The Startup (<.Net 6) & Program (>=.Net 6) Class

~~עד לגרסת .Net 6, פרויקטים שנבנו בארכיטקטורת ASP. NET Core הכילו, ע"פ קונבנציה, את המחלקה Stratup בקובץ Startup.cs.~~

~~במחלקה זו:~~

* ~~הוגדרו ה services הדרושים לאפליקציה – בפונקציה ConfigureServices~~
* ~~הוגדר request handling pipeline – תהליך עיבוד הבקשה – בפונקציה Configure~~

~~החל מגרסת .Net 6, מחלקה זו בוטלה והקוד של שתי הפונקציות נכתב ב Program.cs.~~

~~גם החלוקה לפונקציות בוטלה ולכן תוכן שתי הפונקציות נמצא ישירות ב Main (שכתובה במרומז בקובץ Program.cs, בהתאם לשינוי ב C# 10).~~

~~דוגמה ל startup class עד לגרסת .Net 6:~~

~~public class Startup~~

~~{~~

~~public Startup(IConfiguration configuration)~~

~~{~~

~~Configuration = configuration;~~

~~}~~

~~public IConfiguration Configuration { get; }~~

~~// This method gets called by the runtime.Use this method to add services to the container.~~

~~public void ConfigureServices(IServiceCollection services)~~

~~{~~

~~services.AddControllers();~~

~~}~~

~~// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.~~

~~public void Configure(IApplicationBuilder app, IWebHostEnvironment env)~~

~~{~~

~~if (env.IsDevelopment())~~

~~{~~

~~app.UseDeveloperExceptionPage();~~

~~}~~

~~app.UseRouting();~~

~~app.UseAuthorization();~~

~~app.UseEndpoints(endpoints =>~~

~~{~~

~~endpoints.MapControllers();~~

~~});~~

~~}~~

~~}~~

Program.cs בגרסת .Net 6מותאם ל REST API:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseAuthorization();

app.MapControllers();

app.Run();

*Services* הם רכיבי תוכנה שנמצאים בשימוש האפליקציה ומוסיפים יכולות לאפליקציה. למשל, רכיב של logging – תיעוד הקוד. מבצעים הרשמה (register) של services ב Program.cs (או ב לפני .Net 6, בפונקציה ConfigureServices בתוך ה Startup class) וצורכים (consume) אותם באמצעות dependency injection (DI) או ApplicationServices.

*Request Handling Pipeline* מורכב מסדרה של רכיבי middleware (אמצעי תיווך), לדוגמה: אמצעי תיווך יכול לטפל בבקשות לקבצים סטטיים או להעביר HTTP Request ל HTTPS. כל middleware מבצעת פעולה א-סינכרונית על ה HttpContext ואז מעבירה לטיפול של middleware אחרת, או מסיימת את הטיפול ב request.

הפונקציות Configure ו ConfigureServices נקראות מיד בעליית ה asp.net core wep api.

כיום ב .Net 6, ה main נקראת כמובן מיד בעליית האפליקציה, ובה מוגדרים גם ה services וגם ה request pipeline.

ב asp.net core יש רכיבי middleware שניתן להרכיב מהם pipeline סטנדרטי, למשל:

* [Developer Exception Page](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-3.0#developer-exception-page) – בסביבת פיתוח, מציג מידע מפורט לגבי שגיאה שהתרחשה
* [Exception handler](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-3.0#exception-handler-page) – טיפול בשגיאה
* [HTTP Strict Transport Security (HSTS)](https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-3.0#http-strict-transport-security-protocol-hsts)
* [HTTPS redirection](https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-3.0)
* [Static files](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files?view=aspnetcore-3.0) – גישה לקבצים סטטיים כמו דפי אינטרנט, קבצי תמונות וכדומה
* ASP.NET Core [MVC](https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.0) and [Razor Pages](https://docs.microsoft.com/en-us/aspnet/core/razor-pages/index?view=aspnetcore-3.0) – רלוונטי ליישומים שכוללים גם את התצוגה

כל middleware אחראית להפעיל את הבאה אחריה בתור, או לסיים את הטיפול, בהתאם ל request הספציפי שהתקבלה.

יש משמעות לסדר ה middlewares.

בדרך כלל פונקציה להוספת Middleware (אחת או יותר) ל request pipeline, תתחיל במילה Use.

**~~הסבר על Startup class – רלוונטי עד לגרסה .Net 6, לא כולל~~**

~~כאשר ה host נבנה, מציינים את ה Startup class שבה נעשה שימוש. בנית ה host מתבצעת בפונקציית ה Main של האפליקציה. בבניית פרויקט מסוג API אוטומטית מתווסף קובץ בשם Program.cs שבו מוגדרת הפונקציה Main.~~

~~בד"כ הקוד יראה כך:~~

~~public class Program~~

~~{~~

~~public static void Main(string[] args)~~

~~{~~

~~CreateHostBuilder(args).Build().Run();~~

~~}~~

~~public static IHostBuilder CreateHostBuilder(string[] args) =>~~

~~Host.CreateDefaultBuilder(args)~~

~~.ConfigureWebHostDefaults(webBuilder =>~~

~~{~~

~~webBuilder.UseStartup<Startup>();~~

~~});~~

~~}~~

~~גם האובייקט host מגדיר שימוש ב services, אלו יהיו זמינים באפליקציה כולה אך רק מהשלב של הקריאה לפונקציה Configure ב startup.~~

~~ישנם שלושה services שזמינים להזרקה באמצעות Dependency Injection(DI) ב ctor של startup:~~

[~~IWebHostEnvironment~~](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.hosting.iwebhostenvironment)

[~~IHostEnvironment~~](https://docs.microsoft.com/dotnet/api/microsoft.extensions.hosting.ihostenvironment)

[~~IConfiguration~~](https://docs.microsoft.com/dotnet/api/microsoft.extensions.configuration.iconfiguration)

~~השימושי מביניהם הוא~~ [~~IWebHostEnvironment~~](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.hosting.iwebhostenvironment)~~, שמזהה באיזו סביבה רצה האפליקציה (פיתוח\ייצור וכדו').~~

~~public class Startup~~

~~{~~

~~private readonly IWebHostEnvironment \_env;~~

~~public Startup(IConfiguration configuration, IWebHostEnvironment env)~~

~~{~~

~~Configuration = configuration;~~

~~\_env = env;~~

~~}~~

~~public IConfiguration Configuration { get; }~~

~~public void ConfigureServices(IServiceCollection services)~~

~~{~~

~~if (\_env.IsDevelopment())~~

~~{~~

~~}~~

~~else~~

~~{~~

~~}~~

~~}~~

~~}~~

~~הפונקציה ConfigureServices~~

* ~~בפונקציה זו נוסיף את ה services הדרושים לאפליקציה באמצעות האוסף של ה services שמתקבל כפרמטר לפונקציה. לדוג' services.AddRazorPages();~~
* ~~אופציונלית (אין חובה לממש אותה אם אין צורך בה)~~
* ~~נקראת לפני הפונקציה Configure~~
* ~~במידה ונרצה לשנות את הגדרות ברירת המחדל של האפליקציה שלנו, נציין את ההגדרות בה.~~

~~public void ConfigureServices(IServiceCollection services)~~

~~{~~

~~services.AddDbContext<ApplicationDbContext>(options =>~~

~~options.UseSqlServer(~~

~~Configuration.GetConnectionString("DefaultConnection")));~~

~~services.AddDefaultIdentity<IdentityUser>(~~

~~options => options.SignIn.RequireConfirmedAccount = true)~~

~~.AddEntityFrameworkStores<ApplicationDbContext>();~~

~~services.AddRazorPages();~~

~~}~~

~~הפונקציה Configure~~

~~בפונקציה זו מגדירים איך האפליקציה מגיבה ל Http Requests. ה requests מטופלות באמצעות מה שמכונה request handling pipeline, "צינור" טיפול בבקשות, שמוגדר באמצעות רכיבי תיווך המכונים middleware.~~

~~את רכיבי ה middleware מוסיפים למופע מסוג IApplicationBuilder  (נוצר ע"י ה host ומועבר לפונקציה כפרמטר, יוסבר בהמשך)~~

~~להלן דוגמה של Pipeline:~~

~~public class Startup~~

~~{~~

~~public Startup(IConfiguration configuration)~~

~~{~~

~~Configuration = configuration;~~

~~}~~

~~public IConfiguration Configuration { get; }~~

~~public void ConfigureServices(IServiceCollection services)~~

~~{~~

~~services.AddRazorPages();~~

~~}~~

~~public void Configure(IApplicationBuilder app, IWebHostEnvironment env)~~

~~{~~

~~if (env.IsDevelopment())~~

~~{~~

~~app.UseDeveloperExceptionPage();~~

~~}~~

~~else~~

~~{~~

~~app.UseExceptionHandler("/Error");~~

~~app.UseHsts();~~

~~}~~

~~app.UseHttpsRedirection();~~

~~app.UseStaticFiles();~~

~~app.UseRouting();~~

~~app.UseAuthorization();~~

~~app.UseEndpoints(endpoints =>~~

~~{~~

~~endpoints.MapRazorPages();~~

~~});~~

~~}~~

~~}~~

Dependency Injection(DI)

Dependency הוא אובייקט שנדרש עבור אובייקט אחר.

להלן דוגמא של מחלקה בשם MyDependency שמגדירה פונקציה WriteMessage. מחלקות אחרות ישתמשו בפונקציה זו.

public class MyDependency

{

public MyDependency()

{

}

public Task WriteMessage(string message)

{

Console.WriteLine(

$"MyDependency.WriteMessage called. Message: {message}");

return Task.FromResult(0);

}

}

כאשר ייצרו מופע של MyDependency ב class אחר, ה class יוכל להשתמש ב WriteMessage. לדוגמה:

public class IndexModel : PageModel

{

MyDependency \_dependency = new MyDependency();

public async Task OnGetAsync()

{

await \_dependency.WriteMessage(

"IndexModel.OnGetAsync created this message.");

}

}

התלות של IndexModel ב MyDependency בעייתית מכמה סיבות:

1. אם נרצה להשתמש באובייקט אחר במקום MyDependency, נצטרך לשנות את הקוד של IndexModel.
2. במידה ול MyDependency ישנם תלויות בעצמה, אזי IndexModel תצטרך להוסיף את התלויות האלה גם לעצמה. כשמדובר בפרויקטים גדולים זה הופך להיות מורכב מאד.
3. Unit testing – כאשר רוצים לבחון פעילות של יחידת תוכנה מסוימת, מנטרלים את שאר הרכיבים בשביל לוודא שהיחידה עצמה עובדת כראוי. באופן שבו הוסיפו את MyDependency ל IndexModel, לא ניתן יהיה לנטרל את MyDependency מהפונקציה OnGetAsync.

מה שפותר את הבעיות הנ"ל זה ה Dependency Injection.

1. ב dependency injection מוגדר Interface או base class שבאמצעותו עושים את ההפרדה מהמימוש עצמו.
2. ההוספה – הרשמה - של ה service - ה dependency - מתבצעת ב service container – איזשהו רכיב שמכיל את כל התלויות שבשימוש באיזשהו מקום באפליקציה כולה.asp.net core מספקת רכיב תוכנה כזה מובנה שנקרא .IServiceProvider הוספה של services אליו מתבצעת ב ConfigureServices תחת ה Startup class.
3. לאחר שכל ה services הוגדרו ב service container – הם זמינים לשימוש ברחבי האפליקציה באמצעות injection – הזרקה. כל מחלקה שמעוניינת להשתמש באיזשהו service – dependency – מגדירה ב constructor שלה מצביע אליו מסוג ה interface שלו, והתשתית של dependency injection תדאג ליצור מופע לפי הצורך, ובסיום השימוש בו היא תהיה גם אחראית למחוק את האובייקט מהזכרון.

בדוגמא לעיל, במקום לפתח מחלקה ופונקציה ספציפיות שכותבות לחלון ה console, נגדיר interface IMyDependency שבו WirteMessage. ניתן יהיה לממש את ה Interface בצורות שונות – כתיבה לחלון ה console, כתיבה לקובץ, כתיבה ל DB, וכו'.

public interface IMyDependency

{

Task WriteMessage(string message);

}

והנה מימוש אחד של IMyDependency:

public class MyDependency : IMyDependency

{

public Task WriteMessage(string message)

{

Console.WriteLine(

$"MyDependency.WriteMessage called. Message: {message}");

return Task.FromResult(0);

}

}

את IMyDependency נוסיף כ Service (ב ConfigureServices() עד ל .Net 6, ב Program החל מ .Net 6)

יתכן שה service container כבר מכיל services רבים, כאלו שהתווספו ע"י asp,ent core, כאשר נפוץ מאד שיהיו מאות services. ה services ש asp.net core מוסיפה הם גם לפי ה host שנבחר לאפליקציה (host זהו נושא בפני עצמו).

הוספת Service בפונקציה ConfigureServices: (עד לגרסה .Net 6)

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.AddSingleton<IMyDependency, MyDependency>();

services.AddSingleton<IYourDependency, YourDependency>();

}

הוספת Service בגרסת .Net 6:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddControllersWithViews();

var app = builder.Build();

ה Container הוא מסוג WebApplicationBuilder

לאחר שמוסיפים את ה service הרצוי, ניתן "להזריק" אותו לכל מחלקה שמעונינת להשתמש בו, באמצעות הגדרה שלו כפרמטר ל ctor.

public class ValuesController : ControllerBase

{

IMyDependency myDependency;

IYourDependency yourDependency;

public ValuesController(IMyDependency myDependency, IYourDependency yourDependency)

{

this.myDependency = myDependency;

this.yourDependency = yourDependency;

}

...

}

Service Lifetime (או: Dependency Lifetime)

הרכיב שמכיל את כל ה dependencies ואחראי לספק מופע כאשר מתבקש, הוא ה service collection.

כאשר מוסיפים dependency לservice collection, ניתן להגדיר את משך הזמן שבו המופע יישאר בזיכרון.

מכיוון שה service collection בעצם אחראי ליצירת המופעים מסוג מבוקש, ניתן להגדיר את "אורך חיי" המופע – כלומר, את משך הזמן בו מופע מסוים יישאר ב service collection עד למחיקתו ויצירת מופע חדש. הגדרה זו מתבצעת כאשר מוסיפים את ה dependency ל service collection.

האופציות להגדרת משך זמן הן:

1. Transient (חולף) – בכל פעם שבה ה service collection יתבקש לספק מופע מסוג מסוים, הוא יצור אותו מחדש.
2. Scope (טווח, מרחב) – בכל request שמתבצע לשרת, ייווצר מופע אחד, וכל הבקשות לסוג זה ישתמשו באותו מופע עבור request אחד.

לדוגמא גם controller וגם service מעוניינים לבצע כתיבה ללוג. שניהם יבקשו את ה dependency של Logger. כאשר מדובר באותו טווח של בקשה (כלומר עוד לא החזרנו תשובה ל client), שניהם יקבלו מצביע לאותו מופע.

1. Singleton (יחיד) – ייווצר אובייקט אחד בפעם הראשונה בה הוא יבוקש, ובכל פעם נוספת שיבוקש אובייקט מסוג זה – יתקבל המופע הראשון שנוצר.

כאשר משתמשים בסוג זה אין צורך ומוטב שלא לפתח את הפונקציונליות של singleton במחלקה המסוימת, אלא יש להניח ל service collection לבצע את הפונקציונליות הזו.

כל אחת מהאופציות הנ"ל ניתנת להוספה באמצעות פונקציות מוכנות, שמות הפונקציות בהתאמה לאורך החיים המבוקש.

Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>()

לדוגמא:

services.AddSingleton<IMyDep, MyDep>();

Entity Framework Context

Context של Entity Framework בדרך כלל מוסיפים ל container ב scope lifetime כיון שבדרך כלל פעולות שנעשות מול מסד נתונים (DB) רלוונטיות לבקשה מסוימת (לדוגמא: עדכון נתון, מחיקת נתון, וכדומה).

הוספת context מתבצעת באמצעות פונקציית ההרחבה (extension method). הוספת ה context ל services באמצעות פונקציה זו תוסיף אותו כ scoped, אלא אם כן מסיבה מיוחדת מעוניינים ב lifetime אחר אז ניתן לשלוח לפונקציה.

כלל חשוב: כאשר service מוגדר ב Lifetime מסוים, לא נכון שישתמש ב context ב Lifetime קצר יותר.

תרגיל

מטרת התרגיל להדגים ולהתנסות בהבדלים באורך החיים של האובייקטים, לפי הצורה בה הם הוספו ל service collection.

הנחיות:

שלב א' - בסיס:

1. צרי פרויקט web api core **ריק.**
2. צרי תיקיה ל Controllers.
3. צרי controller עם action get שמחזירה רשימת מחרוזות.
4. בשלב ראשון, החזירי רשימה סתמית של מחרוזות – abc וכדו'.
5. שני את הטעון שינוי על מנת שתוכלי להריץ את הפרויקט ולגשת ל get action שכתבת בסעיף 3.
6. הריצי את הפרויקט וודאי שחוזרת התוצאה שהחזרת מה action.

שלב ב':

1. הוסיפי ל solution פרויקט נוסף מסוג class library.
2. הוסיפי ל class library את ה interfaces וה classes המופיעים בנספח.
3. הוסיפי class ו interface - עם פונקציה אחת: CheckLifetime שלא מקבלת כלום ומחזירה רשימת מחרוזות. ה class יממש את ה Interface.
4. לפי שמות ה interfaces, השתמשי בפונקציה המתאימה עבור כל אחד להוספתו ל service collection. (**למנוע ספק: כמובן שהשמות נקבעו רק לצורך התרגיל אך אין משמעות לשם אלא רק לצורה בה האובייקט מתווסף ל !!!Service collection**).
5. הזריקי **למחלקה** ול Controllerאחד מכל סוג מה interfaces שיצרת.
6. בפונקציה קראי למאפיין Guid של כל אחד מה services והוסיפי אותו לרשימה של המחרוזות.
7. גם ב action ב controller הוסיפי את ה Guid של כל אחד מה services לרשימה של מחרוזות.
8. הזריקי ל controller את הסוג שיצרת בסעיף 9, והוסיפי ב action קריאה לפונקציה CheckLifetime.
9. מה controller החזירי רשימה של תוצר הפונקציה CheckLifetime + אלו שהוספת בסעיף 13.
10. בצעי 2 קריאות get ל Action (מבלי לעצור את ההרצה בין שתי הקריאות), והשווי בין המזהים שקבלת.

שימי לב:

מה השתנה בין הקריאות?

מה השתנה בתוך הקריאות – עם שתי הצבעות לאובייקט?

מה היה זהה בשתי הבקשות?

פלט לדוגמה:

פלט request ראשונה:

[

"In Controller transient: c617e627-4c3b-49bb-9239-1dd81f8616d1",

"In Controller scoped: f5582240-2c41-4c72-a121-f2bfd2a77c2d",

"In Controller singleton: 475cdad9-a112-441d-ba23-47242176f7ba",

"In Service transient: ae295382-eb06-4eb6-8a94-d8c25611f337",

"In Service scoped: f5582240-2c41-4c72-a121-f2bfd2a77c2d",

"In Service singleton: 475cdad9-a112-441d-ba23-47242176f7ba"

]

פלט request שניה:

[

"In Controller transient: 6890442f-b871-4eb0-a445-d57c7c44ed15",

"In Controller scoped: caf7fce7-0e74-4a91-aa1b-0101f4ba721e",

"In Controller singleton: 475cdad9-a112-441d-ba23-47242176f7ba",

"In Service transient: ae7142c2-3774-4c46-9d7a-66c0ff2e41b9",

"In Service scoped: caf7fce7-0e74-4a91-aa1b-0101f4ba721e",

"In Service singleton: 475cdad9-a112-441d-ba23-47242176f7ba"

]

נספח לתרגיל

1. Classes ו- interfaces לצורך התרגיל

public interface IOperation

{

Guid OperationId { get; }

}

public interface IOperationTransient : IOperation

{

}

public interface IOperationScoped : IOperation

{

}

public interface IOperationSingleton : IOperation

{

}

public class Operation : IOperationTransient,

IOperationScoped,

IOperationSingleton

{

public Operation() : this(Guid.NewGuid())

{

}

public Operation(Guid id)

{

OperationId = id;

}

public Guid OperationId { get; private set; }

}

1. GUID

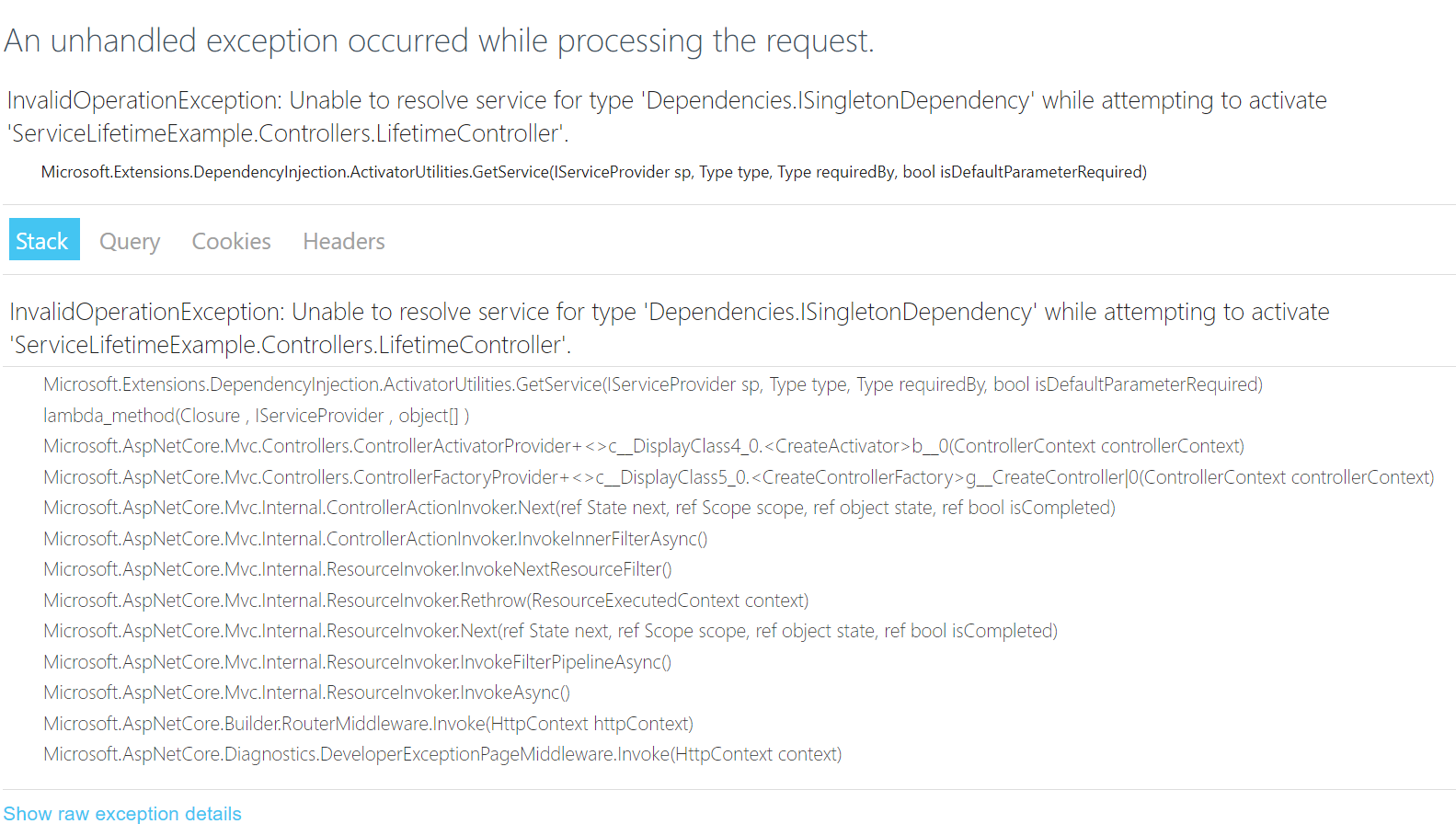
על מנת להבדיל בין במופעים השונים נעשה שימוש ב GUID.

GUID – Globally Unique Identifier – מזהה ייחודי עולמי, הוא מספר אקראי המשמש לזיהוי ייחודי של אובייקט בכל הקשר שהוא. קיימים 2**122** מספרים כאלו, כך שיתכן שמזהה מסוים יוגרל פעמיים, אבל הסבירות לכך נמוכה מאד. הסבירות ששני מספרים יוגרלו באותו הקשר (למשל, באותה תכנית), קטנה אף יותר.

ב C# ישנו struct מובנה ב FW שמאפשר ליצור GUID וכן פונקציונליות נוספת הקשורה ל GUID.

נציין: guid הוא struct ותיק ב C#, הוא לא קשור דווקא לweb api core וניתן להשתמש בו בכל סוגי הפרויקטים ב C#.

1. דוגמת שגיאה שמתקבלת כאשר מבקשים להזריק מופע מסוים ולא הוספנו אותו ל service collection:



ControllerBase class

הטכנולוגיה ASP.Net Core מאפשרת ליצור RESTful services הידועים גם כ Web APIs, באמצעות C#.

על מנת להיות מסוגלים לקבל בקשות ולטפל בהן, web api משתמש ב controllers. Controllers הם בעצם מחלקות שיורשות מהמחלקה ControllerBase. כלומר, Web Api מורכב מ Controller (אחד לפחות( שיורש מהמחלקה ControllerBase.

כדאי לדעת: קיימת מחלקה נוספת, Controller, שיורשת מהמחלקה ControllerBase ומוסיפה לה פונקציונליות שתאפשר החזרה של דפי אינטרנט ממש, אך המושג Web API בדרך כלל מתאר העברת מידע ולא תצוגה ויזואלית.

המחלקה ControllerBase מספקת סוגים שונים של מאפיינים ופונקציות שימושיים לטיפול בבקשות HTTP.

לדוגמא, הפונקציה CreatedAtAction שמוגדרת ב ControllerBase מחזירה קוד 201: (קוד 201 מוחזר כאשר הייתה בקשה ליצור משאב, והבקשה הצליחה)

[HttpPost]

public ActionResult<BookData> Create(BookData book)

{

var createdBook = bookService.Create(book);

return CreatedAtAction(nameof(GetById), new { id = createdBook.Id }, createdBook);

}

דוגמאות נוספות לפונקציות שימושיות:

| **מבצעת** | **פונקציה** |
| --- | --- |
| מחזירה Http Status 400 | [BadRequest](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.badrequest) |
| מחזירה Http Status 404 | [NotFound](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.notfound) |
| מחזירה קובץ | [PhysicalFile](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.physicalfile) |
| מפעילה model binding | [TryUpdateModelAsync](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.tryupdatemodelasync) |
| מפעילה בדיקת תקינות של מודל | [TryValidateModel](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.tryvalidatemodel) |

ActionResult

Web Api core מציע מספר סוגים שניתן להחזיר מ action.

1. Type ספציפי
2. IActionResult
3. ActionResult<T>

מתי כדאי להשתמש בכל אחד מהסוגים הללו?

1. Type ספציפי

הצורה הפשוטה ביותר להחזיר נתונים מ action היא להחזיר primitive type – כמו למשל int או string, או להחזיר complex type – כמו למשל אובייקט שמייצג ספר. לדוגמה:

[HttpGet("all")]

public List<BookData> GetAll()

{

return bookService.GetBooks();

}

במקרה הנ"ל ה action אינו מקבל פרמטרים ולכן אין צורך לבדוק את תקינותם והתאמתם, כמו כן אין תנאים נוספים, ולכן לכאורה יספיק להחזיר את רשימת הספרים.

אך לעיתים רבות נדרשות בדיקות שונות, בין אם על המבקש (לא כל משתמש יכול לקבל כל מידע), בין אם על הפרמטרים, או סתם הכלה של שגיאות שעלולות לקרות. במקרים כאלה נרצה להחזיר ActionResult וכך נוכל לתמוך בסוגים שונים של תשובות, בהתאם למתרחש.

1. IActionResult

IActionResult מייצגים סוגים שונים של Http Status Codes – כמו למשל 200 להצלחה, 404 עבור אובייקט שלא נמצא, או 401 במידה ומדובר בבעיית הרשאות. במחלקת ControllerBase ממנה יורשים ה api controllers, קיימות פונקציות שונות שמחזירות סוגים שונים של תשובות.

עבור הצלחה – הפונקציה מקבלת את סוג הנתונים אותו ה action אמור להחזיר.

[HttpGet("byid/{id}")]

public IActionResult GetById(int id)

{

var book = bookService.GetById(id);

if (book == null)

{

return NotFound();

}

return Ok(book);

}

בדוגמא לעיל, ה action מקבל id של ספר אותו הוא צריך לשלוף.

* במידה והספר עם המזהה המבוקש נמצא, הוא מוחזר באמצעות הפונקציה Ok שקיימת ב ControllerBase. הפונקציה מקבלת כפרמטר את מזהה הספר שיש להחזיר. במקום לקרוא לפונקציה אפשר לכתוב

return new OkObjectResult(book);

* + במידה והספר עם המזהה המבוקש לא נמצא, הפונקציה NotFound (גם היא של ControllerBase), תחזיר תשובה HttpNotFound – סטטוס 404. במקום לקרוא לפונקציה אפשר לכתוב

return new NotFoundResult();

1. ActionResult<T>

סוג זה דומה ל IActionResult בכך שהוא מאפשר החזרה של סוגים שונים של תשובות, אך עם שינויים קטנים:

* + במידה ומעונינים להשתמש ברכיב תיעוד של ה API, בסוג זה הכתיבה תהיה מקוצרת יותר.
  + על מנת להחזיר אובייקט, אין צורך להחזיר ActionResult עם האובייקט אלא ניתן להחזיר את האובייקט עצמו. זה בזכות היכולת של C# לבצע המרה מרומזת לאובייקטים, הוא ידע לקחת את ה T ולהפוך אותו ל ActionResult<T>

(המרה מרומזת ב C# מתאפשרת רק ל classes ולא ל interfaces , לכן לדוגמא ל IEnumerable לא תתבצע המרה מרומזת, אך ל List כן).

הaction יראה בדומה לצורה שראינו ב IActionResult, אך עם שינוי קטן (השורה מסומנת):

[HttpGet("byid/{id}")]

public ActionResult<BookData> GetById(int id)

{

var book = bookService.GetById(id);

if (book == null)

{

return NotFound();

}

return book;

}

Model Binder

Controllers מקבלים מידע מ http requests. מידע יכול להגיע מה url, מטפסים של html, מ query string, וכדומה. לפני השימוש במידע בקוד, יש צורך להמיר אותו לסוג המתאים ב .net. בגלל שמדובר בהמרה ממחרוזת (כך מגיע מ http request), כתיבת קוד של המרה המחרוזת לסוג המתאים עבור כל controller מחדש, הייתה עלולה להיות מייגעת וגם מקור לטעויות.

על מנת להקל על המתכנתים, Microsoft הוסיפו רכיב מיוחד למטרה זו, שמבצע את ההמרה של הנתונים מה request לסוג הנתונים המתבקש ב action. רכיב זה נקרא Model Binder (bind – קשירה, חיבור, בין הנתונים לאובייקטים המתאימים).

פעולות שה Model Binder מבצע:

1. מקבל נתונים ממקורות מידע שונים ב request: שדות טופס (html form), נתוני ניתוב (Route Data), query string
2. מעביר את הנתונים ל action המבוקש
3. ממיר את הנתונים (שהתקבלו במחרוזת) לאובייקטים של דוט נט.
4. מעדכן מאפיינים של אובייקטים.

לדוגמה, עבור ה action הבא שנמצא ב PetController:

[HttpGet("{id}")]

public ActionResult<Pet> GetById(int id, bool dogsOnly)

{

return null;

}

וה – url הבא:

<http://contoso.com/api/pets/2?DogsOnly=true>&name=dogly

לאחר שהרכיב של routingיחבר בין ה url ל action המתאים, ה model binder יבצע את הפעולות הבאות:

1. מזהה את הפרמטר id של הפונקציה GetById
2. מחפש את הערך של id מתוך ה request, באזורים שבהם יכול להיות ערך. מוצא את ה "2" ב route data.
3. ממיר את המחרוזת "2" למספר 2.
4. מזהה את הפרמטר הבא בתור בפונקציה GetById, ששמו dogsOnly,והוא בוליאני.
5. מחפש את הערך שלו ב request ומוצא ב Query String – ""DogsOnly=true (התאמת השמות היא case insensitive )
6. ממיר את המחרוזת "true" לערך בוליאני true.

לאחר שכל הפרמטרים פוענחו, התשתית קוראת לפונקציה ומעבירה לה את הפרמטרים.

הדוגמא לעיל היא דוגמא של simple types (string, int, bool,double וכדומה)

כאשר מדובר ב complex type (אובייקט שיש לו מאפיינים, כמו Person, Shape וכו'), הפעולה של מציאת הערך ב request תתבצע עבור כל מאפיין בנפרד, תתבצע בדיקת התקינות (Model Validation), ולבסוף יורכב ה Model עצמו מהנתונים ומשגיאות הוולידציה, ויוכנס למשתנה ControllerBase.ModelState. אם רוצים לבדוק אם התהליך של הכנסת הערכים ובדיקת תקינותם התבצע בהצלחה, בודקים את המאפיין ModelState.IsValid.

Sources – מקורות המידע

ישנם מספר מקורות מידע ב request, מהם יכול ה model binder לשאוב את המידע הנדרש, **ויחפש לפי הסדר הבא:**

1. נתוני שדות של form – נשלחים במפתחות של מפתח וערך בתוך גוף הבקשה
2. Request Body- מידע שנשלח בתוך גוף הבקשה, **רק ל controllers עם ApiController attribute.**
3. Route Data – הנתונים שנשלחים ב url, עד לסימן שאלה. רלוונטי רק ל simple types
4. Query String – החלק שמסימן השאלה ב url. רלוונטי רק ל simple types
5. קבצים שהועלו – יקושרו רק לפרמטרים שהם מסוג IFormFile או IEnumerable<IFormFile>

אם מעונינים לקבל ערך ממקור אחר בבקשה, ניתן לממש את המחלקות IValueProvider ו IValueProviderFactory.

אם מסיבה מסוימת רוצים לשנות את ההתנהגות ברירת המחדל, אפשר להשתמש ב attributes לציין במפורש מהיכן אמור להגיע ערך לפרמטר מסוים. כותבים את ה attribute לפני שם הפרמטר, לדוגמא:

public ActionResult<Pet> GetById([FromBody]int id, bool dogsOnly)

* [[FromQuery]](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromqueryattribute) – הנתון צריך להגיע ממחרוזת ה query string
* [[FromRoute]](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromrouteattribute) – הנתון צריך להגיע ממחרוזת ה url, החלק שלפני ה query string
* [[FromForm]](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromformattribute) – הנתון צריך להגיע מנתוני טופס html
* [[FromBody]](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.frombodyattribute) – הנתון צריך להגיע מגוף הבקשה
* [[FromHeader]](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromheaderattribute) – הנתון צריך להגיע מ http header (בכל בקשת http יש headers, אפשר להוסיף גם, לפי הצורך)

חשוב לדעת: רק פרמטר אחד יכול להיות מסומן ב [FromBody] (או ברירת המחדל הזו). התהליך בו נקרא ה Body למציאת ערכים יכול להתבצע רק פעם אחת ולכן אי אפשר לסמן שני פרמטרים בצורה זו.

[FromServices]

קיים FromServicesAttribute שנראה דומה לרשימה לעיל, אך למעשה הוא אינו מציין מאיזה חלק ב request לקבל את הפרמטר אלא לקבל הזרקה מה service collection. כך ניתן להזריק dependency עבור פונקציה ספציפית ולא למחלקה כולה.

בעיות המרה

אם נמצא ערך אך הוא אינו מתאים לסוג המבוקש, לדוגמא נמצאה מחרוזת עבור מספר, אזי המודל יסומן לא תקין. (ModelState.IsValid = false)

כאשר מתרחשת שגיאת המרה כזו ומדובר ב controller שמסומן ב ApiController, אוטומטית תוחזר תשובה של httpStatus = 400.

Simple Types

רשימת simpleTypes שה model binder יודע להמיר אליהם:

[Boolean](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.booleanconverter), [Byte](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.byteconverter),  [SByte](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.sbyteconverter), [Char](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.charconverter), [DateTime](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.datetimeconverter), [DateTimeOffset](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.datetimeoffsetconverter), [Decimal](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.decimalconverter), [Double](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.doubleconverter), [Enum](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.enumconverter), [Guid](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.guidconverter), [Int16](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.int16converter), [Int32](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.int32converter), [Int64](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.int64converter), [Single](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.singleconverter), [TimeSpan](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.timespanconverter), [UInt16](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.uint16converter), [UInt32](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.uint32converter), [UInt64](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.uint64converter), [Uri](https://docs.microsoft.com/en-us/dotnet/api/system.uritypeconverter), [Version](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.versionconverter)

~~Complex Types~~

~~ניתן לשלוט על הצורה בה מקושרים נתונים למאפיינים של אובייקטים באמצעות מספר attributes:~~

1. ~~[BindNever] – לא לכלול מאפיין זה כשמנסים לקשר בין נתוני הבקשה לאובייקט זה (לא יעבוד אם הנתונים מגיעים FromBody)~~
2. ~~[BindRequired] – אם לא מוצאים ערך למאפיין זה, לסמן את המודל כלא תקין (ModelState.IsValid = false)~~
3. ~~[Bind] – בחירה של אילו מאפיינים רוצים לקשר לערכים מהבקשה. ניתן לסמן גם על מחלקה וגם על פרמטר בפונקציה מסוימת, לדוגמא:~~

~~מחלקה:~~

~~[Bind("LastName,FirstMidName,HireDate")]~~

~~public class Instructor~~

~~פונקציה:~~

~~[HttpPost]~~

~~public IActionResult Post([Bind("LastName,FirstMidName,HireDate")] Instructor~~ instructor)

~~Model Validation~~

~~כאשר מגדירים action שמקבל אובייקט כפרמטר, ניתן לסמן הגדרות תקינות למאפיינים באמצעות Validation Attribute שמוגדרים ב~~ [~~System.ComponentModel.DataAnnotations~~](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations?view=netcore-3.1)~~.~~

~~רשימה חלקית:~~

1. ~~[CreditCard] – מוודא שהנתון מתאים לפורמט של כרטיס אשראי~~
2. ~~[Compare] – מוודא ששני שדות במודל מכילים את אותו ערך. שימושי מאד עבור שדה סיסמה וחיזוק סיסמה.~~
3. ~~[EmailAddress] – מוודא שהנתון מכיל כתובת מייל תקינה.~~
4. ~~[Phone] – מוודא שהנתון מכיל מספר טלפון.~~
5. ~~[Range] – מוודא שהנתון הוא בטווח מסוים.~~
6. ~~[RegularExpression] – מוודא שהנתון מתאים לביטוי רגולרי מסוים (שולחים כפרמטר את הביטוי).~~
7. ~~[Required] – מוודא שהשדה לא null (ולכן לא יעיל במקרה של value types)~~
8. ~~[StringLength] – מוודא אורך מסוים של מחרוזת~~
9. ~~[Url] – מוודא ששדה מכיל Url בפורמט תקין~~

~~כאשר הבדיקה של אחד מהמאפיינים במודל מחזירה false, המודל יסומן כלא תקין, ויתווסף מידע באובייקט לגבי השגיאה. להלן דוגמא לגישה לשגיאות במודל:~~

~~[HttpPost]~~

~~public ActionResult<bool> Post([FromBody]CBook book)~~

~~{~~

~~var errors = new StringBuilder();~~

~~if (!ModelState.IsValid)~~

~~{~~

~~foreach (var err in ModelState)~~

~~{~~

~~errors.Append($"{err.Key} set to {err.Value.AttemptedValue ?? "null"} {String.Join(",", err.Value.Errors.Select(e => e.ErrorMessage).ToArray())}");~~

~~}~~

~~return BadRequest(errors);~~

~~}~~

~~//save new object to DB...~~

~~return true;~~

~~}~~

~~חשוב לזכור: כאשר מסמנים Controller ב ApiControllerAttribute, שגיאות תקינות במודל אוטומטית יחזירו מה action תשובה של סטטוס 400 ללא הגעה לקוד ה action כלל!~~

ApiControllerAttibute

סימון controller ב ApiController Attribute יגרום למספר התנהגויות, ביניהן:

1. דרישה להמצאות attribute routing (conventional routing לא יעבוד).
2. אם ולידציות על המודל לא יצאו תקינות תוחזר אוטומטית response עם קוד 400.
3. קביעת binding source לפי סוג הפרמטר.