

Loading ASP.NET Core MVC Views From A Database Or Other Location

21 July 2016 08:56

ASP.NET MVC ASP.NET CORE

For the vast majority of ASP.NET Core MVC applications, the conventional method of locating and loading views from the file system does the job nicely. But you can also load views from other sources including a database. This is useful in scenarios where you want to give users the option to craft or modify Razor files but don't want to give them access to the file system. This article looks at the work required to create a component to obtain views from other sources, using the database as a working example.

The Razor View Engine uses components called `FileProviders` to obtain the content of views. The view engine will iterate its collection of locations that it searches for views (`ViewLocationFormats`) and then present those locations to each of the registered `FileProviders` in turn until one returns the view content. At startup, a `PhysicalFileProvider` is registered with the view engine, which is designed to look for physical `.cshtml` files in the various locations, starting with the customary Views folder found in every MVC project template. An `EmbeddedFileProvider` is available for obtaining view content from embedded resources. If you want to store views in another location, such as a database, you can create your own `FileProvider` and register it with the view engine.

`FileProviders` must implement the `IFileProvider` interface. The `IFileProvider` interface specifies the following members:

```
IDirectoryContents GetDirectoryContents(string subpath);
FileInfo GetFileInfo(string subpath);
IChangeToken Watch(string filter);
```

The most important of these is the `GetFileInfo` method which returns an object that implements the `FileInfo` interface representing a file implementation. The `Watch` method returns an implementation of the `IChangeToken` interface. When the view engine first finds a view, it has to compile it. It caches the compiled view so that it doesn't have to be compiled again for subsequent requests. The view engine needs some way in which it can be notified that changes have taken place to the original view so that the cache can be refreshed with the latest version. The `IChangeToken` instance provides that notification. So, in order to get views from a database, we need an implementation of `IFileProvider`, an implementation of `FileInfo`, and an implementation of `IChangeToken`.

Database Schema

The minimum schema for the database table required for storing views is illustrated below together with the DDL for creating the table

	Column Name	Data Type	Allow Nulls
	Location	nvarchar(150)	<input type="checkbox"/>
	[Content]	nvarchar(MAX)	<input type="checkbox"/>
	LastModified	datetime	<input type="checkbox"/>
	LastRequested	datetime	<input checked="" type="checkbox"/>

```
CREATE TABLE [dbo].[Views](
    [Location] [nvarchar](150) NOT NULL,
    [Content] [nvarchar](max) NOT NULL,
    [LastModified] [datetime] NOT NULL,
    [LastRequested] [datetime]
)
```

The *Location* field contains a unique identifier for the view. The view engine looks for views using subpaths, so it makes sense to use them to identify the individual view. So the Location value for the home page will be one of the paths that the view engine expects to find the view for the `Index` method of the `Home` controller e.g. `/views/home/index.cshtml`. The *Content* field contains the Razor and HTML from the view file. The *LastModified* field defaults to `GetUtcDate` when the view is created, and is updated whenever the view content is modified. The *LastRequested* field is updated with the current UTC date and time whenever the view engine successfully retrieves the content. These two fields are used to calculate whether any modifications have taken place since the file was last retrieved, compiled and cached. You would set the default value for LastModified to `GetDate()`, and then reset the value whenever you edit the file as part of the CRUD procedure.

IFileProvider

```
1  using Microsoft.Extensions.FileProviders;
2  using Microsoft.Extensions.Primitives;
3  using System;
4  using System.IO;
5
6  namespace RazorEngineViewOptionsFileProviders
7  {
8      public class DatabaseFileProvider : IFileProvider
9      {
10         private string _connection;
11         public DatabaseFileProvider(string connection)
12         {
13             _connection = connection;
14         }
15         public IDirectoryContents GetDirectoryContents(string subpath)
16         {
17             throw new NotImplementedException();
18         }
19
20         public IFileInfo GetFileInfo(string subpath)
21         {
22             var result = new DatabaseFileInfo(_connection, subpath);
23             return result.Exists ? result as IFileInfo : new NotFoundFileInfo(subpath);
24         }
25
26         public IChangeToken Watch(string filter)
27         {
28             return new DatabaseChangeToken(_connection, filter);
29         }
30     }
31 }
```

DatabaseFileProvider.cs hosted with ❤ by GitHub

[view raw](#)

I have named my implementation `DatabaseFileProvider`. It has a constructor taking a string that represents the connection string for a database. I haven't provided an implementation for the `GetDirectoryContents` method as one is not needed for this use-case. The `GetFileInfo` method returns my custom `IFileInfo` if a result matching the specified path is found, or a `NotFoundFileInfo` object, which tells the view engine to try another provider, or another view location. The `Watch` method returns my custom `IChangeToken` object.

IFileInfo

The `IFileInfo` interface features the following members:

```

public interface IFileInfo
{
    //
    // Summary:
    //     True if resource exists in the underlying storage system.
    bool Exists { get; }
    //
    // Summary:
    //     True for the case TryGetDirectoryContents has enumerated a sub-directory
    bool IsDirectory { get; }
    //
    // Summary:
    //     When the file was last modified
    DateTimeOffset LastModified { get; }
    //
    // Summary:
    //     The length of the file in bytes, or -1 for a directory or non-existing files.
    long Length { get; }
    //
    // Summary:
    //     The name of the file or directory, not including any path.
    string Name { get; }
    //
    // Summary:
    //     The path to the file, including the file name. Return null if the file is not
    //     directly accessible.
    string PhysicalPath { get; }

    //
    // Summary:
    //     Return file contents as readonly stream. Caller should dispose stream when complete.
    //
    // Returns:
    //     The file stream
    Stream CreateReadStream();
}

```

I have left the comments from the source code in as they explain the purpose of each member quite nicely. The important ones are the `Name`, `Exists`, `Length` properties and the `CreateReadStream` method. Here is the `DatabaseFileInfo` class, which is the custom implementation of `IFileInfo` for getting view content from the database:

```

1  using Microsoft.Extensions.FileProviders;
2  using System;
3  using System.Data.SqlClient;
4  using System.IO;
5  using System.Text;
6
7  namespace RazorEngineViewOptionsFileProviders
8  {
9      public class DatabaseFileInfo : IFileInfo
10     {
11         private string _viewPath;
12         private byte[] _viewContent;
13         private DateTimeOffset _lastModified;
14         private bool _exists;
15
16         public DatabaseFileInfo(string connection, string viewPath)
17         {
18             _viewPath = viewPath;
19             GetView(connection, viewPath);
20         }
21         public bool Exists => _exists;
22
23         public bool IsDirectory => false;
24
25         public DateTimeOffset LastModified => _lastModified;
26
27         public long Length
28         {
29             get
30             {
31                 using (var stream = new MemoryStream(_viewContent))

```

```

32         {
33             return stream.Length;
34         }
35     }
36 }
37
38 public string Name => Path.GetFileName(_viewPath);
39
40 public string PhysicalPath => null;
41
42 public Stream CreateReadStream()
43 {
44     return new MemoryStream(_viewContent);
45 }
46
47 private void GetView(string connection, string viewPath)
48 {
49     var query = @"SELECT Content, LastModified FROM Views WHERE Location = @Path;
50                 UPDATE Views SET LastRequested = GetUtcDate() WHERE Location = @Path";
51     try
52     {
53         using (var conn = new SqlConnection(connection))
54         using (var cmd = new SqlCommand(query, conn))
55         {
56             cmd.Parameters.AddWithValue("@Path", viewPath);
57             conn.Open();
58             using (var reader = cmd.ExecuteReader())
59             {
60                 _exists = reader.HasRows;
61                 if (_exists)
62                 {
63                     reader.Read();
64                     _viewContent = Encoding.UTF8.GetBytes(reader["Content"].ToString());
65                     _lastModified = Convert.ToDateTime(reader["LastModified"]);
66                 }
67             }
68         }
69     }
70     catch (Exception ex)
71     {
72         // if something went wrong, Exists will be false
73     }
74 }
75 }
76 }

```

DatabaseFileInfo.cs hosted with ❤ by GitHub

[view raw](#)

The real work is done in the `GetView` method, which is called in the constructor. It checks the database for the existence of an entry matching the file path provided by the view engine. If a match is found, `Exists` is set to `true` and the content is made available as a `Stream` via the `CreateReadStream` method. I've chosen to use plain ADO.NET for this example, but other data access technologies are available.

IChangeToken

The final component in the chain is the implementation of `IChangeToken`. This is responsible for notifying the view engine that a view has been modified, and that the cached version should be replaced with the updated version.

```

1  using Microsoft.Extensions.Primitives;
2  using System;
3  using System.Data.SqlClient;
4
5  namespace RazorEngineViewOptionsFileProviders
6  {
7      public class DatabaseChangeToken : IChangeToken
8      {
9          private string _connection;
10         private string _viewPath;

```

```

11
12     public DatabaseChangeToken(string connection, string viewPath)
13     {
14         _connection = connection;
15         _viewPath = viewPath;
16     }
17
18     public bool ActiveChangeCallbacks => false;
19
20     public bool HasChanged
21     {
22         get
23         {
24
25             var query = "SELECT LastRequested, LastModified FROM Views WHERE Location = @Path;";
26             try
27             {
28                 using (var conn = new SqlConnection(_connection))
29                 using (var cmd = new SqlCommand(query, conn))
30                 {
31                     cmd.Parameters.AddWithValue("@Path", _viewPath);
32                     conn.Open();
33                     using (var reader = cmd.ExecuteReader())
34                     {
35                         if (reader.HasRows)
36                         {
37                             reader.Read();
38                             if (reader["LastRequested"] == DBNull.Value)
39                             {
40                                 return false;
41                             }
42                             else
43                             {
44                                 return Convert.ToDateTime(reader["LastModified"]) > Convert.ToDateTime(reader["L
45                             }
46                         }
47                         else
48                         {
49                             return false;
50                         }
51                     }
52                 }
53             }
54             catch (Exception)
55             {
56                 return false;
57             }
58         }
59     }
60 }
61
62     public IDisposable RegisterChangeCallback(Action<object> callback, object state) => EmptyDisposable.Instance
63 }
64
65 internal class EmptyDisposable : IDisposable
66 {
67     public static EmptyDisposable Instance { get; } = new EmptyDisposable();
68     private EmptyDisposable() { }
69     public void Dispose() { }
70 }
71 }

```

The key member of the interface is the **HasChanged** property. The value of this is determined by comparing the last requested time and the last modified time of a matching file entry. If the file has been modified since it was last requested, the property is set to **true** which results in the view engine retrieving the modified version.

The only thing left to do now is to register the `DatabaseFileProvider` with the view engine so that it knows to use it. This is done in the `ConfigureServices` method in `Startup.cs`:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     // Add framework services.
4     services.AddMvc();
5     services.Configure<RazorViewEngineOptions>(opts =>
6         opts.FileProviders.Add(
7             new DatabaseFileProvider(Configuration.GetConnectionString("DefaultConnection"))
8         )
9     );
10 }
```


RazorEngineViewOptionsFileProviders.Startup.cs hosted with ❤ by GitHub [view raw](#)

There are some points to note. The `PhysicalFileProvider` will be invoked first since it has been registered first. If you have a `.cshtml` file in one of the locations that get checked, it will be returned and the `DatabaseFileProvider` (or any subsequent providers) will not be invoked for that request. In its current form, the `OnChangeToken` will be invoked for every location that the view engine checks. For that reason, it would be sensible perhaps to cache the paths where database entries exist, and to prevent the database request being executed if the requested path is not in the cache.

Summary

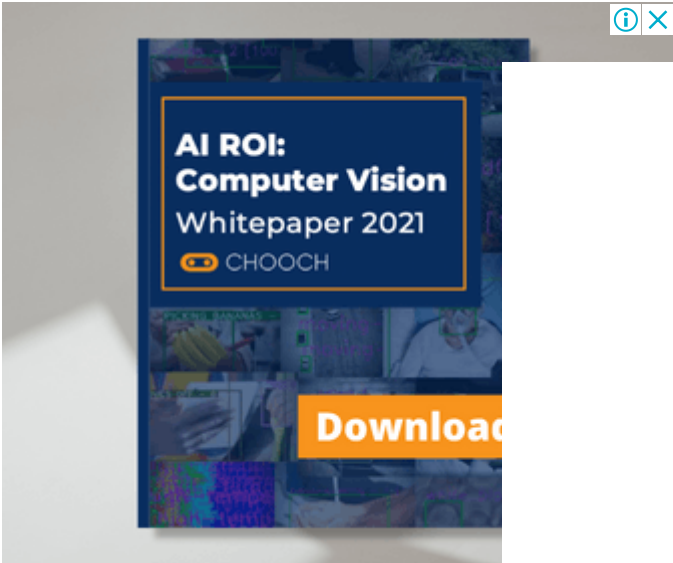
The Razor view engine has been designed to be fully extensible, enabling you to plug in your own `FileProvider` so that you can locate and load view from any source you can write a provider for. This article shows how you can do that using a database as a source. The sample site is available from [GitHub](#).

Free White Paper Download



Other Sites

[Learn Razor Pages](#)



Categories

- [.NET 6 \(4\)](#)
- [ADO.NET \(24\)](#)
- [AJAX \(17\)](#)
- [ASP.NET 2.0 \(39\)](#)
- [ASP.NET 3.5 \(43\)](#)
- [ASP.NET 5 \(16\)](#)