Sitecore CMS 7.2

Sitecore MVC Developer's Reference Guide

A developers guide to MVC in Sitecore

Table of Contents

Chapter 1

Introduction

This chapter gives a brief overview of Sitecore Model-View-Controller (MVC) features and describes how to prepare your development environment and install Sitecore MVC.

This chapter describes the prerequisites for installing Sitecore MVC as well as how to install it.

This manual contains the following chapters:

- **Chapter 1 — Introduction**
  This chapter describes how to install Sitecore MVC.
- **Chapter 2 — Sitecore MVC Pipelines and Request Handling**
  This chapter describes how Sitecore MVC integrates with ASP.NET and the Sitecore pipeline processors. It also discusses the important classes that are created during pipeline processing and that you can use to retrieve information about the current request.
- **Chapter 3 — View Renderings**
  This chapter describes the MVC View Renderings.
- **Chapter 4 — MVC Routing**
  This chapter describes MVC Routing and shows you how to map URLs by registering routes in the `Global.asax` module.
- **Chapter 5 — Additional Considerations for Sitecore MVC**
  This chapter contains some additional details about what has been added to Sitecore for MVC support.

1.1 Overview

Sitecore MVC allows you to use ASP.NET MVC as a first class rendering engine in Sitecore.

Both ASP.NET WebForms and MVC are supported and can be mixed, although a single request must be rendered by either WebForms or MVC.

The secondary goal of Sitecore MVC is to provide a powerful, flexible, and extendable API for controlling the rendering process.

Please note that unless explicitly specified the classic Sitecore rendering engine is unaffected.

In Sitecore CMS 7.2, Sitecore ASP.NET MVC 5 integration is enabled by default.

There are three important aspects to implementing MVC in Sitecore:

- How Sitecore MVC integrates into the `HttpRequestBegin` pipeline.
- How to create MVC View Renderings.
- How to use MVC Routing to manage URLs.

These are discussed later of this document.

1.1.1 What is ASP.NET MVC

ASP.NET MVC is an implementation of a long-standing software architecture design pattern known as the Model-View-Controller pattern. Implementations of this design pattern have been used in software development and web development frameworks for many years, on many different platforms.

In the Model-View-Controller design pattern, the three main components are structured as follows:

- The Model is a class or set of classes that represent the data that your application uses. These classes also contain application logic for managing this data.
- The View consists of presentation components that determine how data will be presented in the application's user interface.
- The Controller is a class or set of classes that receive requests from the user and execute application logic to request data from the Model and select which Views should be used to display this data.

For developers who are new to ASP.NET MVC, it is recommended that you use the resources available from Microsoft to learn more about the basics of working with this web application framework. This document focuses on information that is specific to integrating ASP.NET MVC with a Sitecore website.

1.1.2 Prerequisites

Sitecore CMS 7.2 requires ASP.NET MVC 4.

Microsoft MVC 5 requires.NET Framework 4.5.

Your Sitecore installation must run with a .NET Framework 4.0 application pool.

Microsoft Visual Studio 2012 or 2013.

Download and install the Microsoft MVC Web Platform Installer from http://www.asp.net/mvc. This installs the MVC project templates for Visual Studio.

1.1.3 Installation

In Sitecore CMS 7.2, Sitecore ASP.NET MVC 5 integration is enabled by default.

You *must* install Microsoft ASP.NET MVC 4 before you run the Sitecore CMS 7.2 installation program. Sitecore 7.2 installs the ASP.NET MVC 5 libraries on top of MVC 4.

If you do not have Microsoft ASP.NET MVC 4 installed, the Sitecore installation program will display an error message. You must install Microsoft ASP.NET MVC 4 and then resume the installation program.

Chapter 2

Sitecore MVC Pipelines and Request Handling

This chapter discusses how Sitecore MVC integrates with the ASP.NET and Sitecore pipeline processors. It also discusses important classes that are created during pipeline processing and are available to retrieve information about the currently executing request.

- MVC and the HttpBeginRequest Pipeline
- The PageDefinition Class
- Sitecore MVC Context Objects
- Changes to Existing Pipelines
- MVC Specific Pipelines
- Sitecore MVC Request Handling

2.1 MVC and the HttpBeginRequest Pipeline

Sitecore MVC hooks into the `HttpBeginRequest` pipeline and redirects to the MVC rendering engine either when an MVC route is matched or the layout file is an MVC view — as determined by the `MvcSettings. ViewExtensions` property. The logic is performed by the pipeline processors `TransferRoutedRequest` (route match) and `TransferMvcLayout` (MVC view).

For more information about MVC routing, see the section
MVC Routing.

During MVC processing, Sitecore MVC builds up an in-memory model that represents the high-level rendering information in Sitecore. This is called the `PageDefinition`. The `PageDefinition` represents all the different ways to respond to the current request; much in the same way that a standard Sitecore presentation layout definition for an item can contain multiple rendering collections for different devices.

The `PageDefinition` is not only tied to devices. Other properties might influence which response should be sent to the caller. Also the response might not even be HTML. It could just as well be JSON or XML.

The class that encapsulates the information needed to generate the response is called `PageContext`. To generate the actual response, an MVC view (`IView`) is used. This view is a property of the `PageContext` and is called `PageView`.

An incoming request is handled by Sitecore MVC in the following manner:

- Create a `PageContext`.
- Determine the requested item.
- Determine the relevant controller to invoke.
- Build the `PageDefinition` and assign it to the `PageContext`.
- Select the root rendering to use for generating the response (typically the layout assigned to the requested item).
- Wrap an `IView` around the root rendering and store it in `PageContext.PageView`.
- Pass the `PageView` to the ASP.NET MVC runtime for rendering the response to the client.

Because all of this work is handled by pipeline processors this architecture is pluggable and extensible.

As part of generating its output, the root rendering will typically call the `Html.Sitecore().Placeholder` extension method. This method renders the renderings from the `PageDefinition` that corresponds to the specified placeholder name.

There are rendering support methods belonging to the `Html.Sitecore()` object. These include the `Field` method which invokes the standard Sitecore `RenderField` pipeline and methods for rendering MVC views, XSLT renderings, and other types of presentation components, directly.

2.1.1 The PageDefinition Class

The `PageDefinition` holds high-level information about all the devices, layouts and renderings for the requested item. The `PageDefinition` is built as one of the very first actions during an MVC request. The `PageDefinition` is built by the `BuildPageDefinition` pipeline.

The `PageDefinition` contains a list of `Rendering` objects. One of these is used as the root rendering for generating the actual response (HTML, JSON, or XML). The selection takes place in the `GetPageRendering` pipeline and the output of the selected rendering is generated in the `RenderRendering` pipeline.

Most renderings are rendered by using the `Renderer` property of the `Rendering` object. A `Renderer` is an object that knows how to render a specific type of `Rendering`.

A `Rendering` can be thought of as the definition of what should be rendered — a MVC view, an XSLT, static text, or other presentation information — and the `Renderer` as an object that knows how to transform that definition to the format requested by the client, for example, HTML, JSON or XML.

The `RenderRendering` pipeline handles caching, context, and more, but for the actual rendering of the output, it typically just uses the rendering's `Renderer` property. Examples of renderers include `ViewRenderer, XsltRenderer, ItemRenderer`. All renderers derive from the `Renderer` abstract base class.

The `Renderer` class contains a single method — `Render()` — that must be implemented by custom renderers.

2.1.2 Sitecore MVC Context Objects

There are several context objects that are created during the processing of a Sitecore MVC request. These context objects can be used to access information about the state of the request that is currently being currently executed.

All the context classes have static properties to get the current instance:

- `Current` —tries to get the current context and throws an exception if the calling code is not inside a context of the requested type.
- `CurrentOrNull` — tries to get the current context and returns `null` if the calling code is not inside a context of the requested type.

The `RequestContext` class does not have these properties. However, the extension methods `Current()` and `CurrentOrNull()` can be used instead

RequestContext

The `RequestContext` is an MVC class that holds information about the current request, including URL, form values, user, and route. It can be accessed using the `PageContext.RequestContext` property.

PageContext

The `PageContext` object holds information about the page being built. It includes the requested item, the current device, the `PageDefinition`, and the view that is used to render the response (`PageView`).

PlaceholderContext

The `PlaceholderContext` keeps track of the current placeholder information. This is used to support nested placeholders.

RenderingContext

The `RenderingContext` keeps track of the current rendering and the associated data source/item.

2.2 Changes to Existing Pipelines

2.2.1 Initialize Pipeline

Sitecore MVC installs three processors in the `initialize` pipeline.

1. `InitializeGlobalFilters` — This processor sets up hooks in the MVC global filters (`ActionExecuting, ActionExecuted, ResultExecuting, ResultExecuted` and `Exception`), so that these events can be hooked into using Sitecore pipelines

   For more information, see the section *MVC Specific Pipelines*.

2. `InitializeControllerFactory` — This processor installs a custom controller factory (`IControllerFactory`) into the MVC runtime to facilitate greater control over MVC request handling and controller creation.

3. `InitializeRoutes` — Set up the default Sitecore route handler (fall-through: `{*pathInfo}`) and decorate existing routes with Sitecore specific route keys (

   For more information, see the section *MVC Routes*.

2.2.2 HttpBeginRequest Pipeline

Sitecore MVC installs some processors in the `HttpRequest` pipeline:

1. `TransferRoutedRequest` — This processor is placed before the `LayoutResolver` processor. If the request matches a route in the MVC route table, the pipeline is aborted causing the MVC request handler to take over. Only custom routes are considered. The Sitecore fall-through route (`{*pathInfo}`) is ignored by this processor — otherwise all requests would match and be processed by MVC.

2. `TransferMvcLayout` — This processor is placed after the `LayoutResolver` processor. If the resolved layout file name has an extension matching one of the extensions specified in the `MvcSettings.ViewExtensions` setting, the pipeline is aborted causing the MVC request handler to take over.

3. `TransferControllerRequest` — This processor is placed after the `TransferMvcLayout` processor. If no layout has been resolved, the `Controller` field of the current item is inspected. If it contains a value, the pipeline is aborted causing the MVC request handler to take over. The specified controller is then responsible for rendering the item.

2.3 MVC Specific Pipelines

Note that in the configuration file, the names of all MVC specific pipelines start with “mvc.” to avoid name clashes with the standard Sitecore pipelines.

2.3.1 RequestBegin

This is run as the first action of an MVC request — after it has been passed down from the main HttpRequest pipeline.

A `PageContext` is created and assigned to the current thread.

If the request is a form post, the associated form handler (if any) is executed

For more information, see the section on forms.

2.3.2 RequestEnd

This is run as the last action in an MVC request.

We do not currently do anything in this pipeline.

2.3.3 CreateController

This processor is called by the `SitecoreControllerFactory` to create a controller for the current route selected by the MVC request handler.

If the requested item (`PageContext.Item`) specifies a controller (in the `__Controller Name` field), this controller is instantiated and returned. Otherwise an instance of the default `SitecoreController` is returned.

2.3.4 ActionExecuting

This processor executes before an MVC controller action is executed.

We do not currently do anything in this pipeline.

2.3.5 ActionExecuted

This processor executes after an MVC controller action is executed.

We do not currently do anything in this pipeline.

2.3.6 ResultExecuting

This processor executes before an MVC result is executed.

We do not currently do anything in this pipeline.

2.3.7 ResultExecuted

This processor executes after an MVC result is executed.

We do not currently do anything in this pipeline.

2.3.8 Exception

This processor executes after the MVC runtime catches an unhandled exception.

We do not currently do anything in this pipeline.

If a custom processor is added to this pipeline, it should set the `ExceptionContext.ExceptionHandled` property on the `ExceptionArgs` object that is handed to the processor. Otherwise, Sitecore MVC will perform standard error handling.

2.3.9 GetPageItem

This pipeline resolves the item that was requested using route information. If the item cannot be resolved from the route, `Context.Item` is used.

For more information, see the section *MVC Routes.*

2.3.10 BuildPageDefinition

This pipeline builds the initial `PageDefinition` by typically using the XML based layout definition of the `Renderings` field on the page item.

2.3.11 GetPageRendering

The pipeline selects the `Rendering` to use as the root rendering of the page for creating the output of the current request.

Currently, the root rendering is selected based solely on the current device — `PageContext.Device`.

2.3.12 GetRenderer

This pipeline transforms an abstract rendering definition — `Rendering` — into an object that can render output back to the client — `Renderer`.

2.3.13 GetModel

This pipeline creates the model object to use when rendering an MVC view. This is typicallyAPS.NET MVC 3 Razor. If no model is returned by this pipeline, the view receives an instance of `EmptyModel` to trigger a meaningful error message in case the `.cshtml` file contains the `@model` directive and therefore expects a model.

2.3.14 RenderPlaceholder

This pipeline is called as part of the `Html.Sitecore().Placeholder` extension method. It handles nested placeholders, if applicable.

By default, the pipeline finds all renderings matching the specified placeholder name in the current `PageDefinition` and renders them. Any child renderings are only rendered if the rendering explicitly renders them — typically by calling `Html.Sitecore().Inner()`.

2.3.15 RenderRendering

The pipeline renders the specified rendering. This pipeline handles caching and also sets up the `IRenderingContext` for accessing the renderings source item and supporting nested renderings.

2.4 Sitecore MVC Request Handling

2.4.1 MVC Request Transferring

Sitecore MVC may transfer request processing to the ASP.NET MVC rendering engine in two scenarios. Both are detected and handled in the `httpRequestBegin` pipeline by custom processors.

1. If the request matches an MVC route the request is transferred to MVC after the Sitecore context has been set up, but before the Sitecore layout has been resolved.
2. After the Sitecore layout has been resolved, its file extension is examined and if it matches one of the extensions specified in the `MvcSettings.ViewExtensions` setting, the request is transferred to MVC.
3. If no layout is associated with the current item, but the item has a controller specified (in the Controller field), the request is transferred to MVC.

If none of these conditions are met, the request is handled as a WebForms request.

2.4.2 Placeholder Processing

The `Html.Sitecore().Placeholder` extension method renders the contents of the specified placeholder. This processing is done in the pipeline `RenderPlaceholder`, making it possible to modify output on a placeholder level.

2.4.3 Rendering Processing

When a rendering is being rendered, the `RenderRendering` pipeline is executed. This allows developers to modify the output of a rendering. For example, the output can be validated for HTML standards compliance.

The Page Editor can use this pipeline to augment the output with Page Editor specific tags.

2.4.4 Item Renderings

Sitecore MVC introduces the concept of item renderings. An item rendering is a new type of rendering that renders the item based on the rendering's data source property. The item is not rendered using the `Renderings` field, but rather the `Renderer` field.

An example of using item renderings might be frequently changing information on a *Home* page. On the `Home` item, the user adds an item rendering to the layout definition. The data source for the item renderings is set to a specific `Spot` item. The `Renderer` field of the `Spot` item contains a reference to the `Spot1` rendering. When the `Home` item is rendered, the `Renderer` field of the `Spot` item is parsed and the rendering(s) it specifies are rendered; in this case the `Spot1` rendering.

The elements of an item rendering do not have to point to Sitecore renderings. By default, if an element does not point to a rendering in Sitecore, it is assumed that the element should be rendered by an MVC view. The view is resolved using a combination of the element name and the `path` and `folder` properties.

The following is an example of an item rendering:

```
<modes>
  <mode name="mobile" screen-size="small">
    <view path="spots/spot" color="red">
      <text>
        <header>Header: {{:title}}</header>
        <text>
          <![CDATA[<div>This is embedded HTML. Value of 'text' field is:
              {{:text}}</div>]]>
        </text>
        <field name="title" cachekey="{{:  style}}" cache timeout="00:03:02"
            cac="1"/>
        <field datasource="/sitecore/content/home" vbd="1">{{:title}}</field>
      </text>
    </view>
    <copyright id="{493B3A83-0FA7-4484-8FC9-4680991CF743}"/>
    <spot folder="spots" color="{{:  style}}"/>
  </mode>
</modes>
```

2.4.5 Inner Renderings

A `Rendering` consists of a recursive data structure of renderings. Each rendering may contain any number of nested renderings that are specified by the `ChildRenderings` property.

The child renderings of a rendering are also called inner renderings and can be rendered using the `Html.Sitecore().ChildRenderings()` method in an ASP.NET MVC 3 Razor view. In the future, there will also be an XSLT function for rendering inner renderings.

Inner renderings are useful for reuse of rendering elements. The outer rendering encapsulates the inner renderings and as such may adorn them. This idea is similar to placeholders in Xml Controls.

Since inner renderings are just renderings it will become possible to render a Razor view inside an XSLT rendering.

There is no UI support for inner renderings and they are not supported in WebForms.

Chapter 3

View Renderings

This chapter discusses MVC View Renderings.

This chapter contains the following sections:

- Renderings

3.1 Renderings

MVC uses a different set of controls to render than ASP.NET WebForms. As such, not all Sitecore renderings are supported.

The supported rendering types are:

- Content rendering — static text
- Controller rendering — executes MVC controller and outputs result
- Item rendering — executes the renderings specified by the item
- Method Rendering
- Url Rendering
- View Rendering — MVC View
- Xslt Rendering

The following renderings are not supported:

- WebControl
- Sublayout

Usually `Sublayouts` can be converted to View renderings easily.

MVC is reusing the Sitecore Data templates for XSLT, URL, and Method renderings. However, developers should be aware that the renderer classes in MVC do not inherit from `WebControl` but inherit from `Renderer`.

Chapter 4

MVC Routing

This chapter discusses MVC Routing and shows how to map URLs by registering routes in the `Global.asax` module.

This chapter contains the following sections:

- MVC Routes

4.1 MVC Routes

Sitecore uses the MVC route table to determine which requests to send to the MVC runtime for processing.

The route table is typically set up in `global.asax` and Sitecore provides a set of custom route values that can be used to augment the routes with Sitecore specific information.

The default global.asax in an ASP.NET MVC 3 project contains this code:

```
public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterGlobalFilters(GlobalFilterCollection filters)
        {
            filters.Add(new HandleErrorAttribute());
        }

        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default", // Route name
                "{controller}/{action}/{id}", // URL with parameters
                new { controller = "Home", action = "Index", id =
               UrlParameter.Optional } // Parameter defaults
                );
        }

        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            RegisterGlobalFilters(GlobalFilters.Filters);
            RegisterRoutes(RouteTable.Routes);
        }
    }
```

You can add additional routes.

Examples of the Sitecore route extensions include:

```
// bypasses Sitecore and is handled by the 'MyController' custom controller
routes.MapRoute("example1", "special/{id}", new { controller = " MyController", });

// matches the item '/home/blogPosts/{blogNo}'
routes.MapRoute("example2", "blog/{blogNo}", new { scItemPath =
              "/home/blogPosts/{blogNo}", });

// sets language to {scLanguage} and matches the item '/home/{itemName}'
routes.MapRoute("example3", "home/{scLanguage}/{itemName}");

// matches the item '/home/{itemName}'
routes.MapRoute("example4", "home/{something}/{itemName}",
                new { scKeysToIgnore = "something", });

// matches the item '/home/{itemName}'
routes.MapRoute("example5", "home/{something}/{orOther}/{itemName}",
                new { scKeysToIgnore = new[] { "something", "orOther" }, });
```

A brief explanation of the custom route keys:

- `scLanguage` — The part of the URL that matches this key is used as the context language for the request.
- `scItemPath` — No matter what the route URL is, the item is resolved by using the value of this element. It can either be hard coded as in the previous example or be a part of the route URL, for example, `routes.MapRoute("myRoute", "/special/{*scItemPath}")`
- `scKeysToIgnore` — One or more keys of the route URL that should be ignored by Sitecore. If they occur in the route, they are removed before attempting to resolve an item from the URL.

If a controller is not specified in a custom route, the default Sitecore controller handles the request. Likewise, if an action is not specified, the default action `'Index'` is used.

Note that a default route — `{*pathInfo}` — should not be set up in `global.asax`. Sitecore installs a default route in the `InitializeRoutes` processor, as part of the `initialize` pipeline. This will cause all routes not matching any of custom routes set up in `global.asax` to be handled by Sitecore.

It is possible to hook into the route handling by adding processors to the `GetPageItem` pipeline where the route parsing is done.

Chapter 5

Additional Considerations for Sitecore MVC

This chapter lists additional details that have been added to Sitecore for MVC support and some considerations when implementing ASP.NET MVC with Sitecore.

This chapter contains the following section:

- Sitecore Data Templates
- Debugging
- Conditional Renderings

5.1 Additional Considerations for Sitecore MVC

The following sections describe some of the changes that have been implemented in Sitecore 6.6 to support Sitecore MVC features.

5.1.1 Sitecore Data Templates

The following have been added to the standard template:

- `__Controller Name` — The name of the MVC controller to execute when the item is requested. You can also specify a GUID in this field. The GUID should point to an item which is based on the template `Controller`.
- `__Controller Action` — The controller action to execute.
- The field type of the `__Renderers` field has been changed from 'text' to 'memo'.

The following templates have been added to the system:

- `Controller` — Specifies the name and action of a controller.
- `Model` — Specifies the type to use as a model for an MVC view.
- `Controller rendering` — Specifies the name and action of a controller to execute. If the controller returns any output, it is rendered to the output stream.
- `Item rendering` — Indicates that the item pointed to by the data source of the rendering should be rendered using the rendering definition stored in the item — using the `__Renderers` field.
- `View rendering` — Specifies an MVC view to be rendered.

5.1.2 Debugging

Sitecore MVC uses standard Sitecore tracing and profiling.

Additionally Sitecore MVC supports Glimpse through the `Sitecore.Glimpse.dll` assembly. This ships separately from MVC. This plug-in adds two tabs to the Glimpse view: Sitecore Trace and Sitecore Profile. The information displayed here is the same as the standard Sitecore trace and profile.

5.1.3 Conditional Renderings and Personalization

MVC does not support global conditional rendering rules — rules that apply to all renderings.

This means that when you use MVC, you cannot set up a system of global conditions that determine which content is displayed to website visitors who meet these conditions every time this rendering is called.

However, MVC does allow you to implement the personalization of components when you specify the presentation details for individual elements on your website. In the Page Editor, you can personalize content by specifying that visitors who meet some predefined conditions are shown particular content items. You can also configure these personalization rules in the Content Editor and in the Page Editor in the **Layout Details** dialog box.