# CREATING A JSON API WITH MINIMAL APIS

Kamal Beydoun

Lebanese University – Faculty of Sciences I

Kamal.beydoun@ul.edu.lb

# WHAT IS AN HTTP API, AND WHEN SHOULD YOU USE ONE?

- Traditional web applications use HTML to handle requests, and the generated HTML is displayed in a web browser.

- **Razor Pages** are commonly used to build such applications, allowing the generation of HTML with Razor templates..

- While this approach is common and well-understood, modern application developers have other options to explore.

# WHAT IS AN HTTP API, AND WHEN SHOULD YOU USE ONE?

Traditional web applications
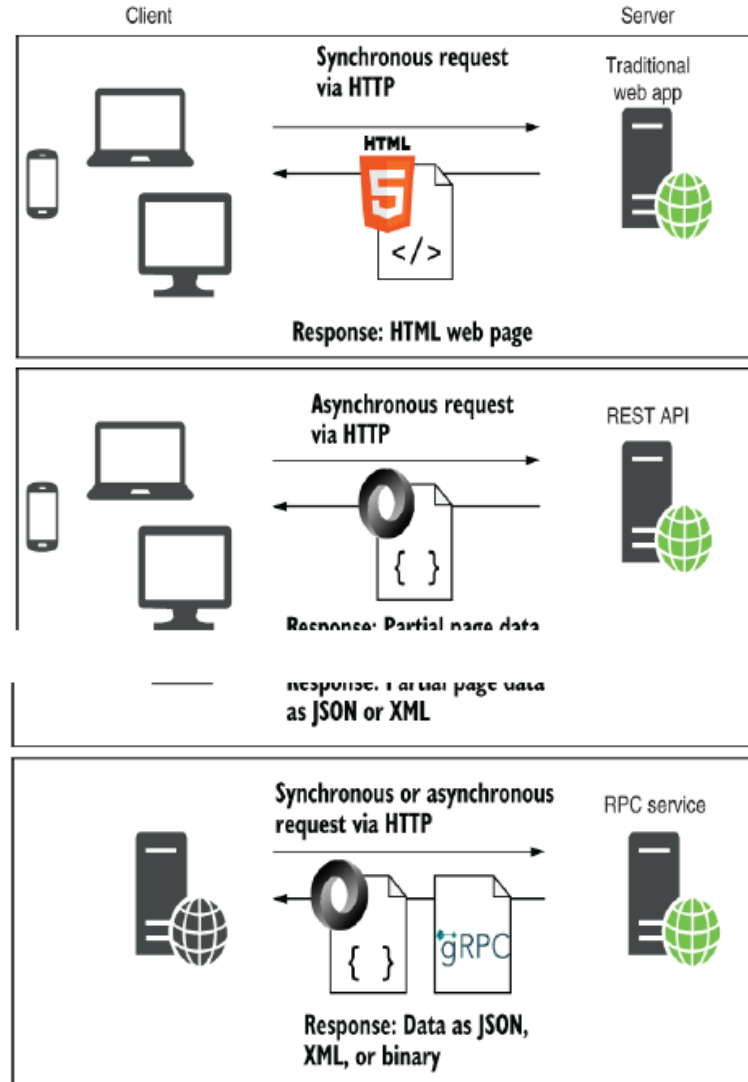
# SPA

- Client-side single-page applications (SPAs) have gained popularity, thanks to frameworks like **Angular**, **React**, and **Vue**.
  - These frameworks leverage **JavaScript** in a web browser to dynamically generate HTML for user interaction.

- In the SPA model, the server sends initial JavaScript to the browser when the user accesses the application.

- The user's browser loads and runs this JavaScript, initializing the SPA before fetching **application data** from the server.

- This approach contrasts **more dynamic and interactive** with traditional server-side rendering, emphasizing a user experience.

# SPA

- SPAs load in the browser and continue communication with the server over **HTTP**.
- Instead of sending HTML directly to the browser, the **server-side application sends data**, typically in **JSON** format, to the client-side application.
- The **SPA parses this data and generates HTML** dynamically to display to the user.
- The server-side application endpoint that the client communicates with is referred to as an **HTTP API, JSON API, or REST API** based on its design specifics.



Initial requests fetch a client-side Blazor application.

Subsequent requests fetch data in JSON format.

# HTTP API

- HTTP API exposes multiple URLs via HTTP for accessing or modifying data on a server.
  - **It commonly returns data in the JSON format**.

- HTTP APIs are also known as web APIs, although in ASP.NET Core, "web API" refers to a specific technology.

- **Mobile applications** typically interact with server applications via an HTTP API, receiving data in JSON format, similar to SPAs.

- Another use case for an HTTP API is when an application is intended for consumption by other backend services.
  - a web application designed for sending emails can provide an HTTP API.

# DEFINING MINIMAL API ENDPOINTS

```
app.MapGet("/", () => "Hello World!");
app.MapGet("/person", () => new Person("Andrew", "Lock"));
```

it always returns the same Person object.

If you create an API using the route template /person/{name}, for example, and send a request to the path /person/Andrew, the name parameter will have the value "Andrew".

# DEFINING MINIMAL API ENDPOINTS

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var people = new List<Person>                                    ❶
{                                                                ❶
    new("Tom", "Hanks"),                                         ❶
    new("Denzel", "Washington"),                                 ❶
    new("Leondardo", "DiCaprio"),                                ❶
    new("Al", "Pacino"),                                         ❶
    new("Morgan", "Freeman"),                                    ❶
};                                                               ❶

app.MapGet("/person/{name}", (string name) =>                    ❷
    people.Where(p => p.FirstName.StartsWith(name)));            ❸

app.Run();
```

If you send a request to /person/Al for the app defined, the name parameter will have the value "Al", and the API will return the following JSON:

[{"firstName":"Al","lastName":"Pacino"}]

# MAPPING VERBS TO ENDPOINTS

- Use **GET** only to retrieve data from the server.
- **Avoid** using GET to send or modify data on the server.
- Instead, utilize HTTP verbs such as POST or DELETE for data modification.

# MAPPING VERBS TO ENDPOINTS

- While each HTTP verb has a defined purpose, many apps only use POST and GET in practice.

  - acceptable for **server-rendered apps like Razor Pages**, as it simplifies the process.

- However, when creating APIs, it's **recommended to use HTTP verbs** with appropriate semantics.

  - Minimal APIs can define endpoints for other verbs using **Map\*** functions, like MapPost().

# MAPPING VERBS TO ENDPOINTS

| Method | HTTP verb | Expected operation |
|---|---|---|
| `MapGet(path, handler)` | GET | Fetch data only; no modification of state. May be safe to cache. |
| `MapPost(path, handler)` | POST | Create a new resource. |
| `MapPut(path, handler)` | PUT | Create or replace an existing resource. |
| `MapDelete(path, handler)` | DELETE | Delete the given resource. |
| `MapPatch(path, handler)` | PATCH | Modify the given resource. |
| `MapMethods(path, methods, handler)` | Multiple verbs | Multiple operations. |
| `Map(path, handler)` | All verbs | Multiple operations. |
| `MapFallback(handler)` | All verbs | Useful for SPA fallback routes. |

# NOTE

- Using the **wrong HTTP verb** for an API endpoint triggers specific behavior.
  - For example, if a GET endpoint is called with a POST request, the **handler won't execute**.
- Instead, a response with status code **405** *Method Not Allowed* is returned.
  - a mismatch between the expected and actual HTTP verb.
- Ensure the correct HTTP verb and URL path are used to avoid encountering a 405 response.

# DEFINING ROUTE HANDLERS WITH FUNCTIONS

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/fruit", () => Fruit.All);                          ❶

var getFruit = (string id) => Fruit.All[id];                    ❷
app.MapGet("/fruit/{id}", getFruit);                            ❷

app.MapPost("/fruit/{id}", Handlers.AddFruit);                  ❸

Handlers handlers = new();                                      ❹
app.MapPut("/fruit/{id}", handlers.ReplaceFruit);               ❹

app.MapDelete("/fruit/{id}", DeleteFruit);                      ❺

app.Run();

void DeleteFruit(string id)                                     ❺
{
    Fruit.All.Remove(id);
}
```
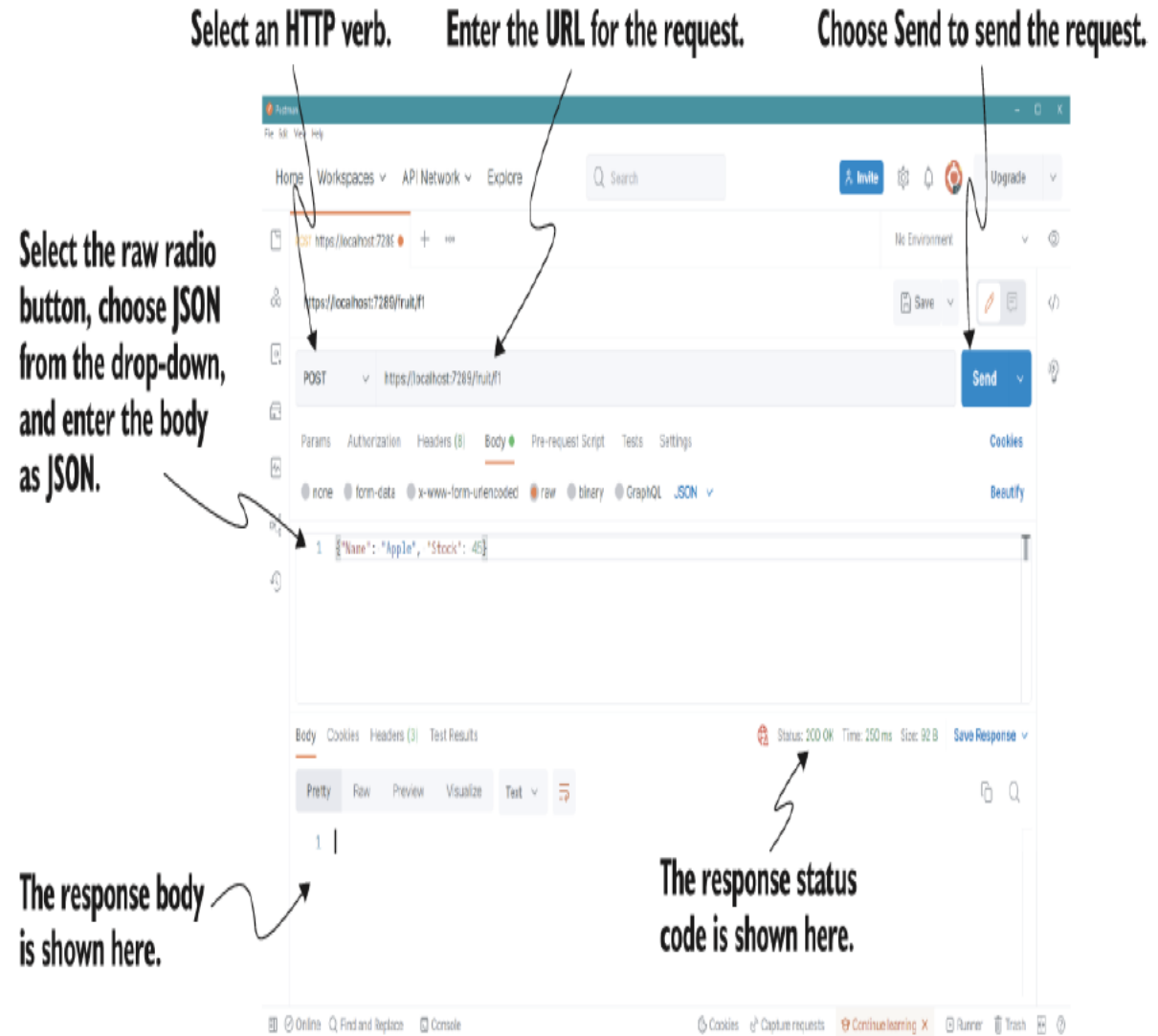
```
record Fruit(string Name, int Stock)
{
    public static readonly Dictionary<string, Fruit> All = new();
};

class Handlers
{
    public void ReplaceFruit(string id, Fruit fruit)            ❻
    {
        Fruit.All[id] = fruit;
    }

    public static void AddFruit(string id, Fruit fruit)         ❼
    {
        Fruit.All.Add(id, fruit);
    }
}
```

13

# NOTE

By contrast with APIs built using ASP.NET and ASP.NET Core web API controllers , **minimal APIs can bind only to JSON bodies** and always use the System.Text.Json library for JSON deserialization.



Select an HTTP verb.   Enter the URL for the request.   Choose Send to send the request.

Select the raw radio button, choose JSON from the drop-down, and enter the body as JSON.

The response body is shown here.

The response status code is shown here.

# GENERATING RESPONSES WITH IRESULT

The endpoint middleware handles each return type as follows:

- **void** or **Task**—The endpoint returns a 200 response with no body.

- **string** or **Task<string>**—The endpoint returns a 200 response with the string serialized to the body as text/plain.

- **IResult** or **Task<IResult>**—The endpoint executes the IResult.ExecuteAsync method.
  - Depending on the implementation, this type **can customize the response, returning any status code**.

- **T** or **Task<T>**—All other types (such as POCO objects) are serialized to JSON and returned in the body of a 200 response as application/json.

# RETURNING STATUS CODES WITH RESULTS AND TYPEDRESULTS

- ASP.NET Core exposes the simple **static** helper types **Results** and **TypedResults** in the namespace Microsoft.AspNetCore.Http.
  - Useful to **create a response** with common **status codes**, optionally including a JSON body.

- The only difference is that the Results methods return an IResult, whereas TypedResults return a concrete generic type, such as Ok<T>.

```csharp
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();            ❶

app.MapGet("/fruit", () => _fruit);

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)                           ❷
        ? TypedResults.Ok(fruit)                                    ❸
        : Results.NotFound());                                      ❹


app.MapPost("/fruit/{id}", (string id, Fruit fruit) =>
    _fruit.TryAdd(id, fruit)                                        ❺
        ? TypedResults.Created($"/fruit/{id}", fruit)               ❻
        : Results.BadRequest(new                                    ❼
            { id = "A fruit with this id already exists" }));        ❼

app.MapPut("/fruit/{id}", (string id, Fruit fruit) =>
{

    _fruit[id] = fruit;
    return Results.NoContent();                                     ❽
});

app.MapDelete("/fruit/{id}", (string id) =>
{
    _fruit.TryRemove(id, out _);                                    ❾
```

```csharp
    return Results.NoContent();                                     ❿
});

app.Run();
record Fruit(string Name, int stock);
```

❶ Uses a concurrent dictionary to make the API thread-safe

❷ Tries to get the fruit from the dictionary. If the ID exists in the dictionary, this returns true . . .

❸ . . . and we return a 200 OK response, serializing the fruit in the body as JSON.

❹ If the ID doesn't exist, returns a 404 Not Found response

❺ Tries to add the fruit to the dictionary. If the ID hasn't been added yet. this returns true . . .

❻ . . . and we return a 201 response with a JSON body and set the Location header to the given path.

❼ If the ID already exists, returns a 400 Bad Request response with an error message

❽ After adding or replacing the fruit, returns a 204 No Content response

❾ After deleting the fruit, always returns a 204 No Content response

# STATUS CODES

- **200 OK**—The standard successful response. It **often** includes content in the body of the response but doesn't have to.

- **201 Created**—Often returned when you successfully created an entity on the server.

- **204 No Content**—Similar to a 200 response but without any content in the response body.

- **400 Bad Request**—Indicates that the request was invalid in some way; often used to indicate data validation failures.

- **404 Not Found**—Indicates that the requested entity could not be found.

# WRITING THE RESPONSE MANUALLY USING HTTPRESPONSE

```
using System.Net.Mime
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/teapot", (HttpResponse response) =>            ❶
{
    response.StatusCode = 418;                              ❷
    response.ContentType = MediaTypeNames.Text.Plain;       ❸
    return response.WriteAsync("I'm a teapot!");            ❹
});

app.Run();
```

❶ Accesses the HttpResponse by including it as a parameter in your endpoint handler

❷ You can set the status code directly on the response.

❸ Defines the content type that will be sent in the response

❹ You can write data to the response stream manually.

# RETURNING USEFUL ERRORS WITH PROBLEM DETAILS

- **Problem Details** is a web **specification** for providing **machine-readable errors** for HTTP APIs.
  - It defines the required and optional fields that should be in the **JSON body for errors**.

- Two helper methods for generating Problem Details responses **from minimal APIs**:
  - Results.Problem()
  - Results.ValidationProblem() (plus their TypedResults complements).

- The only difference is that **Problem() defaults to a 500 status code**, whereas **ValidationProblem() defaults to a 400 status** and requires you to pass in a Dictionary of validation errors

# RETURNING USEFUL ERRORS WITH PROBLEM DETAILS

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit", () => _fruit);

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
```

```
        : Results.Problem(statusCode: 404));          ❶

app.MapPost("/fruit/{id}", (string id, Fruit fruit) =>
    _fruit.TryAdd(id, fruit)
        ? TypedResults.Created($"/fruit/{id}", fruit)
        : Results.ValidationProblem(new Dictionary<string, string[]>   ❷
        {                                             ❷
            {"id", new[] {"A fruit with this id already exists"}}   ❷
        }));                                          ❷
```

❶ Returns a Problem Details object with a 404 status code

❷ Returns a Problem Details object with a 400 status code and includes the validation errors

21

# RETURNING USEFUL ERRORS WITH PROBLEM DETAILS



Attempting to create a fruit with an id that has already been used

Every Problem Details response contains a title and a link to a description of the error.

Validation errors are listed under the errors key.

The status code of the response is set to 400.
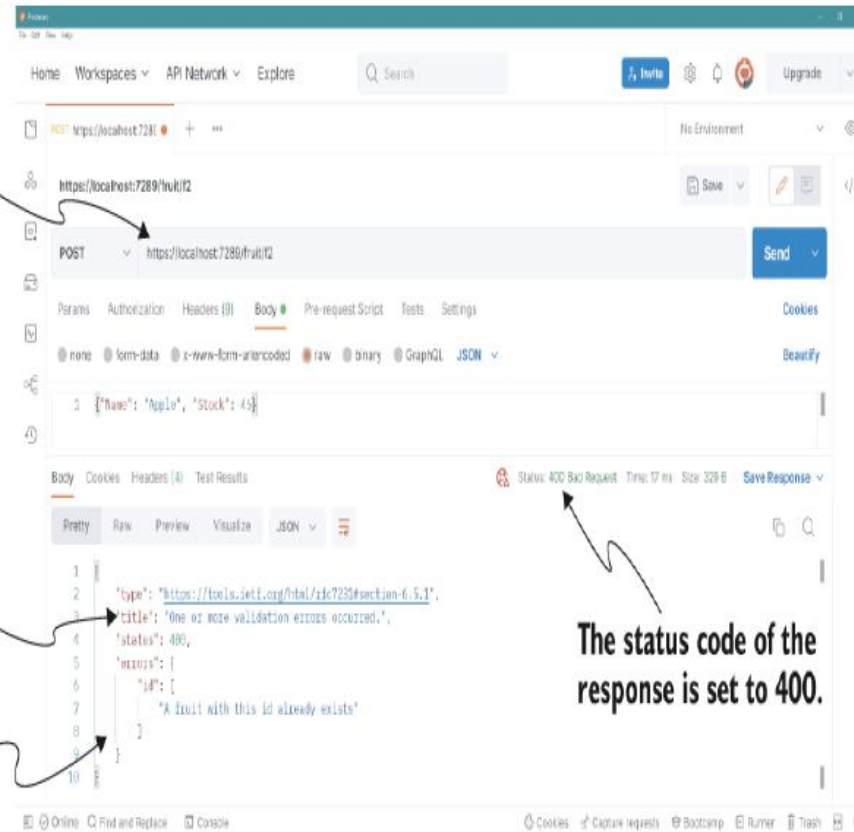
# CONVERTING ALL YOUR RESPONSES TO PROBLEM DETAILS

A minimal API application could generate **an error response** in several **ways**:

- Returning an **error status code** from an endpoint handler

- **Throwing an exception** in an endpoint <span style="color:red">handler</span>, which is **caught** by the ExceptionHandlerMiddleware or the DeveloperExceptionPageMiddleware and converted to an error response

- The middleware pipeline returning a 404 response because **a request isn't handled by an endpoint**

- A **middleware component** in the pipeline throwing an exception

- A middleware component returning **an error response** because a request requires authentication, and no credentials were provided
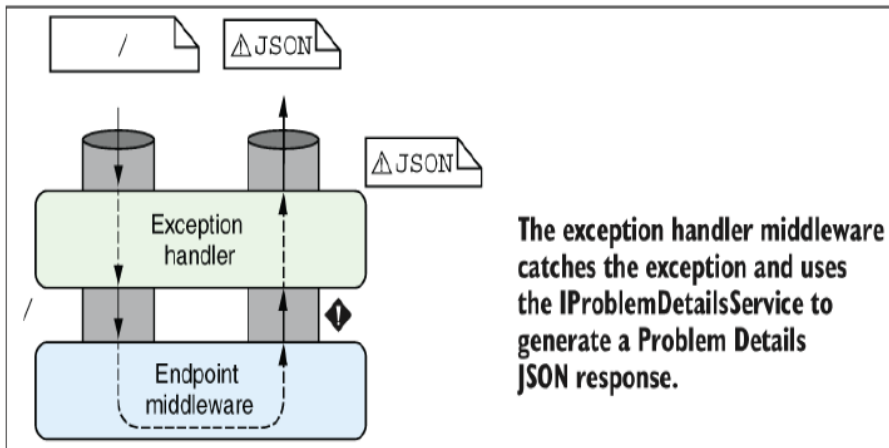
# CONVERTING EXCEPTIONS TO PROBLEM DETAILS
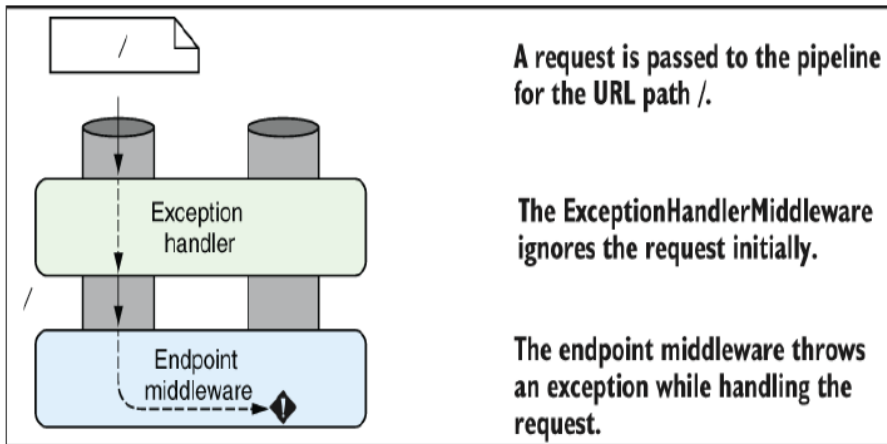
- **IProblemDetailsService**:
  - A new service introduced.
  - Can be added to the application by calling AddProblemDetails() on WebApplicationBuilder.Services.

- **ExceptionHandlerMiddleware** Configuration:
  - If the ExceptionHandlerMiddleware is configured **without specifying an error-handling path**, it automatically utilizes the IProblemDetailsService to generate the response.

# CONVERTING EXCEPTIONS TO PROBLEM DETAILS

A request is passed to the pipeline for the URL path /.

The ExceptionHandlerMiddleware ignores the request initially.

The endpoint middleware throws an exception while handling the request.

The exception handler middleware catches the exception and uses the IProblemDetailsService to generate a Problem Details JSON response.

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();                            ❶

WebApplication app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler();                                   ❷
}

app.MapGet("/", void () => throw new Exception());    ❸

app.Run();
```

❶ Adds the IProblemDetailsService implementation
❷ Configures the ExceptionHandlerMiddleware without a path so that it uses the IProblemDetailsService
❸ Throws an exception to demonstrate the behavior

# CONVERTING ERROR STATUS CODES TO PROBLEM DETAILS



A request is passed to the pipeline for the URL path/.

The StatusCodePageMiddleware ignores the request initially.

The endpoint middleware returns an error status code without a response body, in this case a 404 response.



The response propagates up the pipeline and is intercepted by the StatusCodePageMiddleware. The middleware uses the IProblemDetailsService to generate a Problem Details JSON response.

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();                        ❶

WebApplication app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler();
}

app.UseStatusCodePages();                                    ❷

app.MapGet("/", () => Results.NotFound());                   ❸

app.Run();
```
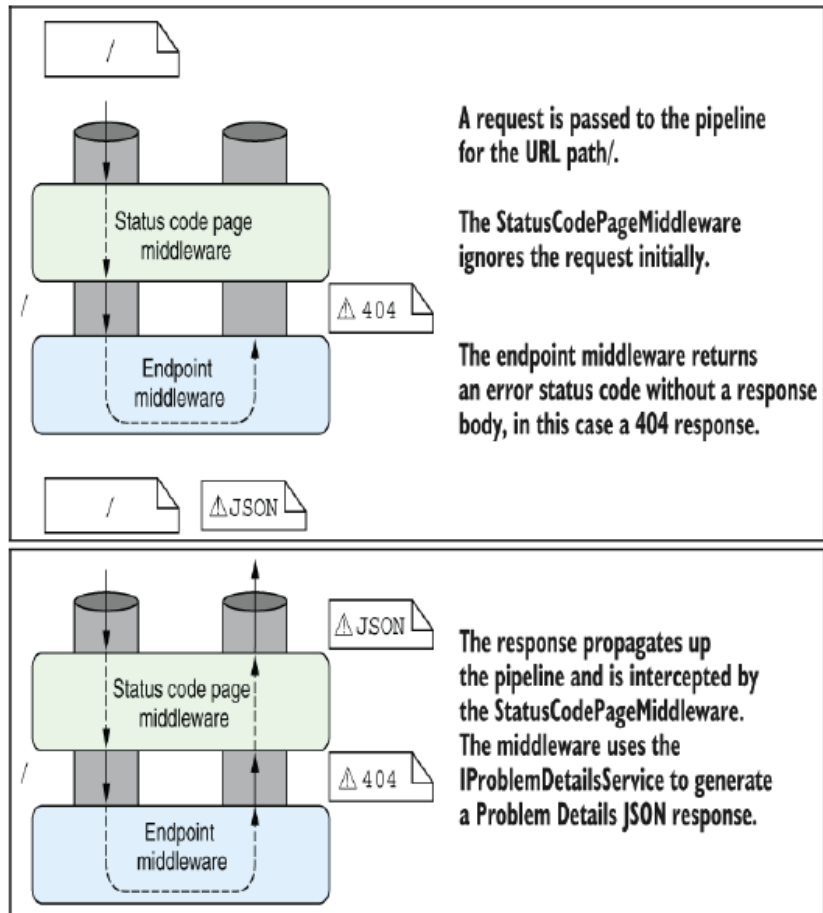
❶ Adds the IProblemDetailsService implementation
❷ Adds the StatusCodePagesMiddleware
❸ The StatusCodePagesMiddleware automatically adds a Problem Details body to the 404 response.

26

# RETURNING OTHER DATA TYPES

Results and TypedResults are convenient to return:

- **Results.File()**—Pass in the path of the file to return, and ASP.NET Core takes care of streaming it to the client.

- **Results.Byte()**—For returning binary data, you can pass this method a byte[] to return.

- **Results.Stream()**—You can send data to the client asynchronously by using a Stream.

# RUNNING COMMON CODE WITH ENDPOINT FILTERS

```
using System.Collections.Concurrent;
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
{
    if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))      ❶
    {
        return Results.ValidationProblem(new Dictionary<string, string[]>
        {
            {"id", new[] {"Invalid format. Id must start with 'f'"}}
        });
    }

    return _fruit.TryGetValue(id, out var fruit)
            ? TypedResults.Ok(fruit)
            : Results.Problem(statusCode: 404);
});

app.Run()
```
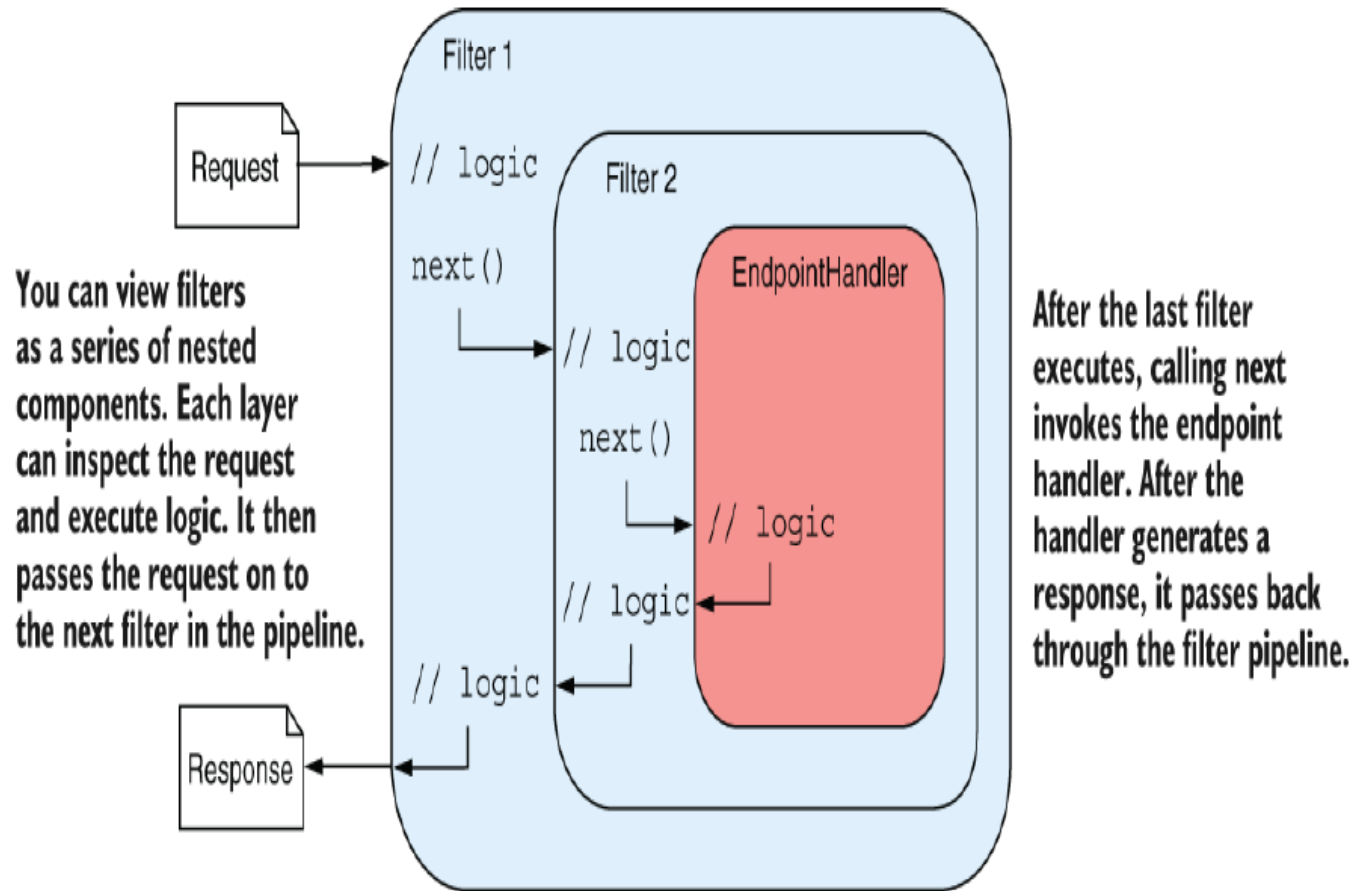
One improvement would be to move the validation code to a helper function.

❶ Adds extra validation that the provided id has the required format

# RUNNING COMMON CODE WITH ENDPOINT FILTERS



Filter 1

Request

// logic

next()

Filter 2

// logic

EndpointHandler

next()

// logic

// logic

// logic

Response

You can view filters as a series of nested components. Each layer can inspect the request and execute logic. It then passes the request on to the next filter in the pipeline.

After the last filter executes, calling next invokes the endpoint handler. After the handler generates a response, it passes back through the filter pipeline.

# RUNNING COMMON CODE WITH ENDPOINT FILTERS

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilter(ValidationHelper.ValidateId);        ❶

app.Run();

class ValidationHelper
{
    internal static async ValueTask<object?> ValidateId(   ❷
        EndpointFilterInvocationContext context,           ❸
        EndpointFilterDelegate next)                       ❹
    {
        var id = context.GetArgument<string>(0);           ❺
        if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
        {
            return Results.ValidationProblem(
                new Dictionary<string, string[]>
                {
```

```
                    {"id", new[]{"Invalid format. Id must start with 'f'"}}
                });
        }

        return await next(context);                         ❻
    }
}
```

❶ Adds the filter to the endpoint using AddEndpointFilter

❷ The method must return a ValueTask.

❸ context exposes the endpoint method arguments and the HttpContext.

❹ next represents the filter method (or endpoint) that will be called next.

❺ You can retrieve the method arguments from the context.

❻ Calling next executes the remaining filters in the pipeline.

# ADDING MULTIPLE FILTERS TO AN ENDPOINT

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
```

```
    .AddEndpointFilter(ValidationHelper.ValidateId)            ❶
    .AddEndpointFilter(async (context, next) =>                ❷
    {
        app.Logger.LogInformation("Executing filter...");      ❸
        object? result = await next(context);                  ❹
        app.Logger.LogInformation($"Handler result: {result}");❺
        return result;                                         ❻
    });

app.Run();
```

❶ Adds the validation filter as before
❷ Adds a new filter using a lambda function
❸ Logs a message before executing the rest of the pipeline
❹ Executes the remainder of the pipeline and the endpoint handler
❺ Logs the result returned by the rest of the pipeline
❻ Returns the result unmodified

31

# ADDING MULTIPLE FILTERS TO AN ENDPOINT

The first request is valid so the logging filter also executes.

The minimal API returns a JSON response with a 200 OK status code.

The second request is invalid, so the logging filter does not execute.



Command Prompt (light) - do

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:7289/fruit/f1 application/json 30
info: MinimalApiFilters[0]
      Executing filter...
info: MinimalApiFilters[0]
      Handler result: Microsoft.AspNetCore.Http.HttpResults.Ok'1[Fruit]
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 GET https://localhost:7289/fruit/f1 application/json 30 - 200
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:7289/fruit/1 application/json 30
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 GET https://localhost:7289/fruit/1 application/json 30 - 400
```

The first filter short-circuits the filter pipeline and returns a Problem Details JSON response with a 400 status code.

# FILTERS OR MIDDLEWARE: WHICH SHOULD YOU CHOOSE?

EBOOK

# GENERALIZING YOUR ENDPOINT FILTERS

EBOOK

# IMPLEMENTING THE IENDPOINTFILTER INTERFACE

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilter<IdValidationFilter>();           ❶

app.Run();

class IdValidationFilter : IEndpointFilter           ❷
{
    public async ValueTask<object?> InvokeAsync(       ❸
        EndpointFilterInvocationContext context,       ❸
        EndpointFilterDelegate next)                   ❸
    {
        var id = context.GetArgument<string>(0);
        if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
        {
            return Results.ValidationProblem(
```

```
                new Dictionary<string, string[]>
                {
                    {"id", new[]{"Invalid format. Id must start with 'f'"}}
                });
        }

        return await next(context);
    }
}
```
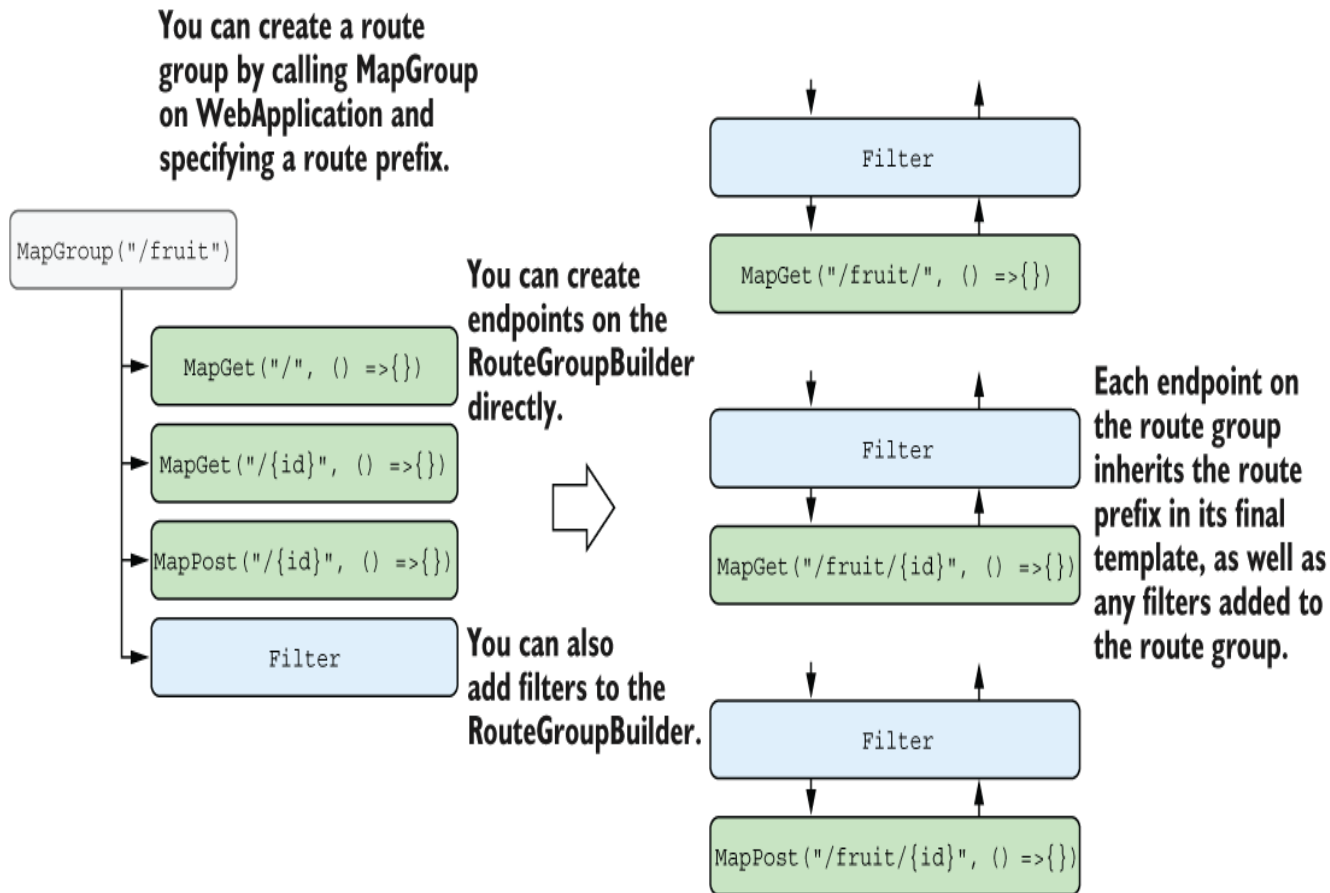
❶ Adds the filter using the generic AddEndpointFilter method

❷ The filter must implement IEndpointFilter . . .

❸ . . . which requires implementing a single method.

35

# ORGANIZING YOUR APIS WITH ROUTE GROUPS

- MapGet("/fruit", () => {/* */})

- MapGet("/fruit/{id}", (string id) => {/* */})

- MapPost("/fruit/{id}", (Fruit fruit, string id) => {/* */})

- MapPut("/fruit/{id}", (Fruit fruit, string id) => {/* */})

- MapDelete("/fruit/{id}", (string id) => {/* */})

You can create a route group by calling MapGroup on WebApplication and specifying a route prefix.

```
MapGroup("/fruit")
```

```
MapGet("/", () =>{})
```

```
MapGet("/{id}", () =>{})
```

```
MapPost("/{id}", () =>{})
```

```
Filter
```

You can create endpoints on the RouteGroupBuilder directly.

You can also add filters to the RouteGroupBuilder.

```
Filter
```

```
MapGet("/fruit/", () =>{})
```

```
Filter
```

```
MapGet("/fruit/{id}", () =>{})
```

```
Filter
```

```
MapPost("/fruit/{id}", () =>{})
```

Each endpoint on the route group inherits the route prefix in its final template, as well as any filters added to the route group.

# ORGANIZING YOUR APIS WITH ROUTE GROUPS

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

RouteGroupBuilder fruitApi = app.MapGroup("/fruit");                    ❶

fruitApi.MapGet("/", () => _fruit);                                     ❷

RouteGroupBuilder fruitApiWithValidation = fruitApi.MapGroup("/")       ❸
    .AddEndpointFilter(ValidationHelper.ValidateIdFactory);            ❹

fruitApiWithValidation.MapGet("/{id}", (string id) =>                   ❺
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404));

fruitApiWithValidation.MapPost("/{id}", (Fruit fruit, string id) =>    ❺
```

```
    _fruit.TryAdd(id, fruit)
        ? TypedResults.Created($"/fruit/{id}", fruit)
        : Results.ValidationProblem(new Dictionary<string, string[]>
          {
              { "id", new[] { "A fruit with this id already exists" } }
          }));

fruitApiWithValidation.MapPut("/{id}", (string id, Fruit fruit) =>     ❺
{
    _fruit[id] = fruit;
    return Results.NoContent();
});

fruitApiWithValidation.MapDelete("/fruit/{id}", (string id) =>         ❺
{
    _fruit.TryRemove(id, out _);
    return Results.NoContent();
});

app.Run();
```

❶ Creates a route group by calling MapGroup and providing a prefix

❷ Endpoints defined on the route group will have the group prefix prepended to the route.

❸ You can create nested route groups with multiple prefixes.

❹ You can add filters to the route group . . .

❺ . . . and the filter will be applied to all the endpoints defined on the route group.