dependency injection

כדי לכתוב קוד שקל לתחזק נשתמש בטכנולוגיית "הזרקת תלות" - dependency injection ש. net מספקת.

כשאנו כותבים קוד בו רכיבים תלויים זה בזה לרוב יהיה קשה לשנות דרישה או להוסיף קוד. וכן לבצע בדיקות.

נלמד מהדוגמה הבאה :

הקוד מיצג 2 מחלקות של הזמנה order ו orderItem. המחלקה orderManager מכילה את הפונקציה Transmit ששולחת את ההזמנה לתהליך בשירות אחר- OrderSender. היא מקבלת את ההזמנה כפרמטר. ב OrderSender ניתן לראות שהפונקציה send מבצעת סרילזציה להזמנה ושולחת אותה בPOST לאיזשהו מיקרו סרוויס שאחראי על התהליך.

מה יקרה אם נצטרך לשנות את דרך שליחת ההזמנה? לדוגמה כשנרצה גם לשלוח את ההזמנות לכתובת מייל מסוימת או לשלוח למיקרו סרוויס אחר שמשתמש בפרוטוקול אחר. מכיון שהorderManager תלוי בorderSender נהייה מוכרחים לשנות באיזושהי דרך את שתי המחלקות כדי לתמוך במספר סוגי שליחה.

השינוי של הרכיב הlower-level- הרמה היורדת לפרטים ישפיע על רכיב הHighlevel.

זהו רק מקרה פשוט. ההשפעה שעשויה להיות לתלות בתרחיש מורכב יותר עם מרכיבים תלויים רבים, יכולה להפוך לבלגן ענק.

עקרונות לעיצוב נכון של הקוד

OOD-Object Oriented Design אלו כללים לכתיבת קוד.

ביניהם כתיבת קוד הניתן לשימוש חוזר, גמיש לשינויים והרחבות, קל לניהול, מטפל בשגיאות בצורה טובה, ומתאים לרכיבי חומרה (למשל מערכות הפעלה) שונים.

בתכנות נכון ידועים ר"ת solid כעקרונות לעיצוב. האחרון מבניהם DIP .

עקרון נוסף הוא הIOC

השיטה ליישם את העקרונות הנ"ל היא DI.

Dependency Inversion Principle

קורס ASP.NET CORE WEB API
כל.הזכויות.שמורות

Dependency Inversion Principle, "היפוך התלות", מציע דרך להקל על בעיית התלות ולהפוך אותה לניתנת יותר לניהול, כדי שבזמן של שינוי של אחת התלויות לא נצטרך לשנות גם את האובייקט שצורך את התלויות. עיקרון זה קובע כי:

1. מודולים בhighlevel לא צריכים להיות תלויים במודולים בlowlevel. שניהם צריכים להיות תלויים ב abstractions.
2. abstractions לא צריכות להיות תלויות בפרטים. הפרטים צריכים להיות תלויים ב abstractions. לדוגמה interface.

High Level Classes → Abstraction Layer ← Low Level Classes

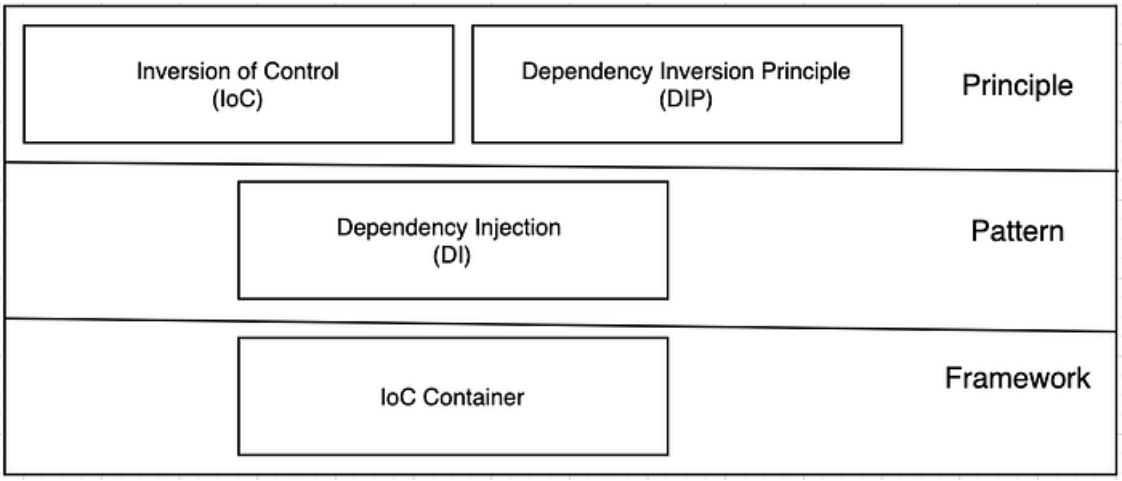
מונח נוסף בנושא התלות הוא הIOC.

Inversion of control , זוהי דרך ליישם את עקרון הDIP . "היפוך שליטה" זהו מנגנון המאפשר לרכיבים בlevel- high להסתמך על האבסטרקציה ולא על רכיב קונקרטי בlow-level. שיהיה גורם חיצוני שאחראי לאורך חיי האובייקט, התלויות השונות. כך, כל אובייקט יהיה פנוי לעסוק במטרה שלו. חיי האובייקט יכולים להיות לאורך כל בקשה, לאורך כל חיי השרת ועוד-נפרט בהמשך.

Dependency Injection זוהי הצורה ליישם את היפוך השליטה. ע"י שמאפשרת הזרקה של רכיב התלות לרכיב שתלוי בו.

נשים לב ב DI יש לנו 3 חלקים , התלות - dependency, הרכיב התלוי - dependent וה – container החלק שאחראי לנהל ולהזריק את התלויות למי שצריך אותם.

השימוש ב DI נכון בכל מסגרת של תכנות. ב. net core יש תמיכה מובנית ב DI שמפשיטה את הניהול שלו.



IOC container

הפיצ'רים הבסיסיים ש.net מספק בקונטיינר ה IOC:

רישום עמ"נ למפות את סוגי התלויות למופעים הנכונים

הקונטיינר אחראי על יצירת והזרקת התלות למחלקות שדורשות אותה, כך שאין צורך לנהל מופעים שלהם ידנית.

הקונטיינר מגדיר את אורך חיי המופע של התלות, והוא זה שאחראי למחוק אותו מהזיכרון בסיום.

הרכיבים האלו, נקראים גם services, שירותים, ומתחלקים לשני סוגים: שירותים שהם כחלק מהתשתית, ושרותי יישום שהמתכנת יוצר.

Framework services : בprogram יש אובייקט builder המכיל אוסף של services מסוג IServiceCollection , עמ"נ להשתמש בשרות קיים נוסיף אותו לאוסף לדוגמה:

```
Builder.services.AddAuthentication()
```

Generic services: נשתמש בפונקציה Add[singleton/scoped/trasnsient] ונציין את התלות(interface) ואת המופע (מחלקה שיורשת מהinterface) שיש ליצור עבורה. לדוגמה:

```
Builder.services.AddSingelton<ilog,mylog>();
```

Service lifetimes

הקונטיינר נותן לשלוט על אורך חיי השרות שנרשם ובצורה אוטומטית ידאג לנקות מהזיכרון לפי הזמן שנגדיר.

Singleton– המופע שיוצר עבור התלות יחיה לאורך כל חיי האפליקציה מהפעם הראשונה שcontroller או פונקציה מסוימת התלויה בו תופעל. בכל מחזור חיי האפליקציה יהיה ממנו מופע בודד לתלות. כדי להשתמש נפעיל את הפונקציה AddSingleton().

Transient –(חולף) השירות יוצר בכל פעם שתיהיה בקשה למופע שלו. השרות יוזרק לבנאי של המחלקה כמספר המופעים של המחלקה. נשתמש בפונקציה AddTransient()

Scoped - ייווצר מופע בכל request זה שימושי בcontext . AddScoped()

בחירת הlifetime הנכון לכל service חשובה להתנהגות נכונה של האפליקציה ולניהול טוב של המשאבים.

נחזור לדוגמה:

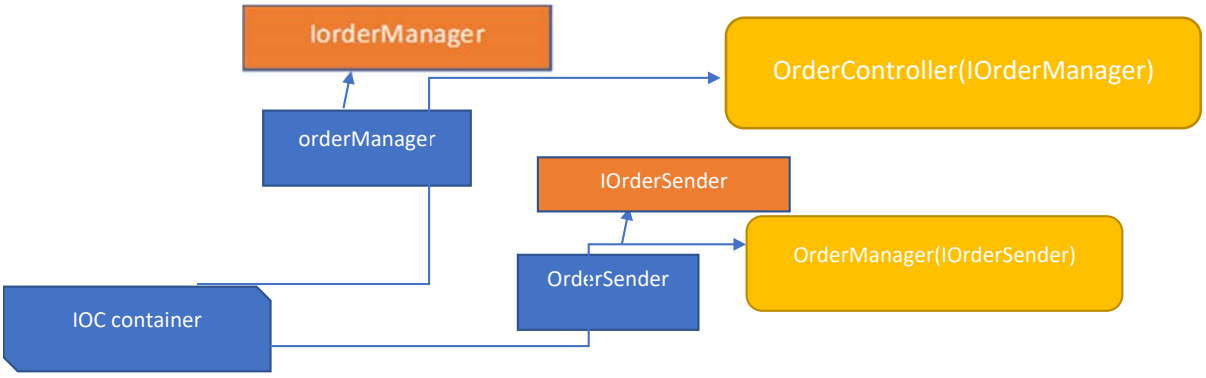
נגדיר interface לשליחה בשם IOederSender שמגדיר פונקציית SEND המקבלת אובייקט של הזמנה. ו interface נוסף בשם IOrderManager ממנו תירש המחלקה OrderManager. בבנאי של המחלקה OrderManager נקבל משתנה מסוג IOederSender במקום שהפונקציה Transmit תייצר מופע של OrderSender. כך שברנו את התלות של הorderManager בשליחה של HTTP ויצרנו

שכבה אבסטרקטית ביניהם, המחלקה OrderManager מסוגלת לקבל גם סרוויס שליחה אחר ובלבד שירש מIOrderSender. כעת עלינו לספר לקונטיינר המנהל את התלויות איך עליו להזריק מופע מתאים לctor. נוסיף את השורות הבאות בprogram.cs :

```
Builder.services.AddScoped<Interfaces.IOrderSender, HttpOrderSender>();
Builder.services.AddScoped<Interfaces.IOrderManager, OrderManager>();
```

רשמנו את התלויות, בקשנו מהקונטיינר שיצור מופע של HttpOrderSender בכל פעם שתהיה בקשה ל IOrderSender וכנ"ל OrderManager לIOrderManager .

כעת נפתח את ה OrderController.cs ונשנה את הבנאי שלו שיקבל פרמטר מסוג הInterface.



לסיכום:

מכיון שאנו לא רוצים לייצר תלות בין רכיבים, נדאג לפרק את הקשר בין רכיבים באפליקציה שכל רכיב יעמוד בפני עצמו ויהיה בלתי תלוי ברכיב אחר. כך נוכל בקלות בלי הרבה עבודה להחליף ברכיב אחר טוב יותר או משודרג יותר או חינמי וכו בנוסף מאוד קל לצרוך רכיבים שבנויים כservices שעומדים בפני עצמם.

השלבים:

1. יצירת Interface/base class –שכבה אבסטרקטית.
2. הגדרת פרמטר מסוג ה abstract בבנאי של צורך הסרוויס.
3. רישום של האובייקט שיוזרק לכל סוג בServiceCollection.