**78. Riepilogo del servizio di logging**

Il servizio di logging

Anche se la nostra applicazione è stata adeguatamente testata, quando l'andiamo a mettere in produzione potranno comunque verificarsi delle **situazioni impreviste**. Ad esempio, un bug non ancora scoperto o dei problemi transitori di connessione al database impediranno all'applicazione di funzionare normalmente.

È importante tenere una traccia di questi accadimenti e di ciò che si è verificato nei momenti immediatamente precedenti, così che possiamo **diagnosticare il problema** più facilmente e trovare delle contromisure per evitare che si verifichi di nuovo in futuro.

Dunque, anche dopo che lo sviluppo di un'applicazione è stato "completato", abbiamo comunque il dovere di studiare il suo comportamento, identificare le criticità e **ridurre al minimo i disservizi** causati ai nostri utenti.

Il servizio di logging ILogger<T> ci serve appunto per **lasciare una traccia di eventi che reputiamo importanti** nella nostra applicazione. Come al solito, riceviamo il servizio nel costruttore di un nostro componente, come il servizio applicativo AdoNetCourseService.

1. public class AdoNetCourseService : ICourseService
2. {
3. private readonly ILogger<AdoNetCourseService> logger;
5. //Ricevo il servizio dal costruttore
6. public AdoNetCourseService(ILogger<AdoNetCourseService> logger)
7. {
8. //Conservo il riferimento al servizio su un campo privato...
9. this.logger = logger;
10. }
12. public async Task<List<CourseViewModel>> GetCoursesAsync()
13. {
14. //...poi scrivo un messaggio nel log per tracciare il fatto
15. //che un utente ha richiesto l'elenco dei corsi.
16. //Il messaggio verrà visualizzato in console.
17. logger.LogInformation("Course list requested");
19. //TODO: ottengo l'elenco dei corsi
20. }
21. }

Nella seguente immagine possiamo vedere che il nome AdoNetCourseService che abbiamo indicato come parametro di tipo T del servizio ILogger<T> sarà usato come *categoria*, ovvero come origine del messaggio di log. In questo modo riusciamo a determinare meglio la provenienza di ciascun messaggio.

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

In alternativa, se vogliamo essere noi stessi a decidere arbitrariamente il nome della *categoria*, possiamo usare il servizio ILoggerFactory.

1. public class AdoNetCourseService : ICourseService
2. {
3. private readonly ILogger logger;
5. //Ricevo il servizio dal costruttore
6. public AdoNetCourseService(ILoggerFactory loggerFactory)
7. {
8. //Creo un ILogger con una categoria dal nome arbitrario...
9. this.logger = loggerFactory.CreateLogger("Corsi");
10. }
12. public async Task<List<CourseViewModel>> GetCoursesAsync()
13. {
14. //...poi, come al solito, scrivo un messaggio nel log
15. logger.LogInformation("Course list requested");
17. //TODO: ottengo l'elenco dei corsi
18. }
19. }

Livelli di criticità dei messaggi di log

Il servizio di logging ci mette a disposizione vari metodi, ciascuno usato per scrivere messaggi a vari ***livelli* di criticità**. Eccoli ordinati dal più importante al meno importante:

* **LogCritical**: usato per tenere traccia di eventi che compromettono il corretto funzionamento dell'intera applicazione per tutti gli utenti, come ad esempio l'impossibilità di scrivere su disco;
* **LogError**: per errori che riguardano una situazione ben precisa, come ad esempio l'impossibilità di aggiornare uno specifico corso;
* **LogWarning**: per situazioni di errore che non necessariamente dipendono dalla nostra applicazione, come per esempio l'apertura di un corso che non è mai esistito nel database;
* **LogInformation**: per messaggi che confermano la corretta esecuzione di un'operazione, come ad esempio  l'acquisto di un corso da parte di uno studente;
* **LogDebug**: per messaggi che ci aiutano a diagnosticare un problema in produzione quando non riusciamo a identificarlo con altri mezzi. Usiamo questo metodo per tenere traccia di query SQL, numero di righe estratte dal database, e così via;
* **LogTrace**: per tenere traccia di ogni singolo passo operativo compiuto dalla nostra applicazione. Può essere utile per avere una trascrizione del comportamento interno di un metodo a scopo di verifica della sua conformità alla specifica.

Usare bene i livelli è importante perché i messaggi di log vengono scritti in console, che è una risorsa contesa dai vari thread della nostra applicazione. Di conseguenza, può anche rappresentare un collo di bottiglia e **deteriorare le prestazioni**, soprattutto quando siamo in produzione e abbiamo tanti utenti contemporanei.  
  
Perciò è importante che in produzione vengano scritti nel log **solo i messaggi con un alto livello di criticità**, come ad esempio da Warning in su. Perciò, apriamo il file appsettings.json e impostiamo questa configurazione.

1. {
2. "Logging": {
3. "LogLevel": {
4. "Default": "Warning"
5. }
6. }
7. }

Mentre invece, finché siamo in sviluppo, abbiamo tutta la libertà di scrivere nel log anche i messaggi meno importanti. Quindi apriamo il file appsettings.Development.json e aggiungiamo questa configurazione per ridefinire il livello per l'ambiente Development.

1. {
2. "Logging": {
3. "LogLevel": {
4. "Default": "Information"
5. }
6. }
7. }

Logging strutturato

Grazie al servizio di logging possiamo scrivere **messaggi strutturati**, che cioè tengono separato il testo dai propri valori. Nel seguente esempio, il messaggio definisce due segnaposto chiamati {query} e {error} che verranno valorizzati con i successivi due parametri forniti al metodo LogError.

1. try
2. {
3. DataSet dataSet = await db.QueryAsync(query);
4. //...
5. }
6. catch (Exception exc)
7. {
8. logger.LogError("Error while executing query {query}: {error}", query, exc.Message);
9. }

Questa funzionalità è utile soprattutto se si usano provider di logging di terze parti come **Serilog** (vedi lezione bonus della sezione 12) che ci permettono di tenere ben separati testi e valori, in modo che poi i nostri log possano addirittura essere esaminati con strumenti di analisi e reportistica.

Configurare i provider di logging

ASP.NET Core, per default, emette i messaggi nell'output della console ma possiamo aggiungere altri provider con il metodo ConfigureLogging nel punto in cui viene creato il web host, nella classe Program.

1. public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
2. WebHost.CreateDefaultBuilder(args)
3. .ConfigureLogging((context, builder) => {
4. //Qui impostiamo i provider di logging
5. })
6. .UseStartup<Startup>();

L'elenco dei provider supportati è disponibile in questa pagina della documentazione:

<https://docs.microsoft.com/it-it/aspnet/core/fundamentals/logging/?view=aspnetcore-2.2#provider-aliases>