

AspNet Core ב- Middlewares

תוכן

1.....	הנחות ידע מוקדם :
1.....	הסבר המונח Middleware
2.....	המתודה RUN
3.....	המתודה USE
4.....	סדר ה- Middleware-ים הרגיל
4.....	היכן וכיצד מגדירים את סדר ה-Middlewares?
5.....	שימוש ב- MAP - כדי להסיט את ה-pipeline ל-Middleware המתאים
6.....	כתיבה ושימוש ב- Middleware מותאם אישית
9.....	סיכום
9.....	אריזה של Middlewares באמצעות extensions methods
9.....	בקצרה על Extension methods ב-C#
10.....	כיצד לעטוף Middlewares באמצעות Extension Methods
11.....	תרגילים
11.....	מקורות להרחבה :

הנחות ידע מוקדם :

בשלב זה, ההנחה היא שיש לך ידע עם הנושאים הבאים :

- מבנה של Url
- פונקציות למדא
- הקבצים והמחלקות הבסיסיות ב- AspNetCore.
- היכרות בסיסית עם HttpContext.

הסבר המונח MIDDLEWARE

המונח Middleware מתאר תבנית תיכנות, שבה מסתכלים על קוד בתוכנה בתור זרם של אירועים שעוברים בתוך צינור (pipeline). כאשר אנחנו יכולים להרכיב בתוך הצינור, מסננים שונים, שיכולים לבצע פעולות על האירועים : לשנות את האירוע, או לעצור את הזרימה שלו בצינור.

אם נדבר בצורה ספציפית יותר, ב-AspNetCore, אנחנו מסתכלים על Http Request (הבקשה שהגיעה מהאינטרנט) בתור האובייקט שזורם בתוך הצינור.

כל קריאה עוברת סידרה של פעולות בתוך AspNetCore.

הרצף של הפעולות מכונה בשם pipeline.

דוגמאות לפעולות נפוצות :

- ראשית, בודקים האם מדובר בקריאה שהגיעה בצורה מאובטחת, דרך SSL.
 - אם לא - יש לנו אפשרות להחליט, שקריאות לא מאובטחות – יחסמו כבר בשלב זה.
- שנית, בודקים האם הקריאה ביקשה רק קובץ סטטי, למשל קובץ CSS, או JS
 - אם כן, - אפשר פשוט להחזיר את הקובץ הסטטי, ואז לא להמשיך את רצף ה-pipeline.
- שלישית, נבדוק את הנתיב שממנו הגיעה הקריאה, למשל אם הקריאה הגיעה מנתיב של my-domain.com/customers, אז נדע להפנות אותה אל ה- Controller שעוסק בנושא Customers.

כל אחת מהפעולות האלו - היא Middleware, כלומר, היא פונקציה שמופעלת על הקריאה בתוך ה- Pipeline (=הצינור). כלומר Middleware הוא פונקציה שמשתלבת בתוך הזרם, ה- pipeline שמופעל על כל HttpRequest שמגיע אלינו.

כל אחד מה-Middlewares יכול לבצע 2 דברים :

- בוחר אם להעביר את הקריאה הלאה אל ה- Middleware הבא.
- יכול להפעיל קוד לפני ואחרי ה-Middleware הבא.

ברגע ש-Middleware מחליט להחזיר תשובה, או במילים אחרות - לכתוב לאובייקט ה-Response, אז זה השלב האחרון, וכאן מסתיימת/נקטעת השרשרת (pipeline) עבור אותו HttpRequest.

דבר נוסף שיש לציין הוא, שכאשר אנחנו מדברים על HttpRequest שעובר תהליך (=pipeline), אזי אנחנו מתייחסים לכל קריאה בנפרד. כלומר כל HttpRequest שהגיעה מהאינטרנט, עובר דרך ה-pipeline בצורה עצמאית. ובמידה ו-http request אחד נעצר, בגלל תנאי מסויים בתוכנה, זה לא משפיע על http request אחר שאינו עונה על התנאי.

כעת לאחר שהבנו מהו Middleware, נלמד :

- כיצד להשתמש ב-Middlewares הקיימים.
- כיצד להגדיר Middleware חדש משלנו, ולהחליט היכן הוא ימוקם בשרשרת.

לצורך בניית ה-Middleware משתמשים במתודות מובנות של הפריימוורק שנקראות : Run, Map, Use.

אפשר להגדיר Middleware בתור פונקציה אנונימית, או בתור קלאס נפרד.

המתודה RUN

נתחיל עם האפשרות הכי פשוטה - להשתמש ב-RUN-

הפונקציה RUN עושה רק דבר אחד - היא מלבישה את ה-Middleware במקום מסויים בצינור - ואז עוצרת את הזרימה.

כלומר לאחר השלב הזה (middleware) – הזרימה של אותו HttpRequest תיעצר, וה- Pipeline לא ימשיך עבורו (יכול להיות כמובן שהיו שלבים קודמים בשרשרת – middlewares קודמים שהקריאה תעבור דרכם).

בדוגמא הבאה למשל, אנחנו ממפים את כל הקריאות מגיעות למקום אחד, ומשתמשים לצורך כך בפונקציה אנונומית.

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

טיפ: חלק מה-Middlewares שמובנים בפריימוורק מציעים גם מתודת Run, במידה ורוצים להשתמש בהם בצורה שתעצור את הזרימה לאחר ה-Middleware הזה.

המתודה USE

אפשרות מעט יותר מורכבת, היא להשתמש ב - USE - שמקבלת גם פרמטר next - כלומר, ניתן להגדיר מי הפונקציה הבאה בתור, מי ה- Middleware הבא בשרשרת ה-pipeline.

וכך USE מאפשרת למעשה ליצור pipeline של Middleware-ים, כאשר כל קריאה יכולה לקרוא לשלב הבא בשרשרת.

אפשר כמובן גם "לחתוך" את הרצף/השרשרת שעובר הריקווסט, אם לא קוראים ל next .

וכאן המקום להזכיר מה שנאמר בתחילה : על מנת לחתוך את השרשרת, להפסיק את ה-pipeline, מה שצריך לעשות הוא לענות למי שפנה, כלומר, לכתוב תשובה לאובייקט ה- Response.

כתיבה של תשובה, מפסיקה את השרשרת.

להלן דוגמא בה, באמצעות שימוש ב-USE, אנחנו לא עושים דבר, אך רק קוראים ל-Middleware הבא בשרשרת, שמורץ באמצעות RUN, ואז הוא מפסיק את השרשרת.

ניתן לראות – שבמקומות בהן יש הערות, יכולנו להכניס קוד, שעושה כל דבר **מלבד** - לכתוב ל- Response , זאת כדי לא לקטוע את השרשרת.

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

כאשר אנחנו מחליטים לחתוך את הריקווסט, ולא להמשיך הלאה, זה נקרא בשם *short-circuiting* וזה מצב רצוי, כי אנחנו למעשה חוסכים עבודה.

למשל, דוגמא לחיסכון במשאבי מערכת היא שימוש ב-Middleware של static files, שבודק האם הבקשה היא רק לקבל קובץ סטטי, כמו קובץ CSS או קובץ JS, ובמידה וכן, אם יש קובץ סטטי להגיש, אזי הקובץ מוחזר לדפדפן, וכל שאר העבודה לא מבוצעת (שאר ה-Middlewares).

יחד עם זאת צריך להיזהר לא לקרוא ל-`next.Invoke` אחרי שה `HttpResponse`-התחיל להיות מוחזר ללקוח, כיוון שזה גורם להפרה של הפרוטוקול, ועלול לחבל בתצורה של ה-body.

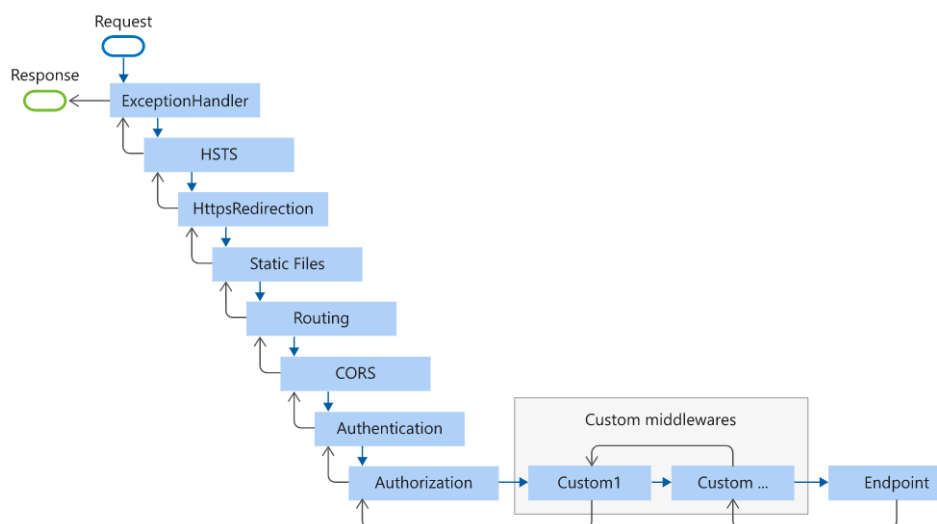
כדי להימנע מזה אפשר להשתמש במאפיין שנקרא: `HasStarted`.

סיכום ביניים - ההבדל בין RUN לעומת USE הוא ש RUN לא מקבלת פרמטר next ולכן גם לא תמשיך הלאה, אלא תסיים את pipeline. כלומר כל Middleware שנכתוב אחרי ה-RUN לא ירוץ בפועל.

סדר ה-MIDDLEWAREים הרגיל

זה המהלך הרגיל למשל של רצף ה-pipeline של ASP.NET Core.

הערה: חלקם רלוונטים רק במקרה שמשתמשים ב-MVC \ Razor pages.



הסדר של ה-Middleware ים הוא קריטי לביצועים ולאבטחה, ולכן, לא מומלץ לשנות את הסדר של המידלוררים המובנים, אלא רק להחליט האם משתמשים או לא משתמשים בהם, וכמובן להוסיף עליהם Middlewares שנכתוב בעצמנו, אם יש צורך בכך.

היכן וכיצד מגדירים את סדר ה-MIDDLEWARES?

ההגדרה מתבצעת במתודת ה-`Configure` שבקובץ `Startup.cs`.

להלן דוגמא לסדר המומלץ:

```

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    // app.UseCookiePolicy();

    app.UseRouting();
    // app.UseRequestLocalization();
    // app.UseCors();

    app.UseAuthentication();
    app.UseAuthorization();
    // app.UseSession();
    // app.UseResponseCompression();
    // app.UseResponseCaching();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}

```

להלן דוגמא לחשיבות של הסדר של ה- Middlewares :

- הסדר של UseCors , UseAuthentication , UseAuthorization , חייב להופיע בסדר שהודגם בתרשים. כיוון שקודם כל בודקים האם יש בעיית Cors (קוד JS שפונה מדומיין שאינו הדומיין שלנו - מצריך הרשאה מיוחדת מאיתנו) , לאחר מכן נראה האם הקריאה הגיעה ממשתמש שמוכר לנו (אוטנטיקציה), ולבסוף נראה האם למשתמש הזה עש הרשאה לבקש את התוכן שהוא ביקש (אוטוריזציה).

ברור, שזה לא הגיוני לבדוק בסדר הפוך, הרי אם אנחנו עדין לא יודעים שהמשתמש מוכר לנו, אז מה הטעם לבדוק האם יש לו הרשאה, שהרי... עדין איננו יודעים את הזהות שלו.

טיפ: עבור טיפול בשגיאות, מוסיפים את ה- Middleware שנקרא [UseExceptionHandler](#) ראשון בסדר.

שימוש ב MAP - כדי להסיט את ה PIPELINE ל- MIDDLEWARE המתאים

המתודה האחרונה שנותר לנו לסקור, היא Map.

המתודה Map מאפשרת לנתב בצורה חכמה את הבקשה, כך שבקשות מסוימות יעברו דרך Middlewares מסוימים, ואילו בקשות אחרות יעברו דרך Middlewares אחרים.

לדוגמא, אפשר להשתמש ב Map לצורך הפניה של קריאות לפי ה- Route שממנו הן הגיעו.

בקוד הבא, נראה שקריאות שהגיעו דרך myDomain.com/map1 ינותבו אל הפונקציה HandleMap1

ולעומתן, קריאות שהגיעו דרך myDomain.com/map2 ינותבו לפונקציה אחרת, שנקראת HandleMap2 .

ואילו כל קריאה שתגיע, תגיע בסופו של דבר לקטע האחרון בקוד, אל פונקציה הלמדא שמחזירה את המשפט Hello from non-map delegate.

כמובן, שזה תלוי אם הקריאה לא נקטעה לפני כן באמצעות המידלוררים שטיפלו בה בדרך, במקרה כזה, היא לא תגיע ל- Middleware האחרון, כיוון שהיא נקטעה לפני כן והתשובה הוחזרה ללקוח.

```
{
  app.Map("/map1", HandleMapTest1);

  app.Map("/map2", HandleMapTest2);

  app.Run(async context =>
  {
    await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
  });
}
```

טיפ למתקדמים : במידה ורוצים לעשות ניתוב מורכב, אפשרי לעשות גם map בתוך map וכדומה, כדי לענות על צרכים מורכבים בניתוב הבקשות. במקרה כזה, כל Middleware בשרשרת, יוריד את החלק במחרוזת של ה- HttpRequest.PathBase. כלומר, נניח שהגיעה בקשה מ- myDomain.com/map1/otherMap2

ובהגדרות שרשמנו, ראשית ענינו על הבקשה באמצעות Handler שטיפל בכל מי שמכיל את map1/,

אז כאשר הקריאה, האובייקט של HttpRequest ינותב אל ה-Map **הפנימי**, אז המחרוזת שתועבר במאפיין HttpRequest.PathBase תכיל רק את החלק הנותר, שעדין לא קיבל מענה, כלומר רק את otherMap2/.

אפשר לעשות את זה עוד יותר מורכב עם MapWhen שמכיל גם, predicate, ואז הניתוב יקרה רק אם ה- predicate מתקיים.

בצורה דומה יש גם UseWhen שגם מקבל predicate. ההבדל בינו לבין MapWhen הוא שב UseWhen- זה לא קוטע את המשך ה pipeline - אלא אחרי ה- Middleware אז הכל ממשיך כרגיל (אלא אם כן אחד ה- Middleware'ים קטע את הריקווסט), לעומת זאת, אם נשתמש ב MapWhen- אז זה מנתב את ה pipeline- שונה, ושאר ה pipeline- לא ממשיך.

כתיבה ושימוש ב- MIDDLEWARE מותאם אישית.

ניתן לכתוב middleware בשתי צורות :

- פונקצייה אנונימית ישירות בתוך Map, use או run - מתאים למצבים בהם הלוגיקה קטנה מאוד.
- או לחלופין, מחלקה חדשה – כפי שיוסבר בהמשך.

אנו נדגים את 2 האפשרויות באמצעות הדרישה הבאה :

עמוד 7 מתוך 11

- הדרישה : יש לכתוב middleware שמקבל קריאה שבתוך ה-url מופיע השם של המשתמש, ואנחנו ננתב את הקריאה לפי שם המשתמש, כאילו שם המשתמש הוא המזהה היחודי שלו (ה-id שלו ב-database).

איך ניגש לפתרון הדרישה ?

עלינו לכתוב את הלוגיקה, בצורה שמחלצת את שם המשתמש, והופכת אותו למספר ה-id של המשתמש.

נניח לצורך כך שיש לנו רשימה מסודרת של המשתמשים, ב- List שנקרא users.

הקוד הבא מדגים כיצד לעשות זאת באמצעות פונקציה אנונימית, ישירות בתוך ה-use :

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, IUsers users)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();

    app.Use(async (context, next) =>
    {
        if (context.Request.Path.Value.Length > 1 && context.Request.Path.Value.IndexOf('/', 1) < 0)
        {
            // Single segment path
            var path = context.Request.Path.Value.Substring(1);
            var user = await users.WithName(path.Replace("-", " "));
            if (user != null)
            {
                context.Request.Path = "/users/" + user.Id;
            }
        }
        await next();
    });

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

ועכשיו נעביר את הלוגיקה הזאת, אל middleware במחלקה נפרדת, על מנת להדגים כיצד כותבים זאת במחלקה.

בעיקרון זה רק מחלקה פשוטה (אין דרישה לממש אינטרפייס מסויים, או לרשת ממחלקה מסויימת)

הדרישות היחידות הן לבצע 2 דברים במחלקה:

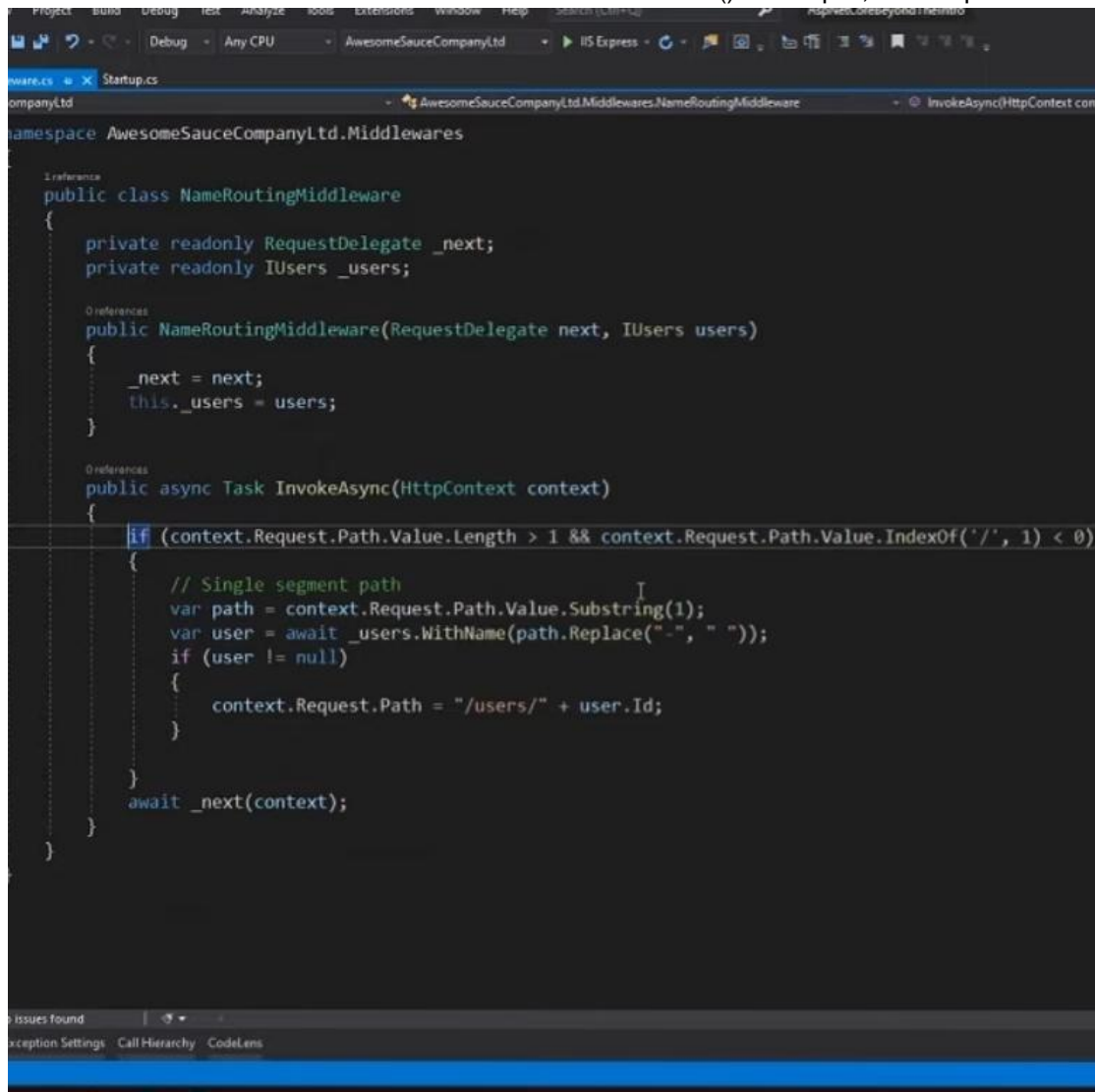
- פונקציית הקונסטרקטור , צריכה לקבל אובייקט מסוג RequestDelegate שיכיל למעשה את הפונקציה הבאה בתור , כלומר זהו פרמטר ה- next שלנו.

עמוד 8 מתוך 11

- למחלקה צריכה להיות פונקציה שנקראת `InvokeAsync` שמקבלת את ה- `HttpContext` , ובסופה - קוראת ל- `Next` . כלומר ממשיכה את השרשרת, את ה- `pipeline`.

להלן דוגמא לאותה לוגיקה שהוצגה לפני כן,
אך הפעם - בתוך מחלקה

שימוש לב לקונסטרטור, ולקריאה ל- `next.Invoke()` :



```
namespace AwesomeSauceCompanyLtd.Middlewares
{
    public class NameRoutingMiddleware
    {
        private readonly RequestDelegate _next;
        private readonly IUsers _users;

        public NameRoutingMiddleware(RequestDelegate next, IUsers users)
        {
            _next = next;
            this._users = users;
        }

        public async Task InvokeAsync(HttpContext context)
        {
            if (context.Request.Path.Value.Length > 1 && context.Request.Path.Value.IndexOf('/', 1) < 0)
            {
                // Single segment path
                var path = context.Request.Path.Value.Substring(1);
                var user = await _users.WithName(path.Replace("-", " "));
                if (user != null)
                {
                    context.Request.Path = "/users/" + user.Id;
                }
            }
            await _next(context);
        }
    }
}
```

במחלקה - בניגוד לפונקציה אנונומית, יש צורך "לקשור" את המחלקה אל השרשרת, אל ה- `pipeline`.

ובכדי לעשות זאת, נצטרך להוסיף `Use/Map/Run` בתוך פונקציית `Configure` בקובץ `Startup.cs`

להלן המחשה על המחלקה שבנינו כעת (שורה 30) :


```

20
21 public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
22 {
23     if (env.IsDevelopment())
24     {
25         app.UseDeveloperExceptionPage();
26     }
27
28     app.UseStaticFiles();
29
30     app.UseMiddleware<NameRoutingMiddleware>();
31
32     app.UseRouting();
33
34     app.UseEndpoints(endpoints =>
35     {
36         endpoints.MapControllers();
37     });
38 }
39
40
41

```

סיכום

כפי שלמדנו – Middlewares הם רכיבי קוד, שמתבצעים לפי סדר מסויים על ה- `HttpRequest`.

כל רכיב בשרשרת, יכול לבצע קוד, מסויים, ולהעביר את האובייקט אל הרכיב הבא בתור, או לקטוע את השרשרת, באמצעות החזרת תשובה ללקוח.

אריזה של MIDDLEWARES באמצעות EXTENSIONS METHODS

בקצרה על EXTENSION METHODS ב-C#

ב-C# ניתן לכתוב לכל אובייקט, פונקציית "הרחבה" בלי לעשות ירושה.

נניח שיש לכם אובייקט שחושף רק מתודות `A()`, `B()`, ואתם רוצים להוסיף לו מתודה `C()`, אבל אין לכם גישה ישירות לקוד המקור שלו.

אז באמצעות `Extension method` תוכלו לעשות זאת, וזה קל מאוד לביצוע.

זוהי יכולת חזקה מאוד של השפה, שמסוגלת במקרים רבים להפוך את הקוד שלכם להרבה יותר נקי, וקריא.

צורת הכתיבה של `Extension method` היא קלה מאוד.

פשוט כותבים פונקציה

- סטטית
- שמקבלת פרמטר `this` על ה-`type` אותו רוצים להרחיב.

זה הכל!

הפונקציה הבאה מוסיפה למחלקה **המובנית** string, את המתודה לספירת מילים בתוך המחרוזת. שימו לב, שהמתודה : 1. סטטית 2. מקבלת this string בתוך הפרמטר הראשון (והיחיד במקרה זה).

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

כיצד לעטוף MIDDLEWARES באמצעות EXTENSION METHODS

להלן דוגמא, בה כתבנו custom middleware ובסוף הקובץ, הוספנו extension method כדי להוסיף אותו בקלות אל ה-pipeline.

במקרה זה, בדוגמא אין לוגיקה עסקית כלל – ה-middleware הזה אינו עושה שום דבר, פרט לקבלת הריקווסט והעברה שלו הלאה.

```
namespace MyWeb.Middlewarees
{
    // You may need to install the
    // Microsoft.AspNetCore.Http.Abstractions package into your project
    public class CustomMiddleware
    {
        private readonly RequestDelegate _next;

        public CustomMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext httpContext)
        {
            // ... Your code come here
            // ... Your code come here
            // ... Your code come here
            await _next(httpContext);

            //... maybe other code...
        }
    }
}
```

```
// Extension method used to add the middleware to the HTTP
request pipeline.
public static class CustomMiddlewareExtensions
{
    public static IApplicationBuilder
        UseCustomMiddleware(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<CustomMiddleware>();
    }
}
```

תרגילים

1. כתוב custom middleware פשוט שמשתמש ב-run על מנת להחזיר "שלום עולם".
2. במכללה, ישנה אפליקציית web שמקבלת http requests , כאשר הוחלט שהאלמנט הראשון אחרי המילה students הוא מספר הכיתה, ואילו האלמנט השני הוא מספר הסטודנט למשל /students/3/51 - מתייחס לסטודנט מספר 51 בכיתה מספר 3
- עליך לכתוב 2 מידלוררים, באמצעות use , כאשר הראשון ב- pipeline מדפיס את מספר הכיתה, והשני מדפיס את מספר הסטודנט.
3. כעת, ארוז את מה שכתבת בתרגיל הקודם עם extensions methods
4. באותה מכללה הוחלט על טיפול מסוג שונה בכיתה 5 , כתוב מידלורר באמצעות map שממפנה את כל הניתובים של כיתה זו לפונקציה נפרדת.

מקורות להרחבה :

[טבלה ה Middlewares-המובנים.](#)