Silverlight & Caching

Welke manieren zijn er en hoe zijn ze te gebruiken?

Kevin Dockx

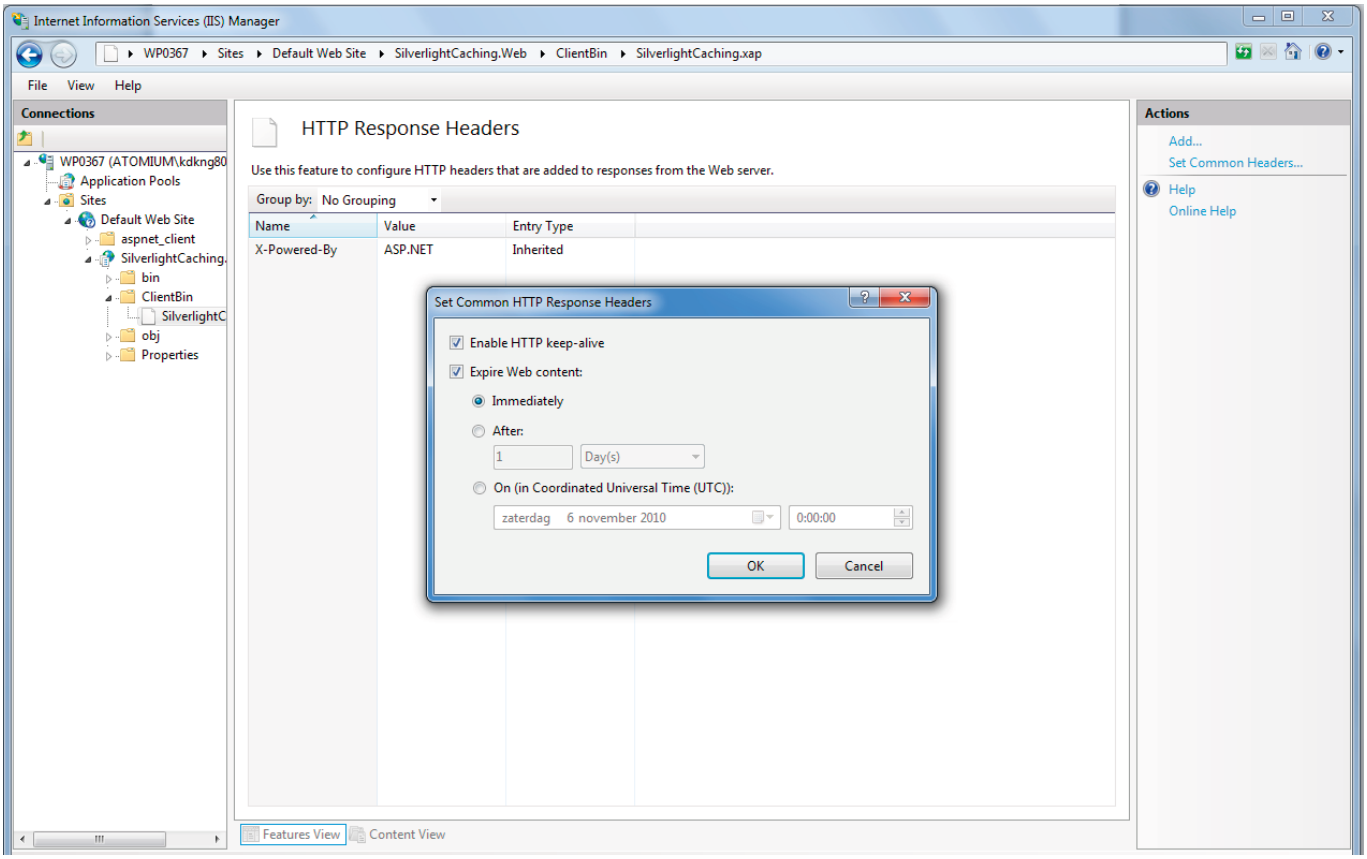
Een van de meest gestelde vragen die ik krijg als het over Silverlight-applicaties gaat, is de vraag hoe je caching mogelijk kan maken in zo'n applicatie. Veel van deze vragen komen voort uit het idee dat Silverlight-applicaties gelijkaardig zijn aan HTML-based (ASP .NET, bvb) applicaties, en op dezelfde manier aan caching moeten kunnen doen. Dat is echter een misvatting.

De manier waarop een Silverlight-applicatie wordt ontwikkeld en hoe deze werkt is fundamenteel verschillend van een gewone webapplicatie. Een Silverlight-applicatie draait immers op de client: je hebt niet constant postbacks nodig om naar een ander deel van je applicatie te gaan. Daarnaast is Silverlight stateful, en heeft het programmeren van een Silverlight-applicatie meer gemeen met het programmeren van een desktop-applicatie dan je zou denken.

Wat betekent dit nu voor caching? We bekijken in dit artikel verschillende manieren van caching: het cachen van de applicatie zelf, het cachen van opgehaalde data, het cachen van opgehaalde data tussen verschillende applicatiesessies door. Daarnaast zullen we ook server-side caching bekijken: het cachen van je resultsets die doorgegeven worden aan de Silverlight-applicatie. Maar laten we beginnen met het begin: het cachen van je applicatie.

Caching: de XAP

In tegenstelling tot een gewone webapplicatie, die bestaat uit verschillende pagina's waarvan je typisch delen zal gaan cachen (de .js-files, image files, css includes, …) bestaat een Silverlight-applicatie uit een XAP-file. Deze file bevat gecompileerde code,



FIGUUR 1.

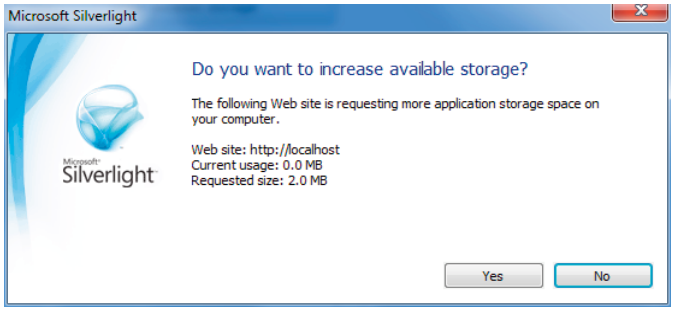
gebruikte assemblies en XAML-files (**grotere Silverlight-applicaties kunnen bestaan uit verschillende XAP-files die dynamisch geladen worden, maar dat valt buiten de scope van dit artikel – qua caching zijn dezelfde principes van toepassing**). Wanneer een gebruiker navigeert naar de Silverlight-applicatie moet deze XAP volledig gedownload en ingeladen worden. Het cachen van die file zal dan ook voor een snellere opstart van je applicatie zorgen, gezien die niet opnieuw moet worden gedownload bij een volgend bezoek aan de pagina. Als developer hoef je hier niks voor te doen: de XAP zal automatisch uit de browser cache komen indien deze daar aanwezig is.

Maar wat als je een nieuwe versie van de XAP uitrolt? Gebruikers die de applicatie reeds bezocht hebben zullen de oude versie zien, gezien deze in hun browser cache zit. Als developer kun je er voor zorgen dat tóch de laatste versie wordt gedownloadt, door Content Expiration aan te zetten in IIS. Dit doe je via IIS Manager: ga naar de website waar je Silverlight-applicatie gehost is, selecteer (in Content view) de XAP, ga naar de Features view, selecteer HTTP Response Headers, en klik bij Actions op Set Common Headers. Nu kan je bij 'Expire Web Content' de waarde Immediately kiezen. Dit zal er voor zorgen dat de nieuwe XAP (enkel indien er een nieuwe is) gedownload zal worden naar de browser van de gebruiker van je applicatie.

Caching: Assembly caching

Een van de zaken die je als developer voor ogen moet houden is dat je moet proberen je XAP-file zo klein mogelijk te houden: dit zal immers de initiële downloadtijd versnellen. Een manier om dit te doen is door Assembly Caching te gebruiken: dit zet je aan bij je project properties van je Silverlight-project door de checkbox 'Reduce XAP size by using application library caching' aan te vinken.

FIGUUR 2.



Deze optie aanvinken zorgt ervoor dat externe assemblies niet meer mee gepackaged worden in de XAP-file, waardoor die file beduidend kleiner wordt. Hoe kan Silverlight die assemblies dan toch gebruiken? De allereerste keer dat deze nodig zijn zullen ze van de Microsoft-site worden gedownload en lokaal gecached worden op de machine van de gebruiker. Met andere woorden: de volgende keer dat zo'n file nodig is, zal deze uit de cache komen (indien de Microsoft-site niet beschikbaar is worden ze, als fallback, in de ClientBin directory gezocht).

Maar heeft deze manier van werken wel voordelen? Immers, je XAP zelf wordt ook gecachet, en de uiteindelijke download-size is even groot (of je assemblies al dan niet mee gepackaged zijn in de XAP doet er niet toe: uiteindelijk heb je ze nodig, en moeten ze worden gedownload). Het voordeel is dat verschillende Silverlight-applicaties diezelfde assemblies automatisch delen. Dat wil zeggen dat indien een bepaalde Silverlight-applicatie bijvoorbeeld de System.Xml.Linq-assembly nodig heeft, en deze zit reeds in de cache op de lokale machine (bijvoorbeeld omdat deze reeds gedownload is omdat de gebruiker naar een andere Silverlight applicatie gesurft is die dezelfde assembly nodig had), deze hergebruikt kan worden en niet opnieuw gedownload moet worden.

Data hergebruiken in dezelfde sessie

Vele applicaties maken gebruik van data die over verschillende schermen heen beschikbaar moet zijn, bijvoorbeeld: data die in codetabellen opgeslagen wordt. Je zou deze data serverside kunnen cachen (zie verderop in dit artikel), maar gezien Silverlight (in tegenstelling tot een standaard webapplicatie) stateful is, kun je zulke data ook gaan bijhouden in properties die beschikbaar zijn over de gehele applicatie en dus vanuit elke pagina van je applicatie toegankelijk is.

Als voorbeeld nemen we het ophalen van enkele landen. Op de MainPage van je applicatie kun je deze ophalen door een WCF Service aan te roepen. Nadat de data correct is opgehaald slaan we deze op in een static property in een LocalStateContainer class:

```
internal void GetCountriesFromServiceOrLocalState()
{
// for demo purposes,
// first clear the collection
// of countries so you can see the "refresh"
Countries.Clear();

// if there are no countries
// (using count for ex purp)
// in the local state container,
// get them from the service;
// else: no need to refetch them,
// they are in mem and
// accessible throughout the application

if (LocalStateContainer.Countries.Count == 0)
{
// call WCF Service
CountryServiceReference.CountryServiceClient client =
new CountryServiceReference.CountryServiceClient();

client.GetCountriesCompleted += (send, args) =>
{
if (args.Error == null)
                    {
                      Countries = args.Result;
                      LocalStateContainer.Countries = args.Result;
}
};
client.GetCountriesAsync();
}
}
```

Als we deze data nodig hebben op een andere pagina is deze makkelijk beschikbaar via deze static property:

```
public static class LocalStateContainer
{
public static ObservableCollection<Country>
Countries { get; set; }
}
```

Dit vermijdt het dubbel ophalen van data.

Data hergebruiken in verschillende sessies

Bovenstaand voorbeeld duidt aan dat de ontwikkeling van een Silverlight-applicatie veel overeenkomsten vertoont met de ont-

wikkeling van andere stateful applicaties (zoals een standaard Windows Forms applicatie): de flow van je applicatie zit anders in elkaar dan bij HTML-based technologieën.

Echter: we slaan alleen data op in memory. Als de gebruiker van je applicatie deze afsluit ben je die data kwijt, en bij een volgende run zal je die opnieuw moeten ophalen. Om dit te vermijden kunnen we de Isolated Storage (specifiek per gebruiker) gebruiken om data te persisteren en te hergebruiken over sessies heen.

Bovenstaand voorbeeld kunnen we nu als volgt aanpassen: alvorens we de lijst met landen gaan ophalen van de server, kijken we eerst of deze reeds aanwezig is in Isolated Storage. Is dat het geval, dan kunnen we die data gebruiken. Is dat niet het geval, dan laden we deze data van de server en slaan we deze op in Isolated Storage, zodat deze bij de volgende run van de applicatie beschikbaar zal zijn:

```
internal void GetCountriesFromServiceOrIsolatedStorage()
{
// for demo purposes,
// first clear the collection
// of countries so you can see the "refresh"
Countries.Clear();

// is there any country data in the Isolated Storage?

if (userIsolatedStorageSettings.Contains("CachingCountriesDemo"))
{
Countries = (ObservableCollection<Country>)
            userIsolatedStorageSettings["CachingCountriesDemo"];
}
else
{
// call WCF Service
CountryServiceReference.CountryServiceClient client =
new CountryServiceReference.CountryServiceClient();

client.GetCountriesCompleted += (send, args) =>
{
if (args.Error == null)
{
Countries = args.Result;

// commit data to Isolated Storage
userIsolatedStorageSettings["CachingCountriesDemo"] =
new ObservableCollection<Country>(args.Result);
}
};
client.GetCountriesAsync();
}
}
```

Als je de code uitvoert zul je zien dat bij een volgende run van de applicatie de data niet opgehaald wordt van de server, maar uit de Isolated Storage komt. Om de Isolated Storage leeg te maken volstaat volgend stukje code:

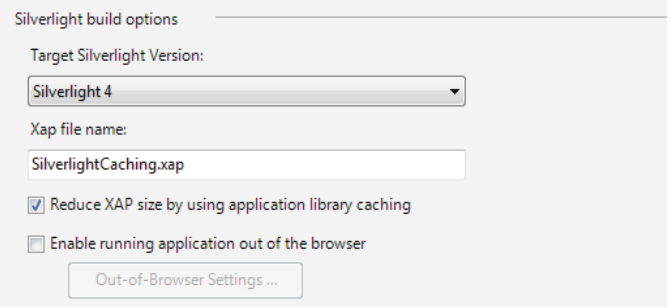
```
internal void ClearIsolatedStorage()
{
userIsolatedStorageSettings.Clear();
}
```

Hoewel de Isolated Storage een makkelijke manier is om data te bewaren over verschillende sessies zijn er enkele zaken waar je rekening mee moet houden, en waar je niet 100% zeker van kan zijn. Ten eerste moet de data die je wilt opslaan in Isolated Storage logischerwijze serializeerbaar zijn. Daarnaast moet de client het recht hebben om van zijn Isolated Storage gebruik te kunnen maken. Standaard staat dit aan, maar een administrator zou dit kunnen uitschakelen – is dat het geval, dan zal je code een Exception werpen.

Ook de grootte van beschikbare Isolated Storage kan beperkt zijn op de client computer (standaard is dit 1MB). Je kan echter wel vragen om meer ruimte, maar dit vereist interactie van de gebruiker. Volgende code geeft aan hoe je dit kan vragen aan je gebruiker:

```
internal void AskForMoreIsolateStorageSpace()
{
using (var isf = IsolatedStorageFile.GetUserStoreForApplication())
{
// ask for 2MB instead of default 1MB
isf.IncreaseQuotaTo(2097152);
}
}
```

De gebruiker zal volgende popup te zien krijgen:



FIGUUR 3.

Tot slot: wees voorzichtig met het opslaan van vertrouwelijke data in Isolated Storage: hoewel dit een verborgen map is op de client computer kan een gebruiker nog steeds aan deze data komen. Gebruik dus steeds de Crypthography classes om vertrouwelijke data te encrypteren.

Server side caching: standaard

Naast het cachen van data aan de clientkant kun je dit ook op de server gaan doen – dit is in feite niet verschillend van hoe je dit bij een standaard webapplicatie zou doen. Vooral in een multi-user omgeving kan dit zeer belangrijk zijn: indien je meerdere gebruikers hebt die tegelijkertijd van dezelfde services gebruik maken kun je er zo voor zorgen dat alleen de eerste request (afhankelijk van hoe lang je iets in cache opgeslagen laat) de data hoeft op te halen/verwerken.

Als voorbeeld kunnen we nu de code van onze WCF Service als volgt wijzigen:

```
[OperationContract]
public List<Country> GetCountriesWithCaching()
{
string cacheKey = "CachedCountries";

// create a new generic list
// to hold the countries.
// In this case, we use cultureinfo for this
List<Country> countryList = new List<Country>();

// is the list in cache?
if (HttpContext.Current.Cache.Get(cacheKey) == null)
{
CultureInfo[] cultures =
    CultureInfo.GetCultures(CultureTypes.SpecificCultures);

                // loop through all the cultures found
foreach (CultureInfo culture in cultures)
{
// LCID = Locale ID
```

Caching kan er voor zorgen dat je applicatie niet alleen sneller werkt, maar ook beduidend minder bandbreedte verbruikt.

```
RegionInfo region = new RegionInfo(culture.LCID);

// avoid doubles
if (countryList.Where(c => c.Description ==
                      region.EnglishName).Count() == 0)

// add a new country
countryList.Add(new Country() { Description =
region.EnglishName });
}

// add list to cache for 3 minutes
HttpContext.Current.Cache.Add(cacheKey,
                              countryList,
                              null,
                              DateTime.Now.AddMinutes(3),
                              Cache.NoSlidingExpiration,
                              CacheItemPriority.Default,
                              null);
}
else
{
countryList = (List<Country>)
              HttpContext.Current.Cache[cacheKey];
}

return countryList;
}
```

De allereerste keer dat je dit uitvoert zal je data opgehaald worden. Elke volgende keer dat jij, of een andere gebruiker, in de volgende 3 minuten de data opnieuw ophaalt zal dit uit cache komen.

Server side caching: via WCF RIA Services

Een groot aantal Silverlight-applicaties worden gebouwd met behulp van WCF RIA Services. Gezien je met deze techniek aan de clientkant een DomainContext hebt met daarin al je reeds opgehaalde data zorgt dit er out of the box al voor dat je je data veel minder vaak moet ophalen: wat vaak voorkomt is de DomainContext beschikbaar maken via een Local State Container, zodat deze voor de hele applicatie beschikbaar is. De DomainContext zelf fungeert dan als container voor de reeds opgehaalde data.

Wat minder bekend is, is dat WCF RIA Services je ook toelaat om je resultsets langs de server kant te cachen: dit zal er ook weer voor zorgen dat je data enkel de eerste keer moet opgehaald worden – de volgende keren komt dit uit de cache op de server. Met WCF RIA Services bereik je dit door middel van het OutputCache attribuut. In volgend voorbeeld kan je zien hoe een lijst met landen server side gecacht wordt:

```
[OutputCache(OutputCacheLocation.Server, 180,
UseSlidingExpiration=true)]
public IQueryable<CountryForRIA> GetCountries()
{
// create a new generic list
// to hold the countries.
// In this case, we use cultureinfo for this
List<CountryForRIA> countryList = new List<CountryForRIA>();

CultureInfo[] cultures = CultureInfo.GetCultures
(CultureTypes.SpecificCultures);
```

```
// loop through all the cultures found
foreach (CultureInfo culture in cultures)
{
// LCID = Locale ID
RegionInfo region = new RegionInfo(culture.LCID);

// avoid doubles
if (countryList.Where(
    c => c.Description == region.EnglishName).Count() == 0)
   // add a new country
   countryList.Add(new CountryForRIA() {
      ID = region.ThreeLetterISORegionName,
      Description = region.EnglishName });
}

return countryList.AsQueryable<CountryForRIA>();
}
```

Conclusie

Er zijn verschillende manieren om er voor te zorgen dat je data niet vaker gaat ophalen dan nodig is voor je applicatie, zowel langs de client kant (in dezelfde sessie of over sessies heen) als langs de server kant (zowel met standaard WCF (of andere) services, als met WCF RIA Services). Caching kan er voor zorgen dat je applicatie niet alleen sneller werkt, maar ook beduidend minder bandbreedte verbruikt. Daarom is het een techniek die zeker bekeken moet worden voor bijna elk Silverlight project.

Kevin Dockx, is technisch specialist/projectleider voor .NET webapplications en solution manager voor rich applications bij Real Dolmen.