Take the State of AI Developer Survey.

Get started

Menu

Getting Started > Next.js App Router

Next.js App Router Quickstart

In this quick start tutorial, you'll build a simple AI-chatbot with a streaming user interface. Along the way, you'll learn key concepts and techniques that are fundamental to using the SDK in your own projects.

Check out Prompt Engineering and HTTP Streaming if you haven't heard of them.

Prerequisites

To follow this quickstart, you'll need:

- Node.js 18+ and pnpm installed on your local development machine.
- An OpenAI API key.

If you haven't obtained your OpenAI API key, you can do so by signing up on the OpenAI website.

Create Your Application

Start by creating a new Next.js application. This command will create a new directory named `my-ai-app` and set up a basic Next.js application inside it.

Be sure to select yes when prompted to use the App Router. If you are looking for the Next.js Pages Router quickstart guide, you can find it here.

```
$ pnpm create next-app@latest my-ai-app
```

Navigate to the newly created directory:

```
$ cd my-ai-app
```

Install dependencies

Install `ai` and `@ai-sdk/openai`, the AI package and AI SDK's OpenAI provider respectively.

The AI SDK is designed to be a unified interface to interact with any large language model. This means that you can change model and providers with just one line of code! Learn more about available providers and building custom providers in the providers section.

pnpm npm yarn

```
$ pnpm add ai @ai-sdk/openai zod
```

Make sure you are using `ai` version 3.1 or higher.

Configure OpenAI API key

Create a `.env.local` file in your project root and add your OpenAI API Key. This key is used to authenticate your application with the OpenAI service.

```
$ touch .env.local
```

Edit the `.env.local` file:

.env.local

```
OPENAI_API_KEY=xxxxxxxxx
```

Replace `xxxxxxxxx` with your actual OpenAI API key.

The AI SDK's OpenAI Provider will default to using the `OPENAI_API_KEY` environment variable.

Create a Route Handler

Create a route handler, `app/api/chat/route.ts` and add the following code:

app/api/chat/route.ts

```
import { openai } from '@ai-sdk/openai';
import { streamText } from 'ai';

// Allow streaming responses up to 30 seconds
export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages,
  });

  return result.toDataStreamResponse();
}
```

Let's take a look at what is happening in this code:

1. Define an asynchronous `POST` request handler and extract `messages` from the body of the request. The `messages` variable contains a history of the conversation between you and the chatbot and provides the chatbot with the necessary context to make the next

2. Call `streamText`, which is imported from the `ai` package. This function accepts a configuration object that contains a `model` provider (imported from `@ai-sdk/openai`) and `messages` (defined in step 1). You can pass additional settings to further customise the model's behaviour.
3. The `streamText` function returns a `StreamTextResult`. This result object contains the `toDataStreamResponse` function which converts the result to a streamed response object.
4. Finally, return the result to the client to stream the response.

This Route Handler creates a POST request endpoint at `/api/chat`.

Wire up the UI

Now that you have a Route Handler that can query an LLM, it's time to setup your frontend. The AI SDK's UI package abstracts the complexity of a chat interface into one hook, `useChat`.

Update your root page (`app/page.tsx`) with the following code to show a list of chat messages and provide a user message input:

app/page.tsx

```tsx
'use client';

import { useChat } from 'ai/react';

export default function Chat() {
  const { messages, input, handleInputChange, handleSubmit } = useChat();
  return (
    <div className="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map(m => (
        <div key={m.id} className="whitespace-pre-wrap">
          {m.role === 'user' ? 'User: ' : 'AI: '}
          {m.content}
        </div>
      ))}

      <form onSubmit={handleSubmit}>
        <input
          className="fixed bottom-0 w-full max-w-md p-2 mb-8 border border-gray-300
          value={input}
          placeholder="Say something..."
          onChange={handleInputChange}
        />
      </form>
    </div>
  );
}
```

Make sure you add the `"use client"` directive to the top of your file. This allows you to add interactivity with Javascript.

This page utilizes the `useChat` hook, which will, by default, use the `POST` API route you created earlier (`/api/chat`). The hook provides functions and state for handling user input and form submission. The `useChat` hook provides multiple utility functions and state variables:

- `messages` - the current chat messages (an array of objects with `id`, `role`, and `content` properties).
- `input` - the current value of the user's input field.
- `handleInputChange` and `handleSubmit` - functions to handle user interactions (typing into the input field and submitting the form, respectively).

Running Your Application

With that, you have built everything you need for your chatbot! To start your application, use the command:

```
$ pnpm run dev
```

Head to your browser and open http://localhost:3000↗. You should see an input field. Test it out by entering a message and see the AI chatbot respond in real-time! The AI SDK makes it fast and easy to build AI chat interfaces with Next.js.

Enhance Your Chatbot with Tools

While large language models (LLMs) have incredible generation capabilities, they struggle with discrete tasks (e.g. mathematics) and interacting with the outside world (e.g. getting the weather). This is where tools come in.

Tools are actions that an LLM can invoke. The results of these actions can be reported back to the LLM to be considered in the next response.

For example, if a user asks about the current weather, without tools, the model would only be able to provide general information based on its training data. But with a weather tool, it can fetch and provide up-to-date, location-specific weather information.

Let's enhance your chatbot by adding a simple weather tool.

Update Your Route Handler

Modify your `app/api/chat/route.ts` file to include the new weather tool:

app/api/chat/route.ts

```ts
import { openai } from '@ai-sdk/openai';
import { streamText, tool } from 'ai';
import { z } from 'zod';

export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages,
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        parameters: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
    },
  });

  return result.toDataStreamResponse();
}
```

In this updated code:

1. You import the `tool` function from the `ai` package and `z` from `zod` for schema validation.

2. You define a `tools` object with a `weather` tool. This tool:

   - Has a description that helps the model understand when to use it.
   - Defines parameters using a Zod schema, specifying that it requires a `location` string to execute this tool. The model will attempt to extract this parameter from the context of the conversation. If it can't, it will ask the user for the missing information.
   - Defines an `execute` function that simulates getting weather data (in this case, it returns a random temperature). This is an asynchronous function running on the server so you can fetch real data from an external API.

   Now your chatbot can "fetch" weather information for any location the user asks about. When the model determines it needs to use the weather tool, it will generate a tool call with the necessary parameters. The `execute` function will then be automatically run, and you can access the results via `toolInvocations` that is available on the message object.

Try asking something like "What's the weather in New York?" and see how the model uses the new tool.

Notice the blank response in the UI? This is because instead of generating a text response, the model generated a tool call. You can access the tool call and subsequent tool result in the `toolInvocations` key of the message object.

Update the UI

To display the tool invocations in your UI, update your `app/page.tsx` file:

app/page.tsx

```tsx
'use client';

import { useChat } from 'ai/react';

```

```
export default function Chat() {
  const { messages, input, handleInputChange, handleSubmit } = useChat();
  return (
    <div className="flex flex-col w-full max-w-md py-24 mx-auto stretch">
      {messages.map(m => (
        <div key={m.id} className="whitespace-pre-wrap">
          {m.role === 'user' ? 'User: ' : 'AI: '}
          {m.toolInvocations ? (
            <pre>{JSON.stringify(m.toolInvocations, null, 2)}</pre>
          ) : (
            <p>{m.content}</p>
          )}
        </div>
      ))}

      <form onSubmit={handleSubmit}>
        <input
          className="fixed bottom-0 w-full max-w-md p-2 mb-8 border border-gray-300
          value={input}
          placeholder="Say something..."
          onChange={handleInputChange}
        />
      </form>
    </div>
  );
}
```

With this change, you check each message for any tool calls (`toolInvocations`). These tool calls will be displayed as stringified JSON. Otherwise, you show the message content as before.

Now, when you ask about the weather, you'll see the tool invocation and its result displayed in your chat interface.

Enabling Multi-Step Tool Calls

You may have noticed that while the tool results are visible in the chat interface, the model isn't using this information to answer your original query. This is because once the model generates a tool call, it has technically completed its generation.

To solve this, you can enable multi-step tool calls using the `maxSteps` option in your `useChat` hook. This feature will automatically send tool results back to the model to trigger an additional generation. In this case, you want the model to answer your question using the results from the weather tool.

Update Your Client-Side Code

Modify your `app/page.tsx` file to include the `maxSteps` option:

app/page.tsx

```
'use client';

import { useChat } from 'ai/react';

export default function Chat() {
  const { messages, input, handleInputChange, handleSubmit } = useChat({
    maxSteps: 5,
  });

  // ... rest of your component code
}
```

Head back to the browser and ask about the weather in a location. You should now see the model using the weather tool results to answer your question.

By setting `maxSteps` to 5, you're allowing the model to use up to 5 "steps" for any given generation. This enables more complex interactions and allows the model to gather and process information over several steps if needed. You can see this in action by adding another tool to convert the temperature from Fahrenheit to Celsius.

Update Your Route Handler

Update your `app/api/chat/route.ts` file to add a new tool to convert the temperature from Fahrenheit to Celsius:

app/api/chat/route.ts

```
import { openai } from '@ai-sdk/openai';
import { streamText, tool } from 'ai';
import { z } from 'zod';

export const maxDuration = 30;

export async function POST(req: Request) {
  const { messages } = await req.json();

  const result = streamText({
    model: openai('gpt-4o'),
    messages,
    tools: {
      weather: tool({
        description: 'Get the weather in a location (fahrenheit)',
        parameters: z.object({
          location: z.string().describe('The location to get the weather for'),
        }),
        execute: async ({ location }) => {
          const temperature = Math.round(Math.random() * (90 - 32) + 32);
          return {
            location,
            temperature,
          };
        },
      }),
      convertFahrenheitToCelsius: tool({
        description: 'Convert a temperature in fahrenheit to celsius',
        parameters: z.object({
          temperature: z
            .number()
            .describe('The temperature in fahrenheit to convert'),
        }),
        execute: async ({ temperature }) => {
          const celsius = Math.round((temperature - 32) * (5 / 9));
          return {
            celsius,
          };
        },
      }),
    },
  });

  return result.toDataStreamResponse();
}
```

Now, when you ask "What's the weather in New York in celsius?", you should see a more complete interaction:

1. The model will call the weather tool for New York.
2. You'll see the tool result displayed.
3. It will then call the temperature conversion tool to convert the temperature from Fahrenheit to Celsius.
4. The model will then use that information to provide a natural language response about the weather in New York.

This multi-step approach allows the model to gather information and use it to provide more accurate and contextual responses, making your chatbot considerably more useful.

This simple example demonstrates how tools can expand your model's capabilities. You can create more complex tools to integrate with real APIs, databases, or any other external systems, allowing the model to access and process real-world data in real-time. Tools bridge the gap between the model's knowledge cutoff and current information.

Where to Next?

You've built an AI chatbot using the AI SDK! From here, you have several paths to explore:

- To learn more about the AI SDK, read through the documentation.
- If you're interested in diving deeper with guides, check out the RAG (retrieval-augmented generation) and multi-modal chatbot guides.
- To jumpstart your first AI project, explore available templates↗.

Previous Next

‹ Navigating the Library

Next.js Pages Router ›