"use client"

import axios from "axios"

import \* as z from "zod"

import { Loader } from "@/components/loader"

import { Heading } from "@/components/heading"

import { MapPin } from "lucide-react"

import { useForm } from "react-hook-form"

import { formSchema } from "./constants"

import { zodResolver } from "@hookform/resolvers/zod"

import { Form, FormControl, FormField, FormItem } from "@/components/ui/form"

import { Input } from "@/components/ui/input"

import { Button } from "@/components/ui/button"

import { useRouter } from "next/navigation"

import { useState, useEffect } from "react"

import { ChatCompletionMessageParam } from "openai/resources/chat/completions"

import { Empty } from "@/components/empty"

import { cn } from "@/lib/utils"

import { UserAvatar } from "@/components/user-avatar"

import { BotAvatar } from "@/components/bot-avatar"

import questionWords from "./questionWords"

const GOOGLE\_MAPS\_API\_KEY = process.env.NEXT\_PUBLIC\_GOOGLE\_MAPS\_API\_KEY;

const CodePage = () => {

  const locationTerm = "at my current location"

  const level1Keys: string[] = [];

  const level2Keys: string[] = [];

  const level3Keys: string[] = [];

  const level4Keys: string[] = [];

  const level5Keys: string[] = [];

  const allPaths: string[] = [];

  Object.keys(questionWords).forEach((level1Key: string) => {

    level1Keys.push(level1Key);

    const level2Object = questionWords[level1Key];

    Object.keys(level2Object).forEach((level2Key: string) => {

      level2Keys.push(level2Key);

      const level3Object = level2Object[level2Key];

      Object.keys(level3Object).forEach((level3Key: string) => {

        level3Keys.push(level3Key);

        const level4Object = level3Object[level3Key];

        Object.keys(level4Object).forEach((level4Key: string) => {

          level4Keys.push(level4Key);

          const level5Array: string[] = level4Object[level4Key];

          level5Array.forEach((level5Key: string) => {

            level5Keys.push(level5Key);

            allPaths.push(

              `${level1Key} > ${level2Key} > ${level3Key} > ${level4Key} > ${level5Key}`

            );

          });

        });

      });

    });

  });

  // Remove duplicates from each level

  const uniqueLevel1Keys = Array.from(new Set(level1Keys));

  const uniqueLevel2Keys = Array.from(new Set(level2Keys));

  const uniqueLevel3Keys = Array.from(new Set(level3Keys));

  const uniqueLevel4Keys = Array.from(new Set(level4Keys));

  const uniqueLevel5Keys = Array.from(new Set(level5Keys));

  const [selectedOptions, setSelectedOptions] = useState<string[]>([]);

  const [newOptions, setNewOptions] = useState<string[][]>([]);

  const [userInput, setUserInput] = useState<string>("");

  const [geocodedLocation, setGeocodedLocation] = useState<string>("");

  const [showDropdown, setShowDropdown] = useState<boolean>(false);

  const [dropdownValue, setDropdownValue] = useState<string>("");

  const [addresses, setAddresses] = useState<string[]>([]);

  const [selectedAddress, setSelectedAddress] = useState<string>("");

  const handleDropdownChange = (

    event: React.ChangeEvent<HTMLSelectElement>,

    level: number

  ) => {

    const newSelectedOption = event.target.value;

    // Limit the selectedOptions array to a maximum of 4 unique elements

    const updatedSelectedOptions = Array.from(new Set([...selectedOptions, newSelectedOption])).slice(0, 4);

    setSelectedOptions(updatedSelectedOptions);

    // Find all elements where the selected option is part of the path

    const filteredPaths = allPaths.filter((path) =>

      //updatedSelectedOptions.every((option) => path.includes(option))

      updatedSelectedOptions.every((option) => path.split(" > ").includes(option))

    );

    const options = filteredPaths.map((path) => path.split(" > "));

    // Transpose options array to align with dropdown structure

    const transposedOptions: string[][] = [];

    options.forEach((option, index) => {

      option.forEach((value, i) => {

        transposedOptions[i] = transposedOptions[i] || [];

        transposedOptions[i][index] = value;

      });

    });

    const uniqueTransposedOptions = transposedOptions.map((array) =>

      Array.from(new Set(array))

    );

    setNewOptions(uniqueTransposedOptions);

  };

  const inputValue = newOptions.reduce((acc, currOptions) => {

    if (currOptions.length === 1) {

      acc.push(currOptions[0]);

    }

    return acc;

  }, []).join(" ");

  //const textareaRef = useRef<HTMLTextAreaElement>(null)

  const resetOptions = () => {

    setSelectedOptions([]);

    setNewOptions([]);

    setUserInput('');

    form.reset();

    setShowDropdown(false);

    setDropdownValue("");

    setAddresses([]);

    setSelectedAddress("");

  };

  const textContent = `${inputValue ? `${inputValue} ` : ''}${userInput}`;

  const handleUserInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {

    const newUserInput = event.target.value.trim();

    const exceedingSubstring = newUserInput.substring(inputValue.length).trim();

    if (!inputValue) {

      setUserInput('');

    } else {

      setUserInput(exceedingSubstring);

    }

  };

// Function to handle address change from dropdown

const handleAddressChange = (

  event: React.ChangeEvent<HTMLSelectElement>

) => {

  setSelectedAddress(event.target.value);

};

  const fetchGeolocation = async () => {

    if ("geolocation" in navigator) {

      navigator.geolocation.getCurrentPosition(async (position) => {

        const { latitude, longitude } = position.coords;

        const geocodeUrl = `https://maps.googleapis.com/maps/api/geocode/json?latlng=${latitude},${longitude}&key=${GOOGLE\_MAPS\_API\_KEY}`;

        try {

          const response = await axios.get(geocodeUrl);

          console.log("Result", response.data.results)

          const addressList = response.data.results.map((result: any) => result.formatted\_address);

          setAddresses(addressList);

          const lengthHalfOrOne = Math.max(Math.floor(response.data.results.length / 2), 1);

          const location = `in ${response.data.results[response.data.results.length - lengthHalfOrOne]?.formatted\_address || "your current location"}`;

          setGeocodedLocation(location);

        } catch (error) {

          console.error("Error fetching geolocation:", error);

          setGeocodedLocation("your current location");

        }

      });

    } else {

      console.error("Geolocation not supported.");

      setGeocodedLocation("your current location");

    }

  };

  const router = useRouter()

  const [messages, setMessages] = useState<ChatCompletionMessageParam[]>([])

  const form = useForm<z.infer<typeof formSchema>>({

    resolver: zodResolver(formSchema),

    defaultValues: {

      prompt: ""

    }

  })

  const isLoading = form.formState.isSubmitting

  useEffect(() => {

    form.setValue('prompt', textContent);

  }, [form, textContent]);

  useEffect(() => {

    if (textContent.includes(locationTerm)) {

      fetchGeolocation();

      setShowDropdown(true);

    } else {

      setShowDropdown(false);

    }

  }, [textContent]);

  const onSubmit = async (values: z.infer<typeof formSchema>) => {

    try {

      const includeLocation = values.prompt.includes(locationTerm);

      let finalPrompt = values.prompt;

      if (includeLocation) {

        if (selectedAddress) {

          finalPrompt = finalPrompt.replace(locationTerm, selectedAddress);

        } else {

          finalPrompt = finalPrompt.replace(locationTerm, geocodedLocation);

        }

      }

      const userMessage: ChatCompletionMessageParam = {

        role: "user",

        content: finalPrompt

      }

      const updatedMessages = [...messages, userMessage];

      setMessages(updatedMessages);

      const response = await axios.post("/api/code", {

        messages: updatedMessages,

        geocodedLocation,

        includeLocation: textContent.includes(locationTerm),

      });

      setMessages((current) => [

        ...current,

        {

          role: "assistant",

          content: response.data.content,

        },

      ]);

      form.reset();

      resetOptions();

      router.refresh();

    } catch (error: any) {

      console.error(error);

    } finally {

      router.refresh();

    }

  }

  return (

    <div>

      <Heading

        title="Tour Guide"

        description="Your smart Travel Butler"

        icon={MapPin}

        iconColor="text-green-700"

        bgColor="bg-green-700/10"

      />

      <div className="px-4 lg:px-8">

        <div>

          <Form {...form}>

            <div className="flex flex-col lg:flex-row mb-4 space-y-4 lg:space-y-0 lg:space-x-4 max-w-full">

              {[uniqueLevel1Keys, uniqueLevel2Keys, uniqueLevel3Keys, uniqueLevel4Keys, uniqueLevel5Keys].map(

                (keys, index) => (

                  <div key={index}>

                    <select

                      value={

                        newOptions[index]?.length === 1

                          ? newOptions[index][0] // Set default value if there's only one option

                          : selectedOptions[index] || "" // Otherwise, use selectedOptions

                      }

                      onChange={(event) => handleDropdownChange(event, index)}

                      className={`rounded-xl ${newOptions[index]?.length === 1 ? 'bg-white' : 'bg-green-600 text-white'} text-center h-[3rem]`}

                    >

                      <option value="" disabled hidden>

                        {index === 0 ? "Question" : index === 1 ? "Perspective" : index === 2 ? "Attribute/Activity" : index === 3 ? "Object/Item" : "Location"}

                      </option>

                      {(newOptions[index] || keys).sort().map((key: string, keyIndex: number) => ( // Sort the keys alphabetically

                        <option key={keyIndex} value={key}>

                          {key}

                        </option>

                      ))}

                    </select>

                  </div>

                )

              )}

            </div>

            {showDropdown && (

              <div className="mb-4">

                <select

                  value={selectedAddress || dropdownValue}

                  onChange={(event) => handleAddressChange(event)}

                  className="rounded-xl bg-white border border-green-600 text-green-600 text-center h-[3rem]"

                >

                  <option value="">Optionally, choose precision...</option>

                  {addresses.map((address, index) => (

                    <option key={index} value={address}>

                      {address}

                    </option>

                  ))}

                </select>

              </div>

            )}

            <form

              onSubmit={form.handleSubmit(onSubmit)}

              className="

                rounded-lg

                border

                w-full

                p-4

                px-3

                md:px-6

                focus-within:shadow-sm

                grid

                grid-cols-12

                gap-2

              "

            >

              <FormField

                name="prompt"

                render={({ field }) => (

                  <FormItem className="col-span-12 lg:col-span-10">

                    <FormControl className="m-0 p-0">

                      <Input

                        className="border-0 outline-none focus-visible:ring-0 focus-visible:ring-transparent"

                        disabled={isLoading}

                        value={textContent}

                        placeholder="Create your question with the buttons above. Choose any order."

                        onChange={handleUserInputChange}

                        //{...field}

                      />

                    </FormControl>

                  </FormItem>

                )}

              />

              <Button className="col-span-12 lg:col-span-2 w-full" disabled={isLoading}>

                Get the answer

              </Button>

            </form>

          </Form>

        </div>

        <div className="space-y-4 mt-4">

          {isLoading && (

            <div className="p-8 rounded-lg w-full flex items-center justify-center bg-muted">

              <Loader />

            </div>

          )}

          {messages.length === 0 && !isLoading && (

            <Empty label="No conversation started" />

          )}

          <div className='flex flex-col-reverse gap-y-4'>

            {messages.map((message, index) => (

              <div

                key={index}

                className={cn(

                  'p-8 w-full flex items-start gap-x-8 rounded-lg',

                  message.role === 'user'

                    ? 'bg-white border border-black/10'

                    : 'bg-muted'

                )}

              >

                {message.role === 'user' ? <UserAvatar /> : <BotAvatar />}

                <div className='text-sm whitespace-pre-line'>

                  {message.content?.toString()}

                </div>

              </div>

            ))}

          </div>

        </div>

      </div>

    </div>

  )

}

export default CodePage

import { auth } from "@clerk/nextjs/server"

import { NextResponse } from 'next/server';

import OpenAI from 'openai';

import { ChatCompletionMessageParam } from "openai/resources/chat/completions"

import { increaseApiLimit, checkApiLimit } from '@/lib/api-limit';

const openai = new OpenAI({

  apiKey: process.env.OPENAI\_API\_KEY // This is also the default, can be omitted

});

const instructionMessage: ChatCompletionMessageParam = {

    role: "system",

    content: "You are a tourist guide. You know everything about all locations"

}

export async function POST(

    req: Request

) {

    try {

        const { userId } : { userId: string | null } = auth();

        const body = await req.json()

        const { messages, selectedAddress, includeLocation } = body;

        if (!userId) {

            return new NextResponse("Unauthorized", {status: 401})

        }

        if (!openai.apiKey) {

            return new NextResponse("OpenAI API key not configured", {status: 500})

        }

        if(!messages) {

            return new NextResponse("Messages are required", {status: 400})

        }

        const freeTrial = await checkApiLimit();

        if (!freeTrial ) {

            return new NextResponse('Free trial has expired', { status: 403 });

        }

        const response = await openai.chat.completions.create({

            model: "gpt-3.5-turbo",

            messages: [instructionMessage, ...messages]

        })

        let responseContent = response.choices[0].message.content;

        if (includeLocation && selectedAddress) {

            responseContent = `Your selected location is ${selectedAddress}. \n\n${responseContent}`;

        }

        //await increaseApiLimit();

        return NextResponse.json({ content: responseContent });

    } catch (error) {

        console.log("[CODE\_ERROR]", error)

        return new NextResponse("Internal error", {status:500})

    }

}

"use client"

import axios from "axios"

import \* as z from "zod"

import {Loader} from "@/components/loader"

import { Heading } from "@/components/heading"

import { Code } from "lucide-react"

import { useForm } from "react-hook-form"

import { formSchema } from "./constants"

import { zodResolver } from "@hookform/resolvers/zod"

import { Form, FormControl, FormField, FormItem } from "@/components/ui/form"

import { Input } from "@/components/ui/input"

import { Button } from "@/components/ui/button"

import { useRouter } from "next/navigation"

import { useState } from "react"

import { ChatCompletionMessageParam } from "openai/resources/chat/completions"

import { Empty } from "@/components/empty"

import { cn } from "@/lib/utils"

import { UserAvatar } from "@/components/user-avatar"

import { BotAvatar } from "@/components/bot-avatar"

const CodePage = () => {

  const router = useRouter()

  const [messages, setMessages] = useState<ChatCompletionMessageParam[]>([])

  const form = useForm<z.infer<typeof formSchema>>({

    resolver: zodResolver(formSchema),

    defaultValues: {

      prompt: ""

    }

  })

  const isLoading = form.formState.isSubmitting

  const onSubmit = async (values: z.infer<typeof formSchema>) => {

    try {

      const userMessage: ChatCompletionMessageParam = {

        role: "user",

        content: values.prompt

      }

      const newMessages = [...messages, userMessage]

      const response = await axios.post("/api/code", {

        messages: newMessages

      })

      setMessages((current) => [...current, userMessage, response.data])

      form.reset()

    } catch (error: any) {

      console.log(error)

    } finally {

      router.refresh()

    }

  }

  return (

    <div>

      <Heading

        title="Code Generation"

        description="Your smart Personal Asisstant"

        icon={Code}

        iconColor="text-green-700"

        bgColor="bg-green-700/10"

      />

      <div className="px-4 lg:px-8">

        <div>

          <Form {...form}>

            <form

              onSubmit={form.handleSubmit(onSubmit)}

              className="

                rounded-lg

                border

                w-full

                p-4

                px-3

                md:px-6

                focus-within:shadow-sm

                grid

                grid-cols-12

                gap-2

              "

            >

              <FormField

                name="prompt"

                render={({field}) => (

                  <FormItem className="col-span-12 lg:col-span-10">

                    <FormControl className="m-0 p-0">

                      <Input

                        className="border-0 outline-none focus-visible:ring-0 focus-visible:ring-transparent"

                        disabled={isLoading}

                        placeholder="Enter your question here"

                        {...field}

                      />

                    </FormControl>

                  </FormItem>

                )}

              />

              <Button className="col-span-12 lg:col-span-2 w-full" disabled={isLoading}>

                Get the answer

              </Button>

            </form>

          </Form>

        </div>

        <div className="space-y-4 mt-4">

          {isLoading && (

            <div className="p-8 rounded-lg w-full flex items-center justify-center bg-muted">

              <Loader />

            </div>

          )}

          {messages.length === 0 && !isLoading && (

            <Empty label="No conversation started" />

          )}

            <div className='flex flex-col-reverse gap-y-4'>

            {messages.map((message, index) => (

              <div

                key={index}

                className={cn(

                  'p-8 w-full flex items-start gap-x-8 rounded-lg',

                  message.role === 'user'

                    ? 'bg-white border border-black/10'

                    : 'bg-muted'

                )}

              >

                {message.role === 'user' ? <UserAvatar /> : <BotAvatar />}

                <p className='text-sm'>

                  {message.content?.toString()}

                </p>

              </div>

            ))}

          </div>

        </div>

      </div>

    </div>

  )

}

export default CodePage

import {

    clerkMiddleware,

    createRouteMatcher

  } from '@clerk/nextjs/server';

const isProtectedRoute = createRouteMatcher([

    '/dashboard(.\*)',

]);

export default clerkMiddleware((auth, req) => {

    if (isProtectedRoute(req)) auth().protect();

  });

export const config = {

  matcher: [

    "/((?!.\*\\..\*|\_next).\*)", // Don't run middleware on static files

    "/", // Run middleware on index page

    "/(api|trpc)(.\*)"], // Run middleware on API routes

};

import {

  clerkMiddleware,

  createRouteMatcher

} from '@clerk/nextjs/server';

const isPublicRoute = createRouteMatcher(['/', '/api/webhook']);

const isProtectedRoute = createRouteMatcher([

  '/dashboard(.\*)',

]);

export default clerkMiddleware((auth, req) => {

  if (isProtectedRoute(req)) auth().protect();

  if(!isPublicRoute(req)){

    auth().protect();

  }

});

export const config = {

matcher: [

  "/((?!.\*\\..\*|\_next).\*)", // Don't run middleware on static files

  "/", // Run middleware on index page

  "/(api|trpc)(.\*)"], // Run middleware on API routes

};