Ai Powered App Development

Leveraging Angular, Python, and AI

GenAI

Practical Solutions

First Edition

Muhammad Awais
Sonu Kapoor
Lars Gyrup Brink Nielsen

Ai Powered App Development

Leveraging Angular, Python, and Generative Ai

Author: Muhammad Awais

Co-Author: Sonu Kapoor

Co-Author: Lars Gyrup Brink Nielsen

First published: November 2024

This is Angular

Free, open and honest Angular education.

https://this-is-angular.github.io/angular-guides/

This amazing tech write is dedicated to my father, who taught me hard work and perseverance, my mother, whose love and encouragement have been my strength, and my wife, whose support was invaluable in organizing this book. To my daughter, the light of our lives, may you be blessed with love and wisdom. To my late grandmother, your spirit and kindness continue to inspire me. And to my siblings, thank you for your unwavering support—here's to many more celebrations together.

Muhammad Awais (Pakistan)

This book is dedicated with all my love and gratitude to my wonderful wife, Jyoti Kapoor, and to my amazing daughters. Jyoti, your unwavering support and encouragement have been the foundation of every step I take. To my children, your curiosity and joy are a constant inspiration. Thank you all for being my biggest cheerleaders and for filling my life with purpose and happiness.

Sonu Kapoor (Canada)

This book is dedicated to the global development community. Keep learning!

Lars Gyrup Brink Nielsen (Denmark)

Contributors

Muhammad Awais is a highly regarded Lead Software Engineer and Angular enthusiast with over 9+ years of experience in creating exceptional web applications. Skilled in JavaScript, TypeScript, Angular, React, NodeJS, and Python, he's passionate about harnessing Artificial Intelligence to solve complex problems. As a tech author, mentor and open-source contributor he has shared insights in renowned publications, including Generative AI and JavaScript in Plain English.

Awais is dedicated to best practices in software development and thrives on building high-performance, seamless applications. Recognized as a Google Developer Mentor, Tech Speaker and AWS Community Builder, he regularly speaks on AI, angular, and modern web development, inspiring developers to push the boundaries of innovation.

Sonu Kapoor is a distinguished expert in web technologies, with a specialization in Angular, and JavaScript. Over a career spanning multiple decades and continents, Sonu has made a lasting impact on the software development community. Sonu's journey in technology started in Germany, where he was inspired by the rapid growth of the tech industry and the high demand for skilled developers.

This passion drove him to immerse himself in web programming languages, sparking a career that spanned the globe—from Germany to India, Canada, and the United States, before ultimately establishing himself in Toronto. His dedication to technological innovation and community contribution has earned him numerous recognitions, including his appointment to the Angular Collaborators Program, the prestigious Microsoft MVP award, and the Google Developer Expert award for Angular.

Lars Gyrup Brink Nielsen is a GitHub Star, Microsoft MVP, Nx Champion, and Angular Hero of Education. He's the co-founder of This is Learning and This is Angular, an open-source maintainer and co-organizes the in-person meetup AarhusJS. Lars is a published book author who does technical reviews on books from other authors.

Foreword

Generative Ai has undoubtedly entered your daily personal and professional life.Would it not be incredible to be able to implement AI-based use cases in web applications of your own?

This book is a happy marriage between modern Angular and Python development for the purpose of learning how to implement generative AI applications with Google Gemini and other large language model Ai services in Google Cloud Platform and AWS Bedrock. Muhammed Awais and Sonu Kapoor wrote this book in a way that beginners and skilled developers alike can achieve a strong foundation in the realm of generative AI web application development.

The first thing you will learn by reading this book is setting up an Angular application, how to write Angular components, and what Angular services are. As Angular services are provided to an inversion of control container and resolved using dependency injection, this is the next topic. Component communication is important in an Angular application so we will building blocks for this.

Other than a more basic AI starter recipe, the middle part of the book consists of recipes for inspiring and interactive AI-based use cases:

- A chatbot for conversational Ai
- A text-based sentiment analyzer
- A skill quiz generator
- A vision-based plant identifier
- An image-based mood detection
- A vision-based health/laboratory report

In the final part of the book, we learn how to create a web API using Python and Flask to support our AI applications. We go through connecting our backend application to Google AI services as well as AWS Bedrock with IAM access control and policies. Additionally, we will expand on the sentiment analyzer by feeding it feedback data from X via Kaggle, cleaning the data, and visualizing it with word clouds.

We leave you with the skills required to write fullstack AI applications using Angular, Flask, and Google Ai services. Where you go from here is up to you.

Lars Gyrup Brink Nielsen
Co-founder of This is Learning and This is Angular

Table of Contents

1. Getting started with Angular

What is Angular

Angular is one of the dynamic layers in web development and is a popular front-end framework aimed at enhancing web application development by providing users, designs, new features, etc. which was made and is maintained by Google. It features a component-based architecture which enables application developers to develop scalable and modular applications with relative ease. Angular enforces clean code principles, uses strong typing (TypeScript) and has wide-ranging features including dependency injection, two-way data binding, and routing. The large proportion of Angular's extensive ecosystem, including a comprehensive suite of developer tools, has predominated many large scale web applications. When you are building single-page applications (SPAs) or progressive web apps (PWAs), Angular supplies needed instruments for creating strong, performant, and highly interactive user interfaces.

Setting up the Development Environment

Before you can start building an Angular application, you'll need to set up the necessary tools and dependencies on your machine. The first steps are installing Node.js and NPM (Node Package Manager), as they are required to manage dependencies and run Angular development tools. You can download Node.js from the official site, and it comes bundled with NPM. Once installed, ensure that Node.js and NPM are available by running the following commands in your terminal:

```
$ node-v
$ npm -v
```

Next, you'll install the Angular CLI (Command Line Interface), a powerful tool that simplifies the process of creating and managing Angular projects. To install the Angular CLI globally on your system, run:

```
$ npm install -g @angular/cli
```

You will also need a code editor. Feel free to install your favorite editor. VS Code and Webstorm are two very popular choices.

With these tools installed, you're ready to create your first Angular project.

Creating Your First Angular Project

Creating an Angular project using the Angular CLI is straightforward. Open your terminal or command prompt and run the following command to create a new project:

```
$ ng new my-angular-app
```

The CLI will guide you through a series of prompts to set up the project. You'll be asked whether you want to include features like Angular routing and which stylesheet format (CSS, SCSS, etc.) you'd prefer. Once the process completes, the CLI will generate the project files and install all necessary dependencies.

To navigate to your project folder and launch the development server, use the following commands:

```
$ cd my-angular-app
$ ng serve
```

By default, the application will run at http://localhost:4200/. Open this URL in your browser, and you should see the default Angular welcome page.

Project Structure Overview

Angular follows a well-organized folder structure to help you manage large projects. Let's take a look at some key directories and files in the generated project:

- **`src/`**: The main folder where your application code resides.
  - **`app/`**: Contains the root module and components of your application.
  - **`assets/`**: Holds static assets like images and fonts.
  - **`environments/`**: Manages environment-specific configurations for development and production builds.
- **`angular.json`**: The main configuration file for Angular CLI, where you can configure build settings and other options.
- **`package.json`**: Contains project metadata and dependencies.
- **`tsconfig.json`**: Configures the TypeScript compiler settings.

Understanding this structure is crucial for effectively managing your Angular project, especially as it grows in complexity.

Running the Angular Application

Once your project is set up, you can easily run and test it. By default, the `ng serve` command spins up a local development server, watches your files for changes, and automatically reloads the app when changes are detected. This allows for rapid iteration during development.

To run the Angular application, simply execute:

```
$ ng serve
```

This will start the development server and make your app available at http://localhost:4200/. You can open this URL in a browser to see the running application. If you make changes to any of the files, Angular will automatically rebuild and reload the app, providing instant feedback.

For production, you'll typically run a different command to build an optimized version of your application:

```
$ ng build --prod
```

This will create a minified, production-ready version of your Angular app in the dist/ directory.

2. Angular App Anatomy: Understanding Standalone Components and Services

In modern Angular applications, standalone components provide a more streamlined approach to structuring an application. Unlike the traditional module-based approach, standalone components eliminate the need for NgModules, simplifying the overall structure while maintaining the power of Angular's component-based architecture. In addition to standalone components, services play an essential role in handling shared logic, data fetching, and application state.

Standalone Components

Standalone components are self-contained units of UI and logic that can be used independently without the need to be declared in a module. This makes them more lightweight and flexible for building Angular applications. A standalone component still consists of three main parts:

- TypeScript Class: Handles the logic, methods, and properties of the component.
- HTML Template: Defines the UI structure of the component.
- Styles (CSS/SCSS): Manages the component's styles, making it look visually consistent.

The key difference is that, unlike regular components, standalone components declare their own dependencies (like directives or pipes) directly, instead of relying on a module to do so. Here's an example of how to define a standalone component:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-my-standalone',
  templateUrl: './my-standalone.component.html',
  styleUrls: ['./my-standalone.component.css'],
  standalone: true,
  imports: [CommonModule, FormsModule], // Declare any dependencies
here
})
export class MyStandaloneComponent {
  // Component logic goes here
}
```

In this example, the standalone: true property tells Angular that this component is self-contained. We also specify the required Angular modules directly in the imports array. This design allows you to build and use components without needing to declare them in a module, leading to more modular, flexible application development.

Using Standalone Components

Standalone components can be bootstrapped directly in the application, allowing you to skip the traditional AppModule. For instance, you can modify the main.ts file to bootstrap a standalone component instead of a module:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { MyStandaloneComponent } from
'./app/my-standalone.component';

bootstrapApplication(MyStandaloneComponent);
```

With this, the Angular application will use the standalone component as its entry point, simplifying the setup process.

Services in Angular

Services are still a crucial part of Angular applications, whether you use standalone components or modules. They encapsulate business logic, handle API interactions, and manage shared state across your application. A service is typically injected into components using Angular's Dependency Injection (DI) system. For example:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class MyService {
  getData() {
    return 'Service Data';
  }
}
```

Here, the providedIn: 'root' property ensures that the service is available throughout the entire application. This service can now be injected into any standalone component:

```
import { Component } from '@angular/core';
import { MyService } from './my-service.service';

@Component({
  selector: 'app-my-standalone',
  templateUrl: './my-standalone.component.html',
  standalone: true,
})
export class MyStandaloneComponent {
  constructor(private myService: MyService) {}

  getDataFromService() {
    return this.myService.getData();
  }
}
```

In this case, services continue to provide a single source of truth across multiple components, ensuring that state and logic are centralized and easy to manage.

3. Dependency Injection in Standalone Components

Dependency Injection (DI) is one of Angular's core features that allows components to acquire dependencies such as services in a modular and reusable way. Traditionally, services were injected via Angular modules (NgModules). However, with standalone components, we can define dependencies directly in the component's metadata, making the overall architecture more streamlined and reducing the need for unnecessary modules.

What is Dependency Injection?

Dependency Injection is a design pattern where a class receives its dependencies from an external source rather than creating them itself. In Angular, services, which often encapsulate logic related to data fetching, state management, or other cross-cutting concerns, are injected into components or other services.

Angular's DI framework takes care of creating and managing service instances, making the application more modular and testable.

Using Dependency Injection in Standalone Components

With standalone components, instead of providing services via NgModules, you can directly specify services in the providers array of the component's metadata.

Basic Example: Injecting a Service into a Standalone Component
Let's create a simple service and inject it into a standalone component.

Generate a Service: First, create a service using the Angular CLI:

```
$ ng generate service my-service
```

This will create a service file `my-service.service.ts` that looks like this:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root', // Ensures that the service is provided
globally.
})
export class MyService {
```

```
  getMessage() {
    return 'Hello from MyService!';
  }
}
```

Create a Standalone Component: Now, create a standalone component that will consume this service. You can generate a component using:

```
$ ng generate component my-component --standalone
```

Inject the Service into the Component: Modify the generated component to inject the `MyService` and use its functionality.

```
import { Component } from '@angular/core';
import { MyService } from './my-service.service'; // Import the
service

@Component({
  selector: 'app-my-component',
  template: `<h1>{{ message }}</h1>`,
  standalone: true,
  providers: [MyService] // Provide the service here for this
component
})
export class MyComponent {
  message: string;

  // Inject MyService into the component's constructor
  constructor(private myService: MyService) {
    this.message = this.myService.getMessage();
  }
}
```

In this example:

- The `providers` array in the `@Component()` decorator specifies that the `MyService` should be available for dependency injection within the `MyComponent`.
- The service is injected into the component's constructor using Angular's DI system.

- The component then calls `this.myService.getMessage()` to retrieve a message and display it.

Bootstrapping the Component: In your `main.ts`, bootstrap the component as follows:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { MyComponent } from './app/my-component';

bootstrapApplication(MyComponent)
  .catch(err => console.error(err));
```

When you run the application, `MyComponent` will display the message provided by `MyService`.

Local vs. Global Providers in Standalone Components

With standalone components, you have the flexibility to provide services at different levels:

- **Local Provider (Component-Specific)**: If you provide a service inside the `providers` array of a standalone component (as in the previous example), Angular creates a new instance of the service every time the component is created. This service is only available to that specific component and its descendants.
- **Global Provider (Application-Wide)**: When a service is decorated with `providedIn: 'root'` inside the `@Injectable()` decorator (like `MyService`), Angular creates a single instance of that service and shares it across the entire application. This is known as a singleton service.
  For most services that handle shared data or application-wide functionality, using `providedIn: 'root'` is the recommended approach. However, if a service should have a narrower scope (such as being available only within a specific component or feature), providing it at the component level is useful.

Injecting Services into Multiple Standalone Components

Let's say you have another standalone component that also needs to inject the same `MyService`. You can either provide the service globally (using `providedIn: 'root'`) or provide it locally in each component's `providers` array.

Using a Global Provider (Preferred)

With `MyService` already provided globally (as seen in the earlier example with `providedIn: 'root'`), you don't need to list it in the `providers` array for each component. Any component can inject and use the service without any extra configuration.

```
@Component({
  selector: 'app-another-component',
  template: `<p>{{ message }}</p>`,
  standalone: true
})
export class AnotherComponent {
  message: string;

  constructor(private myService: MyService) {
    this.message = this.myService.getMessage();
  }
}
```

Both `MyComponent` and `AnotherComponent` will share the same instance of `MyService` since it's provided globally. This is the most common approach for services that handle global state or data.

Best Practices for Dependency Injection in Standalone Components

- **Keep Services Scoped Appropriately**: Decide whether a service should be available globally or locally. If a service is meant to be used across multiple components or features, provide it in the root injector (`providedIn: 'root'`). If it's specific to a single component or its children, provide it locally.
- **Use Angular CLI for Service Generation**: The Angular CLI automatically sets `providedIn: 'root'` for new services, ensuring that they're available globally unless you explicitly change the provider strategy.
- **Testability**: Dependency injection makes it easier to write unit tests for your components. You can mock injected services when writing tests to isolate the logic being tested.

4. Component Communication

In any Angular application, components need to interact and share data with each other. Whether you're building a small or large-scale application, effective communication between components is crucial for maintaining a clean, organized, and scalable architecture. Angular offers several ways to manage communication, each designed for different use cases.

In this chapter, we will explore the primary methods for component communication, including the use of **Input** and **Output** decorators, event binding, and service-based communication.

Input and Output Decorators

Angular provides two powerful decorators—`@Input` and `@Output`—that allow components to pass data between parent and child components.

Input Decorator

The `@Input` decorator allows a parent component to pass data to a child component. This is useful when you want a child component to display or process data provided by its parent.

Example: Parent-to-child data binding using `@Input`.

```
// child.component.ts
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `<h3>{{ message }}</h3>`
})
export class ChildComponent {
  @Input() message: string = ''; // The parent will bind this
property
}

// parent.component.html
<app-child [message]="parentMessage"></app-child>

// parent.component.ts
@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html'
})
export class ParentComponent {
  parentMessage: string = 'Hello from Parent!';
}
```

In this example, the `ChildComponent` uses the `@Input` decorator to receive the `message` property from the `ParentComponent`. The value of `parentMessage` is passed down to the child component's `message` property, and it’s displayed in the template.

Output Decorator

The `@Output` decorator allows a child component to send data back to its parent. It is typically paired with Angular's `EventEmitter` to emit custom events.

Example: Child-to-parent data binding using `@Output` and `EventEmitter`.

```
// child.component.ts
import { Component, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `<button (click)="sendMessage()">Send Message</button>`
})
export class ChildComponent {
  @Output() messageEvent = new EventEmitter<string>();

  sendMessage() {
    this.messageEvent.emit('Hello from Child!');
  }
}

// parent.component.html
<app-child (messageEvent)="receiveMessage($event)"></app-child>

// parent.component.ts
@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html'
})
export class ParentComponent {
  message: string = '';

  receiveMessage(message: string) {
    this.message = message;
  }
}
```

In this example, the child component emits an event when the button is clicked. The parent component listens to the `messageEvent` and invokes the `receiveMessage()` method, updating its `message` property with the emitted value.

Event Binding and Emitting

Angular's event binding mechanism allows components to listen for events from child components and HTML elements. Event binding is typically used when the child component emits events like `click`, `input`, `change`, or custom events created using `EventEmitter`.

Event Binding Syntax

The general syntax for event binding is:

```
<child-component (eventName)="methodToHandleEvent($event)"></child-component>
```

In this syntax, `eventName` can be a standard DOM event or a custom event emitted by the child component using the `@Output` decorator and `EventEmitter`.

Angular supports standard DOM events, such as:

- `(click)`
- `(keyup)`
- `(input)`
- `(submit)`

You can bind to these events directly in your templates and handle them in the component's class.

Example: Handling a standard `click` event.

```
<button (click)="handleClick()">Click Me</button>
```

```
handleClick() {
  console.log('Button clicked!');
}
```

In this example, when the button is clicked, the `handleClick()` method in the component will be called.

Service-based Communication

When two components that do not have a parent-child relationship need to communicate, services are the preferred approach. Services can act as a shared data source or mediator, facilitating communication between unrelated components.

Creating a Shared Service

You can create a service to store the shared data and inject it into the components that need to communicate.

Example: Service-based communication.

```
// message.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class MessageService {
  private message: string = '';

  setMessage(newMessage: string) {
    this.message = newMessage;
  }

  getMessage(): string {
    return this.message;
  }
}

// sender.component.ts
import { Component } from '@angular/core';
import { MessageService } from './message.service';

@Component({
  selector: 'app-sender',
  template: `<button (click)="sendMessage()">Send Message</button>`,
})
export class SenderComponent {
  constructor(private messageService: MessageService) {}

  sendMessage() {
    this.messageService.setMessage('Message from Sender');
  }
}

// receiver.component.ts
import { Component, OnInit } from '@angular/core';
import { MessageService } from './message.service';

@Component({
  selector: 'app-receiver',
  template: `<h3>{{ message }}</h3>`,
})
export class ReceiverComponent implements OnInit {
  message: string = '';
```

```
  constructor(private messageService: MessageService) {}

  ngOnInit() {
    this.message = this.messageService.getMessage();
  }
}
```

In this example, `SenderComponent` and `ReceiverComponent` communicate via the `MessageService`. The `SenderComponent` sets the message in the service, and `ReceiverComponent` retrieves it.

Advantages of Service-based Communication

- **Loose coupling**: Components don't need to know about each other. This decoupling enhances modularity.
- **Scalability**: As applications grow, service-based communication scales better compared to tightly coupled parent-child communication.
- **Data Persistence**: Services persist data for the entire lifecycle of the application or for specific routes, depending on how they are provided (at the module or root level).

Using ViewChild and ViewChildren for Direct Interaction

Sometimes you may need direct access to child components, especially when handling more complex interactions between components. Angular provides the `@ViewChild` and `@ViewChildren` decorators to allow parent components to interact directly with child components in the component tree.

Using ViewChild

`@ViewChild` allows a parent component to access a single child component instance.

Example: Using `@ViewChild` to access child component methods.

```
// child.component.ts
@Component({
  selector: 'app-child',
  template: `<h3>Child Component</h3>`,
})
export class ChildComponent {
  greet() {
    return 'Hello from the Child!';
  }
}
```

```
// parent.component.ts
@Component({
  selector: 'app-parent',
  template: `<app-child></app-child> <button
(click)="greetChild()">Greet Child</button>`,
})
export class ParentComponent {
  @ViewChild(ChildComponent) child!: ChildComponent;

  greetChild() {
    alert(this.child.greet());
  }
}
```

In this example, the parent component accesses the child component's `greet()` method using `@ViewChild` and triggers it when a button is clicked.

Using ViewChildren

`@ViewChildren` is used when you need to access multiple instances of child components.

5. Gemini Recipes with Angular

Introduction

Gemini is a powerful and versatile artificial intelligence model developed by Google AI. It represents a significant advancement in the field of AI, demonstrating exceptional capabilities across various tasks. Here's a breakdown of its key features and potential applications:

Key Features:

- **Multimodal:** Gemini excels in handling diverse types of information, including text, code, audio, images, and video. This multimodal nature allows it to understand and respond to complex queries and requests.
- **Large-Scale:** Gemini is built on a massive scale, enabling it to learn from vast amounts of data and develop a deep understanding of the world.
- **Generalization:** It can generalize its knowledge and apply it to new situations, making it adaptable and flexible.
- **Efficient:** Gemini can operate efficiently on various platforms, from data centers to mobile devices, making it accessible for a wide range of applications.

Potential Applications:

- **Coding Assistance:** Gemini can generate, explain, and debug code in popular programming languages like Python, Java, C++, and Go. It can also translate code between different languages.
- **Natural Language Understanding:** Gemini can understand and respond to natural language queries with high accuracy and fluency. It can also generate creative text formats like poems, scripts, musical pieces, email, letters, etc.
- **Image and Video Analysis:** Gemini can analyze images and videos to extract information, identify objects, and understand the context.
- **AI Assistants:** Gemini can power advanced AI assistants that can engage in natural conversations, provide helpful information, and complete tasks.
- **Enterprise Applications:** Gemini can be used to enhance productivity and efficiency in various enterprise applications, such as customer service, data analysis, and content creation.

In essence, Gemini is a versatile AI model that can be applied to a wide range of tasks, making it a valuable tool for individuals and businesses alike.

5.1 Gemini “Starter Recipe” with Angular (Step by Step):

We'll delve into the exciting world of Google AI's Gemini model and explore how to seamlessly integrate it into your Angular applications. Gemini, a powerful language model, offers a wide range of capabilities, from text generation to code writing. By leveraging Gemini's potential, you can create innovative and intelligent applications.

Setting Up the Angular Project

By following these steps, you can effectively integrate Google AI's Gemini model into your Angular applications. Gemini's powerful capabilities can enhance your user experience and create innovative applications.

1. **Create a New Angular Project:**

```
$ ng new gemini-angular-app
```

2. **Install Required Dependencies:**

```
$ cd gemini-angular-app
$ npm install @google/generative-ai
```

The Google AI JavaScript SDK offers JavaScript developers a straightforward way to integrate the Gemini API, granting access to Gemini models developed by Google

DeepMind. These models are designed from the ground up to be multimodal, allowing seamless reasoning across text, images, and code.

Configuring Google AI API Key

1. **Obtain an API Key:**
   - Visit the Google Cloud Platform console.
   - Create a new project or select an existing one.
   - Enable the Generative AI API.
   - Create an API key and store it securely.
2. **Set Up Environment Variables:**
   - Create an `environment.ts` file in your `/environments` directory inside `/src`
   - Add the following line, replacing `YOUR_API_KEY` with your actual API key:

```
export const environment = {
  production: false,
  googleAiKey: 'YOUR_API_KEY ',
};
```

Integrating Gemini into an Angular Component

1. **Create a New Component:**

```
$ ng generate component text-based-gemini
```

2. **Create the Gemini Service:**

```
import { Injectable } from '@angular/core';
import { GenerativeModel, GoogleGenerativeAI } from
'@google/generative-ai';
import { environment } from '../../environments/environment';

@Injectable({
    providedIn: 'root'
})
export class GeminiGoogleAiService {
    // instance to initiate Gemini - Google Ai with API_KEY
    private genAI: GoogleGenerativeAI;

    constructor() {
        this.genAI = new
GoogleGenerativeAI(environment.googleAiKey);
    }
```

```
    /**
     * Communicate with Gemini - Google Ai using text prompt
     */
    async askGemini(prompt: string): Promise<string> {
        const model: GenerativeModel =
this.genAI.getGenerativeModel({
            model: 'gemini-1.5-flash',
        });

        const content = await model.generateContent(prompt);
        return content.response.text();
    }
}
```

3. **Inject the Gemini Service in "`text-based-gemini.component.ts`":**

```
import { Component, OnInit, signal } from '@angular/core';
import { GeminiGoogleAiService } from
'../../services/gemini-google-ai/gemini-google-ai.service';

@Component({
  selector: 'app-text-based-gemini',
  standalone: true,
  imports: [],
  templateUrl: './text-based-gemini.component.html',
  styleUrl: './text-based-gemini.component.scss',
})
export class TextBasedGeminiComponent implements OnInit {
  response = signal('');

  constructor(private genAiService: GeminiGoogleAiService) {}

  ngOnInit(): void {
    this.askGemini('Hey, How are you doing today?');
  }

  /**
   * Communication with Gemini using Frontend Compatible
Service
   * @param text
   */
  askGemini(text: string) {
    this.genAiService.askGemini(text).then(
```

```
      (result: string) => {
        this.response.set(result);
      },
      (error: Error) => {
        alert(error);
      });
  }
}
```

Routing and Displaying Results

1. **Update the “`app.config`” file:**

```
import { Routes } from '@angular/router';

export const routes: Routes = [
    {
        path: '',
        redirectTo: 'starter',
        pathMatch: 'full',
    },
    {
        path: 'starter',
        loadComponent: () =>

import('./text-based-gemini/text-based-gemini.component').then(
(c) => c.TextBasedGeminiComponent),
    },
];
```

2. **Adding RouterOutlet in your dependency array inside “`app.component.ts`”:**

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {
  title = 'angular-gemini-recipes';
```

```
}
```

3. **Display the Response in the "`text-based-gemini.component.html`":**

```
<h1>Gemini Starter</h1>
<p>Response: {{response() ? response() : 'Loading ...'}}</p>
```

Furthermore for getting more details on Gemini Api usage, please visit:
https://ai.google.dev/gemini-api/docs

5.2 Gemini "Chat Recipe" with Angular (Step by Step):

In this chapter, we'll create a chat interface in Angular that mimics the ChatGPT interface. Our goal is to build a responsive chat component that takes user input, sends it to the Gemini Google AI service, and displays the AI's response as a conversation.

We'll go step-by-step to generate all necessary artifacts: the component, HTML template, CSS, and the associated TypeScript code.

Generating the Angular Component

To begin, create the `ChatComponent` in the `/pages/chat` folder.

```
$ ng generate component pages/chat --standalone
```

This creates the files chat.component.ts, chat.component.html, and chat.component.scss for the standalone ChatComponent.

Structuring the HTML Template

In `chat.component.html`, we'll set up the structure for our chat interface. This includes a message display area, a loading indicator (spinner) while waiting for responses, and a user input form to capture user text.

```
<div class="chat-container">
  <div class="messages">
    <div
      *ngFor="let msg of messages"
      [ngClass]="msg.isUser ? 'user-message' : 'bot-message'"
    >
      {{ msg.text }}
    </div>
    <div *ngIf="isLoading" class="loader"></div>
```

```
  </div>
  <form (submit)="onSubmit()" class="input-form">
    <input
      type="text"
      [(ngModel)]="userInput"
      (keyup.enter)="onSubmit()"
      placeholder="Type your message here..."
      name="chatInput"
      required
    />
  </form>
</div>
```

This template:

- Displays messages dynamically based on whether they are from the user or the AI.
- Shows a loading spinner when waiting for the AI's response.
- Provides an input box for the user to type messages, which can be submitted by pressing Enter.

Styling the Chat Interface

In `chat.component.scss`, we'll add styles to create a modern chat interface with rounded corners, background colors, and appropriate alignment for both user and AI messages.

```
.chat-container {
  display: flex;
  flex-direction: column;
  height: 100%;
  max-width: 600px;
  margin: 0 auto;
  border: 1px solid #ddd;
  border-radius: 8px;
}

.messages {
  flex: 1;
  overflow-y: auto;
  padding: 10px;
  max-height: 600px;
}

.user-message,
.bot-message {
  padding: 8px 12px;
```

```
  border-radius: 8px;
  margin-bottom: 10px;
  word-wrap: break-word;
}

.user-message {
  margin-left: auto;
  align-self: flex-end;
  width: fit-content;
  background-color: #cce7ff;
  text-align: right;
}

.bot-message {
  width: 80%;
  align-self: flex-start;
  background-color: #f0f0f0;
  text-align: left;
}

.input-form {
  display: flex;
  border-top: 1px solid #ddd;
  padding: 10px;
}

input {
  flex: 1;
  padding: 12px;
  font-size: 1rem;
  border: none;
  outline: none;
  color: #ffffff;
  background-color: #333333;
  border-radius: 20px;
}

/* Loader styles */
.loader {
  border: 4px solid #f3f3f3;
  border-top: 4px solid #3498db;
  border-radius: 50%;
  width: 24px;
  height: 24px;
  animation: spin 1s linear infinite;
```

```
  margin: 10px auto;
}

@keyframes spin {
  0% {
    transform: rotate(0deg);
  }
  100% {
    transform: rotate(360deg);
  }
}
```

In this style sheet:

- User messages are aligned to the right, with a `fit-content` width for an adaptive message box.
- AI messages align to the left, taking up a wider area.
- The input field has rounded corners and a blackish background.
- A loading spinner appears while awaiting a response.

Implementing the Chat Logic in TypeScript

In `chat.component.ts`, we'll implement the chat logic, including handling user input, calling the `GeminiGoogleAiService` to get responses, and displaying loading feedback.

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
import { GeminiGoogleAiService } from
'../../services/gemini-google-ai/gemini-google-ai.service';

@Component({
  selector: 'app-chat',
  standalone: true,
  imports: [FormsModule, CommonModule],
  templateUrl: './chat.component.html',
  styleUrls: ['./chat.component.scss'],
})
export class ChatComponent {
  messages: { text: string; isUser: boolean }[] = [];
  userInput: string = '';
  isLoading: boolean = false;

  constructor(private geminiService: GeminiGoogleAiService) {}
```

```
  async onSubmit() {
    if (!this.userInput.trim()) return;

    this.isLoading = true;

    // Append user message to the chat log
    this.messages.push({ text: this.userInput, isUser: true });

    const userMessage = this.userInput;
    this.userInput = '';

    try {
      const response = await
this.geminiService.askGemini(userMessage);
      // Append AI response to the chat log
      this.messages.push({ text: response, isUser: false });
    } catch (err) {
      this.messages.push({
        text: "Error: Couldn't reach the Gemini API",
        isUser: false,
      });
      console.error(err);
    }

    this.isLoading = false;
  }
}
```

This code:

- Initializes an array `messages` to store chat history.
- Uses `isLoading` to manage loading feedback.
- Calls `GeminiGoogleAiService.askGemini()` to get AI responses and adds them to the `messages` array for display.

Connecting with the AI Service

In the previous chapter, we implemented the `GeminiGoogleAiService` to interact with Google's Gemini model. Here's a summary:

```
import { Injectable } from '@angular/core';
import { GenerativeModel, GoogleGenerativeAI } from
'@google/generative-ai';
```

```
import { environment } from '../../../environments/environment';

@Injectable({
  providedIn: 'root',
})
export class GeminiGoogleAiService {
  private genAI: GoogleGenerativeAI;

  constructor() {
    this.genAI = new GoogleGenerativeAI(environment.googleAiKey);
  }

  async askGemini(prompt: string): Promise<string> {
    const model: GenerativeModel = this.genAI.getGenerativeModel({
      model: 'gemini-1.5-flash',
    });

    const content = await model.generateContent(prompt);
    return content.response.text();
  }
}
```

Summary

In this chapter, we've covered the creation of a conversational chat interface in Angular, including:

- Generating the component with a structured HTML template.
- Styling the chat interface to look modern and user-friendly.
- Implementing chat logic in TypeScript to handle user input, loading feedback, and response retrieval.
- Leveraging `GeminiGoogleAiService` for AI responses.

This setup now provides a fully functional chat application that mirrors a conversational AI interface, giving users a real-time interaction experience.

5.3 Gemini "Sentiment Analyzer Recipe" with Angular (Step by Step):

This chapter builds a chat interface in Angular that performs sentiment analysis on user input using Google AI's Natural Language API. Each input message is analyzed, and the sentiment (positive, neutral, or negative) is displayed with color-coded feedback.

Why Sentiment Analysis is Useful

Sentiment analysis provides immediate insights into customer emotions and attitudes, which can improve customer service and experience. For example:

- **Customer Feedback Monitoring**: Businesses can analyze feedback and prioritize responses to negative comments.
- **Social Media Sentiment Tracking**: Brands can gauge public opinion and respond to emerging issues quickly.
- **Product Reviews**: E-commerce sites can use sentiment scores to highlight positive or critical feedback, helping new customers make informed decisions.

Sentiment Analysis Request Structure

When you send a request to the Google Cloud Natural Language API to analyze sentiment, the request payload needs to be structured in a specific way. This request contains the text you want to analyze and specifies the encoding and document type.

Here's an example of what the request payload will look like:

```
{
  "document": {
    "type": "PLAIN_TEXT",
    "content": "I love using Angular for web development! It's so
powerful and efficient."
  },
  "encodingType": "UTF8"
}
```

Breakdown of Request Arguments:

1. **`document`**:
    - This is the core part of the request. It defines the text content that you want to analyze.
    - **`type`**: Specifies the type of document content. In this case, it's `PLAIN_TEXT`, which indicates you are sending plain text. Other options might include `HTML`, but `PLAIN_TEXT` is most commonly used for sentiment analysis.
    - **`content`**: This is the actual text you want to analyze. It can be any string, such as a user review, social media post, or chat message. The API will analyze the sentiment in this text.
2. **`encodingType`**:
    - This parameter specifies the text encoding used in the request. For most cases, **UTF-8** is a good choice because it can handle any Unicode characters.
    - **`UTF8`**: Ensures that the text is properly encoded for processing by the API.

Example Request:

Let's say you are analyzing the following customer review:

"The product was good, but the customer service was terrible."

The request would look like this:

```
{
  "document": {
    "type": "PLAIN_TEXT",
    "content": "The product was good, but the customer service was
terrible."
  },
  "encodingType": "UTF8"
}
```

How the API Handles the Request:

- **Text**: The API will parse the text to analyze its sentiment.
- **Document Type**: The API knows that the text is plain and will handle it accordingly.
- **Encoding Type**: The text will be processed in UTF-8 encoding to ensure special characters (if any) are handled correctly.

Example of Full Request in the Service:

In the service `GeminiGoogleSentimentAiService`, you send this structured request using Angular's `HttpClient`. Here's how it's integrated into the code:

```
generateSentimentAnalysis(prompt: string):
Observable<SentimentResponse> {
  const body = {
    document: {
      type: 'PLAIN_TEXT',
      content: prompt,
    },
    encodingType: 'UTF8',
  };

  return this.#httpClient.post<SentimentResponse>(
    this.#sentimentApiUrl,
    body,
  );
}
```

`prompt`: This is the dynamic input from the user, which will be sent as the `content` in the request payload.

`body`: The request body is constructed with the required structure and passed to the `HttpClient` for the POST request.

`HttpClient.post()`: Sends the POST request to the API with the sentiment analysis parameters.

Sentiment Response Explanation

1. **Sentiment Score**:
    - **Range**: The score ranges from **-1.0 to 1.0**.
    - **Meaning**:
        - A positive score (closer to 1.0) indicates **positive sentiment** (e.g., joy or satisfaction).
        - A negative score (closer to -1.0) signals **negative sentiment** (e.g., anger or disappointment).
        - A score close to zero suggests **neutral sentiment**, meaning there's not enough emotional tone in the text to classify it as strongly positive or negative.
    - **Interpretation**: This score allows you to judge the tone of a message. For example, in customer feedback, a positive score can indicate a satisfied customer, while a negative score could highlight dissatisfaction.
2. **Sentiment Magnitude**:
    - **Range**: The magnitude is a non-negative value and has no set upper bound.
    - **Meaning**: This value reflects the **intensity or strength of the sentiment** within the text.
        - A higher magnitude means more emotionally intense language, regardless of whether it's positive or negative.
        - A low magnitude (near zero) indicates text with very little emotional content.
    - **Interpretation**: For instance, a score of 0.8 with a high magnitude (e.g., 3.0) would suggest strong positive feedback, while a score of -0.6 with a similar magnitude reflects strong dissatisfaction.

Example Breakdown

Consider these scenarios in customer service feedback:

- **Scenario 1**: A review with a score of **0.9** and a magnitude of **2.5** likely represents highly positive, enthusiastic feedback.
- **Scenario 2**: A review with a score of **-0.7** and a magnitude of **3.2** could suggest severe dissatisfaction or anger.

- **Scenario 3**: A review with a score of **0.1** and magnitude **0.3** would be more neutral, indicating little emotional content.

By combining both score and magnitude, you can better interpret user sentiment.

Having a clear understanding of the request and response. Let's build our example.

Generating the Angular Component

In this step, we'll create the main component for our sentiment analysis page. Place this in `src/app/pages/sentiment-analyzer`.

```
$ ng generate component pages/sentiment-analyzer --standalone
--skip-tests
```

This will create the following files:

- `sentiment-analyzer.component.ts`
- `sentiment-analyzer.component.html`
- `sentiment-analyzer.component.scss`

Add HTML Structure and Styling

Open `sentiment-analyzer.component.html` and add the chat UI structure.

```
<div class="chat-container">
  <div class="messages">
    <div *ngFor="let msg of messages" [ngClass]="msg.isUser ?
'user-message' : 'bot-message'">
      <ng-container *ngIf="msg.isUser">
        {{ msg.text }}
      </ng-container>
      <ng-container *ngIf="!msg.isUser">
        <div *ngIf="msg.error" class="bot-message negative">{{
msg.error }}</div>
        <div *ngIf="!msg.error" class="bot-message"
[class]="getSentimentClass(msg.response?.documentSentiment?.score)">
          Score: {{ msg.response?.documentSentiment?.score }}<br />
          Magnitude: {{ msg.response?.documentSentiment?.magnitude }}
        </div>
      </ng-container>
    </div>

    <div *ngIf="isLoading" class="loader"></div>
    <form (submit)="onSubmit()" class="input-form">
```

```
      <textarea rows="1" placeholder="Enter text to analyze..."
[(ngModel)]="userInput" (keyup.enter)="onSubmit()"
name="chatInput"></textarea>
    </form>
  </div>
</div>
```

Then, add styling in `sentiment-analyzer.component.scss`: We will use the same styles from the previous chat application, except with the following additions:

```
.chat-container { /* Chat layout styles */ }
.messages { /* Message list styling */ }
.user-message { /* User message styling */ }
.bot-message { /* Bot message styling */ }
.loader { /* Loader styles */ }
.positive { background-color: #28a745; }
.neutral { background-color: #6c757d; }
.negative { background-color: #dc3545; }
```

Create the Sentiment Service

Create a service to handle API requests for sentiment analysis.

```
$ ng generate service services/gemini-google-ai/gemini-google-sentiment-ai
```

In `gemini-google-sentiment-ai.service.ts`, add the logic to call Google AI's Natural Language API:

```
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { environment } from '../../../environments/environment';
import { SentimentResponse } from '../../model/sentiment.response';

@Injectable({
  providedIn: 'root',
})
export class GeminiGoogleSentimentAiService {
  private httpClient = inject(HttpClient);
  private sentimentApiUrl =
`https://language.googleapis.com/v1/documents:analyzeSentiment?key=${
environment.sentimentKey}`;
```

```
  generateSentimentAnalysis(text: string):
Observable<SentimentResponse> {
    const body = {
      document: {
        type: 'PLAIN_TEXT',
        content: text,
      },
      encodingType: 'UTF8',
    };
    return
this.httpClient.post<SentimentResponse>(this.sentimentApiUrl, body);
  }
}
```

Add the Logic to the Component

In `sentiment-analyzer.component.ts`, inject the service and add functions to handle user input, send data to the API, and display results.

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { catchError, EMPTY, take } from 'rxjs';
import { GeminiGoogleSentimentAiService } from
'../../services/gemini-google-ai/gemini-google-sentiment-ai.service';

@Component({
  selector: 'app-sentiment-analyzer',
  templateUrl: './sentiment-analyzer.component.html',
  styleUrls: ['./sentiment-analyzer.component.scss'],
  imports: [FormsModule],
})
export class SentimentAnalyzerComponent {
  messages = [];
  userInput = '';
  isLoading = false;

  private sentimentService = inject(GeminiGoogleSentimentAiService);

  onSubmit(): void {
    if (!this.userInput.trim()) return;
    this.isLoading = true;
    this.messages.push({ text: this.userInput, isUser: true });
    const userMessage = this.userInput;
    this.userInput = '';
```

```
    this.sentimentService.generateSentimentAnalysis(userMessage)
      .pipe(
        catchError(() => {
          this.isLoading = false;
          this.messages.push({ error: 'Error analyzing sentiment',
isUser: false });
          return EMPTY;
        }),
        take(1)
      )
      .subscribe((response) => {
        this.isLoading = false;
        this.messages.push({ response, isUser: false });
      });
  }

  getSentimentClass(score?: number): string {
    return score ? (score > 0 ? 'positive' : score < 0 ? 'negative' :
'neutral') : '';
  }
}
```

Add a Route for the Sentiment Analyzer

In `app.routes.ts`, define a route for the sentiment analyzer:

```
{
  path: 'sentiment-analyzer',
  loadComponent: () =>

import('./pages/sentiment-analyzer/sentiment-analyzer.component').the
n(
      (c) => c.SentimentAnalyzerComponent,
    ),
}
```

Summary

In this chapter, we explored how to integrate Google's Natural Language API for sentiment analysis into an Angular application. By building a simple component, we allowed users to input text and receive sentiment analysis results, including a sentiment score (ranging from -1 to 1) and magnitude, which indicates the strength of the sentiment.

We covered the setup of the Angular component, service, and API integration, focusing on the request structure, including the `PLAIN_TEXT` document type and `UTF-8` encoding. Real-world use cases for sentiment analysis include customer feedback analysis, social media sentiment monitoring, and product review evaluation.

By the end of this chapter, you should have a working understanding of how to build sentiment-aware applications that leverage Google's AI to analyze text and provide meaningful insights.

5.4 "Skill Quiz Generator Recipe" with Angular (Step by Step):

The **Skill Quiz Generator** application leverages **Google Gemini's Generative AI** and **Angular** to dynamically create personalized skill quizzes based on user input. This component enables recruiters or team leads to assess candidates' technical knowledge by generating quizzes on-demand for various technologies. This chapter walks you through setting up the project, creating necessary components, services, and integrating Google Gemini AI for generating quiz questions.

Understanding the Components:

1. **Skill Quiz Generator Component (`skill-quiz-generator.component.ts`):**

   This component handles the user interface for generating quizzes. It takes user input for candidate name, technology, and desired quiz length. It then utilizes the `GeminiGoogleAiService` to generate questions through Gemini and displays the quiz using the `GeneratedQuizComponent`.

2. **Generated Quiz Component (`generated-quiz.component.ts`):**

   This component receives the generated quiz data from `SkillQuizGeneratorComponent` and displays it with input fields for the candidate to answer. It also handles form submission logic (currently a placeholder).

3. **GeminiGoogleAiService (`gemini-google-ai.service.ts`):**

   This service interacts with the Google Generative AI platform to send prompts to Gemini and retrieve generated quiz questions.

4. **QuizRecipeHelperService (`quiz-recipe-helper.service.ts`):**

   This service parses the raw text response from Gemini and transforms it into a structured quiz format suitable for displaying in the UI.

5. **IQuiz Interface (`iquiz.ts`):**

   This interface defines the structure of a quiz object with title, instructions, and questions.

Defining Routes

Define the route for the **Skill Quiz Generator** page in your routing file. This will allow navigation to the quiz generator component.

```
{
    path: 'skill-quiz-generator',
    loadComponent: () =>
      import(
        './pages/skill-quiz-generator/skill-quiz-generator.component'
      ).then((c) => c.SkillQuizGeneratorComponent),
},
```

Creating Interfaces for Quiz Data

Define interfaces for the quiz structure, including quiz title, instructions, and questions.

```
export interface IQuizQuestion {
    question: string;
}

export interface IQuiz {
    title: string;
    instructions: string;
    questions: IQuizQuestion[];
}
```

Building the Skill Quiz Generator Component

Example Use Case: A recruiter or a hiring manager can use this component to quickly generate a custom quiz for a candidate based on their specific skills (e.g., React, JavaScript) and the required number of questions. This can help assess the candidate's proficiency before an interview or coding challenge.

This component encapsulates several important features for building interactive, AI-powered forms in Angular, and can be expanded or modified to suit various quiz-generation or assessment

Component Template (HTML)

Create the UI for capturing candidate details and quiz preferences. The form includes fields for the candidate's name, chosen technology, and the number of questions. A section below displays the generated quiz.

```
<div class="quiz-form">
    <h2 class="mb-4 pb-4 border-bottom">Skill Quiz Generator</h2>

    <form [formGroup]="candidateForm" (ngSubmit)="onSubmit()"
class="row" novalidate>
        <div class="mb-3 col-md-4 col-12">
            <label for="candidateName" class="form-label">Candidate
Name</label>
            <input type="text" id="candidateName"
class="form-control" formControlName="candidateName" required>

            @if (hasError('candidateName', ['required'])) {
            <div class="text-danger">
                Candidate Name is required.
            </div>
            }
        </div>
        <div class="mb-3 col-md-4 col-12">
            <label for="technology"
class="form-label">Technology</label>
            <select id="technology" class="form-select"
formControlName="technology" required>
                <option value="">Select a Technology</option>
                @for (tech of technologies; track $index) {
                    <option [value]="tech">{{ tech }}</option>
                }
            </select>

            @if (hasError('technology', ['required'])) {
            <div class="text-danger">
                Technology is required.
            </div>
            }
        </div>
        <div class="mb-3 col-md-4 col-12">
            <label for="questionsLength" class="form-label">Questions
Length</label>
            <input type="number" id="questionsLength"
class="form-control" formControlName="questionsLength" min="5"
                required>
```

```
            @if (hasError('questionsLength', ['min', 'required'])) {
            <div class="text-danger">
                Minimum value is 5.
            </div>
            }
        </div>

        <div class="col-12 text-center">
            <button type="submit" class="btn btn-primary"
[disabled]="candidateForm.invalid">Generate Quiz</button>
        </div>
    </form>
</div>

<div class="my-5">
    <h2 class="mb-4 pb-4 border-bottom">Ai Generated Quiz</h2>
    @if (formattedQuizResponse) {
        <app-generated-quiz [quiz]="formattedQuizResponse" />
    } @else {
        <p>Quiz isn't generated yet.</p>
    }
</div>
```

Component Class (TypeScript)

Define the component class to manage form validation, handle the form submission, and request quiz generation from Google Gemini.

```
import { Component } from '@angular/core';
import { GeneratedQuizComponent } from
'./generated-quiz/generated-quiz.component';
import { FormBuilder, FormGroup, ReactiveFormsModule, Validators }
from '@angular/forms';
import { GeminiGoogleAiService } from
'../../services/gemini-google-ai/gemini-google-ai.service';
import { QuizRecipeHelperService } from
'../../helpers/quiz-recipe-helper.service';
import { LoadingService } from
'../../services/loading/loading.service';
import { IQuiz } from '../../interfaces/iquiz';

@Component({
  selector: 'app-skill-quiz-generator',
  standalone: true,
  imports: [GeneratedQuizComponent, ReactiveFormsModule],
```

```
  templateUrl: './skill-quiz-generator.component.html',
  styleUrl: './skill-quiz-generator.component.scss'
})
export class SkillQuizGeneratorComponent {
  candidateForm!: FormGroup;
  technologies = ['Angular', 'React', 'JavaScript', 'TypeScript',
'Python', 'Java'];
  formattedQuizResponse: IQuiz = {
    title: '',
    instructions: '',
    questions: []
  };

  constructor(
    private fb: FormBuilder,
    private geminiService: GeminiGoogleAiService,
    private quizHelperService: QuizRecipeHelperService,
    private loadingService: LoadingService
  ) {
    this.formInit();
  }

  formInit() {
    this.candidateForm = this.fb.group({
      candidateName: ['', Validators.required],
      technology: ['', Validators.required],
      questionsLength: [5, [Validators.required, Validators.min(5)]]
    });
  }

  hasError(controlName: string, errorNames: string[]) {
    const control = this.candidateForm.get(controlName);
    return control && control.touched && errorNames.some(errorName =>
control.hasError(errorName));
  }

  onSubmit() {
    if (this.candidateForm.valid) {
      this.loadingService.onLoadingToggle();

      const prompt: string = `Generate a technical skill quiz for a
candidate named ${this.candidateForm.value.candidateName} who
specializes in ${this.candidateForm.value.technology}. The quiz
should be tailored to the candidate's proficiency level, aiming for a
balanced mix of ${this.candidateForm.value.questionsLength} questions
```

```
that assess both foundational knowledge and practical application,
just return questions strings`;

      this.geminiService.askGemini(prompt).then(
        (res: string) => {
          const formattedResponse =
this.quizHelperService.formatAiReponseQuizString(res);
          this.formattedQuizResponse = formattedResponse;

          this.candidateForm.reset();
          this.loadingService.onLoadingToggle();
        },
        (error: Error) => {
          console.error(error);
          this.loadingService.onLoadingToggle();
        }
      );
    } else {
      console.log('Form is invalid');
    }
  }
}
```

The SkillQuizGeneratorComponent is an Angular component designed to generate a technical skill quiz for a candidate based on the selected technology. The component leverages AI through the GeminiGoogleAiService to dynamically generate quiz questions tailored to the candidate's proficiency level. Below is a detailed breakdown of the component's structure and functionality:

Key Properties:

1. candidateForm: A FormGroup used to manage the form that collects information about the candidate and their technology.
    - Fields in the form:
        - candidateName: The name of the candidate (required).
        - technology: The technology that the candidate specializes in (required).
        - questionsLength: The number of questions in the quiz, which is set to default to 5 and should be a minimum of 5.
2. technologies: An array of string values representing different technologies (e.g., Angular, React, JavaScript, etc.) that the user can select from when generating the quiz.
3. formattedQuizResponse: An object that stores the formatted response from the AI service, structured to match the IQuiz interface. It contains:
    - title: The title of the quiz.
    - instructions: Instructions for the quiz.
    - questions: A list of questions generated by the AI.

Constructor:

- Injecting Services:
  - FormBuilder: Used to initialize and manage the reactive form (candidateForm).
  - GeminiGoogleAiService: A service that communicates with the Gemini AI (Google AI) to generate quiz questions based on the candidate's profile.
  - QuizRecipeHelperService: A helper service to format and process the quiz questions received from the AI.
  - LoadingService: A service to manage the loading state (show/hide loading indicators during asynchronous operations).
- Form Initialization:
  - The form is initialized using formInit(), which sets up the form group with the necessary form controls and their validation rules.

Methods:

1. **formInit():**
   - Initializes candidateForm with three controls: candidateName, technology, and questionsLength. Each control has its own validation rules (e.g., required and min for questionsLength).
2. **hasError():**
   - A helper function to check if a specific form control has validation errors. It takes the control name and an array of error names (e.g., required, minlength, etc.) and returns true if the control has any of the specified errors.
3. **onSubmit():**
   - This method is triggered when the form is submitted. It checks if the form is valid and, if so, sends a prompt to the GeminiGoogleAiService to generate quiz questions based on the form input (candidate name, technology, and number of questions).
   - Prompt Structure: The prompt is dynamically constructed to include details from the form, instructing the AI to generate a technical skill quiz that balances foundational knowledge and practical application.
   - AI Response Handling: Upon receiving the response from the Gemini AI, it is formatted using quizHelperService.formatAiReponseQuizString() and stored in formattedQuizResponse. The form is then reset, and the loading state is toggled off.
4. **Error Handling:**
   - If the form is invalid or if an error occurs during the AI request, appropriate error messages are logged, and the loading state is turned off.

Template and Interaction:

- The GeneratedQuizComponent is used within this component to display the quiz once it's generated. It is passed the formattedQuizResponse as the quiz input.
- The loadingService.onLoadingToggle() is used to control a loading spinner while the AI is processing the quiz generation.

How the Component Works:

1. The user fills out the form by entering their name, selecting their technology expertise, and specifying the number of questions for the quiz.
2. When the form is submitted, the component sends a request to the GeminiGoogleAiService to generate the quiz questions based on the user's input.
3. The AI returns a list of quiz questions, which are formatted and displayed on the page via the GeneratedQuizComponent.
4. The form resets after the quiz is generated, and the loading state is toggled off once the process is complete.

Key Features:

- Dynamic Form Validation: Ensures that the user provides necessary inputs before generating the quiz.
- AI Integration: The quiz generation is powered by Gemini AI, making it highly customizable based on the candidate's skills and preferences.
- Loading State Management: The UI shows a loading indicator while waiting for the AI to generate the quiz, improving user experience.
- Form Reset: After quiz generation, the form is reset, ready for the next quiz creation.

AI Integration with Google Gemini Service

Create a service to interact with **Google Gemini** and send the AI prompt for quiz generation.

```
import { Injectable } from '@angular/core';
import { GenerativeModel, GoogleGenerativeAI } from
'@google/generative-ai';
import { environment } from '../../../environments/environment';

@Injectable({
  providedIn: 'root',
})
export class GeminiGoogleAiService {
  // instance to initiate Gemini - Google Ai with API_KEY
  readonly #genAI: GoogleGenerativeAI = new GoogleGenerativeAI(
    environment.googleAiKey,
  );

  /**
   * Communicate with Gemini - Google Ai using text prompt
   */
  async askGemini(prompt: string): Promise<string> {
    const model: GenerativeModel = this.#genAI.getGenerativeModel({
      model: 'gemini-1.5-flash',
    });
```

```
    const content = await model.generateContent(prompt);
    return content.response.text();
  }
}
```

Formatting AI Response (helper function) for Display

We received the following response from Google AI's Gemini model in response to our prompt, which requested a dynamically generated quiz for the specified tech stack with adjustable question length:

```
## Angular Technical Skill Quiz for Muhammad Awais

**Instructions:** Answer the following questions to the best of your
ability.

**1.  What is the difference between `ngModel` and `[(ngModel)]` in
Angular templates?**

**2. Explain the role of the `@Input()` and `@Output()` decorators in
Angular components.**

**3. Describe the purpose of the `ngOnInit()` lifecycle hook and give
an example of when you would use it.**

**4. How does Angular handle change detection? Explain the difference
between `ChangeDetectionStrategy.Default` and
`ChangeDetectionStrategy.OnPush`.**

**5.  Explain the concept of Dependency Injection (DI) in Angular.
How do you inject a service into a component?**

**6. Describe the process of routing in Angular. How do you configure
routes and navigate between components?**

**7. How would you implement form validation in an Angular component?
Provide an example using `FormsModule` or `ReactiveFormsModule`.**

**8. Explain the difference between `async` and `async/await` in
Angular. When would you use each approach?**

**9.  Describe the purpose of the `HttpClient` service in Angular.
How would you make a GET request to an API endpoint?**
```

```
**10. You are developing a component that displays a list of
products. How would you handle the following scenarios using Angular
techniques:**

* **Dynamically add new products to the list.**
* **Remove products from the list.**
* **Update product details.**
```

To handle AI responses, create a helper service that formats the AI response string into the quiz structure.

```
import { Injectable } from '@angular/core';
import { IQuiz, IQuizQuestion } from '../interfaces/iquiz';

@Injectable({
  providedIn: 'root'
})
export class QuizRecipeHelperService {

  formatAiReponseQuizString(quizString: string): IQuiz {
    const quizData: IQuiz = {
      title: '',
      instructions: '',
      questions: []
    };

    // Split the string by double new lines to separate sections
    const sections: string[] = quizString.split(/\n\s*\n/);

    // Extract title
    quizData.title = sections[0].replace('## ', '').trim();

    // Extract instructions
    const instructionSection = sections[1]?.trim();
    if (instructionSection?.startsWith('**Instructions:**')) {
      quizData.instructions =
instructionSection.replace('**Instructions:**', '').trim();
    }

    // Extract questions
    const questionSections = sections.slice(2); // All questions are
after the first two sections
```

```
    questionSections.forEach((section: string) => {
      const lines: string[] = section.split('\n').map((line: string)
=> line.trim()).filter((line: string) => line.length > 0);
      const mainQuestion: string = lines[0].replace(/^\d+\.\s*/,
'').trim();

      const quizQuestion: IQuizQuestion = {
        question: mainQuestion
      };

      quizData.questions.push(quizQuestion);
    });

    return quizData;
  }
}
```

After applying the above helper for formatting and constructing the desired object, the JSON object for the Angular Technical Skill Quiz has been structured to contain the following main elements:

1. **Title**: A string that holds the title of the quiz.
2. **Instructions**: A string that provides a set of instructions for the quiz taker.
3. **Questions**: An array of objects, where each object contains the question text.

```
{
    "title": "Angular Technical Skill Quiz for Muhammad Awais",
    "instructions": "Answer the following questions to the best of
your ability.",
    "questions": [
        {
            "question": "**1.  What is the difference between
`ngModel` and `[(ngModel)]` in Angular templates?**"
        },
        {
            "question": "**2. Explain the role of the `@Input()` and
`@Output()` decorators in Angular components.**"
        },
        {
            "question": "**3. Describe the purpose of the
`ngOnInit()` lifecycle hook and give an example of when you would use
it.**"
        },
        {
            "question": "**4. How does Angular handle change
```

```
detection? Explain the difference between
`ChangeDetectionStrategy.Default` and
`ChangeDetectionStrategy.OnPush`.**"
        },
        {
            "question": "**5.  Explain the concept of Dependency
Injection (DI) in Angular. How do you inject a service into a
component?**"
        },
        {
            "question": "**6. Describe the process of routing in
Angular. How do you configure routes and navigate between
components?**"
        },
        {
            "question": "**7. How would you implement form validation
in an Angular component? Provide an example using `FormsModule` or
`ReactiveFormsModule`.**"
        },
        {
            "question": "**8. Explain the difference between `async`
and `async/await` in Angular. When would you use each approach?**"
        },
        {
            "question": "**9.  Describe the purpose of the
`HttpClient` service in Angular. How would you make a GET request to
an API endpoint?**"
        },
        {
            "question": "**10. You are developing a component that
displays a list of products. How would you handle the following
scenarios using Angular techniques:**"
        },
        {
            "question": "* **Dynamically add new products to the
list.**"
        }
    ]
}
```

Displaying the Generated Quiz

This code block represents an Angular template that displays a dynamically generated quiz and allows users to submit their answers.

```
<div class="container mt-5">
    <h1 class="text-center">{{ quiz.title }}</h1>
    <p class="lead text-center">{{ quiz.instructions }}</p>

    <form [formGroup]="quizForm" (ngSubmit)="onSubmit()">
        <div formArrayName="answers">
            @for (question of quiz.questions; track $index; let i =
$index) {
            <div class="card mb-3">
                <div class="card-body">
                    <h5 class="card-title"
[innerHTML]="question.question"></h5>
                    <div class="form-group">
                        <label for="answer-{{ i }}">Your
Answer:</label>
                        <textarea id="answer-{{ i }}"
class="form-control" rows="4" formControlName="{{i}}"
                            required></textarea>
                        @if (hasError(i)) {
                            <div class="text-danger">
                                Answer is required.
                            </div>
                        }
                    </div>
                </div>
            </div>
            }
        </div>

        @if (quiz.questions.length > 0) {
            <div class="text-center mt-4">
                <button type="submit" class="btn btn-primary">Submit
Answers</button>
            </div>
        }
    </form>
</div>
```

The `GeneratedQuizComponent` displays the quiz generated by Gemini. This component receives quiz data through an `@Input` property and handles form submission for quiz answers.

```
import { Component, Input, SimpleChanges } from '@angular/core';
import { FormBuilder, FormGroup, ReactiveFormsModule, Validators }
```

```
from '@angular/forms';
import { IQuiz } from '../../../interfaces/iquiz';

@Component({
  selector: 'app-generated-quiz',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './generated-quiz.component.html',
  styleUrl: './generated-quiz.component.scss'
})
export class GeneratedQuizComponent {
  @Input() quiz: IQuiz = {
    title: '',
    instructions: '',
    questions: []
  };
  quizForm!: FormGroup;

  constructor(private fb: FormBuilder) {
    this.initQuizForm();
  }

  ngOnChanges(changes: SimpleChanges): void {
    if (changes['quiz'] && this.quiz.questions) {
      this.generateQuizForm();
    }
  }

  initQuizForm() {
    this.quizForm = this.fb.group({
      answers: this.fb.array([])
    });
  }

  generateQuizForm() {
    const answersArray = this.fb.array(this.quiz.questions.map(() =>
this.fb.control('', Validators.required)));
    this.quizForm.setControl('answers', answersArray);
  }

  onSubmit() {
    if (this.quizForm.valid) {
      console.log(this.quizForm.value);
      // TODO: Handle form submission logic here
    }
```

```
  }

  hasError(controlIndex: number): boolean {
    const control =
this.quizForm.get('answers')?.get(`${controlIndex}`);
    return !!(control?.invalid || control?.touched);
  }
}
```

This Angular component, `GeneratedQuizComponent`, is designed to render a dynamically generated quiz form. Here's a detailed explanation of its structure and functionality:

1. **Imports and Metadata**:
   - The component imports `FormBuilder`, `FormGroup`, `ReactiveFormsModule`, and `Validators` from Angular's forms module for building and validating a reactive form.
   - It also imports `IQuiz`, an interface that likely defines the structure for a quiz object, including fields such as `title`, `instructions`, and `questions`.
   - `ReactiveFormsModule` is added to the component's `imports` array, making reactive form directives available in the component's template.
   - The component is standalone and uses `selector: 'app-generated-quiz'`, which can be used in templates as `<app-generated-quiz>`, and points to its HTML and SCSS files.
2. **Component Properties**:
   - `@Input() quiz`: This property allows the parent component to pass in a `quiz` object, which is of type `IQuiz` and contains fields for `title`, `instructions`, and an array of `questions`.
   - `quizForm`: A `FormGroup` instance used to hold and manage form controls for the quiz answers.
3. **Constructor and Form Initialization**:
   - The constructor injects `FormBuilder` to easily create form controls and groups.
   - `initQuizForm()`: This method initializes `quizForm` with a `FormGroup` containing an empty array named `answers`, which will later hold form controls for each quiz question.
4. **ngOnChanges Lifecycle Hook**:
   - The `ngOnChanges` method listens for changes to the `quiz` input. If the `quiz` input changes and contains questions, it calls `generateQuizForm()` to regenerate the form with the new quiz questions.
5. **`generateQuizForm` Method**:
   - This method creates a form array (`answersArray`) where each control is initialized with an empty string (`''`) and a `Validators.required` validator.

    - `setControl` is used to assign this array of controls to the `answers` array in `quizForm`, aligning with the number of questions in `quiz`.
6. **`onSubmit` Method**:
    - When the form is submitted, `onSubmit()` checks if `quizForm` is valid. If so, it logs the form's value (an array of answers) to the console. Here, additional logic (e.g., sending data to a server) could be added for handling the form submission.
7. **`hasError` Method**:
    - This helper method checks if a specific answer control is invalid or has been touched. It uses the `controlIndex` parameter to find the respective control in the `answers` form array and returns `true` if there's an error.

This component enables flexible quiz generation by dynamically creating form controls based on an input quiz object. Each answer is required, and the form checks for errors upon submission. This setup allows for straightforward integration into larger applications where quiz data may vary across different quizzes.

The **Skill Quiz Generator** combines **Angular** and **Google Gemini AI** to create an interactive platform that generates tailored quizzes on-demand. By following this step-by-step guide, you've built an AI-driven component that can generate skill-based quizzes for various

5.5 Plant Identifier Recipe with Angular (Step by Step) – Integrating Gemini Image Vision Model

In this chapter, we will build a plant identifier using Angular, where we'll use the Google Gemini AI model to process images, analyze them, and give accurate identification results. The implementation includes creating a new Angular component, handling image uploads, and interacting with the Google Generative AI (Gemini model) to analyze the plant in the image. We will also handle sentiment analysis and provide actionable recommendations if the plant requires attention.

Prerequisites:

Before proceeding, make sure you have:

- Angular CLI installed and a new Angular project set up.
- Access to the Google Cloud Vision API or Google Gemini AI API with the appropriate API key.

---

1. Creating the Angular Component

The first step is to create a new Angular component that will handle the image upload, display the uploaded image, and show the AI analysis result.

1.1. Generate the Component

To generate the component, run the following command:

```
$ ng generate component recognise-plant-ai
```

1.2. Component HTML (Template)

In the generated `recognise-plant-ai.component.html`, This template allows the user to upload an image, trigger the identification process, and display the result. we'll include the following HTML structure:

```
<div class="container">
    <h2 class="mb-4 pb-4 border-bottom text-center">Plant Identify
using Vision Api - Google Ai</h2>
    <div class="row justify-content-center">
        <div class="mb-3 col-md-6 col-12">
            <img [src]='image' class="w-100 mb-5 d-block m-auto" />

            <!-- Image upload input -->
            <input type="file" (change)="onFileChange($event)"
accept="image/*" class="form-control" />

            <button (click)="onPlantIdentify()" [disabled]="!image"
class="btn btn-primary mt-2 w-100 mt-3">
                Identify Plant
            </button>

            <!-- Display Analysis Result -->
            @if (result) {
            <div class="pt-5 text-center">
                <h3>Ai Reponse</h3>
                <p>{{result}}</p>
            </div>
            }
        </div>
    </div>
</div>
```

1.2.1 Breakdown of the Template:

- **Image Preview (`<img [src]='image' />`)**: This element will dynamically display the image that the user uploads. It's bound to the `image` property of the component.
- **Image Upload Input (`<input type="file">`)**: This allows the user to select an image file. The `(change)` event triggers the `onFileChange` method when a file is selected.
- **Identify Button (`<button (click)="onPlantIdentify()">`)**: When clicked, this button triggers the plant identification process by calling the `onPlantIdentify()` method. The button is disabled until an image is uploaded.
- **Result Display (`<div *ngIf="result">`)**: Once the AI process completes, the result is shown in this section.

Here, we use a file input for the user to upload the image. Once the image is uploaded, we trigger the `onPlantIdentify` method that interacts with Google's Gemini AI service to get the analysis result.

1.3. Component Typescript (Logic)

Now, let's set up the logic for the component. The component class handles all the logic for uploading the image, interacting with the AI, and displaying the results. Here's the `recognise-plant-ai.component.ts`:

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { GeminiGoogleAiService } from
'../../services/gemini-google-ai/gemini-google-ai.service';
import { ImageHelperService } from
'../../helpers/image-helper.service';
import { IVisionAi } from '../../interfaces/image-ai';
import { LoadingService } from
'../../services/loading/loading.service';

@Component({
  selector: 'app-recognise-plant-ai',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './recognise-plant-ai.component.html',
  styleUrl: './recognise-plant-ai.component.scss'
})
export class RecognisePlantAiComponent {
  image: string = '';
  inlineImageData: IVisionAi = {};
  imageFile: File | null = null;
  result: any | null = null;
```

```
  private geminiAiService = inject(GeminiGoogleAiService);
  private imageHelper = inject(ImageHelperService);
  private loadingService = inject(LoadingService);

  onFileChange(e: Event) {
    const input = e.target as HTMLInputElement;

    if (input?.files && input.files[0]) {
      const file = input.files[0];

      // Getting base64 from file to render in DOM
      this.imageHelper.getBase64(file)
        .then((result: any) => {
          this.image = result
          console.log(this.image);
        })
        .catch(e => console.log(e));

      // Generating content model for Gemini Google AI
      this.imageHelper.fileToGenerativePart(file)
        .then((image: IVisionAi) => {
          this.inlineImageData = image;
          console.log(this.inlineImageData);
        });
    } else {
      console.log("No file selected.");
    }
  }

  onPlantIdentify() {
    this.loadingService.onLoadingToggle();

    this.geminiAiService.onImagePrompt('Which type of plant is this
share the details and if the plant seems in issue please highlights
the steps to fix and make plant sustainable.', this.inlineImageData)
      .then((response) => {
        this.result = response;
        this.loadingService.onLoadingToggle();
      })
      .catch(e => console.log(e));
  }
}
```

Key Elements of the Code:

- `onFileChange(e: Event)`:

    - This method is triggered when the user selects a file. It handles the image file processing and converts it to base64 so that it can be previewed in the UI.
    - The image is then passed to a helper service (`ImageHelperService`) that converts it into a format that the Gemini AI service can process (base64 encoded).
- `onPlantIdentify()`:
    - Once the image is selected and the user clicks the "Identify Plant" button, this method is invoked.
    - It calls the `onImagePrompt()` method from the `GeminiGoogleAiService`, passing in a predefined text prompt and the image data in the expected format.
    - The method waits for a response from Gemini AI and then displays the result.

2. Setting Up the Services

2.1. GeminiGoogleAiService

This service will communicate with Google's Gemini AI, specifically using the Vision API model, to process the plant image. This service interacts with Google's Gemini AI API to send the image data and text prompt, and receive analysis results.

```
import { Injectable } from '@angular/core';
import { GenerativeModel, GoogleGenerativeAI } from
'@google/generative-ai';
import { environment } from '../../../environments/environment';

@Injectable({
  providedIn: 'root',
})
export class GeminiGoogleAiService {
  // instance to initiate Gemini - Google Ai with API_KEY
  readonly #genAI: GoogleGenerativeAI = new GoogleGenerativeAI(
    environment.googleAiKey,
  );

  /**
   * Communicate with Gemini - Google Ai using text prompt
   */
  async askGemini(prompt: string): Promise<string> {
    const model: GenerativeModel = this.#genAI.getGenerativeModel({
      model: 'gemini-1.5-flash',
    });

    const content = await model.generateContent(prompt);
    return content.response.text();
```

```
  }

   /**
   * Communicate with Gemini - Google Ai using image prompt
   */
   async onImagePrompt(prompt: string, imageinlineData: any):
Promise<any> {
    try {
      const model: GenerativeModel = this.#genAI.getGenerativeModel({
        model: 'gemini-1.5-flash',
      });

      const result = await model.generateContent([prompt,
imageinlineData]);

      if (!result || !result.response) {
        throw new Error('Failed to get a valid response from the
model');
      }

      return result.response.text();
    } catch (error) {
      console.error('Error generating content:', error);
      throw new Error('Failed to generate content with Gemini -
Google AI.');
    }
  }
}
```

Key Elements:

- **API Key**: The `GoogleGenerativeAI` instance is initialized with the API key from the environment, allowing access to Google's Gemini AI service.
- **`askGemini(prompt: string)`**: This method sends a text prompt to Gemini AI and returns the response text.
- **`onImagePrompt(prompt: string, imageinlineData: any)`**: This method sends both the image data (in the required format) and a text prompt to the AI for processing. The result is returned after the AI processes the request.

2.2. ImageHelperService

This service is responsible for handling the image data (conversion to base64 format and preparing it for Gemini's AI format).

```
import { Injectable } from '@angular/core';
import { IVisionAi } from '../interfaces/image-ai';

@Injectable({
  providedIn: 'root'
})
export class ImageHelperService {

  constructor() { }

  getBase64(file: File): Promise<string | ArrayBuffer | null> {
    return new Promise((resolve, reject) => {
      const reader = new FileReader();
      reader.readAsDataURL(file);
      reader.onload = () => resolve(reader.result);
      reader.onerror = (error) => reject(`Error: ${error}`);
    });
  }

  async fileToGenerativePart(file: File): Promise<IVisionAi> {
    const base64EncodedData = await new Promise<string>((resolve,
reject) => {
        const reader = new FileReader();
        reader.onloadend = () => {
            const result = reader.result as string;
            const base64Data = result.split(',')[1];  // Extract
base64 data part only
            resolve(base64Data);
        };
        reader.onerror = (error) => reject(`File reading error:
${error}`);
        reader.readAsDataURL(file);
    });

    return {
        inlineData: { data: base64EncodedData, mimeType: file.type },
    };
}
}
```

3. Interface Definition

Define an interface to represent the image and its inline data for processing.

```
interface InlineData {
  data?: string;
  mimeType?: string;
}

export interface IVisionAi {
  inlineData?: InlineData;
}
```

Conclusion

In this chapter, we've built a plant identifier application using Angular and the Google Gemini AI service. We created the necessary components, services, and interfaces to handle image uploads, communicate with Google's Vision API, and display results on the web page. This recipe serves as a foundation for integrating machine learning and AI models in Angular apps, with additional opportunities to enhance and expand the functionality.

5.6 Health/Laboratory Report using Vision API - Google AI

This chapter explains how to implement a health or laboratory report analysis system using the Google Vision API. By utilizing Google's Generative AI (Gemini), the system processes uploaded health-related images (e.g., medical reports, lab tests) and analyzes the content to provide feedback regarding diseases, first aid treatments, and hospital recommendations. The following implementation steps detail how to set up the frontend in Angular, interact with the API, and display the results.

The frontend of the application consists of a basic UI where users can upload health or laboratory report images, and the system will analyze them using the **Google Vision API** and **Gemini AI**.

Component Setup for Health Report Analysis

The component `HealthReportAiComponent` allows users to upload an image and analyze it using the Vision API. Here's the HTML structure and logic for interacting with the service:

HTML Template: `heath-report-ai.component.html`

```
<div class="container">
    <h2 class="mb-4 pb-4 border-bottom text-center">Health/Laboratory
Report using Vision Api - Google Ai</h2>
    <div class="row justify-content-center">
        <div class="mb-3 col-md-6 col-12">
            <img [src]='image' class="w-100 mb-5 d-block m-auto" />
```

```
            <!-- Image upload input -->
            <input type="file" (change)="onFileChange($event)"
accept="image/*" class="form-control" />

            <button (click)="onHealthReportAnalysis()"
[disabled]="!image" class="btn btn-primary mt-2 w-100 mt-3">
                Analyse Laboratory Report
            </button>

            <!-- Display Analysis Result -->
            @if (result) {
            <div class="pt-5 text-center">
                <h3>Ai Reponse on Health Report</h3>
                <p>{{result}}</p>
            </div>
            }
        </div>
    </div>
</div>
```

Explanation:

1. **Image Upload**: The `input` tag allows users to upload image files (e.g., laboratory reports).
2. **Image Preview**: The `img` tag dynamically updates to show the preview of the uploaded image.
3. **Analysis Button**: When clicked, it triggers the `onHealthReportAnalysis()` method to send the image data to the backend for analysis.
4. **Result Display**: After the analysis, the result is shown under the image, displaying the feedback from the AI model.

Component Logic for File Handling and API Interaction

The Angular component (`HealthReportAiComponent`) is responsible for handling the uploaded image, interacting with the backend service (`GeminiGoogleAiService`), and displaying the results. In the TypeScript file, we handle the logic of image uploading, converting the image to a base64 format for AI analysis, and invoking the backend service to analyze the image.

Component Class: `heath-report-ai.component.ts`

```
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { GeminiGoogleAiService } from
```

```
'../../services/gemini-google-ai/gemini-google-ai.service';
import { ImageHelperService } from
'../../helpers/image-helper.service';
import { IVisionAi } from '../../interfaces/image-ai';
import { LoadingService } from
'../../services/loading/loading.service';

@Component({
  selector: 'app-heath-report-ai',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './heath-report-ai.component.html',
  styleUrl: './heath-report-ai.component.scss'
})
export class HeathReportAiComponent {
  image: string = '';
  inlineImageData: IVisionAi = {};
  imageFile: File | null = null;
  result: any | null = null;

  private geminiAiService = inject(GeminiGoogleAiService);
  private imageHelper = inject(ImageHelperService);
  private loadingService = inject(LoadingService);

  onFileChange(e: Event) {
    const input = e.target as HTMLInputElement;

    if (input?.files && input.files[0]) {
      const file = input.files[0];

      // Getting base64 from file to render in DOM
      this.imageHelper.getBase64(file)
        .then((result: any) => {
          this.image = result
        })
        .catch(e => console.log(e));

      // Generating content model for Gemini Google AI
      this.imageHelper.fileToGenerativePart(file)
        .then((image: IVisionAi) => {
          this.inlineImageData = image;
        });
    } else {
      console.log("No file selected.");
    }
```

```
  }

  onHealthReportAnalysis() {
    this.loadingService.onLoadingToggle();

    this.geminiAiService.onImagePrompt('Analysethe provided health
report of a patient and give feedback on the disease and some first
aid treatment and also provide the best hospital to consult with
doctor category to consult.', this.inlineImageData)
      .then((response) => {
        this.result = response;
        this.loadingService.onLoadingToggle();
      })
      .catch(e => console.log(e));
  }
}
```

Explanation:

1. **onFileChange()**: This method reads the file from the user input, converts it into a base64 string for image preview, and prepares the data for sending to the Vision API.
2. **onHealthReportAnalysis()**: Once the image is ready, this method triggers the AI model to analyze the health report and send feedback. The response is stored in the `result` variable, which is displayed to the user.

Interfacing with Google Vision AI (Gemini API)

To interact with Google's Vision API, we use the `GeminiGoogleAiService`. This service sends a request with the image data and a prompt to the Gemini model, which processes the health report and returns the analysis results. The backend communicates with **Gemini AI** using the **Google Generative AI API**. This service handles sending the image data and analysis request to Gemini and retrieves the AI-generated response. It also includes a mechanism to handle image content from the frontend, process it into a usable format, and send it to Google AI for analysis.

Service Class: `gemini-google-ai.service.ts`

```
import { Injectable } from '@angular/core';
import { GenerativeModel, GoogleGenerativeAI } from
'@google/generative-ai';
import { environment } from '../../../environments/environment';

@Injectable({
  providedIn: 'root',
})
```

```
export class GeminiGoogleAiService {
  // instance to initiate Gemini - Google Ai with API_KEY
  readonly #genAI: GoogleGenerativeAI = new GoogleGenerativeAI(
    environment.googleAiKey,
  );

  /**
   * Communicate with Gemini - Google Ai using text prompt
   */
  async askGemini(prompt: string): Promise<string> {
    const model: GenerativeModel = this.#genAI.getGenerativeModel({
      model: 'gemini-1.5-flash',
    });

    const content = await model.generateContent(prompt);
    return content.response.text();
  }

   /**
   * Communicate with Gemini - Google Ai using image prompt
   */
   async onImagePrompt(prompt: string, imageinlineData: any):
Promise<any> {
    try {
      const model: GenerativeModel = this.#genAI.getGenerativeModel({
        model: 'gemini-1.5-flash',
      });

      const result = await model.generateContent([prompt,
imageinlineData]);

      if (!result || !result.response) {
        throw new Error('Failed to get a valid response from the
model');
      }

      return result.response.text();
    } catch (error) {
      console.error('Error generating content:', error);
      throw new Error('Failed to generate content with Gemini -
Google AI.');
    }
  }
}
```

Explanation:

- **onImagePrompt()**: This method sends a request to the Google Gemini model with a prompt that explains the desired analysis (e.g., feedback on disease, first aid treatments). It uses the image data (in base64 format) prepared earlier and returns the AI-generated response.

Helper Service for Image Handling

The `ImageHelperService` is used for processing the uploaded image, converting it to base64 for preview and extracting the necessary data for sending to the AI service. This service helps convert the image into a base64 format, which is required for interacting with Google Vision API. It also processes the image data and prepares it for AI analysis.

Helper Service Class: `image-helper.service.ts`

```
import { Injectable } from '@angular/core';
import { IVisionAi } from '../interfaces/image-ai';

@Injectable({
  providedIn: 'root'
})
export class ImageHelperService {

  constructor() { }

  getBase64(file: File): Promise<string | ArrayBuffer | null> {
    return new Promise((resolve, reject) => {
      const reader = new FileReader();
      reader.readAsDataURL(file);
      reader.onload = () => resolve(reader.result);
      reader.onerror = (error) => reject(`Error: ${error}`);
    });
  }

  async fileToGenerativePart(file: File): Promise<IVisionAi> {
    const base64EncodedData = await new Promise<string>((resolve,
reject) => {
        const reader = new FileReader();
        reader.onloadend = () => {
            const result = reader.result as string;
            const base64Data = result.split(',')[1];  // Extract
base64 data part only
            resolve(base64Data);
        };
        reader.onerror = (error) => reject(`File reading error:
${error}`);
        reader.readAsDataURL(file);
```

```
    });

    return {
        inlineData: { data: base64EncodedData, mimeType: file.type },
    };
}
}
```

Explanation:

- **getBase64()**: Converts an image file into a base64 string, which can be used for preview.
- **fileToGenerativePart()**: Extracts the base64-encoded data from the file, which is required for sending to the Gemini API.

Conclusion

By utilizing **Google Vision API** and **Gemini AI**, this system allows users to easily upload a health or laboratory report, which is then analyzed by AI to generate useful feedback, including potential disease diagnoses, first aid treatments, and recommendations for the best hospitals or specialists to consult.

This integration leverages **Google's advanced AI models** to offer deep insights into medical reports, helping to automate and assist in healthcare decision-making. The system can be expanded further by including other features like medical record analysis, symptoms prediction, and integration with real-time health data.

5.7 Building an Image-Based Mood Detection App Using Google Gemini in Angular

This chapter covers the development of an Angular application that detects emotions from facial expressions in uploaded images using Google Gemini's Vision AI. This application demonstrates how to utilize cloud-based image analysis with REST APIs, giving developers a practical approach to implementing advanced emotion-detection features in web applications. We will cover the front-end design, backend API integration, and necessary configurations to ensure seamless communication between the application and the Vision AI API.

Prerequisites

1. **Angular Development Environment**: Ensure that you have Angular CLI set up.
2. **Google Cloud Account**: Access to Google Cloud with permissions for Vision AI API.
3. **API Key**: Request your own API key from Google, which will be used to authenticate API requests. Insert this key into the environment configuration file.

Setting Up the Mood Detection Component

We'll create an Angular component called `MoodDetectorComponent`, which consists of the following features:

1. **Image Upload**: Allows users to upload an image for analysis.
2. **Mood Detection**: Calls Google Gemini's Vision API to detect moods from the image.
3. **Mood Display**: Shows the mood likelihood results with color-coded labels.

Setting Up the Environment

First, configure the environment for your Angular application to securely store the Google Vision API key.

In `src/environments/environment.ts`:

```
export const environment = {
  …
  visionKey: 'YOUR_GOOGLE_VISION_API_KEY'
};
```

Replace `'YOUR_GOOGLE_VISION_API_KEY'` with the API key you obtained from Google Cloud.

Creating the HTML Structure

Create the HTML file `mood-detector.component.html` to manage the image upload interface and display mood detection results.

```
<div class="feedback-analyzer container">
  <h2 class="mb-4 pb-4 border-bottom">Detect Mood</h2>
  <div class="row">
    <div class="mb-3 col-md-4 col-12">
      <img *ngIf="imageAsBase64" [src]="imageAsBase64" class="w-100
mb-5 d-block m-auto" />
      <input type="file" (change)="onFileChange($event)"
accept="image/*" class="form-control" />
      <button (click)="detectMood()" class="btn btn-primary mt-2"
[disabled]="!imageAsBase64">Detect mood</button>

      <div *ngIf="result">
        <h3>Mood Analysis</h3>
        <ul>
          <li [ngClass]="getMoodLabel(result.joyLikelihood) |
```

```
lowercase">Joy: {{ getMoodLabel(result.joyLikelihood) }}</li>
          <li [ngClass]="getMoodLabel(result.sorrowLikelihood) |
lowercase">Sorrow: {{ getMoodLabel(result.sorrowLikelihood) }}</li>
          <li [ngClass]="getMoodLabel(result.angerLikelihood) |
lowercase">Anger: {{ getMoodLabel(result.angerLikelihood) }}</li>
          <li [ngClass]="getMoodLabel(result.surpriseLikelihood) |
lowercase">Surprise: {{ getMoodLabel(result.surpriseLikelihood)
}}</li>
        </ul>
      </div>
    </div>
  </div>
</div>
```

Applying CSS for Visual Feedback

```
.feedback-analyzer {
  display: flex;
  flex-direction: column;
}

.high {
  color: red;
}
.moderate {
  color: orange;
}
.low {
  color: green;
}
.none {
  color: gray;
}
```

Creating the Service for Google Vision API Integration

In `google-gemini-vision-ai.service.ts`, implement a service to make HTTP requests to the Google Vision API. This service will handle mood detection by sending a base64-encoded image to the Vision AI API.

```
import { Injectable, inject } from '@angular/core';
```

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { environment } from '../../../environments/environment';
import { FaceAnnotationResponse } from
'../../model/face-annotation-response.model';

@Injectable({
  providedIn: 'root',
})
export class GoogleGeminiVisionAiService {
  private httpClient = inject(HttpClient);
  private readonly geminiApiUrl =
`https://vision.googleapis.com/v1/images:annotate?key=${environment.v
isionKey}`;

  detectMood(imageAsBase64: string):
Observable<FaceAnnotationResponse> {
    const body = {
      requests: [
        {
          image: {
            content: this.removeImageBase64Label(imageAsBase64),
          },
          features: [
            {
              type: 'FACE_DETECTION',
              maxResults: 1,
            },
          ],
        },
      ],
    };
    return
this.httpClient.post<FaceAnnotationResponse>(this.geminiApiUrl,
body);
  }

  private removeImageBase64Label(base64Image: string) {
    return base64Image.replace(/^data:image\/[a-z]+;base64,/, '');
  }
}
```

Ensure that the user has added their Vision API key to the environment configuration file for this service to authenticate the API requests.

Building the Angular Component

In `mood-detector.component.ts`, define the component logic. This component handles file uploads, invokes the Vision AI service, and formats the mood likelihood into labels for display.

```
import { Component, inject } from '@angular/core';
import { GoogleGeminiVisionAiService } from
'../../services/gemini-google-ai/gemini-google-vision-ai.service';
import { ImageHelperService } from
'../../helpers/image-helper.service';
import { FaceAnnotation } from
'../../model/face-annotation-response.model';

@Component({
  selector: 'app-mood-detector',
  templateUrl: './mood-detector.component.html',
  styleUrls: ['./mood-detector.component.scss'],
})
export class MoodDetectorComponent {
  imageAsBase64: string = '';
  imageFile: File | null = null;
  result: FaceAnnotation | null = null;

  private geminiService = inject(GoogleGeminiVisionAiService);
  private imageHelper = inject(ImageHelperService);

  onFileChange(event: any) {
    const file = event.target.files[0];
    this.imageHelper.getBase64(file).then((base64Image: string) => {
      this.imageAsBase64 = base64Image;
    });
  }

  detectMood() {

this.geminiService.detectMood(this.imageAsBase64).subscribe((response
) => {
      this.result = response.responses[0].faceAnnotations[0];
    });
  }

  getMoodLabel(likelihood: string): string {
    const moodLabels = {
      VERY_LIKELY: 'High',
      LIKELY: 'Moderate',
```

```
      POSSIBLE: 'Possible',
      UNLIKELY: 'Low',
      VERY_UNLIKELY: 'None'
    };
    return moodLabels[likelihood] || 'Unknown';
  }
}
```

How This Application Can Be Useful

The Mood Detection Application has various practical applications:

1. **User Feedback Analysis**: By detecting users' emotions from images, companies can improve feedback mechanisms by analyzing users' reactions to products or services.
2. **Mental Health Assessment**: With a focus on emotional analysis, this application can assist healthcare professionals in gauging mood patterns, potentially aiding in mental health assessments.
3. **Enhanced User Experiences**: Mood detection enables applications to tailor user experiences based on their emotions, creating more personalized and responsive applications.
4. **Retail and Marketing**: Businesses can utilize emotional data to assess customer satisfaction and make data-driven adjustments to marketing strategies.

Conclusion

This chapter illustrated a practical application of Google's Gemini Vision AI in detecting human emotions within an Angular web application. By following these steps, developers can quickly integrate advanced AI features into their applications, providing unique, data-driven insights into user feedback and enhancing the overall user experience.

6. Getting started with Python

Introduction

Python has become a cornerstone for artificial intelligence and machine learning due to its readability, versatility, and a vast ecosystem of libraries. This chapter will guide you through the essential steps to set up your Python environment and begin your AI journey.

Installing Python

1. **Download the Installer:** Visit the official Python website (https://www.python.org/downloads/) and download the latest stable version for your operating system (Windows, macOS, or Linux).

2. **Run the Installer:** Follow the on-screen instructions to install Python. Ensure you select the option to add Python to your system's PATH environment variable. This will allow you to run Python from the command line anywhere on your system.

Verifying Installation

1. **Open the Command Prompt or Terminal:** On Windows, search for "Command Prompt." On macOS or Linux, open a terminal window.
2. **Type `python --version`:** This command will display the installed Python version. If the installation was successful, you should see the version number.

Using a Python Integrated Development Environment (IDE)

While you can use a simple text editor to write Python code, an IDE provides a more comprehensive environment with features like code highlighting, debugging, and version control. Popular Python IDEs include:

- **PyCharm:** A powerful and full-featured IDE from JetBrains.
- **Visual Studio Code:** A lightweight and customizable code editor with excellent Python support.
- **Jupyter Notebook:** An interactive environment for data science and machine learning.

Basic Python Syntax

Let's explore some fundamental Python concepts:

- **Variables:** Used to store data.

```
// hello.py - Example
x = 5
name = "Alice"
```

- **Data Types:** Python has various data types, including numbers (integers, floats), strings, lists, tuples, dictionaries, and booleans.
- **Operators:** Used to perform operations on data.

```
// Operators
# Arithmetic operators: +, -, *, /, //, %
# Comparison operators: ==, !=, <, >, <=, >=
# Logical operators: and, or, not
```

Control Flow:

- **Conditional statements:** `if`, `else`, `elif`
- **Loops:** `for`, `while`

Functions: Blocks of reusable code.

```
def greet(name):
   print("Hello, " + name + "!")
```

Creating a Python Script

1. **Create a new file:** Use your IDE or a text editor to create a new file with a `.py` extension (e.g., `hello.py`).
2. **Write your code:** Add your Python code to the file.
3. **Run the script:** Open a terminal or command prompt, navigate to the directory where the file is saved, and type `python hello.py`.

```
def greet(name):
     print("Hello, " + name + "!")
     greet("World")
```

This script will print "Hello, World!" to the console.

7. Building APIs with Flask

Introduction to Flask

Flask is one of the most popular Python web frameworks for building APIs and web applications. It is lightweight, flexible, and designed with simplicity in mind, making it a great choice for developers looking to quickly create APIs for their applications. Flask provides the tools to build scalable, RESTful APIs while keeping the codebase manageable.

In this chapter, we'll walk through the steps to create a basic API using Flask. We'll cover how to handle HTTP requests, create endpoints, return JSON responses, and integrate with databases or external services.

Setting Up Flask

Before we start, ensure that Flask is installed in your environment. You can install Flask via **pip**:

```
$ pip install Flask
```

After installation, you can create your first Flask application. The structure of a basic Flask application is simple, with just a few lines of code.

Creating a Basic Flask App

Create a file called `app.py` and add the following code:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to the Flask API!"

if __name__ == '__main__':
    app.run(debug=True)
```

- `Flask(__name__)`: This creates a Flask application object.
- `@app.route('/')`: This decorator defines a route (URL endpoint). In this case, it listens for requests to the root URL `/`.
- `app.run(debug=True)`: Starts the Flask server with debugging enabled.

Run the app by executing `python app.py`. Open your browser and navigate to `http://127.0.0.1:5000/`, where you'll see the message "Welcome to the Flask API!".

Building a RESTful API with Flask

A **REST API** (Representational State Transfer) is a popular architecture for APIs, enabling clients to interact with a server via HTTP requests, such as **GET**, **POST**, **PUT**, and **DELETE**.

Let's start by creating a simple REST API for managing a collection of books.

Creating API Endpoints

Flask makes it easy to define API routes using the `@app.route()` decorator. Let's create endpoints to:

- **GET**: Retrieve a list of books.
- **POST**: Add a new book.
- **GET (by ID)**: Retrieve a single book by its ID.
- **PUT**: Update a book's information.
- **DELETE**: Remove a book.

Start by defining the structure of our data:

```
books = [
   {"id": 1, "title": "The Great Gatsby", "author": "F. Scott"},
   {"id": 2, "title": "To Kill a Mockingbird", "author": "Harper"}
]
```

Now, add the RESTful endpoints:

```
from flask import Flask, jsonify, request

app = Flask(__name__)

books = [
    {"id": 1, "title": "The Great Gatsby", "author": "F. Scott"},
    {"id": 2, "title": "To Kill a Mockingbird", "author": "Harper"}
]

# Get all books
@app.route('/books', methods=['GET'])
def get_books():
    return jsonify(books)

# Get a book by ID
@app.route('/books/<int:id>', methods=['GET'])
def get_book(id):
    book = next((book for book in books if book["id"] == id), None)
    if book:
        return jsonify(book)
    else:
        return jsonify({"error": "Book not found"}), 404

# Add a new book
@app.route('/books', methods=['POST'])
def add_book():
    new_book = request.get_json()
    new_book["id"] = len(books) + 1
    books.append(new_book)
    return jsonify(new_book), 201
```

```
# Update a book
@app.route('/books/<int:id>', methods=['PUT'])
def update_book(id):
    book = next((book for book in books if book["id"] == id), None)
    if book:
        updated_data = request.get_json()
        book.update(updated_data)
        return jsonify(book)
    else:
        return jsonify({"error": "Book not found"}), 404

# Delete a book
@app.route('/books/<int:id>', methods=['DELETE'])
def delete_book(id):
    global books
    books = [book for book in books if book["id"] != id]
    return jsonify({"message": "Book deleted"}), 200

if __name__ == '__main__':
    app.run(debug=True)
```

- **`GET /books`**: This endpoint retrieves the full list of books and returns it as a JSON response.
- **`GET /books/<id>`**: This retrieves a specific book by its `id`. If the book is not found, a `404 Not Found` error is returned.
- **`POST /books`**: This adds a new book. The book data is sent in the body of the request (in JSON format). The new book is appended to the `books` list, and a `201 Created` status code is returned.
- **`PUT /books/<id>`**: This updates an existing book's details by its `id`. If the book is found, it updates its properties.
- **`DELETE /books/<id>`**: This deletes a book by its `id`. The response confirms the deletion.

You can test the endpoints using tools like **Postman** or **cURL** to send HTTP requests and observe the JSON responses.

Handling HTTP Methods

In Flask, routes can be configured to accept specific HTTP methods like **GET**, **POST**, **PUT**, or **DELETE**. By default, routes only respond to `GET`, but you can specify the allowed methods using the `methods` parameter in `@app.route()`.

```
@app.route('/example', methods=['POST', 'PUT'])
def example_method():
    return "This route accepts POST and PUT requests"
```

Flask's `request` object gives access to the request data, such as query parameters, headers, and the request body.

Accessing Request Data

To access the data sent by a client in a **POST** or **PUT** request, Flask provides the `request.get_json()` method:

```
@app.route('/submit', methods=['POST'])
def submit_data():
    data = request.get_json()  # Get JSON data from request body
    return jsonify(data)
```

You can also retrieve query parameters from the URL using `request.args`:

```
@app.route('/search', methods=['GET'])
def search():
    query = request.args.get('query')
    return f'Search results for {query}'
```

Error Handling in Flask APIs

When developing an API, handling errors and returning appropriate status codes and messages is essential for a smooth user experience. Flask provides mechanisms to customize error responses.

Handling 404 Errors

You can define custom error messages for various HTTP error codes. For instance, if a user tries to access a non-existent route, Flask can return a 404 response:

```
@app.errorhandler(404)
def page_not_found(error):
    return jsonify({"error": "Page not found"}), 404
```

Handling Other Errors

Other common errors like **400 Bad Request** or **500 Internal Server Error** can also be handled similarly:

```
@app.errorhandler(400)
def bad_request(error):
    return jsonify({"error": "Bad request"}), 400
```

Structuring a Flask API Project

As your Flask application grows, it's important to structure the project in a way that makes it scalable and maintainable. Here's a typical project structure for a Flask API:

```
/flask_api_project
    /app
        __init__.py
        routes.py
        models.py
    /venv
    app.py
    config.py
    requirements.txt
```

- **app/init.py**: This initializes the Flask app and sets up configurations.
- **app/routes.py**: Contains all the route definitions for the API.
- **app/models.py**: Contains the data models, which may interact with a database.
- **config.py**: Stores configuration details for the app, such as the database URL.
- **requirements.txt**: Lists all dependencies, such as Flask, that need to be installed.

With this structure, your project remains modular, and components can be easily maintained or extended.

8. Getting Started with Gemini and Python

This chapter walks you through the process of setting up and using Google's Gemini API in Python. By the end, you'll be able to generate responses using Google's AI model and retrieve results based on a prompt. We'll break down each part of the code and explain the purpose and function of each step.

Prerequisites

1. **Basic Knowledge of Python**: Familiarity with Python basics, functions, and environment variables.
2. **A Google Cloud Project**: A Google Cloud account and project to enable the Generative AI API.
3. **Gemini API Key**: Access to the Google Gemini API and an API key for authentication.
4. **Python Packages**: The `google-generativeai` and `dotenv` libraries.

Step 1: Setting Up Your Environment

Before diving into the code, we need to make sure we have the necessary Python packages installed. Open your terminal and install the following packages:

```
$ pip install google-generativeai python-dotenv
```

`google-generativeai`: Provides the tools to access Google's Gemini API.

`python-dotenv`: Allows us to load environment variables from a `.env` file, keeping sensitive information (like API keys) secure.

Step 2: Create a `.env` File

For security, it's best to store API keys in environment variables. In your project's root directory, create a file named `.env` and add your Google Gemini API key:

```
GOOGLE_API_KEY=your_google_api_key_here
```

Replace `your_google_api_key_here` with your actual API key.

Step 3: Load Environment Variables with `dotenv`

The `dotenv` package will load the environment variables from the `.env` file, so you can access them in your Python code.

```
from dotenv import load_dotenv
import os
```

- `load_dotenv()`: Loads the environment variables from the `.env` file.
- `os.getenv()`: Retrieves the value of an environment variable by name.

Step 4: Setting Up Your Google API Key

Now that our `.env` file is in place, let's load the API key and set up error handling to ensure the key is loaded correctly.

```
# Load the .env file
load_dotenv()

# Access the GOOGLE_API_KEY environment variable
api_key = os.getenv('GOOGLE_API_KEY')

if api_key is None:
    raise ValueError("GOOGLE_API_KEY is not set in the environment
variables")
```

This code:

1. **Loads** the `.env` file.
2. **Retrieves** the `GOOGLE_API_KEY` environment variable.
3. **Checks** if the key exists. If not, it raises an error to prevent proceeding without a valid key.

Step 5: Configure Google Generative AI

After we have the API key, we can proceed to configure the `google-generativeai` library with our API key.

```
import google.generativeai as genai

# Configure Google AI with the API key
genai.configure(api_key=api_key)
```

The `genai.configure()` function connects the library to Google's servers, using the provided API key to authenticate your requests.

Step 6: Configure the Gemini Model

Gemini offers different model configurations depending on your needs. In this guide, we're using `gemini-1.0-pro-latest`, which is suitable for a range of tasks.

```
# Configure the Gemini model
model = genai.GenerativeModel('gemini-1.0-pro-latest')
```

The `GenerativeModel` class specifies the model you want to use. Here:

- `'gemini-1.0-pro-latest'` refers to the latest professional version of the Gemini model.

Make sure this model name matches the exact format required by the Gemini API in case of updates.

Step 7: Generating Content with the Gemini Model

With the model configured, let's generate some text by providing a prompt. The model will use the context of the prompt to return a relevant response.

```
# Generate a response using the Gemini model
response = model.generate_content("The opposite of hot is")
print(response.text)
```

Explanation:

- **`generate_content()`**: The core function that sends a prompt to Gemini and generates content based on the prompt.
- **`"The opposite of hot is"`**: An example prompt provided to the model.
- **`response.text`**: The response returned by the Gemini model, based on the prompt.
-

```
from dotenv import load_dotenv
import os
import google.generativeai as genai

# Load the .env file
load_dotenv()

# Access the GOOGLE_API_KEY environment variable
api_key = os.getenv('GOOGLE_API_KEY')

if api_key is None:
    raise ValueError("GOOGLE_API_KEY is not set in the environment
variables")

# Configure Google AI
genai.configure(api_key=api_key)

# Configure the Gemini model
model = genai.GenerativeModel('gemini-1.0-pro-latest')
```

```
# Generate a response
response = model.generate_content("The opposite of hot is")
print(response.text)
```

9. Ai Powered REST API using Python and Gemini

In the beginning of your journey with Google's Gemini API, one of the cutting-edge tools for generative AI. In this chapter, we'll cover the steps to set up a REST API using Python and Flask that interfaces with Google Gemini, providing flexible and powerful endpoints for generating AI-based content. This chapter will serve as a comprehensive guide on integrating Google's AI capabilities into your application, making it possible to generate content using text prompts and even a combination of text and images.

Understanding the Basics

Google's Gemini API is part of Google's suite of Generative AI tools, designed to interact with a powerful language model, making it possible to generate human-like text, answer questions, and even respond to image prompts. By leveraging this model, you can build applications that harness the power of Google's AI directly in your projects. However, getting started requires a few foundational steps, from setting up your environment to configuring the model.

This chapter walks through building a Flask-based API to access Gemini's language capabilities. Using Python's `dotenv` library for secure environment configuration and Google's `generativeai` Python SDK, you will have a robust setup to experiment and deploy AI-powered features.

Setting Up the Flask Application

Flask allows us to set up an HTTP server quickly, making it easy to create API routes that accept and return JSON data. Begin by initializing Flask and loading the environment variables securely.

Write the Basic Flask App

Open a new Python file, say `app.py`, and set up the basic Flask structure:

```
from flask import Flask, request, jsonify
from dotenv import load_dotenv
import os

# Initialize Flask app
app = Flask(__name__)

# Load environment variables
load_dotenv()
```

Access and Validate the API Key

To keep our API key secure, we load it using `os.getenv` and validate its presence. Without the API key, the Google Gemini API won't authorize our requests.

```
api_key = os.getenv('GOOGLE_API_KEY')
if api_key is None:
    raise ValueError("GOOGLE_API_KEY is not set in the environment
variables")
```

Configuring Google Gemini API

Now that the API key is loaded, configure the `google.generativeai` library. This SDK connects directly to Google's Gemini API, enabling us to set the model and send prompts.

```
import google.generativeai as genai

# Configure the API with our key
genai.configure(api_key=api_key)

# Set the model
model = genai.GenerativeModel('gemini-1.0-pro-latest')
```

Understanding the Model

The `GenerativeModel` class in the SDK represents different versions of Google's AI models. Here, we are using `gemini-1.0-pro-latest`, a versatile model capable of generating responses to text prompts or image-based prompts, allowing a wide range of content generation tasks.

Defining the API Endpoints

Next, let's define the endpoints that clients will use to interact with the API. We will create two main routes:

1. **Home Route**: This route will serve as a basic check to ensure the server is running.
2. **Generate Content Route**: This route will accept a text prompt and return the model's response.
3. **Generate from Text and Image**: A route that accepts both text and image prompts.

Implementing the Home Route

This route provides a simple response to confirm that the API server is active.

```
@app.route('/', methods=['GET'])
def home():
    return 'Welcome to Gemini Python REST API'
```

Implementing the Text-Based Content Generation Route

This route receives a text prompt in JSON format, sends it to the Gemini API, and returns the generated response.

```
@app.route('/generate', methods=['POST'])
def generate_content():
    data = request.json
    prompt = data.get('prompt')

    if not prompt:
        return jsonify({'error': 'Prompt is required'}), 400

    response = model.generate_content(prompt)
    return jsonify({'text': response.text})
```

Explanation:

- **Input Validation**: If the `prompt` key is missing, the API responds with a `400 Bad Request` error.
- **Content Generation**: The prompt is sent to the Gemini model, and the resulting text is returned as a JSON object.

Implementing the Text and Image-Based Content Generation Route

This endpoint accepts both a text prompt and an image URL, providing more context for generating content.

```
@app.route('/generate-from-image', methods=['POST'])
def generate_content_from_image():
    data = request.json
    prompt = data.get('prompt')
    image_url = data.get('image_url')

    if not prompt:
        return jsonify({'error': 'Prompt is required'}), 400

    response = model.generate_content([prompt, image_url])
    return jsonify({'text': response.text})
```

Explanation:

- Here, `model.generate_content([prompt, image_url])` passes both the prompt and the image URL, enabling the model to consider visual context, enhancing the richness of responses.

Running and Testing the Application

Finally, let's enable Flask to run the application. This setup makes the app accessible on `http://127.0.0.1:5000/`.

```
if __name__ == '__main__':
    app.run(debug=True)
```

Testing Your Endpoints

After starting the server, you can test each endpoint using tools like `curl` or Postman.

1. Home Route:

```
curl http://127.0.0.1:5000/
```

Response:

```
Welcome to Gemini Python Rest Api
```

2. Generate Content Route:

```
curl -X POST http://127.0.0.1:5000/generate -H "Content-Type:
```

```
application/json" -d '{"prompt": "What is AI?"}'
```

3. Generate from Text and Image Route:

```
curl -X POST http://127.0.0.1:5000/generate-from-image -H
"Content-Type: application/json" -d '{"prompt": "Describe this
scene", "image_url": "https://example.com/image.jpg"}'
```

Key Takeaways

In this chapter, we covered:

- Setting up environment variables for secure API key storage.
- Creating a Flask application with endpoints for interacting with the Gemini API.
- Testing endpoints to ensure your API is responsive and reliable.

The foundation set in this chapter prepares you to extend functionality further, offering potential for many AI-powered applications.

10. Sentiment Analysis on Twitter feedback Data from Kaggle using Python

In this chapter, we'll conduct sentiment analysis on a Kaggle dataset containing Twitter feedback. By analyzing the sentiments of tweets, we can understand public opinions or trends about various topics. We'll use Python, with libraries such as **pandas** for data manipulation, **wordcloud** for creating visual representations of word frequencies, and **matplotlib** for plotting. This chapter provides a detailed, step-by-step approach to loading, preprocessing, and analyzing the data, with code examples and explanations for each step.

Requirements and Setup

Before beginning, make sure to install the necessary libraries if they're not already installed:

```
$ pip install pandas matplotlib wordcloud re textblob
```

Step 1: Importing Libraries and Exploring the Dataset

We start by importing the essential libraries and loading the dataset into a pandas DataFrame. This Kaggle dataset consists of tweets, each labeled with a target sentiment: `4` for positive and `0` for negative. Let's examine the structure of the data. The main script loads the dataset, applies cleaning, performs sentiment analysis, and visualizes results.

```
import pandas as pd
from text_cleaner import clean_text
from wordcloud_generator import generate_wordclouds
from visualization import plot_sentiment_distribution
from sentiment_analysis import get_sentiment

# Load and prepare the dataset
data = pd.read_csv('training.csv', encoding='ISO-8859-1',
header=None)
column_names = ['target', 'ids', 'date', 'flag', 'user', 'text']
data.columns = column_names

# Apply the clean_text function to clean tweet text
data['cleaned_text'] = data['text'].apply(clean_text)

# Apply sentiment analysis function to cleaned text
data['sentiment'] = data['cleaned_text'].apply(get_sentiment)

# Display cleaned text and sentiment columns
print(data[['text', 'cleaned_text', 'sentiment']].head())

# Generate word clouds
generate_wordclouds(data)

# Plot sentiment distribution
plot_sentiment_distribution(data)
```

Code Explanation:

- We load the dataset `training.csv` using pandas.
- The `header=None` parameter is used because the dataset doesn't include column headers.
- We assign column names to identify each column: `target`, `ids`, `date`, `flag`, `user`, and `text`.
- The `head()`, `info()`, and `describe()` functions allow us to view the data structure and summary statistics.
- `data = pd.read_csv(...)`: Loads the dataset with ISO-8859-1 encoding.

- `column_names = [...]`: Sets column names to make data easier to work with.
- `data['cleaned_text'] = data['text'].apply(clean_text)`: Cleans the tweet text in the `text` column by applying `clean_text` to each row.
- `data['sentiment'] = data['cleaned_text'].apply(get_sentiment)`: Applies `get_sentiment` to each cleaned tweet, adding a `sentiment` column.
- `print(...)`: Displays the first few rows of the original tweet, cleaned text, and sentiment columns to check the results.
- `generate_wordclouds(data)`: Calls `generate_wordclouds` to visualize the most common words.
- `plot_sentiment_distribution(data)`: Plots the sentiment distribution for visual analysis.

Step 2: Visualizing the Dataset with Word Clouds

To gain insights, we'll create word clouds that represent the most frequent words in both positive and negative tweets. Word clouds are helpful for visualizing the most common words associated with each sentiment.

Word Cloud Code Implementation

```
import matplotlib.pyplot as plt
from wordcloud import WordCloud

# Function to generate word clouds from positive and negative tweets
data
def generate_wordclouds(data):
    # Separate positive and negative tweets based on the 'target'
column
    positive_tweets = data[data['target'] == 4]['text']
    negative_tweets = data[data['target'] == 0]['text']

    # Sample some positive and negative tweets to create word clouds
    sample_positive_text = " ".join(text for text in
positive_tweets.sample(frac=0.1, random_state=23))
    sample_negative_text = " ".join(text for text in
negative_tweets.sample(frac=0.1, random_state=23))

    # Generate word cloud images for both positive and negative
sentiments
    wordcloud_positive = WordCloud(width=800, height=400,
max_words=200,
background_color="white").generate(sample_positive_text)
    wordcloud_negative = WordCloud(width=800, height=400,
max_words=200,
```

```
background_color="white").generate(sample_negative_text)

    # Display the generated image using matplotlib
    plt.figure(figsize=(15, 7.5))

    # Positive word cloud
    plt.subplot(1, 2, 1)
    plt.imshow(wordcloud_positive, interpolation='bilinear')
    plt.title('Positive Tweets Word Cloud')
    plt.axis("off")

    # Negative word cloud
    plt.subplot(1, 2, 2)
    plt.imshow(wordcloud_negative, interpolation='bilinear')
    plt.title('Negative Tweets Word Cloud')
    plt.axis("off")

    plt.show()

# Generate word clouds
generate_wordclouds(data)
```

Code Explanation:

- We define a function `generate_wordclouds()` that takes in the `data` DataFrame.
- We split the data into positive and negative tweets based on the `target` column values.
- Using the `sample()` function, we select a fraction (10%) of positive and negative tweets for visualization.
- We use the `WordCloud` library to generate separate word clouds for positive and negative tweets.
- The generated word clouds are displayed side-by-side using `matplotlib`.

Step 3: Further Data Cleaning and Preprocessing

For sentiment analysis, we'll clean the tweets to remove noise such as URLs, mentions, hashtags, special characters, and extra whitespace. This will help in getting more accurate sentiment analysis results.

This file defines a function called `clean_text` that processes and cleans tweet text by removing unnecessary elements like URLs, mentions, hashtags, and special characters.

```
import re
```

```
# Function to clean tweet text
def clean_text(text):
    # Remove URLs
    text = re.sub(r'http\S+|www\S+', '', text)
    # Remove mentions and hashtags
    text = re.sub(r'@\w+|#\w+', '', text)
    # Remove special characters and numbers
    text = re.sub(r'[^A-Za-z\s]', '', text)
    # Convert to lowercase
    text = text.lower()
    return text
```

Explanation:

- `import re`: Imports the `re` (regular expressions) module to enable text matching and manipulation.
- `clean_text` function:
    - `re.sub(r'http\S+|www\S+', '', text)`: Removes any URLs in the text that start with "http" or "www".
    - `re.sub(r'@\w+|#\w+', '', text)`: Removes Twitter mentions (e.g., `@username`) and hashtags (e.g., `#hashtag`).
    - `re.sub(r'[^A-Za-z\s]', '', text)`: Removes special characters and numbers, keeping only letters and spaces.
    - `text.lower()`: Converts all letters to lowercase to ensure consistency.
- The cleaned text is then returned.

Step 4: Sentiment Analysis on Cleaned Data

With cleaned tweets, we can now proceed to sentiment analysis using libraries like `TextBlob` or `VADER`. Here, we use `TextBlob` to calculate the polarity of each tweet. This function analyzes the sentiment of text data using `TextBlob` and categorizes it as positive, neutral, or negative.

```
from textblob import TextBlob

def get_sentiment(text):
    analysis = TextBlob(text)
    if analysis.sentiment.polarity > 0:
        return 'positive'
    elif analysis.sentiment.polarity == 0:
        return 'neutral'
```

```
    else:
        return 'negative'
```

Explanation:

- `analysis = TextBlob(text)`: Creates a `TextBlob` object to calculate the polarity of the text.
- `analysis.sentiment.polarity`: Returns a value between -1 (negative) and +1 (positive).
    - `if polarity > 0`: Returns `'positive'` if the polarity is positive.
    - `elif polarity == 0`: Returns `'neutral'` for neutral polarity.
    - `else`: Returns `'negative'` if the polarity is negative.

Step 5: Visualizing Sentiment Distribution

Finally, we'll visualize the sentiment distribution to get an overall view of the proportions of positive, negative, and neutral tweets. This function takes in a dataset containing a "sentiment" column and plots a bar chart showing the distribution of sentiments.

```
import matplotlib.pyplot as plt

def plot_sentiment_distribution(data):
    """
    Plots the sentiment distribution as a bar chart.

    Parameters:
    - data (pd.DataFrame): DataFrame containing a 'sentiment' column
with sentiment labels
    """
    sentiment_counts = data['sentiment'].value_counts()
    plt.figure(figsize=(8, 5))
    sentiment_counts.plot(kind='bar', color=['green', 'gray', 'red'])
    plt.title('Sentiment Analysis of Tweets')
    plt.xlabel('Sentiment')
    plt.ylabel('Number of Tweets')
    plt.xticks(rotation=0)
    plt.show()
```

- **Explanation**:
    - `sentiment_counts = data['sentiment'].value_counts()`: Counts the occurrences of each sentiment label (e.g., positive, neutral, negative).
    - `plt.figure(figsize=(8, 5))`: Sets the size of the plot.

- `sentiment_counts.plot(...)`: Creates a bar chart with colors representing different sentiments.
- `plt.title`, `plt.xlabel`, and `plt.ylabel`: Add a title and labels to the chart.
- `plt.xticks(rotation=0)`: Ensures that sentiment labels on the x-axis are horizontal.

Workflow

1. **Data Loading and Preparation**: `main.py` loads and prepares the dataset.
2. **Data Cleaning**: `clean_text` removes URLs, mentions, and special characters.
3. **Sentiment Analysis**: `get_sentiment` determines tweet sentiment.
4. **Visualization**:
   - `generate_wordclouds`: Creates word clouds for visualizing frequent words.
   - `plot_sentiment_distribution`: Plots a bar chart of sentiment frequencies.

This modular structure allows each function to be reused or tested independently.

Summary

In this chapter, we:

1. Loaded and explored a Kaggle Twitter dataset.
2. Preprocessed the tweets to clean unnecessary elements.
3. Performed sentiment analysis using `TextBlob`.
4. Visualized insights using word clouds and a sentiment distribution bar chart.

By following these steps, you've learned how to conduct sentiment analysis on Twitter feedback data, which can be useful for understanding public opinion and tracking trends on social media. This approach can be extended to analyze feedback in various domains, such as product reviews, news articles, or customer surveys.

11. Connecting AWS Bedrock with Angular using AWS-SDK to List Available Models

To connect **AWS Bedrock** with an **Angular application** using the AWS-SDK for JavaScript, this guide covers a step-by-step approach to setting up the SDK, configuring credentials securely, creating a Bedrock service, and displaying data from Bedrock. Here's a more detailed breakdown.

Prerequisites

Ensure you have:

1. An **AWS account** with permissions to access AWS Bedrock.
2. AWS **IAM credentials** with appropriate Bedrock permissions.
3. **Node.js and npm** installed for package management.
4. **Angular CLI** installed to create and manage your Angular project.

Install AWS SDK v3 for JavaScript

AWS SDK v3 offers modular imports, allowing you to include only the services you need, improving efficiency in web apps.

Run the following command to add the AWS Bedrock client to your project:

```
$ npm install @aws-sdk/client-bedrock
```

Note: To install other parts of the SDK, use the corresponding package, such as `@aws-sdk/client-s3` for S3.

Add AWS Configuration in `environment.ts`

In Angular, `environment.ts` is used for configuration settings specific to environments (e.g., development, production). Add AWS region and optionally the Bedrock service endpoint.

```
export const environment = {
  production: false,
  aws: {
    region: 'us-west-2', // Replace with your AWS region
    bedrockEndpoint: 'https://bedrock.us-west-2.amazonaws.com' //
Optional: Use if you need a custom endpoint
  }
};
```

Set Up AWS Credentials

A. Use IAM Roles (Recommended)

If deploying on AWS infrastructure (e.g., EC2 or Lambda), use IAM roles instead of access keys.

B. Use Access Keys for Local Development

To use access keys locally, you can create a `.env` file for security or use a credentials provider package to load them.

C. Use AWS Cognito for Authentication

To integrate with AWS Cognito, install and configure the credentials provider:

```
$ npm install @aws-sdk/credential-provider-cognito-identity
```

Then, update the service to use this provider if Cognito is part of your setup.

Create an Angular Service to Interact with AWS Bedrock

This service will handle all interactions with the AWS Bedrock client.

1. Create a New Service

Generate a service in Angular to encapsulate AWS Bedrock functionality:

```
$ ng generate service services/bedrock
```

2. Configure the Service

Set up the `BedrockService` to initialize the Bedrock client with configuration from `environment.ts`. Here, it uses basic authentication via access keys for simplicity.

```
import { Injectable } from '@angular/core';
import { BedrockClient, ListModelsCommand } from
'@aws-sdk/client-bedrock';
import { from, Observable } from 'rxjs';
import { environment } from '../environments/environment';

@Injectable({
  providedIn: 'root',
})
export class BedrockService {
  private client: BedrockClient;

  constructor() {
    this.client = new BedrockClient({
      region: environment.aws.region,
      credentials: {
        accessKeyId: 'YOUR_ACCESS_KEY_ID', // Replace with your
access key ID
        secretAccessKey: 'YOUR_SECRET_ACCESS_KEY', // Replace with
your secret access key
      },
      endpoint: environment.aws.bedrockEndpoint
    });
```

```
  }

  // Fetch a list of available models
  listModels(): Observable<any> {
    const command = new ListModelsCommand({});
    return from(this.client.send(command));
  }
}
```

Using the Service in an Angular Component

Now, we'll create a component that utilizes the `BedrockService` to call `listModels()` and display data.

1. Generate the Component

```
$ ng generate component components/bedrock
```

2. Set Up the Component to Fetch and Display Data

Inject `BedrockService` into the component and subscribe to the `listModels()` observable to fetch data on initialization.

```
import { Component, OnInit } from '@angular/core';
import { BedrockService } from '../services/bedrock.service';

@Component({
  selector: 'app-bedrock',
  templateUrl: './bedrock.component.html',
  styleUrls: ['./bedrock.component.css']
})
export class BedrockComponent implements OnInit {
  models: any[] = [];
  loading = true;
  error: string | null = null;

  constructor(private bedrockService: BedrockService) {}

  ngOnInit(): void {
    this.bedrockService.listModels().subscribe({
      next: (data) => {
        this.models = data.Models;
        this.loading = false;
      },
```

```
      error: (err) => {
        this.error = 'Failed to load models';
        console.error('Error fetching models:', err);
        this.loading = false;
      }
    });
  }
}
```

Create the Component Template to Display Models

In `bedrock.component.html`, add logic to display loading indicators, errors, and the list of models from AWS Bedrock.

```
<div *ngIf="loading">Loading models...</div>
<div *ngIf="error">{{ error }}</div>

<div *ngIf="models.length">
  <h2>Available Models:</h2>
  <ul>
    <li *ngFor="let model of models">
      <strong>{{ model.ModelName }}</strong>
      <p>ID: {{ model.ModelId }}</p>
      <p>Description: {{ model.Description || 'No description
available' }}</p>
    </li>
  </ul>
</div>

<div *ngIf="!loading && !models.length && !error">
  <p>No models available.</p>
</div>
```

Add CSS Styling (Optional)

Customize the component with CSS for a polished UI:

```
/* bedrock.component.css */

h2 {
  color: #333;
  font-size: 24px;
```

```
}

ul {
  list-style: none;
  padding: 0;
}

li {
  margin: 10px 0;
  padding: 10px;
  border: 1px solid #ddd;
  border-radius: 4px;
}

li p {
  margin: 5px 0;
}
```

Run the Angular Application

Launch the app and verify the integration:

```
$ ng serve
```

Visit `http://localhost:4200` to view the AWS Bedrock models fetched from your API.

Best Practices

1. **Credential Management**: Avoid hardcoding credentials; consider using environment variables or IAM roles.
2. **Error Handling**: Implement comprehensive error handling to manage API request failures gracefully.
3. **Loading Indicators**: Improve UX with loading indicators, especially for long API calls.

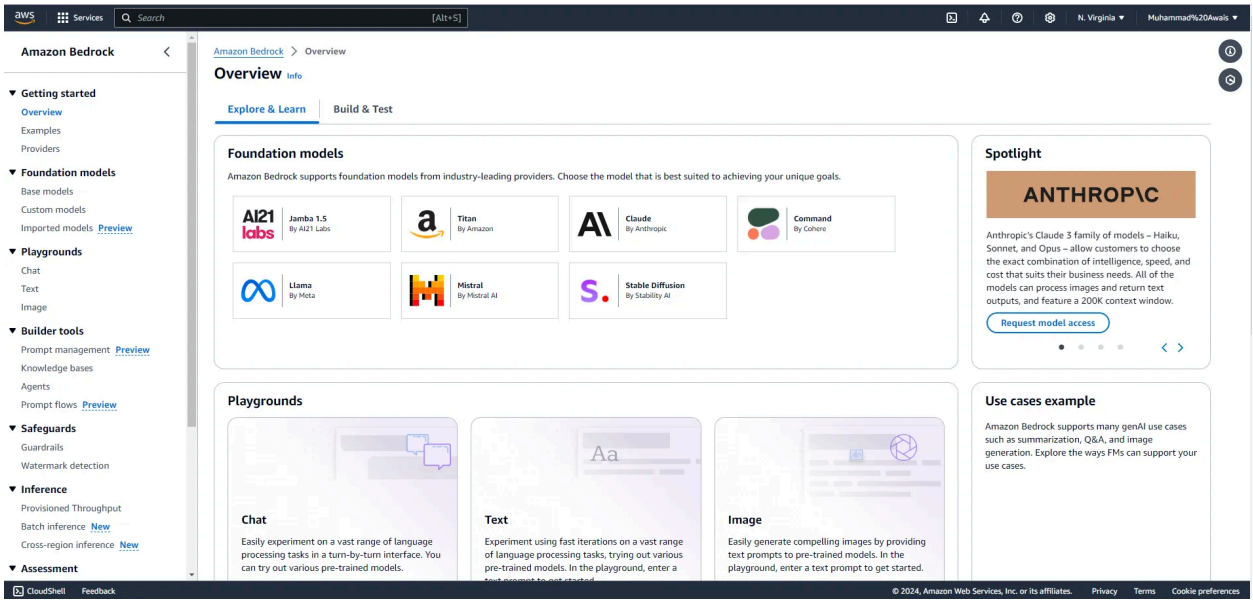
This guide integrates AWS Bedrock with an Angular app, covering SDK setup, service creation, and UI design for an enhanced experience when interacting with Bedrock models. This approach provides a foundation you can build upon for additional AWS services and complex workflows.

12. Building Ai Powered Apps with AWS Bedrock with Angular using AWS-SDK

In this comprehensive guide, you'll learn how to integrate Amazon Bedrock, AWS's powerful generative AI service, with your Angular applications using the AWS-SDK. Discover how to set up and configure AWS credentials, manage environment variables, and implement Bedrock's AI capabilities seamlessly into your frontend code. Whether you're creating AI-powered text generation or other innovative applications, this guide will walk you through every essential step, from initializing the SDK to invoking Bedrock models, ensuring smooth integration and dynamic functionality. Perfect for developers looking to leverage AI with Angular.

Overview — Bedrock:

AWS Bedrock is a fully managed service that makes it easy to build and deploy foundation models into your applications. It provides a simple API and managed infrastructure, allowing developers to focus on creating innovative applications without the need for deep machine learning expertise.



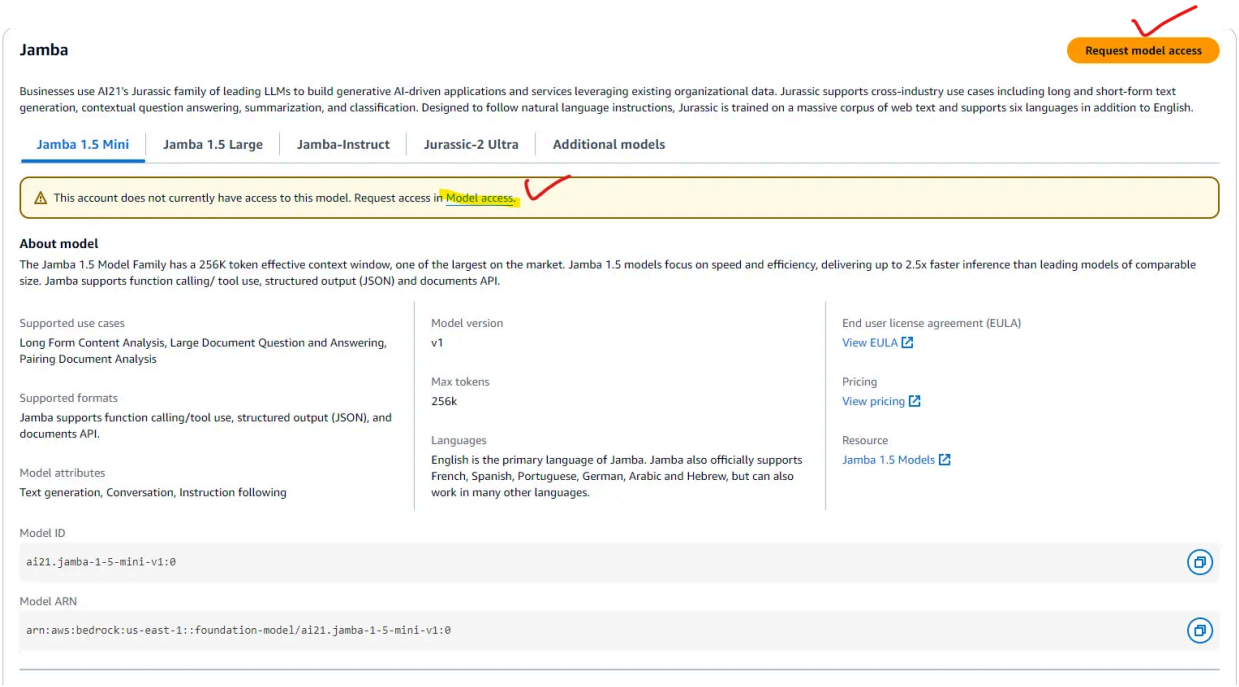
Key Features

- Diverse Model Library: Bedrock offers a wide range of foundation models, including text generation, code generation, and image generation models. This enables developers to choose the best model for their specific use case.
- Customizable Models: You can fine-tune existing models or train your own models on Bedrock to tailor them to your specific needs. This allows for greater flexibility and control over the model's behavior.

- Easy Integration: Bedrock integrates seamlessly with other AWS services, such as Amazon SageMaker and Amazon Lambda, making it easy to incorporate foundation models into your existing applications.
- Managed Infrastructure: AWS handles the underlying infrastructure, including hardware, software, and scaling, so you can focus on building your applications.
- Security and Compliance: Bedrock is built with security and compliance best practices in mind, ensuring that your data is protected and that you meet regulatory requirements.

Requesting Model Access in AWS Bedrock

AWS Bedrock offers a wide range of foundation models, each with its own capabilities and access requirements. To use a specific model in your applications, you'll typically need to request access to it.



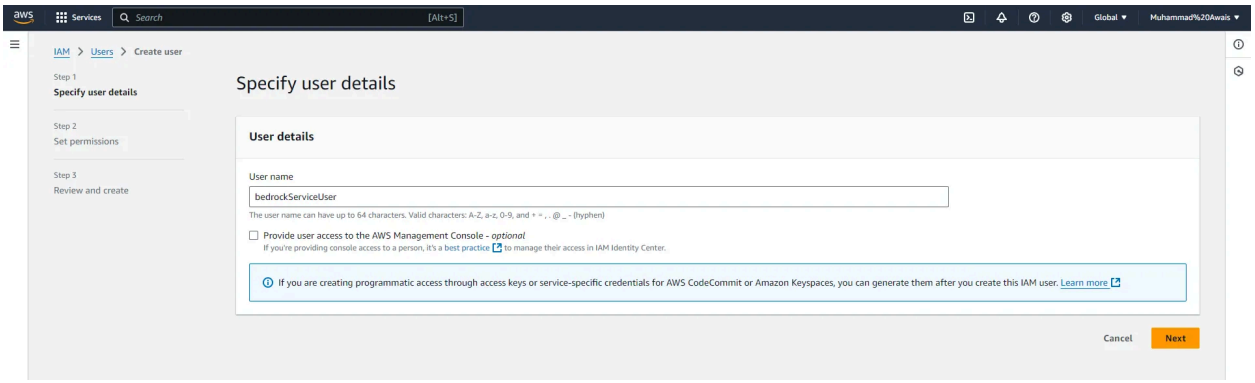
How to Request Model Access

1. **Identify the Model:** Determine which Bedrock model you want to use based on your specific needs and requirements.
2. **Check Availability:** Verify if the model is publicly available or requires special permission. Some models might be limited to certain regions or require specific use cases.
3. **Submit a Request:** If the model requires special permission, follow the instructions provided by AWS to submit a request. This might involve filling out a form, providing additional information, or contacting AWS support directly.

4. **Await Approval:** Once you submit your request, AWS will review it to ensure you meet the necessary criteria. The approval process may take some time.
5. **Start Using the Model**: Once your request is approved, you'll be granted access to the model and can start using it in your applications.

Create IAM User:

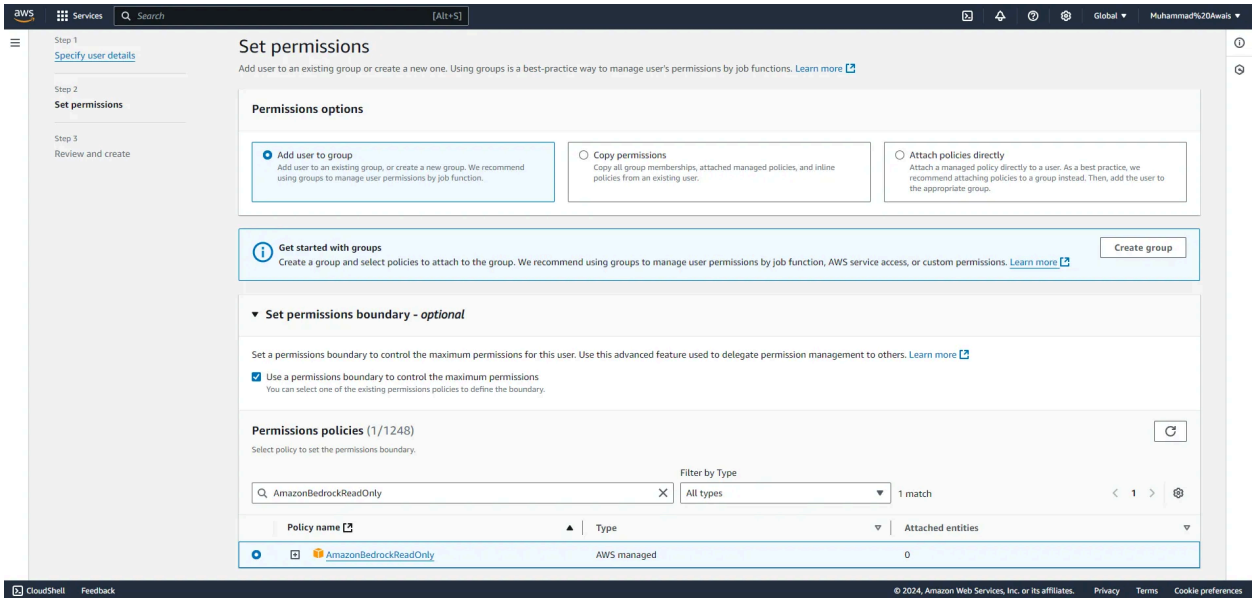
In the context of Bedrock, creating IAM users is essential for managing access to the foundation models and other resources within the Bedrock service. By assigning appropriate permissions to your users, you can ensure that only authorized individuals can interact with and use these resources.



When you create an IAM user, you can assign specific permissions to that user, allowing them to perform certain actions within your AWS environment. This helps to protect your AWS resources and prevent unauthorized access.

Adding User Permissions in Bedrock

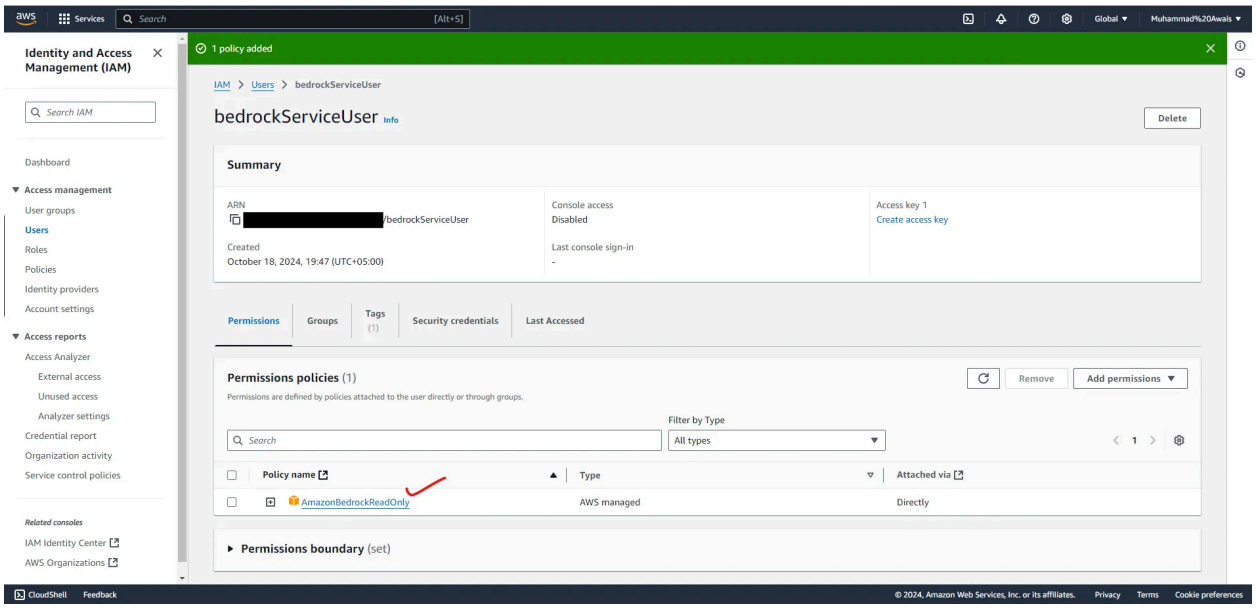
When working with AWS Bedrock, user permissions determine which actions a user can perform within the service. This includes accessing models, creating and managing jobs, and interacting with other Bedrock resources using AmazonBedrockReadOnly permission policy.



Key Permissions

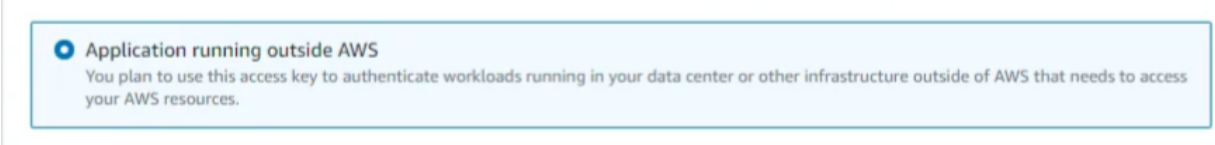
Here are some common permissions you might consider assigning to users in Bedrock:

- **bedrock:ListModels:** Allows users to list available Bedrock models.
- **bedrock:InvokeModel:** Grants users permission to invoke a specific Bedrock model.
- **bedrock:CreateJob:** Enables users to create new jobs for model inference.
- **bedrock:ListJobs:** Allows users to list existing jobs.
- **bedrock:GetJob:** Grants users permission to retrieve information about a specific job.
- **bedrock:StopJob:** Enables users to cancel a running job.



Adding Access Keys for Bedrock Resources

Access keys are a set of credentials (access key ID and secret access key) that allow your applications and services to interact with AWS services, including Bedrock. They provide a way for your code to authenticate and authorize requests to AWS.

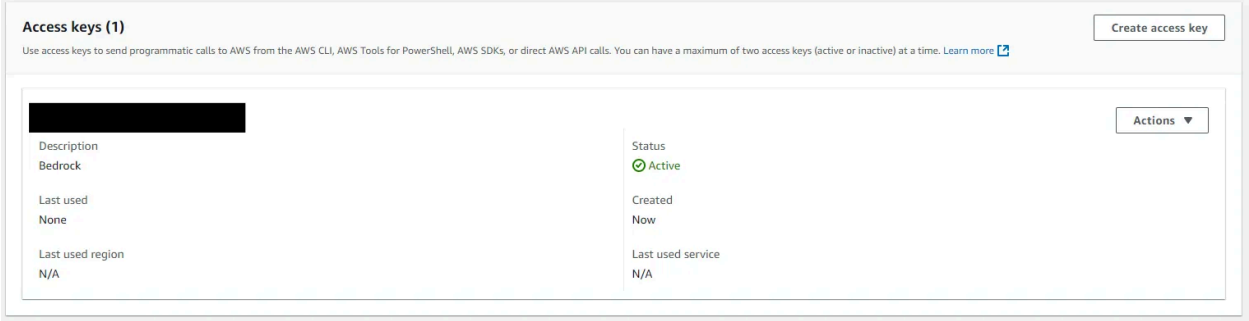


Adding Access Keys

1. **Create an IAM User:** If you haven't already, create an IAM user in your AWS account. This user will be associated with the access keys.
2. **Assign Permissions:** Attach the necessary IAM policies to the user to grant them the required permissions to access and use Bedrock resources.

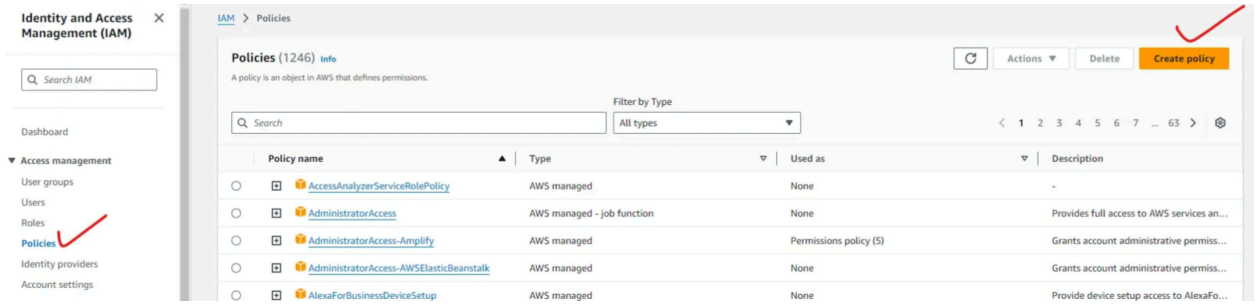
Create Access Keys:

- Log in to the AWS Management Console as the IAM user.
- Navigate to the IAM console and select "Users".
- Choose the user you created.
- In the "Security credentials" tab, click "Create access keys".
- A dialog box will appear asking you to confirm the creation of access keys. Click "Create access keys".
- The access key ID and secret access key will be displayed. Immediately download or copy these credentials and store them securely. Once you close the dialog, you won't be able to view the secret access key again.



Creating an IAM Policy for InvokeModel in Bedrock

The bedrock:InvokeModel action allows users to invoke a specified Bedrock model. This is typically used to generate text, translate languages, or perform other tasks based on the model's capabilities.

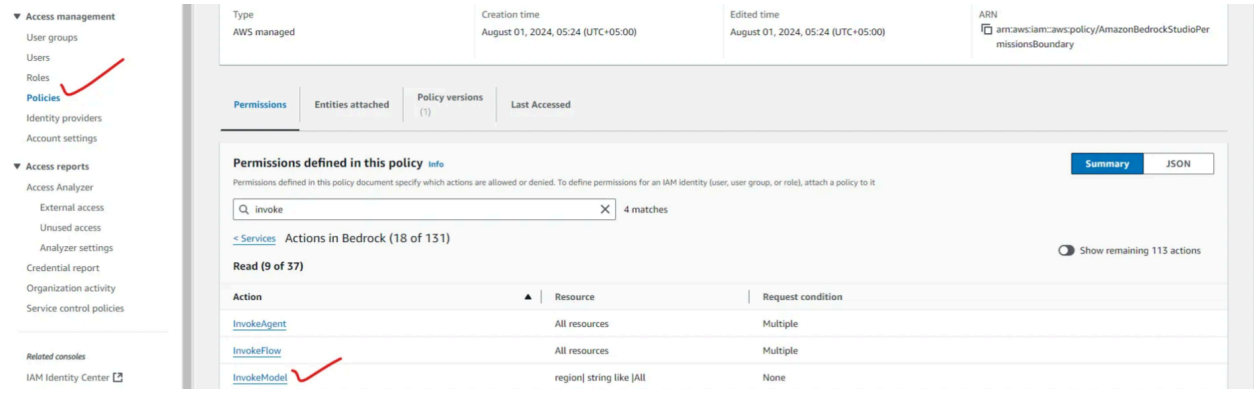


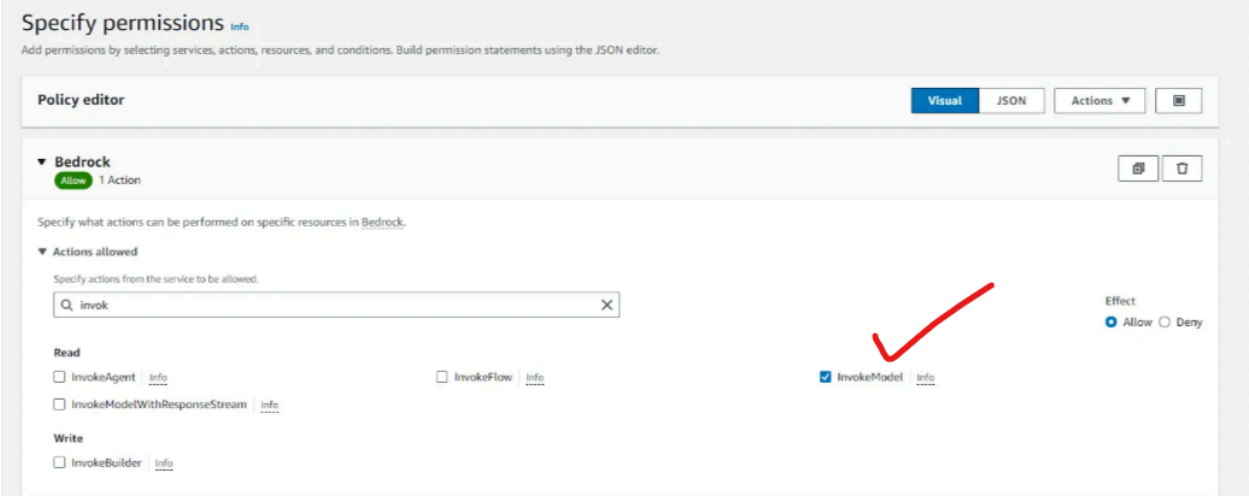
Creating the IAM Policy

Here's a basic IAM policy that grants a user permission to invoke a specific Bedrock model:

```
{
  "Version": "2024-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "bedrock:InvokeModel",
      "Resource":
"arn:aws:bedrock:us-east-1:YOUR_ACCOUNT_ID:model/YOUR_MODEL_ID"
    }
  ]
}
```

We will be using InvokeModelCommand which will not work if this permission is not allowed in policies.

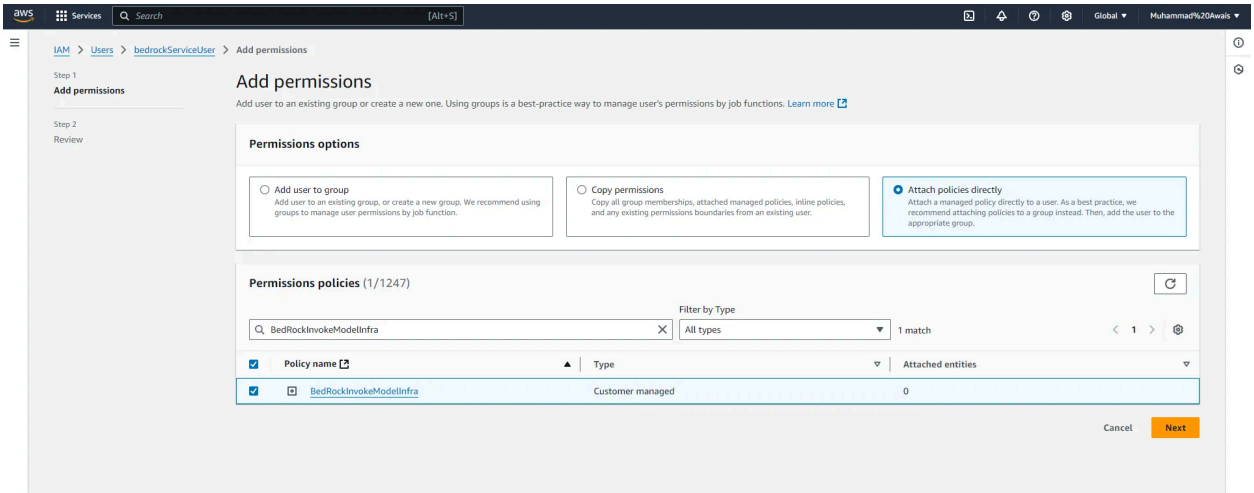




Attaching the Policy to IAM

Once you've created the policy, you can attach it to an IAM user or role. This will grant the user or role the specified permissions.

By following these steps and considering the additional factors, you can effectively create and manage InvokeModel permissions for your Bedrock users and ensure that they have the appropriate access to the models they need.



Angular Integration with Bedrock using AWS-SDK

In this section, we'll integrate AWS Bedrock into an Angular application (version 18.2) using the AWS-SDK. This will enable you to leverage generative AI models within your Angular app, providing seamless interaction with AWS Bedrock's powerful AI capabilities.

Here's the Angular integration part:

1. Install the AWS-SDK in Your Angular App: First, navigate to your Angular project directory and install the AWS SDK package:

```
$ npm install @aws-sdk/client-bedrock-runtime
```

2. Set Up AWS Credentials: Ensure your AWS credentials are set up correctly. You can use AWS IAM roles, AWS CLI, or environment variables. For development, you can configure AWS credentials using the AWS CLI:

```
// environment.ts

export const environment = {
    production: false,
    accessKeyId: 'YOUR_ACCESS_KEY_ID',
    secretAccessKey: 'YOUR_SECRET_KEY_ID'
  };
```

3. Create an AWS Bedrock Service: Create a new Angular service that will handle the interaction with AWS Bedrock.

```
$ ng g service aws-bedrock/aws-bedrock
```

```
// aws-bedrock.service.ts

import { Injectable } from '@angular/core';
import { BedrockRuntimeClient, InvokeModelCommand } from
'@aws-sdk/client-bedrock-runtime';
import { environment } from '../../../environment/environment';

@Injectable({
  providedIn: 'root'
})
export class AwsBedrockService {

  modelId: string = "ai21.jamba-1-5-large-v1:0";

  config: any = {
    region: 'us-east-1',
    credentials: {
```

```
      accessKeyId: environment.accessKeyId,
      secretAccessKey: environment.secretAccessKey
    }
  };

  client: any = new BedrockRuntimeClient(this.config);

  constructor() { }

  async syncBedRock(prompt: string) {
    const params: any = {
      body: prompt,
      modelId: this.modelId,
      maxTokens: 500,
      temperature: 0.5,
    };

    this.client.send(new InvokeModelCommand(params))
      .then((data: any) => {
        console.log('Response:', data);
      })
      .catch((err: any) => {
        console.error('Error:', err);
      });
  }
}
```

This Angular service, named AwsBedrockService, facilitates interaction with AWS Bedrock, a service that enables developers to use foundation models for various tasks such as text generation, translation, and summarization.

Key Components and Functionality:

1. Imports & 3rd Parties from (aws-sdk):

- Injectable: Decorator from Angular that injects the service into other components or services.
- BedrockRuntimeClient, InvokeModelCommand: Classes from the @aws-sdk/client-bedrock-runtime library, providing the necessary methods for interacting with Bedrock.
- environment: Imports the environment configuration file, typically used to store sensitive credentials like access keys.

2. Model ID and Configuration:

- modelId: Specifies the ID of the Bedrock model you want to use. In this case, it's set to "ai21.jamba-1-5-large-v1:0".
- config: Contains the configuration settings for the Bedrock client, including the region and credentials.

3. Bedrock Client:

- client: Creates an instance of the BedrockRuntimeClient class, which is used to interact with the Bedrock service.

Overall Functionality:

When you call the syncBedRock method from another component or service, it sends the provided prompt to the specified Bedrock model. The model processes the prompt and generates a response, which is then logged to the console. You can modify the maxTokens and temperature parameters to control the output of the generated text.

4. Integration with Angular Components:

To use this service in your Angular components, inject it and call the syncBedRock method with the desired prompt. For example:

```
// app.component.ts

import { HttpClient } from '@angular/common/http';
import { Component, inject, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { AwsBedrockService } from
'./services/aws-bedrock/aws-bedrock.service';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent implements OnInit {
  title = 'aws-bedrock-angular';

  http = inject(HttpClient);
  bedrock = inject(AwsBedrockService);

  ngOnInit(): void {
    this.onCheckBedRock()
```

```
  }

  onCheckBedRock() {
    this.bedrock.syncBedRock("hey, how is your day?");
  }
}
```

This code will call the syncBedRock method when a button is clicked or another event triggers it, and the generated text will be displayed in the component's template.

Conclusion:

In this guide, we successfully integrated Amazon Bedrock into an Angular v18.2 application using AWS-SDK. We walked through the process of setting up AWS credentials, installing the necessary SDK, and creating an Angular service to interact with Bedrock’s AI models. By following these steps, your Angular app is now equipped to harness the power of generative AI, providing dynamic content generation and other AI-driven functionalities.

This integration opens up a world of possibilities for building smarter, AI-enhanced applications. You can further explore how to fine-tune the models, enhance performance, and expand the AI capabilities to suit your specific project needs. With AWS Bedrock and Angular working seamlessly together, you’re well-positioned to innovate and deliver more engaging user experiences.

Summary

"AI-Powered App Development: Leveraging Angular, Python, and Google AI" by Muhammad Awais and Sonu Kapoor is a comprehensive guide for developers seeking to integrate AI capabilities into web applications. This book offers a structured journey, beginning with fundamental Angular setup and component communication, and progressing to advanced AI integrations. With a hands-on approach, readers will gain expertise in both Angular and Python, learning how to build intelligent, responsive applications by combining Google AI's Gemini API and AWS Bedrock's AI services. This book is designed to suit both beginners and experienced developers, offering clear explanations of core concepts, step-by-step instructions, and practical examples.

The book is divided into three main sections. The first section covers Angular basics, such as component creation, services, dependency injection, and inter-component communication. This foundation enables readers to build efficient, scalable web applications that can later be enhanced with AI capabilities. The middle section introduces a variety of AI-based applications, including a chatbot for conversational AI, a sentiment analyzer, a skill quiz generator, a plant identifier using vision recognition, a mood detection app, and an AI-based health report interpreter. Each project includes step-by-step recipes that take readers from setup to a fully functioning, interactive AI application. The final section delves into backend development with Python and Flask, explaining how to set up APIs and securely connect to Google AI and AWS Bedrock services. This includes deploying IAM policies, feeding data into the sentiment analyzer from external sources like Kaggle, and visualizing results with word clouds.

By the end, readers will be equipped to develop full-stack AI applications that harness Angular, Flask, and powerful AI models from Google and AWS. This book leaves developers with a strong foundation in generative AI, ready to transform ideas into next-generation web applications that engage users in new, meaningful ways.

References

https://ai.google/build
https://aws.amazon.com/bedrock
https://angular.dev
https://flask.palletsprojects.com/en/stable/
https://www.python.org/psf-landing/

Leveraging Angular, Python, and AI

GenAI

Step by Step Guide

In "Ai Powered App Development: Leveraging Angular, Python, and Google AI," authors Muhammad Awais and Sonu Kapoor guides you through the journey of creating intelligent, cutting-edge applications by combining the best of modern web technologies and artificial intelligence. This comprehensive guide offers developers a step-by-step approach to integrating AI capabilities into real-world applications using Angular, Google Ai and Python, alongside the powerful Google AI ecosystem.

In this book, you'll discover how to:

- Integrate Google AI's Gemini API to seamlessly connect your applications with advanced machine learning models, enriching user experience through powerful AI capabilities.
- Master AWS Bedrock for Generative AI and utilize Amazon's suite of AI tools to design applications with state-of-the-art AI functionality such as language generation, recommendation systems, and predictive insights.
- Build step-by-step AI-powered apps, from chatbots and personalized content engines to image recognition and real-time language processing, that bridge user needs with AI solutions.
- Follow practical recipes that offer a hands-on approach to building smaller, focused AI modules that can be combined into larger applications, making each step achievable and immediately applicable.

Whether you're a seasoned developer or just starting with AI, this book will provide the practical insights and technical skills needed to transform your ideas into next-generation, AI-driven applications.

<u>About the Author:</u>

Muhammad Awais is an internationally recognized Lead Software Engineer, tech speaker, author, mentor, and open-source contributor with over 9+ years of experience. Renowned for his expertise in modern technologies, his contributions reflect a deep-rooted passion for Angular, his first love in the tech world.

Sonu Kapoor is a distinguished web technology expert specializing in Angular and JavaScript. His commitment to innovation and community support has earned recognition as an Angular Collaborator, Microsoft MVP, and Google Developer Expert for Angular.

ISBN-13

979-8346639220