

בתרשים שלעיל מוצגת חלוקת הקוד לשכבות.

- API - השכבה החיצונית, אחראית על התממשקות מול הקליינט. (נקראת Presentation למרות שאין לה ממשק ויזואלי, אך היא חשופה לקריאות מבחוץ ולכן היא 'תצוגתית')
  - Service - אחראית על הלוגיקה העסקית.
  - Data - אחראית על חיבור למקור הנתונים.
  - Core - אחראית על המודלים והממשקים של כלל המערכת.
- אספקט חשוב מאוד בארכיטקטורה הוא ניהול התלויות בין השכבות. תלות נוצרת כאשר שכבה אחת משתמשת ומכירה שכבה אחרת. תלות היא מגבילה מטבעה. היא מקשה על תהליך הפיתוח וגורמת לסיבוכים כשרוצים לשנות את הקוד או להרחיב אותו. לכן, השאיפה היא תמיד לנתק תלויות ולהצליח לנהל שכבות ללא תלות או עם תלות מינימלית ככל האפשר. בתרשים זה התלות היא מחוץ לפנים, כלומר: שכבת ה-API תלויה ב-Service וב-Data, שכבות Service ו-Data מקבילות זו לזו ואינן תלויות אחת בשניה. בנוסף, כל השכבות תלויות ב-Core.
- שכבת ה-Core היא ליבת המערכת. בה נמצאות הגדרות מבני הנתונים והגדרות הפונקציונליות. שכבת ה-Core לא מכילה מימוש כלל. כל השכבות מכירות את שכבת ה-Core והן אלו שמממשות בפועל את הממשקים וההתנהגות שהיא מגדירה.

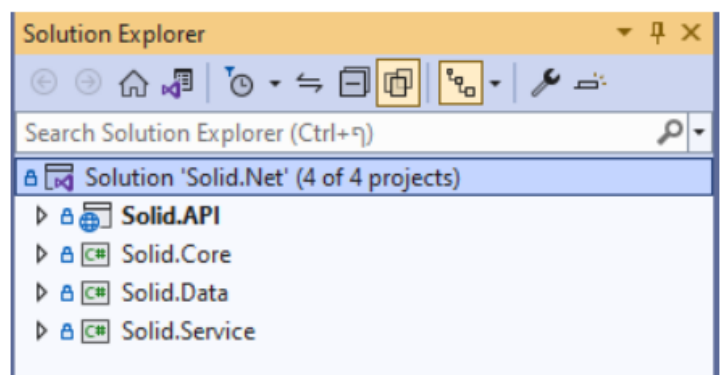
## מבנה הפרויקט ב-.NET

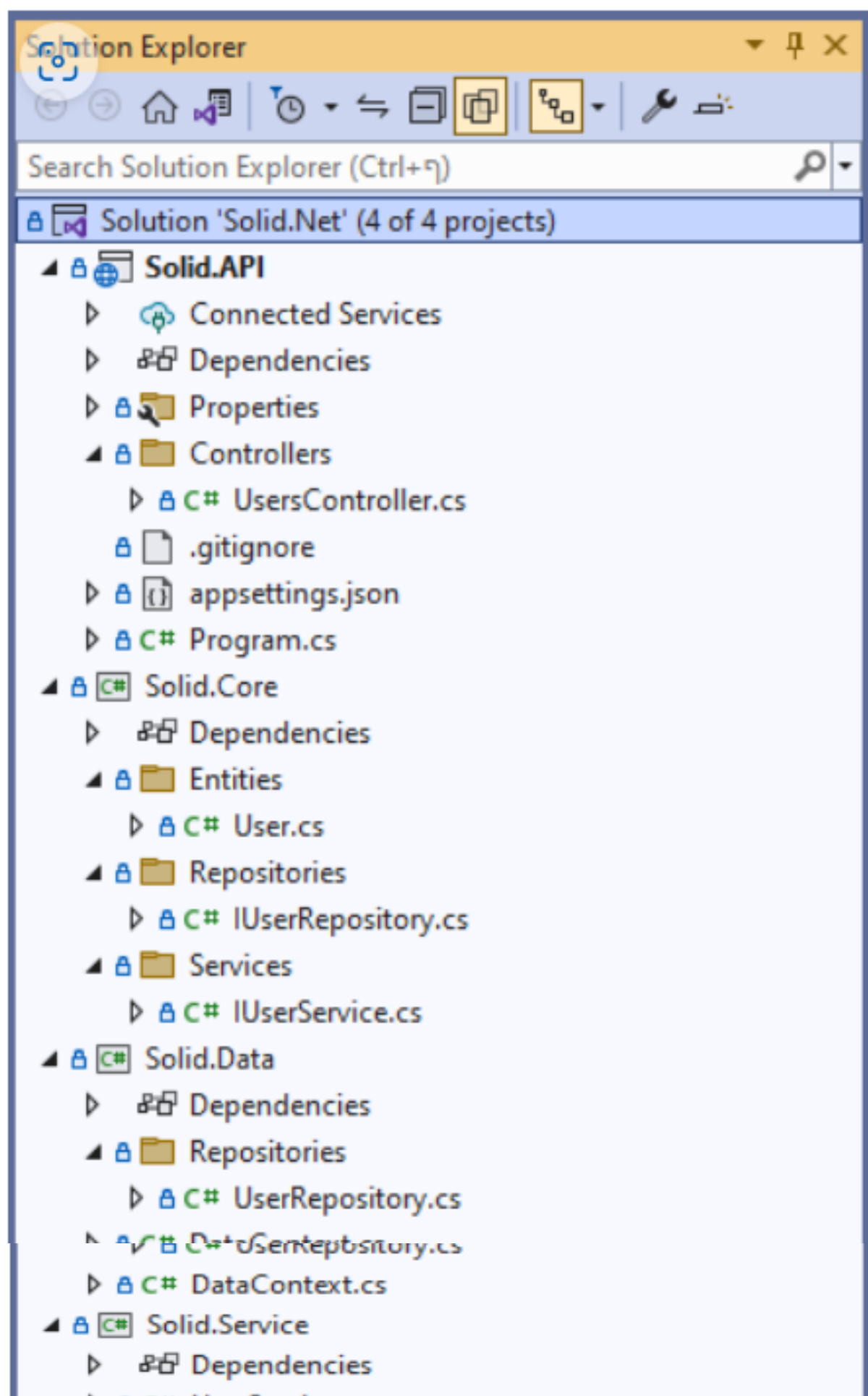


לחצי לצפייה בסרטון << Clean Architecture in .NET



חלוקה לשכבות ב-.NET







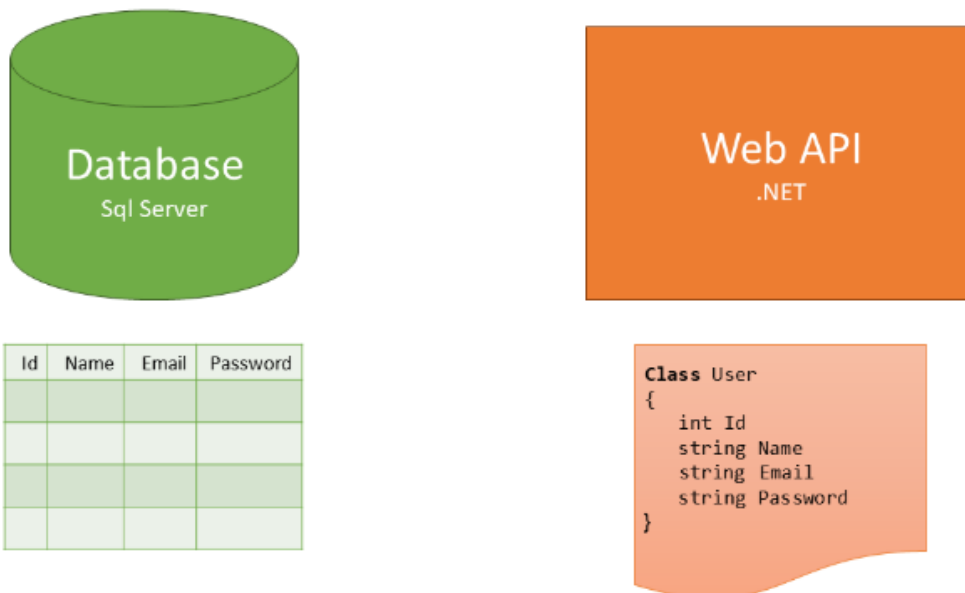
# EF - בסיס

ניתן לטפל בהתממשקות של הסרבר ל-DB במספר דרכים. אנחנו נשתמש ב-Entity Framework שהיא טכנולוגיה פופולרית מסוג ORM.

Entity Framework (או בקיצור EF) מטפלת בהתממשקות ל-DB בשני היבטים:

- מבנה הטבלאות
- נתונים

## התממשקות בין API ל-DB באמצעות EF



בתרשים הזה מוצגים ה-API וה-DB. כל אחד מהם הוא אפליקציה בפני עצמה. ה-API הוא אפליקציה שמספקת פונקציות לקריאה מקליינט. ה-DB הוא אפליקציה שיודעת לנהל נתונים, לשמור ולאחזר מידע.

בדוגמא הזו יש לנו נתוני משתמשים. הנתונים נשמרים ומנוהלים בטבלת Users שב-DB.

נניח שהקליינט שולח קריאה לקבל את רשימת המשתמשים. לעולם הקליינט לא פונה למסד הנתונים בעצמו. תמיד הוא פונה לאפליקציית סרבר כלשהיא שהיא זו שתפנה ל-DB, תקבל את הנתונים (או תעדכן, בהתאם לצורך) ותחזיר לקליינט את התוצאה.

כתוצאה מכך, ה-API צריך להתעסק גם הוא עם נתוני המשתמשים, שהרי הנתונים עוברים דרכו.

בעולם של DB (מסוג sql) נתונים מיוצגים במבנה טבלאי. לעומת זאת, מה קורה באפליקציית הסרבר, האם יש לנו אפשרות להחזיק נתונים כטבלה?

ודאי שלא.

באפליקציית הסרבר (שכתובה ב-C# במקרה שלנו) נתונים מיוצגים במבנה של אובייקטים. הווה אומר, שכדי שהאפליקציה תדע לעבוד עם הנתונים מה-DB היא צריכה קודם כל להגדיר מחלקות מתאימות.

עבור כל טבלה ב-DB תהיה מחלקה עם מאפיינים שתואמים לשדות הטבלה. כיון שלמעשה, הן ב-DB והן באפליקציית הסרבר, המבנה זהה רק בפורמט שונה, אנחנו יכולים לפתח צד אחד בלבד ולהניח לכלי אוטומטי לחולל את הצד השני בהתאם.

בשביל זה יש לנו את Entity Framework שיודע לעשות את העבודה הזו היטב משני הכיוונים:

- DB First – אנחנו יוצרים את ה-DB עם הטבלאות ו-EF מג'נרט מחלקות מתאימות בקוד.
- Code First – אנחנו יוצרים מחלקות בקוד ו-EF מייצר DB עם טבלאות מתאימות.

אחרי שסידרנו את הענינים עם מבנה הנתונים, אנחנו צריכים מישהו שיידע להמיר נתונים של טבלה לתוך אובייקטים וההיפך.

כאמור, ב-DB הנתונים נשמרים בפורמט טבלאי, באפליקציית הסרבר הם מוחזקים בפורמט של אובייקט. מישהו צריך לבצע את ההעברה מפורמט לפורמט – מה שנקרא סריאליזציה (Serialization)

המישהו הזה הוא שוב Entity Framework שסוגר לנו גם את הפינה הזו ויודע להעביר נתונים מאפליקציית הסרבר ל-DB ומה-DB לסרבר.

אז מה נשאר לנו לעשות?



(לא לדאוג, יש מספיק עבודה לכולם...)

בשלב זה נלמד לעבוד בשיטת `Code First`.

## התקנת Nuget Packages

יש להתקין את הפקג'ים הבאים:

### Nuget Packages

```
/* Install in Data Project */
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Tools

/* Install in API Project */
Microsoft.EntityFrameworkCore.Design
```

## DbContext

מחלקת `DbContext` מבית EF מספקת לנו את יכולת ההתממשקות ל-DB.  
מחלקת `DataContext` שלנו צריכה לרשת מהמחלקה `DbContext`.  
במקום List נשתמש ב- `DbSet` - מחלקה שמייצגת את הטבלה מה-DB.

### DataContext.cs

```
1 public class DataContext : DbContext
2 {
3     public DbSet<User> Users { get; set; }
4 }
```

## הזרקת ה-DbContext

עלינו לשנות את הזרקת ה-`DataContext` כדי לקבל את הפונקציונליות של EF.  
נשתמש בפונקציית הרחבה ש-EF מספק, כך:

### Program.cs

```
1 builder.Services.AddDbContext<DataContext>();
```

כדי לגשת ל-DB צריך Connection String - מחרוזת שמגדירה את פרטי ההתחברות למסד הנתונים.

ה- Connection String מורכב מכמה נתונים:

- Host / Server - כתובת המחשב שמארח את מסד הנתונים
- Port - הפורט במחשב המארח שדרכו יש גישה למסד הנתונים
- Database Name - שם מסד הנתונים
- User Name - שם המשתמש במסד הנתונים
- Password - סיסמה במסד הנתונים

בשלב זה, נעבוד מול מסד נתונים שמותקן לוקלית על המחשב שלנו. לכן נוכל לוותר על חלק מהנתונים.  
Connection String של מסד נתונים לוקלי מסוג Sql Server נראה כך:

### Connection String

```
"Server=(localdb)\MSSQLLocalDB;Database=sample_db"
```



אין צורך לשים את ה-Port וגם לא שם משתמש וסיסמה.

### הגדרת Connection String בקוד

ניתן להגדיר את ה- Connection String בכמה דרכים.

#### אפשרות אחת:

DataContext (כלומר, המחלקה שיורשת את DbContext) דורסת את הפונקציה OnConfiguring ומגדירה את החיבור כך:

```
protected override void OnConfiguring(DbContextOptionsBuilder  
optionsBuilder)  
{
```

```
optionsBuilder.UseSqlServer(@"Server=(localdb)\MSSQLLocalDB;Database=  
e=sample_db");  
{
```

ההזרקה ב-Program מגדירה גם את החיבור, כך:


```
builder.Services.AddDbContext<DataContext>(<
    options =>
options.UseSqlServer(@"Server=(localdb)\MSSQLLocalDB;Database=sam
mple_db");
```

במקרה כזה חובה להוסיף Constructor ריק ב-DataContext, כך:

```
public DataContext(DbContextOptions<DataContext>
options)
:base(options){}
```

לשים לב 

יש לשלוח את options ל-base, שהוא ה-Constructor של DbContext, מחלקת האב.

חשוב מאוד! 

למעשה, אין לשים את ה-Connection String בתוך הקוד עצמו מסיבות של אבטחה וצורך בטרנספורמציה, כפי שעוד יוסבר. נכון לעכשיו, נעבוד כך כי זה פשוט יותר ובהמשך נבין למה ואיך נכון לעשות זאת, בנושא ה-Configuration.



בשיטת ה-Code First אנחנו יוצרים את ה-Context והמודלים ב-C# ומחוללים את ה-DB והטבלאות בהתאם להם באופן אוטומטי. תהליך זה מורכב משני שלבים:

1. יצירת מיגרציה

2. עדכון ה-DB

### Add Migration

EF יודע לבחון את הקוד ולנתח לפיו איך לבנות את הטבלאות. פעולת הניתוח והסקת המבנה הדרוש מתבצעת באמצעות הפקודה `Add-Migration`. נריץ את הפקודה ב- `Package Manager Console` על פרויקט ה- `Data`

#### Package Manager Console

Add-Migration MigrationName



בפעם הראשונה, הפקודה תוסיף תיקיה בשם Migrations שלתוכה יתווספו המיגרציות בכל פעם.

### Update Database

לאחר שיצרנו מיגרציה (או מיגרציות), נוכל לעדכן את ה-DB בהתאם. נריץ את הפקודה הבאה.

#### Package Manager Console

Update-Database



הפקודה תריץ את כל המיגרציות שנוצרו לאחר העדכון האחרון של ה-DB. אם ה-DB עדין לא קיים, הפקודה תיצור אותו.

#### נקודות לציון

- לא ניתן ליצור מיגרציות באותו שם.
- ניתן להקליד את הפקודות עם אותיות קטנות.



ביחידת החומר הקודמת ראינו איך לעבוד עם EF כדי ליצור את מבנה הטבלאות. כשהרצנו את ה-API והפעלנו פונקציה של GET קיבלנו את הנתונים מהטבלה (שהוספנו ידנית...).

לעומת השליפה, שמירת הנתונים לא מתבצעת אוטומטית. אומנם EF עושה לנו את כל העבודה, אבל משאיר לנו את הפיקוד. אנחנו צריכים לתת לו את ההוראה מתי לשמור.

## Change Tracking

EF עוקב כל הזמן אחרי השינויים בנתוני האובייקטים הקשורים ל- `DbContext`. עבור כל אובייקט הוא מתעד סטטוס שיכול להיות אחד מהבאים:

- `Unchanged` - לא בוצע שום שינוי
- `Added` - חדש, עדין לא קיים ב-DB
- `Modified` - עודכן
- `Deleted` - נמחק
- `Detached` - מנותק, לא במעקב

הסטטוס נשמר על מאפיין שנקרא `EntityState`.

הסטטוס מנוהל ברמת ה- `Property`, מה שאומר שאם `Property` מסוים באובייקט השתנה, רק הוא ישלח לעדכון בטבלה.

`EntityState` מתייחס למצב האובייקט מאז שהוא נשלף מה-DB. מחזור החיים של המופע מסוג `DbContext` הוא `Scoped`, כלומר: בכל `Request` נוצר מופע חדש של `DbContext`. מובן שאין טעם בכל `Request` להביא את כל הנתונים שב-DB ל-`Context`, זה לא יעיל ולא נצרך. EF מנהל את העניינים בחוכמה ושולף רק את הנתונים שבאים לידי שימוש בפועל. מאז שהנתונים נשלפים הם נמצאים במעקב וכל שינוי שמתרחש בהם נרשם ומתועד. עם סיום ה-`Request` המופע של `DbContext` משתחרר מהזכרון ואיתו גם תיעוד סטטוס הנתונים.

## SaveChanges

כדי לשמור את השינויים שבוצעו על הנתונים נפעיל את הפונקציה `SaveChanges`.

דוגמת קוד לעדכונים שונים

### UserRepository.cs

```
1 public User Add(User user)
2 {
3     _context.Users.Add(user);
4     _context.SaveChanges();
5     return user;
6 }
7
8 public User Update(int id, User user)
9 {
10     var existUser = GetById(id);
11
12     existUser.FirstName = user.FirstName;
13     existUser.LastName = user.LastName;
14
15     _context.SaveChanges();
16     return existUser;
17 }
18
19 public void Delete(int id)
20 {
21     var user = GetById(id);
22     _context.Users.Remove(user);
23     _context.SaveChanges();
24 }
```

בהוספת אובייקט חדש, אם ה- `Id` הוא מספור אוטומטי, הוא נוצר בטבלה עם שמירת הנתונים. לאחר השמירה, EF מכניס לאובייקט את ה- `Id` שנוצר, וכך ניתן להחזיר את אותו אובייקט כשהפעם הוא יחזיק את ה- `Id` החדש.

## Transaction

ברוב מסדי הנתונים, `SaveChanges` היא טרנזקציונית, כלומר, לעולם לא יהיה מצב שרק חלק מהפעולות בוצעו בהצלחה. אם אחת מהפעולות נכשלה, כל הטרנזקציה מבוטלת ולא מתבצעת שום פעולה.

## Relationships

קשרי גומלין מתארים את היחסים בין שתי ישויות במסד הנתונים.

כיון שאנחנו משתמשים ב-EF בשיטת Code First, עלינו להגדיר גם את קשרי הגומלין בקוד.

חשוב להבין שאומנם הנתונים הם אותם נתונים, אך הם מיוצגים במבנה שונה ב-DB ובאפליקציה הסרבר. ב-DB הנתונים מוחזקים במבנה טבלאי ובקוד הם מוחזקים במבנה של אובייקטים. המבנה השונה משפיע על אופן הקישור בין הישויות וממילא על צורת הגישה לנתונים.

במסד נתונים רילציוני (SQL) קשרי הגומלין בין טבלאות מוגדרים באמצעות מפתח זר. טבלת האב מכילה את מפתח טבלת הבן וכך מוגדר הקשר שמאפשר גישה לשאר נתוני רשומת הבן.

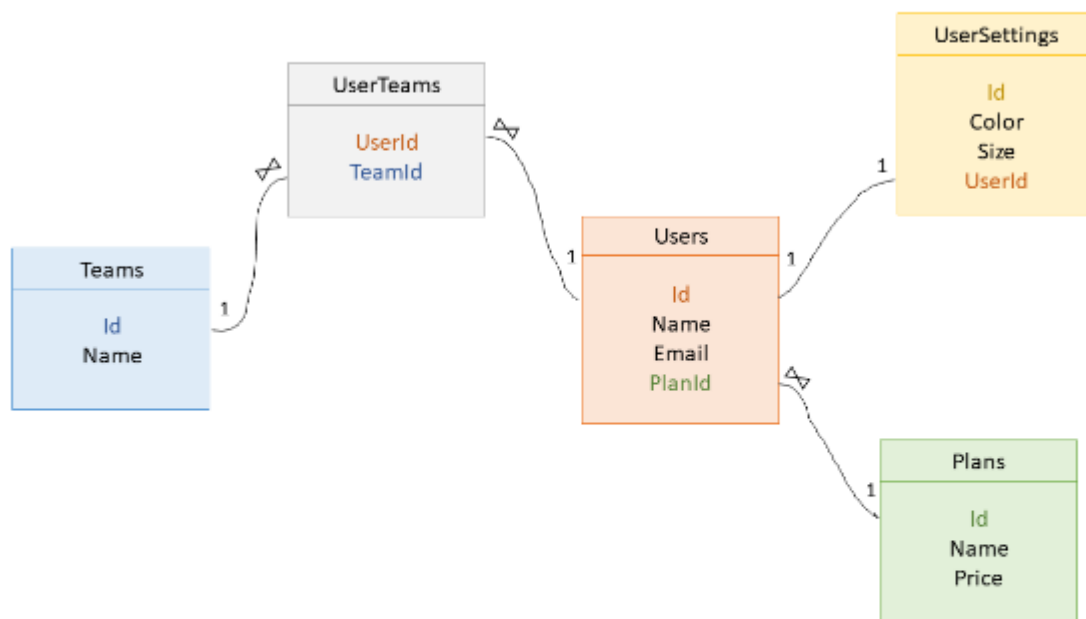
באובייקטים, קשרי הגומלין מוגדרים באמצעות הכלה של אובייקט. אובייקט האב מכיל את אובייקט הבן וכך מוגדר הקשר.

נדגים את סוגי קשרי הגומלין ואיך מטפלים בהם עם EF.



## תרשים קשרי גומלין

בס"ד



כאמור, קשר בין אובייקטים מוגדר באמצעות הכלה של אובייקט.

כאשר נגדיר שהמודל (אובייקט שמכיל מאפיינים בלבד ומגדיר מבנה נתונים) מכיל אובייקט אחר, EF יסיק את הקשר שביניהם ויצור אותו ב-DB בהתאם.

המאפיין שמגדיר את הקשר מכונה **Navigation Property**. נגדיר מאפיינים אלו בהתאם לסוג הקשר.

קשר של יחיד לרבים הוא הסטנדרטי והשימושי ביותר.

כדי לייצג קשר זה בין שני אובייקטים מספיק שאחד מהם יכיל את השני כדי ש-EF יבין את הקשר הנכון.

בדוגמא שלנו יש קשר של יחיד לרבים בין האובייקטים האלו:

- `User` - משתמש
- `Plan` - תוכנית תשלום

בטבלאות, קשר זה מיוצג באמצעות שדה `PlanId` בטבלת `Users` שהוא מפתח זר מטבלת `Plans`.

כיון שטבלאות הן שטוחות לא ניתן להכיל רשומה בתוך עמודה בטבלה.

הדרך לקשר בין רשומה אחת לאחרת היא באמצעות שדה מפתח של הרשומה המקושרת.

כיון שהמפתח הוא יחודי ולא חוזר על עצמו, ברגע שיש לנו אותו ביד יש לנו גישה לכל הרשומה.

אם נרצה לגשת לרשומה המקושרת נריץ שאילתה עם `JOIN` ונצטרף את הנתונים המקושרים.

לעומת טבלאות, אובייקטים הם עמוקים ויכולים להכיל בתוכם אובייקטים אחרים.

מבנה של אובייקטים הוא למעשה דומה הרבה יותר לאופן שבו אנו תופסים ומתיחסים לנתונים בחשיבה אנושית.

כיון שכך, אנחנו יכולים להגדיר ש'משתמש מכיל תוכנית' או ש'תוכנית מכילה רשימת משתמשים'.

שתי ההגדרות נכונות ומציגות שני צדדים של אותו קשר בדיוק.

נגדיר את הקשר ב- `User` וב- `Plan`, כך:

[User.cs](#)[Plan.cs](#)**User.cs**

```
1 public class User
2 {
3     public int Id { get; set; }
4
5     public string FirstName { get; set; }
6
7     public string LastName { get; set; }
8
9     public Plan Plan { get; set; }
10 }
```

הגדרנו שלושה מאפיינים שכולם מתיחסים לשדה אחד בלבד בטבלה.

שדה הקשר הוא `PlanId` שבטבלת `Users`.

כדי ש-EF יצור את השדה מספיק שנגדיר רק את המאפיין `User.Plan` **א** את המאפיין `Plan.Users`.

למה אם כן הגדרנו את שניהם?

אם נרצה לשלוף תוכנית עם המשתמשים השייכים לה, נוכל לעשות זאת בקלות ובצורה ישירה כאשר מוכן לנו מאפיין `Users` באובייקט `Plan`.

כלומר, זה עוזר בקטע של טיפול בנתונים אבל לא נדרש לבניית המבנה.

כעת, אם נריץ את ה-API ונפעיל את שליפת רשימת המשתמשים נקבל מבנה JSON שמכיל גם מאפיין בשם `Plan`. אבל רגע, למה הוא `null`? 😞

## Include

גם כאן באה לידי ביטוי היעילות של EF.

כיון ששליפת התוכנית דורשת פעולה של `JOIN`, היא לא תתבצע באופן דיפולטיבי, אלא רק אם נדרוש זאת.

לא תמיד יש לנו עניין לקבל את האובייקט המקושר, אבל אם כן, פשוט נגיד ל-EF לצרף אותו.

נעשה זאת באמצעות הפונקציה `Include`, כך:

### UserRepository.cs

```
1 public IEnumerable<User> GetList()
2 {
3     return _context.Users.Include(u => u.Plan);
4 }
```

## שימי לב

הפונקציה `Include` נמצאת ב-namespace של `Microsoft.EntityFrameworkCore`.

לכן, אם את מנסה להשתמש בה ממחלקת `Repository` ולא מקבלת אותה ב-IntelliSense (השלמה אוטומטית), זה בגלל שחסר `using`.

תוכלי להקליד את שם הפונקציה ידנית בצורה מדויקת ולצפות להצעה המהירה להוספת `using`, או פשוט להוסיף את ה-`using` ידנית ואז לקבל את ההשלמה האוטומטית.

אם בחרנו להגדיר את הקשר משני הצדדים וכל אובייקט מכיל את השני, תהיה לנו בעיה קטנה כשנשתמש ב- `Include`. למעשה, המבנה שנוצר הוא מעגלי. כאשר `User` מכיל `Plan` שמכילה רשימה של `User` שכל אחד מכיל `Plan` וחוזר חלילה....

בתוך העולם של .NET. זה מסתדר והכל טוב, אבל כאשר האובייקטים נשלחים לקליינט הם עוברים סריאליזציה ל-JSON ושם משהו משתבש 😞.

אם נריץ ונפעיל פונקציה שמחזירה אובייקט מקונן עם `Include` נקבל שגיאה כזו:

| Code                | Details  |
|---------------------|--|
| 500<br>Undocumented | <p>Error: response status is 500</p> <p>Response body</p> <pre>System.Text.Json.JsonException: A possible object cycle was detected. This can either be due to a cycle or if the object depth is larger than the maximum allowed depth of 22. Consider using the ReferenceHandler.Preserve on JsonSerializerOptions to support cycles. Path: \$.Plan.Users.Plan.Users.Plan.Users.Plan.Users.Plan.Users.Plan.Users.Plan.Users.Plan.Users.Plan.Users.Plan.Users.Id    at System.Text.Json.ThrowHelper.ThrowJsonException_SerializerCycleDetected(Int32 maxDepth)    at System.Text.Json.Serialization.JsonConverter`1.TryWrite(Utf8JsonWriter writer, T&amp; value, JsonSerializerOptions options, WriteStack&amp; state)    at System.Text.Json.Serialization.Metadata.JsonTypeInfo`1.GetNumberAndWriteJson(Object obj, WriteStack&amp; state, Utf8JsonWriter writer)    at System.Text.Json.Serialization.Converters.ObjectDefaultConverter`1.OnTryWrite(Utf8JsonWriter writer, T&amp; value, JsonSerializerOptions options, WriteStack&amp; state)    at System.Text.Json.Serialization.JsonConverter`1.TryWrite(Utf8JsonWriter writer, T&amp; value, JsonSerializerOptions options, WriteStack&amp; state)    at System.Text.Json.Serialization.Converters.ListOfTConverter`2.OnWriteValue(Utf8JsonWriter writer, TCollection value, JsonSerializerOptions options, WriteStack&amp; state)    at System.Text.Json.Serialization.JsonCollectionConverter`1.OnTryWrite(Utf8JsonWriter writer, TCollection value, JsonSerializerOptions options, WriteStack&amp; state)</pre> |

הבעיה מתרחשת במנגנון של JSON Serializer שנכנס לפעולה בהחזרת ה-Response לקליינט והוא זה שמעביר את הנתונים לפורמט של JSON. הפתרון הוא פשוט. נוסיף ב- `Program.cs` הגדרה להתעלם מהפניה מעגלית.

נשדרג את השורה הזו בקוד:

| Program.cs |   |
|------------|---|
| 1          | <code>builder.Services.AddControllers();</code> |

ונוסיף לה את הקוד שלהלן:

| Program.cs |  |
|------------|--|
| 1          | <code>builder.Services.AddControllers().AddJsonOptions(options =&gt;</code>                  |
| 2          | <code>{</code>   |
| 3          | <code>options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;</code> |
| 4          | <code>options.JsonSerializerOptions.WriteIndented = true;</code>                             |
| 5          | <code>});</code>   |



נשים לב שאם לא נבצע Include לא יהיה לנו שום מידע על האובייקט המקושר, אפילו לא מה ה-Id שלו. לכן, מומלץ להוסיף מאפיין שיחזיק את המפתח הזר עצמו, במקרה שלנו User.PlanId.

## User.cs

```

1 public class User
2 {
3     public int Id { get; set; }
4
5     public string FirstName { get; set; }
6
7     public string LastName { get; set; }
8
9     //Foreign Key
10    public int PlanId { get; set; }
11
12    public Plan Plan { get; set; }
13 }
```

המאפיין PlanId יקבל את ערכו משדה PlanId בטבלת Users. אם נבצע Include נקבל את המפתח גם בו וגם במאפיין User.Plan.Id.

## הערה

לא ניתן להסתפק רק במאפיין של Foreign Key ליצירת הקשר בטבלאות. EF לא יוכל להסיק ממנו בלבד את הקשר שעליו ליצור. עם זאת, מאפיין זה הוא חלק מההגדרה של הקשר. EF לא יצור שדה חדש בטבלה עבורו אלא יבין שהוא מייצג את שדה המפתח הזר.

## יחיד ליחיד

קשר מסוג יחיד ליחיד חובה להגדיר בשני האובייקטים הקשורים. כמובן, שניהם יכילו מאפיין של אובייקט בודד ולא רשימה.

בדוגמא שלנו לכל User יש רשומה אחת בלבד של הגדרות תצוגה (UserSettings).

## User.cs

## UserSetting.cs

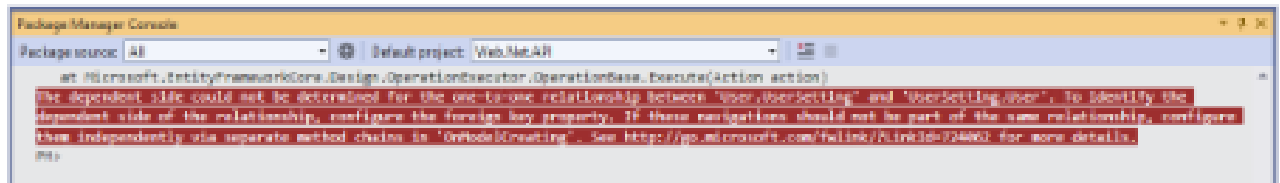
## User.cs

```

1 public class User
2 {
3     public int Id { get; set; }
4
5     public string Name { get; set; }
6
7     public string Email { get; set; }
8
9     public string Password { get; set; }
10
11    //One-To-Many
12    public int PlanId { get; set; }
13
14    public Plan Plan { get; set; }
15
16    //One-To-One
17    public UserSetting UserSetting { get; set; }
18 }
```

חובה להגדיר את המפתח הזר באובייקט שבו נרצה שיפוע, כיוון שבקשר יחיד ליחיד זה בעצם לא משנה...  
כאשר EF לא יכול להסיק בוודאות איך ליצור את המבנה בטבלאות הוא לא לוקח אחריות לבחור בעצמו.  
התפקיד שלנו לספק לו הגדרה דטרמיניסטית (מוחלטת) של מה אנחנו מצפים שיעשה.

אם לא נגדיר את המפתח הזר, נקבל שגיאה כזו כשנריץ את הפקודה `Add-Migration`.



בדרך כלל בקשר של רבים לרבים יש לנו אובייקט מקשר שמכיל עוד שדות מלבד מפתחות הקשר.  
לדוגמא: רופא, פציינט ותור.

לכל רופא הרבה פציינטים, לכל פציינט הרבה רופאים (רק בבריאות... 🤒)  
כלומר, בין פציינט לרופא קיים קשר של רבים לרבים.

האובייקט המקשר הוא תור שמכיל רופא ופציינט עם נתונים נוספים כמו: תאריך, שעה, מחיר, סוג וכדומה.

אבל לפעמים, אין צורך בשדות נוספים כלל. כמו בדוגמא שלנו בין User ל- Team יש קשר של רבים לרבים ואין מה להוסיף עליו.

מצד ה-DB עדין נצטרך טבלת קשר כיון שב-DB טבלאי לא ניתן לייצג קשר של רבים לרבים בצורה ישירה (כיון שאי אפשר להכיל רשומה בתוך שדה).

טבלת הקשר מכילה רק את המפתחות של שתי הטבלאות המקושרות וכך אנחנו משטחים את הקשר של רבים לרבים והופכים אותו ליחיד לרבים.

כאמור, באובייקטים אין לנו את הבעיה של הכלת רשומה בשדה. אובייקט יכול להכיל אובייקט. לכן, כאשר אין שדות נוספים בטבלת הקשר, ניתן להגדיר קשר של רבים לרבים בין שני אובייקטים באופן ישיר בלי אובייקט מקשר.

נגדיר בשני הצדדים מאפיין של רשימה.

User.cs Team.cs

User.cs

```

1 public class User
2 {
3     public int Id { get; set; }
4
5     public string Name { get; set; }
6
7     public string Email { get; set; }
8
9     public string Password { get; set; }
10
11     //One-To-Many
12     public int PlanId { get; set; }
13
14     public Plan Plan { get; set; }
15
16     //One-To-One
17     public UserSetting UserSetting { get; set; }
18
19     //Many-To-Many
20     public List<Team> Teams { get; set; }
21 }
```

כאשר EF מזהה הגדרה כזו של רבים לרבים, הוא יוצר טבלת קשר ב-DB שמכילה את המפתחות הזרים של שני האובייקטים משני הצדדים.

#### הערה

אין בעיה שאובייקט אחד מכיל כמה אובייקטים אחרים במקביל.  
אבל אם אובייקט מקושר **לאותו** אובייקט פעמיים, EF לא יוכל להסיק בעצמו איך ליצור את הקשרים ונצטרך להגדיר אותם ידנית כדי לפתור את העמימות.  
דוגמא למקרה כזה היא טבלת עובדים וטבלת מנהלים כאשר לכל עובד יש שני מנהלים: מנהל ישיר ומנהל כללי.

בעת ביצוע של יחידת קוד, מערכת ההפעלה מקצה Thread (תהליכון) עם חתיכת זכרון ומעבד שעסוקים בביצוע יחידת הקוד.

כאשר הקוד המדובר כתוב ב-C# הוא רץ בצורה סינכרונית.

קוד סינכרוני פירושו שכל שורה בקוד מתבצעת רק לאחר שהביצוע של השורה הקודמת הסתיים. כלומר, הקוד לא ממשיך להתבצע אלא ממתין בכל שורה לסיומה ורק אז עובר הלאה לשורה הבאה.

לעיתים, הקוד מבצע פעולה שאינה תלויה בו, כלומר, היא מתבצעת על ידי Thread אחר. בזמן ביצוע פעולה כזו ה-Thread הראשי של הקוד בעצם לא עושה כלום. אך כיון שמדובר בקוד סינכרוני, הוא חייב להמתין עד שהיא תסתיים כדי להמשיך לשורה הבאה.

יש שני סוגים של פעולות שגורמות ל-Thread להמתין:

- פעולות שכרוכות ב-I/O כמו גישה ל-DB, קריאת אינטרנט או קריאה וכתיבה לקובץ.
- פעולות שכרוכות ב-CPU כלומר דורשות זמן עיבוד ממושך כמו חישוב מורכב.

המתנה זו של ה-Thread גורמת לבזבז משאבים לחינם, מה שכמובן מאוד לא רצוי.

הפתרון הוא להפעיל פעולות אלו בצורה אסינכרונית.

קוד אסינכרוני מיידיע את המנגנון שאחראי על הקצאת התהליכונים שהוא ממתין ופנוי לקבל משימות אחרות בזמן ההמתנה, וכך הוא מאפשר למערכת ההפעלה לנצל את התהליכונים ברמה מקסימלית ולתת מענה ליותר פעולות במקביל.

באפליקציית סרבר קוד אסינכרוני ישפר מאוד את הביצועים. מתוקף תפקידה, אפליקציית סרבר משרתת קליינטים רבים בו זמנית. ככל שהיא תוכל לטפל ביותר בקשות במקביל, היא תציג ביצועים טובים יותר.

בסופו של דבר, אי אפשר להימלט מהתלות בחומרה של כמות זיכרון ומעבד. אם השרת שעליו מותקנת אפליקציית הסרבר מסוגל להקצות פיזית 1000 תהליכונים בכל רגע נתון, זה מספר הבקשות שהוא יוכל לבצע.

עם קוד אסינכרוני, השרת יוכל לתזז בין מספר גדול יותר של בקשות על ידי שיחליף בין התהליכונים בכל קטע שבו יש המתנה.

כך, עם חומרה של 1000 תהליכונים השרת יתן מענה ל-1500 בקשות בלי לפגוע בביצועים. (כמובן המספרים לצורך ההדגמה בלבד.)

שווה, לא?

[חשוב להדגיש](#)

השיפור בביצועים הוא ברמת האפליקציה כולה ולא ברמת הבקשה. כל Request לעצמו לא יהנה משיפור ביצועים ויקח לו את אותו פרק זמן להתבצע.

## Async Example

```

1 var httpClient = new HttpClient();
2
3 var html = await httpClient.GetStringAsync("https://github.com/");
4
5 Console.WriteLine(html.Count());

```

הקוד שלעיל שולח Http Request שפונה לכתובת <https://github.com/> ומקבל את התגובה כ-string. קריאת אינטרנט היא אחת הדוגמאות הנפוצות לקוד שנכון להגדיר כאסינכרוני, ואכן המחלקה HttpClient (מובנית של .NET) אפילו לא מציעה את הפונקציות שלה בגירסה סינכרונית, אלא רק אסינכרונית.

התוספת המיוחדת שיש לנו היא המילה השמורה await לפני הקריאה לפונקציה האסינכרונית. מה זה await ואיך בכלל עובד קוד אסינכרוני?

כאשר מנגנון ביצוע הקוד מגיע להפעלת פונקציה שמוגדרת כאסינכרונית, הוא לא מבצע אותה ב-Thread הראשי אלא מקצה לה Task עצמאי שרץ בנפרד מה-Thread הראשי. באופן הזה, ה-Thread הראשי לא נתקע בהמתנה לסיום הפעולה המתבצעת.

אלא שלמעשה, כמעט תמיד, אין מצב שנמשיך לבצע את הקוד בלי לסיים קודם את הפעולה המבוקשת. המשך הקוד בדרך כלל תלוי בפעולה זו. וגם אם לא, אנחנו רוצים לדעת שהיא הסתיימה וכיצד (בהצלחה או בכשלון).

לא נוכל להרשות לעצמנו לזרוק את הביצוע ל-Task נפרד בלי לוודא מה קרה איתו ומה יצא ממנו. לשם כך, נועד ה-await. באמצעות ה-await אנחנו מסמנים ל-Thread הראשי להמתין עד שה-Task יסתיים ורק אז להמשיך.

וכאן הבן שואל: אם ככה, מה הסיפור? 😞 אם ממילא נמתין לסיום הביצוע, הרי שהקוד חוזר להיות סינכרוני ובשביל מה כל בלבול המוח הזה? התשובה נעוצה בהדגשה שכבר הובאה למעלה.

אכן, ברמת **הבקשה** לא נחווה שיפור ביצועים ולא נרויח יעילות וחסכון בזמן. אבל ברמת **האפליקציה** נוכל לשרת מספר רב יותר של בקשות עם אותה כמות של תהליכונים.

כיון שהקוד מוגדר אסינכרוני, כאשר אנחנו אומרים ל-Thread להמתין באמצעות await, אנחנו בעצם משחררים אותו לקבל משימות אחרות ממערכת ההפעלה, עד שתסתיים הפעולה ותחזור התוצאה מהפונקציה האסינכרונית.

אז נכון שהפונקציה הבודדת תיקח את אותו משך זמן, אבל בינתיים, על הדרך, ה-Thread ביצע עוד כמה משימות מבקשות אחרות, מה שנקרא מולטי טסקינג...

בשורה תחתונה, באופן שגרתי נגדיר פונקציות שכרוכות ב-I/O או ב-CPU כאסינכרוניות, וכך נרויח ביצועים טובים יותר בלי מאמץ ועם שינויים מינוריים בקוד.

אין טעם להפוך את כל הקוד לאסינכרוני, כיון שלמנגנון הקצאת התהליכונים יש גם כן מחיר בביצועים ולכן לא נעמית עלינו לחינם ונכריז על תהליכונים אסינכרוניים בזמן שבפועל הם לא באמת מגיעים למצב של המתנה. לעומת זאת, כאשר תהליכון אכן נזקק להמתין, עלות ההכנסה שלו למנגנון הקצאת התהליכונים היא זניחה ביחס לחיסכון שמושג מניצול מקסימלי של התהליכון.