ASP.NET

שיעור 1

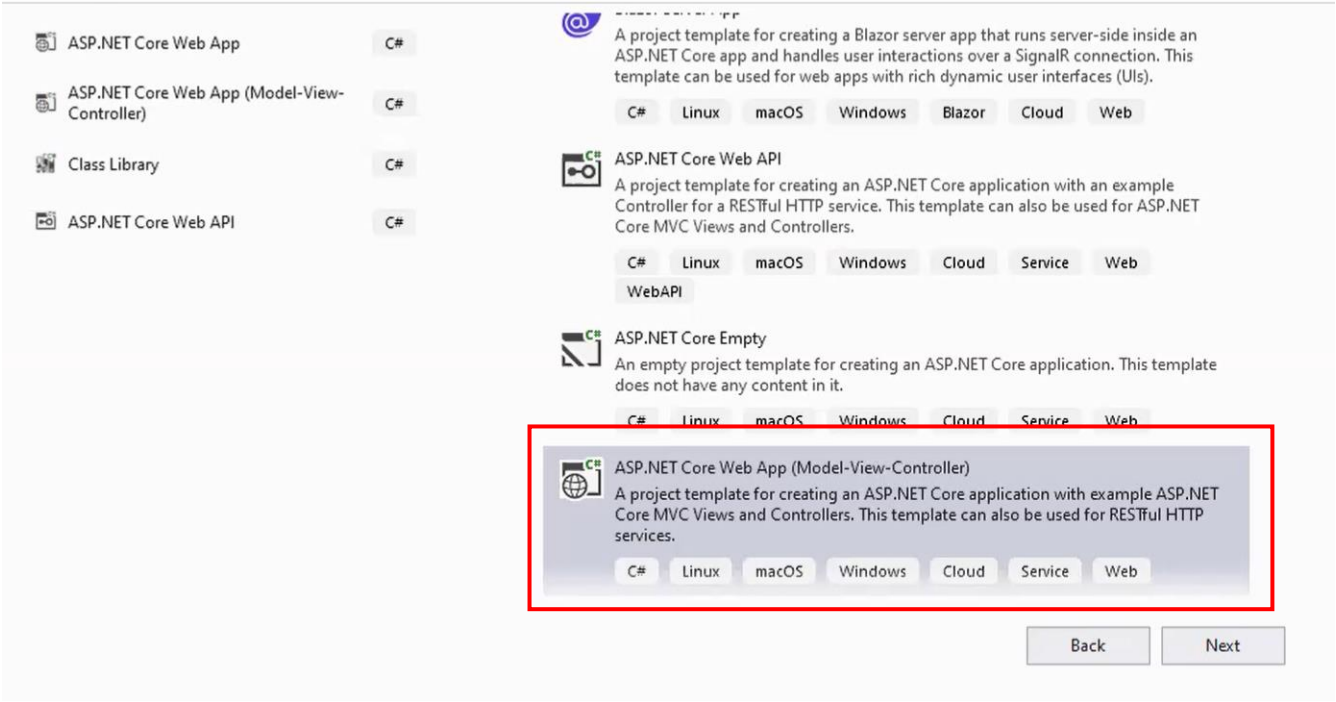
החלק הזה הוא בעצם עבודה של #C עם אתרים, שילוב של .NET. ב-web.

נזכיר ש-.NET. אינה שפה אלא מנוע וספריות שמריץ את השפות שהוא מתאים להן (כמו #C).

בכתיבת התכנית נוכל לשים לב שאנו משלבים בין כתיבה של HTML לבין כתיבה שאנו מכירים מ-#C, השילוב הזה נקרא razor.

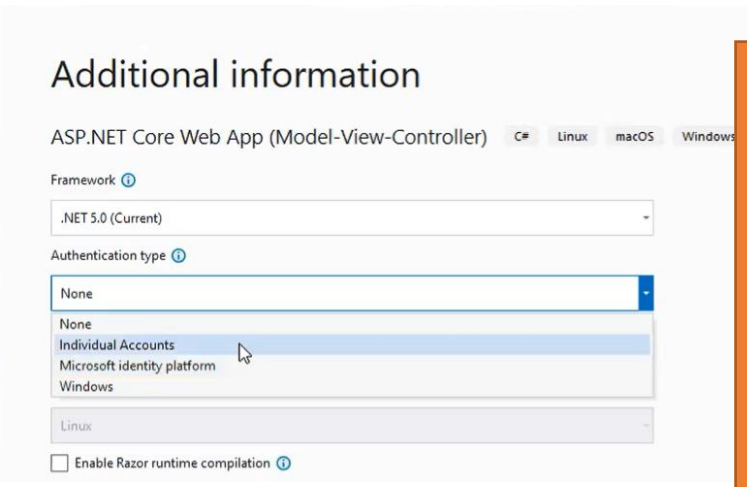
אנחנו נראה בשיעור הזה סוג אחד של פרויקט ובהמשך נעבור ל-API.

נבחר פרויקט:



לצורך הדוגמא בשיעור אנו בונים תכנית של בדיחות.

- רק לצורך הלמידה האחידה עבדנו ב-NET5.



Individual Accounts:

יצירת תכנית עם דרישת הזדהות רגילה כמו שאנו מכירים – כלומר מידע השמור במסד נתונים.

Microsoft identity platform:

הזדהות חיצונית באמצעות מייקרוסופט (כמו שיש הזדהות באמצעות ג'ימייל).

Windows:

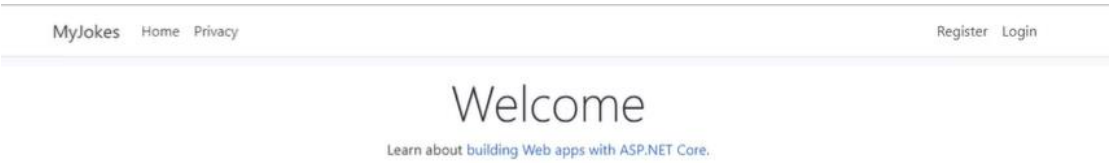
אפשרות שבה זיהוי המשתמש במערכת ההפעלה הוא אוטומטית הזיהוי לתכנית.

נשים לב שההרשאה שאנו בוחרים יוצרת לנו חתימה עצמית להצפנה (כמו המנעול הקטן שרואים ליד שם האתר) אבל חשוב לזכור שברגע שהאתר יעלה לאוויר באמת ההצפנה הזו לא תהיה תקפה אלא יהיה צורך להשתמש באחת בתשלום.

אנו בוחרים לפרויקט שלנו את Individual Accounts.

כאשר עבדנו עם פלטפורמות שונות היה לכל אחת את הדרך להגיע אל השרת המובנה שלה ולהריץ את התכנית. גם אצלנו יש שרת מובנה והוא (IIS Express) **I**nformation **I**nternet **S**erver.

כאשר נריץ (נכון עוד לא כתבנו כלום) נראה שכבר יש שם דף חי וקיים.



וזה כל מה שנראה בעמוד.

הקובץ שהוא שילוב בין HTML ל-C# בא עם סיומת cshtml.

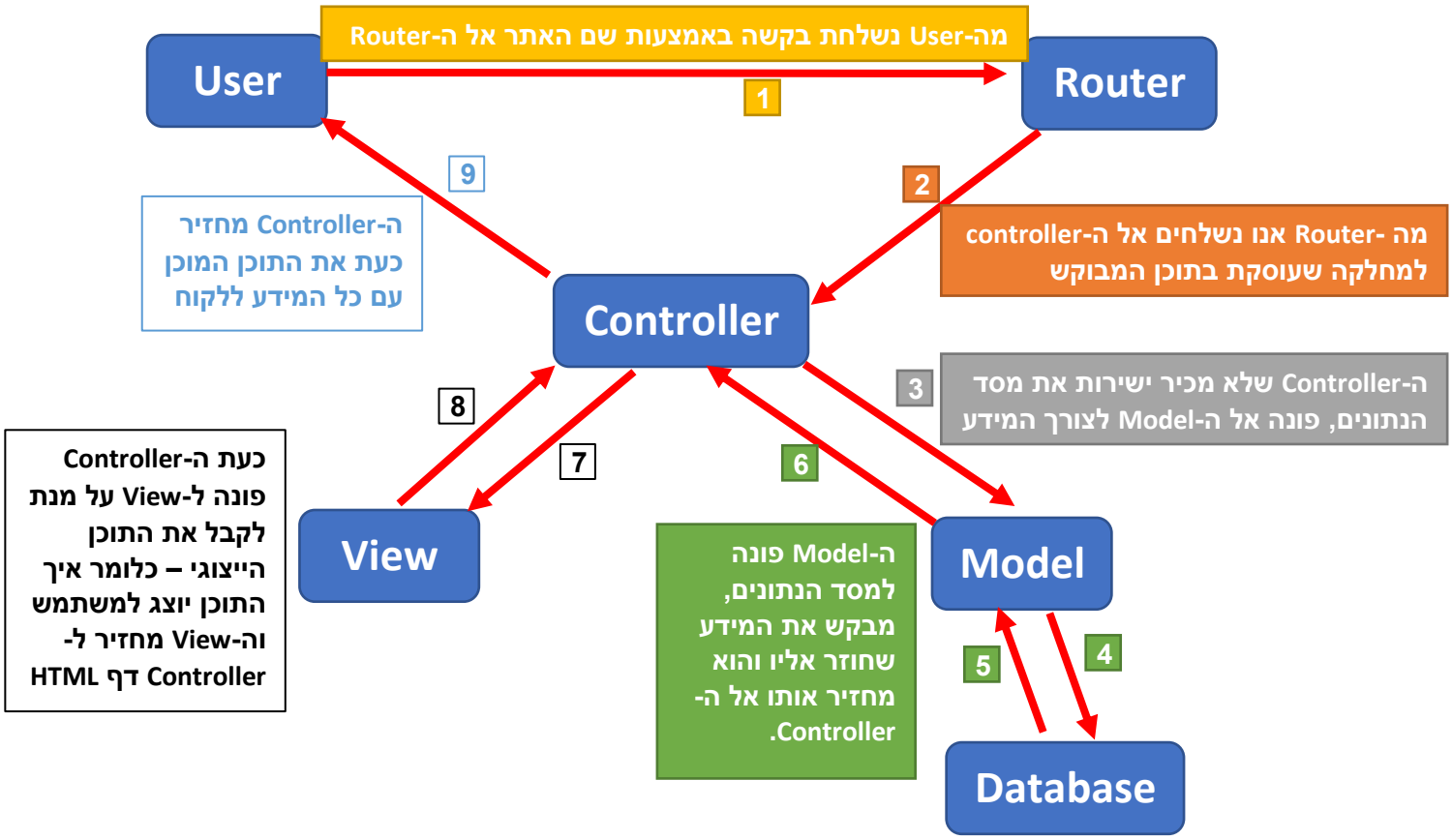
כעת נחזור\נלמד קצת על MVC:

האפליקציה שיצרנו כמו ששמנו היא מראש באה בתבנית הזו (תבנית עיצוב)

Model – ייצוג של המידע, נניח שלנו עוסקת בבדיחות אז תהיה מחלקה של בדיחה.

View – קבצים שמתארים את מה שהמשתמש יראה (קבצי cshtml)

Control – יחידה שמחברת בין ה-Model לבין ה-View ושם תיכתב כל הלוגיקה העסקית.



כעת נסתכל במה שנוצר לנו עם הפרויקט:

ב-program שזה הדף הראשי כביכול, נראה את הקוד הבא:

```
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>();
            });
}
```

שזו בעצם הפעלה של המארח באמצעות הפונקציה שמשתמשת ב-startup שזו מחלקה שמגדירה את המאחורה של העמוד.

ב-Startup נראה את הבנאי שמכניס למשתנה בשם configuration את כל ההגדרות באופן אוטומטי.

בנוסף נראה שם פונקציה אחת שמגדירה סרוויסים שאפשר להזריק ועוד פונקציה שדואגת לסדר של הסרוויסים.

עם הפעלת התוכנה נוצרו ספריות:

Models – כרגע ריקה חוץ מקובץ של תקלות אבל לשם נוסיף מודלים.

Views – מחזיקה את התצוגות (ה-html).

Controller – מחזיקה קבצים שמחזיקים את ה-controller שם יש מחלקה שמפעילה הכל:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }

    public IActionResult Index()
    {
        return View();
    }

    public IActionResult Privacy()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

כאשר יש כאן שתי פונקציות שכדאי להבין שהן Index() ו-Privacy() שבשתיהן ניתן להבין שקוראות להציג את העמוד לפי קריאה (המתודה Index תקרא לעמוד של קובץ ה-HTML שנקרא Index)

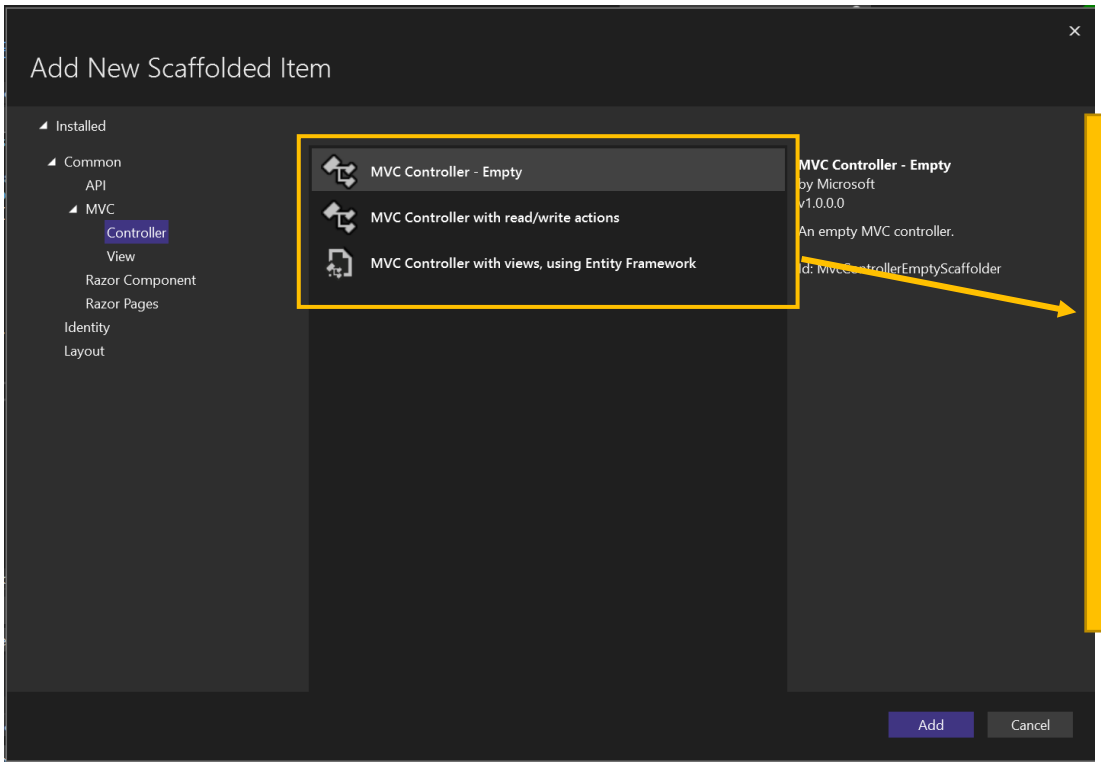
אמרנו שבדוגמא שלנו אנו בונים תכנית של בדיחות ולכן נוסיף מודל שנקרא Joke.

(קוד בעמוד הבא)

```
public class Joke
{
    public int Id { get; set; }
    public string JokeQuestion { get; set; }
    public string JokeAnswer { get; set; }
}
```

כעת, מקש ימני על הספרייה Controllers ואז אפשרות 'Add' ואז ללחוץ על '..Controller'.

נקבל את האפשרות הבאה:



הראשונה:

רק יוצר את הקובץ

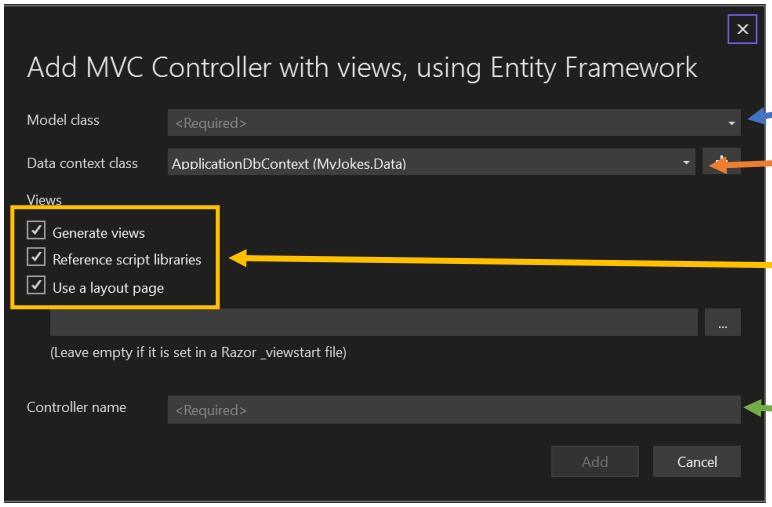
השנייה:

אפשרות רק עם אופציה ליצור או לקרוא תוכן.

השלישית:

שם נבחר רק את המודל שמייצג טבלה במסד הנתונים ואת שם ה-controller והוא כבר יעשה את כל החיבורים.

נבחר את השלישית ונקבל את החלונית הבאה:



בחירה של מודל

בחירה של מסד נתונים

יוסיף לנו לבד את כל הקבצים שנצטרך כולל קבצי ה-HTML

בחירה של שם ל-Controller

נוצר לנו כעת controller בשם JokeController, ובו פונקציות של כל הפעולות האפשריות הבסיסיות כמו יצירה, מחיקה עריכה ומעבר לעמוד המתאים.

גם בספרייה Views נוצרו קבצים המתאימים לכל פעולה.

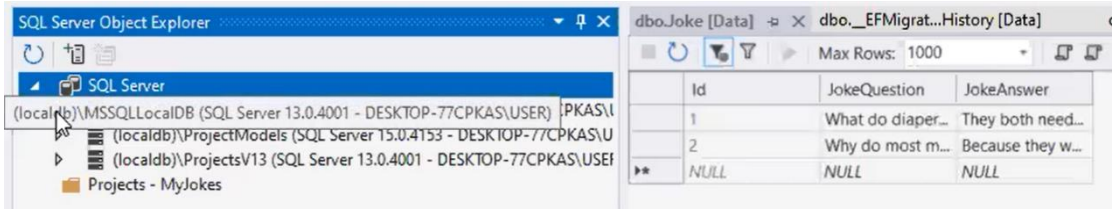
בספרייה הראשית של הפרויקט (כלומר לא באף תיקייה), יש קובץ שנקרא appsetting.json

בקובץ הזה נראה שיש לנו כבר נתון connection string שהמערכת יצרה אוטומטית.

כדי שהמידע שאנו מכניסים ייהפך למסד נתונים אמיתי נריץ מיגרציה ונעדכן את מסד הנתונים – כל זאת באמצעות ה-package manager console.

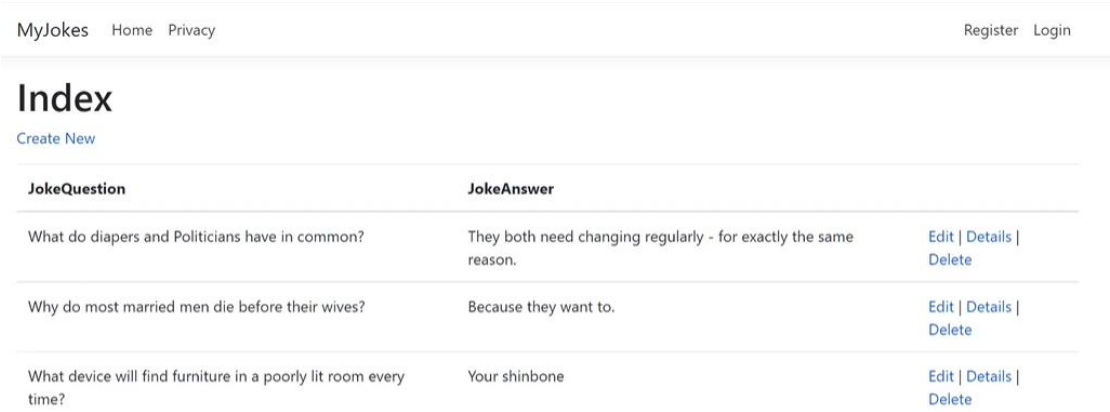
נדגיש כמובן שהוא יוצר מסד נתונים ברקע שהוא מעין מסד נתונים קטן לצורכי פיתוח.

כעת הוספנו ידנית במסד הנתונים שתי בדיחות:



כמובן, נזכור שאין צורך לפתוח את SQL אלא אפשר להשתמש ב-SQL Server Object Explorer שהוא כלי ב-#C.

ואם נעבור ב-path ל-jokes כלומר נוסיף בכתובת האתר '/jokes' נראה שהמערכת יצרה בעצמה טבלה שבה הוצגו הבדיחות:



אפילו נוצרו גם ה'כפתורים' של יצירת חדש, עריכה, פרטים ומחיקה.

כעת ניתן להכנס בספרייה Views לתוך תיקיית Shared שם יש קובץ שנקרא Layout, הוא קובץ HTML שבו ניתן להוסיף דברים לעמוד הראשי.

הוספנו שם לחלק העליון של העמוד (header) עוד פריט לניווט וקראנו לו Jokes.

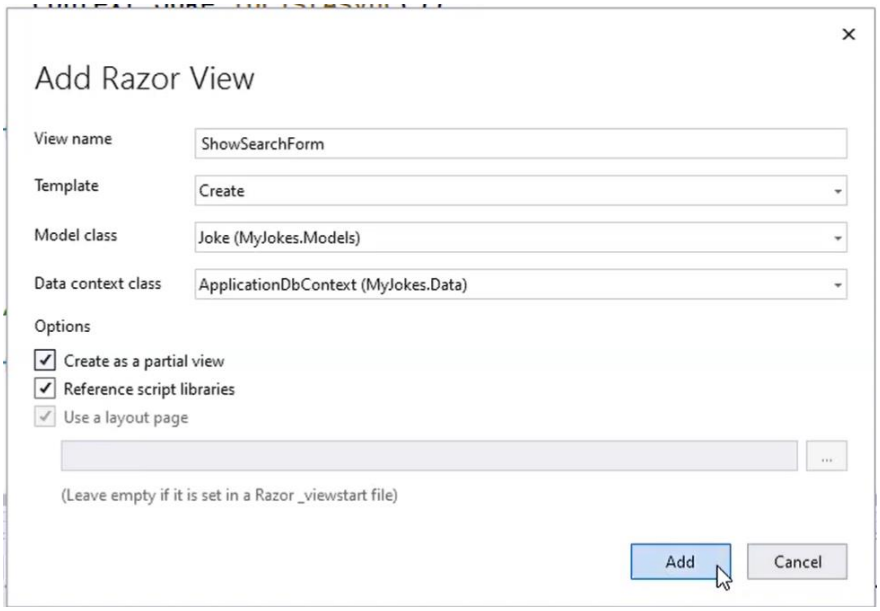
- סקירה קטנה כללית על הקובץ Layout – הוא בעצם נותן לנו את החלק של ה-header וה-footer ובחלק של ה-body הוא פשוט קורא לפונקציה ()RenderBody שבעצם בונה את ה-body מכל מה שחוזר מה-View.

בקובץ שנוצר עם יצירת ה-controller (JokeController), ישנן כל המתודות שנצטרך כאשר מתודת GET אחת קיימת שם והיא מובילה אותנו לעמוד עם תצוגת הבדיחות.

כעת יצרנו חלק נוסף ב-header, והוא אפשרות חיפוש בדיחה וב-JokeController הוספנו לו מתודה של GET שתוביל לעמוד המתאים.

כעת המערכת לא מכירה את הפונקציה הזו, אז יש טריק שעושים – מקש ימני על המתודה (קראנו לה ()ShowSearchForm) ואז 'Add View'.

יופיע לנו חלון ליצירת ה-View, שם ניתן שם ל-view, ובגלל שזה יהיה טופס ליצירת בדיחה חדשה בנוסף לחיפוש אז נשתמש ב-template של create, ובמודל בחרנו כמובן את joke.



זה יצר לנו את ה-view וכעת נשנה את זה לצורך שלנו משום שכרגע זה טופס פשוט ליצירת בדיחה.

```
<h4>Joke</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="ShowSearchForm">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="JokeQuestion" class="control-label"></label>
                <input asp-for="JokeQuestion" class="form-control" />
                <span asp-validation-for="JokeQuestion" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="JokeAnswer" class="control-label"></label>
                <input asp-for="JokeAnswer" class="form-control" />
                <span asp-validation-for="JokeAnswer" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>
```

לפני

הכותרת כמובן תשתנה למשהו שקשור לחיפוש.

בתחילת הטופס הייתה הפנייה למודל שאותה מחקנו ולכן נתאים את הטופס בהתאם.

הורדנו את האפשרות של תשובה לבדיחה והשארנו רק את השאלה, בנוסף שינינו את האפשרויות שהיו קשורות ל-asp לאפשרויות רגילות של HTML והורדנו את הואלידציה משום שאין לי בה צורך בחיפוש (אם מה שנכתב לא קיים המערכת פשוט לא תמצא – אין צורך בואלידציה).

```
<h4>Search some joke</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="ShowSearchForm">

            <div class="form-group">
                <label for="searchTerm"  class="control-label"></label>
                <input name="searchTerm" class="form-control" />

            </div>

            <div class="form-group">
                <input type="submit" value="Search" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>
```

אחרי

מכאן אפשר כמובן להמשיך ולשחק עם העמוד כאוות נפשנו.

לצורך ההסבר הבא, נזכור שכאשר נכתוב בפונקציה שקוראת לעמוד שתחזיר את המתודה ()view אז בברירת המחדל ההנחה היא קריאת GET.

בנוסף נזכיר גם שבקובץ של החיפוש שראינו בתמונות כאן למעלה, הנחת המוצא של המערכת היא שימוש במתודה POST והפונקציה שהיא תפנה אליה ל-POST היא זו שכתובה בחלק של asp-action.

זה POST ולא ב-GET כי אני כותב משהו בחיפוש ושולח אותו ואז עם המידע שנכנס מתבצעות פעולות וחוזר אליי מה שצריך.

```
[HttpPost]
0 references
public async Task<IActionResult> ShowSearchForm(string searchTerm)
{
    return View("Index", await _context.Joke.Where(j => j.JokeQuestion.Contains(searchTerm)).ToListAsync());
}
```

אז בתמונה אנו רואים את הפונקציה החדשה שנוסיף כדי לשלוח את קריאת החיפוש.

הפונקציה מקבלת כארגומנט את מילות החיפוש שהכנסנו בתיבה, ושולחת אותנו לעמוד הראשי כאשר בשאילתה אנו מחזירים רק את הבדיחות המכילות את הטקסט המבוקש.

כמובן אנו מסתמכים על הפונקציה הקודמת שמחזירה אותנו ל-Index שם הצגנו את המידע כולו ללא סינון.

- המתודות בנויות כך שמחזירות Task – זה משום שבסופו של דבר יכול להיות שכמה קריאות לעמוד מגיעות במקביל ואם לא נמקבל אותן במשימות תחזור שגיאה משום שכל אחת תעצור את ה-thread הראשי.

כעת, נרצה להסתיר את התשובה של הבדיחה כך שנצטרך ללחוץ על כפתור שיוביל לתשובה.

במקרה זה, בדף ה-Index פשוט נמחק את החלק של הטבלה שמכיל את התשובה ובעצם כדי להגיע אליה צריך ללחוץ על Details.

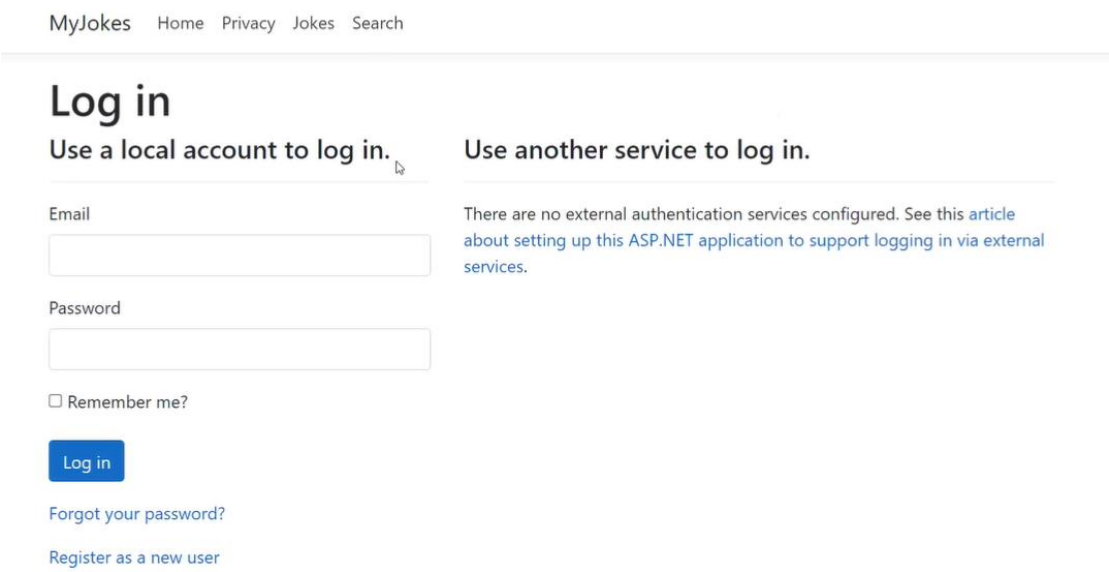
- נשים לב שבראש דף הקוד של עמוד, יש הגדרת משתנה שמחזיק תוכן, כמו שבעמוד ה-Index אנו מחזיקים משתנה בשם מודל שמחזיק מידע שמגיע ממסד הנתונים.

עכשיו נעבוד קצת על אפשרות של הגנה כלשהי (כמו שבחרנו בהתחלה בהגדרות – individual account).

מעל לפונקציה ()ShowSearchForm הראשית, כלומר זו שמובילה לעמוד החיפוש הראשי ולא זו שמחזירה את התוכן שחיפשנו, נוסיף attribute שנקרא [Authorize].

```
[Authorize]
0 references
public async Task<IActionResult> ShowSearchForm()
{
    return View();
}
```

וזה מה שנקבל כאשר נלחץ על search:



כמובן שניאלץ להירשם, ולאחר מכן מכיוון שזו לא אפליקציה עם רישיון אז אישור במייל הוא בהדמיה ולא אמיתי.

דברנו קצת ממש על coockies:

מה זה בעצם coockies?

זו אפשרות לשמור קטע קטן של מידע בצורה מוצפנת על הדפדפן עבור הלקוח, כלומר, אפשר לבקש מהדפדפן לשמור פריט מידע מסוים כך שכל פעם שניכנס לאתר הזה, אם קיים coockie עבוד האתר הספציפי הזה, אז לכל אח מקריאות ה-HTTP שיוצאות החוצה הוא מצרף את ה-coockie הזה.

השוני בין coockie לבין local storage הוא שזה אוטומטי וכולל הצפנה.

שיעור 2

החל מהשיעור הזה נעבוד בתצורה של WEB API, ולא בפורמט שראינו בשיעור קודם.

למה אנחנו צריכים את WEB API?

אז נניח שיש לנו כמה שכבות של נתונים. יש לנו מסד נתונים ושרת שמתחבר אליו ויודע לשלוף נתונים. בנוסף, נניח שאנחנו מגישים את הנתונים בכמה צורות כמו אפליקציית אנדרואיד, IOS ואתר – האם נצטרך לכתוב את כל הלוגיקה שוב ושוב? האם נחבר את המערכת שלנו כולה למסד הנתונים? זה לא כל כך בטוח למידע...

אז הבעיות שלנו הן:

- שכפול של קוד לשפות שונות.
- ריבוי קוד = חשיפה לריבוי תקלות.
- חלק מהשפות לא מסוגלות לפנות ישירות למסד הנתונים.
- קשה לתחזוקה.

פתרון:

בניית שכבה שאחראית מצד אחד לקחת מידע ממסד הנתונים כולל מניפולציות שצריך לעשות המידע – וזה בעצם שכבת ה-API.

כך בעצם אנו משאירים את הפרונט "טיפש" ואת הלוגיקה משאירים ל-back.

Application **P**rogramming **I**nterface

זה בעצם ממשק או צורת התחברות תכנותית שזה אומר שיש צורה כלשהי באמצעות תכנות להתחבר לאובייקט שלי (יכול להיות גם רובוט שמנקה את הבית).

אנחנו נראה איך עושים API ל-WEB.

אז כמה מונחים:

Hyper **T**ext **T**ransfer **P**rotocol (HTTP):

פרוטוקול להעברת טקסט – מיועד לתקשר בין שרת ללקוח, כאשר שרת יכול להיות מחשב שמחכה לקריאה ולקוח יכול להיות משתמש באתר או אפילו מכשיר חכם שיכול להתחבר לשרת.

REpresentation State Transfer (REST):

זוהי צורה לייצג מצב מסוים בשרת, לדוגמא אם יש לי פרטים של לקוח ואני רוצה לראות אותם וגם לשנות אותם. לשם כך אני מבקש מהשרת שיחזיר לי מה ה-state הנוכחי של הלקוח כמו הכתובת או המייל ואני רוצה לשנות אז אני שולח פקודה שמבצעת את זה.

ב-REST אנו משלבים את הכתובת שאליה פונים, את הפעולה (GET, POST etc.), את הקודים שקיבלנו בחזרה (כמו קוד 404 שזה עמוד לא נמצא) ואת stateless.

מה זה stateless?

כשאנחנו מדברים על קריאות HTTP שפונות לשרת ומביאות סטייט מסוים, כל קריאה כשלעצמה מתחילה בנקודה מסוימת ומחזירה סטייט מסוים.

אבל כל קריאת HTTP מחזירה את הסטייט ולא מחזיקה אותו, כלומר אם קראנו לשרת והוא החזיר לי עמוד מסוים, שם הקשר שלו עם הסטייט נגמר והיא מחכה לקריאה הבאה.

פרטי בקשת HTTP:

- הכתובת.
- הפועל (GET, POST etc)
- Headers
- Body – יכול להיות בהרבה צורות. (נקרא data או payload)

רכיבי התשובה המתקבלת:

- סטטוס.
- Body
- Headers

קוד סטטוס:

כל קוד הוא מספר המייצג מצב מסוים.

קוד שמתחיל ב-1: מידע כללי, כמעט לא רואים את זה כי זה לרוב מידע לדפדפן ולא לנו.

קוד שמתחיל ב-2: קוד הצלחה, כלומר ייצוג שפעולה מסוימת הצליחה.

קוד שמתחיל ב-3: אמירה לדפדפן שהמידע אינו קיים בכתובת מסוימת אלא בכתובת אחרת ושולחת אותו לשם (מעין עקוב אחרי).

קוד שמתחיל ב-4: משהו אצלי (הלקוח) לא תקין – המוכר ביותר הוא 404 שהוא עמוד לא קיים.

קוד שמתחיל ב-5: משהו בשרת לא עובד.

לצורך העבודה הפעם נבחר את האפשרות הבאה:

אם נריץ את התכנית גם מבלי לעשות כלום, נקבל עמוד שנקרא swagger ודרכו ניתן להגיע למידע השמור במערך בקובץ JSON.

דוגמא ל-headers שאנו מקבלים בחזרה כאשר ניקח את הכתובת פלוס שם מס הנתונים ונכתוב בפוסטמן:

Body | Cookies | Headers (5) | Test Results | Status: 200 OK

KEY	VALUE
Transfer-Encoding	chunked
Content-Type	application/json; charset=utf-8
Server	Microsoft-IIS/10.0
X-Powered-By	ASP.NET
Date	Thu, 12 May 2022 15:25:50 GMT

- נדגיש שכל החלק הזה לא אמור להיות כשנבנה את האתר, זה רק מצב ברירת מחדל שנמצא בינתיים עד שנתכנת.

אז נתחיל מלבדוק באמצעות הפקודה dotnet –version ב-CMD לבדוק שדוטנט מותקן לנו על המחשב ובאיזו גרסה.

על מנת לפתוח פרויקט API חדש גם באמצעות ה-CMD נכתוב:

dotnet new webapi –name <projName>

אפשר גם להריץ את התוכנה באמצעות הפקודה dotnet run

נקבל את האפשרויות הבאות:

```
C:\Users\USER\source\repos\api333>dotnet run
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7023
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5058
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\USER\source\repos\api333\
```

מה שנעשה כעת הוא חשוב אך כנראה שלא נעבוד בצורה הזו:

נראה איך אנחנו פותחים פרויקט קונסול רגיל והופכים אותו ל-WEB API.

אז אפשר להכנס למאפיינים של הפרויקט ואפשר ללחוץ פעם אחת על הפרויקט וזה יפתח לנו קובץ cdproj.

```
consoleweb.csproj   Program.cs
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

אז קודם אנו מוחקים את השורה של OutputType.

ב-Sdk נוסיף 'Web.' .

```
consoleweb.csproj   Program.cs
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

כעת נוסיף שני קבצים:

<u>הקובץ program.cs קיים לנו ונעשה בו שינויים:</u>

בדוט נט יש דבר כזה שנקרא מארח, הוא בעצם עוטף את האפליקציה ודואג כל הזמן שהיא תרוץ כל עוד אנו עובדים איתה.

המארח בנוי בתצורה של תבנית עיצוב מסוג builder – שזה אומר שרשור פקודות לבניית אובייקט ובסוף מפעילים איזשהי פונקציית build שהיא בונה את האובייקט.

אז המארח הוא אובייקט שמוסיף תכונות לאפליקציה.

אנו נשתמש במתודה CreateDefaultBuilder() והיא תיתן לנו כמה דברים:

- Dependence Injection – היכולת להזריק תלויות.
- איסוף קונפיגורציות מכמה מקומות כמו appsetting, קריאת משתנים סביבתיים מהמערכת, קריאת פרמטרים משורת הפקודה (לדוגמא היא תדע לקרוא שינוי בפורט ולשנות באפליקציה.
- Logging
- מגדירה את התיקייה הנוכחית כברירת המחדל של האפליקציה.

```
internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
        // קריאה לפונקציה שמארחת את כל האפליקציה בתוכה והפעלה של המארח
        CreateHostBuilder(args).Build().Run();
    }


    public static IHostBuilder CreateHostBuilder(string[] args)
    {
        return Host.CreateDefaultBuilder()
            .ConfigureWebHostDefaults(webHost =>
            {
                webHost.UseStartup<Startup>();
            });
    }
}
```

אנו רואים שההבניה נעשית בשלבים בפונקציה CreateHostBuilder().

בפונקציה Main אנו רואים את הקריאה ל-build ול-run.

אז באופן כללי, מה המטרה של הקובץ program? המטרה היא ליצור מארח, את היכולת שהאפליקציה תעבוד ללא הפסקה ותאזין לקריאות מהאינטרנט.

<u>הקובץ השני שניצור, הוא ה-Startup:</u>

לקובץ הזה, נשים לב, קראנו כבר בפונקציה שבנינו ב-program.

בקובץ הזה נוסיף שתי פונקציות: Configure ו-ConfigureServices.

ברגע שהמתודה UseStartup פועלת, היא קוראת לשתי הפונקציות.

Endpoint – נקודת גישה. קביעת נקודת גישה מתבצעת ב-Configure.

```
public class Startup
{
    //נגדיר פונקציה שתכיל את כל השירותים = סרוויסים בהם נשתמש
    //זה רק מקום לשמור בו את הסרוויסים

    public void ConfigureServices(IServiceCollection services)
    {

    }

    //נגדיר פונקציה שקובעת את סדר המידלוורים שרצים
    //בעצם כאן המידלוורים מתלבשים על הצינור == פייפליין
    //ומתחילים לעבוד

    public void Configure(IApplicationBuilder app, IWebHostEnvironment environment)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Hello World");
            });
        });
    }
}
```

כעת נחזור לדבר על פרויקט רגיל שפתחנו מראש כ-WEB API.

בתיקיית properties יש קובץ שנקרא launchSettings.json ובו כתובים כל מיני פרמטרים שניתן לערוך או בקובץ עצמו או במאפיינים של הפרויקט => Debug => General => open debug launch profiles UI.

בקובץ מפורטים פרופילים שונים, כאשר ניתן להריץ את התוכנית כל פעם לפי פרופיל אחר וזה כמובן ינותב על ידי פורט אחר.

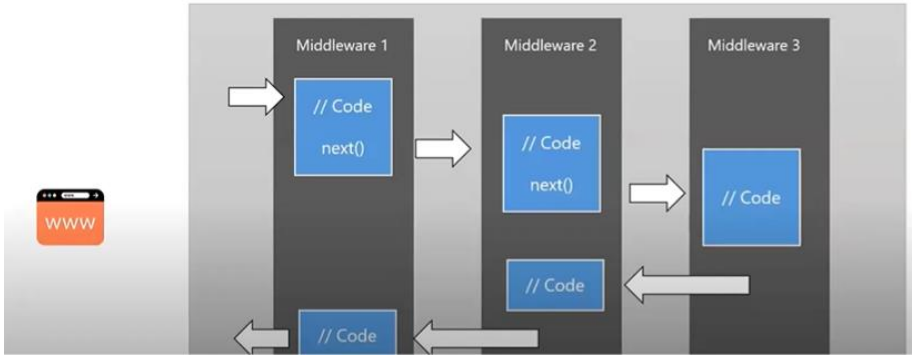
נשים לב לכמה פרטים נוספים:

בקובץ Startup יש הוספת סרוויס שנקרא AddControllers() שבעצם קורא את הקונטרולרים ומסדר אותם.

פונקציה נוספת שקיימת היא mapController() שהיא יוצרת מצב שכל קונטרולר מקבל שם שיהיה ה-endpoint שלו.

מידלוורים:

כל request עובר קודם כל ל-pipeline ולא מתבצע ניתוב ישיר, שם בצינור הזה הוא עובר דרך הראשון ואז השני ואז השלישי ואז חוזר באותה הדרך (בהנחה כמובן שיש שלושה).



לפי התרשים שבתמונה נשים לב שמתבצע מעבר למידלוור הבא באמצעות `next()` עוד לפני שהסתיים הקוד כך שלאחר שהגענו למידלוור השלישי חזרנו אחורה באותו הסדר שהגענו (רק הפוך) והמשכנו לבצע את הקוד הנותר בכל מידלוור שעברנו דרכו.

- אין הגבלה על כמות המידלוורים שניתן לשרשר.
- יש חשיבות ומשמעות לסדר המידלוורים.
- אם המידלוור מפסיק ולא קרא להבא בתור אז תתחיל החזרה אחורה.

<u>דוגמאות נפוצות למידלוורים:</u>

- ראוטינג – ניתוב קריאות.
- אותנטיקציה – זיהוי משתמש.
- עמוד שגיאה.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello From Run Middleware!");
    });
```

אם נוסיף מידלוור כמתואר בתמונה באמצעות `Run()`, המערכת תבצע את הקוד של המידלוור הזה ולא תמשיך לאחר, בנוסף לא משנה איזה פורט נכתוב בשורת הקריאה, זה עדיין יישאר על המידלוור הזה.

בכללי ישנן שלוש פעולות לעבודה עם מידלוורים והן: `Run()`, `Use()`, `Map()`

נרחיב עליהן בשיעור הבא.

שיעור 3

אז ראינו את המתודות לעבודה עם מידלוורים:

Run() – מבצע את המידלוור ועוצר את הפייפליין.

Use() – מלביש את המידלוור ויש אפשרות להמשיך למידלוור הבא.

Map() – קצת מיוחדת יותר, כאשר מלבישים אותה היא מכניסה מידלוורים לנתיב מסוים בלבד.

כמו שאמרנו בשיעור קודם, אם נשתמש ב-Run() בתחילת הפונקציה Configure() אז המילדוור הזה יתבצע וכל מה שאחריו לא יתבצע, המשמעות היא שהמערכת לא תמשיך אפילו לקונטרולרים. (דוג' ניתן לראות בסוף השיעור הקודם)

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello From Use Middleware \n");
    await next();
    await context.Response.WriteAsync("Hello From Use Middleware - After next() \n");
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello From Run Middleware \n");
});
```

בדוגמא הזו אנו רואים שהמשכנו למידלוור הבא באמצעות השימוש ב-Next(), הגענו למידלוור Run(), ביצענו אותו וחזרנו למידלוור לפי הסדר (כמו בתמונה בשיעור קודם).

נזכיר שאם לא נוסיף Next(), אז המתודה Use() תפעל בדיוק כמו Run().

נראה עכשיו דוגמא ל-Map():

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello From Use1 Middleware - A\n");
    await next();
    await context.Response.WriteAsync("Hello From Use1 Middleware - B \n");
});

app.Map("/gil", HandleGilRoute);
```

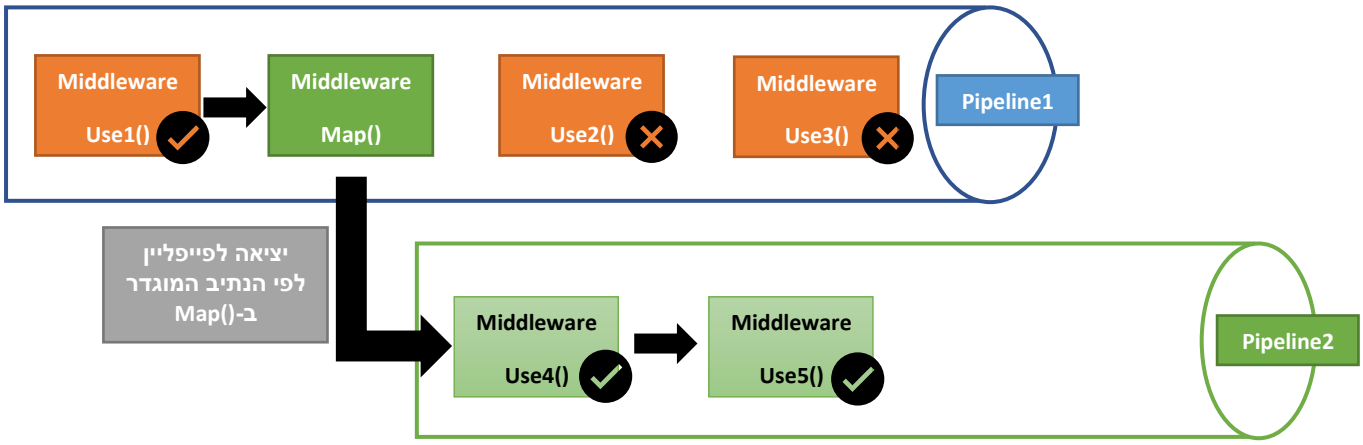
קבענו ל-Map() את הנתיב בה היא פועלת.

כעת אם אנו לא שם, המידלוורים יפעלו לפי הסדר הכתוב, אבל אם אנחנו בנתיב הזה אז המתודה המוגדרת כארגומנט (HandleGilRoute) תפעל.

לצורך הדוגמא זו המתודה:

```
private void HandleGilRoute(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Hello From Use2 Middleware - A\n");
    });
}
```

כך בעצם ה-Map() מפנה אותנו לנתיב אחר של מידלוורים.



בתרשים כאן אנו רואים את המקרה שבו המתודה `Map()` מפנה אותנו לנתיב אחר שהמידלוורים שלו הם אלו שיתבצעו בהמשך (יתבצעו 1, 4, 5 בתרשים – המסומנים ב- ✓).

כמובן שאם לא היינו בנתיב שהוגדר ב-`Map()` אז היינו ממשיכים עם המידלוורים שבפייפליין הראשון – כלומר מידלוורים 2 ו-3.

כעת נראה אפשרות של כתיבת מידלוור ב-`class`:

ניצור מחלקה ולצורך הדוגמא נקרא לה `CustomMiddleware`.

יש תנאי למידלוור כזה שהוא חייב לממש אינטרפייס `IMiddleware` של מייקרוסופט.

```
public class CustomMiddleware : IMiddleware
{
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        await context.Response.WriteAsync("Hello From Middleware in class - C 1\n");
        await next(context);
        await context.Response.WriteAsync("Hello From Middleware in class - C 2 \n");
    }
}
```

<u>ואז פשוט נשתמש בזה בצורה הבאה:</u>

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello From Use1 Middleware - A\n");
    await next();
    await context.Response.WriteAsync("Hello From Use1 Middleware - B \n");
});

app.UseMiddleware<CustomMiddleware>();
```

ואז בעצם מה שיקרה זה שהמערכת תבצע את המתודה שכתובה בתוך המחלקה ומבצעת אותה.

בהמשך הפונקציה `Configure` יש לנו קריאה (שנעשית אוטומטית) ל-`UseRouting` ול-`MapController` שבעצם דואגות לנתב את כל הקונטרולרים שיהיו בפרויקט.

אז יצרנו קונטרולר משלנו ובעצם הורדנו ממנו את ה-`route` הראשי וניתבנו כל מתודה לנתיב משלה.

את הניתוב עשינו באמצעות שימוש ב-`attribute` של `route` מעל לכל מתודה בנפרד.

בנוסף ראינו אפשרות לשימוש בנתיב בסוגריים מסולסלים כך שהנתיב ייקבע בהתאם לתוכן שיוכנס לפונקציה.

```
[ApiController]
public class ValuesController : ControllerBase
{
    [Route("api/get-all-authors")]
    public string GetAllAuthors()
    {
        return "Get all authors";
    }

    [Route("api/book/{id}")]
    public string GetBook(int id)
    {
        return "Book " + id;
    }
}
```

ישנה אפשרות להוסיף עוד ניתוב שיקבל מידע מעוד משתנה, האפשרות הזו נקראת routeParams:

```
[Route("api/book/{id}/{name}")]
public string GetBook(int id, string name)
{
    return "Book : Id : " + id + ", Name : " + name;
}
```

<u>אפשרות מורחבת יותר תהיה בצורה הבאה:</u>

```
[Route("search")]
public string Search(int id, int authorId, string name, int rating, int price)
{
    return "Hello from search";
}
```

<u>בצורה הזו אנו יכולים להעביר פרמטרים בצורה הבאה:</u>

https://localhost:44303/search/?id=23&name=Eliyahu&rating=42

וכך בעצם להזין נתונים לפונקציה.

נשים לב לפרט חשוב, אם ניתן לשתי מתודות שונות שמחזירות דברים שונים את אותו הנתיב ב-attribute זה ייצור לנו שגיאה בזמן ריצה (לא בזמן כתיבה), לכן חשוב לשים לב מראש.

שיטה פשוטה (שנקראת token replacement) כדי לשים לב לכך מראש היא לשים את ה-attribute הבא:

```
[Route("[controller]/[action]")]
```

באפשרות הזו אנו בעצם נותנים למערכת תבנית שבה ייכנסו החלקים הנכונים בהתאמה, נניח המילה controller תתחלף בשם הקונטרולר עצמו והמילה action תתחלף עם שם הפונקציה.

את האפשרות הזו אפשר גם להוסיף מעל שם המחלקה וזה יהיה תקף לכל המתודות.

אפשר כמובן גם לשים את האופציה הבאה מעל לפונקציה (לא מעל כל המחלקה):

```
[Route("[controller]/[action]/{id}/{name}")]
```

אם נכתוב את ה-Route מעל למחלקה כולה, נוכל להוסיף המשך ספציפי לפונקציה באמצעות Route נוסף (בתוכו נוסיף רק את התוספת):

```
[Route("{id}/{name}")]
```

אם נרצה לדרוס את ה-Route שהגדרנו בהתחלה, נוסף טילדה ב-Route הספציפי:

[Route("~/api/get-all-authors")] (אפשר גם בלי טילדה אבל לא קריטי).

כעת אנחנו רוצים ליצור מצב שבו נוכל להוסיף רק פרמטר אחד ב-path של האתר ולהגדיר לו סוג כך שיזהה לאיזו פונקציה לפנות.

```
[Route("api/[controller]")]
[ApiController]
public class CakesController : ControllerBase
{
    [Route("{id:int}")]
    public string GetByIdInt(int id)
    {
        return "Cake Id (INT)" + id;
    }

    [Route("{id}")]
    public string GetByIdString(string id)
    {
        return "Cake Id (string)" + id;
    }

}
```

כך באמצעות נקודותיים נוכל להגדיר type של המשתנה שיוזן.

עכשיו אם נזין מספר זה יוגדר כ-int אבל אם נזין טקסט זה יוזן כ-string ויגיע לפונקציה המתאימה.

אם היינו מגדירים את הפונקציה הראשונה בדוגמא בצורה הבאה:

```
[Route("{id:int:min(10)}")]
public string GetByIdInt(int id)
{
    return "Cake Id (INT - min 10)" + id;
}
```

כך שאם נזין מספר קטן מ-10 זה יוזן כ-string ומ-10 ומעלה זה יהיה int.

ניתן להוסיף גם מספר מקסימלי:

```
[Route("{id:int:min(10):max(20)}")]
public string GetByIdInt(int id)
{
    return "Cake Id (INT - min 10)" + id;
}
```

דבר נוסף שניתן להוסיף הוא אורך מינימלי ל-string:

```
[Route("{id:minlength(5)}")]
public string GetByIdString(string id)
{
    return "Cake Id (string)" + id;
}
```

במקרה זה אם נכתוב רצועה באורך פחות מחמש זה יחזיר לנו שהדף לא נמצא.

אפשר גם לדרוש גודל מדויק:

```
[Route("{id:length(5)}")]
```

או טווח של אורכים:

```
[Route("{id:range(10,15)}")]
```

או שיהיו רק אותיות בלי מספרים:

```
[Route("{id:alpha}")]
```

שתי מתודות נוספות ונחמדות הן:

- Required
- Regex

אז ב-Regex:

```
[Route("{id:regex(a(b|c))}")]
public string GetByIdRegex(string id)
{
    return "Cake Id (REGEX)" + id;
}

[Route("{id}")]
public string GetByIdString(string id)
{
    return "Cake Id (string)" + id;
}
```

הגדרנו שיהיה חייב להתחיל ב-a ואחר b או c, אבל אם לא יעמוד בתנאי אז המידע יישלח למתודה השנייה. (סדר הפונקציות לא משנה)

שיעור 3

ראינו שהכתובת מורכבת מהפעולה של ה-http (REST) ומה-URL, נשים לב לפירוט הבא:

כאשר אנו רוצים להציג מידע אנו שתמשים ב-GET וראינו בעצם אפשרויות ההצגה שיש לנו.

RESTful Get URLs

- **Get + domain.com/employees**

  Gets a list of all the employees
- **Get + domain.com/employees/{employeeId}**

  Gets the details of a single employee
- **Get + domain.com/employees/{employeeId}/accounts**

  Gets the account details of a single employee
- **Get + domain.com/employees/{employeeId}/accounts/{accountId}**

  Gets the single account details of a single employee

אם נרצה להוסיף רשומות נשתמש ב-POST:

- **Post + domain.com/employees**

  Body: Employee object

  Adds a new employee
- **Post + domain.com/employees/{employeeId}/accounts**

  Body: Account object

  Adds a new account for the employee

כאן אנו רואים בתמונה את הכתיבה של ההוספה.

לשינוי של פרטים של משתמש קיים (שינוי של כל הרשומה) נשתמש ב-PUT:

- **Put + domain.com/employees/{employeeId}**

  Body: Employee object

  Updated the employee
- **Put + domain.com/employees/{employeeId}/accounts/{accountId}**

  Body: Account object

  Updates the specific account for a specific employee

אם נרצה למחוק, נשתמש ב-DELETE:

- **Delete + domain.com/employees/{employeeId}**

  Deletes the employee

- **Delete + domain.com/employees/{employeeId}/accounts/{accountId}**

  Deletes the specific account for a specific employee

אז בשיעור הזה נדבר על החזרת מידע.

פתחנו קונטרולר שנקרא Employee ופתחנו לו גם Model.

כעת, את הפונקציות שיצרנו עד עכשיו שמחזירות string, הן יחזירו עובד ממש, כלומר, יחזירו מופע של המודל, או יותר נכון רשימה של עובדים.

```
public List<EmployeeModel> GetEmplpyees()
{
    return new List<EmployeeModel>()
    {
        new EmployeeModel(){Id = 1, Name = "Gil"},
        new EmployeeModel(){Id = 2, Name = "Avia"},
        new EmployeeModel(){Id = 3, Name = "Ziv"}
    };
}
```

כאשר נקרא לזה ב-URL, זה יחזיר לנו מערך ב-JSON.

יכולנו להחליט שהפונקציה מחזירה סוג של List<T>, IEnumerable<T>, IAsyncEnumerable<T>.

עד עכשיו גם לא היה לנו שליטה על סוג התשובה שתחזור.

הפתרון לכך הוא שהמתודה תחזיר IActionResult.

```
[Route("{id}")]
public IActionResult GetEmployeeById(int id)
{
    if (id == 0)
    {
        return NotFound();
    }
    return Ok(new List<EmployeeModel>()
    {
        new EmployeeModel(){Id = 1, Name = "Gil"},
        new EmployeeModel(){Id = 2, Name = "Avia"},
        new EmployeeModel(){Id = 3, Name = "Ziv"}
    });
}
```

בפונקציה הזו אנו רואים שאם היינו מזינים id שהוא 0, היינו מקבלים קוד 404, אחרת היינו מקבלים קוד 200.

ישנה אפשרות להשתמש גם ב-ActionResult (לא האינטרפייס) שם לא נחזיר Ok().

יש אפשרות להריץ את המערכת דרך ה-CMD בצורה של watch כמו שראינו במקרים קודמים (מונגו לדוגמא).

את הפקודה נריץ ב-CMD בתוך התיקייה של הפרויקט והיא `dotnet run --watch`.

אחד הדברים ששונים מהמצב הקודם הוא הפורטים.

יש קובץ שנקרא `launchSetting` והוא קובץ JSON שבתוכו יש אובייקט שנקרא `Profiles`

אנחנו בדרך כלל עובדים עם הפרופיל של `IISExpress` אבל המצב שעובדים מתוך ה-CMD הוא בפרופיל `Project`.

הפורטים שונים בין הפרופילים – אז חשוב לשים לב.

נמשיך...

לצורך ההמשך נפתח קונטרולר חדש ונעבוד איתו, בדוגמא שלנו יהיה קונטרולר בשם `Animal`.

ראינו אפשרות להשתמש ב-`Accepted()` במקום ב-`OK()` והיא מחזירה סטטוס 202, שאומר שה-`request` התחיל בהצלחה אבל טרם הושלם.

בתוך ה-`Accepted` נכניס כתובת ששם נמצא המידע והוא יכניס את כתובת ל-`header` שנקרא `location`.

במקום להזין את הכתובת נוכל להשתמש ב-`AcceptedAtAction(nameof(<function>))`

ואז הוא בעצם יכניס פשוט את המיקום של אותה הפונקציה במקום שנזין אותו בעצמנו.

בנוסף נוכל להגדיר שם לפעולה בתוך ה-`Route`:

```
[Route("", Name = "MyDefaultRoute")]
```

ואז להשתמש באפשרות :

```
public IActionResult Test()
{
    return AcceptedAtRoute("MyDefaultRoute");
}
```

עכשיו נתחיל לראות איך אנחנו בעצם משתמשים בכל פעולות ה-REST:

כדי ליצור חיה חדשה בדוגמא שלנו, נגדיר את הפונקציה באמצעות `attribute` להיות POST:

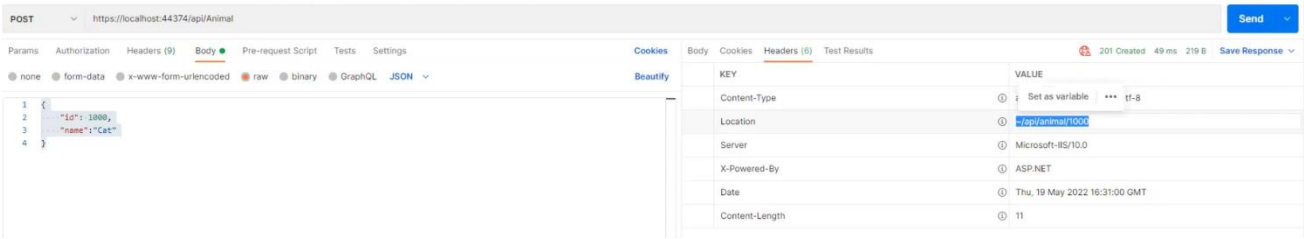
```
[Route("{id}")]
public IActionResult GetAnimal(int id)
{
    return Accepted(new AnimalModel { Id = id, Name = "Dog"});
}

[HttpPost("")]
public IActionResult CreateAnimal(AnimalModel animal)
{
    /*
     some code
     */
    return Created("~/api/animal", new {Id = animal.Id});
}
```

הגדרנו פונקציה שהכתובת המסתיימת ב-`id` תוביל אליה להצגת מידע על החיה.

החיה תגיע בעצם מהיצירה בפונקציה השנייה שם נקבל סטטוס 201 (`created`).

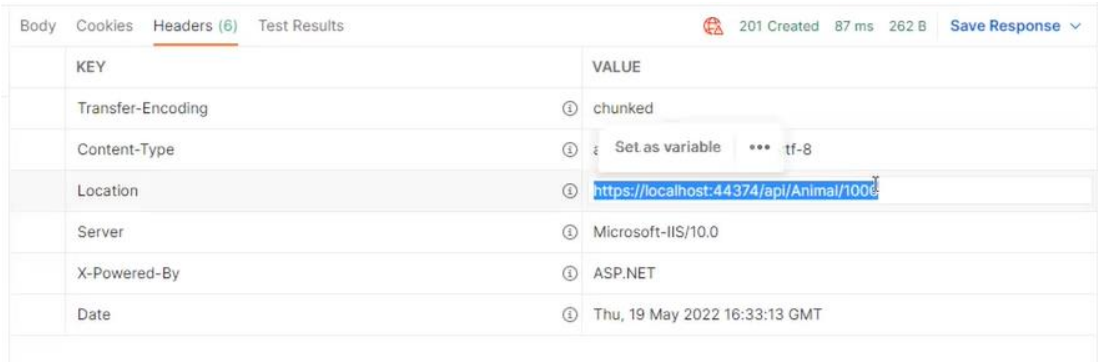
הכתובת שתוזן תוזן ב-`header` בשם `Location` כמו שראינו קודם.

בעמוד הבא נראה תמונה שמראה זאת מתוך ה-PostMan כאשר ה-`header` המדובר מסומן בכחול.



הקוד הבא יהיה אפילו יותר טוב:

```
[Route("{id}")]
public IActionResult GetAnimal(int id)
{
    return Accepted(new AnimalModel { Id = id, Name = "Dog"});
}

[HttpPost("")]
public IActionResult CreateAnimal(AnimalModel animal)
{
    /*
     some code
     */
    return CreatedAtAction(nameof(GetAnimal), new {Id = animal.Id}, animal);
}
```



אפשרויות נוספות:

`File()` – יוריד קובץ למשתמש.

`Forbid()` – אסור למשתמש לפנות.

`Redirect()` – על זה נרחיב עכשיו.

נבנה שני ראוטים שאחד הוא זה שכבר לא קיים וממנו מפנים לאחד אחר והשני הוא התקין.

מה שנראה זה שהדפדפן יודע במקרה ספציפי כשהסטטוס שחוזר מתחיל ב-3 (300 ו..) לעשות את ה-`redirect` בעצמו לפי הערך שנקרא `location`.

```
[Route("test")]
public IActionResult Test()
{
    return LocalRedirect("~/api/animal/NewAnimal");
}


[Route("NewAnimal")]
public IActionResult CreateAnimal()
{
    return Ok("New Animal");
}
```

הפונקציה Test לא תצליח לשלוח ותפנה לכתובת המוזנת ב-LocalDirect. ברגע שננסה להגיע ל-test זה יעבור לבד אוטומטית ל-NewAnimal.

- את ה-LocalRedirect ניתן להפנות רק בתוך האתר שלנו.

ה-LocalRedirect הוא הפנייה זמנית, זה אומר שהדפדפן לא שומר וזוכר אותה להבא, אבל אם נעשה LocalRedirectPermanent זה בעייתי וצריך להיזהר איתו, כי אז זה כן שומר ב-cash את הכתובת.

מתודות נוספות:

NoContent() – אין באמת מה להוסיף, שלחו בקשה ופשוט מחזירים שהכל בסדר אבל אין באמת מה להגיד (סטטוס 204)

NotFound() – לא נמצא (סטטוס 404)

Model Binder:

החלק הזה עוסק בגדול בכך שכאשר מגיעים לנו נתונים מבחוץ (מהקריאה) אל המתודה\קונטרולר שלנו, איך אנחנו דואגים לקשור אותם לפרמטרים נכונים במתודה?

כאשר מגיעה קריאה, ראינו שהיא מחולקת לכמה חלקים:

- URL – והיא יכולה להתחלק לכמה חלקים, ל-Routes (כל משמופרד על ידי '\') ולעוד פרמטרים שייכתבו בצורה של queryString.
- Headers
- Body
- Form-Data – אם שלחנו טופס ב-POST או ב-GET והוא לא הגיע כ-JSON אלא כטופס ממש.

ה-ModelBinder הוא בעצם משהו שקורה מאחורי הקלעים שנמצא בתהליך שבין ה-request לבין המתודה שלנו, הוא בעצם בודק את המשתנה שנכנס – נניח לקבל id שהוא int, להכניס למתודה לאחר שהוא וידא שהוא אכן מתאים למתודה.

ישנן מתודות מובנות שעוזרות לנו לנתב את ה-ModelBinding.

וכמובן נוכל ליצור מודל כזה בעצמנו.

BindProperty – שמים אותו כ-attribute מעל ערך מסוים ברמת המחלקה (בתוך המחלקה עצמה עוד לפני שימוש במתודות) והוא גורם לכך שהערך יתמלא אוטומטית (בד"כ) מתוך Form-Data.

יצרנו לצורך הדוגמא קונטרולר של ארצות:

```
[BindProperty]
public string Name { get; set; }

[HttpPost("")]
public IActionResult AddCountry()
{
    return Ok(this.Name);
}
```

בתוך הקונטרולר יצרנו מאפיין של שם, ומתודה של הוספת מדינה שמוגדרת כ-POST.

את המאפיין קשרנו עם BindProperty.

מה שקורה בעצם זה שהמאפיין מחפש בטופס שנשלח פרמטר בשם name וקושר אותו אוטומטית למאפיין במחלקה ומוכנס לפונקציה (זה רק לצורך הדוגמא – לא היה חייב להכניס לפונקציה).

נדגיש כאן את ההבדל בין GET ל-POST:

כאשר אנו שולחים את המידע ב-POST, הוא עובר לתוך ה-body של ה-request.

אבל כאשר אנחנו שולחים את המידע ב-GET, הוא יעבור בצורה של queryString.

את BindProperty ניתן לעשות גם על אובייקט, כלומר, אם יצרנו מודל country ובתוכו שני מאפיינים של שם ועיר בירה, ובקונטרולר ניצור רק מופע של המודל, הוא עדיין ידע להכניס את המידע לתוך המאפיינים של המופע.

בעיקרון BindProperty עובד רק עם POST, אבל אם נרצה שיעבוד ב-GET אז אפשר לעשות שינוי קטן:

```
[BindProperty(SupportsGet = true)]
```

דבר אחרון לשיעור:

מה הצורה הבסיסית (ברירת המחדל) שדברים מתקבלים?

אם נכתוב מתודה הכי פשוטה שתהיה GET, בברירת המחדל שלו הוא מחפש את המידע ב-URL בחלק של ה-queryString.

אם שמנו משתנה מורכב יותר ולא פרימיטיבי, אז בברירת המחדל הוא ייקח מידע מה-body והוא יצפה לקבל שם JSON.

שיעור 4

נתחיל עם ברירת המחדל לקשירת משתנים למתודות שלנו.

אם לא כתבנו מאיפה הוא אמור להגיע, הדפדפן יצפה לקבל משתנים פרימיטיביים מה-queryString שב-URL.

אבל משתנים מורכבים הוא יצפה לקבל מה-body.
לדוגמא, המתודה הבאה:

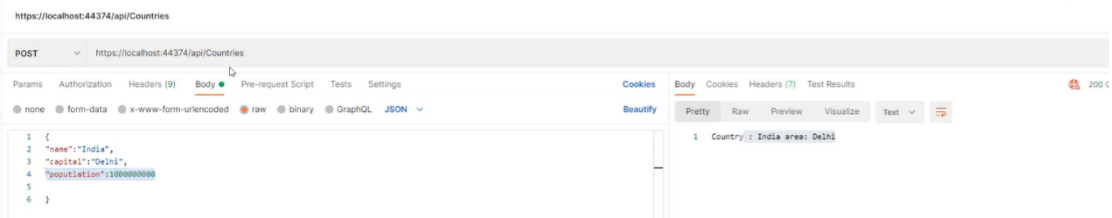
```
[HttpPost("{name}/{area}")]
public IActionResult AddCountry(string name, string area)
{
    return Ok($"Country: {name}, Area: {area}");
}
```

ברגע שנקרא ל-URL הזה אז המידע יילקח מה-Route ואם לא היה Route שונה ב-attribute אז המידע היה נלקח לפי ה-queryString שהיינו מזינים.

נזכור שבשימוש של Route יש משמעות לסדר הכתיבה שלנו משום שזה מותאם לתבנית שב-attribute.

ובמשתנה מורכב:

```
[HttpPost("")]
public IActionResult AddCountry(CountryModel country)
{
    return Ok($"Country: {country.Name}, Area: {country.Capital}");
}
```



וכאן בתמונה אנו רואים שבמשתנה מורכב זה חוזר ב-body ואנו מצפים לקבל אותו כ-JSON.

בנוסף ראינו שאנו יכולים לתת לו עוד משתנים שלא קיימים בסוג המשתנה שלנו (כאן הוספנו אוכלוסייה) וזה לא ישפיע כלל על התוכן המוחזר.

עכשיו נעבור על האפשרויות שיש לנו לקבוע את סוג המשתנה אותו נקבל ומאיפה נקבל אותו.

:[FromQuery]

```
[HttpPost("")]
public IActionResult AddCountry(CountryModel country,
                                [FromQuery] int population)
{
    return Ok($"Country: {country.Name}, Area: {country.Capital}, Population: {population}");
}
```

במקרה הזה, המשתנה המורכב של country הגיע מה-body והמשתנה של אוכלוסייה יגיע מה-queryString כמו שנדרש ממנו.

אם נשתמש גם ב-attribute הזה וגם ב-Route מעל לפונקציה, תהיה התעלמות מה-Route.

```
[HttpPost("GetName/{name}")]
public IActionResult GetName([FromQuery] string name)
{
    return Ok($"Name is: {name}");
}
```

אפשרי לקבוע תחת ה-attribute הזה גם משתנה מורכב אבל זה לא נהוג.

כלומר ככה:

```
[HttpPost("")]
public IActionResult AddCountry([FromQuery] CountryModel country)
{
    return Ok($"Country: {country.Name}, Area: {country.Capital}");
}
```

:[FromRoute]

כך אומרים לדפדפן לקחת את המידע מה-Route.

```
[HttpPost("GetName/{name}")]
public IActionResult GetName([FromRoute] string name)
{
    return Ok($"Name is: {name}");
}
```

כאשר מדובר במשתנה מורכב:

```
[HttpGet("GetName/{name}/{capital}")]
public IActionResult GetName([FromRoute] CountryModel country)
{
    return Ok($"Country: {country.Name}, Area: {country.Capital}");
}
```

אבל שוב נאמר שזה לא נהוג.

:[FromForm]

```
[HttpPost("{id}")]
public IActionResult AddCountry([FromQuery] int id,
                                [FromForm] CountryModel country)
{
    return Ok($"Id: {id} , Country: {country.Name}, Area: {country.Capital}");
}
```

כאן אנו מבקשים ממנו לקבל מידע של id מתוך ה-queryString ואת המידע של country מתוך ה-form.

נבחין בין המקרים, כאשר אני מעביר FromForm אני בעצם מעביר את המידע בצורה נקייה מתוך טופס ללא מעבר דרך ה-request. לעומת זאת, אם נרצה שהמידע מ-form יחזור דרך ה-request אז היינו צריכים גם קוד של JavaScript ברקע שיעשה עוד כמה דברים.

אם הייתי כותב בצורה הבאה:

```
[HttpGet("")]
public IActionResult AddCountry([FromForm] CountryModel country)
{
    return Ok($"Country: {country.Name}, Area: {country.Capital}");
}
```

וכמובן בהנחה שיש לי form בעמוד html שמחזיר לי מידע של שם מדינה ועיר בירה.

אז נשים לב שהגדרנו את המתודה ב-GET, אבל עדיין ביקשנו ממנה לקחת מידע מה-form.

הבעיה כאן היא שבגלל הגדרת ה-GET המידע לא נשלח ל-body אבל הפונקציה מחפשת שם את המידע ב-FromForm, המידע כן הגיע ונשלח ב-queryString.

ולכן כדי לתקן את המצב וכן להצליח גם לשלוף את המידע נשנה את ה-attribute של country ל-[FromQuery] וזה יעבוד.

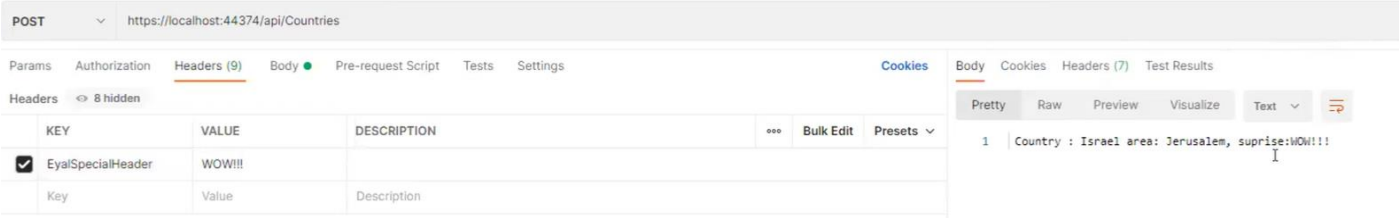
ניתן להוסיף גם אפשרות לקבל מידע מ-header.

:[FromHeader]

```
[HttpPost("")]
public IActionResult AddCountry([FromForm] CountryModel country,
                                [FromHeader(Name ="EyalSpecialHeader")] string surprise)
{
    return Ok($"Country: {country.Name}, Area: {country.Capital}, Surprise: {surprise}");
}
```

נשים לב שאנו מוסיפים לו את שם ה-header שממנו נקבל מידע.

את הניסיון עשינו מהפוסטמן כי במערכת עצמה אין לנו דרך לתאם headers.



הוספנו header משלנו והוא שלף מתוכו את המידע.

דוגמא מעשית:

נניח שצריך לממש API של PayPal לדוגמא, ושם בדוקומנטציה כתוב שצריך לשלוח headers עם פרטים מסוימים, אז אנחנו נצטרך להוסיף אותם לשליפת המידע.

עכשיו נראה איך ליצור אחד משלנו, כלומר attribute שיוכל לקבל מידע מאיפה שנגדיר לו.

יצרנו פונקציה של חיפוש מדינה, שמקבל מערך של string עם מדינות, אבל המידע יגיע אליה כ-string שלם מהצורה "Canada|Israel|China".

ניצור בנוסף מחלקה מיוחדת שבה נירש מאינטרפייס של מייקרוסופט שנקרא IModelBinder.

האינטרפייס הזה יחייב אותי להשתמש במתודה BindModelAsync() ששם נוכל לגשת למה שמגיע רגע לפני שהוא מגיע לפונקציה כארגומנט.

נתאים את הפונקציה הזו שתקבל את ה-string ויהפוך אותו למערך.

נקדים קודם לומר שהקריאה תיראה בצורה הבאה:

www.mySite.com/api/search?countries=Canada|India|Israel

כמובן שזו רק דוגמא.

הקריאה תתפוס את המשתנה מתוך ה-queryString, תעבור איתו דרך המחלקה שיצרנו.

בגלל שאנחנו יודעים איך המשתנה ייקרא, נשלח את השם שלו במתודה TryGetValue() שבתוכה גם נכניס ארגמנט out.

את המתודה שבמחלקה אנו חותמים עם Task משום שזה דבר שיכול לקחת זמן, מה שבדוגמא שלנו לא קרה בעקרון, ואז נצטרך שזה לא יתקע את התוכנה.

כמובן שסיבה נוספת היא שאנו מחויבים לחתימה שמגיעה מהאינטרפייס.

תזכורת

מה זה out?

זה משתנה שאני יוצר אותו כארגומנט והוא יודע גם לחזור החוצה כמשתנה שאפשר להשתמש בו.

```
public void OtherFunction()
{
    // equal to do: someName = ...
    var myResult = GetSomething(out var someName);
}
public bool GetSomething(out string name)
{
    name = "GilGil";
    return true;
}
```

בעצם בפונקציה OtherFunction() שני המשתנים, גם myResult וגם someName זמינים לי לשימוש, כאשר myResult קיבל את המידע מה-return ו-someName קיבל מה-out.

נשים לב שאם הגדרנו שיהיה ארגומנט out אז אהיה חייב למלא אותו בתוכן במהלך הקוד של הפונקציה.

נמשיך..

הפונקציה TryGetValue() תחזיר לנו משתנה בוליאני אם היא מצאה את המשתנה שהכנסנו ותחזיר גם את המשתנה out שנתנו לה.

על המשתנה שיוחזר נבצע split ונחזיר אותו כמערך של מדינות.

את המערך נחזיר לפונקציה בצורה שנראה תכף בקוד, ונסיים את המתודה באמצעות החזרת הודעה של Task.CompletedTask.

```
public class CustomBinder : IModelBinder
{
    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        var data = bindingContext.HttpContext.Request.Query;

        var result = data.TryGetValue("countries", out var country);

        if (result)
        {
            var array = country.ToString().Split('|');
            bindingContext.Result = ModelBindingResult.Success(array);
        }
        return Task.CompletedTask;
    }
}
```

זהו הקוד של המחלקה שיצרנו.

כעת נראה איך אנחנו קושרים את ה-binder הזה לפונקציה שלנו.

אנו בעצם משתמשים כמו שראינו ב-attributes המובנים רק שנשתמש בצורה הבאה:

[ModelBinder(typeof(<Model Binder Name>))]

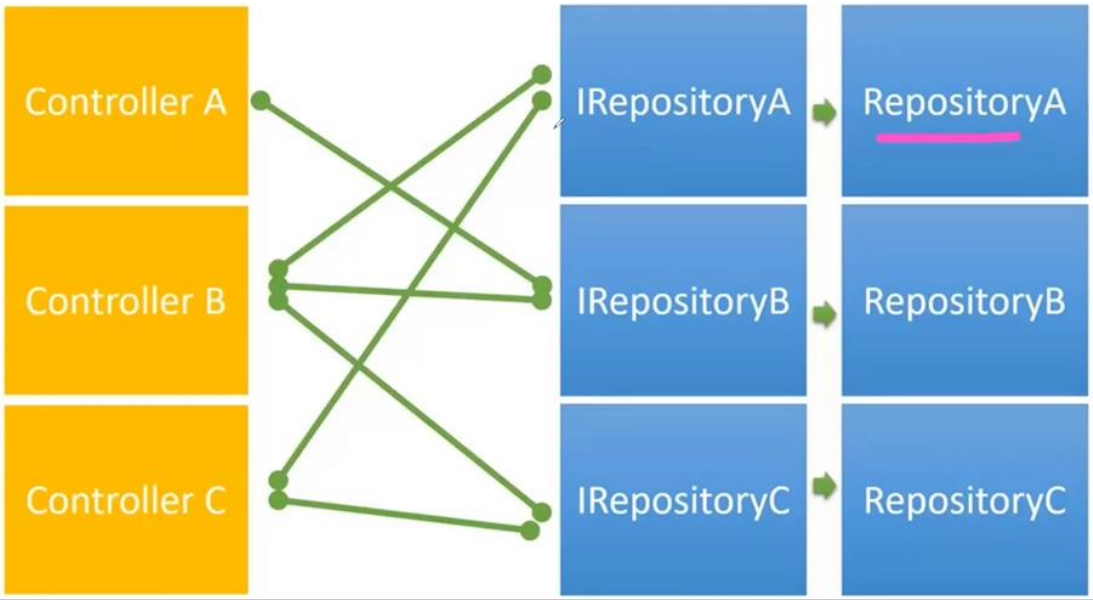
```
[HttpGet("search")]
public IActionResult SearchCountries(
                    [ModelBinder(typeof(CustomBinder))] string[] countries)
{
    return Ok();
}
```

כמובן שניתן גם להחזיר את המידע במקום סתם להחזיר OK() ריק.

Dependency Injection:

למדנו את הנושא כבר באנגולר אבל נלמד עליו גם כאן.

נניח ויש לנו מסד נתונים ויש לנו קונטרולר, נוכל לפנות ישירות מהקונטרולר ונוכל גם לעבור דרך מחלקה אחרת (repository pattern).



כך בעצם עובדת התבנית של repository, אני עובד מול Interfaces ולא אכפת לי מי המחלקה שמממשת אותה (כביכול לא אכפת).

הרעיון הכללי של Dependency Injection הוא שאנחנו עושים מעין רשימה של כל הממשקים שאנו משתמשים בהם ואז כשנרצה להשתמש בהם נקבל אותם בלי קריאה מיוחדת.

את "הרשימה", את כל הממשקים לשימוש, נזריק בפונקציה שנקראת ConfigureServices ואז הם יהיו זמינים לנו בכל קונטרולר שנרצה.

נניח שניצור בנאי בקונטרולר Countries ונקרא בו לאחד מהרשומים ברשימה – כלומר, בלי שיהיה רשום לא נוכל לקרוא לו. זה כמובן חוסך לנו ליצור מופע חדש בכל שימוש (במקום להשתמש ב-new)

דוגמא לאחד שקיים שם בברירת המחדל, היא Logger שזה בעצם ממשק שמאפשר לנו לעשות logging במחלקה.

יש לנו שלושה סוגים של טווחי זמן חיים למשתנה בהזרקה:

- Singleton – חי למשך כל האפליקציה במופע אחד.
- Scoped – חי לאורך קריאה אחת, כל קריאה מופע חדש.
- Transient – יכול להיות בכמה מופעים חדשים במהלך קריאה אחת.

Singleton:

ניצור מחלקה שמדמה תקשורת עם מסד נתונים (יצירה\מחיקה\עריכה של מוצרים).

לצורך הדוגמא יצרנו מודל של מוצר ומחלקה של repository למוצר שם יש פעולה של הוספת מוצר חדש ופעולה שמחזירה את רשימת המוצרים.

בנוסף יצרנו קונטרולר חדש למוצרים.

קוד לריפוזיטורי:

```
public class ProductRepository
{
    private List<ProductModel> products = new List<ProductModel>();

    public int AddProduct(ProductModel product)
    {
        product.Id = products.Count + 1;
        products.Add(product);
        return product.Id;
    }

    public List<ProductModel> GetAllProducts()
    {
        return products;
    }
}
```

קוד לקונטרולר:

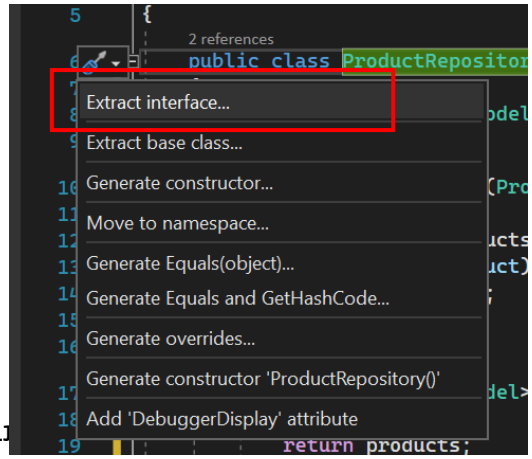
```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    private ProductRepository _productsRepo;
    public ProductsController()
    {
        _productsRepo = new ProductRepository();
    }

    [HttpPost]
    public IActionResult AddProduct([FromBody] ProductModel product)
    {
        _productsRepo.AddProduct(product);
        var products = _productsRepo.GetAllProducts();

        return Ok(products);
    }
}
```

וכמובן שהמודל מחזיק פשוט מאפיינים כידוע.

עכשיו תהיה לנו בעיה בגלל בכל פעם שנקרא לקונטרולר הוא יצור מופע חדש כך שהרשימה של המוצרים כל הזמן תתאפס.

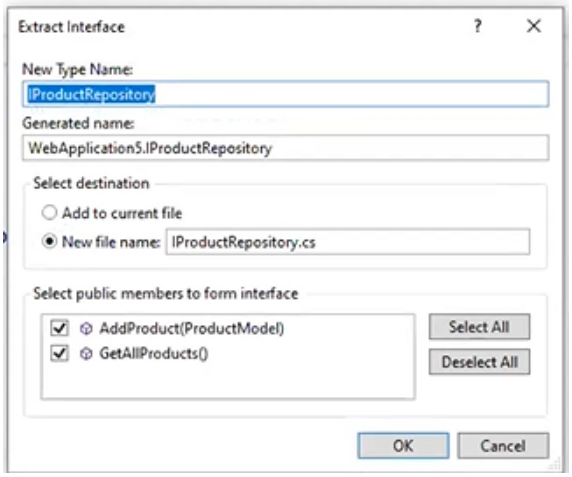
כדי להתמודד עם זה, קודם כל רק בשביל הסדר, ניצור גם אינטרפייס מסודר.

דרך מהירה ליצירה של אינטרפייס היא מקש ימני על שם המחלקה ולחיצה על:



מתוך הרצאות של אייל ברדוגו

לאשר את השם שהוא ייתן או להחליף וזה יצור אותו.



אז קודם שינינו קצת את הקונטרולר:

```
public class ProductsController : ControllerBase
{
    private IProductRepository _productsRepo;
    public ProductsController(IProductRepository productRepository)
    {
        _productsRepo = productRepository;
    }
```

(בעצם קראנו בבנאי לאינטרפייס והשתמשנו בו במקום במחלקה ישירות)

וזה בעצם השלב שבו אנחנו מזריקים את האינטרפייס לפונקציה ב-Startup:

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "WebApplication1", Version = "v1" });
    });
    services.AddTransient<CustomMiddleware>();

    services.AddSingleton<IProductRepository, ProductRepository>();
}
```

זה יפתור את הבעיה משום שזה יהפוך את המופע לסינגלטון ללא צורך במימוש של התבנית כמו שלמדנו ב-Design Patterns.

נשים לב לפרט חשוב, אם יש עוד רשומים ברשימה שב-ConfigureServices ונניח שיש שלושה כאלו שאחד תלוי בשני, כלומר, A צריך את B ו-B צריך את C, אז כל עוד שלושתם רשומים אפשר להשתמש באחד והוא לבד ידע להשתמש באחרים גם ללא יצירת מופע חדש.

Scoped:

במקום להגדיר בקוד AddSinglton נכתוב AddScoped.

```
services.AddScoped<IProductRepository, ProductRepository>();
```

כרגע זה יחזור ל"בעיה" שהייתה לפני כן שבכל הוספה המידע הקודם נעלם, רק שהפעם זה בכוונה תחילה.

זה משמש אותנו נניח במצב שאני שולח מידע לריפוזיטורי והוא מתקשר מול מסד הנתונים כך שאני לא צריך שישמור אצלו את המידע שהוספתי.

Transient:

סרוויסים שרשומים לאותו הרגע, אם הזרקתי אותם כמה פעמים באותה הקריאה, אז כל הזרקה היא מופע חדש.

בעצם ברגע שנגדיר:

```
services.AddTransient<IProductRepository, ProductRepository>();
```

גם אם נזריק שניים או שלושה מופעים שונים של IProductRepository בקונטרולר, אז כל אחד יהיה מופע בפני עצמו (מה שלא יקרה בשניים הקודמים שהכל יהיה תחת אותו המופע).