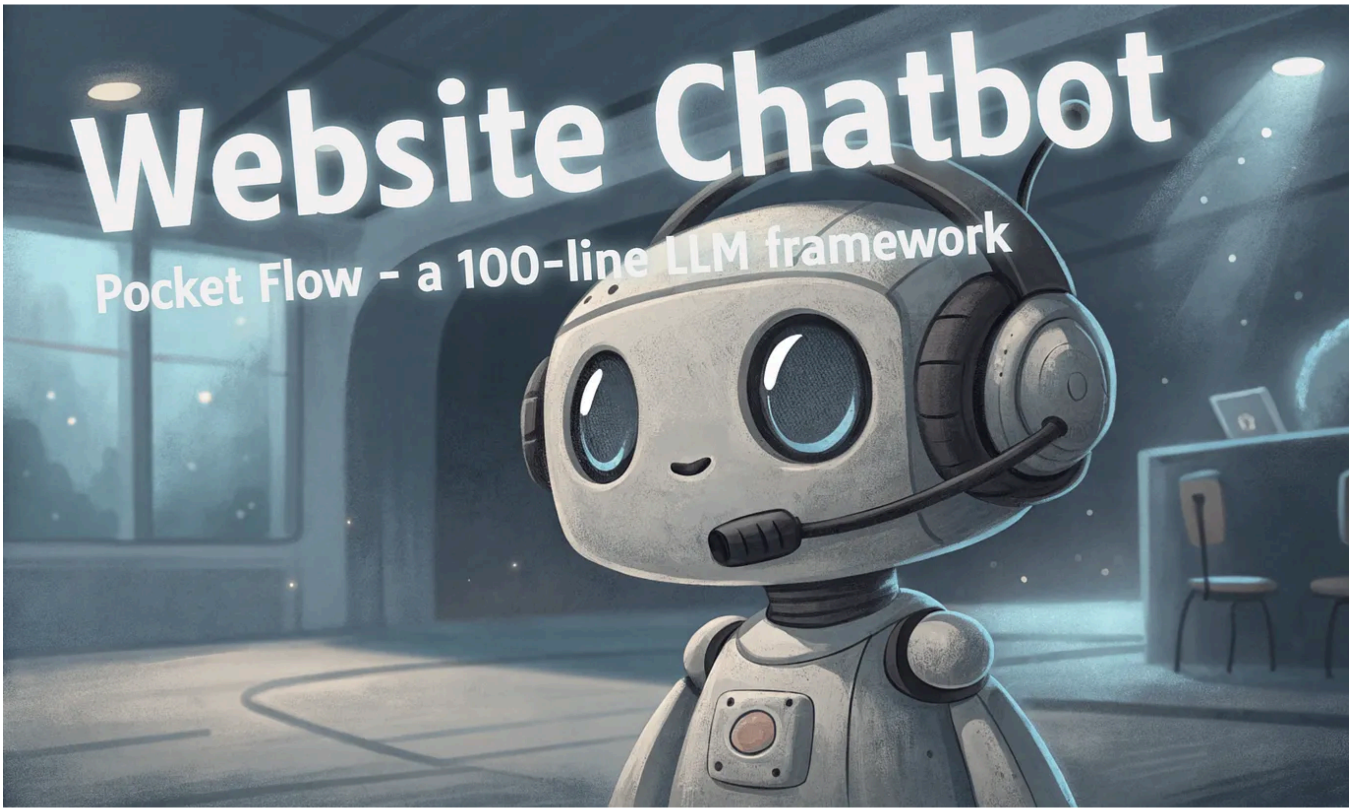
The Easiest Way to Build an AI Chatbot for You Website (Full Dev Tutorial)

ZACHARY HUANG

JUN 19, 2025

12 4 1 Sl



> *Want to build an AI chatbot for your website, but worried about the complexity? Are y picturing a maintenance nightmare of endless data updates and complex pipelines? Gc news. This tutorial shows you how to build a lightweight AI chatbot that learns directly from your live website. No vector databases, no manual updates—just a chatbot that w The project is open-*

1. That "Sim

So, you want to build u c an API, write a clever prompt, and you're basically done, right?

Except for one tiny, soul-crushing detail: Your brand-new AI knows... *nothing*.

It has no idea what your company sells, what your return policy is, or who you are. just an empty brain in a box. To make it useful, you have to feed it knowledge. Anc that's where the "simple" project becomes a total nightmare.

The Old, Broken Way to Build a Chatbot's Brain

Here’s the standard, painful process everyone seems to follow:

1. **The Scavenger Hunt.** First, you go on a company-wide scavenger hunt, diggir through folders and old emails to find every PDF, FAQ, and policy document can.
2. **The Data Janitor Job.** Then, you become a data janitor. You write a bunch of tedious scripts to chop all that messy information into clean little "chunks" th can understand.
3. **The Expensive Brain Surgery.** Finally, you perform some expensive brain surg You set up a complicated (and often pricey) "vector database" and shove all th data chunks into it.

After all that, you *finally* have a chatbot that knows things. For about a day.

And Now... Your Chatbot Is a Liar

The moment your bot goes live, it starts to rot.

The marketing team ur
Suddenly, your chatb all
talking liability. You ng
high-maintenance ch

But what if this entire approach is wrong? What if the knowledge base wasn't som clunky database you have to constantly update? What if... *the website itself* was the brain? That’s the chatbot we’re building today. A bot so simple, it feels like cheati

This project is powered by [PocketFlow](), a tiny but mighty AI framework that makes buil this kind of intelligent, looping agent incredibly straightforward. Forget vector databases manual updates. Let's build a chatbot that just works.

2. Our Solution: A "Dumb" Crawler That's Actually Smart

Let's throw that entire, complicated process in the trash. We are not going to hun documents, clean up data, or set up a single database.

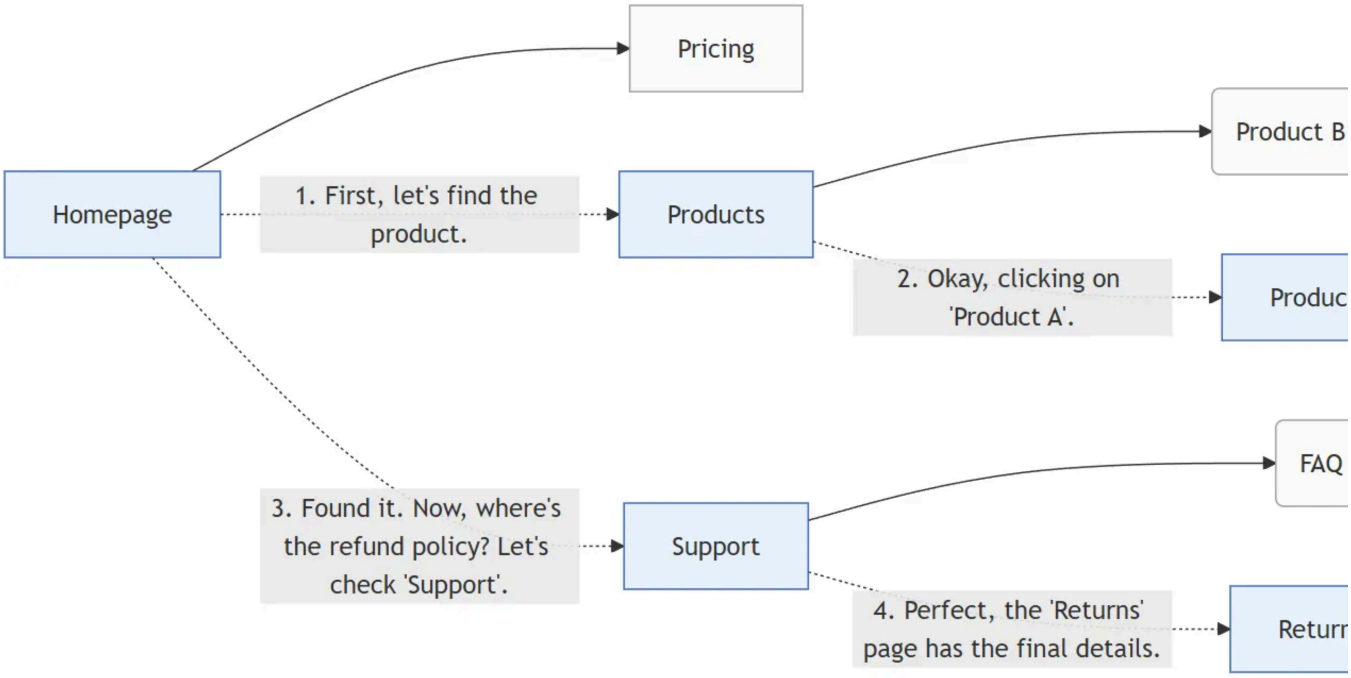
Instead, our chatbot will get its information directly from the source: your live we Think of it like this. The old way is like printing a map once a year and hoping the roads don't change. Our new way is like using Google Maps on your phone—it's always live, always current.

The Master Plan: Let the Bot Read

Our chatbot works like a very fast, very focused intern. When a user asks a questio the bot doesn't look up the answer in some dusty old database. Instead, it visits yo website and starts reading, right then and there.

Let's imagine your website has a realistic structure. A user asks a question that requires information from multiple places: "**How do I get a refund for Product A?**

The bot needs to be s ces the puzzle. In the dia ned shows the *exact path*


Pricing
Product B
Homepage
1. First, let's find the product.
Products
2. Okay, clicking on 'Product A'.
Produc
FAQ
3. Found it. Now, where's the refund policy? Let's check 'Support'.
Support
4. Perfect, the 'Returns' page has the final details.
Returr

Here's a play-by-play of the bot's clever thought process:

1. **It starts on the Homepage.** It sees both "refund" and "Product A" in the ques It decides to find the product page first to confirm the product's details.
2. **It navigates to the "Product A" page.** It reads the content and finds key info, a "30-day warranty," but it doesn't find the *process* for actually getting a refunc
3. **It intelligently changes course.** It realizes the refund steps aren't on the prod page. So, it thinks like a human would: "Okay, I need to find the general comp policies." It navigates back to the site's main "Support" section to find the off information. It doesn't need a direct link; it understands the site's structure.
4. **It finds the final piece of the puzzle.** On the Support page, it sees a link to "Shipping & Returns Policy," reads it, and learns the exact steps to submit a refund request.

Now, it combines the "30-day warranty" from the product page with the "how-to steps" from the retur

Why This is Sc

The beauty of this ap

- **Your Knowledge is Always Fresh:** You change your pricing? The bot knows instantly. You update your team bio? The bot knows that too. There is no sync step. There is no "stale data." Ever.
- **There is Zero Maintenance:** You never have to tell the bot about updates. Jus update your website like you normally would, and the chatbot takes care of th rest.

But what stops it from wandering off your site and crawling the entire internet? Simple. We give it a leash. We provide a list of approved website domains (like `yourwebsite.com`) and tell it: "You are only allowed to visit links on these sites. Don't go anywhere else."

This all sounds great, but building an agent that can make decisions and get stuck loop sounds complicated, right? You'd think you need a massive, heavy framewor manage that kind of logic.

Actually, you don't. And that's where PocketFlow comes in.

---

3. PocketFlow: The Tiny Engine That Powers O Bot

You wouldn't use a bulldozer to plant a single flower. In the same way, we don't ne massive, heavyweight AI framework for our straightforward crawling task. We ne something small, fast, and built for exactly this kind of job.

That's why we're using PocketFlow. PocketFlow is a minimalist AI framework tha just 100 lines of code ıre core ideas.

The Node: A S

In PocketFlow, each task is a **Node**. A Node is like a specialist worker who is a pro *one specific thing*. Here's what a Node looks like in the actual PocketFlow code:

```
class BaseNode:
    def __init__(self):
        self.params, self.successors = {}, {}
    def prep(self, shared): pass
    def exec(self, prep_res): pass
    def post(self, shared, prep_res, exec_res): pass
    def run(self, shared):
        p = self.prep(shared)
        e = self.exec(p)
        return self.post(shared, p, e)
```

Don't worry if `__init__` or `self` look weird; they're just Python things! The important bit is the `prep -> exec -> post` cycle:

1. `prep(shared)`: "Hey, I'm about to start. What info do I need from the `shar` whiteboard?"
2. `exec(data_from_prep)`: "Okay, I have my info. Now I'll do my main job!" calling an AI).
3. `post(shared, ..., ...)`: "Job's done! I'll write my results to the `sharec` whiteboard and tell the manager what to do next by returning a signal (like a keyword, e.g., `"explore"` or `"answer"`)."

The Shared Store: The Central Whiteboard

This is just a plain old Python dictionary (we'll call it `shared`). All our Node work can read from it and write to it. It's how they pass information—like the user's question or the list o

For our chatbot, it m

```
shared = {
    "user_question": "How do I get a refund?",
    "urls_to_process": ["https://example.com"],
    "visited_urls": set(),
    "final_answer": None
}
```

As Nodes do their work, they'll update this `shared` dictionary.

The Flow: The Workshop Manager

A `Flow` object is the manager of your workshop. You tell it which Node to start wi
and it handles the rest. When you `run` a Flow, it just keeps doing one thing over a
over:

1. Run the current Node.
2. The Node finishes and returns a *signal* (just a string, like `"explore"`).
3. The Flow looks at the Node's connections to see where that signal leads, and moves to the next Node.

Here's how tiny the `Flow` manager class actually is in PocketFlow:

```
class Flow(BaseNode):
    def __init__(self, start):
        self.start = start
    def orch(self, shared, params=None):
        curr = self.start
        while curr:
            signal = curr.run(shared)
            curr = curr.successors.get(signal or "default")
```

That's it! It starts a `w` de
there's no next node

Tiny Math Exa

Let's build a super-tiny workflow: take a number, add 3, then multiply by 2.

Worker 1: The Adder Node

```
class AddFive(BaseNode):
    def prep(self, shared):
        return shared.get("number_to_process", 0) # Read from whiteboa

    def exec(self, current_number):
        return current_number + 5 # Do the work

    def post(self, shared, prep_res, addition_result):
        shared["intermediate_result"] = addition_result # Write to
whiteboard
        print(f"AddFive Node: Added 5, result is {addition_result}")
```

Notice `post` *doesn't return anything? PocketFlow automatically treats that as the signal* `"default"`.

Worker 2: The Multiplier Node

```
class MultiplyByTwo(BaseNode):
    def prep(self, shared):
        return shared["intermediate_result"] # Read from whiteboard

    def exec(self, current_number):
        return current_number * 2 # Do the work

    def post(self, shared, prep_res, multiplication_result):
        shared["final_answer"] = multiplication_result # Write to
whiteboard
        print(f"MultiplyByTwo Node: Multiplied, final answer is
{multiplication_result}")
```

Connecting the Wor

```
# Create our spe
adder_node = AddF
multiplier_node = MultiplyByTwo()
```

```
# Connect them: after adder_node is done, run multiplier_node
adder_node.add_successor(multiplier_node)

# Create the Flow manager
math_flow = Flow(start=adder_node)

# Create the whiteboard with our starting number
shared_math_data = {"number_to_process": 10}
print(f"Starting math game with: {shared_math_data}")

# Run the flow!
math_flow.run(shared_math_data)

print(f"Math game finished. Whiteboard looks like: {shared_math_data}"
```

If you run this, you get exactly what you'd expect:

```
Starting math game with: {'number_to_process': 10}
AddFive Node: Added 5, result is 15
MultiplyByTwo Node: Multiplied, final answer is 30
Math game finished. Whiteboard looks like: {'number_to_process': 10,
'intermediate_result': 15, 'final_answer': 30}
```

See? Each Node is simple. The `shared` dictionary carries the data. The `Flow` mar makes sure `AddFive` runs, then `MultiplyByTwo`.

Now, just swap our math workers for chatbot workers:

- `AddFive` becomes `CrawlAndExtract`.
- `MultiplyByTwo` becomes `AgentDecision`.
- And instead of ju `"explore"` to l

The pattern is exactl th "workers" that make our chatbot come to life.

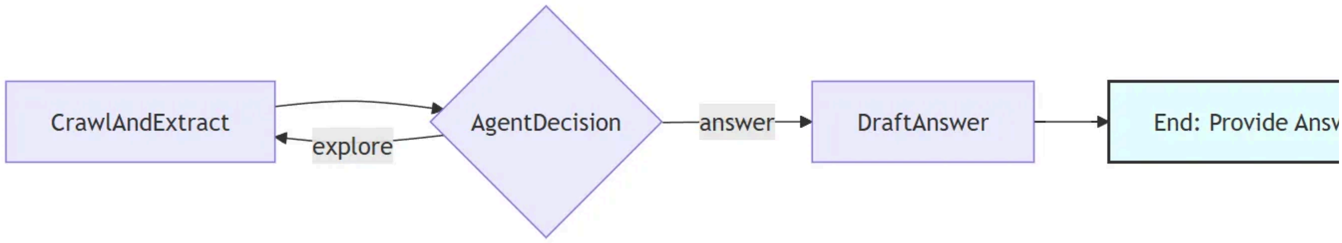
4. Building the Brain: A Look Under the Hood

Alright, theory's over. Let's look at the actual code that makes our chatbot's brain By the end of this section, you'll understand the entire backend, from the high-lev workflow down to the individual "workers."

(Note: We've simplified the code below to focus on the core ideas. For the complete, unabridged version, you can view the full code in the [project on GitHub](https://github.com).)

The Game Plan

First, let's look at our workflow diagram. This is the entire brain of our operation: simple loop.



The Assembly Line Instructions (`flow.py`)

Before we build the individual workers, let's look at the instructions that tell them to work together. This is our `flow.py` file, and it's the "manager" that directs the assembly line.

```
# From flow.py
from pocketflow import Flow
from nodes import CrawlAndExtract, AgentDecision, DraftAnswer

# 1. Create an ir
crawl_node = Crav
agent_node = Ager
draft_answer_node

# 2. Define the (
crawl_node >> agent_node                # After crawling, always go t
the agent to decide.
```

```
agent_node - "explore" >> crawl_node     # If the agent says "explore'
loop back to the crawler.
agent_node - "answer" >> draft_answer_node # If the agent says "answe
move to the writer.

# 3. Create the final flow, telling it where to start
support_bot_flow = Flow(start=crawl_node)
```

That's the entire orchestration logic. It's a simple, readable blueprint for our agen behavior.

The Shared Whiteboard (`shared` dictionary)

Next, our workers need a central place to read and write information. This is just a simple Python dictionary that holds the state of our operation.

```
shared = {
    "user_question": "How do I get a refund?",
    "urls_to_process": [0], # A "to-do" list of URL indices to crawl
    "visited_urls": set(),  # A set of URL indices it has already
crawled
    "url_content": {},      # Where it stores the text from each URL
    "all_discovered_urls": ["https://example.com"], # Master list of
every URL
    "final_answer": None
}
```

The Workers (`nodes.py`)

Now let's look at the simplified code for our three specialist nodes.

1. `CrawlAndExtr`

This `BatchNode` eff nd return its text and an

```
class CrawlAndExtract(BatchNode):
    def prep(self, shared):
        return shared.get("urls_to_process", [])

    def exec(self, url_index):
        url = shared["all_discovered_urls"][url_index]
        content, new_links = crawl_webpage(url)
        return {"url_index": url_index, "content": content, "new_links
new_links}

    def post(self, shared, prep_res, all_results):
        for result in all_results:
            idx = result["url_index"]
            shared["url_content"][idx] = result["content"]
            shared["visited_urls"].add(idx)

            for link_url in result["new_links"]:
                if link_url not in shared["all_discovered_urls"]:
                    shared["all_discovered_urls"].append(link_url)

        shared["urls_to_process"] = []
```

In English: It crawls each page on its to-do list, stores the content, and adds any n
unique links to the master URL list.

2. `AgentDecision`: The Brain

This node looks at what we've learned and decides what to do next, returning a sig
to the Flow.

```
class AgentDecision(Node):
    def prep(self, shared):
        knowledge = "\n".join(shared["url_content"].values())
        unvisite
enumerate(shared


        return {
            "ques
            "knowledge": knowledge,
            "unvisited_urls": unvisited_urls
        }

    def exec(self, prepared_data):
        prompt = f"""
        User Question: {prepared_data['question']}
        Knowledge I have: {prepared_data['knowledge']}
        URLs to explore next: {prepared_data['unvisited_urls']}

        Should I 'answer' or 'explore'? If exploring, which URLs are
best?
        Respond in YAML:
        decision: [answer/explore]
        selected_urls: [...]
        """
        response_yaml = call_llm(prompt)
        return parse_yaml(response_yaml)

    def post(self, shared, prep_res, decision):
        if decision["decision"] == "explore":
            selected_indices = [shared["all_discovered_urls"].index(u
                                for url in decision["selected_urls"]]
            shared["urls_to_process"] = selected_indices
            return "explore" # This signal matches our flow.py
instruction
        else:
            return "answer" # This signal also matches flow.py
```

In English: It asks the AI for a strategy (`answer` or `explore`) and returns that exa signal to the Flow, which knows what to do next.

3. `DraftAnswer`: The Writer

Once the Brain says `"answer"`, this node crafts the final response.

```
class DraftAnswe
    def prep(sel
        knowledg
        return {
knowledge }
```

```
    def exec(self, prepared_data):
        prompt = f"""
        Answer this question: {prepared_data['question']}
        Using ONLY this information:
        {prepared_data['knowledge']}
        """
        return call_llm(prompt)

    def post(self, shared, prep_res, final_answer_from_ai):
        shared["final_answer"] = final_answer_from_ai
```

In English: It gathers all the text we found, gives it to the AI, and asks it to write a beautiful, helpful response.

And that's the core of the system. Three simple nodes, each with a clear job, passi data through a simple dictionary.

Now that the magic is revealed (and you see it's not so magical after all), let's give our ch a pretty face so you can put it on your website.

---

5. Giving Our Bot a Face: From Terminal to Website

Okay, we have a functional AI brain that runs in the terminal. That's a great start, it's not very useful for your website visitors.

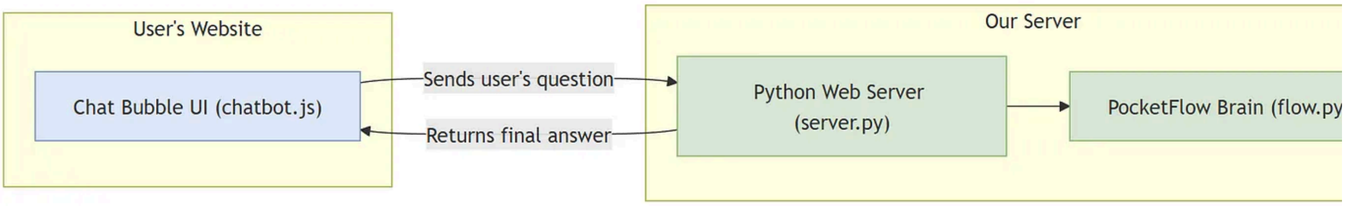
Let's connect that brain to a user-friendly chat bubble. This is a classic web development pattern with two simple parts: a **backend** (our Python script) and a **frontend** (the chat bu

The Architect

Think of it like this:

1. **The Backend (The Brain):** This is our Python script, `server.py`. Its only job wait for a question, run our PocketFlow logic to find the answer, and send the answer back. It's the powerhouse that does all the heavy lifting.
2. **The Frontend (The Face):** This is a small piece of JavaScript, `chatbot.js`, th you add to your website. It creates the chat icon and the chat window. When a user types a question, the JavaScript simply sends it to our backend for proces

They communicate over the network. The frontend asks a question, and the backe provides the answer.


User's Website
Chat Bubble UI (chatbot.js)
Sends user's question
Returns final answer
Our Server
Python Web Server (server.py)
PocketFlow Brain (flow.py

Let's look at the minimal code that makes each part work.

The Backend: `server.py`

We use a lightweight Python framework called **FastAPI** to create a simple web se Its job is to expose a single "endpoint" (like a URL) that the frontend can send questions to.

Here's the core logic in `server.py`:

```
from fastapi import FastAPI
from flow import support_bot_flow # Our PocketFlow brain

app = FastAPI()

@app.post("/get-a
def get_answer(da
    # 1. Get the
    question = da
    start_urls =

    # 2. Set up the shared dictionary for our flow
    shared_state = {"user_question": question, "all_discovered_urls":
start_urls, ...}

    # 3. Run the PocketFlow brain to find the answer
    support_bot_flow.run(shared_state)

    # 4. Return the final answer as a response
    return {"answer": shared_state.get("final_answer")}
```

In English: The server waits for a POST request at `/get-answer`. When it gets o
runs the same PocketFlow we built before and sends the result back.

The Frontend: `chatbot.js`

This is the JavaScript that lives on your website. It listens for the user to click "ser
then makes a simple web request to our Python backend.

Here's the simplified logic from `chatbot.js`:

```
async function handleSendClick() {
    const userInput = document.getElementById("chat-input").value;
    const siteUrls = ["https://your-site.com"]; // The URLs for the bo
to crawl

    // 1. Send the user's question to our Python backend
    const response = await fetch("/get-answer", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ question: userInput, urls: siteUrls })
    });

    // 2. Get the answer back from the server
    const data =
    const botAnsw

    // 3. Display
    displayMessa
}
```

In English: When the user sends a message, it packages up the question and sends to the `/get-answer` endpoint on our server. When the server responds, it display answer.

Running It Live

Now the process is clear:

1. **Start the Backend:** First, you need to run the brain. In your terminal, run `pyt server.py`. This starts the web server and gets it ready to answer questions.
2. **Add the Frontend to a Page:** Next, you add the `<script src="chatbot.j </script>` tag to your website's HTML. This makes the chat bubble appear.

To make testing easy, the project includes a sample `static/chatbot.html` file already has the script included. Once your server is running, just open that file in browser to see your live, interactive chatbot in action

6. Conclusion: Simple, Maintainable, and Live

Let's take a step back. We just built a fully-functional AI chatbot that can intellige answer questions about any website.

And we did it without touching a single vector database, writing a complex data-syncing script, or worrying about our information ever going stale. Its brain is you live website, which means its knowledge is always up-to-date.

This isn't just another chatbot. This is a better, simpler way to build one. Here's w this approach wins:

- **Always Up-to-D** yo website, you've i
- **Practically Zero** nl is to keep your w .

- **Incredibly Simple Code.** Because the entire system is built on PocketFlow an few straightforward Python scripts, the logic is easy to read and modify. Ther no black boxes to fight with.

The days of babysitting your AI are over. You now have the blueprint for a system that's not only intelligent but also practical and sustainable.

Ready to add a real-time brain to your own website?

---

*The complete, open-source code for this chatbot is waiting for you on GitHub. It's power the 100-line **PocketFlow** framework. Dive in, experiment, and see for yourself how eas building a truly smart chatbot can be! Get the AI Website Chatbot Code on GitHub*

Thanks for reading Pocket Flow! Subscribe for free to receive new posts and support my work.

---

12 Likes · 1 Restack

← Previous

Next

Discussion about this post

Comments Restacks

Write a comme

Looks like an article worth saving!

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later Hide Forever

Gopinathan K. Munappy Jul 20

♥ Liked by Zachary Huang

Is it possible to use LLM other than Gemini, specifically open source LLMs?

♡ LIKE (1) REPLY

3 replies by Zachary Huang and others

3 more comments...