**Gemini LLM API Entegrasyonu ile Akıllı Chatbot Geliştirme Kılavuzu**

Bu kılavuz, mevcut bir Node.js/Express/TypeScript backend'i ve Next.js frontend'i olan bir uygulamaya, Google Gemini API'sinin "Function Calling" (Araç Kullanımı) özelliğinden faydalanarak akıllı bir chatbot ekleme sürecini detaylandıracaktır.

**Mimari Yaklaşım: Function Calling (Araç Kullanımı)**

Hedeflediğiniz senaryo için en doğru ve güçlü yaklaşım, Gemini'nin "Function Calling" özelliğini kullanmaktır. Bu özellik, LLM'in sadece metin üretmekle kalmayıp, sizin tarafınızdan tanımlanan fonksiyonları (bu durumda mevcut API servislerinizi) belirli parametrelerle çağırması gerektiğini anlamasını sağlar.

**İş Akışı Şu Şekilde Olacaktır:**

1. **Kullanıcı (Frontend):** Chatbot'a bir komut yazar. Örn: "İstanbul'daki satılık 2+1 daireleri listele."
2. **Frontend (Next.js):** Bu mesajı, kendi backend'inizde oluşturacağınız yeni bir /api/chat endpoint'ine gönderir.
3. Backend (Node.js):

a. Kullanıcının mesajını alır.

b. Bu mesajı, önceden tanımlanmış fonksiyonlarla (tools) birlikte Gemini API'ye gönderir. Bu fonksiyonlar, sizin mevcut GET /apartments, POST /book-viewing gibi API endpoint'lerinizi Gemini'ye tanıtan bir şema içerir.

1. Gemini API:

a. Kullanıcının niyetini anlar ("daire listelemek istiyor").

b. Bu niyetin, sizin tanımladığınız getApartments fonksiyonu ile eşleştiğini fark eder.

c. Gerekli parametreleri (city: 'İstanbul', roomCount: '2+1') metinden çıkarır.

d. Cevap olarak metin yerine, "Şu fonksiyonu şu parametrelerle çağır: getApartments({ city: 'İstanbul', roomCount: '2+1' })" diyen bir functionCall nesnesi döndürür.

1. Backend (Node.js):

a. Gemini'den gelen functionCall nesnesini yakalar.

b. Kendi içindeki ilgili API servisini (/api/apartments?city=İstanbul&rooms=2+1) çağırır.

c. Bu servisten dönen JSON verisini alır.

1. Backend'den Gemini'ye (2. Tur):

a. Backend, az önce çalıştırdığı fonksiyondan dönen sonucu (örn: dairelerin listesi) alıp tekrar Gemini API'ye gönderir. Bu istekte, "Senden istediğin fonksiyonu çalıştırdım ve sonuç bu. Şimdi bu sonucu kullanarak kullanıcıya anlamlı bir cevap oluştur." denir.

1. Gemini API:

a. Gelen JSON verisini anlar.

b. "İstanbul'da aradığınız kriterlere uygun 5 adet daire buldum. İşte öne çıkanlar: ... Detaylarını görmek ister misiniz?" gibi kullanıcı dostu, doğal bir metin cevabı oluşturur.

1. **Backend'den Frontend'e:** Bu nihai metin cevabı frontend'e gönderilir ve kullanıcıya gösterilir.

Bu döngü, kullanıcının sonuçlar hakkında "En ucuz olan hangisi?" gibi takip soruları sormasına da olanak tanır, çünkü sohbet geçmişi korunur.

**Adım 1: Backend Geliştirmesi (Node.js / Express / TypeScript)**

Backend, bu operasyonun beyni olacaktır.

**A. Gerekli Paketlerin Kurulumu**

Projenizin package.json dosyasına Google'ın Generative AI SDK'sını ekleyin.

Bash

npm install @google/generative-ai

Ayrıca, ortam değişkenleri için dotenv kullanmıyorsanız kurmanız tavsiye edilir.

Bash

npm install dotenv

**B. API Anahtarının Güvenli Bir Şekilde Saklanması**

Projenizin kök dizininde bir .env dosyası oluşturun ve Google AI Studio'dan aldığınız API anahtarınızı buraya ekleyin.

.env

GEMINI\_API\_KEY="YOUR\_API\_KEY\_HERE"

**C. Gemini Servisini ve Araçları (Tools) Tanımlama**

Sisteminizdeki mevcut servisleri Gemini'ye tanıtacak bir yapı kuralım. Örnek olarak, daire listeleme ve randevu alma fonksiyonlarımız olsun.

src/services/gemini.service.ts

TypeScript

import { GoogleGenerativeAI, HarmCategory, HarmBlockThreshold, FunctionDeclarationSchemaType } from "@google/generative-ai";

import 'dotenv/config';

// API Anahtarını .env dosyasından al

const API\_KEY = process.env.GEMINI\_API\_KEY;

if (!API\_KEY) {

throw new Error("GEMINI\_API\_KEY is not defined in .env file");

}

const genAI = new GoogleGenerativeAI(API\_KEY);

// Mevcut API servislerinizi Gemini'ye "tool" olarak tanıtın

const tools = [

{

functionDeclarations: [

{

name: "listApartments",

description: "Belirtilen şehirdeki ve oda sayısındaki satılık daireleri listeler.",

parameters: {

type: FunctionDeclarationSchemaType.OBJECT,

properties: {

city: { type: FunctionDeclarationSchemaType.STRING,   
 description: "Dairelerin aranacağı şehir, örn: İstanbul, Ankara" },

roomCount: { type: FunctionDeclarationSchemaType.STRING,   
 description: "Oda sayısı, örn: 2+1, 3+1" },

},

required: ["city"],

},

},

{

name: "bookApartmentViewing",

description: "Belirli bir daire ID'si için ve belirli bir tarihte randevu oluşturur.",

parameters: {

type: FunctionDeclarationSchemaType.OBJECT,

properties: {

apartmentId: { type: FunctionDeclarationSchemaType.NUMBER,   
 description: "Randevu alınacak dairenin ID'si" },

date: { type: FunctionDeclarationSchemaType.STRING,   
 description: "Randevu tarihi, YYYY-MM-DD formatında" },

},

required: ["apartmentId", "date"],

},

},

],

},

];

const model = genAI.getGenerativeModel({

model: "gemini-1.5-flash", // Veya tercih ettiğiniz başka bir model

tools: tools,

});

export const chatSession = model.startChat({

history: [], // Sohbet geçmişini burada tutabilirsiniz

});

// Bu kısım, Gemini'nin çağırmasını istediği fonksiyonları gerçekten çalıştıracak olan yerdir.

// Pratikte bu fonksiyonlar sizin mevcut API controller'larınızı çağıracaktır.

export const availableFunctions: { [key: string]: Function } = {

listApartments: async ({ city, roomCount }: { city: string; roomCount?: string }) => {

console.log(`Fonksiyon çağrıldı: listApartments. Şehir: ${city}, Oda Sayısı: ${roomCount}`);

// TODO: Buraya kendi `GET /api/apartments?city=...` servisinizi çağıran kodu yazın.

// Örnek statik veri:

return {

apartments: [

{ id: 101, address: "Örnek Mah. No: 5", price: "5.000.000 TL" },

{ id: 102, address: "Deneme Cad. No: 12", price: "6.500.000 TL" },

],

};

},

bookApartmentViewing: async ({ apartmentId, date }: { apartmentId: number; date: string }) => {

console.log(`Fonksiyon çağrıldı: bookApartmentViewing. Daire ID: ${apartmentId}, Tarih: ${date}`);

// TODO: Buraya kendi `POST /api/bookings` servisinizi çağıran kodu yazın.

// Örnek statik veri:

return {

success: true,

bookingId: Math.floor(Math.random() \* 1000),

message: `Daire #${apartmentId} için ${date} tarihine randevunuz oluşturulmuştur.`

};

}

};

**D. Yeni Chat Endpoint'ini Oluşturma**

src/controllers/chat.controller.ts

TypeScript

import { Request, Response } from 'express';

import { chatSession, availableFunctions } from '../services/gemini.service';

export const handleChat = async (req: Request, res: Response) => {

try {

const { message } = req.body;

if (!message) {

return res.status(400).json({ error: 'Message is required' });

}

// 1. Kullanıcı mesajını Gemini'ye gönder

const result = await chatSession.sendMessage(message);

const response = result.response;

const functionCalls = response.functionCalls();

if (functionCalls && functionCalls.length > 0) {

// 2. Gemini bir fonksiyon çağırmak istedi

const call = functionCalls[0]; // Şimdilik ilk çağrıyı ele alalım

const functionToCall = availableFunctions[call.name];

if (functionToCall) {

// 3. İlgili fonksiyonu argümanlarla çalıştır

const functionResult = await functionToCall(call.args);

// 4. Fonksiyonun sonucunu tekrar Gemini'ye göndererek nihai cevabı al

const secondResult = await chatSession.sendMessage([

{

functionResponse: {

name: call.name,

response: functionResult,

},

},

]);

// 5. Nihai cevabı kullanıcıya gönder

const finalResponse = secondResult.response.text();

return res.json({ reply: finalResponse });

} else {

return res.status(500).json({ error: `Function ${call.name} not found.` });

}

} else {

// Gemini doğrudan bir metin cevabı verdiyse

const textResponse = response.text();

return res.json({ reply: textResponse });

}

} catch (error) {

console.error('Chat Error:', error);

res.status(500).json({ error: 'An error occurred during the chat.' });

}

};

Ve bu controller'ı Express router'ınıza ekleyin.

src/routes/chat.routes.ts

TypeScript

import { Router } from 'express';

import { handleChat } from '../controllers/chat.controller';

const router = Router();

router.post('/chat', handleChat);

export default router;

**Adım 2: Frontend Entegrasyonu (Next.js)**

Şimdi frontend'de kullanıcı arayüzünü oluşturalım ve backend'imizdeki yeni endpoint ile konuşturalım.

**A. Chat Arayüzü Bileşeni Oluşturma**

components/Chatbot.tsx adında bir bileşen oluşturun.

TypeScript

'use client';

import { useState, FormEvent } from 'react';

interface IMessage {

sender: 'user' | 'bot';

text: string;

}

export default function Chatbot() {

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

const [input, setInput] = useState('');

const [isLoading, setIsLoading] = useState(false);

const handleSubmit = async (e: FormEvent) => {

e.preventDefault();

if (!input.trim()) return;

const userMessage: IMessage = { sender: 'user', text: input };

setMessages(prev => [...prev, userMessage]);

setInput('');

setIsLoading(true);

try {

// Kendi backend'imize istek atıyoruz, ASLA doğrudan Gemini'ye değil!

const response = await fetch('/api/chat', { // Next.js API Route proxy kullanabilir (veya Express URL'i)

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ message: input }),

});

if (!response.ok) {

throw new Error('Network response was not ok');

}

const data = await response.json();

const botMessage: IMessage = { sender: 'bot', text: data.reply };

setMessages(prev => [...prev, botMessage]);

} catch (error) {

console.error("Failed to get bot reply:", error);

const errorMessage: IMessage = { sender: 'bot', text: "Üzgünüm, bir hata oluştu." };

setMessages(prev => [...prev, errorMessage]);

} finally {

setIsLoading(false);

}

};

return (

<div className="chatbot-container">

<div className="messages-area">

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

<div key={index} className={`message ${msg.sender}`}>

{msg.text}

</div>

))}

{isLoading && <div className="message bot">...</div>}

</div>

<form onSubmit={handleSubmit} className="chat-form">

<input

type="text"

value={input}

onChange={(e) => setInput(e.target.value)}

placeholder="Bir mesaj yazın..."

disabled={isLoading}

/>

<button type="submit" disabled={isLoading}>Gönder</button>

</form>

</div>

);

}

*Not: Bu bileşeni projenizin uygun bir sayfasına (app/page.tsx veya başka bir rota) eklemeyi unutmayın.*

**B. API İsteği için Yapılandırma**

Eğer Next.js uygulamanız ve Express backend'iniz farklı portlarda çalışıyorsa, Next.js'in next.config.js dosyası üzerinden bir proxy rewrite ayarlaması yapmak en temiz yöntemdir. Bu, CORS hatalarını önler.

next.config.mjs

JavaScript

/\*\* @type {import('next').NextConfig} \*/

const nextConfig = {

async rewrites() {

return [

{

source: '/api/:path\*',

destination: 'http://localhost:8000/api/:path\*', // Express sunucunuzun adresi

},

];

},

};

export default nextConfig;

Bu ayar sayesinde, frontend'deki fetch('/api/chat') isteği Next.js tarafından yakalanıp http://localhost:8000/api/chat adresine yönlendirilecektir.

**Adım 3: Çalıştırma ve Test**

1. **Backend'i Başlatın:** npm run dev (veya start script'iniz ne ise) komutu ile Express sunucunuzu başlatın.
2. **Frontend'i Başlatın:** npm run dev komutu ile Next.js sunucunuzu başlatın.
3. **Test Edin:**
   * Uygulamanızı açın ve chatbot arayüzüne gidin.
   * "Ankara'daki 3+1 daireleri listeler misin?" yazıp gönderin.
   * Backend konsolunda "Fonksiyon çağrıldı: listApartments..." logunu görmelisiniz.
   * Frontend'de botun, "Ankara'da aradığınız kriterlere uygun 2 adet daire buldum..." gibi bir cevap verdiğini görmelisiniz.
   * Takip sorusu sorun: "101 ID'li daire için yarın bir randevu oluştur."
   * Backend konsolunda "Fonksiyon çağrıldı: bookApartmentViewing..." logunu ve frontend'de randevu onayı mesajını görmelisiniz.

**İleri Seviye Konular ve İpuçları**

* **Sohbet Geçmişi (Context):** gemini.service.ts dosyasındaki chatSession'ı her kullanıcı için ayrı yönetmeniz gerekir. Örneğin, kullanıcı ID'sine göre Map<string, ChatSession> tutabilirsiniz. Her yeni istekte, kullanıcının geçmiş mesajlarını da startChat metodunun history parametresine ekleyerek daha bağlamsal konuşmalar sağlayabilirsiniz.
* **Streaming Cevaplar:** Daha iyi bir kullanıcı deneyimi için cevabın tamamını beklemek yerine kelime kelime akmasını sağlayabilirsiniz. Bunun için chatSession.sendMessageStream metodunu kullanıp backend'den frontend'e "Server-Sent Events (SSE)" ile veri basabilirsiniz.
* **Hata Yönetimi:** availableFunctions içindeki fonksiyonlarda oluşabilecek hataları (örn: veritabanı hatası) yakalayıp, bu hata bilgisini Gemini'ye geri göndererek "Aradığınız daire bulunamadı, başka bir kriter denemek ister misiniz?" gibi daha akıllı hata mesajları ürettirebilirsiniz.
* **Güvenlik ve Doğrulama:** handleChat controller'ında, fonksiyon çağırmadan önce kullanıcının o işlemi yapma yetkisi olup olmadığını (authentication/authorization) kontrol etmelisiniz.