**Riepilogo dei servizi di caching**

Il servizio di caching

La qualità di un'applicazione è in parte determinata dalle sue prestazioni: se un utente riesce a navigare velocemente all'interno del nostro sito, avrà meno ostacoli nel raggiungere il suo obiettivo e perciò la sua **esperienza di utilizzo risulterà positiva**.

Quando ci sono molti utenti contemporanei, le prestazioni della nostra applicazione potrebbero deteriorare, soprattutto se non abbiamo un server con delle caratteristiche hardware in grado di reggere il carico.

Prima di migliorare l'hardware del nostro server, però, dovremmo attuare degli accorgimenti che ci permettono di accontentare più utenti in maniera quasi "gratis". Se sfruttiamo il servizio di caching di ASP.NET Core, infatti, possiamo evitare di interrogare il database a ogni richiesta e mostrare all'utente un risultato già pronto, recuperato dalla memoria RAM dell'applicazione. Così manterremo ottime prestazioni anche con molti utenti contemporanei ma dobbiamo accettare il **compromesso** che tale risultato possa non essere il più aggiornato possibile.

Il servizio IMemoryCache ci permette di "salvare" degli oggetti nella memoria RAM, in modo che poi possano essere recuperati in seguito grazie a una **chiave** che li rappresenta. Questo è vantaggioso perché la memoria RAM è incredibilmente più rapida da leggere rispetto a un database.

Gli oggetti che dovremo valutare di mettere in cache sono quelli che devono essere visualizzati tali e quali da molti utenti, come il nostro elenco di corsi. Nel seguente esempio, quindi, vediamo come ricevere il servizio IMemoryCache dal costruttore di un nostro servizio applicativo.

1. public class MemoryCacheCourseService : ICachedCourseService
2. {
3. private readonly IMemoryCache memoryCache;
4. private readonly ICourseService courseService;
6. //Riceviamo il servizio IMemoryCache dal costruttore
7. //E così anche il servizio applicativo che ci permette di recuperare i dati dal database
8. public MemoryCacheCourseService(IMemoryCache memoryCache, ICourseService courseService)
9. {
10. //Conserviamo i riferimenti su campi privati
11. this.memoryCache = memoryCache;
12. this.courseService = courseService;
13. }
15. public Task<List<CourseViewModel>> GetCoursesAsync()
16. {
17. //Gli oggetti sono scritti e letti dalla cache in base a una chiave che li rappresenta
18. string key = "Courses";
20. //Invochiamo il metodo GetOrCreateAsync per recuperare l'oggetto dalla cache.
21. //Se non dovesse esistere, allora verrà eseguita la lambda che si occuperà
22. //di recuperare l'oggetto dal database e di impostare una scadenza
23. return memoryCache.GetOrCreateAsync(key, cacheEntry =>
24. {
25. //Imposto la scadenza di permanenza in cache (60 secondi da adesso)
26. cacheEntry.SetAbsoluteExpiration(TimeSpan.FromSeconds(60));
28. //Recupero l'oggetto usando il servizio applicativo che lo ottiene dal database
29. //Tale oggetto verrà automaticamente messo in cache
30. return courseService.GetCoursesAsync();
31. });
32. }
33. }

Chiamando il metodo GetOrCreateAsync siamo stati in grado, **in maniera atomica** (cioè con una sola operazione), di controllare la presenza in cache di un oggetto identificato dalla chiave "Courses" e, nel caso in cui non fosse presente, posizionarlo in cache dopo averlo recuperato dal database.

Impostare una scadenza per l'oggetto vuol dire che verrà automaticamente rimosso dalla cache al raggiungimento di una certa data/ora. Ci sono due approcci che possiamo seguire.

* **Absolute expiration** (mostrata nell'esempio) ci permette di indicare una precisa data/ora alla quale l'oggetto verrà rimosso dalla cache;
* **Sliding expiration** ci permette invece di indicare una scadenza (es. 60 secondi da adesso) che verrà automaticamente prorogata fintanto che l'oggetto continua ad essere recuperato dalla cache.

Se vogliamo rimuovere un oggetto prima della sua scadenza, possiamo usare il metodo Remove del servizio IMemoryCache.

Limitare il consumo di RAM

Di tanto in tanto, facciamo attenzione alla quantità di RAM occupata dalla nostra applicazione perché se teniamo troppi oggetti in cache potremmo correre il rischio di esaurirla. Eventualmente possiamo porre dei limiti per evitare che troppi oggetti finiscano in cache, come nel prossimo esempio, in cui impostiamo un limite di "1000" dal metodo ConfigureServices della classe Startup.

1. services.Configure<MemoryCacheOptions>(options =>
2. {
3. options.SizeLimit = 1000;
4. });

Ogni qualvolta aggiungiamo un oggetto in cache, ricordiamoci anche di indicare la sua "dimensione". In questo esempio, la dimensione viene impostata a 1 e ciò vuol dire che potremo mettere in cache ancora 999 oggetti prima del raggiungimento del limite.

1. //Mettiamo questo nella lambda passata come parametro a GetOrCreateAsync
2. cacheEntry.SetSize(1);

Cache distribuita

Quando la nostra applicazione è in **web farm**, cioè quando funziona contemporaneamente su più macchine server, è preferibile usare il servizio IDistributedCache che ci permette di centralizzare la cache su servizi esterni come SQL Server o Redis. In questo modo, tutte le macchine otterranno risultati coerenti quando accedono alla cache.

Per iniziare a usare la cache distribuita, rechiamoci nel metodo ConfigureServices della classe Startup e indichiamo quale provider intendiamo usare. In questo esempio usiamo Redis.

1. services.AddStackExchangeRedisCache(options =>
2. {
3. options.Configuration = "localhost"; //Stringa di connessione a Redis
4. options.InstanceName = "MyCourseCache"; //Nome dell'istanza
5. });

Ora siamo pronti a ricevere il servizio IDistributedCache nel costruttore di un nostro componente.

1. public class DistributedCacheCourseService : ICachedCourseService
2. {
3. private readonly IDistributedCache distributedCache;
4. private readonly ICourseService courseService;
6. //Riceviamo sia il servizio di cache distribuita
7. //che il servizio applicativo che ci serve per ottenere gli oggetti dal database
8. public DistributedCacheCourseService(IDistributedCache distributedCache, ICourseService courseService)
9. {
10. //Conserviamo un riferimento ai servizi
11. this.distributedCache = distributedCache;
12. this.courseService = courseService;
13. }
15. public async Task<List<CourseViewModel>> GetCoursesAsync()
16. {
17. //La chiave che identifica l'oggetto in cache
18. string key = "Courses";
20. //Verifichiamo se esiste in cache
21. string serializedObject = await distributedCache.GetStringAsync(key);
22. //Se esisteva, allora lo deserializziamo e lo restituiamo
23. if (serializedObject != null) {
24. return Deserialize<List<CourseViewModel>>(serializedObject);
25. }
27. //Se invece non esisteva, lo recuperiamo dal database
28. List<CourseViewModel> courses = await courseService.GetCoursesAsync();
29. //Lo serializziamo
30. serializedObject = Serialize(courses);
31. //E lo conserviamo in cache per 60 secondi
32. var cacheOptions = new DistributedCacheEntryOptions();
33. cacheOptions.SetAbsoluteExpiration(TimeSpan.FromSeconds(60));
34. await distributedCache.SetStringAsync(key, serializedObject, cacheOptions);
35. //Infine restituiamo l'oggetto
36. return courses;
37. }
39. private string Serialize(object obj)
40. {
41. //Convertiamo un oggetto in una stringa JSON
42. return JsonConvert.SerializeObject(obj);
43. }
45. private T Deserialize<T>(string serializedObject)
46. {
47. //Riconvertiamo una stringa JSON in un oggetto
48. return JsonConvert.DeserializeObject<T>(serializedObject);
49. }
50. }

Come si vede nell'esempio, ogni oggetto deve essere *serializzato*, ovvero convertito in una rappresentazione stringa o binaria, prima di poter essere inserito in cache. Allo stesso modo, deve essere *deserializzato*, cioè riconvertito in oggetto, nel momento in cui viene recuperato dalla cache. Ne consegue quindi che le prestazioni non potranno essere elevate come quelle offerte dal servizio IMemoryCache.

Response caching

ASP.NET Core dispone anche di un meccanismo di *response caching* che permette ai browser o a qualsiasi altro dispositivo si trovi fra il server e l'utente di **porre in cache l'intero contenuto HTML di una risposta**. Questo può avvenire se nella risposta fornita dal server è presente un'intestazione **Cache-Control** che dà le istruzioni sul tempo e sulle modalità di permanenza in cache. Ecco un esempio di tale intestazione.

Cache-Control: public,max-age=60

Per emettere questa intestazione nella risposta, ci basta porre l'attributo ResponseCache in corrispondenza di un controller o di un'action, come si vede nel seguente esempio.

1. //La risposta prodotta da questa action può restare in cache per 60 secondi
2. [ResponseCache(Duration = 60)]
3. public IActionResult Index()
4. {
5. ViewData["Title"] = "Benvenuto su MyCourse!";
6. return View();
7. }

In alternativa possiamo avvalerci di un *profilo*, in modo che le impostazioni di caching possano essere riutilizzate in più punti dell'applicazione e, soprattutto, possano essere tratte da valori di configurazione anziché essere cablate nel codice C#.

1. [ResponseCache(CacheProfileName = "Home")]
2. public IActionResult Index()
3. {
4. ViewData["Title"] = "Benvenuto su MyCourse!";
5. return View();
6. }

Poi, nel metodo ConfigureServices della classe Startup, andiamo a modificare così la chiamata a AddMvc per definire il nostro profilo.

1. services.AddMvc(options =>
2. {
3. Configuration.Bind("ResponseCache:Home", homeProfile);
4. options.CacheProfiles.Add("Home", homeProfile);
5. });

Ed ecco il frammento di configurazione in appsettings.json, da cui otteniamo i valori.

1. "ResponseCache": {
2. "Home": {
3. "Duration": 60,
4. "Location": "Client",
5. "VaryByQueryKeys": ["page"]
6. }
7. }

Ogni profilo di response caching è caratterizzato da questi valori:

* **Duration** esprime la durata in secondi di permanenza in cache;
* **Location** indica quali sono i dispositivi autorizzati a mettere il risultato in cache. Se impostato su "Client", allora solo il browser è autorizzato, altrimenti se "Any" (il default), qualunque altro dispositivo è autorizzato (ad esempio un proxy aziendale). Se invece è impostato su "None" allora nessun dispositivo è autorizzato e di fatto stiamo disabilitando esplicitamente l'intero meccanismo di response caching;
* **VaryByQueryKeys**indica i nomi delle varabili querystring che influenzeranno il contenuto di pagina. Dunque, il browser non potrà avvalersi della cache se cambia anche uno soltanto dei valori querystring indicati. Utile quando abbiamo elenchi paginati;
* **VaryByHeader** indica i nomi di intestazioni della richiesta che influenzano il contenuto di pagina. Utili per esempio quando da uno stesso indirizzo forniamo testi tradotti nella lingua dell'utente in base all'intestazione **Accept-Language** che indica la sua lingua preferita.

ASP.NET Core offre anche un ResponseCachingMiddleware che è in grado di tenere in cache le risposte HTML, proprio come farebbero un browser o un proxy. Il middleware si abilita così dal metodo Configure della classe Startup.

1. app.UseStaticFiles();
2. //Aggiungiamolo DOPO il middleware dei file statici ma PRIMA del middleware di routing di MVC
3. app.UseResponseCaching();
4. app.UseMvc(...)

Internamente, il middleware si avvale del servizio MemoryCache di cui abbiamo già parlato.  
  
E' importante ricordare che **il risultato in cache sarà lo stesso servito a tutti gli utenti** e per questo non deve contenere informazioni specifiche che variano da utente a utente, come per esempio lo username o il contenuto del suo carrello. Su questo aspetto torneremo affrontando il tema dell'autenticazione.