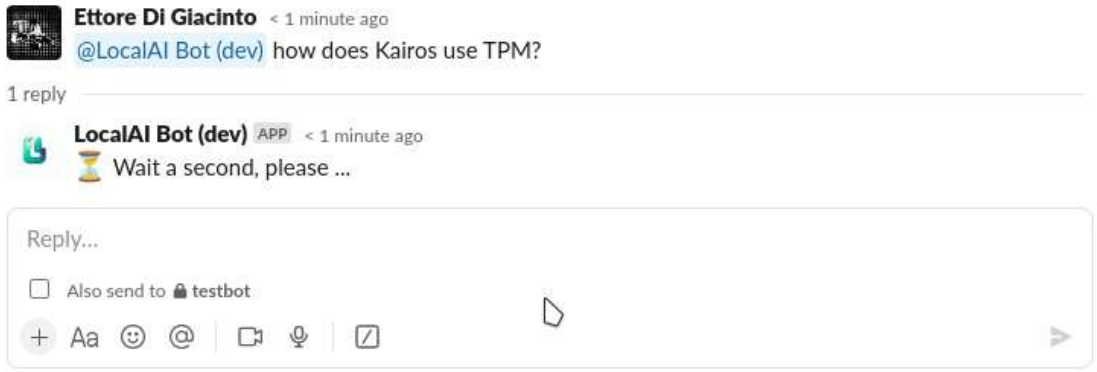
10 minutes 2077 Words

2023-06-22 00:00

Create a Question answering bot for Slack on your data, that you can run locally

There has been a lot of buzz around AI, Langchain, and the possibilities they offer nowadays. In this blog post, I will delve into the process of creating a small assistant for yourself or your team on Slack. This assistant will be able to provide answers related to your documentation.



The problem

I work at Spectro Cloud, and we have an exciting open source project called Kairos (check it out at https://kairos.io if you want to learn more about it!). Kairos is a Meta-Linux, immutable distribution designed for running Kubernetes at the Edge.

One of the challenges we face, aside from creating good documentation, is making it easily accessible and consumable for our community. Documentation evolves rapidly, and it's easy to lose track. Documentation is a critical part of any project - it's

the first thing people see when they visit your website and when a project generates a large amount of documentation, it becomes difficult not only to navigate through it but also to find exactly what you're looking for.

Nowadays, there are several services that offer question answering to improve documentation and enhance this experience. However, if you're like me and want to understand how things work behind the scenes, and perhaps build your own solution, then keep reading.

In this post, I will show you how to set up your own personal Slack bot that can answer questions based on documentation websites, GitHub issues, and code. By the end of this article, you will be able to deploy this bot using Docker or Kubernetes, either for yourself or for your team at work!

You can also try the bot live in our channel (#kairos) by joining the Kairos Slack channel and opening a thread with `@LocalAI Bot (dev)` (for example, `@LocalAI Bot (dev) does Kairos use TPM?` . Keep in mind that we self-host this on a small instance without GPU, so answers can be *slow*, but typically in range of 1-2 minutes.).

The plan

Here's how it works: our code will create a vector database that contains vector representations of different sections of the documentation, code snippets, and GitHub issues. To accomplish

this, we will use Langchain and ChromaDB to create the vector database. Langchain is a powerful library that allows interaction with LLMs (Large Language Models), and ChromaDB is a local database that can store documents in the form of embeddings. Embeddings are vectors that represent strings. Embedding databases enable semantic searching within a dataset.

For LLM inference, we will also utilize LocalAI. LocalAI allows us to run LLMs and serves as a drop-in replacement for OpenAI. Although there are other ways to interact with LLMs locally, in this case, I want a clear separation between the model execution and the application logic. This separation enables me to focus more on the core functionality of my bot. It also makes maintenance and updates easier on the go. We can replace the underlying models behind the scenes without modifying our code. Additionally, we can leverage the existing OpenAI libraries, which is quite handy. We will simulate writing code that works with OpenAI, but we will actually test it locally. This approach also allows us to use the same code with OpenAI directly or Azure, if needed.

A summary of what we will need:

- Basic knowledge of Python and Docker to create a container image for our Slack bot.
- LocalAI for running LLMs locally (no GPU required, just a modern CPU).
- An LLM model of your choice (I personally found airoboros to be quite good for Q&A).
- No OpenAI API keys or external services are needed. We will host the bot on our own without relying on remote AI APIs.

- If deploying on Kubernetes in the cloud, you will need a cluster. If running on bare metal, I've tested this on Kairos (https://kairos.io).

Tools we will use

LocalAI: It's a project created by me and it is completely community-driven. I encourage you to help and contribute if you want! LocalAI lets you run LLM from different families and it has an OpenAI compatible API endpoint which allows to be used with exiting clients. You can learn more about LocalAI here https://github.com/go-skynet/LocalAI and in the official website https://localai.io.

Langchain: is a development framework created by Harrison Chase to build applications powered by language models. See: https://python.langchain.com/docs/get_started/introduction.html

Docker: we will run the slack bot with Docker to simplify configuration. A `docker-compose.yml` file is provided as an example on how to start the slack bot and LocalAI.

How the bot works

If you're not interested in the details, you can skip directly to the Setup section below. In this section, I will explain how the bot

works.

The bot is a generic Slack bot customized to provide answers using Langchain on datasets. You can view the full code of the bot here: https://github.com/spectrocloud-labs/Slack-QA-bot. The interesting part of the bot lies in the `memory_ops.py` file (https://github.com/spectrocloud-labs/Slack-QA-bot/blob/main/app/memory_ops.py). Here's what we do in that file:

- Build a knowledge base for the bot to use for answering questions.
- When asked questions, the bot utilizes the knowledge base to enhance its answers.

Building a knowledge base

The core of the bot lies in this Python function:

```
def build_knowledgebase(sitemap):
    # Load environment variables
    repositories = os.getenv("REPOSITORIES").split(",")
    issue_repos = os.getenv("ISSUE_REPOSITORIES").split
    embeddings = HuggingFaceEmbeddings(model_name=EMBED
    chunk_size = 500
    chunk_overlap = 50

    git_loaders = []
    for repo in repositories:
        git_loader = GitLoader(
```

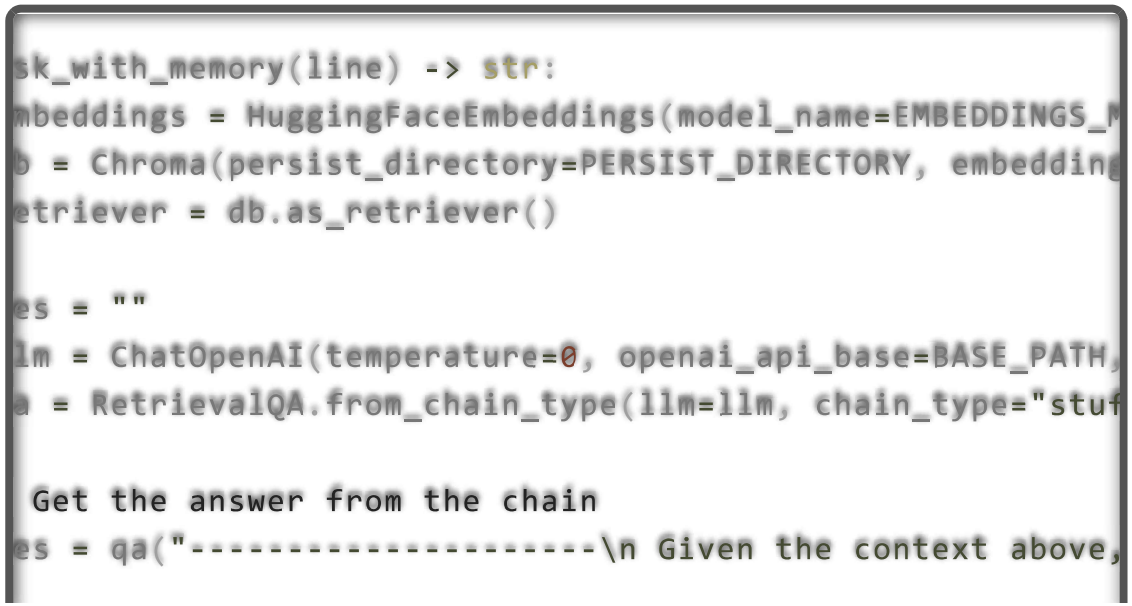
```
            clone_url=os.getenv(f"{repo}_CLONE_URL"),
            repo_path=f"/tmp/{repo}",
            branch=os.getenv(f"{repo}_BRANCH", "main")
        )
        git_loaders.append(git_loader)
    for repo in issue_repos:
        loader = GitHubIssuesLoader(
            repo=repo,
        )
        git_loaders.append(loader)

    sitemap_loader = SitemapLoader(web_path=sitemap)
    documents = []
    for git_loader in git_loaders:
        documents.extend(git_loader.load())
    documents.extend(sitemap_loader.load())

    for doc in documents:
        doc.metadata = fix_metadata(doc.metadata)

    text_splitter = RecursiveCharacterTextSplitter(chun
    texts = text_splitter.split_documents(documents)

    print(f"Creating embeddings. This may take a few mi
    db = Chroma.from_documents(texts, embeddings, persi
    db.persist()
    db = None
```

We use the locally run `HuggingFaceEmbeddings ( embeddings = HuggingFaceEmbeddings(model_name=EMBEDDINGS_MODEL_NAME) )` and Langchain to split the document into chunks. We then utilize Chroma to construct a vector database.

The code above utilizes the Github Loaders and GithubIssue loader from Langchain to retrieve information about code and GitHub issues from various GitHub repositories. The repositories can be defined via environment variables. We also use the `SitemapLoader` to ingest a `sitemap.xml` file and scrape an entire website. This is particularly useful if you already have documentation or a website.

Querying the knowledge base

Another crucial part of the code is how we interact with the AI and enhance the search results.



```
nswer, docs = res['result'], res['source_documents']
es = answer + "\n\n\n" + "Sources:\n"
 Print the relevant sources used for the answer
or document in docs:
    if "source" in document.metadata:
        res += "\n----------------------\n" + document.meta
    else:
        res += "\n----------------------\n No source availa
    res += "```\n"+document.page_content+"\n```"
eturn res
```

In this section, we load the previously created embedding database and configure the Langchain `RetrievalQA` object. Once the knowledge base has been built, we simply point to the embedding database and specify the embedding engine. In this case, we use local embeddings with HuggingFace, but other options could have been used as well (for example, LocalAI also has its own embedding mechanism).

We then configure the `llm` to use LocalAI. Note that we use ChatOpenAI and set `openai_api_base` to use LocalAI instead.

Setup

Now, let's proceed with setting up our bot! Here's what we need:

- Set up a Slack server and gain access to add new applications.
- Create a GitHub repository (optional) and obtain a Personal Access Token to fetch issues from a repository.
- Ensure your website has an accessible `sitemap.xml` file so that our bot can scrape the website content.
- Install the `docker` and `docker-compose` applications locally if missing.
- Choose a model for use with LocalAI (refer to https://github.com/go-skynet/LocalAI).

That's it! We don't need an OpenAI API key or any external services except GitHub, optionally, which we use to fetch the content we want to index.

Clone the required files

We will run everything locally using Docker. At the end of this article, I will also provide a deployment file that works with Kubernetes.

To get started, clone the LocalAI repository locally:

```
git clone https://github.com/go-skynet/LocalAI
cd LocalAI/examples/slack-qa-bot
```

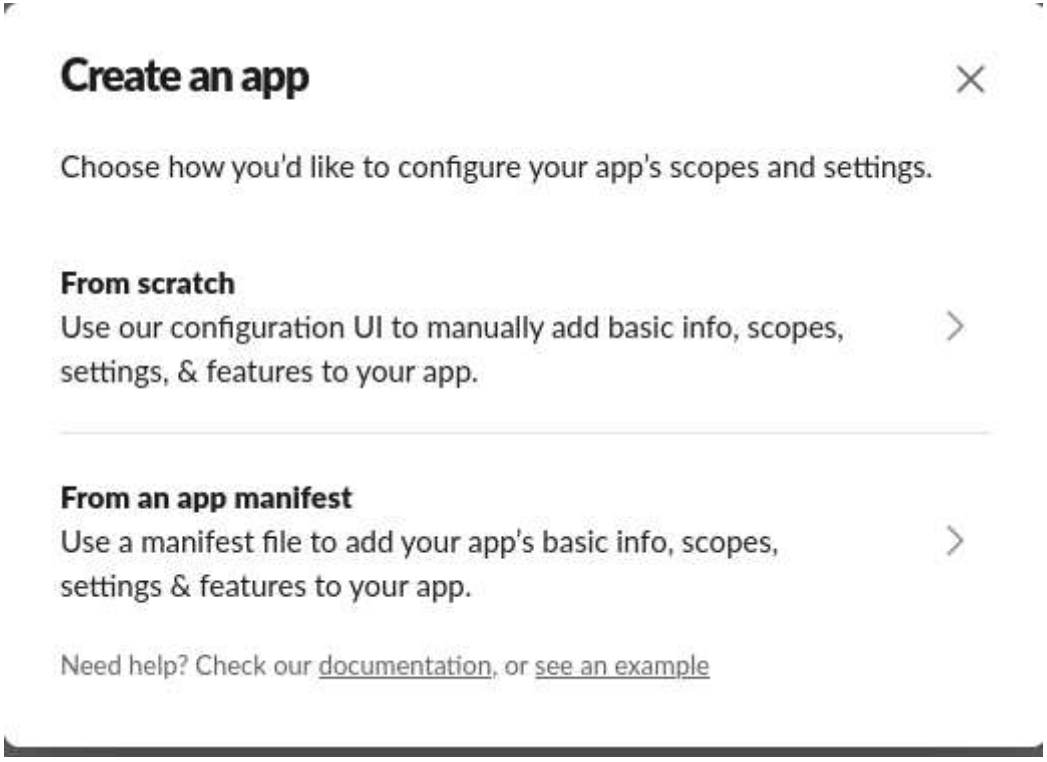
You will find a `docker-compose.yaml` file and a `.env.example` file. We need to edit the `.env` file and add the Slack tokens to allow the bot to connect.

Configuring Slack

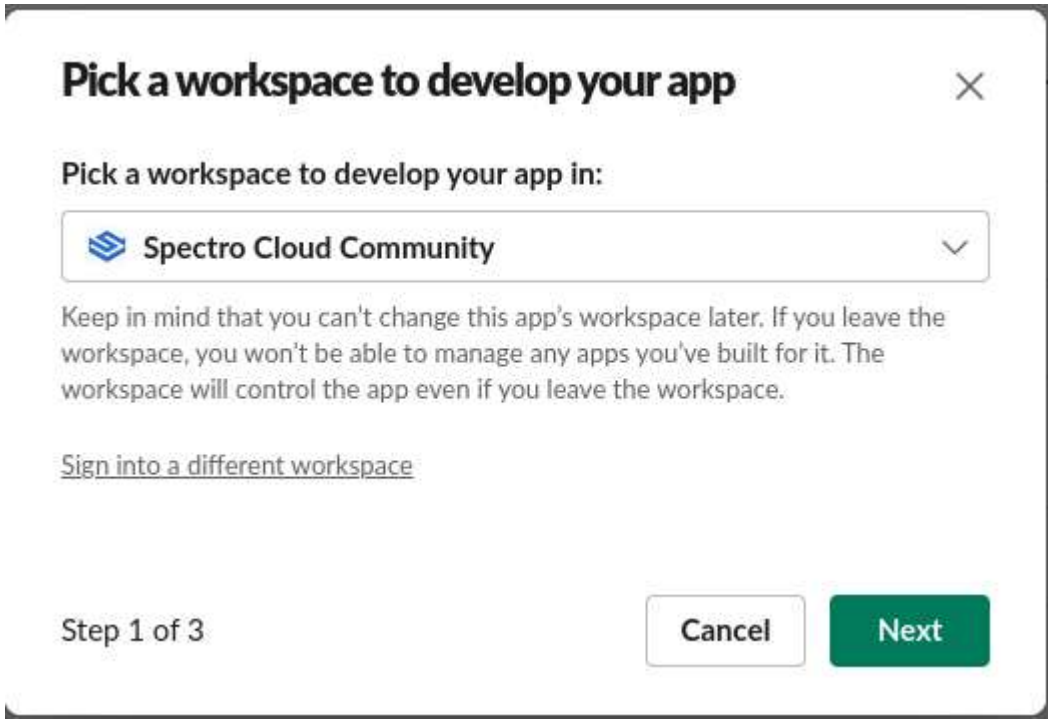
To install the bot, we need to create an application in the Slack workspace. Follow these steps:

1. Go to https://api.slack.com/apps/ and click on “Create new App”.

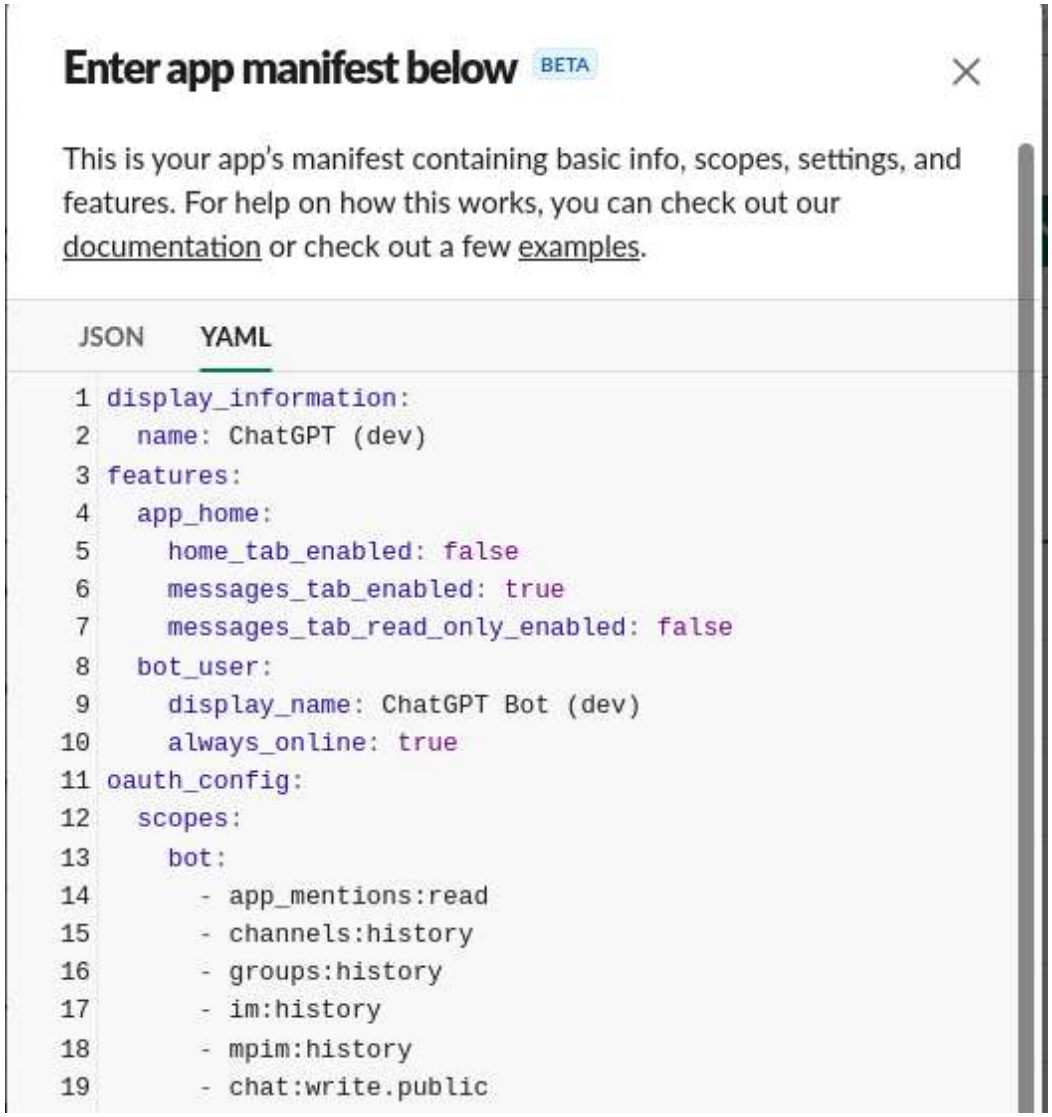
2. Select “From an app Manifest”.



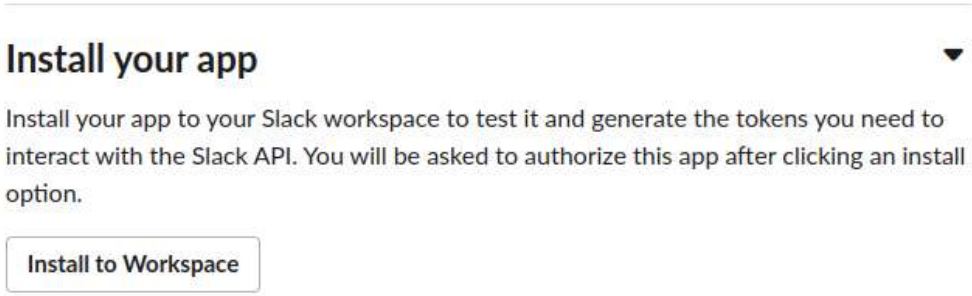
3. Choose the workspace where you want to add the bot.



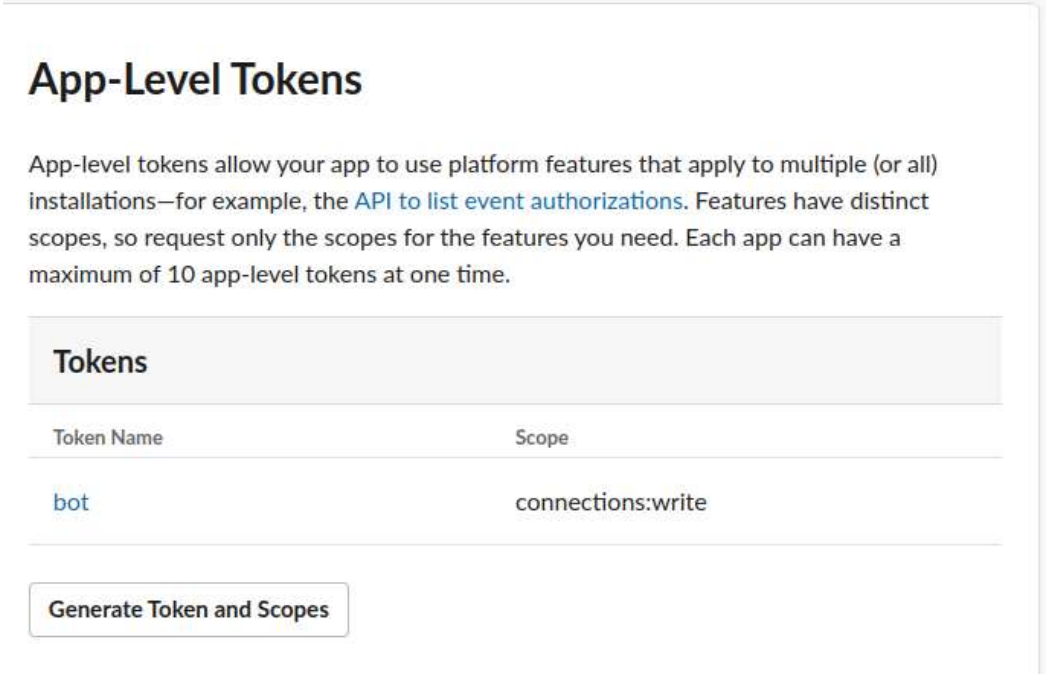
4. Copy the content of the manifest-dev.yml file from the repository and paste it into the app manifest.

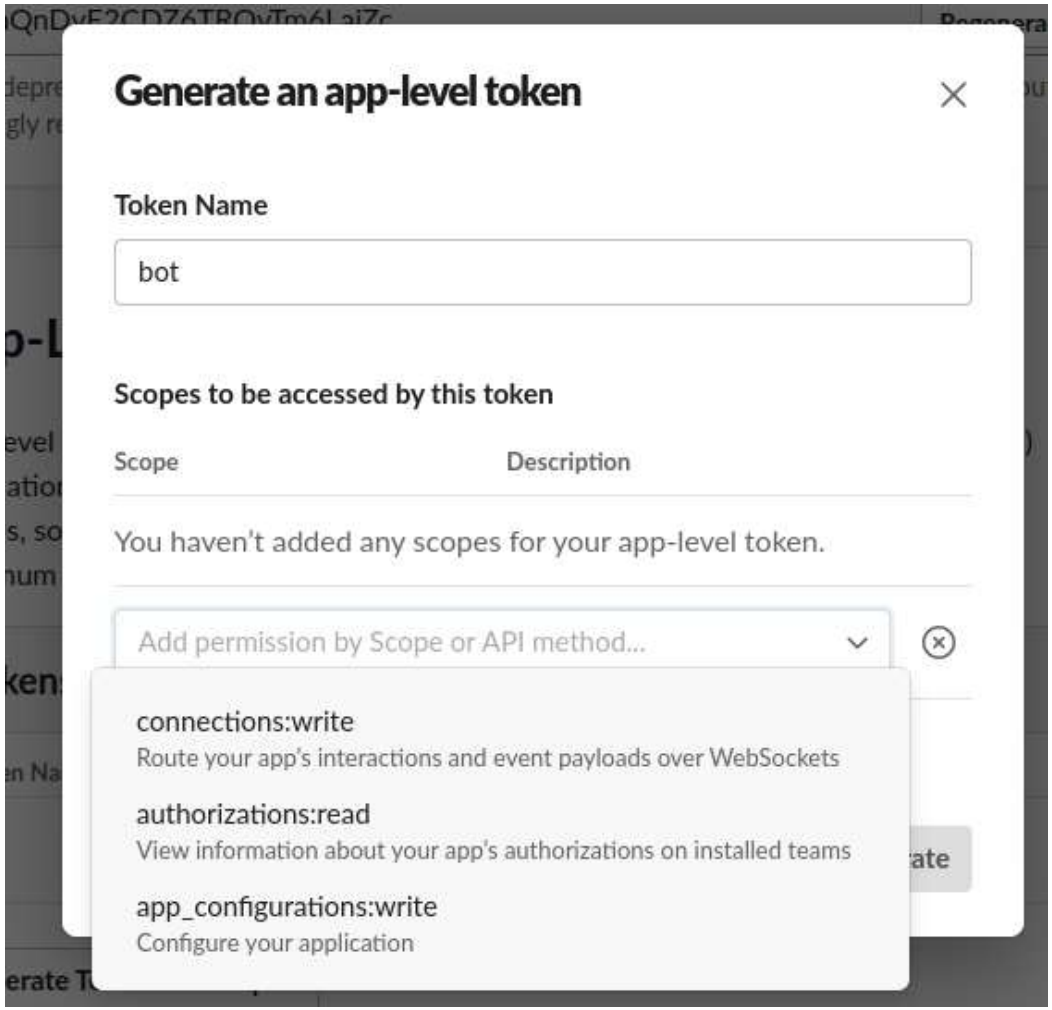


5. Install the app in your workspace.

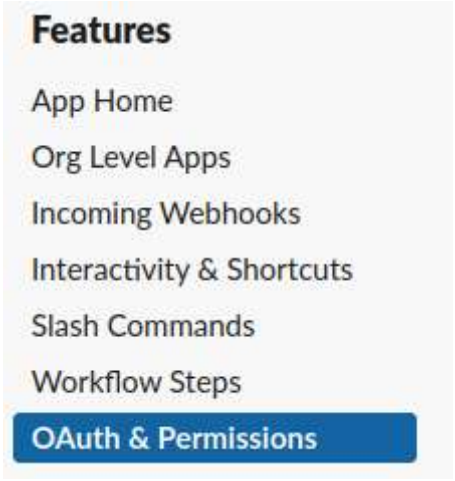


6. Create an app level token with the `connection:write` scope. Save this token as `SLACK_APP_TOKEN` .





7. Obtain the OAuth token by going to OAuth & Permissions and copying the OAuth Token. Use this token as `SLACK_BOT_TOKEN` .



Modifying the .env File

Follow these steps to modify the .env file:

1. Copy the example env file using the following command:

```
cp -rfv .env.example .env
```

2. Open the .env file and update the values of `SLACK_APP_TOKEN` and `SLACK_BOT_TOKEN` with the tokens generated in the previous steps.

3. Additionally, if needed, modify the URL of the website to be indexed and set it as the value for `SITEMAP` in the .env file.

Running with Docker Compose

To run the bot using Docker Compose, follow these steps.

Run the following command if you're using Docker and `docker-compose`:

```
docker-compose up
```

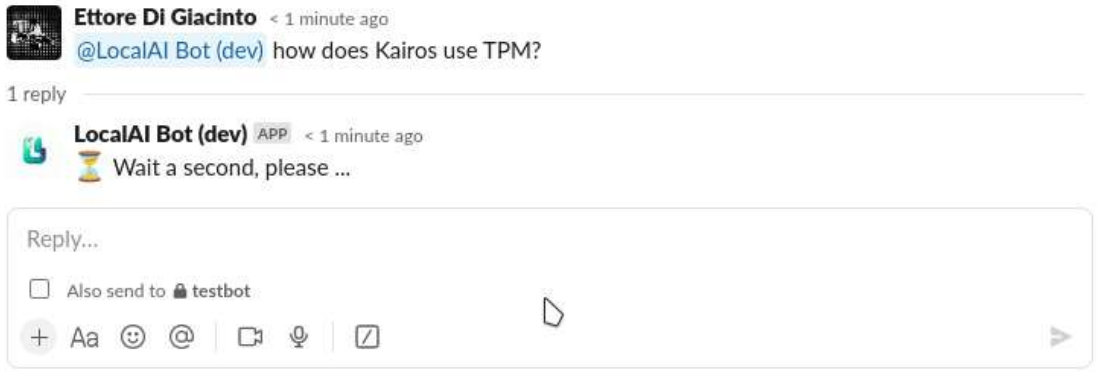
If you're running Docker with `docker compose`, use the following command:

```
docker compose up
```

By default, the local-ai setup will prepare and use the gpt4all-j model, which should work for most cases. However, if you want to change models, refer to the documentation or ask for assistance in the forums or Discord community.

Trying It Out!

Once the bot starts successfully, you can ask it questions about the documentation in the designated channel. Check out this video for an example of how it works, including linking to the relevant sources in the documentation:



Bonus: Setup other models

The `.env` file specifies to configure gpt4all automatically, however you can use other models by copying the manually in the models folder, or use the gallery:

```
# See: https://github.com/go-skynet/model-gallery
PRELOAD_MODELS=[{"url": "github:go-skynet/model-gallery/gp
```

The `PRELOAD_MODELS` environment variable in the `.env` file specifies the configuration for the `gpt-3.5-turbo` model. See

also: https://github.com/go-skynet/model-gallery in order to run other models from the gallery.

To run manually models, see the `chatbot-ui-manual` example in LocalAI, and comment the `PRELOAD_MODELS` environment variable.

Bonus: Kubernetes setup

This is a manifest which can be used as a starting point:

```
apiVersion: v1
kind: Namespace
metadata:
  name: slack-bot
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: knowledgebase
  namespace: slack-bot
  labels:
    app: localai
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
---
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: localai
  namespace: slack-bot
  labels:
    app: localai
spec:
  selector:
    matchLabels:
      app: localai
  replicas: 1
  template:
    metadata:
      labels:
        app: localai
      name: localai
    spec:
      containers:
        - name: localai-slack
          env:
          - name: OPENAI_API_KEY
            value: "x"
          - name: SLACK_APP_TOKEN
            value: "xapp-1-"
          - name: SLACK_BOT_TOKEN
            value: "xoxb-"
          - name: OPENAI_MODEL
            value: "gpt-3.5-turbo"
          - name: OPENAI_TIMEOUT_SECONDS
            value: "400"
          - name: OPENAI_SYSTEM_TEXT
```

```
    value: ""
  - name: MEMORY_DIR
    value: "/memory"
  - name: TRANSLATE_MARKDOWN
    value: "true"
  - name: OPENAI_API_BASE
    value: "http://local-ai.default.svc.cluster.lo
  - name: REPOSITORIES
    value: "KAIROS,AGENT,SDK,OSBUILDER,PACKAGES,IM
  - name: KAIROS_CLONE_URL
    value: "https://github.com/kairos-io/kairos"
  - name: KAIROS_BRANCH
    value: "master"
  - name: AGENT_CLONE_URL
    value: "https://github.com/kairos-io/kairos-ag
  - name: AGENT_BRANCH
    value: "main"
  - name: SDK_CLONE_URL
    value: "https://github.com/kairos-io/kairos-sd
  - name: SDK_BRANCH
    value: "main"
  - name: OSBUILDER_CLONE_URL
    value: "https://github.com/kairos-io/osbuilder
  - name: OSBUILDER_BRANCH
    value: "master"
  - name: PACKAGES_CLONE_URL
    value: "https://github.com/kairos-io/packages"
  - name: PACKAGES_BRANCH
    value: "main"
  - name: IMMUCORE_CLONE_URL
    value: "https://github.com/kairos-io/immucore"
```

```
        - name: IMMUCORE_BRANCH
          value: "master"
        - name: GITHUB_PERSONAL_ACCESS_TOKEN
          value: ""
        - name: ISSUE_REPOSITORIES
          value: "kairos-io/kairos"
        image: quay.io/spectrocloud-labs/slack-qa-local-
        imagePullPolicy: Always
        volumeMounts:
          - mountPath: "/memory"
            name: knowledgebase
    volumes:
      - name: knowledgebase
        persistentVolumeClaim:
          claimName: knowledgebase
```

Note:

- `OPENAI_API_BASE` is set to the default if installing the `local-ai` chart into the default namespace listening on `8080`. Specify a different LocalAI url here.

About the Author

I'm the creator of LocalAI and I've been contributing to Free Open Source software for almost 15 years, I've been working at SUSE and now I'm working at SpectroCloud.

Stay updated

If you want to stay-up-to-date on my latest posts or what I am to follow me on Twitter at @mudler_it and on Github.

READ OTHER POSTS

Question Answer... →