Great — let’s go **line by line** through the **WeatherService** and **WeatherController**. This will help you fully understand what each part does, how data flows, and what you could be asked about in an interview.

**🔧 WeatherService.cs**

Let’s assume this is your class (tell me if yours differs):

public class WeatherService

{

private readonly HttpClient \_httpClient;

private readonly string \_apiKey;

public WeatherService(IConfiguration configuration)

{

\_httpClient = new HttpClient();

\_apiKey = configuration["WeatherApi:ApiKey"];

}

public async Task<WeatherModel> GetCurrentConditionsAsync(string city)

{

var url = $"http://api.weatherapi.com/v1/current.json?key={\_apiKey}&q={city}";

var response = await \_httpClient.GetStringAsync(url);

dynamic data = JsonConvert.DeserializeObject(response);

var weather = new WeatherModel

{

City = data.location.name,

Temperature = data.current.temp\_c + " °C",

WeatherText = data.current.condition.text,

IconUrl = "https:" + data.current.condition.icon

};

return weather;

}

public async Task<List<ForecastDay>> GetFiveDayForecastAsync(string city)

{

var url = $"http://api.weatherapi.com/v1/forecast.json?key={\_apiKey}&q={city}&days=5";

var response = await \_httpClient.GetStringAsync(url);

dynamic data = JsonConvert.DeserializeObject(response);

var forecastList = new List<ForecastDay>();

foreach (var day in data.forecast.forecastday)

{

forecastList.Add(new ForecastDay

{

Date = day.date,

MinTemp = day.day.mintemp\_c + " °C",

MaxTemp = day.day.maxtemp\_c + " °C",

IconPhrase = day.day.condition.text,

IconUrl = "https:" + day.day.condition.icon

});

}

return forecastList;

}

}

**🔍 Line-by-Line Breakdown: WeatherService.cs**

**private readonly HttpClient \_httpClient;**

* Declares a single reusable HTTP client instance.
* Best practice to avoid socket exhaustion (though here, it's instantiated manually — ideally should be via IHttpClientFactory).

**private readonly string \_apiKey;**

* Stores your API key after it's read from the config.
* Keeps it private to prevent external access.

**public WeatherService(IConfiguration configuration)**

* Constructor with dependency injection.
* IConfiguration reads from appsettings.json or environment variables.

**Inside it:**

\_httpClient = new HttpClient();

\_apiKey = configuration["WeatherApi:ApiKey"];

* HttpClient is created.
* API key is retrieved using the key path WeatherApi:ApiKey.

**🔹 GetCurrentConditionsAsync(string city)**

var url = $"http://api.weatherapi.com/v1/current.json?key={\_apiKey}&q={city}";

* Builds the full API request URL dynamically based on the user's city input.

var response = await \_httpClient.GetStringAsync(url);

* Sends a GET request and gets the response as a string.

dynamic data = JsonConvert.DeserializeObject(response);

* Uses **dynamic typing** with Newtonsoft.Json to parse the JSON.
* Allows access like data.location.name without needing custom classes for deserialization.

var weather = new WeatherModel

{

City = data.location.name,

Temperature = data.current.temp\_c + " °C",

WeatherText = data.current.condition.text,

IconUrl = "https:" + data.current.condition.icon

};

* Constructs a WeatherModel with the parsed data.

**🔹 GetFiveDayForecastAsync(string city)**

Same pattern as above but for a different endpoint.

var url = $"http://api.weatherapi.com/v1/forecast.json?key={\_apiKey}&q={city}&days=5";

* Grabs forecast for 5 days.

foreach (var day in data.forecast.forecastday)

* Loops over each day in the forecast data.

forecastList.Add(new ForecastDay

{

Date = day.date,

MinTemp = day.day.mintemp\_c + " °C",

MaxTemp = day.day.maxtemp\_c + " °C",

IconPhrase = day.day.condition.text,

IconUrl = "https:" + day.day.condition.icon

});

* Builds a ForecastDay object and adds it to the list.

**🧭 WeatherController.cs**

public class WeatherController : Controller

{

private readonly WeatherService \_weatherService;

public WeatherController(IConfiguration configuration)

{

\_weatherService = new WeatherService(configuration);

}

public IActionResult Index()

{

return View();

}

[HttpPost]

public async Task<IActionResult> Index(string city)

{

try

{

var weather = await \_weatherService.GetCurrentConditionsAsync(city);

var forecast = await \_weatherService.GetFiveDayForecastAsync(city);

weather.Forecasts = forecast;

return View(weather);

}

catch

{

ViewBag.Error = "City not found or API failed.";

return View();

}

}

}

**🔍 Line-by-Line Breakdown: WeatherController.cs**

**public class WeatherController : Controller**

* Inherits from Controller, meaning it’s responsible for handling HTTP requests and returning views.

**private readonly WeatherService \_weatherService;**

* Stores a reference to your service layer (encapsulation of API logic).

**Constructor**

public WeatherController(IConfiguration configuration)

* Uses DI to pass in app config.
* Creates the WeatherService.

**Alternative (Better):**  
Inject WeatherService directly and register it in Startup.cs or Program.cs.

**GET: Index()**

public IActionResult Index()

{

return View();

}

* Handles the initial page load.
* Returns an empty view (user hasn’t searched yet).

**POST: Index(string city)**

public async Task<IActionResult> Index(string city)

* Handles form submission.
* Takes the city name from the form input.

var weather = await \_weatherService.GetCurrentConditionsAsync(city);

* Gets **current weather**.

var forecast = await \_weatherService.GetFiveDayForecastAsync(city);

* Gets **5-day forecast**.

weather.Forecasts = forecast;

* Adds the forecast data to the existing model object.

return View(weather);

* Passes the populated model to the view.

catch

{

ViewBag.Error = "City not found or API failed.";

return View();

}

* If anything fails (e.g. API error, invalid city), shows a friendly error message.

**✅ Ready for Interviews**

You now understand:

* How data flows from the controller → service → API → model → view.
* Why dynamic deserialization works and its pros/cons.
* How DI and configuration is used to manage keys and service setup.
* How controller actions are split between GET and POST.

Let me know if you'd like to:

* Go line-by-line through the **view** (Index.cshtml)
* Walk through **Razor conditional logic**
* Add improvements like **dependency injection** for WeatherService
* Simulate **interview questions** about this code

Ready when you are.