RASA Chatbot

rasa run --enable-api (enable api)

Typical Service Methods:

```
import { Component, ViewChild } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { MessageService } from '../service/api.service';

export interface Message {
  type: string;
  message: string;
}

@Component({
  selector: 'app-chat-support',
  templateUrl: './chat-support.component.html',
  styleUrls: ['./chat-support.component.scss'],
})
export class ChatSupportComponent {
  isOpen = false; // Variable to toggle chat interface visibility
  loading = false; // Indicates if a message is being sent/loading
  messages: Message[] = []; // Array to store chat messages
  chatForm = new FormGroup({
    message: new FormControl('', [Validators.required]), // Input field for user message
  });
```

```
  @ViewChild('scrollMe') private myScrollContainer: any; // Reference to chat container element

  constructor(private messageService: MessageService) {}

  // Function to open or close the chat interface
  openSupportPopup() {
    this.isOpen = !this.isOpen;
  }

  // Function to send a user message to the Rasa chatbot
  sendMessage() {
    const sentMessage = this.chatForm.value.message!;
    this.loading = true;

    // Add user message to the messages array
    this.messages.push({
      type: 'user',
      message: sentMessage,
    });

    this.chatForm.reset(); // Clear the input field
    this.scrollToBottom(); // Scroll to the bottom of the chat container

    // Send user message to Rasa chatbot backend via messageService
    this.messageService.sendMessage(sentMessage).subscribe((response: any) => {
      for (const obj of response) {
        let value;
```

```
      // Check if the response has a 'text' property
      if (obj.hasOwnProperty('text')) {
        value = obj['text'];
        this.pushMessage(value); // Add the text response to the messages array
      }

      // Check if the response has an 'image' property
      if (obj.hasOwnProperty('image')) {
        value = obj['image'];
        this.pushMessage(value); // Add the image response to the messages array
      }
    }
  });
}

// Function to add a client message (bot response) to the messages array
pushMessage(message: string) {
  this.messages.push({
    type: 'client',
    message: message,
  });
  this.scrollToBottom(); // Scroll to the bottom of the chat container
}

// Function to scroll to the bottom of the chat container
scrollToBottom() {
  setTimeout(() => {
    try {
      // Set the scrollTop property to the maximum scroll height
```

```
        this.myScrollContainer.nativeElement.scrollTop =
          this.myScrollContainer.nativeElement.scrollHeight + 500;
      } catch (err) {}
    }, 150);
  }
}

<div id="assistant">
  <button id="assistant-popup-button" (click)="openSupportPopup()">
    Chat Support?
  </button>
  <div id="assistant-popup" [style.display]="isOpen ? 'block' : 'none'">
    <div id="assistant-popup-header">
      Your friendly Assistant
      <button id="assistant-popup-close-button" (click)="openSupportPopup()">
       X
      </button>
    </div>
    <div id="assistant-popup-body">
      <div class="messages" #scrollMe>
        <!-- Display chat messages -->
        <div *ngFor="let message of messages" class="message">
          <div [class]="message.type">
            {{ message.message }}
          </div>
        </div>
        <!-- Display loading indicator -->
        <div
          *ngIf="loading"
```

```
          class="message"
          style="width: 100%; display: block"
        >
          <div [class]="'client'">...</div>
        </div>
      </div>
    </div>
    <form id="assistant-popup-footer" [formGroup]="chatForm">
      <!-- User input field -->
      <input
        formControlName="message"
        type="text"
        id="assistant-popup-input"
        placeholder="Type your message here..."
      />
      <!-- Submit button -->
      <button
        id="assistant-popup-submit-button"
        [disabled]="!chatForm.valid"
        (click)="sendMessage()"
      >
        Submit
      </button>
    </form>
  </div>
</div>
```

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

@Injectable({
  providedIn: 'root',
})
export class MessageService {
  constructor(private http: HttpClient) {}

  // Function to send a message to the Rasa chatbot backend
  sendMessage(message: string) {
    // Make a POST request to the Rasa chatbot webhook endpoint
    return this.http.post('http://localhost:5005/webhooks/rest/webhook', { message: message });
  }
}
```

Ionic

1. **Ionic Framework Documentation**: The official Ionic Framework documentation provides comprehensive information about Ionic's features, components, CLI commands, and platform-specific guidance. You can access it at ionicframework.com/docs.
2. **Ionic CLI Documentation**: The Ionic CLI documentation focuses on the command-line interface (CLI) tool used for Ionic development. It covers commands for project creation, building, testing, and deployment. You can find it at ionicframework.com/docs/cli.
3. **Ionic Angular**: As Ionic is built on top of Angular, it's crucial to refer to the Angular documentation as well. Angular documentation at angular.io/docs provides extensive information on Angular concepts, directives, modules, and other Angular-specific features.
4. **Ionic Native**: Ionic Native documentation covers the Ionic Native library, which provides a set of native device plugins for accessing native functionality on iOS, Android, and other platforms. It includes plugins for camera, geolocation, storage, and more. You can find it at ionicframework.com/docs/native.
5. **Ionic Capacitor**: Capacitor is the official Ionic alternative to Cordova for building native mobile apps. The Capacitor documentation offers details on using Capacitor for accessing native functionality, configuring plugins, and deploying to various platforms. Visit capacitorjs.com/docs for more information.
6. **Ionic Blog**: The Ionic Blog provides articles, tutorials, and updates on Ionic development. It covers various topics, including best practices, new features, and tips for efficient Ionic app development. You can explore it at ionicframework.com/blog.
7. **Ionic Forum**: The Ionic Forum is a community platform where you can ask questions, seek guidance, and engage with other Ionic developers. It's an excellent resource for getting help, sharing knowledge, and staying updated with Ionic-related discussions. You can access the forum at forum.ionicframework.com.

Certainly! Here are some coding-related notes that will help you while coding with Ionic:

Ionic Components and UI Elements:

- Ionic provides a wide range of UI components such as buttons, cards, forms, lists, modals, tabs, and more. These components can be easily integrated into your app by using Ionic's Angular directives.

- Use Ionic's CSS utility classes for styling and responsive design. Examples include `ion-text-center` for center-aligned text, `ion-padding` for adding padding to elements, and `ion-hide` for hiding elements.

- Leverage Ionic's theming system to customize the appearance of your app. You can define custom color palettes, typography, and other design variables in the `variables.scss` file.

- Take advantage of Ionic's pre-built Ionicons library for including icons in your app. Ionic also supports other icon libraries like Font Awesome.

Navigation and Routing:

- Use Ionic's built-in `ion-router` and `ion-nav` components for handling navigation and routing in your app. Define routes in the `app-routing.module.ts` file and use `ion-router-outlet` to render the appropriate page based on the current route.

- Utilize Ionic's navigation stack and history management to handle forward and backward navigation within your app. Use methods like `navigateForward()`, `navigateBack()`, and `navigateRoot()` to control the navigation flow.

Forms and User Input:

- Ionic provides form-related components such as `ion-input`, `ion-checkbox`, `ion-radio-group`, and `ion-select` for capturing user input. Bind form controls to properties in your component using `[(ngModel)]` or reactive forms with `formControl` and `formGroup` directives.

- Take advantage of Ionic's validation system to validate user input. Use built-in validators like `required`, `minLength`, and `pattern` or create custom validators for more complex validation scenarios.

- Handle form submission using events like `(ngSubmit)` or `(ionChange)`. You can perform actions such as data submission, validation, or navigation based on the form inputs.

Data Retrieval and Services:

- Use Angular's `HttpClient` or Ionic's `HTTP` module to make HTTP requests and retrieve data from APIs. Leverage `Observables` and `Promises` for handling asynchronous operations and handling the response data.

- Implement services to encapsulate business logic, data retrieval, and manipulation. Services help promote code reusability and separation of concerns in your application. Use Angular's dependency injection to inject services into components.

Device Features and Native Integration:

- Use Ionic Native plugins (based on Cordova or Capacitor) to access native device features like camera, geolocation, storage, and notifications. Install and import the necessary plugins, and then use their methods and events in your code.

- Handle platform-specific code and platform detection using Ionic's `Platform` API. This allows you to write platform-specific logic and adapt your app's behavior based on the current platform (iOS, Android, web).

Testing and Debugging:

- Utilize tools like `ionic serve` to run your app in the browser during development. Leverage browser developer tools for debugging and inspecting elements, and use console logs for logging relevant information.

- Write unit tests using frameworks like Jasmine and run them using Karma. Test components, services, and other parts of your app to ensure proper functionality and behavior.

- Use emulators, simulators, or physical devices to test your app on different platforms. Leverage tools like Ionic DevApp or Ionic View for testing on real devices over the network.

Deployment and App Store Submission:

- Use the Ionic CLI to build your app for different platforms (`ionic build`). Generate platform-specific builds, signing configurations,

and app icons/splash screens.

- Follow platform-specific guidelines to submit your app to app stores (Google Play Store, Apple App Store). Ensure compliance with platform-specific policies, performance optimizations, and any required app signing or provisioning profiles.

These coding-related notes should provide you with a good starting point for working with Ionic. Remember to consult the official Ionic documentation, explore community resources, and refer to specific API references as needed while coding. Good luck with your exam and happy coding!

Apologies for the confusion. Here are some practical code snippets and tips that can help you during your exam:

1. Creating a New Page:

To create a new page in Ionic, you can use the Ionic CLI command `ionic generate` or its shorthand `ionic g`. For example, to generate a new page called "HomePage," you can run the following command:

```
ionic generate page home
```

This command will create the necessary files for the home page, including a TypeScript file, an HTML template, and a SCSS style file. It will also update the necessary module files to include the new page.

2. Handling Button Clicks:

To handle button clicks in Ionic, you can use the `(click)` event binding. For example, to execute a function called `handleButtonClick()` when a button is clicked, you can use the following code:

```html
<button (click)="handleButtonClick()">Click me</button>
```

In your component TypeScript file, define the `handleButtonClick()` function to perform the desired actions:

```typescript
handleButtonClick() {
  // Perform actions when the button is clicked
}
```

3. Displaying Data Using ngFor:

To display a collection of data using Ionic's `ngFor` directive, you can iterate over an array in your template. For example, if you have an array of items in your component called `items`, you can display them using the following code:

```html
<ion-list>
  <ion-item *ngFor="let item of items">
    {{ item.name }}
  </ion-item>
</ion-list>
```

This will create an `ion-item` for each item in the `items` array and display the name of each item.

4. Show/Hide Elements Based on a Condition:

To show or hide elements based on a condition, you can use the `*ngIf` directive. For example, to display a message only if a variable `showMessage` is true, you can use the following code:

```html
<p *ngIf="showMessage">This is a message.</p>
```

In your component TypeScript file, set the value of `showMessage` based on your condition:

```typescript
showMessage = true; // Set to true or false based on your condition
```

```

**5. Fetching Data from an API:**

To fetch data from an API using the `HttpClient` module in Ionic, you can use the `get` method. Here's an example of fetching data from an API endpoint:

```typescript
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) {}

fetchData() {
  this.http.get('https://api.example.com/data').subscribe((response) => {
    // Process the response data
    console.log(response);
  });
}
```

Make sure to import the `HttpClient` module and inject it into your component's constructor. You can then call the `get` method with the API endpoint URL and subscribe to the response to handle the data.

These code snippets cover some common scenarios you may encounter while coding with Ionic. Remember to adapt them to your specific use cases and modify them as per your requirements during the exam.

Certainly! Here are some more practical code snippets and tips for coding with Ionic:

1. Displaying an Alert:
```

To display an alert dialog in Ionic, you can use the `AlertController`. Here's an example of creating and presenting an alert:

```typescript
import { AlertController } from '@ionic/angular';

constructor(private alertController: AlertController) {}

async showAlert() {
  const alert = await this.alertController.create({
    header: 'Alert',
    message: 'This is an alert message.',
    buttons: ['OK']
  });

  await alert.present();
}
```

In this example, the `showAlert()` function creates an alert using the `AlertController`, sets the header, message, and button text, and presents the alert using the `present()` method.

2. Handling Form Submissions:

When working with forms in Ionic, you can handle form submissions using the `(ngSubmit)` event binding. Here's an example:

```html
<form (ngSubmit)="submitForm()">
  <ion-input [(ngModel)]="name" name="name" placeholder="Name"></ion-input>
```

```
  <ion-button type="submit">Submit</ion-button>
</form>
```

In the component TypeScript file, define the `submitForm()` function to handle the form submission:

```typescript
name: string;

submitForm() {
  // Access the form data
  console.log(this.name);
  // Perform other form submission actions
}
```

3. Navigating to Another Page:

To navigate from one page to another in Ionic, you can use the `NavController` or the `(click)` event binding. Here's an example of navigating to another page on button click:

```html
<button (click)="goToPage()">Go to Another Page</button>
```

In the component TypeScript file, import the `NavController` and inject it into the constructor. Then, define the `goToPage()` function to navigate to the desired page:

```typescript
import { NavController } from '@ionic/angular';
```

```
constructor(private navController: NavController) {}

goToPage() {
  this.navController.navigateForward('/another-page');
}
```

Make sure to import the `NavController` module and inject it into your component's constructor. The `navigateForward()` method is used to navigate to the specified page path.

4. Using Ionicons:

Ionic comes with a built-in Ionicons library for including icons in your app. You can use the `<ion-icon>` component to display icons. Here's an example:

```html
<ion-icon name="heart"></ion-icon>
```

In this example, the `name` attribute of `<ion-icon>` specifies the name of the Ionicon to display. You can find a list of available icons and their names in the Ionicons documentation.

5. Custom CSS Styling:

To apply custom CSS styling to Ionic components, you can add classes or use inline styles. Here's an example of adding a custom class to an `ion-button`:

```html
<ion-button class="custom-button">Custom Button</ion-button>
```

In your global styles or component-specific SCSS file, define the styles for the custom class:

```scss
.custom-button {
  background-color: #ff0000;
  color: #ffffff;
}
```

These code snippets and tips should provide you with additional practical guidance for coding with Ionic. Remember to adapt them to your specific use cases and modify them as per your requirements during the exam.

Certainly! Here are a few more vital code snippets and tips that can be helpful while coding with Ionic:

1. Handling Ionic Events:

Ionic provides several events that you can listen to and handle in your code. Here's an example of handling the `ionChange` event on an input field:

```html
<ion-input (ionChange)="handleInputChange($event)"></ion-input>
```

In your component TypeScript file, define the `handleInputChange()` function to handle the event:

```typescript
handleInputChange(event: any) {
```

```
  const value = event.target.value;
  // Perform actions with the input value
}
```

You can handle other events like `ionBlur`, `ionFocus`, `ionSelect`, etc., by attaching event handlers in a similar way.

2. Using Ionic Modals:

Ionic provides a `ModalController` for creating and managing modals in your app. Here's an example of opening a modal:

```typescript
import { ModalController } from '@ionic/angular';
import { MyModalComponent } from '../path/to/my-modal.component';

constructor(private modalController: ModalController) {}

async openModal() {
  const modal = await this.modalController.create({
    component: MyModalComponent,
    componentProps: {
      // Pass data to the modal component
    }
  });

  await modal.present();
}
```

In this example, the `openModal()` function uses the `ModalController` to create a modal based on the `MyModalComponent`. You can pass data to the modal component using the `componentProps` property.

3. Using Ionic Slides:

Ionic's `ion-slides` component allows you to create slideable content. Here's an example of using slides:

```html
<ion-slides>
  <ion-slide>
    <h1>Slide 1</h1>
  </ion-slide>
  <ion-slide>
    <h1>Slide 2</h1>
  </ion-slide>
  <ion-slide>
    <h1>Slide 3</h1>
  </ion-slide>
</ion-slides>
```

You can add content inside `<ion-slide>` elements to create multiple slides. You can also use options like `slidesPerView`, `loop`, and `pagination` to customize the behavior of the slides.

4. Handling Platform-Specific Code:

To handle platform-specific code in Ionic, you can use the `Platform` API. Here's an example of conditionally executing code based on the platform:

```typescript
import { Platform } from '@ionic/angular';

constructor(private platform: Platform) {}

doPlatformSpecificStuff() {
  if (this.platform.is('ios')) {
    // Code specific to iOS
  } else if (this.platform.is('android')) {
    // Code specific to Android
  } else {
    // Code for other platforms
  }
}
```

The `is()` method of the `Platform` API is used to check the current platform. You can add conditions and execute platform-specific code accordingly.

5. Local Storage in Ionic:

Ionic provides a simple way to store and retrieve data using the `LocalStorage` API. Here's an example of storing and retrieving data from local storage:

```typescript
import { Storage } from '@ionic/storage';

constructor(private storage: Storage) {}

async storeData() {
```

```
  await this.storage.set('key', 'value');
}

async retrieveData() {
  const value = await this.storage.get('key');
  // Use the retrieved value
}
```

Make sure to install the `@ionic/storage` package and import the `Storage` module. You can then use the `set()` method to store data and the `get()` method to retrieve data from

local storage.

These additional code snippets and tips should further enhance your coding experience with Ionic. Remember to adapt them to your specific use cases and modify them as per your requirements during the exam.

Application Security

```
private readonly UserManager<AppUser> _userManager;
    private readonly SignInManager<AppUser> _signInManager;
    private readonly IConfiguration _configuration;


    public AccountController(UserManager<AppUser> userManager,
SignInManager<AppUser> signInManager, IConfiguration configuration)
    {
        _userManager = userManager; // Injected UserManager for managing user
operations
        _signInManager = signInManager; // Injected SignInManager for handling user
sign-in
        _configuration = configuration; // Injected IConfiguration for accessing
configuration values
    }


    [HttpPost("Register")]
    public async Task<IActionResult> Register(RegisterModel model)
    {
        if (ModelState.IsValid)
        {
            var user = new AppUser { UserName = model.Email, Email = model.Email };
            var result = await _userManager.CreateAsync(user, model.Password); // Creating
a new user using UserManager
            if (result.Succeeded)
            {
                await _signInManager.SignInAsync(user, isPersistent: false); // Sign-in the user
after successful registration
                return Ok(); // Return HTTP 200 (OK) status
            }
```

```
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description); // Add errors to ModelState if user creation fails
            }
        }
        return BadRequest(ModelState); // Return HTTP 400 (Bad Request) with ModelState errors
    }

    [HttpPost("Login")]
    public async Task<IActionResult> Login(LoginModel model)
    {
        if (ModelState.IsValid)
        {
            var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, lockoutOnFailure: false); // Sign-in the user using SignInManager
            if (result.Succeeded)
            {
                var user = await _userManager.FindByEmailAsync(model.Email); // Find the user by email
                var token = await GenerateJwtToken(user); // Generate a JWT token for the user
                return Ok(new { Token = token }); // Return the generated token in response
            }
            ModelState.AddModelError(string.Empty, "Invalid login attempt."); // Add error to ModelState if login attempt fails
        }
        return BadRequest(ModelState); // Return HTTP 400 (Bad Request) with ModelState errors
```

```
    }

    private async Task<string> GenerateJwtToken(AppUser user)
    {
        var claims = new List<Claim>
    {
        new Claim(JwtRegisteredClaimNames.Sub, user.UserName),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };

        var roles = await _userManager.GetRolesAsync(user); // Get user roles using UserManager
        claims.AddRange(roles.Select(role => new Claim(ClaimsIdentity.DefaultRoleClaimType, role))); // Add user roles as claims

        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Tokens:Key"])); // Get token signing key from configuration
        var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256); // Create signing credentials for JWT token
        var expires = DateTime.Now.AddDays(Convert.ToDouble(_configuration["Tokens:ExpireDays"])); // Set token expiration date

        var token = new JwtSecurityToken(
            _configuration["Tokens:Issuer"],
            _configuration["Tokens:Audience"],
            claims,
            expires: expires,
            signingCredentials: creds
        );
```

```
    return new JwtSecurityTokenHandler().WriteToken(token); // Write the JWT token as a string
  }
```

```
[HttpPost]
    [Route("Register")]
    public async Task<IActionResult> Register(UserViewModel uvm)
    {
      var user = await _userManager.FindByIdAsync(uvm.emailaddress);

      if (user == null)
      {
        user = new AppUser
        {
          Id = Guid.NewGuid().ToString(),
          UserName = uvm.emailaddress,
          Email = uvm.emailaddress
        };

        var result = await _userManager.CreateAsync(user, uvm.password);

        if (result.Errors.Count() > 0) return StatusCode(StatusCodes.Status500InternalServerError, "Internal Server Error. Please contact support.");
      }
      else
      {
        return Forbid("Account already exists.");
      }
```

```
        return Ok();
    }

    [HttpPost]
    [Route("Login")]
    public async Task<ActionResult> Login(UserViewModel uvm)
    {
        var user = await _userManager.FindByNameAsync(uvm.emailaddress);

        if (user != null && await _userManager.CheckPasswordAsync(user,
uvm.password))
        {
            try
            {
                //var principal = await _claimsPrincipalFactory.CreateAsync(user);
                //await HttpContext.SignInAsync(IdentityConstants.ApplicationScheme,
principal);
                return GenerateJWTToken(user);
            }
            catch (Exception)
            {
                return StatusCode(StatusCodes.Status500InternalServerError, "Internal
Server Error. Please contact support.");
            }
        }
        else
        {
            return NotFound("Does not exist");
        }
```

```
        //var loggedInUser = new UserViewModel { EmailAddress = uvm.EmailAddress,
Password = uvm.Password };


        //return Ok(loggedInUser);
    }


    [HttpGet]
    private ActionResult GenerateJWTToken(AppUser user)
    {
        // Create JWT Token
        var claims = new[]
        {
            new Claim(JwtRegisteredClaimNames.Sub, user.Email),
            new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
            new Claim(JwtRegisteredClaimNames.UniqueName, user.UserName)
        };


        var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Tokens:Key"]));
        var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);


        var token = new JwtSecurityToken(
            _configuration["Tokens:Issuer"],
            _configuration["Tokens:Audience"],
            claims,
            signingCredentials: credentials,
            expires: DateTime.UtcNow.AddHours(3)
        );
```

```
            return Created("", new
            {
                token = new JwtSecurityTokenHandler().WriteToken(token),
                user = user.UserName
            });
        }
-----------------------------------------------------------------------------------------------------
builder.Services.AddSwaggerGen(c =>
{
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        In = ParameterLocation.Header,
        Description = "Add Bearer Token",
        Name = "Authorization",
        Type = SecuritySchemeType.Http,
        BearerFormat = "JWT",
        Scheme = "bearer"
    });
    c.AddSecurityRequirement(new OpenApiSecurityRequirement
                {
                    {
                        new OpenApiSecurityScheme
                        {
                            Reference=new OpenApiReference
                            {
                                Type=ReferenceType.SecurityScheme,
                                Id = "Bearer"
                            }
```

```
                    },
                    new string[]{ }
                }
            });
});

builder.Services.AddIdentity<AppUser, IdentityRole>(options =>
{
    options.Password.RequireUppercase = false;
    options.Password.RequireLowercase = false;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireDigit = true;
    options.User.RequireUniqueEmail = true;
})
.AddEntityFrameworkStores<AppDbContext>()
.AddDefaultTokenProviders();

builder.Services.AddAuthentication()
            .AddCookie()
            .AddJwtBearer(options =>
            {
                options.TokenValidationParameters = new TokenValidationParameters()
                {
                    ValidIssuer = builder.Configuration["Tokens:Issuer"],
                    ValidAudience = builder.Configuration["Tokens:Audience"],
                    IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Tokens:Key"]))
                };
            });
```

Apologies for the confusion. Here are some vital code snippets along with notes for login and registration functionality using ASP.NET Identity:

1. User Registration:

a. Register User:

```csharp
[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);

        if (result.Succeeded)
        {
            // Handle successful registration
            return RedirectToAction("Login", "Account");
        }
        else
        {
            // Handle registration failure
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
```

```
            }
        }
    }

    // Handle invalid registration form
    return View(model);
}
```

b. Register View Model:

```csharp
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

- The `Register` action in the controller receives the registration form data from the `RegisterViewModel`.

- It creates a new `IdentityUser` object and sets the `UserName` and `Email` properties.

- The `CreateAsync` method of the `UserManager` is used to create the user.

- If the registration is successful, the user is redirected to the login page. Otherwise, any errors are added to the model state.

2. User Login:

a. Authenticate User:

```csharp
[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> Login(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, lockoutOnFailure: false);

        if (result.Succeeded)
        {
            // Handle successful login
            return RedirectToAction("Index", "Home");
        }
        else
        {
            // Handle login failure
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
        }
```

```
    }

    // Handle invalid login form
    return View(model);
}
```

b. Login View Model:

```csharp
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Remember me")]
    public bool RememberMe { get; set; }
}
```

- The `Login` action in the controller receives the login form data from the `LoginViewModel`.

- The `PasswordSignInAsync` method of the `SignInManager` is used to authenticate the user based on the provided email and password.

- If the login is successful, the user is redirected to the home page. Otherwise, an error message is added to the model state.

These code snippets provide the essential code for implementing login and registration functionality using ASP.NET Identity. Customize them according to your application's needs and ensure that the necessary view files and corresponding form inputs are present.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { BehaviorSubject, Observable } from 'rxjs';
import { map } from 'rxjs/operators';

interface RegisterModel {
  email: string;
  password: string;
  confirmPassword: string;
}

interface LoginModel {
  email: string;
  password: string;
}

interface LoginResponse {
  token: string;
}

@Injectable({
  providedIn: 'root'
```

```
})
export class AuthenticationService {
  private currentUserSubject: BehaviorSubject<any>;
  public currentUser: Observable<any>;

  private apiUrl = 'http://localhost:5240/api/';

  constructor(private http: HttpClient) {
    this.currentUserSubject = new
BehaviorSubject<any>(localStorage.getItem('currentUser') ?
JSON.parse(localStorage.getItem('currentUser') as string) : null);

    this.currentUser = this.currentUserSubject.asObservable();
  }

  public get currentUserValue() {
    return this.currentUserSubject.value;
  }

  register(model: RegisterModel) {
    return this.http.post<any>(`${this.apiUrl}account/register`, model);
  }

  login(model: LoginModel) {
    return this.http.post<LoginResponse>(`${this.apiUrl}account/login`, model)
      .pipe(map(response => {
        // store user details and jwt token in local storage to keep user logged in between
page refreshes
        localStorage.setItem('currentUser', JSON.stringify(response));
        this.currentUserSubject.next(response);
```

```
        console.log(this.currentUserSubject.value);
        return response;
      }));
  }

  logout() {
    // remove user from local storage to log user out
    localStorage.removeItem('currentUser');
    this.currentUserSubject.next(null);
  }
}
```

Reporting

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { Chart,registerables } from 'node_modules/chart.js';
import { RegionService } from '../service/region.service';
import { RegionModel } from '../service/Models/regionsModel';



Chart.register(...registerables);

@Component({
  selector: 'app-charts',
  templateUrl: './charts.component.html',
  styleUrls: ['./charts.component.css']
})

export class ChartsComponent implements OnInit{

  data: any;
  @ViewChild('myTemp')
  myTempRef!: ElementRef;



  constructor(private regionService : RegionService) {}

  ngOnInit(): void {
    this.regionService.getRegions().subscribe(response => {
      let regionList = response;
```

```
    this.data = response.$values;

    this.populateChartData(this.data);
    console.log('data',regionList)
    return regionList
  });
}

populateChartData(data: RegionModel[]) {

  let labelsData: string [] = [];
  let labelsPopulation: number [] = [];

  data.forEach((element: any) => {
    labelsData.push(element.code);
    labelsPopulation.push(element.population)
  });



  new Chart("barchart", {
    type: 'bar',
    data: {
      labels: labelsData,
      datasets: [{
        label: '# of Population',
        data: labelsPopulation,
        borderWidth: 1
      }]
    },
```

```
  options: {
    scales: {
      y: {
        beginAtZero: true
      },
    }

  }
});


new Chart("piechart", {
  type: 'pie',
  data: {
    labels: labelsData,
    datasets: [{
      label: '# of Population',
      data: labelsPopulation,
      borderWidth: 1
    }]
  },
  options: {
    scales: {
      y: {
        beginAtZero: true
      }
    }
  }
```

```
});

new Chart("dochart", {
  type: 'doughnut',
  data: {
    labels: labelsData,
    datasets: [{
      label: '# of Population',
      data: labelsPopulation,
      borderWidth: 1
    }]
  },
  options: {
    scales: {
      y: {
        beginAtZero: true
      }
    }
  }
});

new Chart("pochart", {
  type: 'polarArea',
  data: {
    labels: labelsData,
    datasets: [{
      label: '# of Population',
      data: labelsPopulation,
      borderWidth: 1
```

```
    }]
  },
  options: {
    scales: {
      y: {
        beginAtZero: true
      }
    }
  }
});

new Chart("rochart", {
  type: 'radar',
  data: {
    labels: labelsData,
    datasets: [{
      label: '# of Population',
      data: labelsPopulation,
      borderWidth: 1
    }]
  },
  options: {
    scales: {
      y: {
        beginAtZero: true
      }
    }
  }
});
```

```
new Chart("linechart", {
  type: 'line',
  data: {
    labels: labelsData,
    datasets: [{
      label: '# of Population',
      data: labelsPopulation,
      borderWidth: 1

    }]

  },
  options: {
    scales: {
      y: {
        beginAtZero: true
      }
    }
  }
});

new Chart("bubchart", {
  type: 'bubble',
  data: {
    labels: labelsData,
    datasets: [{
      label: '# of Population',
      data: labelsPopulation,
```

```
        borderWidth: 1

      }]

    },
    options: {
      scales: {
        y: {
          beginAtZero: true
        }
      }
    }
  });

 }

}
```

```
<div class="row">
  <div class="col-lg-6">
    <h2>Line Chart</h2>
    <canvas id="linechart"></canvas>
  </div>
  <div class="col-lg-6">
    <h2>Bar Chart</h2>
    <canvas id="barchart"></canvas>

  </div>
  <div class="col-lg-6">
```

```
      <h2>Pie Chart</h2>
      <canvas id="piechart"></canvas>

    </div>
    <div class="col-lg-6">
      <h2>Doughnut Chart</h2>
      <canvas id="dochart"></canvas>
    </div>
    <div class="col-lg-6">
      <h2>polarArea Chart</h2>
      <canvas id="pochart"></canvas>
    </div>

    <div class="col-lg-6">
      <h2>Radar Chart</h2>
      <canvas id="rochart"></canvas>
    </div>
</div>
```

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpClientModule, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs/internal/Observable';

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

  constructor(private httpclient : HttpClient) { }
```

```
  public getRegions(): Observable<any> {
    let appheaders = this.getHeaderConfigurations();
    return this.httpclient.get<any[]>('https://localhost:7250/Regions', { headers: appheaders});
  }

  private getHeaderConfigurations()
    {
        return new HttpHeaders({
            'Content-Type': 'application/json; charset=utf-8',
              'Access-Control-Allow-Origin': '*'
        });
    }
}
```

The code you provided is an Angular component called `ChartsComponent` that is responsible for displaying various types of charts using the Chart.js library. Let's break down how it works:

1. Import Statements: The necessary imports are included at the beginning of the code. This includes importing the `Component` decorator and other required modules from Angular, importing `Chart` and `registerables` from the `chart.js` library, and importing necessary services and models.

2. Component Decorator: The `@Component` decorator is used to define the metadata for the component. It specifies the selector, template URL, and style URLs for the component.

3. Class Declaration: The `ChartsComponent` class is declared and it implements the `OnInit` interface, indicating that it has an `ngOnInit` lifecycle hook.

4. Properties and ViewChild: The component has a property called `data` to store the chart data, and a `ViewChild` decorator is used to get a reference to an element with the template reference variable `myTemp`.

5. Constructor and ngOnInit: The constructor injects the `RegionService` for retrieving region data. The `ngOnInit` lifecycle hook is implemented, which is called after the component is initialized. In `ngOnInit`, a subscription is made to the `getRegions` method of the `RegionService` to fetch region data. The response is then used to populate the chart data and call the `populateChartData` method.

6. populateChartData: This method takes the region data as input and populates different types of charts using the Chart.js library. It iterates over the `data` array and extracts the necessary labels and population data for each region. It then creates different types of charts (bar, pie, doughnut, polar area, radar, line, bubble) using the extracted data and Chart.js. Each chart is created by providing the chart's element ID, the chart type, the labels, and the dataset (containing labels and data). The options for each chart include scale settings.

Overall, this component retrieves region data, populates different types of charts using the extracted data, and displays the charts on the web page using the Chart.js library.

Apologies for the confusion. Here are some code snippets and tips for styling Chart.js charts:

1. Customizing Chart Colors:

You can customize the colors of your Chart.js charts by specifying different color values for datasets, labels, and other chart elements. Here's an example of customizing the colors:

```javascript
const chartData = {
  labels: ['Red', 'Blue', 'Yellow'],
  datasets: [
```

```
    {
      label: 'My Dataset',
      data: [10, 20, 30],
      backgroundColor: ['red', 'blue', 'yellow'], // Specify custom background colors
      borderColor: ['red', 'blue', 'yellow'], // Specify custom border colors
    },
  ],
};
```

In this example, the `backgroundColor` and `borderColor` properties are set to custom color values for each dataset.

2. Changing Chart Fonts:

You can change the font styles used in your Chart.js charts by specifying the font properties in the chart options. Here's an example:

```javascript
const chartOptions = {
  plugins: {
    tooltip: {
      titleFont: {
        size: 16, // Specify the font size for tooltip titles
      },
      bodyFont: {
        size: 14, // Specify the font size for tooltip content
      },
    },
  },
```

};
```

In this example, the `titleFont` and `bodyFont` properties inside the `tooltip` plugin options define the font sizes for tooltip titles and content.

**3. Configuring Axes and Gridlines:**

Chart.js allows you to configure the appearance of axes and gridlines in your charts. Here's an example of customizing the axes and gridlines:

```javascript
const chartOptions = {
  scales: {
    x: {
      display: true,
      grid: {
        color: 'rgba(0, 0, 0, 0.1)', // Specify the color of x-axis gridlines
        borderDash: [5, 5], // Specify a dashed line for gridlines
      },
    },
    y: {
      display: true,
      grid: {
        color: 'rgba(0, 0, 0, 0.1)', // Specify the color of y-axis gridlines
        zeroLineColor: 'black', // Specify the color of the zero line
      },
    },
  },
};
```
```

```

In this example, the `grid` property is used to define the appearance of gridlines for the x-axis and y-axis.

**4. Adding Chart Titles and Labels:**

You can add titles and labels to your Chart.js charts to provide context and improve readability. Here's an example of adding chart titles and labels:

```javascript
const chartOptions = {
  plugins: {
    title: {
      display: true,
      text: 'My Chart Title', // Specify the chart title
      font: {
        size: 20, // Specify the font size for the title
      },
    },
    legend: {
      display: true,
      labels: {
        font: {
          size: 14, // Specify the font size for the legend labels
        },
      },
    },
  },
};
```
```

```

In this example, the `title` plugin options are used to define the chart title and its font size. The `legend` plugin options are used to customize the legend labels' font size.

These code snippets demonstrate how you can style Chart.js charts by customizing colors, fonts, axes, gridlines, and labels. Feel free to modify them according to your specific chart requirements and desired visual aesthetics.

Certainly! Here are some more code snippets and tips for styling Chart.js charts:

**1. Changing Chart Styles and Borders:**

You can apply various styles and borders to your Chart.js charts to enhance their visual appearance. Here's an example:

```javascript
const chartOptions = {
  elements: {
    point: {
      radius: 5, // Set the radius of data points
      backgroundColor: 'red', // Set the background color of data points
      borderColor: 'black', // Set the border color of data points
      borderWidth: 2, // Set the border width of data points
    },
    line: {
      tension: 0.3, // Set the tension of line charts
      backgroundColor: 'rgba(0, 0, 255, 0.2)', // Set the background color of line charts
      borderColor: 'blue', // Set the border color of line charts
      borderWidth: 2, // Set the border width of line charts
    },
```
```

```
  },
};
```

In this example, the `elements` property is used to customize the appearance of data points and lines in the chart.

2. Adjusting Chart Padding and Margins:

You can adjust the padding and margins of your Chart.js charts to control the spacing between elements. Here's an example:

```javascript
const chartOptions = {
  layout: {
    padding: {
      left: 10, // Set the left padding
      right: 10, // Set the right padding
      top: 10, // Set the top padding
      bottom: 10, // Set the bottom padding
    },
  },
};
```

In this example, the `padding` property is used to set the padding for the chart layout.

3. Applying Gradient Fill Colors:

You can apply gradient fill colors to certain chart types, such as line charts and bar charts, to create visually appealing effects. Here's an example of applying a gradient fill color to a line chart:

```javascript
const chartData = {
  labels: ['January', 'February', 'March', 'April', 'May'],
  datasets: [
    {
      label: 'My Dataset',
      data: [10, 20, 30, 40, 50],
      backgroundColor: 'rgba(0, 123, 255, 0.4)', // Set the starting color of the gradient
      borderColor: 'rgba(0, 123, 255, 1)', // Set the border color
      borderWidth: 2,
      fill: 'start', // Apply a gradient fill starting from the specified color
    },
  ],
};
```

In this example, the `backgroundColor` property specifies the starting color of the gradient, and the `fill` property is set to `'start'` to apply the gradient fill effect.

4. Handling Tooltips and Legends:

Chart.js provides options to customize tooltips and legends, allowing you to control their appearance and behavior. Here's an example:

```javascript
const chartOptions = {
  plugins: {
    tooltip: {
      backgroundColor: 'rgba(0, 0, 0, 0.8)', // Set the background color of tooltips
```

```
      titleColor: 'white', // Set the color of tooltip titles
      bodyColor: 'white', // Set the color of tooltip content
    },
    legend: {
      position: 'bottom', // Set the position of the legend (e.g., 'top', 'bottom')
      labels: {
        color: 'black', // Set the color of legend labels
      },
    },
  },
};
```

In this example, the `tooltip` plugin options are used to customize the appearance of tooltips

, and the `legend` plugin options are used to control the position and color of legend labels.

Feel free to experiment with these code snippets and modify them according to your specific chart styling requirements.