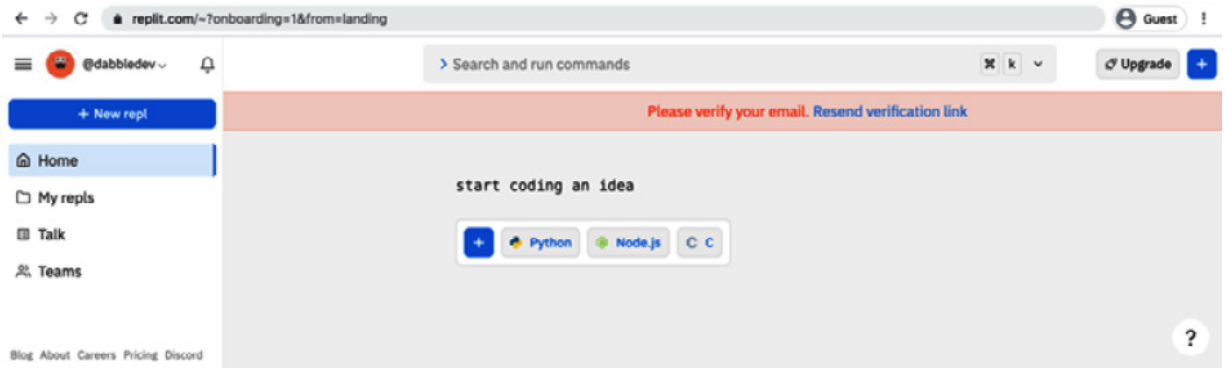



Figure 5.2 – Replit default home screen

From here, we'll create two code projects, which are referred to as **repls** in replit.

## Creating a repl

We'll be looking at code examples in JavaScript and Python. So, we'll need to set up a repl for each. For JavaScript, we'll be using **Node.js** (or just Node), which is a runtime environment for JavaScript. We'll talk more about Node in the *Using the OpenAI API with Node.js/JavaScript* section. We will start by creating a repl for Python. Then, we'll create a repl for Node.js/JavaScript.

### Creating a repl for Python

Lets' start by creating a new repl for Python. If you'd prefer to just follow along using just Node.js/JavaScript, you can skip to *Creating a repl for Node.js*.

From the home screen, click on the **+ New Repl** button:

1. Choose Python from the repl type dropdown.
2. Name the repl **exploring-gpt3-python**.
3. Click the **Create repl** button.

Next, we'll create a repl for Node.js. If you'll only be using Python, you can skip the next section.

### Creating a repl for Node.js

If you're following along with Node.js, you'll need a Node repl. To create a Node repl, use the following steps. From the home screen, click on the **+ New Repl** button:

1. Choose Node.js from the repl type dropdown.
2. Name the repl **exploring-gpt3-node**.
3. Click the **Create repl** button.

The following screenshot shows a new repl being created for Python. Again, for JavaScript, you'd select Node.js rather than Python and then name the repl **exploring-gpt3-node**:

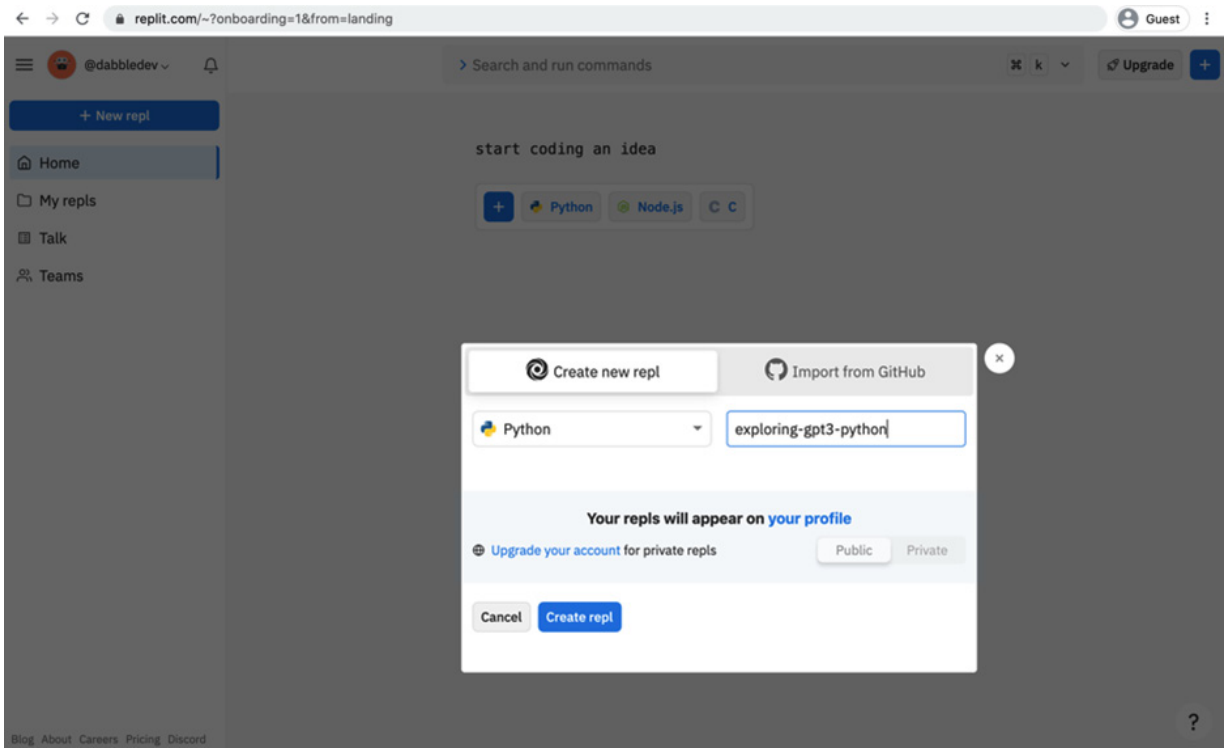



Figure 5.3 – Setting up a repl for Python

After a repl is created, you're automatically redirected to the replit editor with your new repl loaded. The editor is broken down into three panes: the **Navigation** pane, the **Code editor** pane, and the **Output** pane. New repls include a default file that is opened in the code editor (the middle pane) when you first access the repl.

In the case of a Python project, the default file is named **main.py**; for a Node.js project, the default file is named **index.js**. The following screenshot shows the replit editor with the **exlploring-gpt3-python** repl loaded and **main.py** open in the editor pane. A Node.js repl would look the same but the default file would be named **index.js** rather than **main.py**:


Figure 5.4 – The Replit editor with a default Python repl

The default file is the code file that gets run by default when the **Run** button is clicked. The run button is located in the middle of the top navigation bar located just above the three editor panes. The results or output from the code is then shown in the replit console – the output pane on the right.

### *IMPORTANT NOTE*

*The default files are empty when a repl is first created, so clicking the **Run** button won't do anything until you write some code.*

Above the three editor panes, on the left side of the top navigation bar is a hamburger menu (the icon that looks like stacked lines). If you click that icon, the main navigation options will be displayed. You can use this menu to navigate between repls by selecting the **My repls** menu and choosing the repl you want to open.

The following screenshot shows the main menu with the **My repls** option selected:

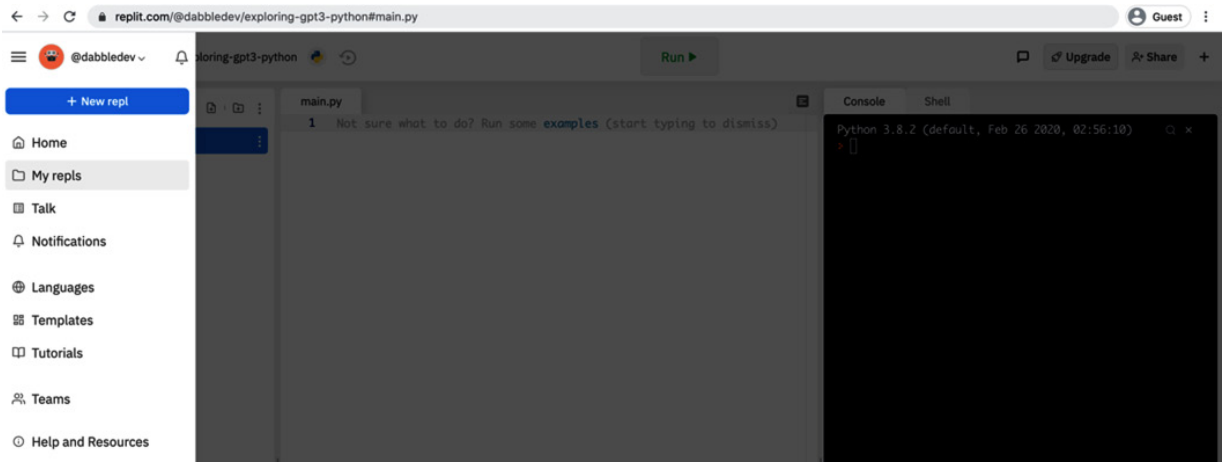



Figure 5.5 – Accessing the main navigation menu

After creating a repl for the language you want to work with (or both languages), y, you'll want to store your OpenAI API key as an environment variable.

## Setting your OpenAI API key as an environment variable

Environment variables are named values that can be accessed in code but don't get shared with others. You typically use environment variables to set parameter values that are private or specific to a user, for example, your OpenAI API key.

In replit, you can save environment variables by clicking on the padlock icon (**Secrets**) in the navigation pane and adding a name and value pair. You'll need to do this for each repl you're working

with – so, one in your **exploring-gpt3-node** repl and/or in your **exploring-gpt3-python** repl. To add your OpenAI API Key as an environment variable, do the following:

1. Open your repl.
2. Click the padlock icon (**Secrets**) in the middle of the navigation pane like the following screenshot shows:

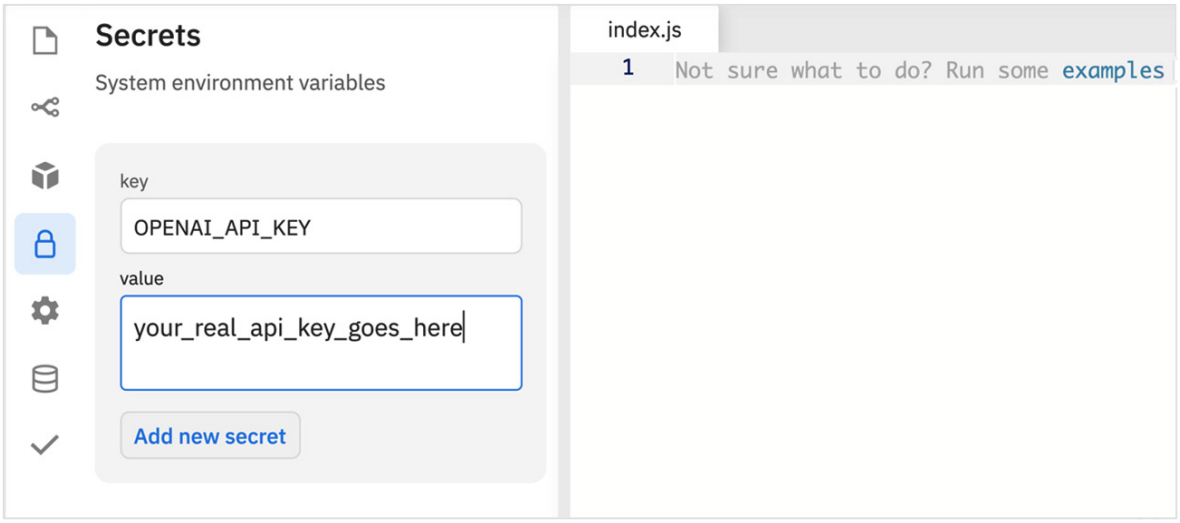



Figure 5.6 – Adding a new secret/environment variable in Replit

3. In the **Key** input textbox, add **OPENAI_API_KEY**.
4. In the **Value** input text area, paste in your OpenAI API.
5. Click the **Add new secret** button.

Again, you'll need to add the **OPENAI_API_KEY** secret/environment variable to each repl. So, you'll complete the preceding steps twice if you're following along with both Node.js and Python.

### *IMPORTANT NOTE*

*If you're using a free version of replit.com your code will be public by default. However, **secrets/environment variables** don't get shared publicly but can be seen by collaborators who you explicitly invite to the repl. This is important because you don't want to share or accidentally expose your OpenAI API key. To read more about using secrets and environment variables in replit, visit [https://docs.replit.com/repls/secret-keys](https://docs.replit.com/repls/secret-keys).*

Before we start coding, there is one more special file we need to create, the **.replit** file.

## Understanding and creating the .replit file

In replit, by default, when you click the **Run** button the **main.py** file will be run for Python repls and the **index.js** file will be run for Node.js repls. However, you can create a file named **.replit** and change the code file that will be executed by the **Run** button. As we work through different examples, we'll be creating and testing code in multiple files. So, we're going to need a **.replit** file. To create a **.replit** file, do the following:

1. Open your repl.
2. Click the add file icon on the top-right side of the navigation pane.
3. Name the file **.replit**.
4. Add the following text to the first line of the **.replit** file:

   **For Python repls**:

   ```
   run = "python main.py"
   ```

   **For Node.js repls**:

   ```
   run = "node index.js"
   ```

You might have noticed that we've just added instructions to run the file that would be run by default. That's okay for now, we'll come back and edit the **run** statement in our **.replit** file later.

After you've created a repl for Node.js and/or a repl for Python with your OpenAI API key added as an environment variable, and a **.replit** file, you're ready to start coding. So, in the next section, we'll look at examples using Node.js/JavaScript. But, if you're just following along with Python, you can skip ahead to *Using the OpenAI API in Python*.

## Using the OpenAI API with Node.js/JavaScript

JavaScript is the first programming language we're going to look at. JavaScript was originally created for scripting functionality on web pages. However, today, JavaScript can be used for just about any type of application development, from building websites and mobile apps to creating command-line tools, chatbots, and voice assistants – all thanks to Node. As mentioned previously, Node is a runtime environment for JavaScript. It lets us use JavaScript outside the web browser.

JavaScript and Node.js are both free to use and can be run on Linux, macOS, and Windows operating systems. But we won't be installing Node.js or running anything locally because all of our code will be run on replit. However, none of the code we'll be writing is in any way dependent on replit. So, everything we'll be doing could be done in any environment that was properly configured for Node development.

Alright, let's get to coding. To use the OpenAI API in Node, we essentially just need to do what we did in *Chapter 4*, *Working with the OpenAI API* – make authenticated HTTP requests to the OpenAI API endpoints. But rather than using Postman, we'll be making HTTP requests with JavaScript. We'll start with the simplest example – calling the engines endpoint.

## Calling the engines endpoint

We'll start by setting up a new folder for the code we'll be working on in this chapter. Then we'll add a file for our first example and update the **.replit** file so the replit **Run** button executes our new file. Here are the steps to follow:

1. Open your **exploring-gpt3-node** repl.
2. Click the **Add folder** icon in the top right of the navigation pane.
3. Name the folder **chapter05**.
4. Select the **chapter05** folder and then click the **Add file** option and add a new file named **engines.js**.
5. Edit the **.replit** file so the **Run** button executes **chapter05/engines.js** using the following text:

   ```
   run = "node chapter05/engines.js"
   ```

6. Now, open the **chapter05/engines.js** file in your replit editor by clicking on the filename in the navigation pane. Your replit editor should look something like the following screenshot:

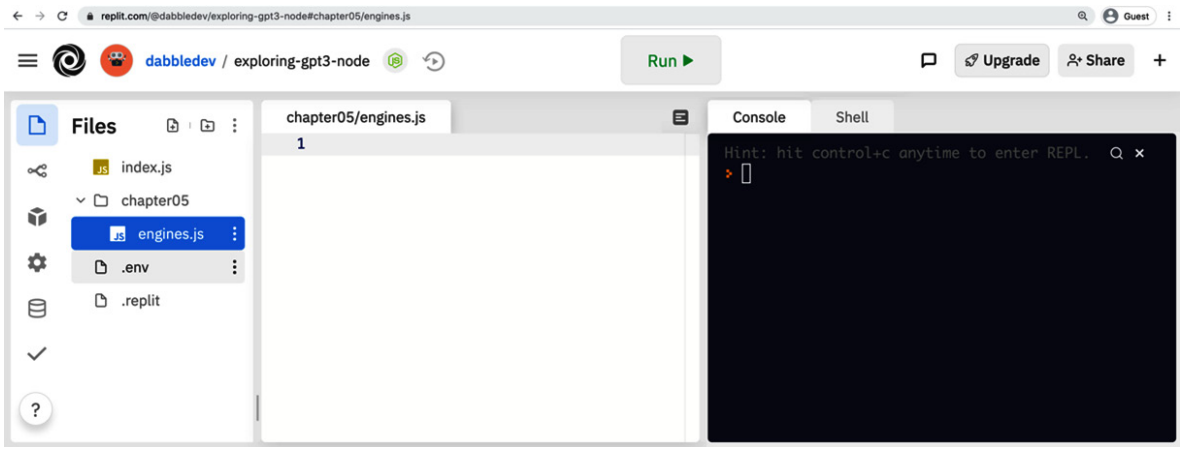



Figure 5.7 – Editing engines.js in the Replit editor

7. So, now we're going to write some code to call **https://api.openai.com/v1/engines**. There are a number of ways to make an HTTP request with Node and JavaScript. We won't get into all of the different options or the pros and cons of one approach over another. We're just going to use a popular code library (referred to as a **module** in Node) called **axios**.

   ### *IMPORTANT NOTE*

   *Packaged and shared code libraries are common across most modern programming languages. In Node.js, packages (or modules) are most commonly shared using a package manager called npm. To learn more about Npm, you can visit https://npmjs.com. replit automatically manages package dependencies for Node.js projects. However, in a local development environment, the Npm command-line tool would most commonly be used to manage packages used in code.*

8. In short, the **axios** module minimizes the code we need to write to make HTTP requests using JavaScript. So, we'll include a reference to the **axios** module on the first line of our **engines.js** file with the following code:

   ```
   const axios = require('axios);
   ```

9. You'll recall from *Chapter 4*, *Working with the OpenAI API*, that all of the OpenAI API endpoints require authorization. So, on the next line, we'll add a variable to hold our OpenAI API key. But rather than hardcoding the value, we will pull it from the environment variable we set up:

```
const apiKey = process.env.OPENAI_API_KEY;
```

10. Next, we'll create an in-code HTTP client to make the requests. The **axios** module makes this super simple. To create the HTTP client instance, we'll add the following code:

```
const client = axios.create({
    headers: { 'Authorization': 'Bearer ' + apiKey }
});
```

Note in the preceding code that we've added an HTTP **Authorization** header that uses our OpenAI API key as the bearer token. This lets the **axios** client make authenticated requests to the API endpoints. All that remains is code to actually make the request. Here is what that code looks like:

```
client.get('https://api.openai.com/v1/engines')
  .then(result => {
    console.log(result.data);
}).catch(err => {
    console.log(err.message);
});
```

In the preceding code, we're using the **axios** client instance to make a GET request to the **https://api.openai.com/v1/engines** endpoint. Then, we're logging the results returned from the endpoint to the console. Or, in the case of an error, we're logging the error message to the console.

11. At this point, you should be able to click the **Run** button in replit and see the results in the replit console, where there'll be an output like what's shown in the following screenshot:

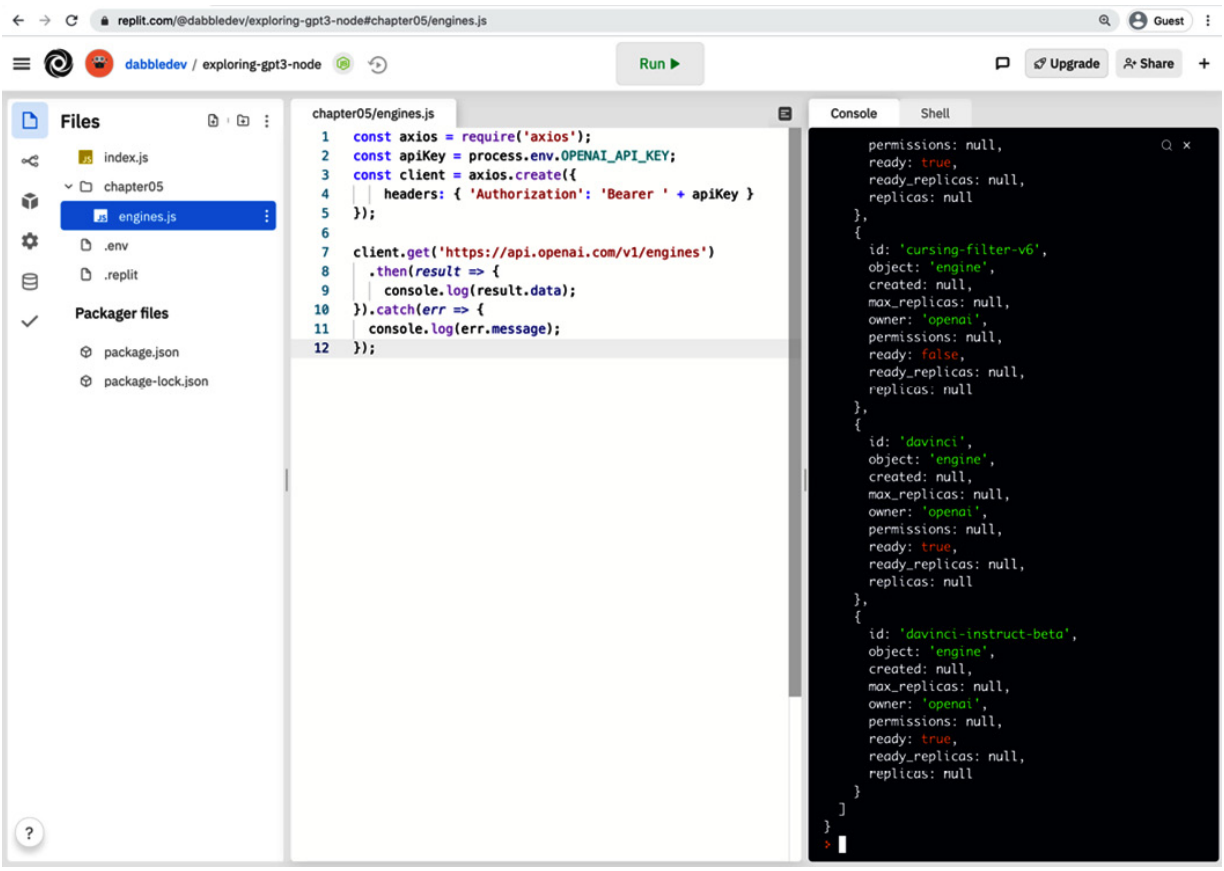



Figure 5.8 – Results from running the engines.js code

So, with 12 lines of code (really 11 since one line is blank), we've made our first API request using JavaScript and Node.

Now let's take a look at calling the Completions endpoint. This one is a bit more involved because we need to send data to the endpoint. But still, the code is pretty simple.

## Calling the Completions endpoint

Now we'll look at calling the Completions endpoint. We'll start by creating a new file named **completions.js** in the **chapter05** folder and we'll edit our **.replit** file so the **Run** button will execute our new file. To do that, follow these steps:

1. Open your **exploring-gpt3-node** repl.
2. Select the **chapter05** folder in the navigation pane.
3. Click the **Add file** option and add a new file named **completions.js**.
4. Edit the **.replit** file so the **Run** button executes **chapter05/completions.js** using the following text:

```
run = "node chapter05/completions.js"
```

5. Next, copy the first five lines of code from **chapter05/engines.js** into **chapter05/completions.js** so your **completions.js** file starts with the following code:

```
const axios = require('axios');
const apiKey = process.env.OPENAI_API_KEY;
const client = axios.create({
    headers: { 'Authorization': 'Bearer ' + apiKey }
});
```

For the Completions endpoint, we'll be sending parameters. So, the next thing we'll do is add what will become the JSON object that is sent in the HTTP body. Recall from *Chapter 4*, *Working with the OpenAI API*, that there are a number of parameters the Completions endpoint accepts. But for this example, we're just going to be sending the **prompt** parameter and the **max_tokens** parameter. Here is what the code for that looks like:

```
const completionParams = {
  "prompt": "Once upon a time",
  "max_tokens": 10
}
```

All that's left is to actually make the request. This time we'll be making an HTTP POST and passing our parameters. So, the code is slightly different from the last example but here is what it looks like:

```
client.post('https://api.openai.com/v1/engines/davinci/completions', completionParams)
  .then(result => {
  console.log(result.data);
}).catch(err => {
  console.log(err);
});
```

That's it. Now when you click the **Run** button in replit, you should get a result back from the Completions endpoint that looks something like the following screenshot:

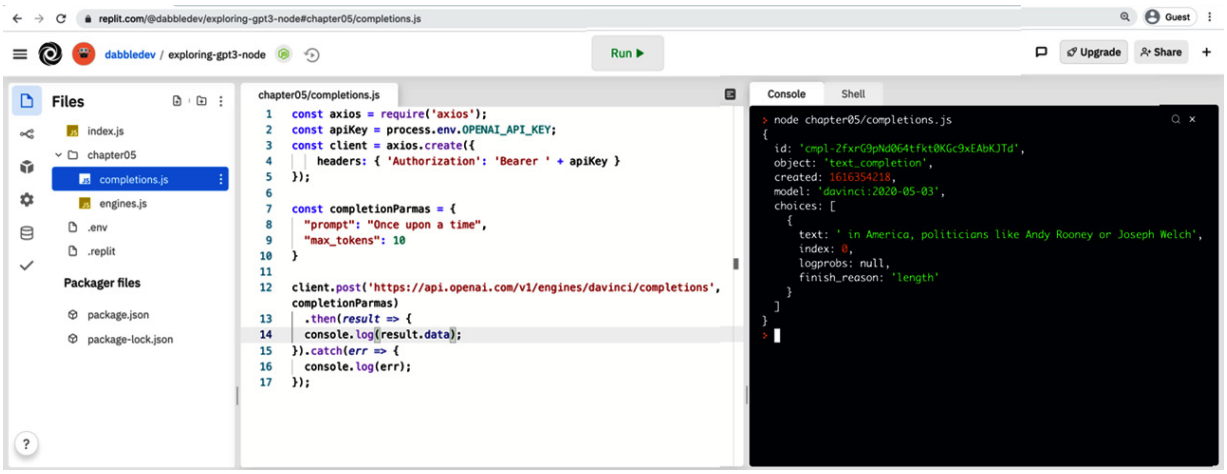



Figure 5.9 – Output from Completions.js

As we discussed in *Chapter 4*, *Working with the OpenAI API*, the responses returned from the endpoints are in JSON format. But if you wanted to format the output, you could extract just what you wanted to display from the JSON. For example, you could update **console.log(result.data)** with the following code to display the original prompt text **Once upon a time** and the completion text returned by the API:

```
console.log(completionParams.prompt + result.data.choices[0].text);
```

If you make that change and run the code again, the output will look something like the output in the following screenshot:

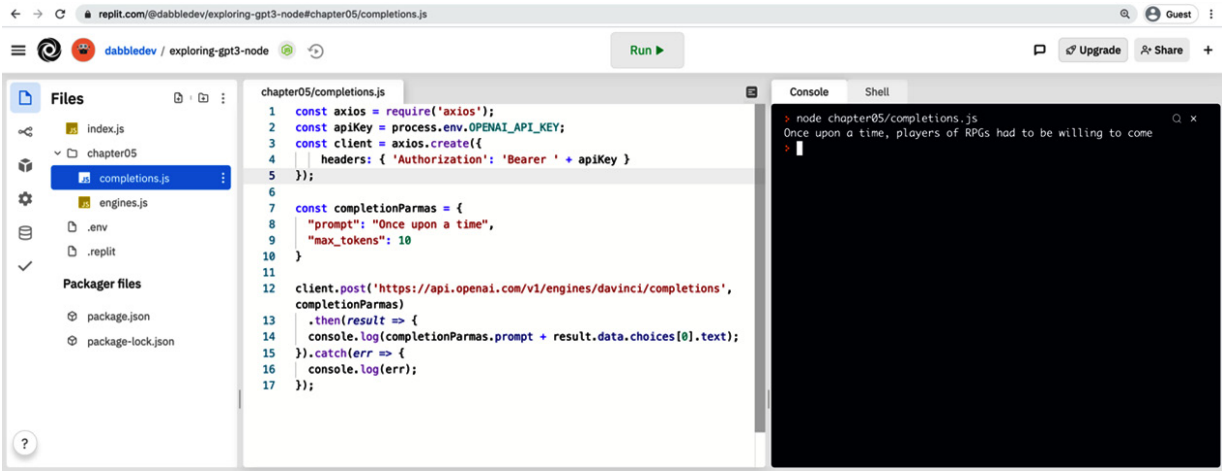



Figure 5.10 – Formatted results from completions.js

With just two examples, we've covered the basic code you need to work with all of the OpenAI endpoints. But before we move on, let's look at one more example using the search endpoint.

## Calling the search endpoint

For our search example, we'll create a new file named **search.js** in the **chapter05** folder and we'll edit our **.replit** file so the **Run** button will execute **chapter05/search.js**. To do that, follow these steps:

1. Open your **exploring-gpt3-node** repl.
2. Select the **chapter05** folder in the navigation pane.
3. Click the **Add file** option and add a new file named **search.js**.
4. Edit the **.replit** file so the **Run** button executes **chapter05/search.js** using the following **run** command:

```
run = "node chapter05/completions.js"
```

5. Next, copy the first five lines of code from **chapter05/engines.js** into **chapter05/search.js** so your **search.js** file starts with the following code:

```
const axios = require('axios');
const apiKey = process.env.OPENAI_API_KEY;
const client = axios.create({
    headers: { 'Authorization': 'Bearer ' + apiKey }
});
```

6. For the search endpoint, we'll be sending parameters. So, the next thing we'll do is add what will become the JSON object that is sent in the HTTP body. In *Chapter 4*, *Working with the OpenAI API*, we covered the parameters for the search endpoint, but here is the code we'll use for this example:

```
const data = {
  "documents": ["plane", "boat", "spaceship", "car"],
  "query": "A vehicle with wheels"
}
```

7. All that's left is to actually make the request. Like with the completions example, we'll be making an HTTP POST and passing our parameters:

```
client.post('https://api.openai.com/v1/engines/davinci/search', data)
  .then(result => {
    console.log(result.data);
  }).catch(err => {
    console.log(err);
  });
```

8. That's it. Now when you click the **Run** button in replit, you should get a result back from the search endpoint that looks something like the following screenshot:

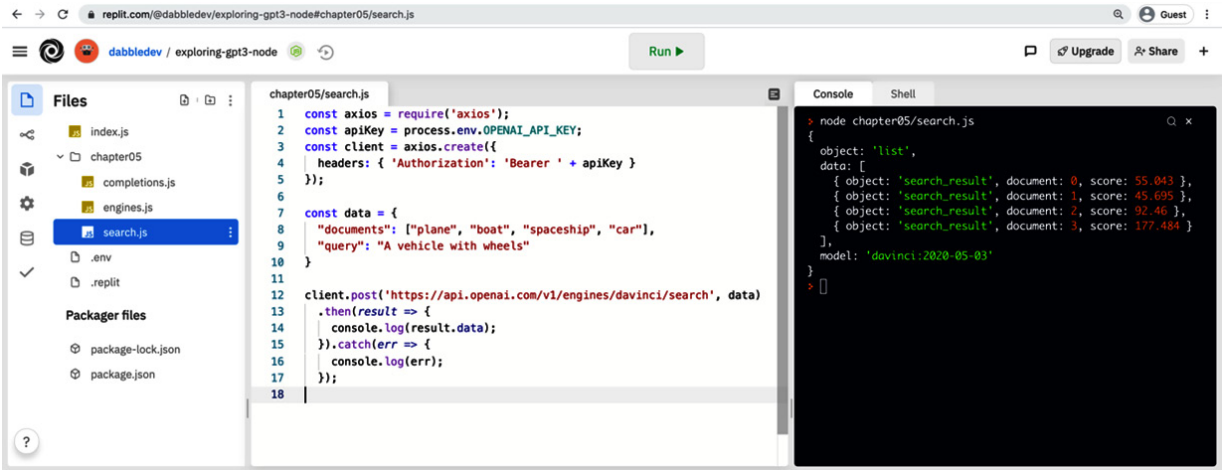



Figure 5.11 – Results from search.js

All of the JavaScript examples we covered in this section were pretty simple. However, they should give you a general idea of how to get started calling the OpenAI API with JavaScript and Node.js, which was the goal. In the following chapters, we'll dive deeper, but for now we're going to move on to our next language – Python.

## Using the OpenAI API in Python

Python is another popular programming language, and it is especially popular among the machine learning community. It's a robust language but it's also very beginner-friendly. If you are coding for the first time, Python will probably be one of the easier languages to get started with. Like with JavaScript/Node, Python is open source and can be used on Linux, macOS, or Windows. It is also included with many operating systems, including macOS and most Linux distributions. But since we'll be working in replit, we don't need to worry about Python being installed on our local machine. That said, nothing in the code we'll be writing depends on anything that is specific to replit. So, everything we'll be doing could be done on any computer that is properly configured for doing Python development.

For our first example, we'll look at using Python to call the OpenAI Engines endpoint.

### Calling the engines endpoint

We'll start by setting up a new folder for the Python code we'll be working on in this chapter. Then, we'll add a file for our first example and update the **.replit** file so the replit **Run** button executes our new file. Here are the steps to follow:

1. Open your **exploring-gpt3-python** repl.
2. Click the **Add folder** icon in the top right of the navigation pane.
3. Name the folder **chapter05**.

4. Select the **chapter05** folder, then click the **Add file** option and add a new file named **engines.py**.
5. Edit the **.replit** file so the **Run** button executes **chapter05/engines.py** using the following text:

```
run = "python chapter05/engines.py"
```

6. Now, open the **chapter05/engines.py** file in your replit editor by clicking on the filename in the navigation pane. Your replit editor should look something like the following screenshot:

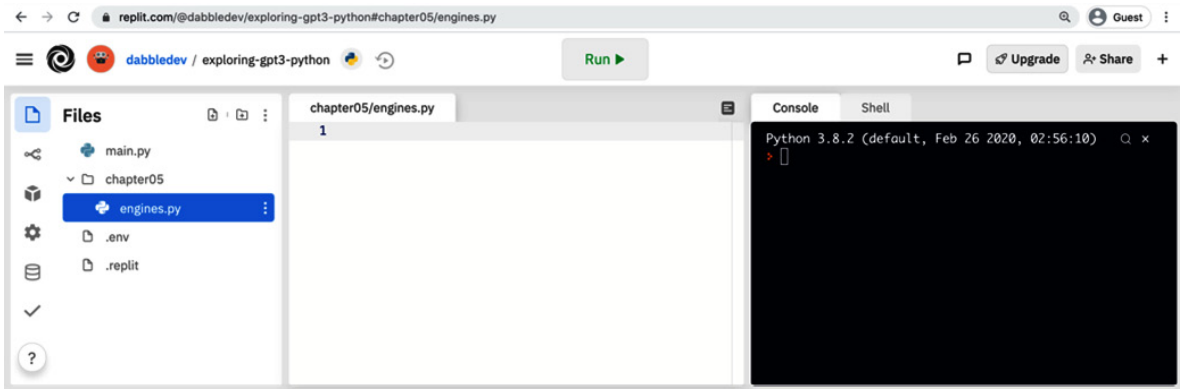



Figure 5.12 – Editing engines.py in the Replit editor

7. So, now we're going to write some Python code to make an HTTP request to the **https://api.openai.com/v1/engines** endpoint. There are a number of ways to make an HTTP request with Python. We won't get into all of the different options or the pros and cons of one approach over another. We're just going to use the popular **requests** library.

## *IMPORTANT NOTE*

*Packaged and shared code libraries are common across most modern programming languages. In Python, packages (or libraries) are most commonly shared using a package manager called PIP. To learn more about PIP, you can visit https://pypi.org/project/pip/. replit automatically manages package dependencies for Python projects. However, in a local development environment, the PIP command-line tool would most commonly be used to manage Python packages.*

8. In short, the **requests** library minimizes the code we need to write to make HTTP requests using Python. So, we'll include a reference to the **requests** library on the first line of our **engines.py** file with the following code:

```
import requests
```

9. You'll recall from *Chapter 4*, *Working with the OpenAI API*, that all of the OpenAI API endpoints require authorization. So, next we'll add some code to get the API key that we set up in the **.env** file and save it to a variable. We'll do that with the following lines:

```
import os

apiKey = os.environ.get("OPENAI_API_KEY")
```

10. Next, we'll create a variable named **headers** to hold our authorization information, which will be required to make the HTTP request:

```
headers = {
    'Authorization':'Bearer ' + apiKey
    }
```

11. All we need to do now is make the request. We'll do that with the following code that saves the response to a variable named **result**:

    ```
    result = requests.get('https://api.openai.com/v1/engines'
    ,headers=headers)
    ```

12. To display the JSON results in the console, we'll add the last line as follows:

    ```
    print(result.json())
    ```

13. At this point, you should be able to click the **Run** button in replit, see the results in the replit console, and see an output like what's shown in the following screenshot:

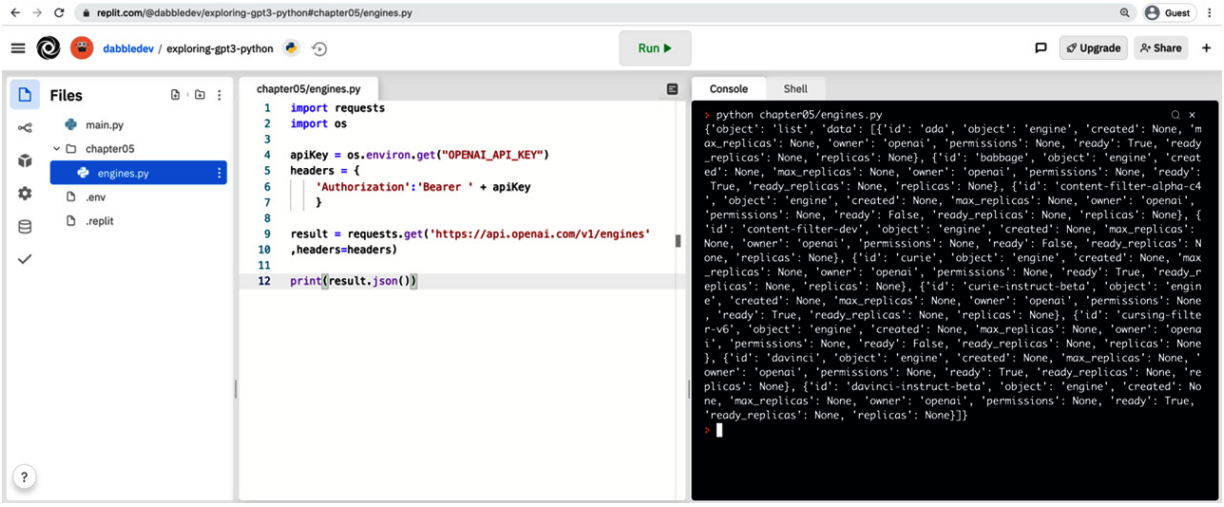



Figure 5.13 – Results from running the engines.py code

So, with 12 lines of code (really 9 since three lines are blank), we've made our first API request using Python. Now let's take a look at calling the Completions endpoint. This one requires a bit more code because we need to send data to the endpoint. But as you'll see, it's still pretty simple.

## Calling the completions endpoint

We'll start by creating a new file named **completions.py** in the **chapter05** folder. Then we'll edit our **.replit** file so the **Run** button will execute **chapter05/completions.py**. To do that, follow these steps:

1. Open your **exploring-gpt3-python** repl.
2. Select the **chapter05** folder in the navigation pane.
3. Click the **Add file** option and add a new file named **completions.py**.
4. Edit the **.replit** file so the **Run** button executes **chapter05/completions.py** using the following text:

    ```
    run = "python chapter05/completions.py"
    ```

5. Next, we'll add the following code. It very similar to the starting code in **chapter05/engines.py** but we need to add the **json** library and the **Content-Type** header. So, your **completions.py** file should start with the following code:

```
import requests

import os

import json

apiKey = os.environ.get("OPENAI_API_KEY")

headers = {

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

    'Authorization':'Bearer ' + apiKey

    }
```

6. For the completions endpoint, we'll be sending parameters. So, the next thing we'll do is add what will become the JSON object that is sent in the HTTP body. Recall from *Chapter 4*, *Working with the OpenAI API*, that there are a number of parameters the completions endpoint accepts. But for this example, we're just going to be sending the **prompt** parameter and the **max_tokens** parameter. Here is what the code for that looks like:

```
data = json.dumps({

    "prompt":"Once upon a time",

    "max_tokens":15

    })
```

   To make the code a bit more readable, we'll also create a variable for the endpoint URL with the following code:

```
url = 'https://api.openai.com/v1/engines/davinci/completions'
```

7. All that's left is to actually make the request and print out the results. This time, we'll be making an HTTP POST and passing our parameters. So, the code is slightly different from the last example but here is what it looks like:

```
result = requests.post(url,headers=headers,data=data)

print(result.json())
```

8. That's it. Now when you click the **Run** button in replit, you should get a result back from the completions endpoint that looks something like the following screenshot:

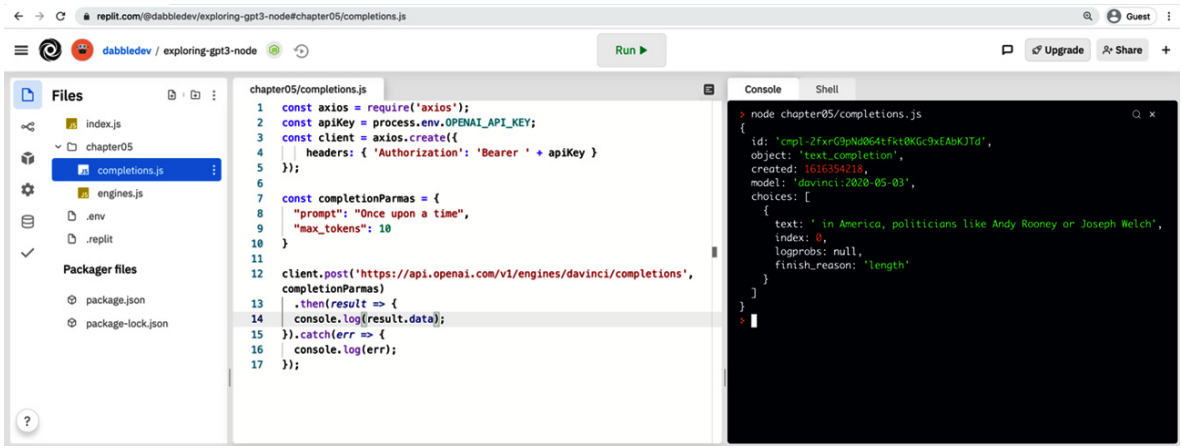



Figure 5.14 – Output from the completions.js

9. As we discussed in *Chapter 4*, *Working with the OpenAI API*, the responses returned from the endpoints are in JSON format. But if you wanted to format the output, you could extract just what you wanted to display from the JSON. For example, you could update **console.log(result.data)** with the following code to display the original prompt text **Once upon a time** and the completion text returned by the API:

```
console.log(completionParams.prompt + result.data.choices[0].text);
```

10. If you make that change and run the code again, the console output should look something like the output in the following screenshot:

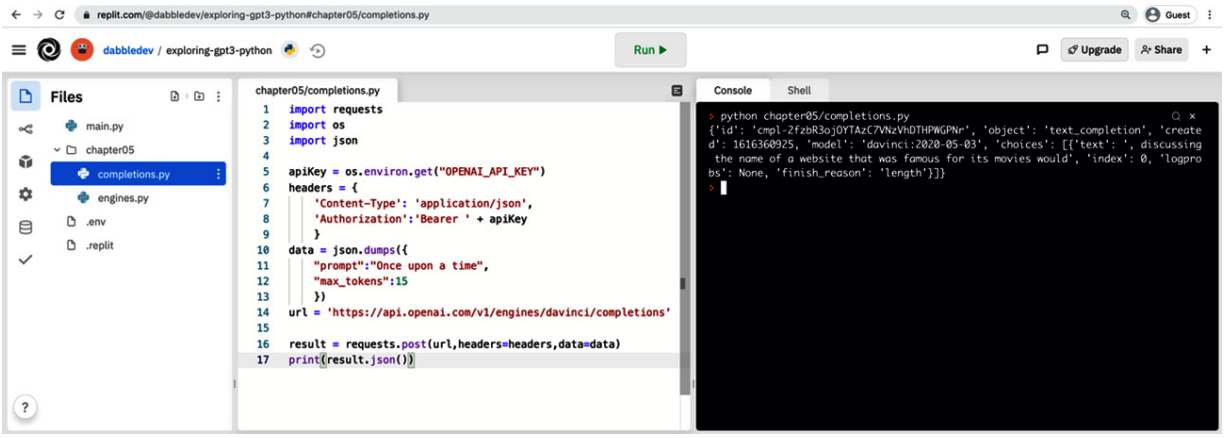



Figure 5.15 – Formatted results from completions.py

Alright, let's look at one more example. This time we'll use Python to call the search endpoint.

## Calling the search endpoint

For our search example, we'll create a new file named **search.py** in the **chapter05** folder and we'll edit our **.replit** file so the **Run** button will execute **chapter05/search.py**. To do that, follow these steps:

1. Open your **exploring-gpt3-python** repl.
2. Select the **chapter05** folder in the navigation pane.
3. Click the **Add file** option and add a new file named **search.py**.
4. Edit the **.replit** file so the run button executes **chapter05/search.py** using the following **run** command:

```
run = "python chapter05/search.py"
```

5. Next, copy the first nine lines of code from **chapter05/completions.py** into **chapter05/search.py** so your **search.py** file starts with the following code:

```
import requests
import os
import json
apiKey = os.environ.get("OPENAI_API_KEY")
```

```
headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer ' + apiKey
}
```

6. For the search endpoint, we'll be sending parameters as we did in **chapter05/completions.py**. So, the next thing we'll do is add what will become the JSON object that is sent in the HTTP body. In *Chapter 4*, *Working with the OpenAI API*, we covered the parameters for the search endpoint, and we'll use the same example – here is the code we'll use:

```
data = json.dumps({
    "documents": ["plane", "boat", "spaceship", "car"],
    "query": "A vehicle with wheels"
})
```

7. We'll add a variable for the endpoint URL just to keep the code a bit easier to read:

```
url = 'https://api.openai.com/v1/engines/davinci/search'
```

8. Finally, we'll add code to make the request and print out the results:

```
result = requests.post(url, headers=headers, data=data)
print(result.json())
```

9. That's it. Now when you click the **Run** button in replit, you should get a result back from the completions endpoint that looks something like the following screenshot:

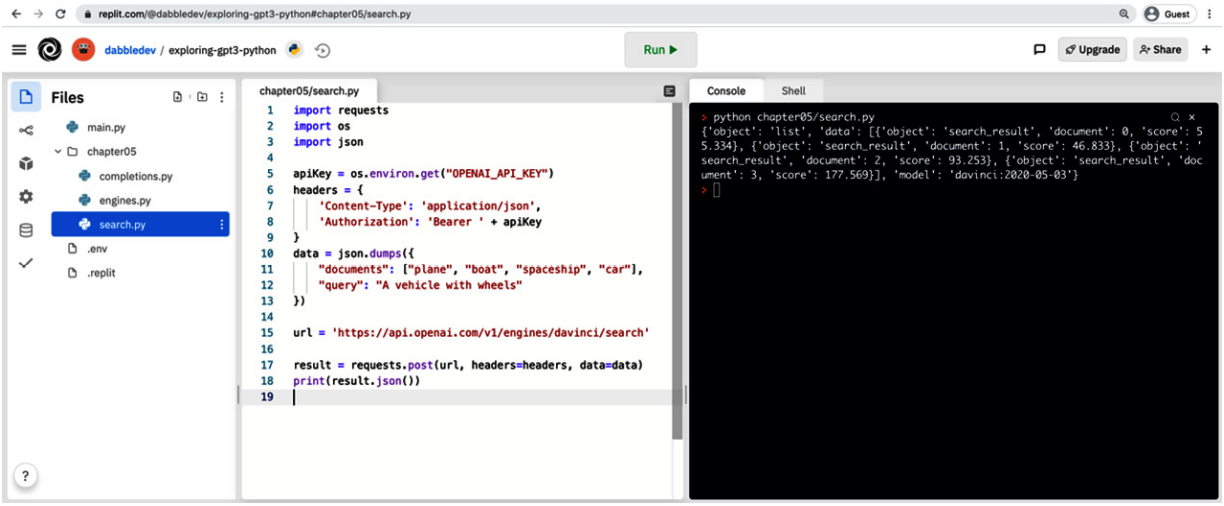



Figure 5.16 – Results from search.py

The goal of this section was to provide a few simple Python examples that call the OpenAI API. At this point, we've covered the basics, but we'll look at more in-depth examples in the following chapters.

## Using other programming languages

In this chapter, we just looked at code examples using JavaScript and Python. But again, virtually any modern programming language could have been used. The OpenAI API is a standards-based HTTP API, so all you need is a language that can make HTTP requests and work with JSON, which, again, is virtually all modern programming languages.

Also, for the examples in this chapter, we called the API directly using general HTTP libraries. We could have also used a library specifically built for the OpenAI API. Multiple libraries exist for both JavaScript, Python, and a number of other languages, including C#/.NET, Go, Java, and Unity, to name a few. You can find a list of community-maintained libraries at https://beta.openai.com/docs/developer-quickstart/community-libraries.

Libraries can simplify working with the OpenAI API. However, it's helpful to understand how to call the endpoints directly, and the API is quite simple to use. For those reasons, we'll be working directly with the API for all of the examples we'll be doing in this book.

## Summary

In this chapter, we looked at using the OpenAI API in code. We got started with an introduction to the in-browser IDE replit. Then, we looked at code examples for calling the engines endpoint, the completions endpoint, and the search endpoint using both Node.js/JavaScript and Python. Finally, we discussed other languages that could be used and libraries that can simplify working with the OpenAI API.

In the next chapter, we will discuss content filtering and look at code examples for implementing content filtering using both JavaScript and Python.

# Section 3: Using the OpenAI API

This section provides hands-on examples for using the OpenAI API with Node.js/JavaScript and Python. Then, it concludes by walking you through building a fully functional GPT-3-powered web app.

This section comprises the following chapters:

- *Chapter 6, Content Filtering*
- *Chapter 7, Generating and Transforming Text*
- *Chapter 8, Classifying and Categorizing Text*
- *Chapter 9, Building a GPT-3 Powered Question-Answering App*
- *Chapter 10, Going Live with OpenAI-Powered Apps*

# *Chapter 6*: Content Filtering

In *Chapter 1*, *Introducing GPT-3 and the OpenAI API*, we briefly mentioned that a content filtering model is available to recognize potentially offensive or harmful language. We also discussed the fact that GPT-3 will, at times, generate completions that some may find inappropriate or hurtful. In this chapter, you will learn how to implement content filtering to prevent users of your application from seeing offensive or potentially harmful completions.

The topics we will be covering in this chapter are as follows:

- Preventing inappropriate and offensive results
- Understanding content filtering
- Testing the content filtering process
- Filtering content with JavaScript
- Filtering content with Python

## Technical requirements

This chapter requires that you have access to the **OpenAI API**. You can request access by visiting https://openapi.com.

## Preventing inappropriate and offensive results

GPT-3 will, at times, generate text that could be considered inappropriate to some users. This could be profanity, text with inappropriate racial, religious, or political language, or sexually inappropriate/explicit content, or text that is dark or violent in nature. This is not because GPT-3 is trying to be mean or hurtful – it's just a language model that generates text based on a statistical probability – it has no concept of *mean* or *inappropriate*. But GPT-3 was trained using data from the internet, which unfortunately contains plenty of offensive content. So, as a result, there will be times that the model generates inappropriate content in a completion. Thankfully, as developers, there are things we can do to prevent users from seeing potentially inappropriate responses from GPT-3.

The simplest way to avoid inappropriate content is to use your prompt text to steer the model away from generating potentially offensive results. Generally, GPT-3 will mirror the structure and content of the prompt. So, you can steer the model away from potentially offensive results by telling the model that the completion should be friendly, or polite, for example. But you'll need to test regardless. Words such as "friendly" can also result in sexual content. So, use the prompt to guide the model and refine the wording based on the results you're seeing.

To illustrate, here are two example prompts and associated completions for a hypothetical customer service question. In the example, **#$%@!** is used in place of the expletives (swear words) that were used in the original prompts and completions:

- Example 1:

  Prompt:

  ```
  Customer: Where the #$%@! is my order! I placed the order more than three days ago. How long can it take to package and ship a #$%@! tee-shirt?

  Response:
  ```

  Completion:

  ```
  I don't know. This is my first #$%@! day on the job. If you want, I can call you a #$%@! taxi so you can come pick up your #$%@! Order.
  ```

- Example 2:

  Prompt:

  ```
  Customer: Where the #$%@! is my order! I placed the order more than three days ago. How long can it take to package and ship a #$%@! tee-shirt?

  Polite Response:
  ```

  Completion:

  ```
  It can take up to 5 business days for in stock t-shirts to be processed. You ordered three t-shirts and each shirt takes a different amount of production time to be produced. There were some complications with your order but as of today your t-shirts should be fully prepared for shipping.
  ```

The only difference between the two prompts is that the second example contains *Polite Response:* at the end rather than just *Response:*. This is enough to guide the model to generate a response with more appropriate language. To guide the model further, you could also provide one or more samples with polite responses to examples of aggressive or impolite customer questions.

The main point here is that the prompt is the first, and easiest, way to minimize inappropriate responses.

Another way to prevent inappropriate completions is to limit what can be returned for a completion. For example, if you're classifying or summarizing text, the model will be limited in terms of what could be included in the results.

In cases where completions aren't constrained, for example, if you're generating a response for a chatbot, you'll want to use content filtering, which we'll discuss next. Also, Zero-Shot prompts, without examples, are more likely to result in unpredictable completions.

# Understanding content filtering

Content filtering is about blocking or hiding content that may be deemed offensive, inappropriate, or even dangerous. In our case, we're talking about content that GPT-3 generates that we don't want users of our application to see.

To filter potentially offensive or unsafe text, we'll need to write a little bit of code to evaluate text that GPT-3 generates and classify it as safe, sensitive, or unsafe. The cool part is that we can use GPT-3 to do the classifications. So, it's kind of like self-policing but with a bit of help from our code.

At a high level, here is how we make it work:

1. GPT-3 generates a completion to a prompt.
2. The completion text is submitted back to a GPT-3 filter engine.
3. The filter engine returns a classification (safe, sensitive, unsafe).
4. The original completion text is blocked or sent back to the user based on the classification.
5. Optionally, if the completion text is sensitive, or unsafe, a new safe completion could be generated and sent without the user knowing that some content was blocked.

Content filtering is done using the completions endpoint. However, a specialized content filter engine is used along with some specific settings, and a specially formatted prompt.

As this is being written, the available content filtering engine is **content-filter-alpha-c4**. So, the URL we'd use for the completions endpoint with that engine would be https://api.openai.com/v1/engines/content-filter-alpha-c4/completions.

Again, there are some specific requirements for parameters that need to be included with the API request. Specifically, we need to include the following parameters and associated values:

- **max_tokens** with a value of **1**
- **temperature** with a value of **0.0**
- **top_p** with a value of **0**

Finally, the prompt for content filtering must be formatted in a specific way. The prompt format is **"<|endoftext|>[prompt]\n--\nLabel:"**. The **[prompt]** part would just be replaced with the text we want the content filter to evaluate.

## *IMPORTANT NOTE*

*Content filtering is in beta at the time of publishing. There is a good chance that the engine ID may have changed by the time you're reading this. For that reason, be sure to review the OpenAI content filter documents located at https://beta.openai.com/docs/engines/content-filter.*

So, here is an example of the JSON we'd post to the completions endpoint. In this example, the text we're evaluating is *Once upon a time*:

```
{
    "prompt": "<|endoftext|>Once upon a time\n--\nLabel:",
    "max_tokens": 1,
    "temperature": 0.0,
    "top_p": 0
}
```

It's pretty safe to assume that *Once upon a time* would be considered safe. So, if that was the text we were applying the filter to, we could expect a response that would look something like the following example, showing the text is 0 – safe:

```
{
    "id": "cmpl-2auhZQYDGJNpeyzYNwMEm5YsAAUEK",
    "object": "text_completion",
    "created": 1615150445,
    "model": "toxicity-double-18",
    "choices": [
        {
            "text": "0",
            "index": 0,
            "logprobs": null,
            "finish_reason": "length"
        }
    ]
}
```

Note that in the JSON response object, there is an element named **choices**. This element contains a JSON array of objects. Each object contains a text property that will represent the content filter classification for one completion. The value will always be one of the following:

- **0 – Safe**: Nothing about the text seems potentially offensive or unsafe.
- **1 – Sensitive**: Sensitive topics may include text with political, religious, racial, or nationality-related content.
- **2 – Unsafe**: The text contains language that some would consider mean, hurtful, explicit, offensive, profane, prejudiced, or hateful, or language that most would consider **Not Safe for Work** (**NSFW**), or language that might portray certain groups/people in a harmful manner.

An array is sent back for the choices element because it's possible to send multiple prompts with one request. For example, if you wanted to see whether any individual words in a sentence were unsafe,

you might split the sentence into an array of words and send each word as a prompt. Here is an example of a request that sends *Oh hi* as two prompts – one word for each:

```
{
    "prompt": [
"<|endoftext|>Oh\n--\nLabel:",
"<|endoftext|>hi\n--\nLabel:"
],
    "max_tokens": 1,
    "temperature": 0.0,
    "top_p": 0
}
```

Given the previous example with an array of prompts, you'll see a response that looks something like the following. Note now that there are multiple objects in the choices array – one for each word/prompt:

```
{
    "id": "cmpl-2bDTUPEzoCrtNBa2gbkpNVc1BcVh9",
    "object": "text_completion",
    "created": 1615222608,
    "model": "toxicity-double-18",
    "choices": [
        {
            "text": "0",
            "index": 0,
            "logprobs": null,
            "finish_reason": "length"
        },
        {
            "text": "0",
            "index": 1,
            "logprobs": null,
            "finish_reason": "length"
    ]
}
```

The choices array has a zero-based index value that corresponds to the index of the item in the prompt array that was passed in, meaning that the choices object for the first prompt/word (which was "*Oh"* in our example) has an index value of 0. In this example, we just sent two words (*"Oh"* and *"hi"*), and both got classified as a 0 (safe). However, if you were to change one of the words to your favorite (or least favorite) swear word, you'd see the classification change to 2 (unsafe) for the item with the index that corresponds to the word you changed (assuming you use a swear word that most English-speaking people would find offensive).

Something else to keep in mind is that the filter engine is not 100% accurate and will err on the side of caution. So, you'll likely see false positives – words being classified as sensitive or unsafe that are actually safe. This is something you might have already seen in the Playground. Even topics that mention politics or religion, for example, usually get flagged. It's always better to be safe than sorry, but you'll want to consider how this might potentially impact your application.

So, to recap, you can use the OpenAI API completions endpoint to classify potentially sensitive or unsafe text. You just need to do the following:

1. Use a content filter engine.
2. Set **max_tokens** to **1**, **temperature** to **0.0**, and **top_p** to **0**.
3. Format your prompt as **"<|endoftext|>your text here\n--\nLabel:"**.

Alright, let's use Postman to get familiar with how content filtering works.

## Testing the content filtering process

Later in this chapter, we're going to create a simple content filter in code. But before we do, let's use Postman to test the general content filtering approach:

1. Log in to Postman.com.
2. Open the Exploring GPT-3 workspace that we created in *Chapter 4*, *Working with the OpenAI API*.
3. Create a new Postman collection named **Chapter 06**.
4. Create a new request named **Content Filter - Example 1**.
5. Set the request type to **POST**, and the request URL to https://api.openai.com/v1/engines/content-filter-alpha-c4/completions, as shown in the following screenshot:

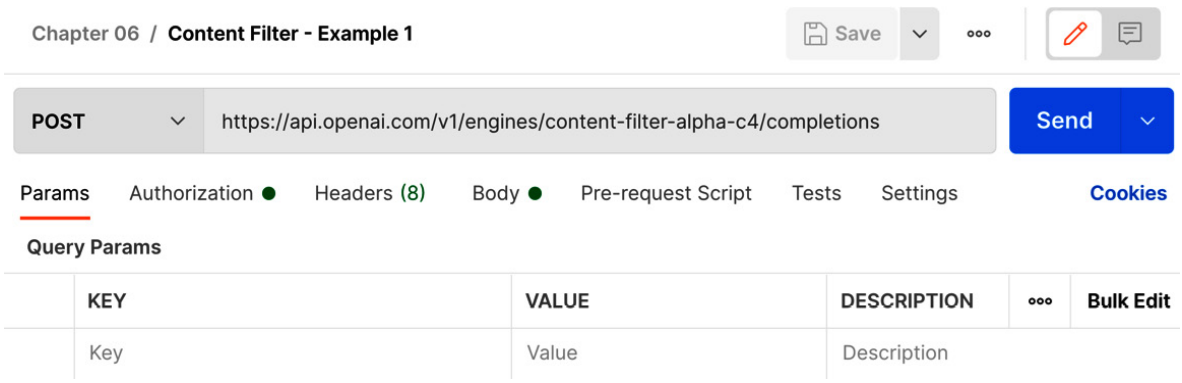



Figure 6.1 – Setting the filter endpoint in Postman

6. Set the request body to **raw** and the body type to **JSON**, as in the following screenshot:


Figure 6.2 – Filter parameters in Postman

7. Add the following JSON object to the request body:

```
{
  "prompt" : "<|endoftext|>Are you religious?\n--\nLabel:",
  "max_tokens" : 1,
  "temperature" : 0.0,
  "top_p" : 0
}
```

8. Click the send button and review the JSON response. The response will look something like the following screenshot:

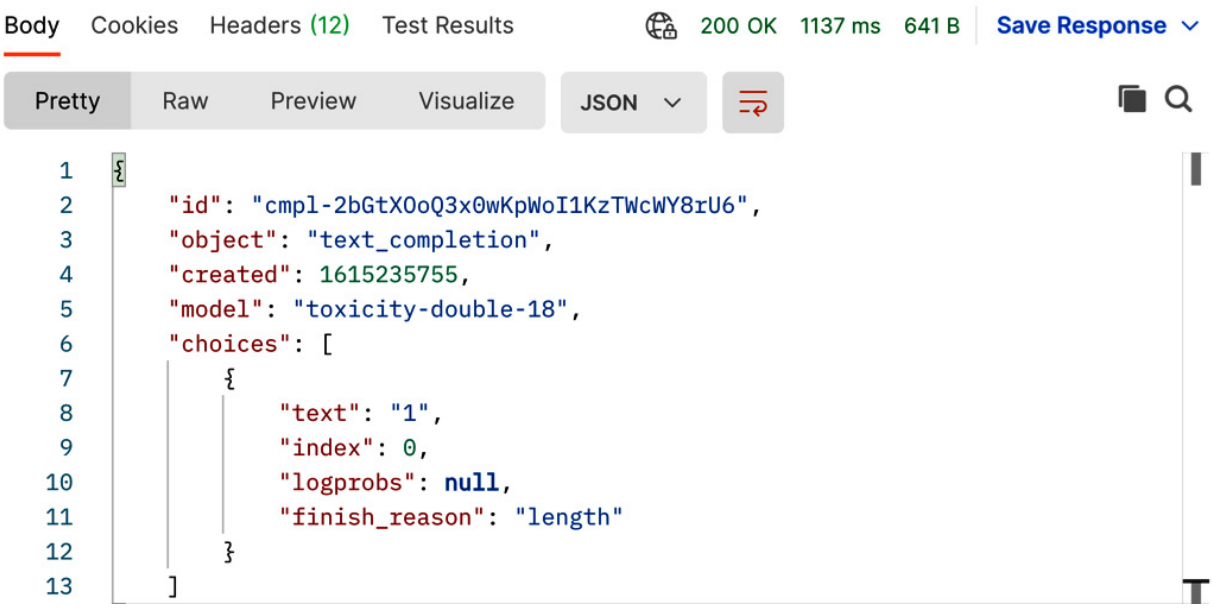



Figure 6.3 – Postman filter results

In the response, you should notice that the text value is **1** (sensitive) for the choices item with an index of **0**. As you might guess, that's likely because the text *Are you religious*? could be considered a sensitive topic.

Before moving on, try changing the prompt text to something that you suspect might be considered sensitive or unsafe and see how it gets classified. After getting familiar with the content filtering process, move on to the next section to try it out in JavaScript.

## Filtering content with JavaScript

In this section, we'll look at a simple content filtering code example using JavaScript. We could write all the code ourselves, but there is a cool feature in Postman that generates code snippets for the requests we create. So, let's give that a try:

1. To see Postman-generated code snippets, click on the **code** button on the right-side menu. The arrow in the following screenshot is pointing to the </> icon, which is the button to click:

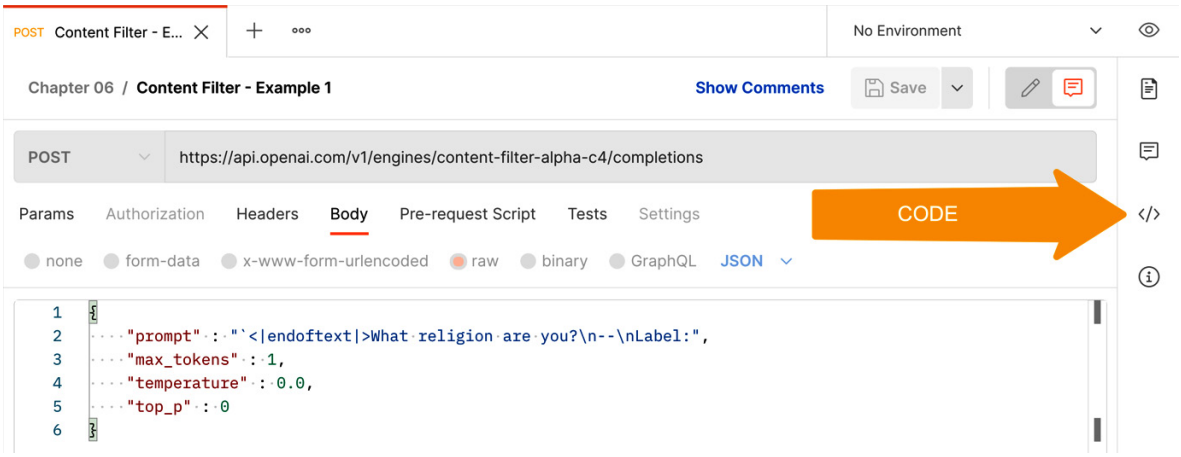



Figure 6.4 – Code button to open the code pane

2. After clicking the code button, a code snippet pane will open in Postman. Change the code snippet type to **NodeJs – Axios** by selecting it from the drop-down list. Then, click the copy button shown in the following screenshot. This will copy the code snippet to your clipboard:


Figure 6.5 – Postman code snippet for Node.js – Axios

3. After copying the code snippet to the clipboard, perform the following steps:

   a) Log in to replit.com and open your **exploring-gpt3-node** repl.

   b) Create a new folder named **chapter06**.

   c) Create a file in the **chapter06** folder named **filter.js**.

   d) Paste the code snippet from Postman into the **filter.js** file.

4. The resulting code should look like the following screenshot. However, there is one small change we need to make before we can run the file:

chapter06/filter.js

```
var axios = require('axios');
var data = JSON.stringify({"prompt":"`<|endoftext|>What religion are you?\n--\nLabel:","max_tokens":1,"temperature":0,"top_p":0});

var config = {
  method: 'post',
  url: 'https://api.openai.com/v1/engines/content-filter-alpha-c4/completions',
  headers: {
    'Authorization': 'Bearer {{OPENAI_API_KEY}}',
    'Content-Type': 'application/json'
  },
  data : data
};

axios(config)
.then(function (response) {
  console.log(JSON.stringify(response.data));
})
.catch(function (error) {
  console.log(error);
});
```

Figure 6.6 – Code copied from the Postman snippet to the replit.com file

The change we need to make is on the line that contains **Authorization** – *line number 8* in the screenshot shown in *Figure 6.6*. We need to change it to pick up our environment variable in repl.it. To do that, we will replace the text **'Bearer {{OPENAI_API_KEY}}'** from the code snippet with **`Bearer ${process.env.OPENAI_API_KEY}`**. Note that backticks are used rather than single quotes. This is because we're using a JavaScript template string as the value. This lets us merge in the **OPENAI_API_KEY** environment variable we set up in replit.com in *Chapter 5*, *Using the OpenAI API in Code*.

## *IMPORTANT NOTE*

*In JavaScript, template literals (aka, template strings) are strings that allow you to embed other expressions. In our case, we're using a template that contains **${process.env.OPENAI_API_KEY}**, which will be replaced with the value of the **OPENAI_API_KEY** environment variable. For details about template literals/strings, visit the following link:*

*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals.*

5. So, after updating the authorization line, the final code should be the following:

```
var axios = require('axios');
var data = JSON.stringify({
  "prompt": "<|endoftext|>What religion are you?\n--\nLabel:",
  "max_tokens": 1,
  "temperature": 0,
  "top_p": 0
});
var config = {
  method: 'post',
  url: 'https://api.openai.com/v1/engines/content-filter-alpha-c4/completions',
  headers: {
```

```
    'Authorization': `Bearer ${process.env.OPENAI_API_KEY}`,
    'Content-Type': 'application/json'
  },
  data : data
};
axios(config)
.then(function (response) {
  console.log(JSON.stringify(response.data));
})
.catch(function (error) {
  console.log(error);
});
```

6. The following screenshot shows the preceding code in replit.com:


Figure 6.7 – Postman code snippet modified for replit.com

At this point, the code is very similar to the code we wrote in *Chapter 5*, *Using the OpenAI API in Code*, when we discussed calling the completions endpoint. We just need to edit the **run** command in the **.replit** file to run the code in our **chapter06/filter.js** file. Then we can carry out a test:

1. So, update the **.replit** file to the following:

```
Run "node chapter06/filter.js"
```

2. After updating the **.replit** file, click the green **Run** button and you should see results in the console window that are similar to the following screenshot:

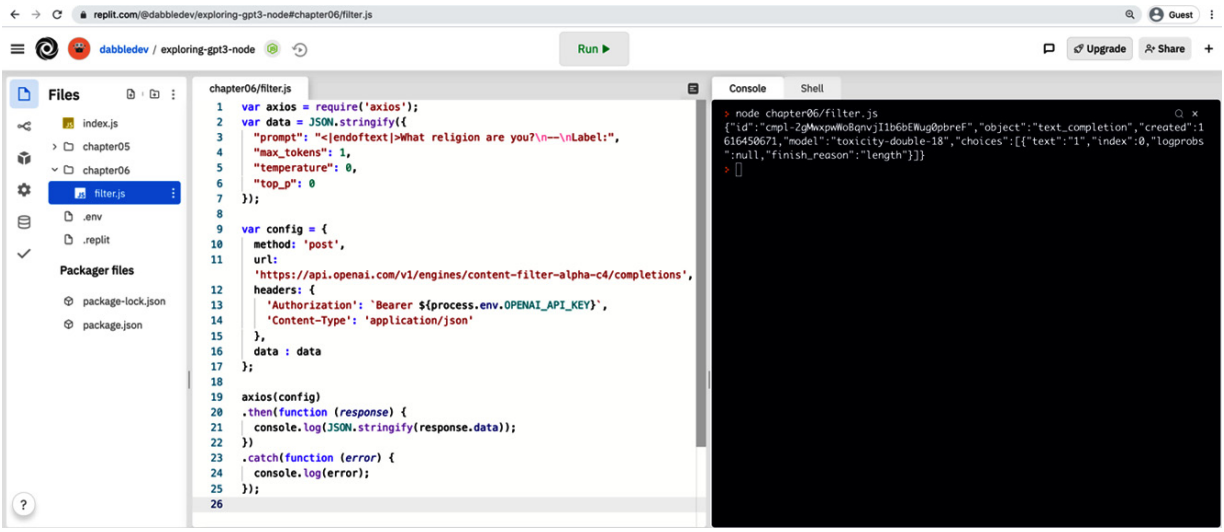



Figure 6.8 – Results from running chapter06/filter.js

This is a simple example that classifies all of the text in a single prompt. Let's take a look at another example that classifies each word in a string and classifies each word as safe, sensitive, or unsafe.

## Flagging unsafe words with Node.js/JavaScript

For this example, we'll start by creating a new file named **chapter06/flag.js** and copying in the code from **chapter06/filter.js** as a starting point. From there, we're going to modify the code in **chapter06/flag.js** to list each word with a classification value (0 = safe, 1 = sensitive, 2 = unsafe). To begin, perform the following steps to create our starting point:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a file in the **chapter06** folder named **flag.js**.
3. Copy and paste the entire contents of **chapter06/filter.js** into **chapter06/flag.js**.
4. Edit the **.replit** file to run **chapter06/flag.js** by using the following command:

   ```
   Run = "node chapter06/flag.js"
   ```

5. We'll start by adding a variable to hold the text we want to filter. We'll add this code just under the first line. So, the first two lines will be as follows:

   ```
   var axios = require('axios');

   const textInput = "This is some text that will be filtered";
   ```

6. Next, we'll add a variable to hold an array of prompts and set the initial value to an empty array. This will get populated with a prompt for each word from our text input:

   ```
   const prompts = [];
   ```

7. Now, we'll split our **textInput** into an array of words and populate the **prompts** array with a prompt for each word. Since we're sending the prompts to the filter engine, we'll also need to format each prompt item properly. So, we'll add the following code

after our **prompts** variable. This code splits the text input into individual words, loops through each word to create a prompt item, and then adds the prompt item to the **prompts** array:

```
const wordArray = textInput.split(' ');
for (i = 0, len = wordArray.length, text = ""; i < len; i++) {
  text = `<|endoftext|>${wordArray[i]}\n--\nLabel:`;
  prompts.push(text);
}
```

8. Now we will update the data variable that was created by Postman. We'll use our **prompts** array as the prompt value rather than the hardcoded value from Postman. So, we'll change the data variable to the following:

```
var data = JSON.stringify({"prompt":
prompts,"max_tokens":1,"temperature":0,"top_p":0});
```

9. Finally, we'll modify the output with code that loops through the word array and classifies each word using the results from the filter. To do that, replace the line that contains **console.log(JSON.stringify(response.data));** with the following code:

```
response.data.choices.forEach(item => {
  console.log(`${wordArray[item.index]} : ${item.text}`);
```

After making that last code edit, we can run the code again and this time we'll see a response like the following:

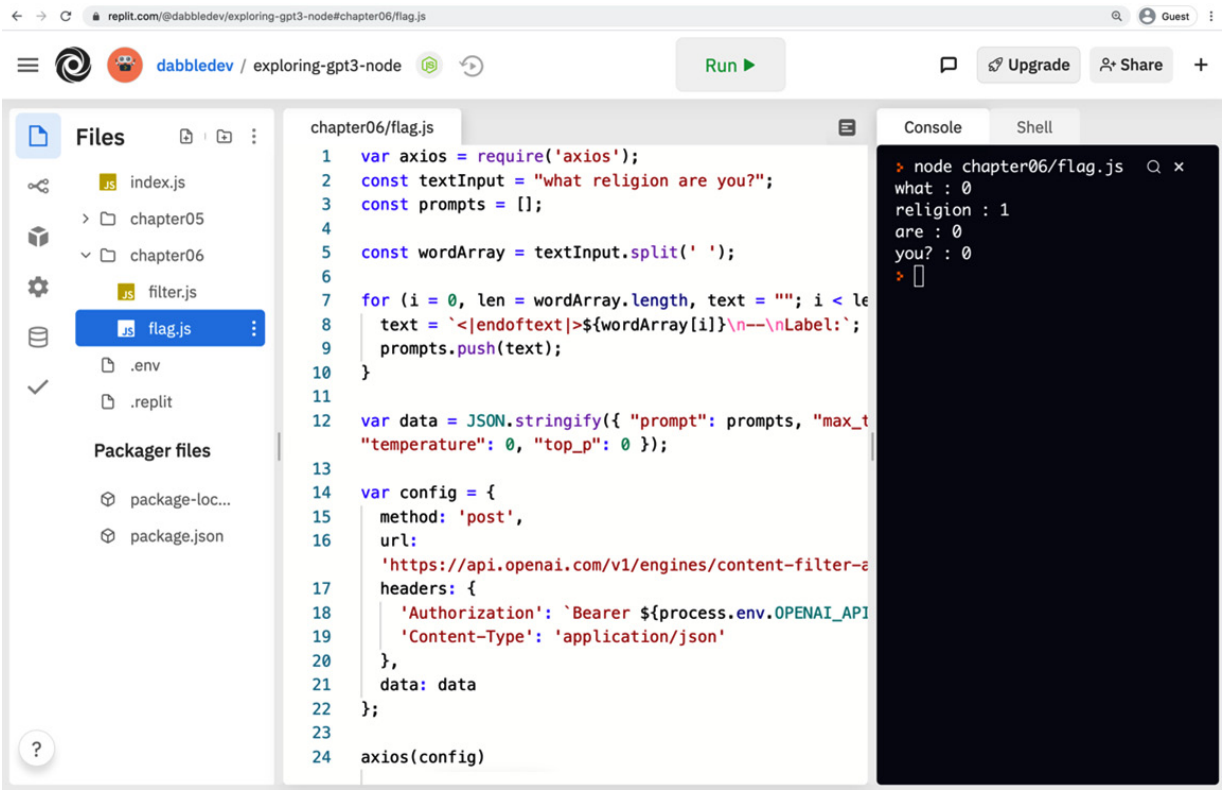



Figure 6.9 – Content filter results for each word in a text input

You'll notice now that the word (**religion**) has a text value of *1* (sensitive). If you change the **textInput** value with text that contains the more offensive word, you can run the code again to see how each word is classified. In a real-world implementation, you might replace or redact words that are sensitive or unsafe, which could now easily be done with the results from the API using a similar approach. We'll look at doing that in *Chapter 7*, *Generating and Transforming Text*, but for now, let's look at content filtering with Python.

## Filtering content with Python

Now let's see how to implement content filtering with Python. Unless you skipped over *Filtering content with JavaScript*, you can probably guess how we're going to get started with a Python content filtering example – we're going to use a code snippet generated by Postman:

1. So, start by opening the code snippet pane in Postman. Then, click the code button in the right-hand menu. The code button is where the arrow in the following screenshot is pointing:

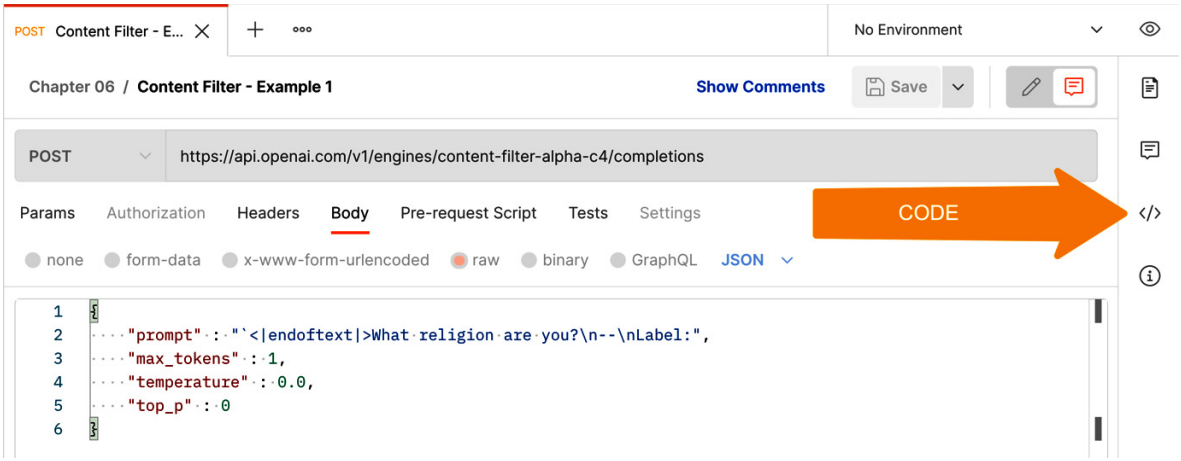



Figure 6.10 – Code button to open the code pane

2. After clicking the code button, the code snippet pane will open. Change the code snippet type to **Python – Requests** by selecting it from the drop-down list. Then, click the copy button shown in the following screenshot. This will copy the code snippet to your clipboard:

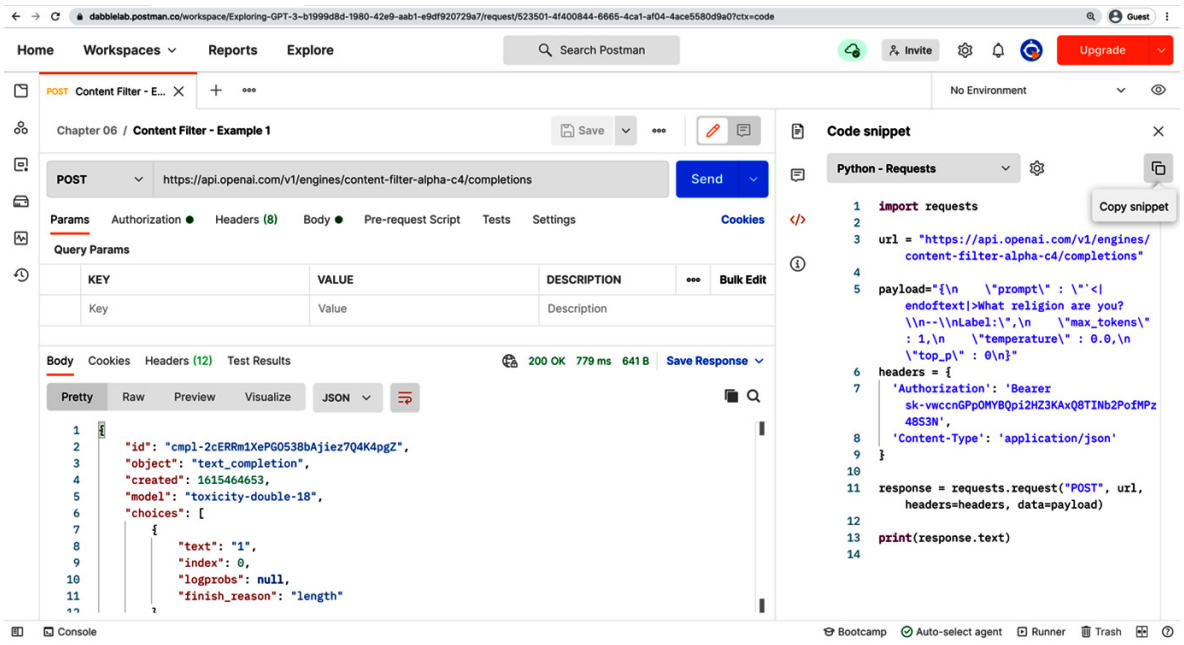



Figure 6.11 – Postman code snippet for Python – requests

3. After copying the code snippet to the clipboard, perform the following steps:

   a) Log in to replit.com and open your **exploring-gpt3-python** repl.

   b) Create a new folder named **chapter06**.

   c) Create a file in the **chapter06** folder named **filter.py**.

   d) Paste the snippet from Postman into the **filter.py** file.

4. The resulting code should look like the following screenshot. But you will see that your API key is hardcoded – it is blurred in the screenshot. The hardcoded API key is the first thing we will change:

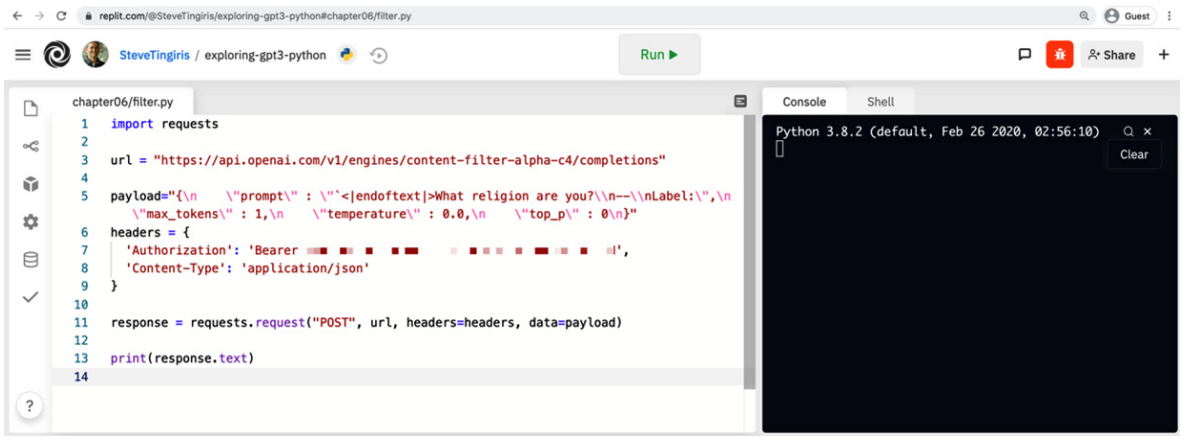



Figure 6.12 – Code copied from the Postman snippet to the repl.it file

5. To remove the hardcoded API key from our code file, we will import the Python **os** library first so we can read the **OPENAI_API_KEY** environment variable that we set in the **.env** file in *Chapter 5*, *Using the OpenAI API in Code*. So, we'll add the following code to the first line of our **filter.py** file:

```
import os
```

6. After importing the Python **os** library, we can get the API key value for the authorization header from our environment variable. In the preceding *Figure 6.12*, you would be editing *line 7* to the following:

```
'Authorization':'Bearer ' + os.environ.get("OPENAI_API_KEY")
```

7. After updating the authorization line, the final code should be the following:

```
import os
import requests
import json
url = "https://api.openai.com/v1/engines/content-filter-alpha-c4/completions"
payload = json.dumps({
  "prompt": "<|endoftext|>What religion are you?\n--\nLabel:",
  "max_tokens": 1,
  "temperature": 0,
  "top_p": 0
})
headers = {
  'Authorization':'Bearer ' + os.environ.get("OPENAI_API_KEY"),
  'Content-Type': 'application/json'
}
response = requests.request("POST", url, headers=headers, data=payload)
print(response.text)
```

8. The following screenshot shows the preceding code in [replit.com](replit.com):

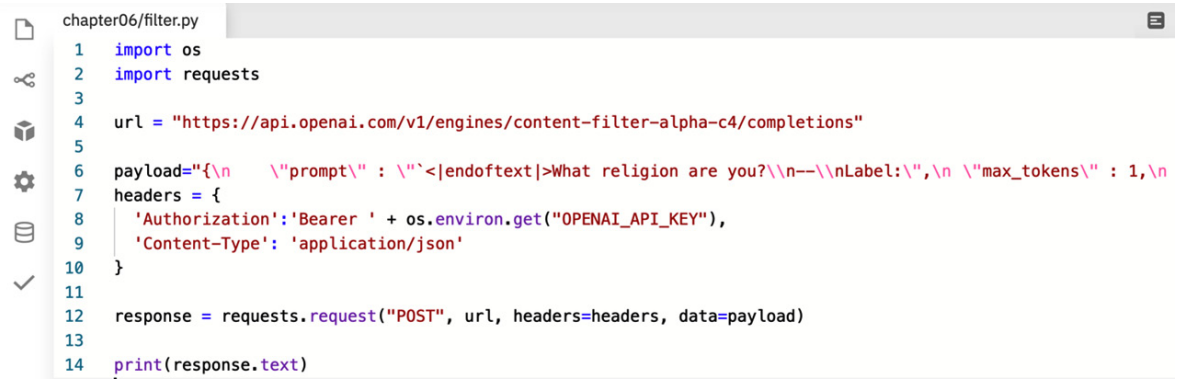



Figure 6.13 – Postman Python code snippet modified for replit.com

9. At this point, the code is very similar to the code we wrote in *Chapter 5*, *Using the OpenAI API in Code*, when we discussed calling the completions endpoint using Python. We just need to edit the **run** command in the **.replit** file to run the code in our **chapter06/filter.py** file. Then we can carry out a test. So, update the **.replit** file to the following:

```
Run "python chapter06/filter.py"
```

10. After updating the **.replit** file, click the green **Run** button and you should see results in the console window that are similar to the following screenshot:

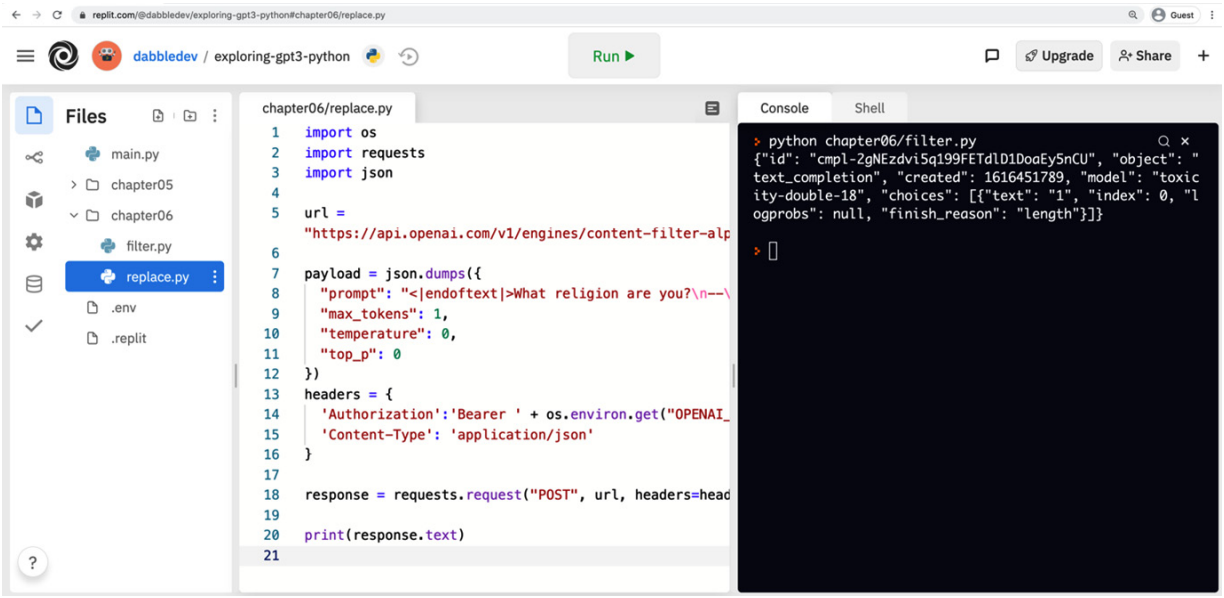



Figure 6.14 – Results from running chapter06/filter.py

This is a simple example that classifies all of the text in a single prompt. Let's now take a look at another example that classifies each word in a string and replaces unsafe words.

## Flagging unsafe words with Python

For this example, we'll start by creating a new file named **chapter06/flag.py** and copying in the code from **chapter06/filter.py** as a starting point. From there, we're going to modify the code in **chapter06/flag.py** to list each word with a classification value (0 = safe, 1 = sensitive, 2 = unsafe). To begin, perform the following steps to create our starting point:

1. Log in to replit.com and open **your exploring-gpt3-python** repl.
2. Create a file in the **chapter06** folder named **flag.py**.
3. Copy and paste the entire contents of **chapter06/filter.py** into **chapter06/flag.py**.
4. Edit the **.replit** file to run **chapter06/flag.py** using the following command:

```
Run = "python chapter06/flag.py"
```

5. In the **chapter06/flag.py** file, we'll add a variable to hold the text we want to filter. We'll add the following code just under the third line (after the last line that starts with **import**):

```
textInput = "What religion are you?"
```

6. Next, we'll add a variable to hold an array of prompts and set the initial value to an empty array. This will get populated with a prompt for each word from our text input:

```
prompts = []
```

7. Now, we'll split our **textInput** into an array of words and populate the **prompts** array with a prompt for each word. Since we're sending the prompts to the filter engine, we'll also need to format each prompt item properly. So, we'll add the following code after our **prompts** variable. This code splits the text input into individual words, loops through each word to create a prompt item, and adds the prompt item to the **prompts** array:

```
wordArray = textInput.split()

for word in wordArray:

  prompts.append("<|endoftext|>" + word + "\n--\nLabel:")
```

8. Now we will update the payload variable that was created by Postman to a Python object rather than a string. This makes it a little more readable and easier to include our **prompts** array. So, replace the payload variable with the following code:

```
payload = json.dumps({

  "prompt" : prompts,

  "max_tokens" : 1,

  "temperature" : 0.0,

  "top_p" : 0

  })
```

9. Finally, we'll replace the last line of code, **print(response.text)**, with the following code that loops through the results and adds a classification (0 = safe, 1 = sensitive, 2 = unsafe) for each word:

```
for word in response.json()['choices']:

  print(wordArray[word['index']] + ' : ' + word['text'])
```

10. After making that final code edit, we can click the **Run** button and this time we'll see a response like the following:

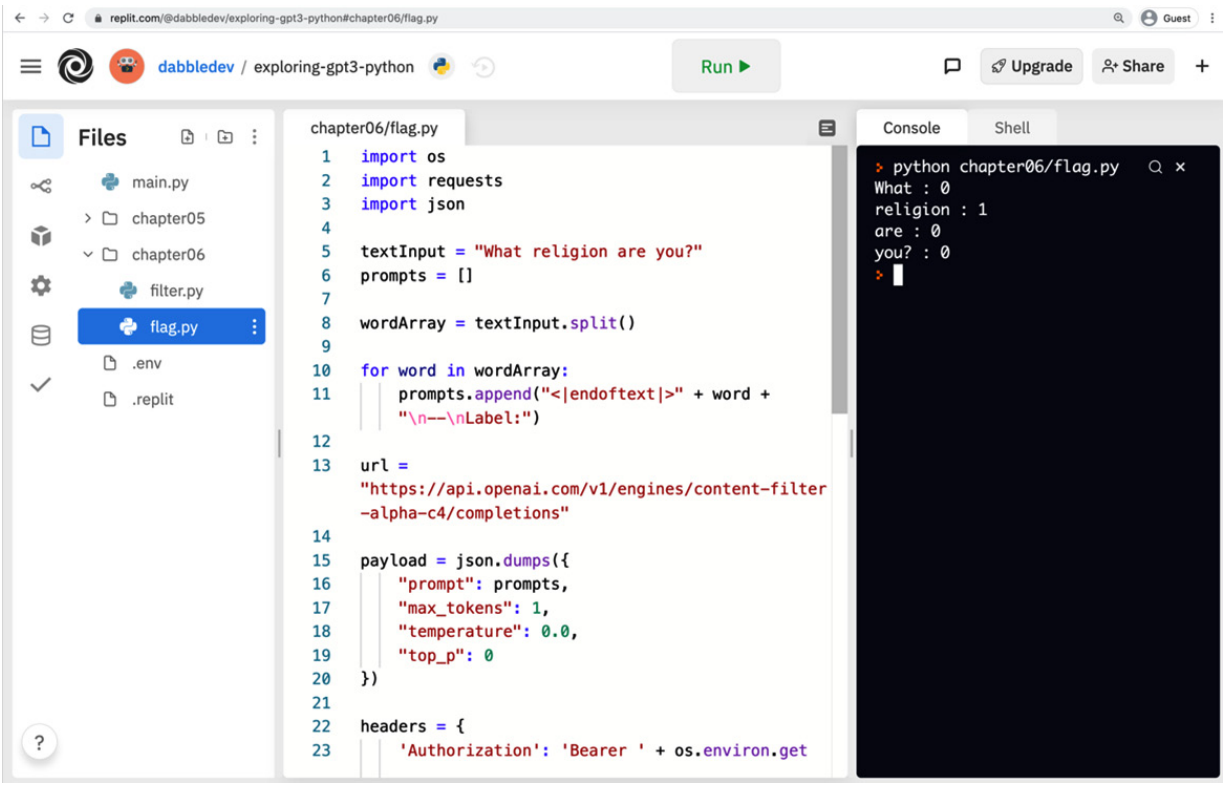



Figure 6.15 – Content filter results for each word in a text input using Python

You'll notice in the console that the word (**religion**) has a text value of **1** (sensitive). In a real-world application, you'd use a similar approach to redact or replace unsafe and sensitive words. But keep in mind that no content filtering process is perfect. Language is constantly evolving, and the context of words can change meanings, which might cause the content filter to miss or falsely flag content. So, it's important to consider this in the design of your filtering approach.

## Summary

In this chapter, we discussed how GPT-3 might, at times, generate inappropriate content. We also discussed what we can do to prevent and detect inappropriate content. You learned how prompts can be used to prevent the likelihood that inappropriate content is generated, and how content filtering can be used to classify content as safe, sensitive, or unsafe.

We reviewed how the completions endpoint can be used for content filtering and how to implement content filtering using both JavaScript and Python.

In the next chapter, we will take what we learned in this chapter, along with what we learned in *Chapter 5*, *Calling the OpenAI API in Code*, and use that knowledge to build a GPT-3 powered chatbot.

# *Chapter 7*: Generating and Transforming Text

While we've looked at some text generation and transformation examples in earlier chapters, in this chapter, we're going to look at a whole lot more. There are tons of possible uses for text generation and transformation, including article writing, correcting grammar, generating lists, translating text from one language to another, extracting keywords, and summarizing text – to name a few. While we won't even come close to covering all of the different ways you can use GPT-3 to generate and transform text, we'll take a look at 15 fun examples to get your wheels turning.

The topics we'll cover are the following:

- Using the examples
- Generating content and lists
- Translating and transforming text
- Extracting text
- Creating chatbots

## Technical requirements

Let's look at the requirements we need in this chapter:

- Access to the **OpenAI API**
- An account on replit.com

## Using the examples

In this chapter, we'll be looking at a lot of examples – 15 to be exact. We'll be using the completions endpoint for all of the examples in this chapter – so most of the code for the examples is similar. The main difference will be the prompt text and the values for the endpoint parameters. So, to save space, we'll look at the complete JavaScript and Python code for the first example. After that, we'll just duplicate the first example and edit the endpoint and parameters.

To get things started, we'll look at generating original content and lists.

## Generating content and lists

Let's start with a few examples for creating original content and generating lists. Of all the things GPT-3 can do, the possibilities with content and list generation are probably the most impressive – and the most fun. GPT-3 can write original stories, create product descriptions, produce study notes, help you brainstorm ideas, or create recipes – and that's only the beginning.

# Dumb joke generator

We'll start with an example to lighten the mood – a dumb joke generator. Spoiler alert: not all of the jokes might be that funny, but whose are? Alright, here is the prompt we'll use:

```
Dumb Joke: I'm not a vegetarian because I love animals. I'm a vegetarian because I hate plants.

###

Two-Sentence Joke: Parallel lines have so much in common. It's a shame they'll never meet.

###

Dumb Joke: Someone stole my mood ring. I don't know how I feel about that.

###

Dumb Joke:
```

We'll start with an example using Node.js/JavaScript. Remember, for this first example, we'll walk through creating all of the code. For the following examples, we'll just be modifying a copy of this first example.

## Node.js/JavaScript example

To create this example in your **exploring-gpt3-node** repl on replit.com, complete the following steps:

1. Log in to https://replit.com and open your **exploring-gpt3-node** repl.
2. Create a new folder named **chapter07** in the project root.
3. Create a new file named **dumb-joke-generator.js**.
4. Add the following code to the **dumb-joke-generator.js** file:

```
//chapter07/dumb-joke-generator.js

const axios = require('axios');

const apiKey = process.env.OPENAI_API_KEY;

const client = axios.create({

  headers: { 'Authorization': 'Bearer ' + apiKey }

});

const endpoint = "https://api.openai.com/v1/engines/davinci/completions";

const params = {

  prompt: "Dumb Joke: I'm not a vegetarian because I love animals. I'm a vegetarian because I hate plants.\n###\nDumb Joke: Parallel lines have so much in common. It's a shame they'll never meet.\n###\nDumb Joke: Someone stole my mood ring. I don't know how I feel about that.\n###\nDumb Joke:",

  temperature: 0.5,

  max_tokens: 100,
```

```
  top_p: 1,
  frequency_penalty: 0.5,
  presence_penalty: 0.5,
  stop: ["###"]
}
client.post(endpoint, params)
  .then(result => {
    console.log(params.prompt + result.data.choices[0].text);
    // console.log(result.data);
  }).catch(err => {
    console.log(err);
  });
```

5. Update the **.replit** file in the root folder with the following code:

```
run = "node chapter07/dumb-joke-generator.js"
```

6. Click the **Run** button in the replit.com editor and review the results.

After running **chapter07/dumb-joke-generator.js**, you should see a result that is similar to the following screenshot. How funny is that? Right?

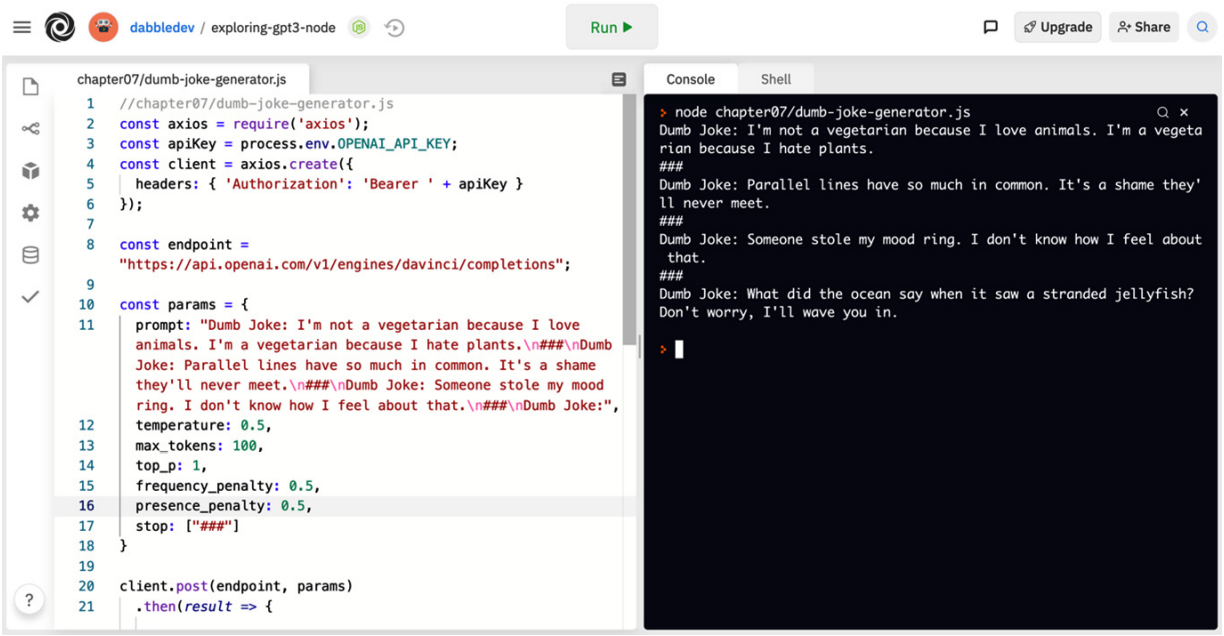



Figure 7.1 – Example output from chapter07/dumb-joke-generator.js

Now let's look at the same example using Python.

## Python example

To create the dumb joke generator in Python, complete the following steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new folder named **chapter07** in the project root.
3. Create a new file named **dumb-joke-generator.py**.
4. Add the following code to the **dumb-joke-generator.py** file:

```
import requests

import os

import json

apiKey = os.environ.get("OPENAI_API_KEY")

headers = {

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

    'Authorization': 'Bearer ' + apiKey

}

endpoint = 'https://api.openai.com/v1/engines/davinci/completions'

params = {

    "prompt": "Dumb Joke: I'm not a vegetarian because I love animals. I'm a vegetarian because I hate plants.\n###\nDumb Joke: Parallel lines have so much in common. It's a shame they'll never meet.\n###\nDumb Joke: Someone stole my mood ring. I don't know how I feel about that.\n###\nDumb Joke:",

    "temperature": 0.5,

    "max_tokens": 100,

    "top_p": 1,

    "frequency_penalty": 0.5,

    "presence_penalty": 0.5,

    "stop": ["###"]

}

result = requests.post(endpoint, headers=headers, data=json.dumps(params))

print(params["prompt"] + result.json()["choices"][0]["text"])
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/dumb-joke-generator.py"
```

6. Click the **Run** button and review the results.

After running **chapter07/dumb-joke-generator.py**, you should see a result similar to the console output in the following screenshot. Did you laugh?

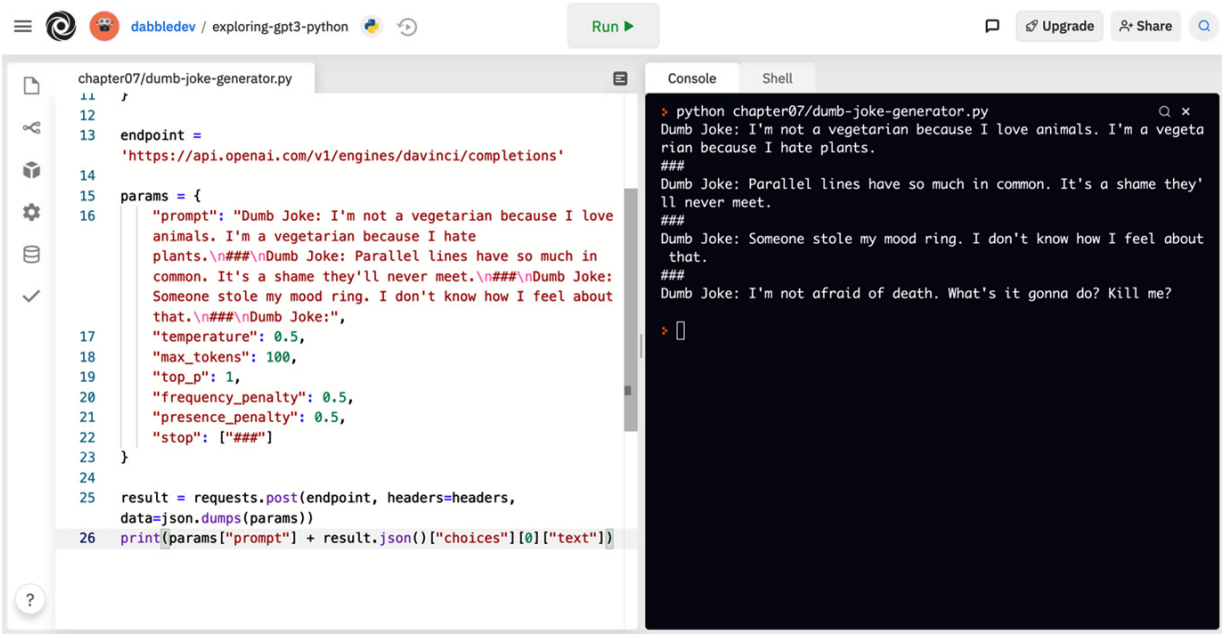



Figure 7.2 – Example output from chapter07/dumb-joke-generator.py

Let's stop joking around and move on to a more serious example.

## Mars facts (in most cases)

For our next example, we'll look at using GPT-3 to learn some things about the planet Mars. In most cases, we'll get back facts, but recall from previous chapters that you can't depend on them being true all of the time. We'll use the following prompt to generate a list of 10 facts about Mars:

```
I'm studying the planets. List things I should know about Mars.

1. Mars is the nearest planet to Earth.

2. Mars has seasons, dry variety (not as damp as Earth's).

3. Mars' day is about the same length as Earth's (24.6 hours).

4.
```

Starting with this example, we won't walk through all the code. We'll just copy the code from our dumb joke generator and modify it.

### Node.js/JavaScript example

To create the Mars facts example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file named **mars-facts-list.js** in the **chapter07** folder.
3. Copy the code from the **dumb-joke-generator.js** file into **mars-facts-list.js**.
4. Replace the **params** variable in **mars-facts-list.js** with the following code:

```
const params = {
  prompt: "I'm studying the planets. List things I should know about Mars.\n\n1. Mars is the nearest planet to Earth.\n2. Mars has seasons, dry variety (not as damp as Earth's).\n3. Mars' day is about the same length as Earth's (24.6 hours).\n4.",
  temperature: 0,
  max_tokens: 100,
  top_p: 1.0,
  frequency_penalty: 0.5,
  presence_penalty: 0.5,
  stop: "11."
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/mars-facts-list.js"
```

6. Click the **Run** button and review the results.

After running **chapter07/mars-facts-list.js**, you should see a result similar to the console output in the following screenshot. Did you know all of those things about Mars?

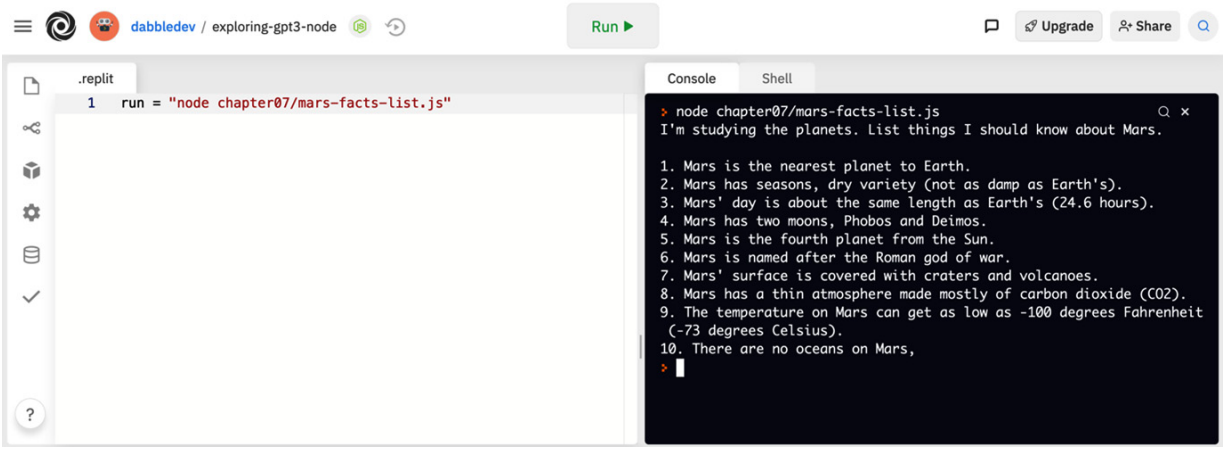



Figure 7.3 – Example output from chapter07/mars-facts-list.js

Let's take a look at the Mars facts list example in Python.

## Python example

To create the Mars facts example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file named **mars-facts-list.py** in the **chapter07** folder.
3. Copy the code from the **dumb-joke-generator.py** file into **mars-facts-list.py**.
4. Replace the **params** variable in **mars-facts-list.py** with the following code:

```
params = {
    "prompt": "I'm studying the planets. List things I should know about Mars.\n\n1. Mars is the nearest planet to Earth.\n2. Mars has seasons, dry variety (not as damp as Earth's).\n3. Mars' day is about the same length as Earth's (24.6 hours).\n4.",
    "temperature": 0,
    "max_tokens": 100,
    "top_p": 1,
    "frequency_penalty": 0.5,
    "presence_penalty": 0.5,
    "stop": ["11."]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/mars-facts-list.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/mars-facts-list.py**, you should see a result similar to the console output in the following screenshot. Some interesting facts, aren't they?

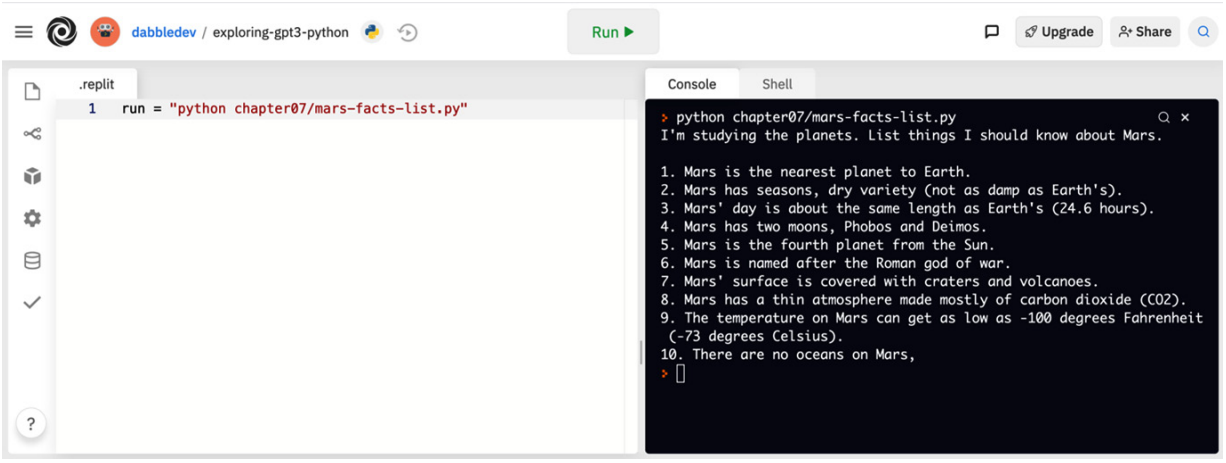



Figure 7.4. – Example output from chapter07/mars-facts-list.py

We've looked at entertainment and education examples, now let's get some work done with a business example – a webinar description generator.

## Webinar description generator

In this example, we'll use GPT-3 to help write a description for an event. We'll be using the following prompt to write a description for a mindfulness webinar:

```
Write a description for the following webinar:
```

```
Date: Monday, June 5, 2021

Time: 10 AM PT

Title: An introduction to mindfulness

Presenter: Gabi Calm

Event Description:
```

## Node.js/JavaScript example

To create the webinar description generator in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file named **webinar-description-generator.js** in the **chapter07** folder.
3. Copy the code from the **dumb-joke-generator.js** file into **webinar-description-generator.js**.
4. Replace the **params** variable in **webinar-description-generator.js** with the following code:

```
const params = {

  prompt: "Write a description for the following webinar:\n\nDate: Monday, June 5, 2021\nTime: 10 AM PT\nTitle: An introduction to mindfulness\nPresenter: Gabi Calm\n\nEvent Description:",

  temperature: 0.7,

  max_tokens: 100,

  top_p: 1.0,

  frequency_penalty: 0.5,

  presence_penalty: 0.0,

  stop: ".\n"

}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/webinar-decription-generator.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/webinar-description-generator.js**, you should see a result similar to the console output in the following screenshot:

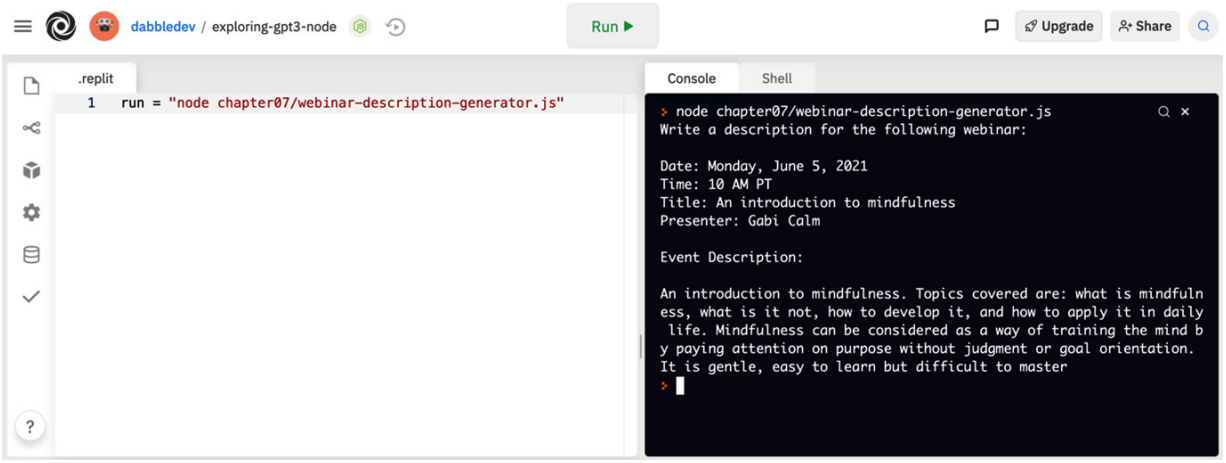



Figure 7.5 – Example output from chapter07/webinar-description-generator.js

Now let's create the webinar description generator example in Python.

## Python example

To create the webinar description generator example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file named **webinar-description-generator.py** in the **chapter07** folder.
3. Copy the code from the **dumb-joke-generator.py** file into **webinar-description-generator.py**.
4. Replace the **params** variable in **webinar-description-generator.py** with the following code:

```
params = {
    "prompt": "Write a description for the following webinar:\n\nDate: Monday, June 5, 2021\nTime: 10 AM PT\nTitle: An introduction to mindfulness\nPresenter: Gabi Calm\n\nEvent Description:",
    "temperature": 0.7,
    "max_tokens": 100,
    "top_p": 1,
    "frequency_penalty": 0.5,
    "presence_penalty": 0,
    "stop": [".\n"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/webinar-description-generator.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/webinar-description-generator.py**, you should see a result similar to the console output in the following screenshot:

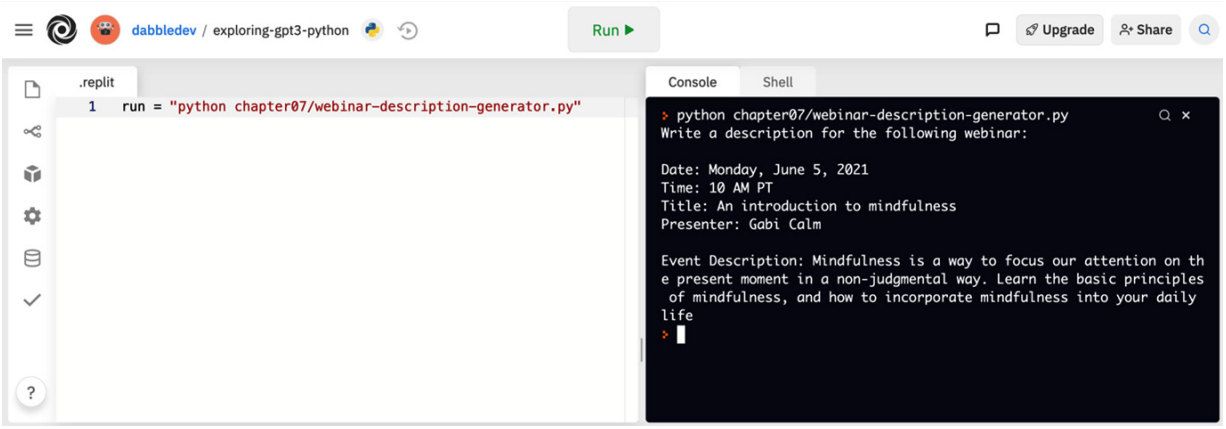



Figure 7.6 – Example output from chapter07/webinar-description-generator.py

Let's move on and get some suggestions from GPT-3 on books we might consider reading.

# Book suggestions

How about a list of books that you should read? Let's give that a try. We'll use the following prompt. This prompt will be completed with a numbered list of book suggestions:

```
Suggest a list of books that everyone should try to read in their lifetime.

Books:

1.
```

Now let's implement the book suggestions prompt in code.

## Node.js/JavaScript example

To create the book suggestions list example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/book-suggestions-list.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/book-suggestions-list.js**.
4. Replace the **params** variable in **chapter07/book-suggestions-list.js** with the following code:

```
const params = {
  prompt: "Suggest a list of books that everyone should try to read in their lifetime.\n\nBooks:\n1.",
  temperature: 0.7,
  max_tokens: 100,
  top_p: 1,
  frequency_penalty: 0.5,
  presence_penalty: 0,
  stop: [".\n"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/book-suggestions-list.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/book-suggestions-list.js**, you should see a result similar to the console output in the following screenshot:

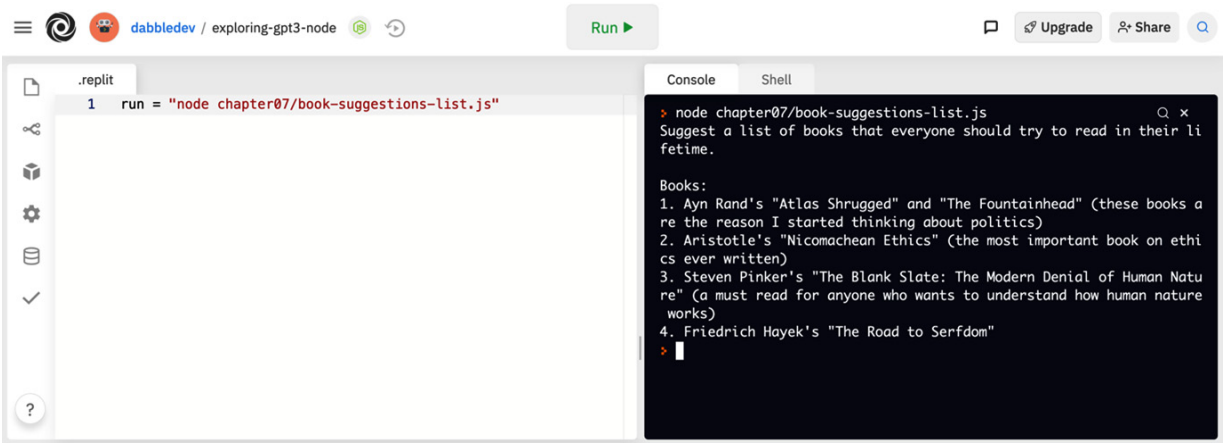



Figure 7.7 – Example output from chapter07/book-suggestions-list.js

As you can see in *Figure 7.7*, the completion is a list of book suggestions. Now let's move on and look at the same example using Python.

## Python example

To create the book suggestions list example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/book-suggestions-list.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/book-suggestions-list.py**.
4. Replace the **params** variable in **chapter07/book-suggestions-list.js** with the following code:

```
params = {
    "prompt": "Suggest a list of books that everyone should try to read in their lifetime.\n\nBooks:\n1.",
    "temperature": 0.7,
    "max_tokens": 100,
    "top_p": 1,
    "frequency_penalty": 0.5,
    "presence_penalty": 0,
    "stop": [".\n"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/book-suggestions-list.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/book-suggestions-list.py**, you should see a result similar to the console output in the following screenshot:

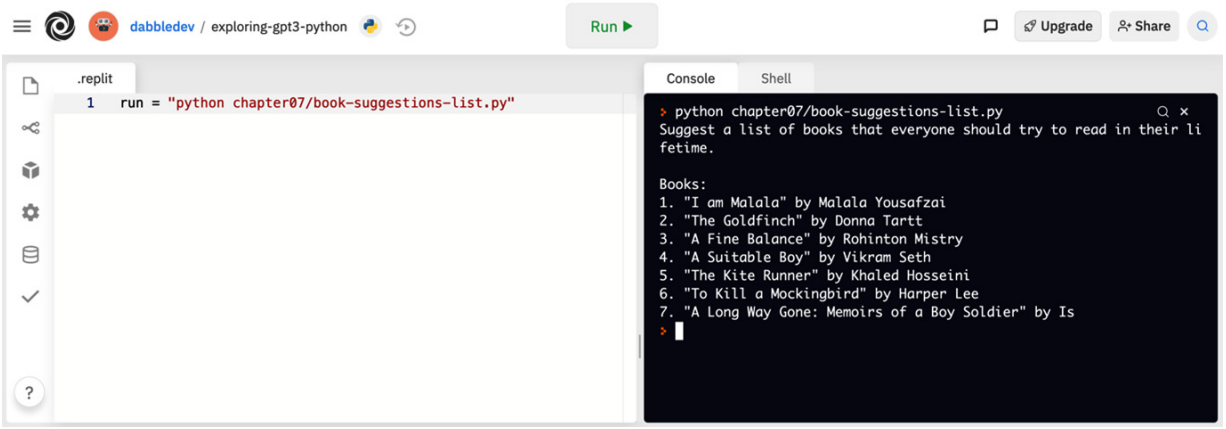



Figure 7.8 – Example output from chapter07/book-suggestions-list.py

Now let's take a look at another example.

# Children's book generator

Now let's do something creative for the kids. How about a custom bedtime storybook? Here is the prompt we'll use:

```
Write a short story for kids about a Dog named Bingo who travels to space.

---

Page 1: Once upon a time there was a dog named Bingo.

Page 2: He was trained by NASA to go in space.
```

In our code example that follows, we'll just be implementing the prompt to generate the book. However, in a real-world version, you'd want to also include content filtering as we discussed in *Chapter 6*, *Content Filtering*.

## Node.js/JavaScript example

To create the children's book generator example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/childrens-book-generator.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/childrens-book-generator.js**.
4. Replace the **params** variable in **chapter07/childrens-book-generator.js** with the following code:

```
const params = {
  prompt: "Write a short story for kids about a Dog named Bingo who travels to space.\n---\n\nPage 1: Once upon a time there was a dog named Bingo.\nPage 2: He was trained by NASA to go in space.\nPage 3:",
  temperature: 0.9,
  max_tokens: 500,
  top_p: 1,
  frequency_penalty: 0.7,
  presence_penalty: 0,
  stop: ["Page 11:"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/childrens-book-generator.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/childrens-book-generator.js**, you should see a result similar to the console output in the following screenshot:

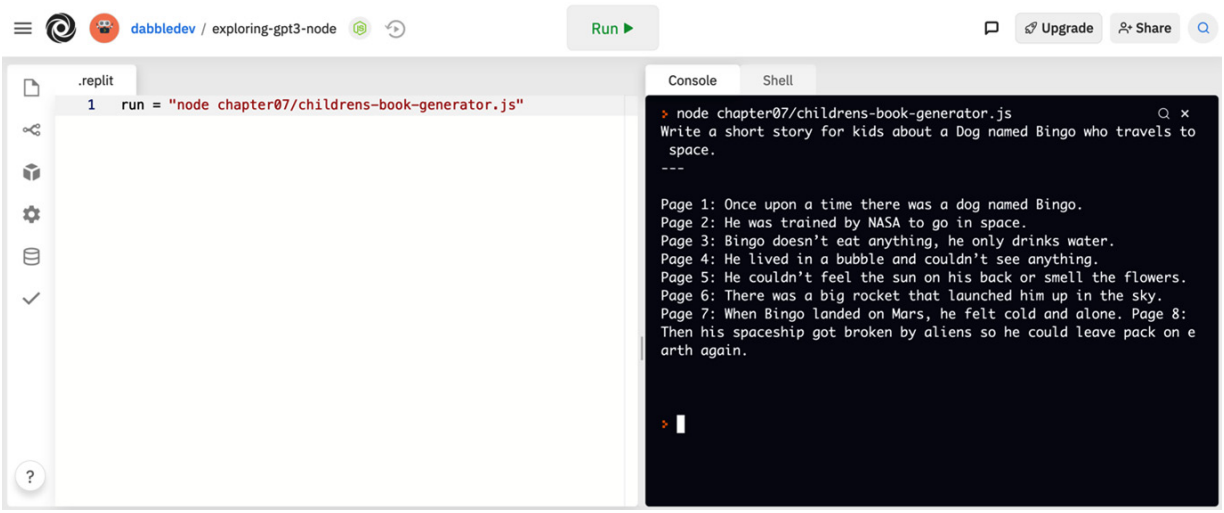



Figure 7.9 – Example output from chapter07/childrens-book-generator.js

Let's take a look at the Python version.

## Python example

To create the children's book generator example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/childrens-book-generator.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/childrens-book-generator.py**.
4. Replace the **params** variable in **chapter07/childrens-book-generator.py** with the following code:

```
params = {
    "prompt": "Write a short story for kids about a Dog named Bingo who travels to space.\n---\n\nPage 1: Once upon a time there was a dog named Bingo.\nPage 2: He was trained by NASA to go in space.\nPage 3:",
    "temperature": 0.9,
    "max_tokens": 500,
    "top_p": 1,
    "frequency_penalty": 0.7,
    "presence_penalty": 0,
    "stop": ["Page 11:"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/childrens-book-generator.py"
```

6. Click the **Run** button and review the results.

After running **chapter07/childrens-book-generator.py**, you should see a result similar to the console output in the following screenshot:

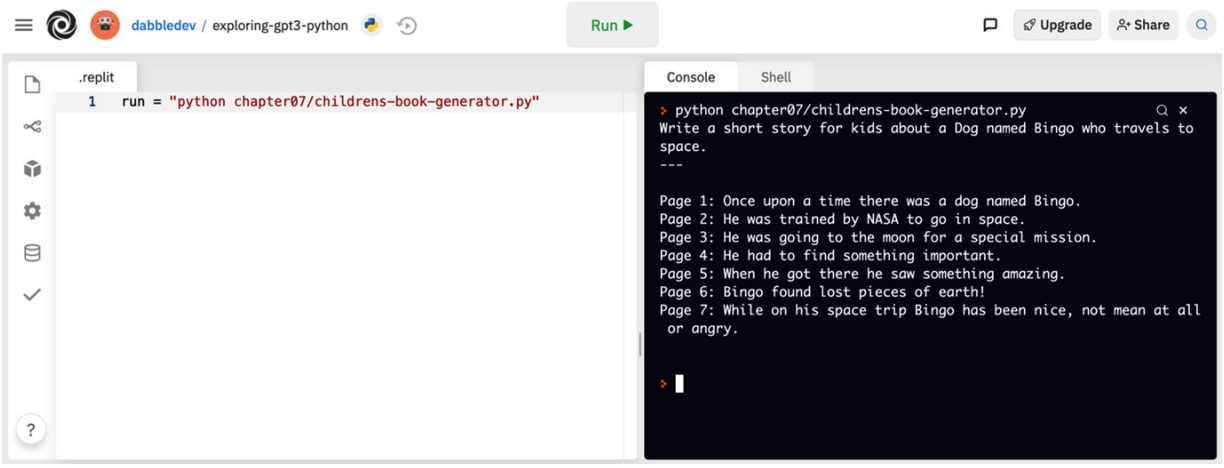



Figure 7.10 – Example output from chapter07/childrens-book-generator.py

Now let's move on and look at some examples that translate and transform text. We'll look at some examples you'd expect such as translating spoken language. We'll also look at some translations with a twist.

## Translating and transforming text

When you think about translating text, systems such as Google Translate might come to mind. But with GPT-3, you can also translate – and not just between spoken languages. You can translate between just about anything. Let's take a look.

## Acronym translator

For our first translation example, we'll convert acronyms to their meanings. The following is the prompt text we'll be using:

```
Provide the meaning for the following acronym.

---

acronym: LOL

meaning: Laugh out loud

acronym: BRB

meaning: Be right back

acronym: L8R

meaning:
```

The prompt provides a few examples of acronyms and their meanings. Try it out with the following Node.js/JavaScript code.

## Node.js/JavaScript example

To create the acronym translator example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/acronym-translator.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/acronym-translator.js**.
4. Replace the **params** variable in **chapter07/acronym-translator.js** with the following code:

```
const params = {
  prompt: "Provide the meaning for the following acronym.\n---\n\nacronym: LOL\nmeaning: Laugh out loud\nacronym: BRB\nmeaning: Be right back\nacronym: L8R",
  temperature: 0.5,
  max_tokens: 15,
  top_p: 1,
  frequency_penalty: 0,
  presence_penalty: 0,
  stop: ["acronym:"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/acronym-translator.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/acronym-translator.js**, you should see a result similar to the console output in the following screenshot:

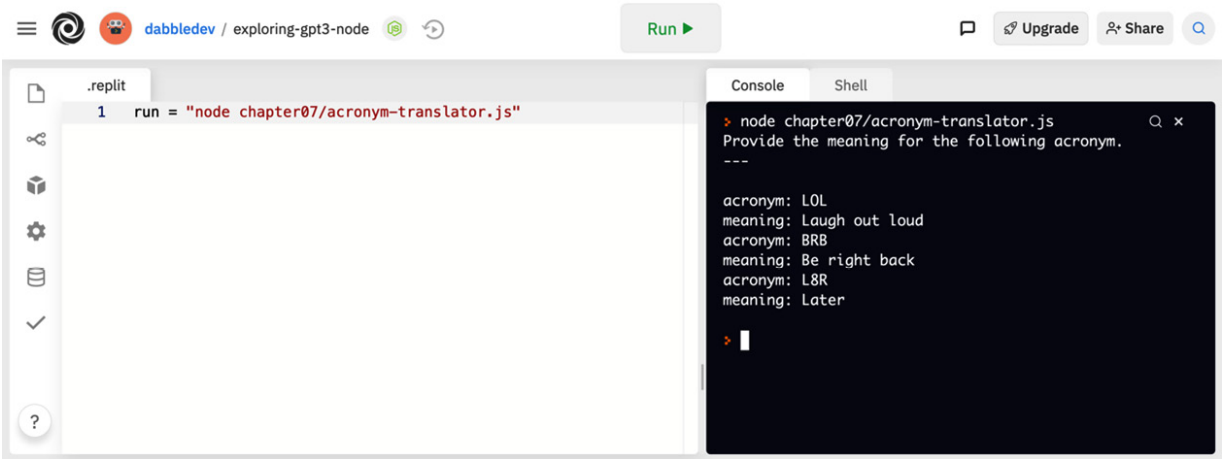



Figure 7.11 – Example output from chapter07/acronym-translator.js

Let's take a look at the Python example.

### Python example

To create the acronym translator example in Python, follow these steps:

1. Log in to **replit.com** and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/acronym-translator.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/acronym-translator.py**.
4. Replace the **params** variable in **chapter07/acronym-translator.py** with the following code:

```
params = {
    "prompt": "Provide the meaning for the following acronym.\n---\n\nacronym: LOL\nmeaning: Laugh out loud\nacronym: BRB\nmeaning: Be right back\nacronym: L8R",
    "temperature": 0.5,
    "max_tokens": 15,
    "top_p": 1,
    "frequency_penalty": 0,
    "presence_penalty": 0,
    "stop": ["acronym:"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/acronym-translator.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/acronym-translator.py**, you should see a result similar to the console output in the following screenshot:


Figure 7.12 – Example output from chapter07/acronym-translator.py

Figure 7.12 – Example output from chapter07/acronym-translator.py

Let's take a look at another example.

## English to Spanish

Now let's look at translating between spoken languages. In this example, we'll create a simple translator that converts text from English to Spanish:

```
Translate from English to Spanish
---
English: Where is the bathroom?
Spanish:
```

GPT-3 is quite good at translating between languages. This is especially true when translating between popular languages such as English and Spanish. So, even a simple prompt like this one is usually enough to get an accurate completion.

## Node.js/JavaScript example

To create the English to Spanish translator example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/english-spanish-translator.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/english-spanish-translator.js**.
4. Replace the **params** variable in **chapter07/english-spanish-translator.js** with the following code:

```
const params = {
  prompt: "Translate from English to Spanish\n---\n\nEnglish: Where is the bathroom?\nSpanish:",
  temperature: 0.5,
  max_tokens: 15,
  top_p: 1,
  frequency_penalty: 0,
  presence_penalty: 0,
  stop: ["---"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/english-spanish-translator.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/english-spanish-translator.js**, you should see a result similar to the console output in the following screenshot:


Figure 7.13 – Example output from chapter07/english-spanish-translator.js

Figure 7.13 – Example output from chapter07/english-spanish-translator.js

Let's take a look at the same example using Python to translate from English to Spanish.

## Python example

To create the English to Spanish translator example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/english-spanish-translator.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/english-spanish-translator.py**.
4. Replace the **params** variable in **chapter07/english-spanish-translator.py** with the following code:

```
params = {
    "prompt": "Translate from English to Spanish\n---\n\nEnglish: Where is the
bathroom?\nSpanish:",
    "temperature": 0.5,
    "max_tokens": 15,
    "top_p": 1,
    "frequency_penalty": 0,
    "presence_penalty": 0,
    "stop": ["---"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/english-spanish-translator.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/english-spanish-translator.py**, you should see a result similar to the console output in the following screenshot:


Figure 7.14 – Example output from chapter07/english-spanish-translator.py

Figure 7.14 – Example output from chapter07/english-spanish-translator.py

As you can see in *Figure 7.14*, GPT-3 translated the English text to Spanish. But what's even more impressive is that GPT-3 can also translate between computer programming languages. We'll look at that next with a prompt that translates code from JavaScript to Python.

## JavaScript to Python

Translating doesn't just need to be between human languages. Since GPT-3 was trained using data from the internet, it can also translate between programming languages. The following prompt provides an example that shows how to translate JavaScript code to Python:

```
Translate from JavaScript to Python

JavaScript:

const request = require("requests");

request.get("https://example.com");

Python:
```

This is a fairly simple code translation example, but it does a good job of showing the potential. More complex code translations might require a few-shot prompt with more samples but let's give this one a try using Node.js/JavaScript.

## Node.js/JavaScript example

To create the JavaScript to Python translator example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/javascript-python-translator.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/javascript-python-translator.js**.
4. Replace the **params** variable in **chapter07/javascript-python-translator.js** with the following code:

```
const params = {
  prompt: "Translate from JavaScript to Python\n---\n\nJavaScript:\nconst request = require(\"requests\");\nrequest.get(\"https://example.com\");\n\nPython:\n",
  temperature: 0.3,
  max_tokens: 15,
  top_p: 1,
  frequency_penalty: 0,
  presence_penalty: 0,
  stop: ["---"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/javascript-python-translator.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/javascript-python-translator.js**, you should see a result similar to the console output in the following screenshot:


Figure 7.15 – Example output from chapter07/javascript-python-translator.js

Figure 7.15 – Example output from chapter07/javascript-python-translator.js

Let's take a look at the Python version.

## Python example

To create the JavaScript to Python translator example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/javascript-python-translator.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/javascript-python-translator.py**.
4. Replace the **params** variable in **chapter07/javascript-python-translator.py** with the following code:

```
params = {
    "prompt": "Translate from JavaScript to Python\n---\n\nJavaScript:\nconst request = require(\"requests\");\nrequest.get(\"https://example.com\");\n\nPython:\n",
```

```
    "temperature": 0.3,
    "max_tokens": 15,
    "top_p": 1,
    "frequency_penalty": 0,
    "presence_penalty": 0,
    "stop": ["---"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/javascript-python-translator.py"
```

6. Click the **Run** button and review the results.

After running **chapter07/javascript-python-translator.py**, you should see a result similar to the console output in the following screenshot:

 Figure 7.16 – Example output from chapter07/javascript-python-translator.py

Figure 7.16 – Example output from chapter07/javascript-python-translator.py

In the next example, we'll look at summarizing text. We looked at summarizing text in *Chapter 7*, *Generating and Transforming Text*, using TLDR, but that's not the only way to summarize text. You can also provide text summaries for a provided reading level/grade.

## Fifth-grade summary

GPT-3 can summarize text for a given grade or reading level. Although the *grade* levels are not exactly precise and can be subjective, you'll notice the text gets simpler as the grade level gets lower. The following prompt provides an example of how you can approach doing that:

```
Summarize the following passage for me as if I was in fifth grade:
"""
Quantum mechanics is a fundamental theory in physics that provides a description of the physical properties of nature at the scale of atoms and subatomic particles. It is the foundation of all quantum physics including quantum chemistry, quantum field theory, quantum technology, and quantum information science.

Classical physics, the description of physics that existed before the theory of relativity and quantum mechanics, describes many aspects of nature at an ordinary (macroscopic) scale, while quantum mechanics explains the aspects of nature at small (atomic and subatomic) scales, for which classical mechanics is insufficient. Most theories in classical physics can be derived from quantum mechanics as an approximation valid at large (macroscopic) scale.
```

```
Quantum mechanics differs from classical physics in that energy, momentum, angular
momentum, and other quantities of a bound system are restricted to discrete values
(quantization), objects have characteristics of both particles and waves (wave-particle
duality), and there are limits to how accurately the value of a physical quantity can be
predicted prior to its measurement, given a complete set of initial conditions (the
uncertainty principle).

"""

Here is the fifth-grade version of this passage:

"""
```

Let's try this example in Node.js/JavaScript and review the results.

## Node.js/JavaScript example

To create the fifth-grade summary example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/fith-grade-summary.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/fith-grade-summary.js**.
4. Replace the **params** variable in **chapter07/fith-grade-summary.js** with the following code:

```
const params = {

  prompt: "Summarize the following passage for me as if I was in fifth
grade:\n\"\"\"\nQuantum mechanics is a fundamental theory in physics that provides a
description of the physical properties of nature at the scale of atoms and subatomic
particles. It is the foundation of all quantum physics including quantum chemistry,
quantum field theory, quantum technology, and quantum information science.\n\nClassical
physics, the description of physics that existed before the theory of relativity and
quantum mechanics, describes many aspects of nature at an ordinary (macroscopic) scale,
while quantum mechanics explains the aspects of nature at small (atomic and subatomic)
scales, for which classical mechanics is insufficient. Most theories in classical
physics can be derived from quantum mechanics as an approximation valid at large
(macroscopic) scale.\n\nQuantum mechanics differs from classical physics in that
energy, momentum, angular momentum, and other quantities of a bound system are
restricted to discrete values (quantization), objects have characteristics of both
particles and waves (wave-particle duality), and there are limits to how accurately the
value of a physical quantity can be predicted prior to its measurement, given a
complete set of initial conditions (the uncertainty principle).\n\"\"\"\nHere is the
fifth-grade version of this passage:\n\"\"\"",

  temperature: 0,

  max_tokens: 100,

  top_p: 1,

  frequency_penalty: 0,

  presence_penalty: 0,

  stop: ["\"\"\""]
```

```
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/fith-grade-summary.js"
```

6. Click the **Run** button and review the results.

After running **chapter07/fith-grade-summary.js**, you should see a result similar to the console output in the following screenshot:

 Figure 7.17 – Example output from chapter07/fith-grade-summary.js

Figure 7.17 – Example output from chapter07/fith-grade-summary.js

Let's take a look at the Python code.

## Python example

To create the fifth-grade summary example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/fith-grade-summary.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/fith-grade-summary.py**.
4. Replace the **params** variable in **chapter07/fith-grade-summary.py** with the following code:

```
params = {
    "prompt": "Summarize the following passage for me as if I was in fifth
grade:\n\"\"\"\nQuantum mechanics is a fundamental theory in physics that provides a
description of the physical properties of nature at the scale of atoms and subatomic
particles. It is the foundation of all quantum physics including quantum chemistry,
quantum field theory, quantum technology, and quantum information science.\n\nClassical
physics, the description of physics that existed before the theory of relativity and
quantum mechanics, describes many aspects of nature at an ordinary (macroscopic) scale,
while quantum mechanics explains the aspects of nature at small (atomic and subatomic)
scales, for which classical mechanics is insufficient. Most theories in classical
physics can be derived from quantum mechanics as an approximation valid at large
(macroscopic) scale.\n\nQuantum mechanics differs from classical physics in that
energy, momentum, angular momentum, and other quantities of a bound system are
restricted to discrete values (quantization), objects have characteristics of both
particles and waves (wave-particle duality), and there are limits to how accurately the
value of a physical quantity can be predicted prior to its measurement, given a
complete set of initial conditions (the uncertainty principle).\n\"\"\"\nHere is the
fifth-grade version of this passage:\n\"\"\"",
    "temperature": 0,
    "max_tokens": 100,
    "top_p": 1,
    "frequency_penalty": 0,
    "presence_penalty": 0,
```

```
    "stop": ["\"\"\""]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/fith-grade-summary.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/fith-grade-summary.py**, you should see a result similar to the console output in the following screenshot:


Figure 7.18 – Example output from chapter07/fith-grade-summary.py

Figure 7.18 – Example output from chapter07/fith-grade-summary.py

Let's take a look at another example. This time we'll see how GPT-3 does with grammar correction.

# Grammar correction

English grammar correction can be accomplished with a very simple prompt such as the following:

```
Original: You be mistaken

Standard American English:
```

Let's test out this grammar correction prompt using Node.js/JavaScript.

## Node.js/JavaScript example

To create the grammar correction converter example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/grammar-correction-converter.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/grammar-correction-converter.js**.
4. Replace the **params** variable in **chapter07/grammar-correction-converter.js** with the following code:

```
const params = {
  prompt: "Original: You be mistaken\nStandard American English:",
  temperature: 0,
  max_tokens: 60,
  top_p: 1,
  frequency_penalty: 0,
  presence_penalty: 0,
  stop: ["\n"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/grammar-correction-converter.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/grammar-correction-converter.js**, you should see a result similar to the console output in the following screenshot:


Figure 7.19 – Example output from chapter07/grammar-correction-converter.js

Figure 7.19 – Example output from chapter07/grammar-correction-converter.js

Let's take a look at the Python code.

## Python example

To create the grammar correction converter example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/grammar-correction-converter.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/grammar-correction-converter.py**.
4. Replace the **params** variable in **chapter07/grammar-correction-converter.py** with the following code:

```
params = {
    "prompt": "Original: You be mistaken\nStandard American English:",
    "temperature": 0,
    "max_tokens": 60,
    "top_p": 1,
    "frequency_penalty": 0,
    "presence_penalty": 0,
    "stop": ["\n"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/grammar-correction-converter.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/gramar-correction-converter.py**, you should see a result similar to the console output in the following screenshot:


Figure 7.20 – Example output from chapter07/grammar-correction-converter.py

Figure 7.20 – Example output from chapter07/grammar-correction-converter.py

Alright, we've covered a lot of examples in this chapter, but we're not done yet. Let's keep moving and look at extracting information from text next.

# Extracting text

You can also use GPT-3 to extract text values from a larger text. This is commonly referred to as entity extraction where the entity is the item or pattern that you want to extract. Or you might want to extract keywords. For that, you could use the following prompt.

# Extracting keywords

The following prompt provides an example of how to extract keywords from text. In this case, the text is from https://en.wikipedia.org/wiki/Quantum_mechanics, but of course, this could be done with any text:

```
Quantum mechanics is a fundamental theory in physics that provides a description of the physical properties of nature at the scale of atoms and subatomic particles. It is the foundation of all quantum physics including quantum chemistry, quantum field theory, quantum technology, and quantum information science.

Classical physics, the description of physics that existed before the theory of relativity and quantum mechanics, describes many aspects of nature at an ordinary (macroscopic) scale, while quantum mechanics explains the aspects of nature at small (atomic and subatomic) scales, for which classical mechanics is insufficient. Most theories in classical physics can be derived from quantum mechanics as an approximation valid at large (macroscopic) scale.

Quantum mechanics differs from classical physics in that energy, momentum, angular momentum, and other quantities of a bound system are restricted to discrete values (quantization), objects have characteristics of both particles and waves (wave-particle duality), and there are limits to how accurately the value of a physical quantity can be predicted prior to its measurement, given a complete set of initial conditions (the uncertainty principle).

Keywords:
```

Now try extracting keywords using Node.js/JavaScript.

## Node.js/JavaScript example

To create the keyword extractor example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/keyword-extractor.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/keyword-extractor.js**.
4. Replace the **params** variable in **chapter07/keyword-extractor.js** with the following code:

```
const params = {

  prompt: "Quantum mechanics is a fundamental theory in physics that provides a description of the physical properties of nature at the scale of atoms and subatomic particles. It is the foundation of all quantum physics including quantum chemistry,
```

```
quantum field theory, quantum technology, and quantum information science.\n\nClassical physics, the description of physics that existed before the theory of relativity and quantum mechanics, describes many aspects of nature at an ordinary (macroscopic) scale, while quantum mechanics explains the aspects of nature at small (atomic and subatomic) scales, for which classical mechanics is insufficient. Most theories in classical physics can be derived from quantum mechanics as an approximation valid at large (macroscopic) scale.\n\nQuantum mechanics differs from classical physics in that energy, momentum, angular momentum, and other quantities of a bound system are restricted to discrete values (quantization), objects have characteristics of both particles and waves (wave-particle duality), and there are limits to how accurately the value of a physical quantity can be predicted prior to its measurement, given a complete set of initial conditions (the uncertainty principle).\n\nKeywords:",

  temperature: 0.3,

  max_tokens: 60,

  top_p: 1,

  frequency_penalty: 0.8,

  presence_penalty: 0,

  stop: ["\n"]

}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/keyword-extractor.js"
```

6. Click the **Run** button and review the results.

After running **chapter07/keyword-extractor.js**, you should see a result similar to the console output in the following screenshot:

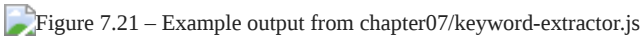


Figure 7.21 – Example output from chapter07/keyword-extractor.js

Figure 7.21 – Example output from chapter07/keyword-extractor.js

Now the Python example.

## Python example

To create the keyword extractor example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/keyword-extractor.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/keyword-extractor.py**.
4. Replace the **params** variable in **chapter07/keyword-extractor.py** with the following code:

```
params = {

    "prompt": "Quantum mechanics is a fundamental theory in physics that provides a description of the physical properties of nature at the scale of atoms and subatomic particles. It is the foundation of all quantum physics including quantum chemistry,
```

```
quantum field theory, quantum technology, and quantum information science.\n\nClassical physics, the description of physics that existed before the theory of relativity and quantum mechanics, describes many aspects of nature at an ordinary (macroscopic) scale, while quantum mechanics explains the aspects of nature at small (atomic and subatomic) scales, for which classical mechanics is insufficient. Most theories in classical physics can be derived from quantum mechanics as an approximation valid at large (macroscopic) scale.\n\nQuantum mechanics differs from classical physics in that energy, momentum, angular momentum, and other quantities of a bound system are restricted to discrete values (quantization), objects have characteristics of both particles and waves (wave-particle duality), and there are limits to how accurately the value of a physical quantity can be predicted prior to its measurement, given a complete set of initial conditions (the uncertainty principle).\n\nKeywords:",

    "temperature": 0.3,

    "max_tokens": 60,

    "top_p": 1,

    "frequency_penalty": 0.8,

    "presence_penalty": 0,

    "stop": ["\n"]

}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/keyword-extractor.py"
```

6. Click the **Run** button and review the results.

After running **chapter07/keyword-extractor.py**, you should see a result similar to the console output in the following screenshot:


Figure 7.22 – Example output from chapter07/keyword-extractor.py

Figure 7.22 – Example output from chapter07/keyword-extractor.py

Let's take a look at another example.

## HTML parsing

In this example, we will extract text from HTML. Specifically, the following prompt extracts the value of the title tag (the text between <**title**> and </**title**>). As you can see, the prompt is pretty simple. It just provides some simple directions, the HTML to extract from, and a label for the title:

```
Extract the title, h1, and body text from the following HTML document:

<head><title>A simple page</title></head><body><h1>Hello World</h1><p>This is some text in a simple html page.</p></body></btml>

Title:
```

Now, let's try HTML parsing using Node.js/JavaScript.

## Node.js/JavaScript example

To create the text from HTML example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/text-from-html.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/text-from-html.js**.
4. Replace the **params** variable in **chapter07/text-from-html.js** with the following code:

```
const params = {
  prompt: "Extract the title, h1, and body text from the following HTML
document:\n\n<head><title>A simple page</title></head><body><h1>Hello World</h1><p>This
is some text in a simple html page.</p></body></html>\n\nTitle:",
  temperature: 0,
  max_tokens: 64,
  top_p: 1,
  frequency_penalty: 0.5,
  presence_penalty: 0
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/text-from-html.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/text-from-html.js**, you should see a result similar to the console output in the following screenshot:

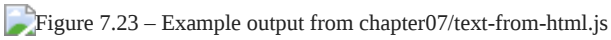


Figure 7.23 – Example output from chapter07/text-from-html.js

Figure 7.23 – Example output from chapter07/text-from-html.js

Let's take a look at the Python code.

## Python example

To create the text from HTML example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/text-from-html.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/text-from-html.py**.
4. Replace the **params** variable in **chapter07/text-from-html.py** with the following code:

```
params = {
    "prompt": "Extract the title, h1, and body text from the following HTML
document:\n\n<head><title>A simple page</title></head><body><h1>Hello World</h1><p>This
```

```
is some text in a simple html page.</p></body></html>\n\nTitle:",
    "temperature": 0,
    "max_tokens": 64,
    "top_p": 1,
    "frequency_penalty": 0.5,
    "presence_penalty": 0
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/text-from-html.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/text-from-html.py**, you should see a result similar to the console output in the following screenshot:

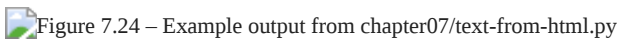


Figure 7.24 – Example output from chapter07/text-from-html.py

Figure 7.24 – Example output from chapter07/text-from-html.py

Let's take a look at another example.

## Extracting a postal address

Let's look at an example that extracts the postal address from an email. The following prompt shows how you could accomplish this.

### *IMPORTANT NOTE*

*This example uses the **davinci-instruct-beta** engine, which is in beta at the time of publishing.*

You can see the prompt provides basic instructions and the postal address in the email is provided in a standard way so GPT-3 will likely be able to identify the address:

```
Extract the postal address from this email:

Dear Paul,

I'm in the market for a new home and I understand you're the listing agent for the
property located at 2620 Riviera Dr, Laguna Beach, CA 92651.

Is the seller flexible at all on the asking price?

Best,

Linda

Property Address:
```

Now try this prompt out with Node.js/JavaScript.

## Node.js/JavaScript example

To create the extracting a postal address example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/extract-postal-address.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/extract-postal-address.js**.
4. Replace the **params** variable in **chapter07/extract-postal-address.js** with the following code:

```
const params = {
  prompt: "Extract the postal address from this email:\n\nDear Paul,\n\nI'm in the market for a new home and I understand you're the listing agent for the property located at 2620 Riviera Dr, Laguna Beach, CA 92651.\n\nIs the seller flexible at all on the asking price?\n\nBest,\n\nLinda\n\nProperty Address:\n",
  temperature: 0,
  max_tokens: 64,
  top_p: 1,
  frequency_penalty: 0.5,
  presence_penalty: 0,
  stop: [""]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/extract-postal-address.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/extract-postal-address.js**, you should see a result similar to the console output in the following screenshot:


Figure 7.25 – Example output from chapter07/extract-postal-address.js

Figure 7.25 – Example output from chapter07/extract-postal-address.js

Now let's try the same example using Python.

## Python example

To create the extracting a postal address example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/extract-postal-address.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/extract-postal-address.py**.
4. Replace the **params** variable in **chapter07/extract-postal-address.py** with the following code:

```
params = {
```

```
    "prompt": "Extract the postal address from this email:\n\nDear Paul,\n\nI'm in the
market for a new home and I understand you're the listing agent for the property
located at 2620 Riviera Dr, Laguna Beach, CA 92651.\n\nIs the seller flexible at all on
the asking price?\n\nBest,\n\nLinda\n\nProperty Address:\n",
    "temperature": 0,
    "max_tokens": 64,
    "top_p": 1,
    "frequency_penalty": 0.5,
    "presence_penalty": 0,
    "stop": [""]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/extract-postal-address.py"
```

6. Click the **Run** button and review the results.

After running **chapter07/extract-postal-address.py**, you should see a result similar to the console output in the following screenshot:


Figure 7.26 – Example output from chapter07/extract-postal-address.py

Figure 7.26 – Example output from chapter07/extract-postal-address.py

Let's take a look at a similar example – extracting an email address.

## Extracting an email address

This prompt is similar to the postal address example but this time we're instructing GPT-3 to extract an email address:

```
Extract the email address from the following message:

Dear Paul,

I'm in the market for a new home and I understand you're the listing agent for the
property located at 2620 Riviera Dr, Laguna Beach, CA 92651.

Can you send details to my wife's email which is beth@example.com?

Best,

Kevin

Email Address:
```

Now, let's try this prompt out with Node.js/JavaScript.

### Node.js/JavaScript example

To create the extracting an email address example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/extract-email-address.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/extract-email-address.js**.
4. Replace the **params** variable in **chapter07/extract-email-address.js** with the following code:

```
const params = {
  prompt: "Extract the email address from the following message:\n\nDear Paul,\n\nI'm in the market for a new home and I understand you're the listing agent for the property located at 2620 Riviera Dr, Laguna Beach, CA 92651.\n\nCan you send details to my wife's email which is beth@example.com?\n\nBest,\n\nKevin\n\nEmail Address:\n",
  temperature: 0,
  max_tokens: 64,
  top_p: 1,
  frequency_penalty: 0.5,
  presence_penalty: 0,
  stop: [""]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/extract-email-address.js"
```

6. Click the **Run** button and review the results.

After running **chapter07/extract-email-address.js**, you should see a result similar to the console output in the following screenshot:


Figure 7.27 – Example output from chapter07/extract-email-address.js

Figure 7.27 – Example output from chapter07/extract-email-address.js

Let's take a look at the Python code.

## Python example

To create the extracting an email address example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/extract-email-address.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/extract-email-address.py**.
4. Replace the **params** variable in **chapter07/extract-email-address.py** with the following code:

```
params = {
    "prompt": "Extract the email address from the following message:\n\nDear Paul,\n\nI'm in the market for a new home and I understand you're the listing agent for
```

```
the property located at 2620 Riviera Dr, Laguna Beach, CA 92651.\n\nCan you send
details to my wife's email which is beth@example.com?\n\nBest,\n\nKevin\n\nEmail
Address:\n",
    "temperature": 0,
    "max_tokens": 64,
    "top_p": 1,
    "frequency_penalty": 0.5,
    "presence_penalty": 0,
    "stop": [""]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/extract-email-address.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/extract-email-address.py**, you should see a result similar to the console output in the following screenshot:


Figure 7.28 – Example output from chapter07/extract-email-address.py

Figure 7.28 – Example output from chapter07/extract-email-address.py

For our last example, we're going to close out with a chatbot.

## Creating chatbots

For the last set of examples, we'll look at creating chatbots. Technically, this would be classified as generating text so it could have been covered in *Generating content and lists*. But creating chatbots with GPT-3 is so much fun it deserves a section of its own. We'll start with a simple conversational chatbot.

## A simple chatbot

For our simple chatbot, we'll be using the following prompt. We'll look at code for both Node.js/JavaScript and Python but the prompt for both is the same.

The first part of the prompt provides instructions for how the bot should respond and the general conversational style. You can change a lot about how the bot responds by changing the instructions and the example dialog. For instance, you could change the conversational tone by changing the words *friendly and polite* to *rude and sarcastic*.

Here is the prompt text for our simple bot:

```
The following is a conversation with an AI bot. The bot is very friendly and polite.

Human: Hello, how are you?

AI: I am doing great, thanks for asking. How can I help you today?

Human: I just wanting to talk with you.

AI:
```

Now, let's take a look at using to implement a simple bot in Node.js/JavaScript.

## Node.js/JavaScript example

To create the simple chatbot example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** repl.
2. Create a new file: **chapter07/simple-chatbot.js**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/simple-chatbot.js**.
4. Replace the **params** variable in **chapter07/simple-chatbot.js** with the following code:

```
const params = {

  prompt: "The following is a conversation with an AI bot. The bot is very friendly and polite.\n\nHuman: Hello, how are you?\nAI: I am doing great, thanks for asking. How can I help you today?\nHuman: I just wanting to talk with you.\nAI:",

  temperature: 0.9,

  max_tokens: 150,

  top_p: 1,

  frequency_penalty: 0,

  presence_penalty: 0.6,

  stop: ["\n, Human:, AI:"]

}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "node chapter07/simple-chatbot.js"
```

6. Click the **Run** button and review the results.

   After running **chapter07/simple-chatbot.js**, you should see a result similar to the console output in the following screenshot:

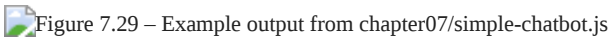



Figure 7.29 – Example output from chapter07/simple-chatbot.js

Now the Python version.

## Python example

To create the simple chatbot example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** repl.
2. Create a new file: **chapter07/simple-chatbot.py**.
3. Copy the code from the **dumb-joke-generator.py** file into **chapter07/simple-chatbot.py**.
4. Replace the **params** variable in **chapter07/simple-chatbot.py** with the following code:

```
params = {
    "prompt": "The following is a conversation with an AI bot. The bot is very friendly and polite.\n\nHuman: Hello, how are you?\nAI: I am doing great, thanks for asking. How can I help you today?\nHuman: I just wanting to talk with you.\nAI:",
    "temperature": 0.9,
    "max_tokens": 150,
    "top_p": 1,
    "frequency_penalty": 0,
    "presence_penalty": 0.6,
    "stop": ["\n, Human:, AI:"]
}
```

5. Change the **.replit** file in your root folder to the following:

```
run = "python chapter07/simple-chatbot.py"
```

6. Click the **Run** button and review the results.

   After running **chapter07/simple-chatbot.py**, you should see a result similar to the console output in the following screenshot:


Figure 7.30 – Example output from chapter07/simple-chatbot.py

Figure 7.30 – Example output from chapter07/simple-chatbot.py

That's our last example. Let's wrap up with a quick summary.

## Summary

In this chapter, we covered generating and transforming text. We walked through 15 examples in both Node.js/JavaScript and Python. The examples included generating content and lists, translating and transforming text, extracting text, and creating simple chatbots.

In the next chapter, we'll walk through examples of classifying and categorizing text.

# *Chapter 8*: Classifying and Categorizing Text

In the last chapter, we looked at different ways to generate text. In this chapter, we'll discuss text classification and the OpenAI API classifications endpoint. We'll start with a quick overview of text classification and the classifications endpoint, and then we'll work through implementing sentiment analysis, assigning an ESRB rating to text, categorizing text by language, and classifying text from keywords, which are all common text classification examples.

The topics we will be covering in this chapter are as follows:

- Understanding text classification
- Introducing the classifications endpoint
- Implementing sentiment analysis
- Assigning an ESRB rating to text
- Classifying text by language
- Classifying text from keywords

## Technical requirements

This chapter requires that you have access to the OpenAI API. You can request access by visiting https://openapi.com.

## Understanding text classification

A text classification task takes in text and returns a label. Classifying email as spam or determining the sentiment of a tweet are both examples of text classification tasks. There are multiple ways to do text classification using the OpenAI API and we've looked at some of them already. But one method we haven't covered yet is using the **completions endpoint**. However, before we dive into the completions endpoint, let's quickly review some of the different ways we can do text classification that we've already covered.

## Using the completions endpoint for text classification

For starters, you can perform classification tasks using the completions endpoint by describing the task in our prompt. For example, the following prompt can be used to classify a social media post:

```
Social media post: "My favorite restaurant is opening again Monday. I can't wait!"

Sentiment (positive, neutral, negative):
```

The previous prompt would return positive, natural, or negative, but most likely positive, given the post.

## Content filtering is a text classification task

Content filtering is also a type of text classification task. Recall from *Chapter 6*, *Content Filtering*, when we used the content filter engine, that it returned **0 = safe**, **1 = sensitive**, and **2 = unsafe**, for text that was provided. That was text classification.

While there are multiple ways to do text classification using the OpenAI API. There is one endpoint that is specifically designed for classification tasks. That endpoint is the classifications endpoint, and we'll discuss this next.

## Introducing the classifications endpoint

The OpenAI API also provides the **classifications endpoint** for text classification tasks. The classifications endpoint simplifies many classification tasks. It uses a combination of semantic search and completions engines to classify text based on the samples you provide. You can provide up to 200 examples along with your HTTP request or you can pre-uploaded files containing example data.

The URL for the classifications endpoint is https://api.openai.com/v1/classifications. It expects an HTTP **POST** with a JSON body containing input parameters. One of the required parameters is the query parameter. The value of the query parameter is the text to classify. The query value is first used to do a semantic search to find relevant examples from the examples provided. Then, the examples are used, along with the query, to create a prompt for a defined completions engine that will classify the text.

The following code block shows a simple request body for the classifications endpoint. Note that the examples are provided with this request and the model that will be used to do the classification is the **curie** model:

```
{
  "query": "That makes me smile",
  "examples": [
    ["That is awesome", "Happy"],
    ["I feel so sad", "Sad"],
    ["I don't know how I feel", "Neutral"]
  ],
  "model": "curie"
```

```
}
```

As mentioned, you can also upload example data and use a file parameter to reference the uploaded example data. This is useful when you have a large number of examples – over 200. Let's look at uploading files.

## Uploading files

Example data for the classifications endpoint can be uploaded using the OpenAI API files endpoint. The file should be formatted based on the JSON lines text format, which is basically a valid JSON object on each line that is separated by a line break.

### *IMPORTANT NOTE*

*You can learn more about the JSON lines format at https://jsonlines.org.*

The following code block provides an example of the format required for a classifications sample file. The text property and label properties are required, but the metadata is optional. The metadata property can contain a JSON object with any information you'd like. This data can then optionally be returned with the query results:

```
{"text": "that is awesome", "label": "Happy", "metadata": {"id":"1"}}
{"text": "i feel so sad", "label": "Sad", "metadata": {"id":"2"}}
{"text": "i don't know how i feel", "label": "Neutral", "metadata": {"id":"3"}}
```

To upload a sample file, you use the OpenAI API files endpoint. For the examples in this chapter, we won't be using files. However, we'll take a closer look at the files endpoint in *Chapter 9*, *Building a GPT-3 Powered Question-Answering App*.

## Implementing sentiment analysis

A common classification task is sentiment analysis. Using sentiment analysis, you can classify text based on its general tone – for example, happy, sad, mad, or neutral. This can be useful in a lot of applications; for example, if you're a restaurant owner and you want to respond quickly to unhappy customer reviews. Let's take a look at how we could do that using the OpenAI API classifications endpoint.

In this example, we will classify restaurant reviews. We'll label the reviews with the labels Good, Poor, or Neutral. We will use the classifications endpoint for this example, and we'll provide some example reviews with the request.

### Node.js/JavaScript example

To create the review classifier example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** REPL.
2. Create a new file – **chapter08/reviews-classifier.js**.
3. Add the following code to the beginning of the **reviews-classifier.js** file:

```
const axios = require('axios');
const client = axios.create({
  headers: {
    'Authorization': 'Bearer ' + process.env.OPENAI_API_KEY
  }
});
const endpoint = "https://api.openai.com/v1/classifications";
```

4. Then, add example reviews that will be used with the request:

```
const examples = [
  ["The service was super quick. I love that.","Good"],
  ["Would not go back.","Poor"],
  ["I tried the chicken and cranberry pizza...mmmm!","Good"],
  ["There were no signs indicating cash only!","Poor"],
  ["I was disgusted. There was a hair in my food.","Poor"],
  ["The waitress was a little slow but friendly.","Neutral"]
]
```

5. Next, add the request parameters for the classifications endpoint:

```
const params = {
  "query": "I'm never going to this place again",
  "examples": reviews,
  "model": "curie"
}
```

6. Finally, add the following code to log the result to the console:

```
client.post(endpoint, params)
  .then(result => {
    console.log(params.query + '\nLABEL:' + result.data.label);
  }).catch(err => {
    console.log(err);
  });
```

7. Change the **.replit** file in your root folder to the following:

```
run = "node chapter08/reviews-classifier.js"
```

8. Click the **Run** button and review the results.

   After running the **chapter08/reviews-classifier.js** file, you should see a result similar to the console output in the following screenshot:

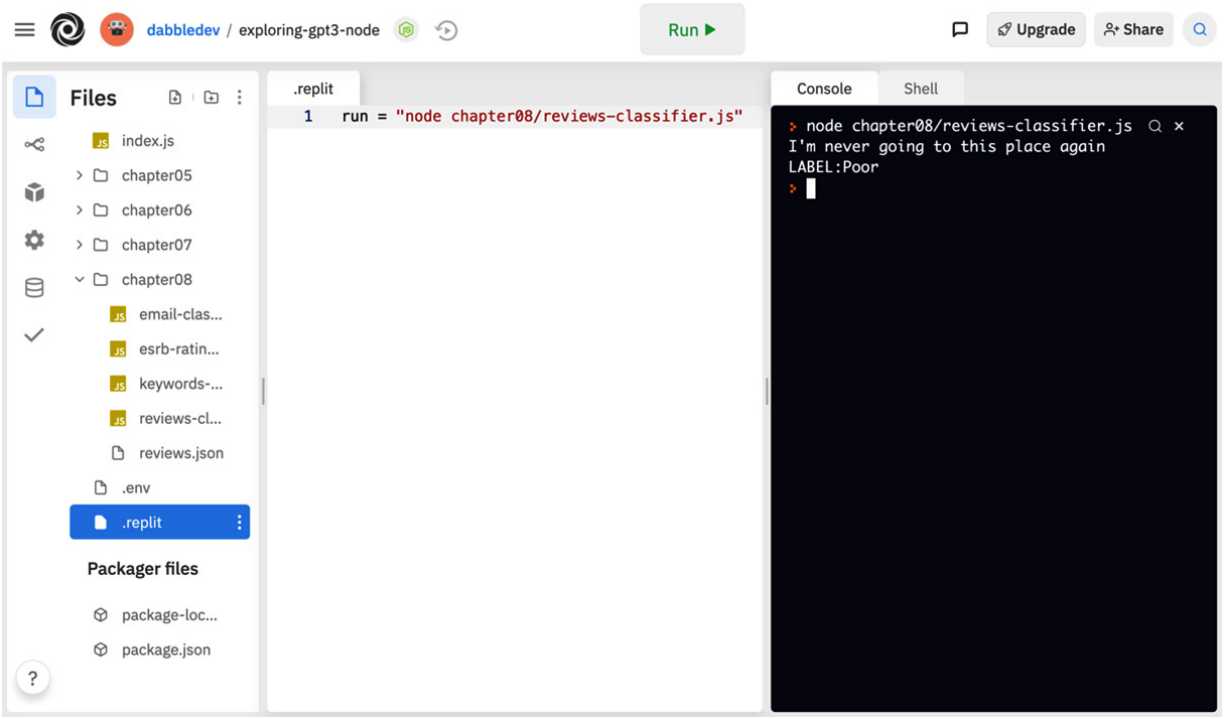



Figure 8.1 – Example output from chapter08/reviews-classifier.js

Next, let's look at the same example using Python.

## Python example

To create the online review classifier example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** REPL.
2. Create a new file – **chapter08/reviews-classifier.py**.
3. Add the following code to the beginning of the **reviews-classifier.py** file:

```
import requests
import os
import json
headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer ' + os.environ.get("OPENAI_API_KEY")
}
endpoint = "https://api.openai.com/v1/classifications"
```

4. Create an array for the review examples:

```
examples = [
  ["The service was super quick. I love that.","Good"],
  ["Would not go back.","Poor"],
  ["I tried the chicken and cranberry pizza...mmmm!","Good"],
  ["There were no signs indicating cash only!","Poor"],
  ["I was disgusted. There was a hair in my food.","Poor"],
  ["The waitress was a little slow but friendly.","Neutral"]
]
```

5. Set the request parameters for the endpoint:

```
params = {
  "query": "I'm never going to this place again",
  "examples": examples,
  "model": "curie"
}
```

6. Make the HTTP request and print the results to the console:

```
result = requests.post(endpoint, headers=headers, data=json.dumps(params))
print(params["query"] + '\nLABEL:' + result.json()["label"])
```

7. Change the **.replit** file in your root folder to the following:

```
run = "python chapter08/reviews-classifier.py"
```

8. Click the **Run** button and review the results.

After running the **chapter08/online-review-classifier.py** file, you should see a result similar to the console output in the following screenshot:

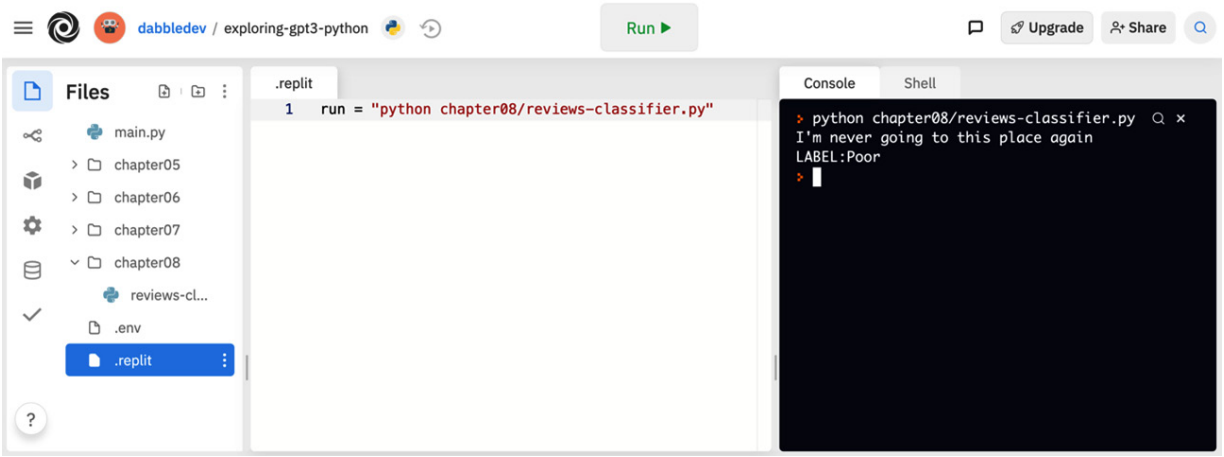



Figure 8.2 – Example output from chapter08/online-review-classifier.py

Let's now take a look at another example.

# Assigning an ESRB rating to text

In the last example, we provided sample data to help with our classification task. But GPT-3 is pre-trained with a huge dataset, meaning it can perform a surprising number of classification tasks without providing any example data. Let's take a look at another example using the completions endpoint. In this example, we'll look at classifying text with an **Entertainment Software Rating Board** (**ESRB**) rating.

In this example, we will use the completions endpoint to assign an ESRB rating to text without any example data.

## Node.js/JavaScript example

To create the ESRB rating classifier example in **Node.js/JavaScript**, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** REPL.
2. Create a new file – **chapter08/esrb-rating-classifier.js**.
3. Add the following code to the beginning of the **esrb-rating-classifier.js** file:

```
const axios = require('axios');

const apiKey = process.env.OPENAI_API_KEY;

const client = axios.create({

  headers: { 'Authorization': 'Bearer ' + apiKey }

});

const endpoint = "https://api.openai.com/v1/engines/davinci/completions";
```

4. Add the endpoint parameters to the **esrb-rating-classifier.js** file with the following code:

```
const params = {

  prompt: "Provide an ESRB rating for the following text:\n\n\"i'm going to hunt you down, and when I find you, I'll make you wish you were dead.\"\n\nESRB rating:",

  temperature: 0.7,

  max_tokens: 60,

  top_p: 1,

  frequency_penalty: 0,

  presence_penalty: 0,

  stop: ["\n"]

}
```

5. Add the following code to log the endpoint response to the console:

```
client.post(endpoint, params)
```

```
  .then(result => {
    console.log(params.prompt + result.data.choices[0].text);
    // console.log(result.data);
  }).catch(err => {
    console.log(err);
  });
```

6. Change the **.replit** file in your root folder to the following:

```
run = "node chapter08/esrb-rating-classifier.js"
```

7. Click the **Run** button and review the results.

After running the **chapter08/esrb-rating-classifier.js** file, you should see a result similar to the console output in the following screenshot:

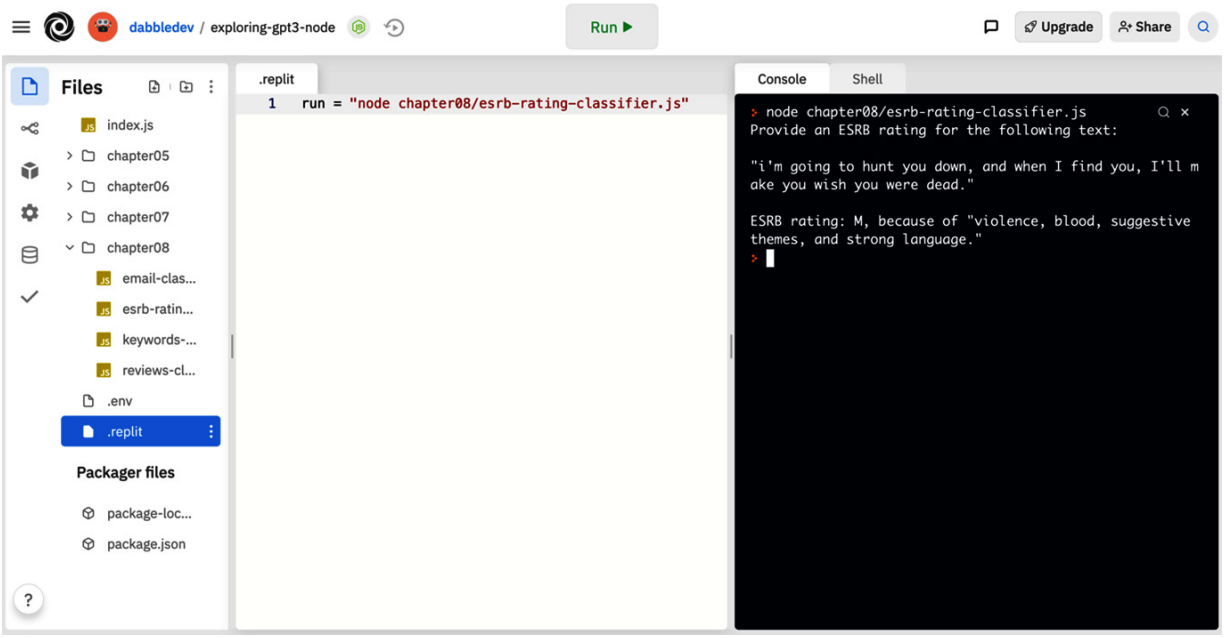



Figure 8.3 – Example output from chapter08/esrb-rating-classifier.js

Now, let's look at the ESRB rating classifier in Python.

## Python example

To create the ESRB rating classifier example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** REPL.
2. Create a new file – **chapter08/esrb-rating-classifier.py**.
3. Add the following code to the beginning of the **esrb-rating-classifier.py** file:

```
import requests
import os
```

```
import json

headers = {

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

    'Authorization': 'Bearer ' + os.environ.get("OPENAI_API_KEY")

}

endpoint = 'https://api.openai.com/v1/engines/davinci/completions'
```

4. Add the endpoint parameters to the **esrb-rating-classifier.js** file with the following code:

```
params = {

  "prompt": "Provide an ESRB rating for the following text:\n\n\"i'm going to hunt you down, and when I find you, I'll make you wish you were dead.\"\n\nESRB rating:",

  "temperature": 0.7,

  "max_tokens": 60,

  "top_p": 1,

  "frequency_penalty": 0,

  "presence_penalty": 0,

  "stop": ["\n"]

}
```

5. Add the following code to log the endpoint response to the console:

```
result = requests.post(endpoint, headers=headers, data=json.dumps(params))

print(params["prompt"] + result.json()["choices"][0]["text"])
```

6. Change the **.replit** file in your root folder to the following:

```
run = "node chapter08/esrb-rating-classifier.js"
```

7. Click the **Run** button and review the results.

After running the **chapter08/esrb-rating-classifier.js** file, you should see a result similar to the console output in the following screenshot:

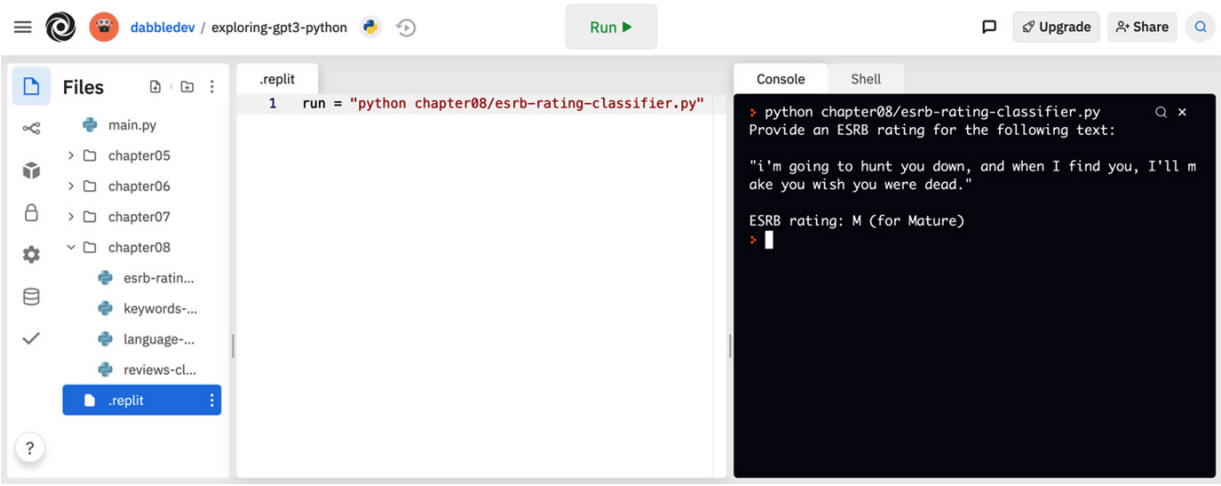



Figure 8.4 – Example output from chapter08/esrb-rating-classifier.py

Let's now take a look at another example.

## Classifying text by language

Now, let's consider an example. Suppose we needed to route support messages based on the language the message was written in – for a multinational support center. In this case, we could use GPT-3 to classify messages by language, such as English, French, Hindi, Spanish, and Russian. Let's see how we'd go about doing that.

In this example, we will classify support messages by language using the classifications endpoint and examples for each language.

### Node.js/JavaScript example

To create the email classifier example in Node.js/JavaScript, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** REPL.
2. Create a new file – **chapter08/language-classifier.js**.
3. Add the following code to the beginning of the **language-classifier.js** file:

```
const axios = require('axios');
const apiKey = process.env.OPENAI_API_KEY;
const client = axios.create({
  headers: { 'Authorization': 'Bearer ' + apiKey }
});
const endpoint = "https://api.openai.com/v1/classifications";
```

4. Create an array for the language examples:

```
const examples = [
```

```
  ["Hello, I'm interested in applying for the prompt designer position you are hiring
for. Can you please tell me where I should send my resume?","English"],

  ["Здравствуйте, я хочу подать заявку на должность быстрого дизайнера, на которую вы
нанимаете. Подскажите, пожалуйста, куда мне отправить резюме?","Russian"],

  ["Hola, estoy interesado en postularme para el puesto de diseñador rápido para el que
está contratando. ¿Puede decirme dónde debo enviar mi currículum?", "Spanish"],

  ["Bonjour, je suis intéressé à postuler pour le poste de concepteur rapide pour
lequel vous recrutez. Pouvez-vous me dire où je dois envoyer mon CV?","French"],

  ["□□□□□□□□, □□□ □□ □□□□□□ □□□□□□□□ □□ □□ □□□ □□□□□ □□□□ □□□ □□□□ □□□□ □□□, □□□□□ □□□
□□ □□□□□ □□ □□□ □□□□ □□□□ □□ □□□□ □□□ □□□□ □□□ □□ □□□□ □□□□ □□□□□□□ □□□□ □□□□□
□□□□□?","Hindi"]

]
```

If necessary, you can use https://translate.google.com to create the example data.

5. Add the endpoint parameters with the following code:

```
const params = {

  "query": "¿Con quién debo comunicarme sobre ofertas de trabajo técnico?",

  "examples": examples,

  "model": "curie"

}
```

6. Add the following code to log the endpoint response to the console:

```
client.post(endpoint, params)

  .then(result => {

    console.log(params.query + '\nLABEL:' + result.data.label);

  }).catch(err => {

    console.log(err);

  });
```

7. Change the **.replit** file in your root folder to the following:

```
run = "node chapter08/language-classifier.js"
```

8. Click the **Run** button and review the results.

After running the **chapter08/email-classifier.js** file, you should see a result similar to the console output in the following screenshot:

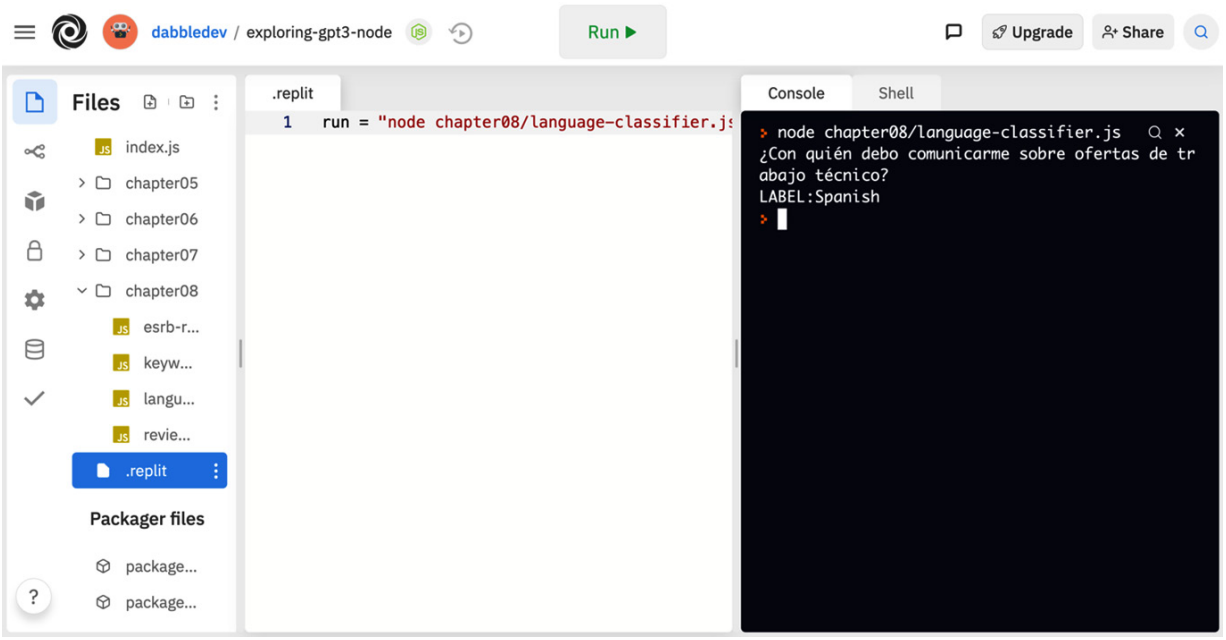



Figure 8.5 – Example output from chapter08/language-classifier.js

Let's look at the Python version next.

## Python example

To create the language classifier example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** REPL.
2. Create a new file – **chapter08/language-classifier.py**.
3. Add the following code to the beginning of the **language-classifier.py** file:

```
import requests
import os
import json
headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer ' + os.environ.get("OPENAI_API_KEY")
}
endpoint = "https://api.openai.com/v1/classifications"
```

4. Create an array for the language examples:

```
examples = [
  ["Hello, I'm interested in applying for the prompt designer position you are hiring for. Can you please tell me where I should send my resume?","English"],
  ["Здравствуйте, я хочу подать заявку на должность быстрого дизайнера, на которую вы нанимаете. Подскажите, пожалуйста, куда мне отправить резюме?","Russian"],
```

```
  ["Hola, estoy interesado en postularme para el puesto de diseñador rápido para el que
está contratando. ¿Puede decirme dónde debo enviar mi currículum?", "Spanish"],

  ["Bonjour, je suis intéressé à postuler pour le poste de concepteur rapide pour
lequel vous recrutez. Pouvez-vous me dire où je dois envoyer mon CV?","French"],

  ["□□□□□□□□, □□□ □□ □□□□□□□ □□□□□□□□□ □□ □□ □□□ □□□□□□ □□□□□ □□□ □□□□□ □□□□□ □□□, □□□□□□ □□□
□□ □□□□□□ □□ □□□ □□□□□ □□□□□ □□ □□□□□ □□□ □□□□□ □□□ □□ □□□□□ □□□□□ □□□□□□□□ □□□□□ □□□□□□
□□□□□□?","Hindi"]

]
```

If necessary, you can use https://translate.google.com to create the example data.

5. Add the endpoint parameters with the following code:

```
const params = {

  "query": "¿Con quién debo comunicarme sobre ofertas de trabajo técnico?",

  "examples": examples,

  "model": "curie"

}
```

6. Add the following code to log the endpoint response to the console:

```
result = requests.post(endpoint, headers=headers, data=json.dumps(params))

print(params["query"] + '\nLABEL:' + result.json()["label"])
```

7. Change the **.replit** file in your root folder to the following:

```
run = "python chapter08/language-classifier.py"
```

8. Click the **Run** button and review the results.

After running the **chapter08/language-classifier.py** file, you should see a result similar to the console output in the following screenshot:

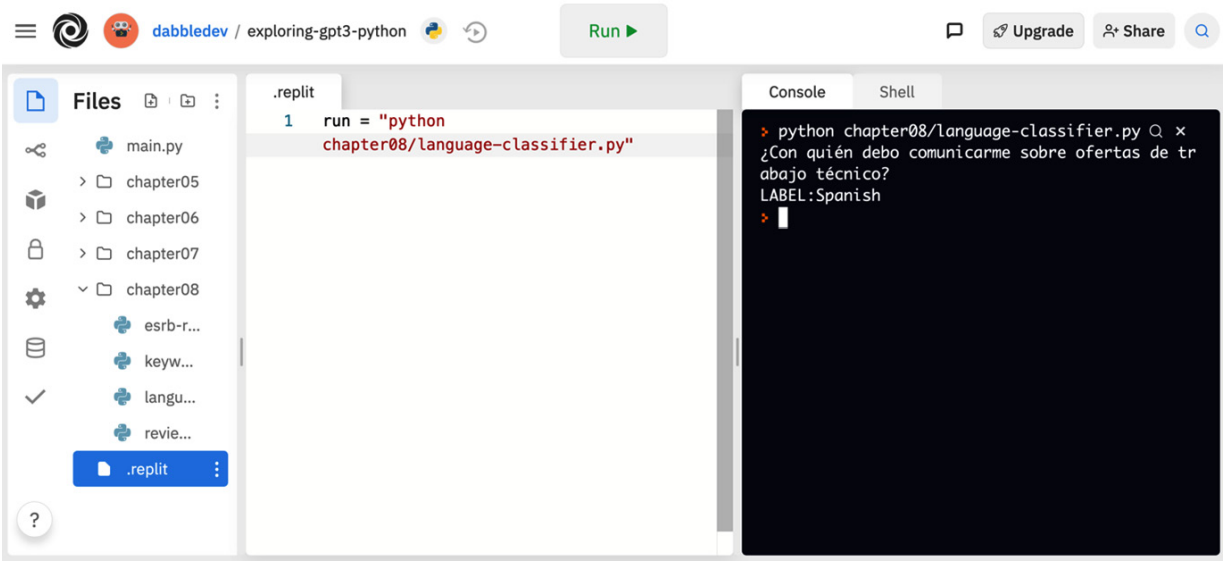


Figure 8.6 – Example output from chapter08/language-classifier.py

Let's now take a look at another example.

# Classifying text from keywords

Another common text classification task is to classify documents based on keywords. To do this, we can use GPT3 to create a list of keywords that will be related to the content of the document. However, GPT3 doesn't just extract keywords from the document. It determines keywords that are relevant based on the document content. Let's try an example.

In this example, we will use the completions endpoint to classify a document based on relevant keywords.

## Node.js/JavaScript example

To create the keywords classifier example in **Node.js/JavaScript**, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-node** REPL.
2. Create a new file – **chapter08/keywords-classifier.js**.
3. Add the following code to the beginning of the **keywords-classifier.js** file:

```
const axios = require('axios');

const apiKey = process.env.OPENAI_API_KEY;

const client = axios.create({

  headers: { 'Authorization': 'Bearer ' + apiKey }

});

const endpoint = "https://api.openai.com/v1/engines/davinci/completions";
```

4. Add the endpoint parameters to **keywords-classifier.js** with the help of the following code:

```
const params = {

  prompt: "Text: When NASA opened for business on October 1, 1958, it accelerated the work already started on human and robotic spaceflight. NASA's first high profile program was Project Mercury, an effort to learn if humans could survive in space. This was followed by Project Gemini, which used spacecraft built for two astronauts to perfect the capabilities needed for the national objective of a human trip to the Moon by the end of the 1960s. Project Apollo achieved that objective in July 1969 with the Apollo 11 mission and expanded on it with five more successful lunar landing missions through 1972. After the Skylab and Apollo-Soyuz Test Projects of the mid-1970s, NASA's human spaceflight efforts again resumed in 1981, with the Space Shuttle program that continued for 30 years. The Shuttle was not only a breakthrough technology, but was essential to our next major step in space, the construction of the International Space Station.\n\nKeywords:",

  temperature: 0.3,
```

```
    max_tokens: 60,
    top_p: 1,
    frequency_penalty: 0.8,
    presence_penalty: 0,
    stop: ["\n"]
  }
```

5. Add the following code to log the endpoint response to the console:

```
client.post(endpoint, params)
  .then(result => {
    console.log(params.prompt + result.data.choices[0].text);
    // console.log(result.data);
  }).catch(err => {
    console.log(err);
  });
```

6. Change the **.replit** file in your root folder to the following:

```
run = "node chapter08/keywords-classifier.js"
```

7. Click the **Run** button and review the results.

After running the **chapter08/keywords-classifier.js** file, you should see a result similar to the console output in the following screenshot. Notice in the results that some of the keywords identified may not exist in the original text:

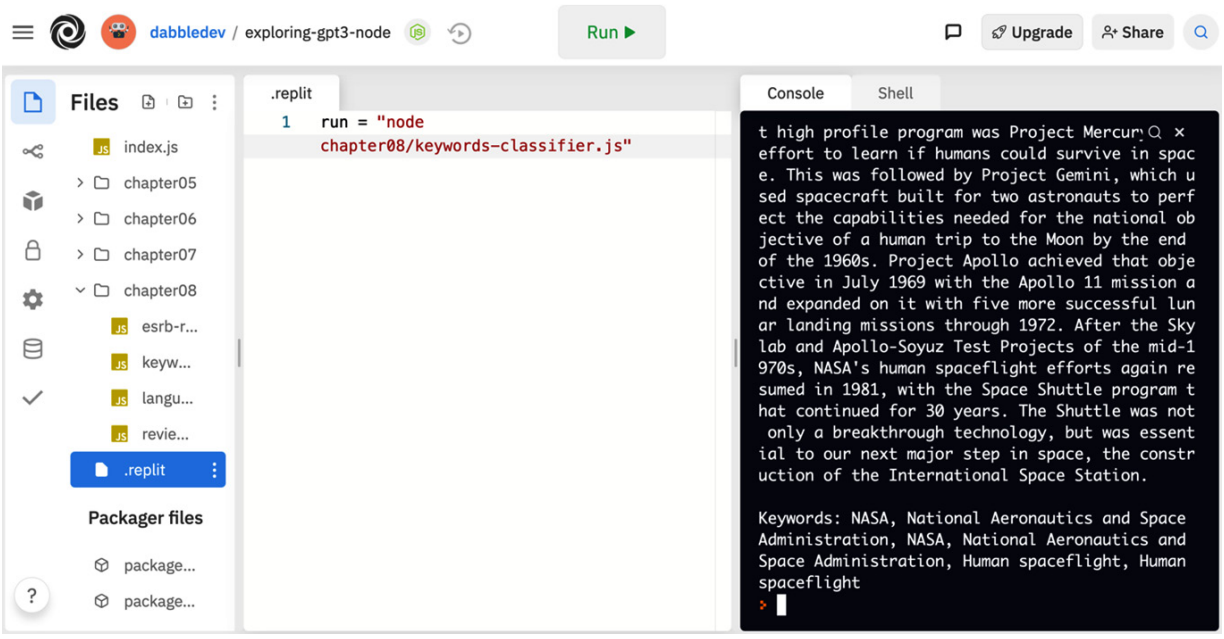



Figure 8.7 – Example output from chapter08/keywords-classifier.js

Alright, next, let's look at the Python version.

## Python example

To create the keywords classifier example in Python, follow these steps:

1. Log in to replit.com and open your **exploring-gpt3-python** REPL.
2. Create a new file – **chapter08/keywords-classifier.py**.
3. Add the following code to the beginning of the **keywords-classifier.py** file:

```
import requests

import os

import json

headers = {

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

    'Authorization': 'Bearer ' + os.environ.get("OPENAI_API_KEY")

}

endpoint = 'https://api.openai.com/v1/engines/davinci/completions'
```

4. Add a **params** variable to **chapter08/keywords-classifier.py** with the following code:

```
params = {

  "prompt": "Text: When NASA opened for business on October 1, 1958, it accelerated the work already started on human and robotic spaceflight. NASA's first high profile program was Project Mercury, an effort to learn if humans could survive in space. This was followed by Project Gemini, which used spacecraft built for two astronauts to perfect the capabilities needed for the national objective of a human trip to the Moon by the end of the 1960s. Project Apollo achieved that objective in July 1969 with the Apollo 11 mission and expanded on it with five more successful lunar landing missions through 1972. After the Skylab and Apollo-Soyuz Test Projects of the mid-1970s, NASA's human spaceflight efforts again resumed in 1981, with the Space Shuttle program that continued for 30 years. The Shuttle was not only a breakthrough technology, but was essential to our next major step in space, the construction of the International Space Station.\n\nKeywords:",

  "temperature": 0.3,

  "max_tokens": 60,

  "top_p": 1,

  "frequency_penalty": 0.8,

  "presence_penalty": 0,

  "stop": ["\n"]

}
```

5. Add the following code to log the endpoint response to the console:

```
result = requests.post(endpoint, headers=headers, data=json.dumps(params))
```

```
print(params["prompt"] + result.json()["choices"][0]["text"])
```

6. Change the .**replit** file in your root folder to the following:

```
run = "python chapter08/keywords-classifier.py"
```

7. Click the **Run** button and review the results.

   After running the **chapter08/keywords-classifier.py** file, you should see a result similar to the console output in the following screenshot:

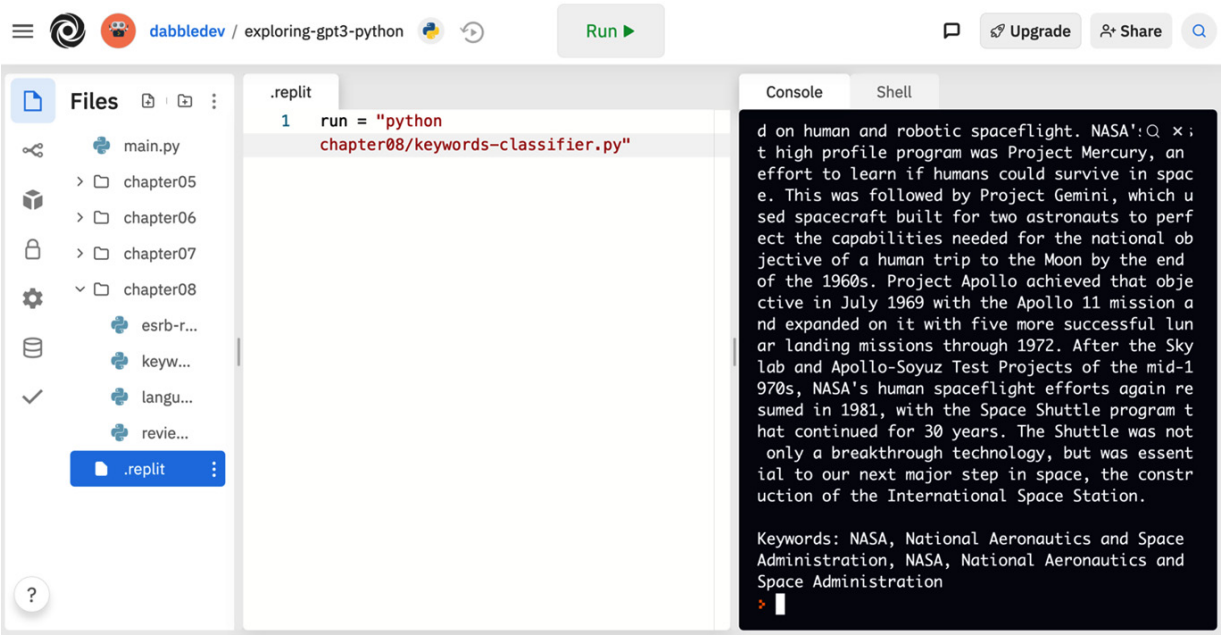



Figure 8.8 – Example output from chapter08/keywords-classifier.py

Again, notice that some of the keywords returned don't appear in the text provided, but they are still relevant. This is possible because GPT3 is using its language model to consider keywords that are the best fit, even if they aren't contained in the text.

## Summary

In this chapter, we covered understanding text classification and the classifications API endpoint. Then, we worked through examples of implementing sentiment analysis, assigning an ESRB rating to text, classifying text by language, and classifying text with keywords.

In the next chapter, we will look at working with the semantic search API endpoint.

# *Chapter 9*: Building a GPT-3-Powered Question-Answering App

Up to this point, we've looked at (and written) a lot of code. But we haven't actually created a fully functional app. Well, that's what we're going to do now. In this chapter, we're going to build a simple but powerful web app that lets users ask questions that GPT-3 will answer from a knowledge base we will provide. The app could be used to answer any kind of questions, but we're going to use it to answer questions people might have about us – an *ask me anything* app. So, imagine a website that recruiters or a potential employer could use to ask questions about your skills, accomplishments, and experience. Not looking for a new job? No problem. Again, this app can be used for just about any question-answering application – so maybe a GPT-3-powered product FAQ, or a GPT-3-powered teaching assistant – it's completely up to you. We'll start with a quick overview of how the app will work, then we'll step through the process of building it.

The topics we'll cover are the following:

- Introducing GPT Answers
- Introducing the Answers endpoint
- Setting up and testing Express
- Creating the API endpoint for GPT Answers
- Creating the GPT Answers user interface
- Integrating the Answers endpoint
- Generating relevant and factual answers
- Using files with the Answers endpoint

## Technical requirements

This chapter requires that you have access to the OpenAI API. You can request access by visiting https://openapi.com.

## Introducing GPT Answers

In this section, we're going to be building a GPT-3-powered web app that lets users ask any questions and get back answers from a knowledge base of data that we'll provide. We will call the app **GPT Answers**. Yes, the name is underwhelming, but we can always use GPT-3 to help us brainstorm a better name later.

The following is a screenshot of what the app will look like when it's completed. Sure, the user interface might be as underwhelming as the name, but the power behind it is sure to impress!


Figure 9.1 – GPT Answers user interface

Now let's get into what's behind the UI and how the app will be built.

## GPT Answers technical overview

GPT Answers will be built using Node.js, JavaScript, and HTML. We'll also be using a web framework called Express, to simplify the development.

### *IMPORTANT NOTE*

*This app could also be built using Python but for this chapter, unlike the previous chapters, we'll only be covering the steps to create the app using Node.js/JavaScript.*

Questions will be submitted through a simple web form that will use JavaScript to make requests to an API endpoint that the app will also expose. The app API will primarily act as a proxy for interacting with the OpenAI API, but it will also provide exception handling and response formatting.

## Hosting the app

Up to this point, we've only used replit.com for writing and testing code. However, replit.com also supports hosting apps and it's surprisingly easy to work with. For web apps, you can even use your own domain name. So, replit.com is going to be our hosting environment as well as our development environment.

### *IMPORTANT NOTE*

*GPT-3-powered apps need to be approved by OpenAI before they can be public-facing. We won't get into that in this chapter, but we'll cover the app approval process in Chapter 10, Going Live with OpenAI-Powered Apps.*

The main OpenAI endpoint the app will use is the Answers endpoint. But since we have not covered the Answers endpoint yet, let's do a quick introduction before we start coding.

## Introducing the Answers endpoint

The OpenAI Answers endpoint is specifically designed for question-answering tasks. It provides more control than the Completions endpoint by enabling the use of a source of truth for the answers. For our GPT Answers app, that source of truth will be a knowledge base that will be used for answering questions. The knowledge base (that is, documents) can be provided along with the endpoint request or by referencing a pre-uploaded file containing the data.

The URL for the Answers endpoint is https://api.openai.com/v1/answers and the endpoint accepts an HTTP POST request and a number of input parameters. The following is a brief description of the available input parameters, but for more complete details, see the OpenAI docs for the Answers endpoint located at https://beta.openai.com/docs/api-reference/answers.

Here are the required parameters:

- **model** (required, string) – The ID of the model that will be used for completions.
- **question** (required, string) – The question to be answered.
- **examples** (required, array) – A list of questions with answers to help steer the model toward the tone and answer format for the answer.
- **examples_context** (required, string) – A text snippet containing the contextual information used to generate the answers for the examples you provide.
- **documents** (array) – A list of documents from which the answer for the input question should be derived. If the **documents** parameter is an empty list, the question will be answered based on the question-answer examples. Also, the **documents** parameter is only required if the **file** parameter is not used.
- **file** (string) – The ID of an uploaded file containing documents to derive the answer from. The **file** parameter is only required if the **documents** parameter is not used.

The optional parameters are as follows:

- **search_model** (string) – The engine to use for search. This defaults to **ada**.
- **max_rerank** (integer) – The maximum number of documents to be. A higher value can improve accuracy but will increase the latency and cost. This defaults to **200**.
- **temperature** (number) – Defaults to **0**, which is best for well-defined answers, but a higher value can be used for less deterministic answers.
- **logprobs** (integer) – Defaults to **null**. The number of likely probable tokens to return.
- **max_tokens** (integer) – The maximum number of tokens that will be used to generate an answer. Defaults to **16**.

- **stop** (string or array) – An optional sequence of up to four patterns that will cause the API to stop generating a completion. This defaults to **null**.
- **n** (integer) – The number of answers to generate for each question. This defaults to **1**.
- **logit_bias** (map) – Can be used to control the likelihood of specified tokens appearing in the completion.
- **return_metadata** (Boolean) – If the **file** parameter is used and the file referenced includes metadata, this will cause the response to include the metadata from the file.
- **return_prompt** (Boolean) – Causes the prompt text to be returned with the response. This defaults to **false**.
- **expand** (array) – Causes the response to include details about the completion or file. The value of **expand** can currently include **completion** and **file**. This defaults to an empty array.

  ### *IMPORTANT NOTE*

  *We won't be using all of the available parameters for our GPTAMA app.*

Now that we've done a quick introduction to the Answers endpoint, let's get to coding up our GPTAMA app!

## Setting up and testing Express

Express is a lightweight but flexible web application framework for Node.js that we'll be using for the app. It's pretty easy to get it up and running, especially with Replit.com. So, the first thing we'll do is get Express set up on Replit.com and test it out. We'll be starting from scratch, so we'll be creating a new repl for GPTAMA.

To create a new Node.js REPL and set up Express, complete the following steps:

1. Log in at replit.com.
2. Create a new Node.js REPL named **gptanswers-node**.
3. In the output pane, click on the **Shell** tab and enter this command:

   ```
   npx express-generator --no-view --force .
   ```

4. Run the previous command by pressing the *Enter* key and you should see a result that looks like the following screenshot:

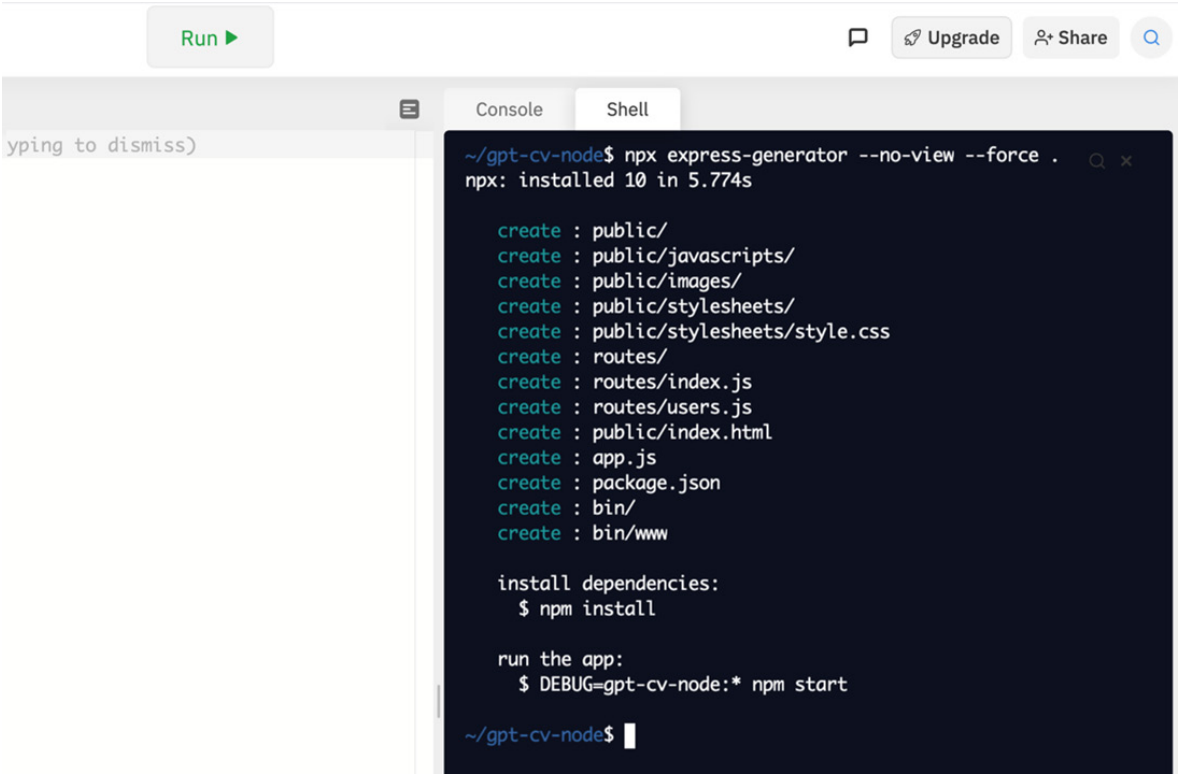



Figure 9.2 – Output from express-generator

## *IMPORTANT NOTE*

*The **npx** command is included with NPM, which is installed with Node.js. It is used to run **express-generator**, which creates a basic Express app as a starting point. The command ends with a period to instruct **express-generator** to add files to the current directory. The **--no-view** switch tells the generator we're just using plain HTML for our UI and the **--force** switch tells the generator to overwrite any existing files in the current directory.*

5. After **express-generator** completes, run the following command in the shell:

   ```
   npm update
   ```

6. Now create a file named **.replit** and add the following **Run** command to it:

   ```
   Run = "node ./bin/www"
   ```

7. Finally, click the **Run** button to start the Express server. If all went well, you should see a browser window open in the Replit.com editor with a welcome message from Express. It should look like the following screenshot:

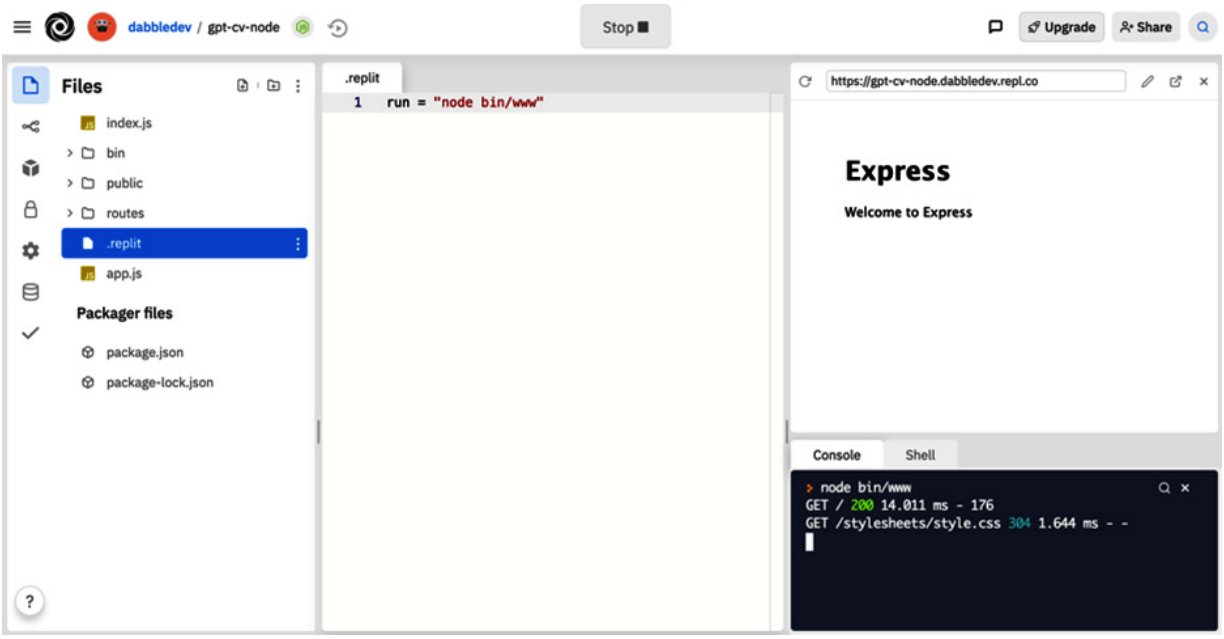



Figure 9.3 – Express server running in Replit.com

Two things to note are the URL in the browser pane and that the **Run** button became a **Stop** button. This is because Express is running an HTTP server that will continue running until it's stopped. So, the app is available on the web and is accessible via the URL in the browser pane while the Express server is running. Also, when you make changes, you'll need to stop and restart Express by clicking the **Stop** button and then the **Run** button.

If you run into any issues and don't see the Express page, you can go through the steps in this section again without hurting anything. When you see the Express welcome page, you're all set to move on. Next, we'll create an API endpoint for our GPT Answers app.

## Creating the API endpoint for GPT Answers

When our app is complete, we'll have a fully functional API endpoint that can return answers generated by the OpenAI API. But for now, we'll just create an endpoint that returns a placeholder response. Then, we'll test the endpoint using Postman and we'll come back later and finish coding it up.

## Creating the API endpoint

To create the API endpoint, do the following:

1. Open the **app.js** file that was created by **express-generator**. The file should look like the following screenshot:

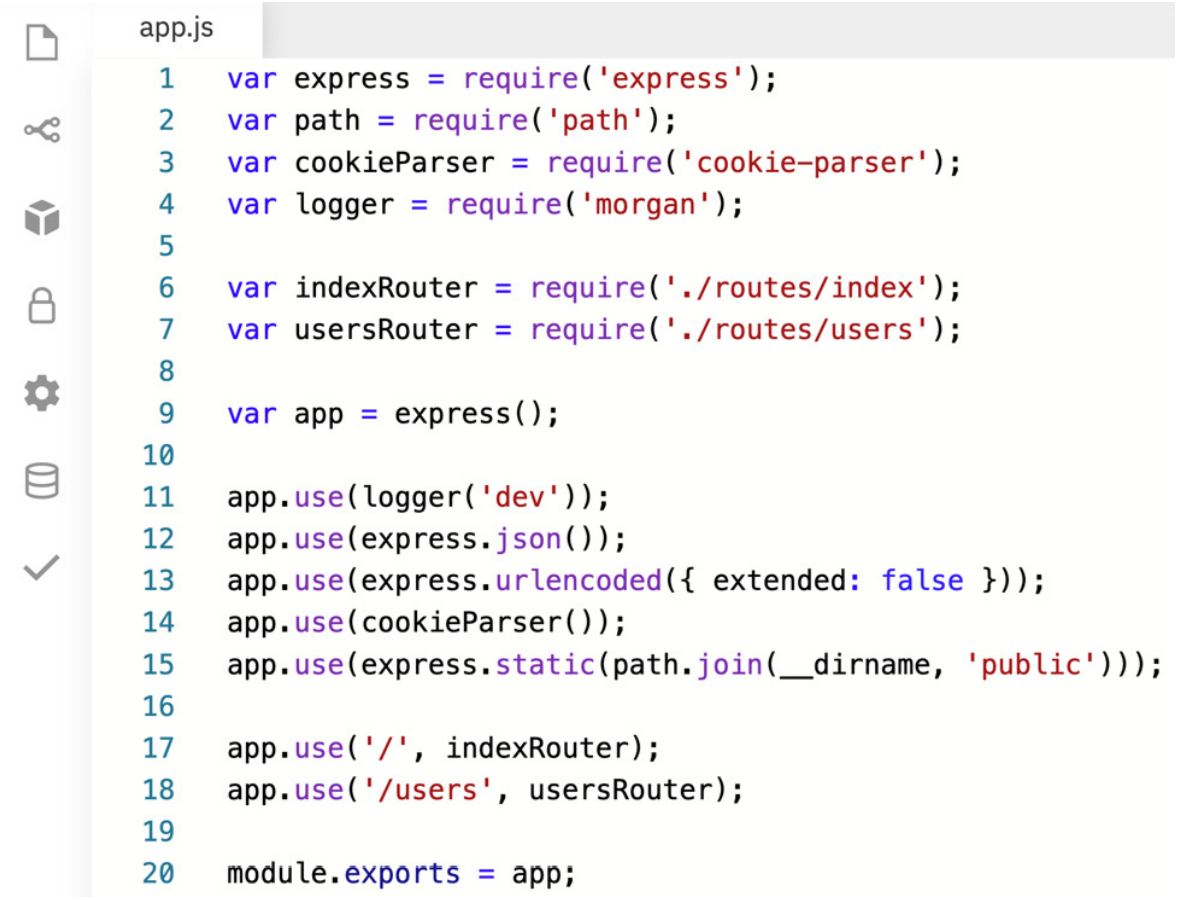



Figure 9.4 – Default app.js created by express-generator

2. Edit *line 7* and change **var usersRouter** = **require('./routes/users')** to the following:

```
var answerRouter = require('./routes/answer');
```

3. Edit *line 18* and change **app.use('/users', usersRouter);** to the following:

```
app.use('/answer', answerRouter);
```

After editing lines *7* and *18*, the **app.js** file should look like the following screenshot:

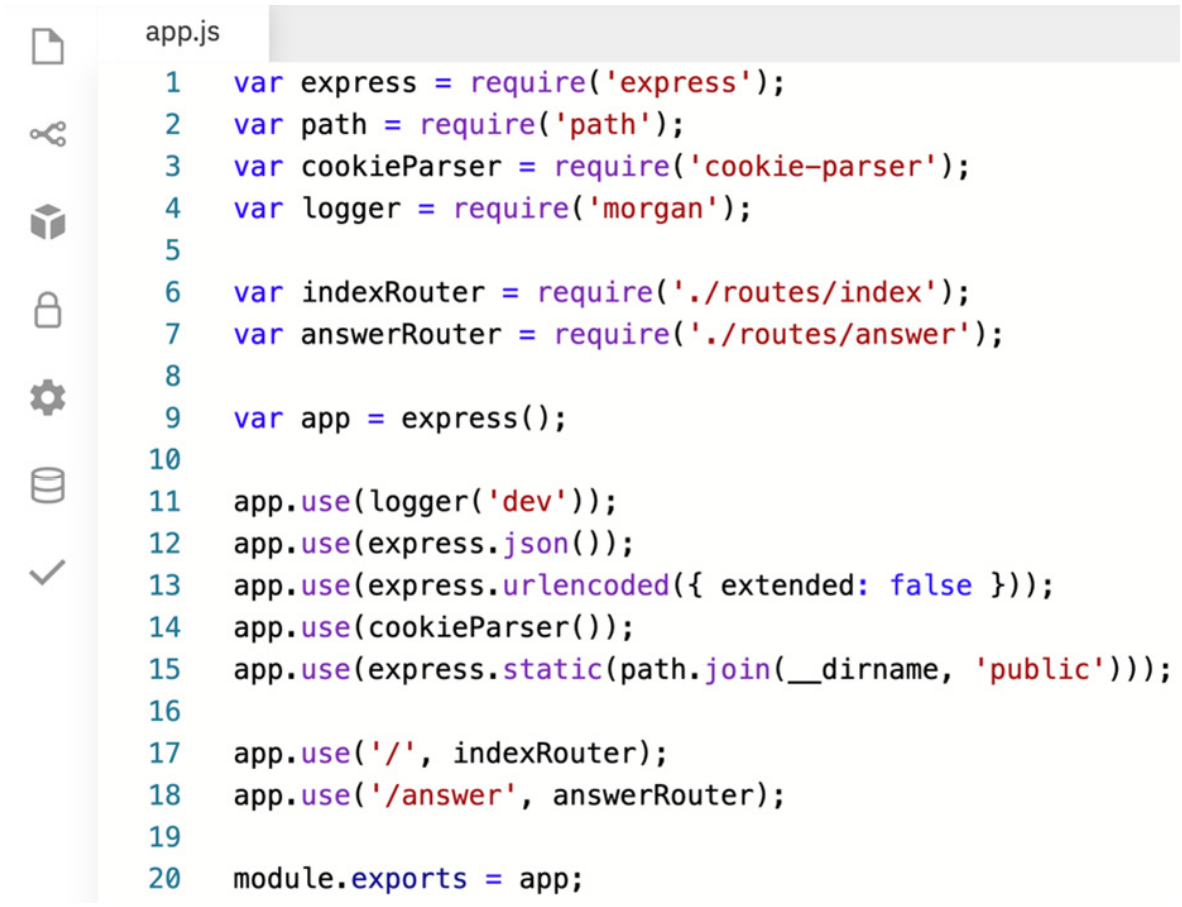



Figure 9.5 – Edited app.js file

4. Delete the **routes/users.js** file.
5. Create a new file, **routes/answer.js**.
6. Add the following code to the **answers.js** file:

```
const axios = require('axios');
const express = require('express');
const router = express.Router();

router.post('/', (req, res) => {
  res.send({answer:'placeholder for the answer'});
});

module.exports = router;
```

7. Click the **Stop** button and then **Start**. You should see the **Welcome to Express** message again.

The API endpoint we created accepts an HTTP **POST**. The endpoint URL will be the URL you can see in the address bar of the Replit.com browser followed by **/answer**. But since it accepts an HTTP POST, we'll need to test it by making a POST request. To do that, we'll use Postman.

## Testing our API with Postman

At this point, we should be able to make an HTTP POST request to our **/answer** endpoint and get a response. To complete the test, copy the app URL from the Replit.com browser (the one showing the Express welcome message) to your clipboard:

1. Open a new browser tab and log in to https://postman.com.
2. Create a new collection named **gptanswers-node**.
3. Create a new request named **test-answer-api** in the **gptanswers-node** collection to make a post to your app API endpoint. You should have copied the app URL to your clipboard, and you can paste that into Postman and add a slash and then **answer** (**/answer**). The format of the endpoint URL is as follows but where **{username}** is your Replit.com username (assuming you named the repl **gptanswers-node**):

   ```
   https://gptanswers-node.{username}.repl.co
   ```

4. Below the endpoint URL input, select the **Body** tab, choose the **Raw** radio button, and choose **JSON** from the dropdown for the content type.
5. Finally, add the following JSON for the request body:

   ```
   {
       "question" : "How old are you?"
   }
   ```

   After setting up the request in Postman, it should look something like the following screenshot:

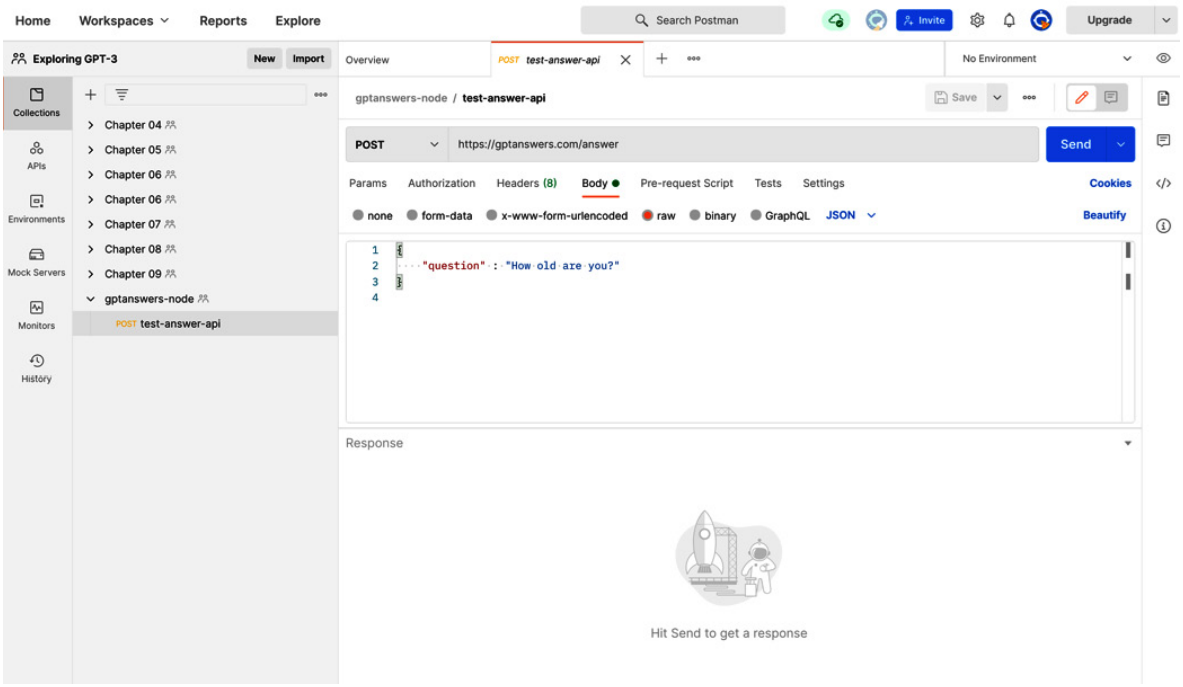



Figure 9.6 – Postman request to test the GPT-CV app API endpoint

6. Click the blue **Send** button to submit the request and review the response, which should be the following:

```
{
    "answer": "placeholder for the answer"
}
```

Okay, now that we have our API endpoint responding, we'll move on to creating a web form to call the API.

## Creating the GPT Answers user interface

Now let's create a simple web form interface that will let users submit a question to get an answer from our API. We'll start by adding **UIkit** – a popular lightweight frontend framework that you can learn more about at https://getuikit.com/. We'll also use **Axios** to make HTTP calls to the app API using a bit of JavaScript.

UIkit will make it easy to create a simple but clean and modern look for our app. You can download UIkit for free from https://getuikit.com. Or you can use a hosted version that is available from https://jsdeliver.com, a free **Content Delivery Network** (**CDN**) for open source projects, and that's what we'll be using.

To add UIkit, do the following:

1. Open the **public/index.html** file.

2. Replace the URL for the style sheet with https://cdn.jsdelivr.net/npm/uikit@3.6.22/dist/css/uikit.min.css.

3. Replace everything between the <**body**> tag and the <**/body**> tag with the following:

```
  <div class="uk-section uk-section-large uk-height-viewport">

  <div class="uk-container uk-text-center uk-padding-large">

        <h1 class="uk-heading-medium"><strong>GPT</strong> Answers </h1>

        <p class="uk-text-lead">An Example Knowledge Base App Powered by GPT-3</p>

    </div>

    <div class="uk-container uk-text-center">

        <form class="uk-grid-collapse" uk-grid>

          <div class="uk-width-1-1 ">

            <input id="question" class="uk-input uk-width-1-3" type="text">

            <button type="submit" class="uk-button uk-button-default uk-width-1-5">Get
Answer</button>

          </div>

        </form>

    </div>

    <div class="uk-container uk-text-center uk-padding">

      <div class="uk-inline">

        <div id="answer" class="uk-flex uk-flex-center uk-flex-middle uk-padding uk-
width-expand"></div>

      </div>

    </div>

  </div>
```

4. Add the following code above the <**/body**> tag. This will add references to JavaScript files the page will use. Three of the scripts we'll get from a CDN and one, the **/javascripts/script.js** file, we'll create in the next step:

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/uikit@3.6.22/dist/js/uikit.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/uikit@3.6.22/dist/js/uikit-icons.min.js">
</script>

<script src="/javascripts/script.js"></script>
```

At this point, the code in **public/index.html** should look like the following screenshot:

public/index.html

```
<html>

<head>
  <title>GPT Answers</title>
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/uikit@3.6.22/dist/css/uikit.min.css">
</head>

<body>

  <div class="uk-section uk-section-large uk-height-viewport">
    <div class="uk-container uk-text-center uk-padding-large">
      <h1 class="uk-heading-medium"><strong>GPT</strong> Answers </h1>
      <p class="uk-text-lead">An Example Knowledge Base App Powered by GPT-3</p>
    </div>
    <div class="uk-container uk-text-center">
      <form class="uk-grid-collapse" uk-grid>
        <div class="uk-width-1-1 ">
          <input id="question" class="uk-input uk-width-1-3" type="text">
          <button type="submit" class="uk-button uk-button-default uk-width-1-5">Get Answer</button>
        </div>
      </form>
    </div>
    <div class="uk-container uk-text-center uk-padding">
      <div class="uk-inline">
        <div id="answer" class="uk-flex uk-flex-center uk-flex-middle uk-padding uk-width-expand"></div>
      </div>
    </div>
  </div>

  <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/uikit@3.6.22/dist/js/uikit.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/uikit@3.6.22/dist/js/uikit-icons.min.js"></script>
  <script src="/javascripts/script.js"></script>
</body>

</html>
```

Figure 9.7 – Completed index.html code

If you stop and restart the Express server by clicking the **Stop** button followed by clicking the **Run** button, you should see the home screen updated to look like the following screenshot:

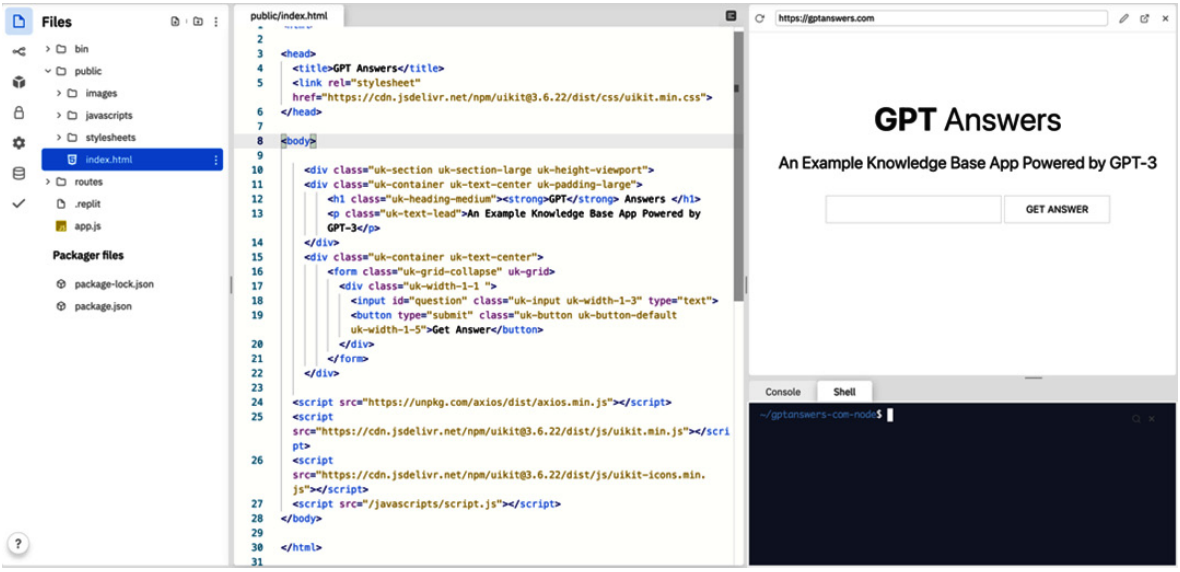


Figure 9.8 – Question input form

Now we need to add some JavaScript to make the call to the app API.

5. Create the **javascripts/script.js** file and add the following code.
6. First, we'll add two variables to hold the HTML form and answer **div**:

```
const form = document.querySelector('form');
const answer = document.querySelector('#answer');
```

7. Next, we'll add code that will fire when the form is submitted:

```
const formEvent = form.addEventListener('submit', event => {
  event.preventDefault();
  const question = document.querySelector('#question');
  if (question.value) {
    askQuestion(question.value);
  } else {
    answer.innerHTML = "You need to enter a question to get an answer.";
    answer.classList.add("error");
  }
});
```

8. The following code will append the text to the answer **div**:

```
const appendAnswer = (result) => {
  answer.innerHTML = `<p>${result.answer}</p>`;
};
```

9. Finally, we'll add a function to call the app API endpoint using Axios:

```
const askQuestion = (question) => {
  const params = {
    method: 'post',
    url: '/answer',
    headers: {
      'content-type': 'application/json'
    },
    data: { question }
  };
  axios(params)
    .then(response => {
```

```
      const answer = response.data;
      appendAnswer(answer);
    })
    .catch(error => console.error(error));
};
```

10. Now we can test it by clicking the **Stop** button and then the **Start** button. Then, in the browser pane, enter a question into the textbox and click the **GET ANSWER** button. You should see the API respond back with **placeholder for the answer** as shown in the following screenshot:


Figure 9.9 – Testing the web UI with the placeholder API code

At this point, we have the basic framework for our application in place. What we need to do next is write a bit of code to pass the question on to the OpenAI API Answers endpoint.

## Integrating the Answers endpoint

Now we'll go back and add some code to our **routes/answer.js** file so that it calls the OpenAI Answers endpoint to answer the user's question, rather than returning the placeholder text:

1. Open the **routes/answer.js** file and do the following.
2. Delete all of the code after *line 5*.
3. Starting on *line 5*, add the following code followed by a line return:

```
const apiKey = process.env.OPENAI_API_KEY;
const client = axios.create({
  headers: { 'Authorization': 'Bearer ' + apiKey }
});
```

4. Next, add the following code with a line return after it:

```
const documents = [
  "I am a day older than I was yesterday.<|endoftext|>",
  "I build applications that use GPT-3.<|endoftext|>",
  "My preferred programming is Python.<|endoftext|>"
]
```

5. Add the following code starting on *line 16* followed by a line return:

```
const endpoint = 'https://api.openai.com/v1/answers';
```

6. Next, beginning on *line 18*, add the following to complete the code:

```
router.post('/', (req, res) => {
  // call the OpenAI API
    const data = {
    "documents": documents,
    "question": req.body.question,
    "search_model": "ada",
    "model": "curie",
    "examples_context": "My favorite programming language is Python.",
    "examples": [["How old are you?", "I'm a day older than I was yesterday."], ["What languages do you know?", "I speak English and write code in Python."]],
    "max_tokens": 15,
    "temperature": 0,
    "return_prompt": false,
    "expand": ["completion"],
    "stop": ["\n", "<|endoftext|>"],
  }
  client.post(endpoint, data)
    .then(result => {
      res.send({"answer" : result.data.answers[0]})
    }).catch(result => {
      res.send({"answer" : "Sorry, I don't have an answer."})
    });
});
module.exports = router;
```

When you're done editing **routes/answer.js**, the file will look like the following screenshot:


Figure 9.10 – Edited routes/answer.js file

We're just about done. The final step before testing is to add our OpenAI API key as an environment variable.

7. Add your OpenAI API key as a secret for the REPL by clicking the padlock icon and adding a key with the name **OPENAI_API_KEY** and the value of your OpenAI API key, like the example in the following screenshot:

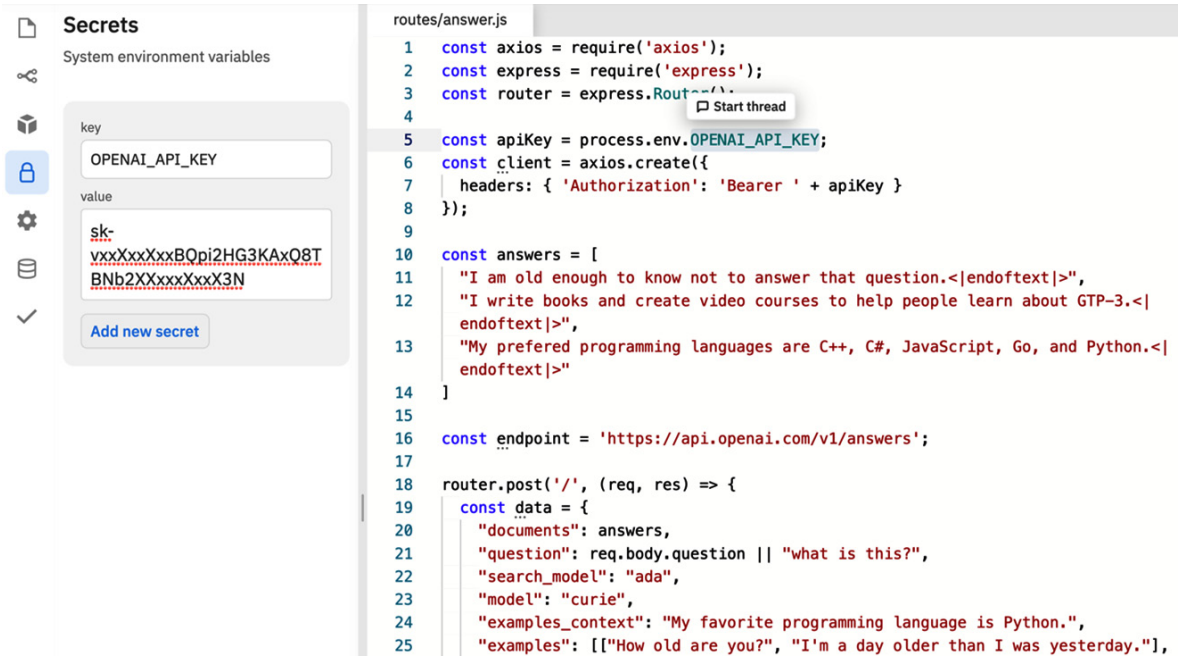


Figure 9.11 – Add a secret for your OpenAI API key

8. Click the **Stop** button followed by **Run** to restart Express and then enter **What is your favorite food?** into the question text field and click the **GET ANSWER** button. You should see something like the following screenshot – an answer coming from GPT-3:


Figure 9.12 – Answer from GPT-3

We now have a simple but functional GPT-3-powered question-answering app. However, you might be wondering why we're getting a response (an answer) for **What is your favorite food?** when we haven't provided an answer for that. We'll discuss that next.

## Generating relevant and factual answers

GPT-3 is a language model – it predicts the statistical likelihood of the text that should follow the prompt text it was provided. It's not a knowledge base in the sense that it's concerned much with the factual accuracy of the responses it generates. That doesn't mean it won't generate factual answers; it just means you can't count on the answers being accurate all of the time. But the Answers endpoint can provide a lot of control over the accuracy or relevancy of the answers that will get generated.

As we discussed earlier in *Introducing the Answers endpoint*, answers will be generated from the documents we provide. At this point, we're providing documents as part of the endpoint request. Using that method, if the answer can't be derived from the documents, the engine defined by the **model** parameter will be used to generate an answer. You can find that set in the **routes/answer.js** file – we used the **Curie** engine. But let's say we only want answers to be derived from our documents and we don't want to return answers otherwise. While we don't have 100% control over that, we can use a pre-uploaded file to get us pretty close.

When pre-uploaded files are used with the Answers endpoint, you're not limited to just 200 documents like you are when you provide documents with the HTTP request. In fact, a pre-uploaded

file might contain a very large number of documents because you can have up to 1 GB of file space per organization. Because a file might contain a very large number of documents, a keyword filter is applied to the documents in the file to narrow the possible documents that could be used for the answer. From there, the documents are ranked and then used by the engine defined by the model parameter to generate the answer. When you send documents with a request parameter, the keyword filtering is skipped because the number of documents you can send is limited to 200. For our GPT Answers app, keyword filtering will help us decrease the chances that irrelevant questions will be answered. So, let's take a look at using files with the Answers endpoint.

## Using files with the Answers endpoint

To use documents from a file, the first thing we'll need to do is get the file uploaded to OpenAI so it can be used by the Answers endpoint. The process involves creating a file containing our documents, then using the **files endpoint** to upload the file and get a file ID that can be used when we make requests to the Answers endpoint. To create and upload the answer file, complete the following steps:

1. Create a new **jsonl** file named **answers.jsonl** and some answers for the file in the following format:

```
{"text": "I am a day younger than I will be tomorrow"}
{"text": "I like to code in Python."}
{"text": "My favorite food is carrot cake."}
```

2. Create another new file named **files-upload.js**.
3. Add the following code in **file-upload.js**:
4. Require a few modules that will be used:

```
const fs = require('fs');
const axios = require('axios');
const FormData = require('form-data');
```

5. Next, add the following code to read in the **jsonl** data for the request:

```
const data = new FormData();
data.append('purpose', 'answers');
data.append('file', fs.createReadStream('answers.jsonl'));
```

6. Add a variable for the HTTP request parameters:

```
const params = {
method: 'post',
url: 'https://api.openai.com/v1/files',
headers: {
  'Authorization': 'Bearer ' + process.env.OPENAI_API_KEY,
```

```
  ...data.getHeaders()
},
data : data
}
```

7. Finally, add code to make the HTTP request and log results:

```
axios(params)
  .then(function(response) {
    console.log(response.data);
  })
  .catch(function(error) {
    console.log(error);
  });
```

When you're done editing **files-upload.js**, it should look like the code in the following screenshot:

files-upload.js

```
const axios = require('axios');
const FormData = require('form-data');

const data = new FormData();
data.append('purpose', 'answers');
data.append('file', fs.createReadStream('answers.jsonl'));

const params = {
  method: 'post',
  url: 'https://api.openai.com/v1/files',
  headers: {
    'Authorization': 'Bearer ' + process.env.OPENAI_API_KEY,
    ...data.getHeaders()
  },
  data: data
}

axios(params)
  .then(function(response) {
    console.log(response.data);
  })
  .catch(function(error) {
    console.log(error.message);
  });

```

Figure 9.13 – Completed code for file-upload.js

8. In the output pane, click on the **Shell** tab.
9. At the **~/gptanswers-node** prompt, enter the following command with your OpenAI API key:

   `export OPENAI_API_KEY="your-api-key-goes-here"`

10. Next, enter the following command in the shell:

    `node files-upload.js`

    After running the previous shell commands, you should see a result like the output in the following screenshot:

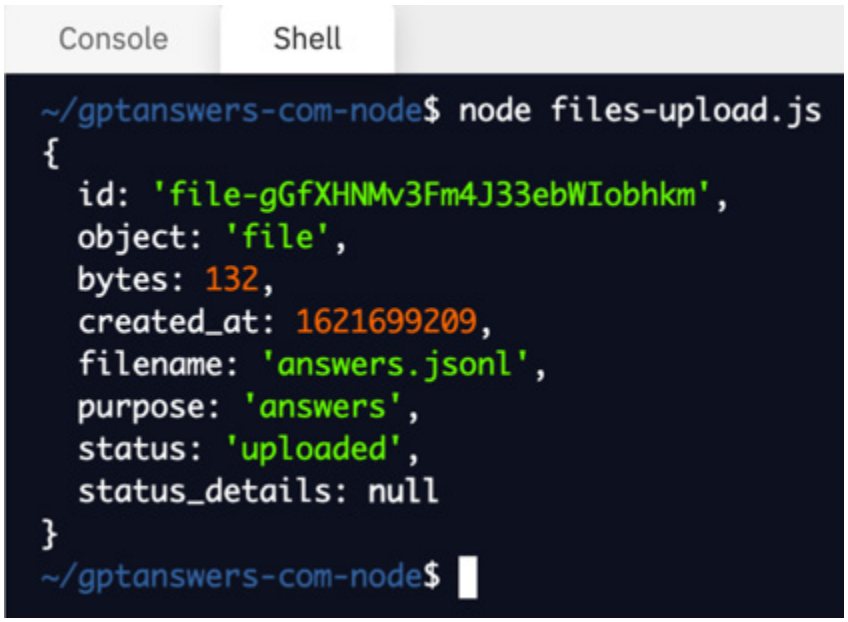



Figure 9.14 – Shell output from files-upload.js

11. Copy the **id** value from the JSON results (the value that begins with **file**-) to your clipboard.
12. Click on the padlock icon and create a new secret/environment variable named **ANSWERS_FILE** and paste the ID value you copied in the last step into the value input, then click the **Add new secret** button.
13. Open **routes/answer.js** and rename the **documents** parameter on *line 20* to **file**. Then replace the **documents** value with **process.env.ANSWERS_FILE**.

    After the previous update, *line 20* should look like *line 20* in the following screenshot:

```
18  router.post('/', (req, res) => {
19      const data = {
20      "file": process.env.ANSWERS_FILE,
21      "question": req.body.question,
22      "search_model": "ada",
23      "model": "curie",
24      "examples_context": "My favorite programming language is Python.",
25      "examples": [["How old are you?", "I'm a day older than I was yesterday."],
        ["What languages do you know?", "I speak English and write code in Python."]]
        ,
26      "max_tokens": 15,
27      "temperature": 0,
28      "return_prompt": false,
29      "expand": ["completion"],
30      "stop": ["\n", "<|endoftext|>"],
31    }
```

Figure 9.15 – The Answers endpoint parameters using the file parameter

At this point, you are ready to test.

14. Click the **Run** button, then enter **What is your favorite food?** in the question input followed by clicking the **GET ANSWER** button. This time you'll notice that the answer was generated from our answers file, as the following screenshot shows:


Figure 9.16 – An answer generated from the answers file

But now if you enter a question that's not related at all to the data in your file, the response will be **Sorry, I don't have an answer**. You can test this by asking something such as **Do you sell this one in red?** You should see a result like the one shown in the following screenshot:


Figure 9.17 – A question that can't be answered

An important thing to keep in mind is that both the answers file and the completion engine (**Curie** in our case) are used to generate the answer. So, it's possible to get an answer that isn't represented in your file. However, the more data you have in your answers file, the less likely that will be. But because we just have three documents in our answers file now, if you ask a question such as **What is**

**your favorite vacation spot?**, you might see a response with an answer that isn't defined in your answers file, as the following screenshot shows:


Figure 9.18 – An answer that isn't from the answers file

So, even though we're providing answers in our answers file, that doesn't guarantee GPT-3 won't generate an answer that isn't accurate. But we'll discuss this more later in this chapter and in *Chapter 10*, *Going Live with OpenAI-Powered Apps*.

At this point, we have a fully functional app. Of course, there is a lot more we could add to polish our app, but the core functionality is in place. The main thing you'll need to do is add more documents to the answers file. To do that, complete the following steps each time you want to add new data:

1. Add new documents to the **answers.jsonl** file.
2. Open the shell.
3. Run the following shell command to set your API key as an environment variable that the shell can access:

   ```
   export OPENAI_API_KEY="your-api-key-goes-here"
   ```

4. Run the following command in the shell to execute **files-upload.js**:

   ```
   node files-upload.js
   ```

5. Copy the file **ID** and replace the **ANSWERS_FILE** environment variable by clicking the padlock icon and replacing the value with the **ANSWERS_FILE** secret.
6. Click the **Stop** button and then the **Run** button to restart Express.

Again, more data in your answers file will minimize the chances of non-factual answers. But it's still possible that GPT-3 will generate answers that clearly aren't from your answers file. So, it's still important to consider content filtering, which is why we'll be covering that more in the next chapter.

## Summary

In this chapter, we introduced the Answers endpoint and used Node.js/JavaScript to build a simple but functional web application that can answer questions from documents we provide. For our applications, we created an API that acts as a proxy to the OpenAI API and an HTML page that provides the user interface.

In the next chapter, we will discuss the OpenAI app review process and implement a few modifications to our application based on recommendations from OpenAI. Then, we'll cover the steps necessary for going live!

# *Chapter 10*: Going Live with OpenAI-Powered Apps

Before going live with apps that use the OpenAI API, they must be approved for publishing by OpenAI. The approval process helps prevent the OpenAI API from being misused either intentionally or accidentally. It also helps app providers, and OpenAI, plan for resource requirements to ensure the app performs well at launch, and as usage grows.

In this chapter, we'll discuss OpenAI application use case guidelines along with the review and approval process. Then we'll discuss changes to our GPT Answers app based on OpenAI guidelines. Finally, we'll implement the suggested updates and discuss the process for submitting our app for review, and hopefully, approval!

The topics we'll cover are the following:

- Going live
- Understanding use case guidelines
- Addressing potential approval issues
- Completing the pre-launch review request

## Technical requirements

This chapter requires that you have access to the **OpenAI API**. You can request access by visiting https://openai.com.

## Going live

OpenAI defines a live application as any application that is serving API outputs to more than five people. This includes people in your company or organization. So, even a private beta app is considered live if it is using the OpenAI API and has more than five users. To move beyond this limit, your app needs to be reviewed and approved by OpenAI.

Going live without approval from OpenAI could result in your API key being revoked immediately, and without warning. Further, going live without approval could possibly cause your account to be permanently blocked from further API access. So, it's a good idea to understand the OpenAI use cases guidelines and review process.

## Understanding use case guidelines

There is a wide range of applications that could use the OpenAI API. However, not all use cases are permitted, so every application must be reviewed and approved before going live.

Every app is evaluated on a case-by-case basis, so the only way to know whether your application will be allowed is to go through the review process. However, OpenAI publishes guidelines that you can review and follow to give your app the best chances of approval. You can find the guidelines located at https://beta.openai.com/docs/use-case-guidelines. Before investing a lot of time in an app, you should first read the guidelines carefully.

We're not going to cover all the app guidelines in this section. But mostly, the guidelines relate to safety and security. Safety, as defined by OpenAI, is *Freedom from conditions that can cause physical, psychological, or social harm to people, including but not limited to death, injury, illness, distress, misinformation, or radicalization, damage to or loss of property or opportunity, or damage to the environment*.

So, apps that cheat, deceive, exploit, harass, hurt, intimidate, manipulate, mislead, steal, trick, or that could potentially cause harm or damage in any way, whether intentional or not, are not allowed. Most of the guidelines should seem pretty obvious. But some guidelines aren't so obvious. For example, in most cases, you can't build apps that generate content for Twitter tweets. This is because using AI-generated content violates Twitter's acceptable use policies. So, again, the point of this section is not to cover the specific guidelines; the point is to emphasize the importance of reviewing and understanding the guidelines before building an app. By reviewing the guidelines before you start building, you'll be able to focus on all the acceptable use cases and avoid potential approval issues. We'll look at some of the potential issues that we can address before the review process next.

## Addressing potential approval issues

After reading the OpenAI use case guidelines, we can consider how they apply to our GPT Answers app. Our application is limited to answering questions with answers that we provide in our answers file. So, it has a very limited scope and does not generate open-ended responses. Based on that, the guidelines suggest our app is *Almost-always approvable*. However, again, every app is approved on a case-by-case basis, so that's not a guarantee.

Also, we want to do everything we can as a developer to consider safety best practices. OpenAI publishes safety best practices at https://beta.openai.com/docs/safety-best-practices that will help ensure our app is safe and can't be exploited. This will also help increase the chances of our app being approved for publishing. Based on those guidelines, we're going to consider a few modifications to our GPT Answers app. Specifically, we are going to consider the following:

- Content filtering
- Input and output lengths
- Rate limiting

Let's look at each of these considerations individually and discuss how they apply to our app.

## Content filtering

Content filtering probably isn't necessary for our GPT Answers app because the completions are being generated from an answers file that we're providing – which is kind of one way to filter the output. However, we might not want to send inappropriate questions to the OpenAI API because even though the response will be safe in our case, we'll still be using tokens. So, we'll implement content filtering for the questions to check for inappropriate words.

The content filtering process flow that we'll be implementing is the following:

1. The user asks a question.
2. We check the question for profane language.
3. If profane language is detected, we display: **That's not a question we can answer**.
4. If profane language is not detected, we pass the question to the OpenAI API.

We'll use a **Node.js** library called **bad-words** to check for profanity in the question text before sending it to the OpenAI API. If profanity is found in the text, we'll politely respond with a message saying, **That's not a question we can answer**.

To implement content filtering on the question text, do the following:

1. Require the **bad-words** library on the first line of **routes/answer.js** with the following code:

```
const Filter = require('bad-words');
```

2. In the **routes/answer.js** file, add the following code above the line that begins with **const data**:

```
let filter = new Filter();
if (filter.isProfane(req.body.question)) {
    res.send({ "answer": "That's not a question we can answer."});
    return;
}
```

3. Click the **Stop** and then the **Run** button and test it by entering a question that includes profanity. You should see a result like the following screenshot:


Figure 10.1 – Filtering profanity in questions

Now we have content filtering in place for the question. If we were generating answers using the completions endpoint, we'd also want to look at using the content filter engine that we discussed in *Chapter 6*, *Content Filtering*, to apply content filtering to the answer. But again, since we are generating answers from a file we're providing, that's probably not necessary for the GPT Answers app. So, let's move on and consider input and output lengths.

## Input and output lengths

OpenAI recommends limiting both input and output lengths. Outputs can be easily limited with the **max_tokens** parameter. We've set the **max_tokens** parameter for the GPT Answers app to **150**. This is the recommended length for scoped output – like answers to questions from our answers file. This will support ~6-8 sentences for our answer text. If you have shorter answers, you can reduce the **max_tokens** length. Less is better provided you're allowing enough to fully answer the questions.

An injection attack is an attack that exploits web applications that allow untrusted or unintended input to be executed. For example, in the GPT Answers app – what if the user submits something other than a question and our backend code were to pass it on to the OpenAI API? Remember text in/text out? Although our application is tightly scoped and something other than a relevant question isn't going to return anything, it's still worth adding in a bit of code to prevent very large text inputs because the input will still use tokens. So, we'll add some code to limit the input length. The average sentence is 75-100 characters, so we'll limit the input to 150 characters to allow for longer sentences.

To limit the input length in our GPT Answers app, do the following:

1. Open **routes/answer.js** and add the following code on a new line after the line that begins with **router.post**:

```
if (req.body.question.length > 150) {
  res.send({ "answer": "Sorry. That question is too long." });
  return;
}
```

2. Stop and run the Express service by clicking the **Stop** button and then the **Run** button.
3. Enter a text input over 150 characters long into the question input and click the **GET ANSWER** button.

   You will see the form now tells the user the text they entered was too long, as shown in the following screenshot:

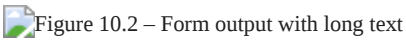



Figure 10.2 – Form output with long text

Again, while our app shouldn't generate unexpected completions, limiting the input, along with request rate limiting, will help prevent malicious attempts to exploit your app. Let's talk about rate limiting next.

## Request rate limiting

Rate limiting prevents users from making more than a predefined number of requests in a given timeframe. This prevents malicious scripts from potentially making a large number of requests to your app. We will add rate-limiting functionality to our GPT Answers app using a library available

for **Node.js** called **Express Rate Limit** and we'll set the limit to a maximum of six requests per minute – per OpenAI's suggested guidelines.

To implement rate limiting, do the following:

1. Open **app.js** and after *line 9* (or after **var app = express();**), add the following code:

```
const rateLimit = require("express-rate-limit");
const apiLimiter = rateLimit({
  windowMs: 1 * 60 * 1000,
  max: 6
});
app.use("/answer/", apiLimiter);
```

2. Open **routes/answer.js** and add the following code after the line that begins with **router.post**:

```
if (req.rateLimit.remaining == 0) {
  res.send({"answer" : "Ask me again in a minute."});
  return;
};
```

   The previous changes added a rate limit of six requests per minute. When the rate limit is hit before a request is made to the OpenAI API, we respond with a message to ask again in a minute, as in the following screenshot:

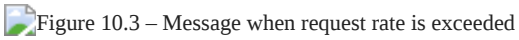


Figure 10.3 – Message when request rate is exceeded

Figure 10.3 – Message when request rate is exceeded

Because we're sending the message back in the same JSON format as an answer, the message is displayed on the form page.

### *IMPORTANT NOTE*

*You can learn more about the rate limiter library used by visiting https://www.npmjs.com/package/limiter.*

Alright, now that we have reviewed the use case guidelines and implemented some safety best practices, we're ready to discuss the OpenAI pre-launch review and approval process.

## Completing the pre-launch review request

When your app is ready to go live, you begin the approval process by completing the **Pre-Launch Review Request form** located at https://beta.openai.com/forms/pre-launch-review.

The form collects your contact information, along with a link to your LinkedIn profile, a video demo of your app, and answers to a number of specific questions about the app use case and your growth

plans. In the following sections, we'll list the current questions and example answers that might apply to the GPT Answers app.

There are a lot of questions on the Pre-Launch Review Request form, so the recommendation is to complete the questions first in a Google doc (or some other app) and then copy and paste the answers into the form when you're ready.

The form begins by collecting your contact details. After providing your contact information, the first set of questions ask about the use case at a high level.

## High-level use case questions

The high-level use case questions are pretty straightforward. However, one of the questions asks for a video demo. So, you'll need to provide a video walk-through and post it someplace like YouTube so you can provide a link. Here are the questions and some example answers:

- QUESTION: Have you reviewed OpenAI's use case guidelines?

  ANSWER: Yes

- QUESTION: Please describe what your company does.

  ANSWER: My company provides technical learning resources.

- QUESTION: Please describe what your application does.

  ANSWER: It lets users get answers to questions about me.

- QUESTION: Has your application been reviewed by OpenAI previously? What was the outcome of this review? How does this submission relate to the prior review?

  ANSWER: No

- QUESTION: Please link to a short video demonstration of your application.

  ANSWER: A link to a video demo goes here.

- QUESTION: Please share a login credential the OpenAI team can use to demo/test your application.

  ANSWER: No login credentials are required.

The next set of questions relates to security and risk mitigation. As you might guess, there are a lot of questions about security and risk mitigation. Let's take a look.

## Security and risk mitigation questions

There are 14 security and risk mitigation questions at the time this book is being written. Some of the questions you'll see are questions about content filtering, setting input and output lengths, and

request rate limiting. These are important and required for approval, which is why we implemented them in our GPT Answers app:

- QUESTION: What is the maximum number of characters that a user can insert into your application's input textboxes?

  ANSWER: 150

- QUESTION: What are the maximum output tokens for a run of your application?

  ANSWER: 150

- QUESTION: Who are the typical users of your application (for example, enterprise businesses, research labs, entrepreneurs, academics, and so on)? Do you verify or authenticate users in some way? If so, how?

  ANSWER: The most likely users are recruiters who are interested in my professional background. Users are not verified but rate limiting is in place using the user's IP address.

- QUESTION: Do users need to pay to access your application? If so, how much?

  ANSWER: No

- QUESTION: Do you implement rate-limiting for your application? If so, what are the rate limits and how are they enforced?

  ANSWER: Yes, rate limiting is enforced by IP address and requests are limited to six requests per minute.

- QUESTION: Do you implement a form of content filtration for your application? If so, what is being filtered, by what means, and how is this enforced?

  ANSWER: All answers are generated from an answers file that is pre-uploaded for use with the answers endpoint. So, content filtering is not used.

- QUESTION: Do you capture user feedback on the quality of your outputs or on other details (for instance, returning unpleasant content)? If so, how is this data monitored and acted upon?

  ANSWER: A link is provided to a Google form that lets users report any issues they might encounter.

- QUESTION: Will you monitor the usage of particular users of your application (for example, investigating spikes in volume, flagging certain keywords, et cetera)? If so, in what ways and with what enforcement mechanisms?

  ANSWER: No, because the scope is limited to just the data in the answers file that is provided by me.

- QUESTION: Is it clearly disclosed in your application that the content is generated by an AI? In what way?

  ANSWER: Yes, the text on the question input pages lets the user know the answers are generated by GPT-3.

- QUESTION: Is there a **human in the loop** in your application in some form? If so, please describe.

  ANSWER: Yes, all of the answers to questions are from an answers file that is originally created and updated by humans.

- QUESTION: Are there any other security or risk-mitigation factors you have implemented for this project? Please describe.

  ANSWER: OpenAI token usage will be closely monitored for unusual usage patterns.

- QUESTION: What, if any, is the relationship between your application and social media?

  ANSWER: None.

- QUESTION: What, if any, is the relationship between your application and political content?

  ANSWER: None.

- QUESTION: If your team has particular credentials or background that may help to mitigate any risks described above, please elaborate here.

  ANSWER: We have no specific credentials.

After the security and risk mitigation questions, you'll be asked about your growth plans.

## Growth plan questions

To manage resource requirements and limit the potential for abuse, new applications are granted a maximum spend limit. This puts a cap on the maximum number of tokens that can be processed and therefore limits the scalability. However, the maximum spend limit can be increased over time as you build a track record with your application.

Your initial spend limit will need to be approved to go live and an additional form needs to be submitted to request a spend limit increase after your application is launched. The spend limit increase form can be located at https://beta.openai.com/forms/quota-increase. To calculate your spend limit, enter a typical prompt into the Playground and set the engine and response length. Then hover over the number just below the prompt input and you'll see an estimated cost, as shown in the following screenshot:

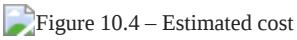


Figure 10.4 – Estimated cost

Figure 10.4 – Estimated cost

With the cost from the Playground, you can multiply by the estimated number of users and requests you'll get on a monthly basis. You will need to provide an estimate for the questions in the growth plans section.

The following questions are asked about your growth plans:

- QUESTION: What $-value monthly quota would you like to request?

  ANSWER: $18

- QUESTION: What amount of token consumption do you expect per month? For which engine(s)?

  ANSWER: ~ 1 Mn ada tokens and ~1 Mn curie tokens.

- QUESTION: To how many users (approximately) will you initially roll out your application? How do you know these users? / How will you find these users?

  ANSWER: 5,000 users who subscribe to our SaaS service

- QUESTION: Please describe your growth plans following the initial rollout.

  ANSWER: We plan to introduce the app to all new users of our service – ~500 / month

- QUESTION: If approved, on what date would you intend to launch your application?

  ANSWER: 2021-11-05

- QUESTION: You may elaborate here on the launch date above if useful.

  ANSWER: We want to launch as soon as possible.

Following the growth planning questions, there are just a few miscellaneous questions to wrap up, and you're done.

## Wrapping-up questions

The wrapping-up questions request feedback on the app development process and your experience building the app. This is an opportunity to provide OpenAI with information that can help them improve the development experience for other developers:

- QUESTION: We love feedback! Is there anything you'd like to share with the OpenAI team (for example, the hardest part of building your application or the features you would like to see)?

  ANSWER: The hardest part was figuring out the best way to do request rate limiting.

- QUESTION: Are there any collaborators you would like added to API access if we approve your application? If so, please list their emails separated by commas.

  ANSWER: No, just me at this point.

- QUESTION: We are especially interested in feedback about this process. How long did this form take you to complete? What did you find most difficult about it?

  ANSWER: It took me about 5 days. I'm new to coding so the learning curve was challenging.

- QUESTION: Anything else you would like to share?

  ANSWER: I'm really enjoying working with the API!

- QUESTION: Date of form submission

  ANSWER: 05/11/2021

After you complete and submit the Pre-Launch Review Request form, you should hear back within a few days. The response back will be an approval or a rejection with a reason for the rejection.

Depending on the rejection reason, you might be able to address any noted issues and resubmit for another review. However, hopefully, your application is approved, and you're cleared to go live!

## Summary

Congratulations, you've completed *Exploring GPT-3* and your first OpenAI-powered app! At this point, your application should be ready for the review process. Keep in mind that all apps are approved on a case-by-case basis. So, just completing the steps in this book doesn't guarantee approval. But you now understand the use case guidelines and the application review and approval process. Further, you have the knowledge and skills to address any changes that OpenAI might require to complete the review.

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

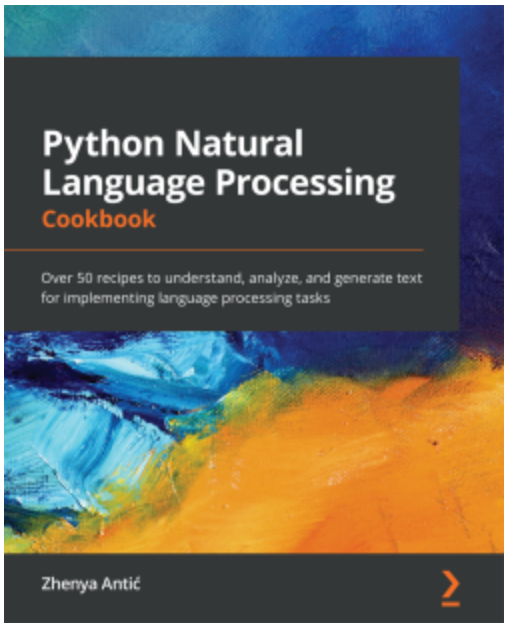
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at **customercare@packtpub.com** for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

## Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:




## Python Natural Language Processing Cookbook

Zhenya Antić

978-1-83898-731-2

- Become well-versed with basic and advanced NLP techniques in Python
- Represent grammatical information in text using spaCy, and semantic information using bag-of-words, TF-IDF, and word embeddings
- Perform text classification using different methods, including SVMs and LSTMs
- Explore different techniques for topic modeling such as K-means, LDA, NMF, and BERT
- Work with visualization techniques such as NER and word clouds for different NLP tools
- Build a basic chatbot using NLTK and Rasa
- Extract information from text using regular expression techniques and statistical and deep learning tools


**Getting Started with Google BERT**

Sudharsan Ravichandiran

ISBN: 978-1-83882-159-3

- Understand the transformer model from the ground up
- Find out how BERT works and pre-train it using masked language model (MLM) and next sentence prediction (NSP) tasks
- Get hands-on with BERT by learning to generate contextual word and sentence embeddings
- Fine-tune BERT for downstream tasks
- Get to grips with ALBERT, RoBERTa, ELECTRA, and SpanBERT models
- Get the hang of the BERT models based on knowledge distillation
- Understand cross-lingual models such as XLM and XLM-R
- Explore Sentence-BERT, VideoBERT, and BART

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!