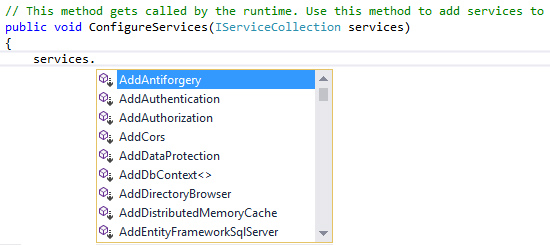
# Gestire il ciclo di vita dei servizi in ASP.NET Core

# Nel precedente script (<https://www.aspitalia.com/script/1229/Configurare-Middleware-Servizi-ASP.NET-Core.aspx>) abbiamo iniziato ad esplorare la classe **Startup** di un'applicazione ASP.NET Core ed abbiamo visto come il suo metodo *ConfigureServices* sia usato per aggiungere dei *servizi*, ovvero classi a cui abbiamo affidato dei compiti specifici come compiere delle elaborazioni o interagire con una risorsa. Usando degli extension method, ci è possibile aggiungere una varietà di servizi sviluppati da Microsoft e da terze parti, di cui parleremo nei prossimi script.



Per il momento è importante sapere che ad un'applicazione ASP.NET Core possiamo aggiungere anche servizi applicativi ed infrastrutturali creati da noi. Vediamo l'esempio di un servizio personalizzato che si occuperà di inviare e-mail agli utenti.

public class SmtpEmailSender : IEmailSender

{

public async Task SendEmailAsync(string email, string subject,

string message)

{

*//TODO: logica di invio e-mail*

}

}

Dove *IEmailSender* è un'interfaccia così definita:

public interface IEmailSender

{

Task SendEmailAsync(string email, string subject, string message);

}

**Aggiungere un servizio** come *SmtpEmailSender* ad un'applicazione ASP.NET Core significa delegare la creazione delle sue istanze al meccanismo di **dependency injection**, che si occuperà anche del loro rilascio al termine del proprio **ciclo di vita**. In questo modo, si verificheranno due conseguenze vantaggiose:

* Non dovendo usare la parola chiave *new* per costruire i servizi, siamo incentivati a **disaccoppiare** i vari componenti del nostro software affinché siano più manutenibili e testabili;
* Riduciamo il rischio che, dimenticando l'invocazione ad un Dispose(), restino risorse attive in memoria. Le istanze dei servizi vengono rilasciate per noi.

E' possibile aggiungere servizi con diversi cicli di vita e per fare ciò usiamo uno dei 3 extension method di *IServiceCollection*:

* **AddSingleton** ci garantisce che verrà creata e riutilizzata una singola istanza per l'intera applicazione. E' consigliato per servizi che non mantengono uno stato interno, come le classi che usiamo per recupare valori di configurazione, per scrivere delle righe di log o per inviare e-mail. Potenzialmente, i metodi e le proprietà di un servizio *singleton* verranno usati da vari thread contemporaneamente e quindi è essenziale che la sua implementazione sia *thread-safe*. Utile quando vogliamo regolare l'accesso ad una risorsa da parte della nostra applicazione, come ad esempio limitare il numero di e-mail inviate nell'unità di tempo;
* **AddScoped** lo usiamo per riutilizzare l'istanza di un servizio nel contesto di una richiesta HTTP. E' consigliato per oggetti che non sono *thread-safe* o che mantengono uno stato interno come il *DbContext* di *Entity Framework*. In questo modo, dato che la creazione di un'istanza del servizio avviene una sola volta per ogni richiesta HTTP, riusciamo a razionalizzare alcuni "costi prestazionali", come quelli legati all'ottenimento di una connessione al database;
* **AddTransient** causerà la creazione di una nuova istanza del servizio ogni volta che un componente della nostra applicazione lo richiede. Dato che nessuna istanza verrà mai riutilizzata, è un ciclo di vita indicato per servizi semplici, di veloce creazione come un client HTTP usato per inviare richieste ad una API remota.

Come esempio, andiamo a registrare il servizio *SmtpEmailSender* con il metodo *AddSingleton*. Apriamo il file *Startup.cs* del progetto e scriviamo:

public void ConfigureServices(IServiceCollection services)

{

*//Qui configurazione di altri servizi del framework*

*//Qui aggiungiamo il nostro servizio SmtpEmailSender*

services.AddSingleton<IEmailSender, SmtpEmailSender>();

*//Non è obbligatorio che il servizio implementi un'interfaccia*

*//come IEmailSender, ma è una pratica utile per il*

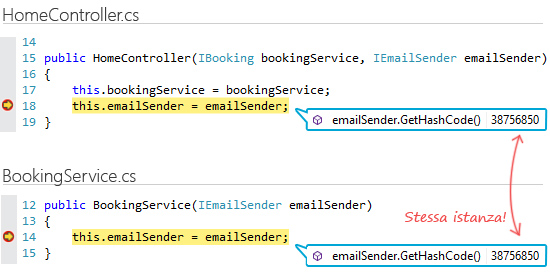
*//disaccoppiamento e lo unit testing*

*//In alternativa, avremmo potuto scrivere semplicemente:*

*//services.AddSingleton<SmtpEmailSender>();*

}

Il meccanismo di *dependency injection* di ASP.NET Core si occupa di creare un'istanza di *SmtpEmailSender* e fornire quella stessa istanza a tutti i componenti che manifestino una dipendenza da *IEmailSender* nel proprio costruttore. Qui di seguito vengono appunto illustrati due di questi componenti: un controller ed una classe di business.



Il meccanismo di *dependency injection* si occupa anche di **rilasciare le istanze** al termine del loro ciclo di vita, senza che sia necessario alcun intervento da parte nostra. Ad esempio, nel caso di servizi aggiunti con il metodo *AddScoped*, le istanze verranno rilasciate automaticamente alla conclusione della richiesta HTTP. Per i servizi che implementano l'interfaccia *IDisposable*, ASP.NET Core invoca il loro metodo **Dispose()** affinché le eventuali risorse *unmanaged* possano essere liberate tempestivamente.