**Riepilogo della sezione**

Accoppiamento forte e debole dei componenti

Quando costruiamo un'applicazione, dovremmo sempre fare in modo che i vari componenti siano **sostituibili individualmente**, un po' come sostituire la candele di un'automobile non comporta la sostituzione del motore o della carrozzeria.

Questo è un accorgimento importante da tenere a mente perché **i requisiti cambiano nel tempo** e il nostro committente ci chiederà di modificare il comportamento dell'applicazione in alcuni punti. Perciò dobbiamo farci trovare pronti per acconsentire alla sua richiesta con il minimo dispendio di tempo ed energie.

Nella scorsa sezione abbiamo fatto interagire due componenti: il CoursesController e il CourseService. Rivediamo qual era il codice che avevamo usato all'interno dell'action Index.

1. public IActionResult Index()
2. {
3. var courseService = new CourseService();
4. List<CourseViewModel> courses = courseService.GetCourses();
5. return View(courses);
6. }

Questo codice presenta un problema perché rende il CoursesController **fortemente accoppiato** al CoursesService. Cioè, nel momento in cui vogliamo sostituire il CourseService con un'altra implementazione, che magari attinge i dati da un'altra fonte, dobbiamo tornare all'interno del CoursesController e sostituire a mano tutte le occorrenze di CourseService. In grandi applicazioni, questo può essere un lavoro dispendioso e propenso a errori. È come se le candele dell'automobile fossero inglobate all'interno del motore, costringendoci così a sostituire l'intero motore quando le candele si logorano dopo appena 50.000 km percorsi.  
È per evitare questo problema che gli ingegneri meccanici hanno progettato sul motore un alloggiamento filettato facilmente accessibile dall'esterno, che rappresenta l'**interfaccia di collegamento** con la candela.

Immagine che contiene Ricambio auto, chiusura, strumento, Ferramenta

Descrizione generata automaticamenteUna candela vecchia può essere facilmente sostituita con uno nuova semplicemente svitandola. Un semplice accorgimento che fa risparmiare molto tempo e denaro dato che la candela si usura molto prima del motore. [Foto di Ascaron su Wikimedia Commons](https://commons.wikimedia.org/wiki/File:Spark_plugs_3.jpg).

Anche noi, nella nostra applicazione, dovremmo fare in modo che i componenti siano facilmente sostituibili. Per questo, il linguaggio C# ci viene incontro con le **interfacce**.

Iniziamo definendo un'interfaccia che definisca i membri pubblici del servizio applicativo .

1. public interface ICourseService
2. {
3. List<CourseViewModel> GetCourses();
4. CourseDetailViewModel GetCourse(int id);
5. }

Come si vede, l'interfaccia descrivere semplicemente quali sono i membri pubblici che una classe dovrà implementare e **non il modo in cui deve implementarli**. Il *modo*, invece, lo definirà la nostra classe CourseService che è un'**implementazione concreta** dell'interfaccia che abbiamo appena scritto.

1. //Ora CourseService implementa ICourseService
2. public class CourseService : ICourseService
3. {
4. public List<CourseViewModel> GetCourses()
5. {
6. //Omissis...
7. }
8. public CourseDetailViewModel GetCourse(int id)
9. {
10. //Omissis..
11. }
12. }

Poi rechiamoci nel CoursesController e facciamo in modo che abbia una dipendenza da tale interfaccia. Le dipendenze le possiamo esprimere come parametri del costruttore.

1. public class CoursesController : Controller
2. {
3. private readonly ICourseService courseService;
5. //Ecco il costruttore. Un'istanza di un ICourseService dovrà essere fornita dall'esterno.
6. //Questo è equivalente a una candela che viene avvitata sull'esterno del motore.
7. public CoursesController(ICourseService courseService)
8. {
9. //La copiamo su un campo privato in modo che sia accessibile dalle action.
10. this.courseService = courseService;
11. }
13. public IActionResult Index()
14. {
15. //Usiamo l'istanza del courseService che avevamo ricevuto nel costruttore
16. List<CourseViewModel> courses = courseService.GetCourses();
17. return View(courses);
18. }
19. }

Ora il nostro CoursesController è diventato **debolmente accoppiato** al servizio applicativo che fornisce i corsi perché non dipende più da un'implementazione concreta ma da un'interfaccia. Infatti, se volessimo sostituire CourseService con un'altra implementazione lo potremo fare senza toccare il controller.

Ci basta andare nel metodo ConfigureServices della classe Startup e indicare ad ASP.NET Core quale classe concreta deve costruire quando incontra componenti che dipendono dall'interfaccia ICourseService. Questo è necessario perché le interfacce non possono essere costruite.

1. public void ConfigureServices(IServiceCollection services)
2. {
3. services.AddScoped<ICourseService, CourseService>();
4. //Nel momento in cui vogliamo cambiare implementazione, ci basterà cambiare la riga con:
5. //services.AddScoped<ICourseService, AltraImplementazioneDiCourseService>();
6. }

Questa pratica è chiamata **dependency injection**, perché le dipendenze di un componente vengono passate dall'esterno (si dice anche "iniettate"). L'utilità di questa pratica risulterà ancor più evidente in futuro, quando affronteremo l'argomento del testing.

Ciclo di vita dei servizi

ASP.NET Core, grazie al suo meccanismo di dependency injection, si preoccuperà al posto nostro di costruire le istanze di servizi come CourseService da cui i controller e gli altri componenti dipendono.

Ma... quand'è che ASP.NET Core distrugge le istanze ha creato, liberando così la memoria?  
Dipende dal **ciclo di vita** che abbiamo indicato quando abbiamo registrato il servizio dal metodo ConfigureServices.

* Se usiamo services.AddTransient<ICourseService, CourseService>(), l'istanza potrà essere distrutta quando viene distrutto il componente che l'ha utilizzata, cioè quando il controller ha terminato la sua esecuzione;
* Se usiamo services.AddScoped<ICourseService, CourseService>(), l'istanza potrà essere distrutta solo al termine della richiesta HTTP corrente. Questo vuol dire che ASP.NET Core passerà la stessa identica istanza ad altri componenti che ne abbiano bisogno, almeno fintanto che la richiesta HTTP è ancora in corso. Come vedremo in seguito, questo può essere un ottimo modo per riutilizzare efficacemente servizi che sono "costosi" da costruire come il DbContext di Entity Framework Core che vedremo in seguito;
* Se usiamo services.AddSingleton<ICourseService, CourseService>(), l'istanza potrà essere distrutta solo quando l'applicazione viene arrestata. Questo vuol dire che ASP.NET Core creerà al massimo una sola istanza e la passerà a ogni componente che ne abbia bisogno, anche per richieste HTTP che arrivano da utenti diversi. Bisogna fare attenzione perché più thread contemporaneamente accederanno alla stessa istanza, causando severi effetti collaterali se non è stata progettata per essere *thread-safe*.