When using campaigns on MVC sites, the system hashes the URLs of *Page visit* conversions/journey steps and activities to be able to put data for reports together as quickly as possible.

By default, the hash function removes the protocol and all query string parameters from the URL. The campaign then works regardless of the used protocol (for example, for both *http* and *https*). However, the default hash function does not work for sites where displaying of pages is performed via query strings (for example, if the URL of a page on your site looks like *http://www.mysite.com/index.html?viewpage=123*). More specifically, the noted address will be trimmed to *www.mysite.com/index.html*, and all pages trimmed to this "URL root" will be logged as conversions for a single page. Therefore, SEO-friendly URLs are recommended.

If you use campaigns and your site displays content via query string parameters or uses any other scenarios that are not supported by default, you can customize the hash function to fulfill your needs.

How it works – Using the hash to display reports

For every activity that occurs on a site, the system creates a record in the database in the **OM_Activity** table. However, when a page visit activity occurs on an MVC site, the process is more complex. The system:

1. Trims the page URL – the result is a "URL root" without the protocol, query string parameters, and any other fragment identifiers
2. Hashes the URL root
3. Saves the hashed URL root into the **ActivityURLHash** column (in the *OM_Activity* table)

Then, when a report is recalculated, the system:

1. Trims the URLs of page visit conversions and campaign journey steps the same way as for page visit activities
2. Hashes the URL roots
3. Compares the hashed URL root of the given conversion or journey step against the *ActivityURLHash* column with hashed URL roots of the tracked activities

If the hashes match, the activity is included in the reports.

Implementing a custom hash function

To implement a custom hash function that overrides the default hash function:

1. Create a new **Class Library** project in Visual Studio (separately from the solutions of your MVC and Kentico applications).
2. Add the Kentico API libraries to the project:
    a. Right-click the project in the **Solution Explorer** and select **Manage NuGet Packages**.
    b. Search for the **Kentico.Libraries** package.
    c. Install the *Kentico.Libraries* version that matches the exact version of your Kentico project and your MVC project's *Kentico.Libraries* package.
3. Edit the project's **AssemblyInfo.cs** file (in the *Properties* folder) and add the **AssemblyDiscoverable** assembly attribute:

```
using CMS;

[assembly:AssemblyDiscoverable]
```

4. Add a new class to the project, for example named **CustomActivityUrlHashPreprocessor.cs**.
5. Make the class implement the **IActivityUrlPreprocessor** interface.
6. Add the **RegisterImplementation** assembly attribute above the class declaration.

```
using CMS;
using CMS.WebAnalytics;

[assembly: RegisterImplementation(typeof(IActivityUrlPreprocessor), typeof
(CustomActivityUrlHashPreprocessor))]

public class CustomActivityUrlHashPreprocessor : IActivityUrlPreprocessor
{
}
```

7. Implement the class's **PreprocessActivityUrl** method.

```
public string PreprocessActivityUrl(string activityUrl)
{
        ...
}
```

> The *PreprocessActivityUrl* method:
>
> - Takes the original URL of an activity, conversion, or campaign journey step
> - Returns a processed URL in a form that can be hashed. The hash is then used for later calculations or saving into the database

8. Save and build the project.

You then need to apply the customization to BOTH your MVC and Kentico application:

1. Copy the project's assembly file (DLL) into a subfolder in the directories of your MVC and Kentico applications (or use any other method for deploying DLLs).
2. Open your Kentico and MVC projects in Visual Studio.
3. Add a reference to the custom DLL from your MVC project and your Kentico web project.

> **Improving performance**
>
> To improve performance of report calculation, create a database index for the **ActivityURLHash** column in the **OM_Activity** database table. The index should be non-clustered and non-unique. See the Create Nonclustered Indexes article for more information.

Example – Hash function for multilingual sites

The following example demonstrates how to implement a custom hash function that removes the culture code from URLs. The example assumes that the URL has the following patterns:

```
<protocol>://<domain>/<culture code>/<other identifiers>
```

For example:

- http://www.mysite.com/en-US/Articles/11/Coffee-Beverages-Explained
- https://myothersite.net/es-ES/Contacts

Logging of page visits will then work for all language variants.

To implement the example:

1. Create a new **Class Library** project in Visual Studio (according to the procedure described in Implementing a custom hash function).
2. Add the **CustomActivityUrlHashPreprocessor.cs** class:

```
using System;
using System.Linq;

using CMS;
using CMS.Helpers;
using CMS.SiteProvider;
using CMS.WebAnalytics;

[assembly: RegisterImplementation(typeof(IActivityUrlPreprocessor), typeof
(CustomActivityUrlHashPreprocessor))]

public class CustomActivityUrlHashPreprocessor : IActivityUrlPreprocessor
{
    /// <summary>
    /// Prepares the specified URL for hashing. Trims the protocol, query
strings, and the culture code.
    /// </summary>
    /// <param name="activityUrl">Original URL that is trimmed.</param>
    /// <returns>URL root that is ready to be hashed.</returns>
    public string PreprocessActivityUrl(string activityUrl)
    {
        // Removes query strings
        string url = URLHelper.RemoveQuery(activityUrl);

        // Removes fragment identifiers
        url = RemoveAnchor(url);

        // Removes the protocol
        url = URLHelper.RemoveProtocol(url);

        // Removes the culture code
        url = RemoveCultureCode(url);

        // Removes the trailing slash (the slash after the URL)
        url = URLHelper.RemoveTrailingSlashFromURL(url);

        return url.ToLower();
    }

    /// <summary>
    /// Removes the culture code from the specified URL.
    /// </summary>
    /// <param name="url">URL in the expected format (domain.com/en-US/path/to
/page).</param>
    /// <returns>URL without the culture code.</returns>
    private string RemoveCultureCode(string url)
    {
        // Removes the protocol and the domain from the URL
        string path = URLHelper.RemoveProtocolAndDomain(url);

        // Separates the culture code (the first segment between slashes after
the domain)
        string cultureCode = path.Split(new[] { '/' }, StringSplitOptions.
RemoveEmptyEntries).FirstOrDefault();

        // Remove the culture code from the URL
```

```
        if (!string.IsNullOrEmpty(cultureCode) && CultureSiteInfoProvider.
IsCultureOnSite(cultureCode, SiteContext.CurrentSiteName))
        {
            return url.Replace('/' + cultureCode, string.Empty);
        }

        return url;
    }

    /// <summary>
    /// Removes fragment identifiers from the specified URL.
    /// </summary>
    /// <param name="url">URL from which fragment identifiers are removed.</param>
    /// <returns>URL without fragment identifiers.</returns>
    private string RemoveAnchor(string url)
    {
        // Gets the position of the hashtag symbol
        int index = url.IndexOf("#", StringComparison.Ordinal);

        // Removes the found hashtag and all content after
        if (index > 0)
        {
            return url.Substring(0, index);
        }

        return url;
    }
}
```

3. Build the project and transfer the DLL to BOTH your MVC and Kentico application.

Now, when you create a campaign with page visit conversions or campaign journey steps, you will see reports regardless of the displayed language variant.